Project — Introduction to Database Systems

California Highway Patrol

Clément Chaffard, Matthias Zeller, Nicolas Delamaide

Milestone 3 for the course
CS-322: Introduction to Database Systems

EPFL

EPFL, Swizerland
May 31, 2021

Contents

I Milestone 1 2
- I-A Entity Relationship Schema 2
  - I-A1 First normal form 2
  - I-A2 Design choices 2
- I-B Relational Schema 2
- I-C Data types and additional constraints 2
- I-D DDL Commands 2

II Milestone 2 7
- II-A Data cleaning and data loading 7
  - II-A1 collisions2028.csv 7
  - II-A2 parties2028.csv 7
  - II-A3 victims2028.csv 7
- II-B Query Implementation 7
  - II-B1 Query 1 7
  - II-B2 Query 2 7
  - II-B3 Query 3 7
  - II-B4 Query 4 8
  - II-B5 Query 5 8
  - II-B6 Query 6 8
  - II-B7 Query 7 8
  - II-B8 Query 8 8
  - II-B9 Query 9 8
  - II-B10 Query 10 8

III Milestone 3 9
- III-A Changes from the previous milestone 9
- III-B Queries 9
  - III-B1 Query 1 9
  - III-B2 Query 2 9
  - III-B3 Query 3 9
  - III-B4 Query 4 10
  - III-B5 Query 5 10
  - III-B6 Query 6 10
  - III-B7 Query 7 10
  - III-B8 Query 8 11
  - III-B9 Query 9 11
  - III-B10 Query 10 12
- III-C Query optimization 13
  - III-C1 Query 1 13
  - III-C2 Query 2 13
  - III-C3 Query 4 14
  - III-C4 Query 7 14
  - III-C5 Query 10 15

General Comment

The modifications performed on milestone 2 for milestone 3 are stated under the corresponding section (III-A) in milestone 3. The modifications performed on milestone 2 for milestone 1 are not stated in milestone 2, since milestone 1 was completely re-written.

All SQL queries are available in a more readable format in the submission zip file:

- `queries_milestone2.sql`
- `queries_milestone3.sql`

The notebooks used for data cleaning are included in the submission zip file (in the folder `notebooks`).

I. Milestone 1

A. Entity Relationship Schema

1) First normal form: We refer to *duplicated attributes* as the set of more than one attribute that describe the same thing, i.e. they are not in first normal form. For instance, the table `Party` has attributes `weather_1` and `weather_2`. We describe the way we handle weather, but we handle all such duplicated attributes in the same way (`road_cond`, `safety_equipment`, `other_associated_factor`).

We create an entity `Weather`, with an attribute `weather`, connected to `Collision` with a many-to-many relationship `weather_rel`. Thus, a collision can be associated to any number of weathers $(0, 1, 2, \ldots)$.

2) Design choices: We decided to make the entity `Collision` the central entity of our ER model as the data in our dataset was collected following a traffic collision in the state of California in 2018. Each collision happens at a `Location`, under given `Conditions` and `Weather` as well as road conditions (`RoadCondition`). A collision happens because someone committed a `Violation`, and there are parties (`Party`) involved in a collision. Following a collision there can be victims (`Victim`) that are associated to a party involved in the collision. Moreover, there can be an associated factor (`OtherAssociatedFactor`) that causes a party to be involved in a collision. Both parties and victims can use a safety equipment (`PartyEquipment`, `VictimEquipment`). Naturally, a driver drives exactly one `Vehicle`.

B. Relational Schema

Figure 1 displays the entity-relational diagram. for completeness and to ease understanding, we attach figure 2, which displays the SQL tables and focuses on the relationship between identifiers.

The ER model is centered around `Collision`, with the primary key `case_id`. Each collision has exactly one `Violation`, `Location` and `Conditions`. This enables two possibilities: either migrate those entities into `Party`, or store in separate tables with their corresponding relationship. We chose the latter option, as this avoids data redundancy. For instance, the table `Conditions` has only 30 combinations of `Lighting` and `Road_surface`. Having the relationship table `ConditionsRel` with only two attributes `case_id` and `cid` enables a structured way of handling conditions, although this design choice is not optimally performant (joining, key constraints). The primary key of those three entities is a customly created identifier: a simple integer that enumerates all rows.

As explained in section I-A2, `Weather` and `RoadCondition` are entities created to ensure first normal form. They should normally have two associated tables, for the entity as well as for the relationship. However, those entities have a single attribute. Thus, it is sufficient to store a single table for both, with a composite primary key: `(case_id, weather)` and `(case_id, road_cond)`.

A `Party` is a weak entity, since it depends on its owner entity `Collision`. As a weak entity, its primary key sof invertible matrixhould be composed the owner's primary key (`case_id`). However, there may be several parties per collision, and the party id (`pid`) is the only unique identifier. In particular, `(case_id, party_number)` is not a unique identifier. We could have used the composite primary key `(case_id, pid)`, but this is not a minimal key. Thus, we use `pid` as the primary key, with the foreign key `case_id` stored as attribute. A party is further described by its `party_number`. Note that each party is involved in exactly one collision, as data anonymization doesn't allow to identify the same person across different collisions.

`Victim` is another weak entity related to `Party`. We chose the victim id `vid` as the primary key, for the same reasons as described above. The owner key (party id, `pid`) is stored as an attribute with foreign key constraint.

Both `Party` and `Victim` use safety equipment(s) (either 0, 1 or 2). Since this entity has a single attribute, it is sufficient to store it in the relationship table. Thus, both parties and victims have their table `PartyEquipment` and `VictimEquipment`.

Remains the `Vehicle` entity, which is driven by exactly one party. Thus, the vehicle relationship is merged with the entity, and the party id `pid` is used as primary key.

C. Data types and additional constraints

All our primary key ids are of type `INTEGER`. `Collision.datetime` is of type `TIMESTAMP`, `Collision.process_date` is of type `DATE`. Note that the `datetime` attribute is a combination of `collision_date` and `collision_time`, the latter was set to `00:00:00` when it was missing.

All other attributes are either `CHAR` or `VARCHAR2`. `CHAR` is only used to represent binary attributes: either truth value encoded as `T/F` (e.g. `tow_away`, `at_fault`, `hazardous_materials`) or gender. All other attributes have variable-length strings, we thus use `VARCHAR2`, with a maximum length determined during data exploration (we set max length exactly to longest string encountered). The only exception is `Collision.officer_id`, whose longest string is 8, but was set to 10, to allow longer strings to be added later in the database.

Note that `VARCHAR2` is used instead of `VARCHAR`, although they are currently synonyms, but this is an official recommendation by Oracle, to prevent unexpected behaviour in future releases [1].

The `ON DELETE CASCADE` constraint is added whenever there is a reference to a foreign key (`case_id`, `pid` and `vid`).

D. DDL Commands

[1]https://docs.oracle.com/cd/A57673_01/DOC/server/doc/SCN73/ch6.htm#varchar2

```
-- Collision entity
CREATE TABLE Collision(
   case_id INTEGER,
   datetime TIMESTAMP NOT NULL,
   officer_id VARCHAR2(10), -- allow longer
       ids if added later
   type_collision CHAR(10),
   process_date DATE,
   primary_collision_factor VARCHAR2(22),
   collision_severity VARCHAR2(20),
   tow_away CHAR(1), -- T/F
   hit_and_run VARCHAR2(15),
   -- Key constraints
   PRIMARY KEY (case_id)
);

-- Violation entity
CREATE TABLE Violation(
   vid INTEGER, -- custom id
   pcf_violation INTEGER,
   pcf_violation_category VARCHAR2(33),
   pcf_violation_subsection CHAR(1),
   -- Key constraints
   PRIMARY KEY (vid)
);

-- Relationship table: Collision <->
    Violation
CREATE TABLE Violated(
   case_id INTEGER,
   vid INTEGER NOT NULL,
   -- Key constraints: case_id sufficient PK
   PRIMARY KEY (case_id),
   FOREIGN KEY (case_id) REFERENCES Collision
                ON DELETE CASCADE,
   FOREIGN KEY (vid) REFERENCES Violation
                ON DELETE CASCADE
);

-- Location entity
CREATE TABLE Location(
   lid INTEGER, -- custom id
   jurisdiction INTEGER,
   location_type VARCHAR2(12),I found it
   ramp_intersection VARCHAR2(50),
   county_city_loc INTEGER,
   population VARCHAR2(30),
   -- Key constraints
   PRIMARY KEY (lid)
);

-- Relationship table: Collision <-> Location
CREATE TABLE Located(
   case_id INTEGER,
   lid INTEGER NOT NULL,
   -- Key constraints: case_id sufficient PK
   PRIMARY KEY (case_id),
   FOREIGN KEY (case_id) REFERENCES Collision
               ON DELETE CASCADE,
   FOREIGN KEY (lid) REFERENCES Location
               ON DELETE CASCADE
);

-- Conditions entity
CREATE TABLE Conditions(
   cid INTEGER,
   lighting VARCHAR2(39),
   road_surface VARCHAR2(8),
   -- Key constraints
   PRIMARY KEY (cid)
);

-- Conditions relationship: Collision <->
    Conditions
CREATE TABLE ConditionsRel(
   case_id INTEGER,
   cid INTEGER NOT NULL,
   -- Key constraints
   PRIMARY KEY (case_id),
   FOREIGN KEY (case_id) REFERENCES Collision
                     ON DELETE CASCADE
);

-- Weather entity: ensure 1st normal form
CREATE TABLE Weather(
   case_id INTEGER,
   weather VARCHAR2(7),
   -- Key constraints: composite PK
   PRIMARY KEY (case_id, weather),
   FOREIGN KEY (case_id) REFERENCES Collision
                     ON DELETE CASCADE
);

-- Road conditions entity: ensure 1st normal
    form
CREATE TABLE RoadCondition(
   case_id INTEGER,
   road_cond VARCHAR2(14),
   -- Key constraints: composite PK
   PRIMARY KEY (case_id, road_cond),
   FOREIGN KEY (case_id) REFERENCES Collision
                     ON DELETE CASCADE
);

-- Party entity (weak entity)
CREATE TABLE Party(
   pid INTEGER,
   case_id INTEGER,
   party_number INTEGER,
   party_type VARCHAR2(14),
   party_age INTEGER,
   party_sex CHAR(1), -- M/F
   party_sobriety VARCHAR2(37),
   at_fault CHAR(1), -- T/F
   party_drug_physical VARCHAR2(21),
   cellphone_use VARCHAR2(21),
   movement_preceding_collision VARCHAR2(26),
   hazardous_materials CHAR(1), -- T/F
   financial_responsibility VARCHAR2(111),
   -- Key constraints: PK is pid since
       (case_id, party_num)
   --           is not sufficient
   PRIMARY KEY (pid),
   FOREIGN KEY (case_id) REFERENCES Collision
              ON DELETE CASCADE
);

CREATE TABLE PartyEquipment(
   pid INTEGER,
   equipment VARCHAR2(37),
   PRIMARY KEY (pid, equipment),
   FOREIGN KEY (pid) REFERENCES Party
                     ON DELETE CASCADE
);
```

```sql
-- Vehicle entity
CREATE TABLE Vehicle(
   pid INTEGER,
   statewide_vehicle_type VARCHAR2(35),
   vehicle_make VARCHAR2(28),
   vehicle_year INTEGER,
   -- Key constraints: the primary key of Party is sufficient
   PRIMARY KEY (pid),
   FOREIGN KEY (pid) REFERENCES Party
               ON DELETE CASCADE
);

-- Other associated factor entity
CREATE TABLE OtherAssociatedFactor(
   pid INTEGER,
   factor VARCHAR2(29),
   PRIMARY KEY (pid, factor),
   FOREIGN KEY (pid) REFERENCES Party
                          ON DELETE CASCADE
);

CREATE TABLE Victim(
   vid INTEGER,
   pid INTEGER,
   victim_role VARCHAR2(17),
   victim_age INTEGER,
   victim_ejected VARCHAR2(17),
   victim_sex CHAR(1), -- M/F
   victim_seating_position VARCHAR2(29),
   -- Key constraints
   PRIMARY KEY (vid),
   FOREIGN KEY (pid) REFERENCES Party
              ON DELETE CASCADE
);

CREATE TABLE VictimEquipment(
   vid INTEGER,
   equipment VARCHAR2(37),
   PRIMARY KEY (vid, equipment),
   FOREIGN KEY (vid) REFERENCES Victim
                     ON DELETE CASCADE
);
```

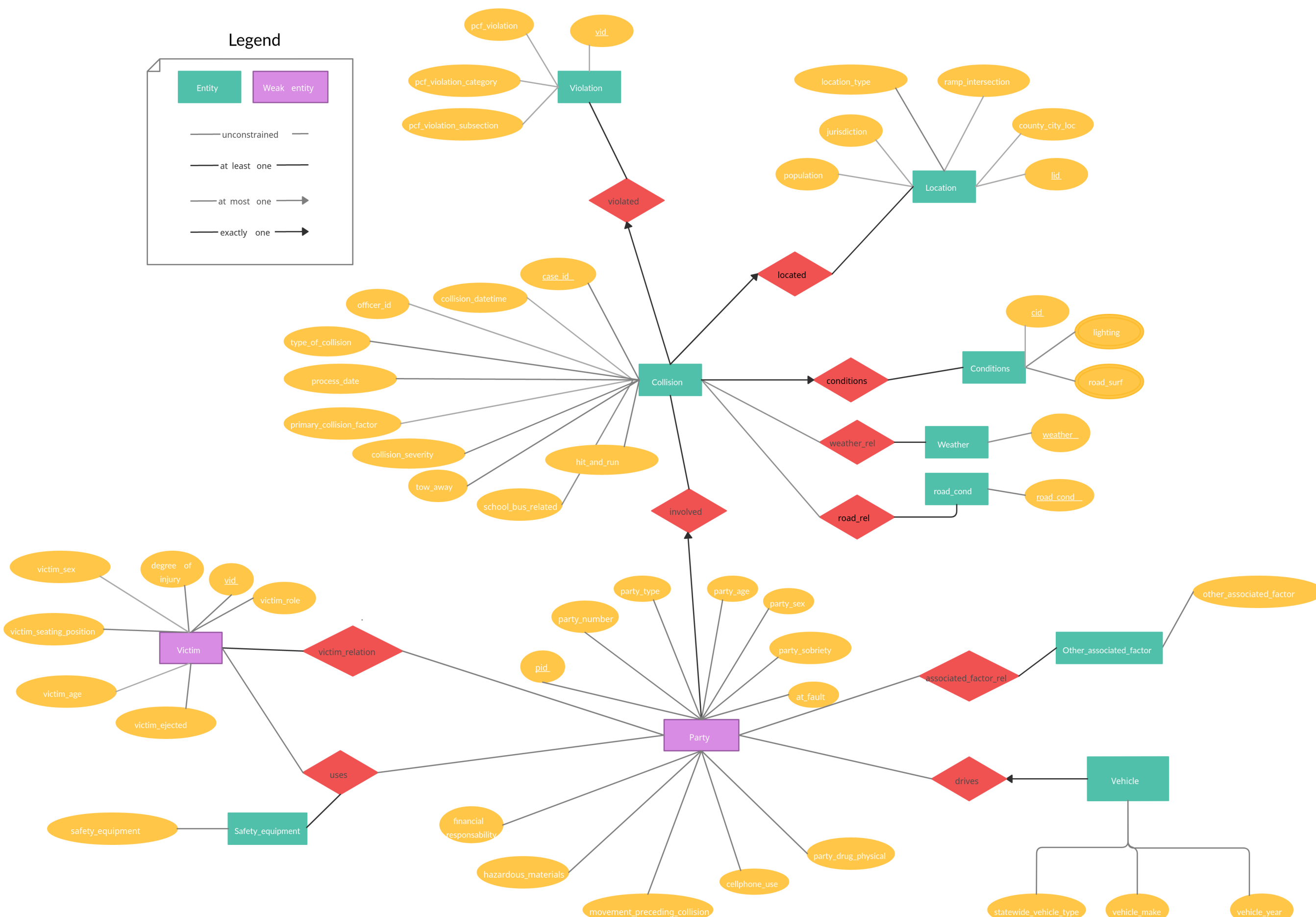

Fig. 1. Entity-relationship diagram. One can find the full size image on the following link https://drive.google.com/file/d/1CbjMK2y3QNdwD1jj7A3-cRbYN5FJuhhA/view?usp=sharing.

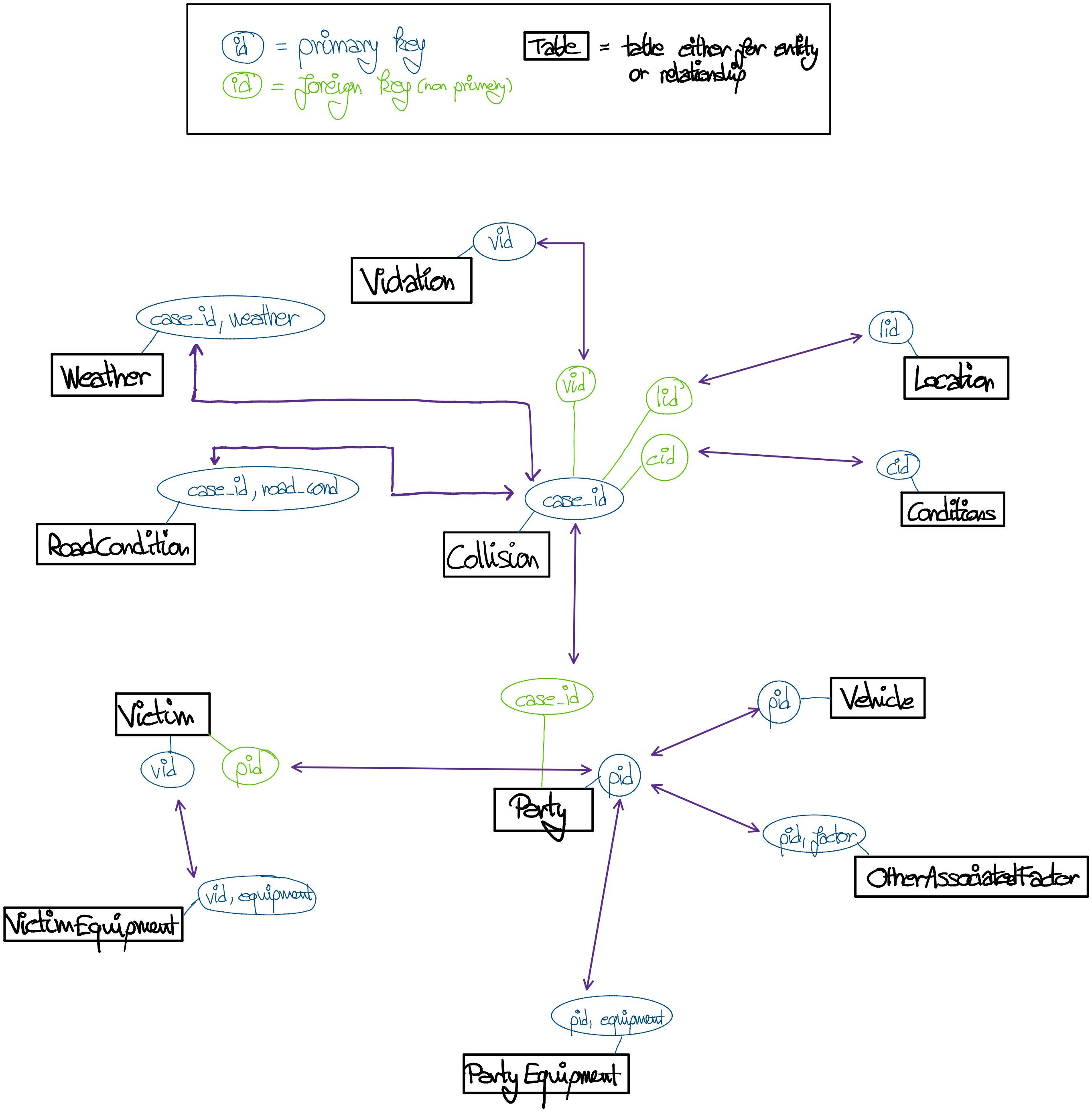

Fig. 2. Relationship between the primary keys and foreign keys of the SQL tables, as stored in our database.

II. Milestone 2

A. Data cleaning and data loading

1) collisions2028.csv: The first thing that we noticed is that the case_id were not unique because three of them were used two times : 97293, 373108, 965874. We decided that it was better to remove the data referring to those case_id in the files collisions2018.csv as well as in parties2018.csv and victims2018.csv. Our reasoning was that they represent a very small subset of the data and that we couldn't know if a party linked to the case_id 97293 for example belonged to the first case with that id or the second. There were also entries with a case_id in collisions2018.csv but not in parties2018.csv, which we removed from the dataframe. There were also fields with inconsistent values, such as :

- hit_and_run : an entry has value D
- pcf_violation_category : an entry has value 21804
- road_surface : and entry has value H
  The values in those entries were set to NaN. We also noticed that the officer_id could be composed of numbers but also characters or a mix of the two. We decided to set the type of that field to the type string. Similarly for the field pcf_violation_subsection some entries were a single number while others were a single letter. Since we don't have any more information on the field from the documentation, we decided to keep those values and set the type of the field to type string.
  We also added a new field to the dataframe which we called datetime. It contains the values of collision_date and collision_time merged together and converted from the type string to the type datetime.
  Lastly, we changed the boolean values of tow_away from 1.0 and 0.0 to a one character string T and F respectively.

2) parties2028.csv: There were a few inconsistencies in the data. First, for the field `cellphone_use` some entries used the characters B, C or D while others used 1, 2 or 3 instead. As the provided documentations mentions that this field should be characters, we mapped each integer 1, 2 and 3 to their respective character, that is 1 to B, 2 to C and 3 to D. Secondly, for the make of the vehicle (the field `vehicle_make`) there were to spelling mistakes, for example TOYTA instead of TOYOTA, or inconsistencies such as WHITE VOLVO instead of VOLVO. Such mistakes were changed and for the rows with the value WHITE, which refers to the color of the vehicle and not its make, the `vehicle_make` was set to NaN. Lastly, the values of the fields `party_sex`, `at_fault`, `school_bus_related` and `hazardous_materials` were changed to one character strings. For `party_sex`, this means that the values $female$ and $male$ were changed to F and M respectively. For `at_fault`, the values 1 and 0 (for True / False) where changed to T, F respectively. For `hazardous_materials` and `school_bus_related`, the values representing True (A and E respectively) were changed to T and the others to F.

3) victims2028.csv: We noticed that victim_degree_of_injury has four entries with value 7 instead of the string $possibleinjury$, and that victim_ejected has four entries with value 4 which isn't a value present in the documentation. We changed the values 7 for the first field to the corresponding string and for the second field we changed 4 to NaN. Similarly to party_sex in parties2018.csv, values $female$ and $male$ in the field victim_sex were changed to F and M.

B. Query Implementation

1) Query 1: We list the year and the number of collision per year and counting every different `case_id` which uniquely identify a collision and we group those results by year. To do so we use tht inbuilt sql function TO_CHAR(datetime, 'yyyy') to extract the year of the collision from datetime. We then group by on the different years to count the number of collisions per year.

```sql
SELECT TO_CHAR(datetime, 'yyyy') AS year,
    COUNT(case_id) AS collisions_per_year
FROM Collision
GROUP BY TO_CHAR(datetime, 'yyyy')
ORDER BY TO_CHAR(datetime, 'yyyy');
```

YEAR	COLLISIONS_PER_YEAR
2001	522562
2002	544737
2003	538949
2004	538294
2005	532724
2006	498849
2007	501908
2017	7
2018	21

2) Query 2: We count every vehicle per make and group the results per vehicle make and rank them in descending order to find the most represented make taking only the first result using FETCH FIRST 1 ROWS ONLY.

```sql
SELECT vehicle_make, COUNT(vehicle_make) AS
    vehicle_per_make
FROM Vehicle
GROUP BY vehicle_make
ORDER BY vehicle_per_make DESC
FETCH FIRST 1 ROWS ONLY;
```

VEHICLE_MAKE	VEHICLE_PER_MAKE
FORD	1129606

3) Query 3: We find the fraction of total collisions that happened under dark lighting conditions. To do so we need to use the table Conditionsrel that allows to identify each collision (case_id) to a condition (cid).

```sql
SELECT lighting, COUNT(case_id)/(SELECT
    COUNT(case_id) FROM Collision) AS RATIO
FROM Conditions Cd, Collision C
WHERE Cd.lighting LIKE '%dark%' AND C.cid =
    Cd.cid
GROUP BY lighting;
```

LIGHTING	RATIO
dark with no street lights	0.078685044013649862
dark with street lights not functioning	0.001158940121796895
dark with street lights	0.201584569064565003

TABLE I

4) Query 4: To find the number of collisions that have occurred under snowy weather conditions we use the relationship table Conditionsrel similarly.

```sql
SELECT COUNT(case_id) AS snowy_collision
FROM Conditions Cd, Collision C
WHERE Cd.road_surface LIKE '%snow%' AND
    C.cid = Cd.cid;
```

SNOWY_COLLISION
19738

5) Query 5: To compute the number of collisions per day of the week we perform a group by on the days of the week to be able to count the collisions independently on each day. To do that we use the inbuilt sql function TO_CHAR(datetime, 'DAY') which allows to extract the day of the week from datetime. Then we order the table according to descending number of collisions per day to find the day with the most collisions.

```sql
SELECT TO_CHAR(datetime, 'DAY') day,
    COUNT(case_id) AS col_per_day
FROM Collision
GROUP BY TO_CHAR(datetime, 'DAY')
ORDER BY col_per_day DESC;
```

DAY	COL_PER_DAY
FRIDAY	614850
THURSDAY	536813
WEDNESDAY	536066
TUESDAY	535741
MONDAY	516797
SATURDAY	509497
SUNDAY	428287

6) Query 6: We list all distinct weather types and their corresponding number of collisions in descending order of the number of collisions using group by on the distinct weathers.

```sql
SELECT DISTINCT weather, COUNT(case_id)
FROM Weather
GROUP BY weather
ORDER BY COUNT(case_id) DESC;
```

WEATHER	COUNT(CASE_ID)
clear	2940204
cloudy	548158
raining	223747
fog	21252
wind	13952
snowing	8530
other	6960

7) Query 7: To find the number of at-fault collision parties with financial responsibility and loose material road, we count only the party ids where those conditions are verified. conditions.

```sql
SELECT COUNT(pid)
FROM Party P, Roadcondition R
WHERE P.case_id = R.case_id AND R.road_cond
    LIKE '%loose material%'
   AND P.financial_responsibility = 'Y';
```

COUNT(pid)
3176

8) Query 8: To find the most common victim seating position, we count victims id grouping by victim seating position and show the row with the most victims.

```sql
SELECT victim_seating_position,
MEDIAN(victim_age),
COUNT(victim_seating_position) AS
    victim_per_seating_position
FROM Victim
GROUP BY victim_seating_position
ORDER BY victim_per_seating_position DESC
FETCH FIRST 1 ROWS ONLY;
```

VIC_SEATING_POS	VIC_PER_SEAT_POS	VICTIM_MEDIAN
3.0	1331633	25

9) Query 9: We compute the fraction of all participants that have been victims of collisions while using a belt ('C') using the table Victimequipment which stores the victim id and the equipments the victim was using during collision.

```sql
SELECT equipment, COUNT(vid)/(SELECT
    COUNT(vid) FROM Victimequipment) AS
    fraction
FROM Victimequipment V
WHERE V.equipment = 'C'
GROUP BY equipment;
```

EQUIPMENT	FRACTION
C	0.02178

10) Query 10: We compute the fraction of the collisions happening for each hour of the day and we display the ratio as percentage for all the hours of the day.

```sql
SELECT TO_CHAR(datetime,'HH24')AS hour,
    100*COUNT(case_id)/(SELECT
    COUNT(case_id) FROM Collision) AS
    percentage
FROM Collision C
GROUP BY TO_CHAR(datetime,'HH24')
ORDER BY TO_CHAR(datetime,'HH24') ASC;
```

HOUR	PERCENTAGE
00	2.71488242222368931306306931061093176234
01	1.82895064601619187477011210484940713195
02	1.80764700758775684337775844669846308673
03	1.15605525409304475040069148059732295373
04	0.9827412789617139220941476575151635858684
05	1.45014532521058147282482362085293332793
06	2.63088838946157827783384160633872008406
07	5.1871863053110969210139866153234186348
08	5.25057017745819854436233050708762064434
09	4.09151909921115511767208929331282118425
10	4.22785683733063191362706094671371959863
11	4.88838058506226598591185952469585450059
12	5.77325827499335648254738256508011887874
13	5.77115010244054259923251423432559837427
14	6.54215873540714637208468862014003813573
15	7.74634134975192911658138738835700681661
16	7.31954736425726584444238558586947416074
17	7.90384403073715582002641873078026240091
18	6.29969115272101276609437178954446274609
19	4.42757844759721033293037649187882002254
20	3.49030157408368002600819191471443415817
21	3.27254954066803549940666038283303994599
22	2.85607450503714544559832988351237081202
23	2.380681594377614758095521298367997053

III. Milestone 3

A. Changes from the previous milestone

Based on the feedback we received from the previous milestone we made a few changes. In particular, we removed the tables for the relationships between Collision and the following entities : Location, Conditions and Violation. Instead we added the primary keys of the mentioned entities to the table Collision as a foreign key reference, each having its own column lid, cid and vid respectively.
We had also previously mentioned that we found entries where the case_id was duplicated in the file collisions.csv, for example 97293. After the feedback regarding duplicates, we realized that this was due to how we imported the table with Pandas. We didn't specify the data type for the case_id column and Pandas inferred that it was of type int, which was wrong. We imported it using the type string for that column and noticed that we didn't have duplicate case_id anymore. This is due to that fact that some case_ids had leading zeros, for example one was 097293 and the other 0097293 which Pandas interpreted as the same int 97293.

B. Queries

1) Query 1: We solve this query using a select case query separating into different groups of age. Considering that every age is equally represented, we would deduce that young people in the 19-21 and 21-24 are the most likely to be involved in an accident. The query takes 6557ms.

```
SELECT CASE
        WHEN party_age <= 18 THEN '1-18'
        WHEN party_age <= 21 THEN '19-21'
        WHEN party_age <= 24 THEN '21-24'
        WHEN party_age <= 60 THEN '24-60'
        WHEN party_age <= 64 THEN '60-64'
        ELSE '64+'
      END AS age,
      COUNT(*)/(SELECT COUNT(*) FROM Party)
          AS ratio
FROM Party
WHERE PARTY_AGE IS NOT NULL
GROUP BY CASE
        WHEN party_age <= 18 THEN '1-18'
        WHEN party_age <= 21 THEN '19-21'
        WHEN party_age <= 24 THEN '21-24'
        WHEN party_age <= 60 THEN '24-60'
        WHEN party_age <= 64 THEN '60-64'
        ELSE '64+'
       END;
```

AGE	RATIO
1-18	0.068494
21-24	0.071767
24-60	0.548133
19-21	0.083880
60-64	0.021614
64+	0.055443

TABLE II
OUTPUT OF QUERY 1

2) Query 2: This query can be made with two independent queries. The inner query selects all parties involved in collisions with holes, by joining the tables `Party` and `RoadCondition`. The outer query selects the vehicle types of the aforementioned parties, groups by the vehicle type while counting occurrences in each group, eventually sorts them and selects the top 5. The query takes 2076 ms.

```
SELECT STATEWIDE_VEHICLE_TYPE AS TYPE,
    COUNT(*) AS COUNTS
FROM VEHICLE
WHERE PID IN (
    -- Select parties involved in collisions
        with holes
    SELECT P.PID FROM PARTY P, ROADCONDITION R
    WHERE P.CASE_ID = R.CASE_ID AND
        R.ROAD_COND LIKE 'hole%'
)
GROUP BY STATEWIDE_VEHICLE_TYPE
ORDER BY COUNTS DESC
FETCH FIRST 5 ROWS ONLY;
```

TYPE	COUNTS
passenger car	3399
pickup or panel truck	764
motorcycle or scooter	141
bicycle	133
truck or truck tractor with trailer	117

TABLE III
OUTPUT OF QUERY 2

3) Query 3: We include the SQL query, but we cannot run it: we forgot the attribute `degree_of_injury` during the data import process.

```
SELECT v.vehicle_make, SUM(t.count_vic) as
    sum_vic
FROM (SELECT pid, COUNT(*) as count_vic
```

```
    FROM Victim
    WHERE degree_of_injury IN ('severe
        injury', 'killed')
    GROUP BY pid) t, vehicle v
WHERE t.pid = v.pid and v.vehicle_make IS
    NOT NULL
GROUP BY v.vehicle_make
ORDER BY sum_vic DESC
FETCH FIRST 10 ROWS ONLY;
```

4) Query 4: To find the most common victim seatingposition, we count victims id grouping by victim seatingposition. The query takes 5267 ms.

```
SELECT victim_seating_position,
    COUNT(victim_seating_position)/(SELECT
    COUNT(victim_seating_position) FROM
    Victim) AS
    victim_per_seating_position_RATIO
FROM Victim
GROUP BY victim_seating_position
ORDER BY victim_per_seating_position_RATIO
    DESC;
```

VICTIM SEATING POSITION	VICTIM PER SEATING POSITION
Passenger	0.615014320204506722
Driver	0.342180990816133308
Other	0.018862431357697405
Station Wagon Rear	0.013253290096572618
Rear Occupant of Truck or Van	0.010688967525089945

TABLE IV
OUTPUT OF QUERY 4

5) Query 5: In the innermost query, we first join the tables `Vehicle`, `Party` and `Location`, and then group by by vehicle type and city, counting the number of occurrences in each group. In the intermediate query, we select vehicle types that had a collision in more than 10 cities, we group by vehicle type and count the number of unique cities a vehicle type is involved in. In the outermost query, we count the number of rows whose number of unique cities is at least half the total number of cities. The query outputs the value `14` in 1min 7s 665ms, i.e. in 67665 ms.

```
SELECT COUNT(*) FROM (
    -- Subquery: number of cities per vehicle
        type (at least 10 collisions per city)
    SELECT COUNT(DISTINCT CITY) as NCITY FROM
        (
        -- Subquery: vehicle type, city and
            number of collision in that city
        SELECT V.STATEWIDE_VEHICLE_TYPE as
            VTYPE,
              L.COUNTY_CITY_LOC AS CITY,
              COUNT(*) AS COUNTS
        FROM VEHICLE V, PARTY P, COLLISION C,
            LOCATION L
        -- Join vehicle and location
        WHERE V.PID = P.PID AND
             P.CASE_ID = C.CASE_ID AND
             C.LID = L.LID
        -- Aggregate
        GROUP BY V.STATEWIDE_VEHICLE_TYPE,
            L.COUNTY_CITY_LOC
    ) WHERE COUNTS >= 10
      GROUP BY VTYPE
) WHERE NCITY >= (SELECT COUNT(DISTINCT
    COUNTY_CITY_LOC) FROM LOCATION) / 2;
```

.

6) Query 6: First we select the top 3 cities in terms of population that we join with and Victims (where the victim age ¿ 0) on the party_id pid. From there for each of the top 3 cities we can calculate the mean age of the victims in each collision. Once we have that we partition by city and order by the mean victim age to get the bottom 10 collisions in terms of the mean victim age for each of the top 3 cities. The query takes 2331 ms.

```
SELECT city, population, case_id, mean_age
FROM
    (SELECT city, population, case_id,
        mean_age, row_number() OVER(PARTITION
        BY city ORDER BY mean_age ASC) as rn
       FROM (
          SELECT t2.city, t2.population,
              t.case_id, AVG(t.victim_age)
              OVER(PARTITION BY t.case_id) as
              mean_age
          FROM (
             SELECT p.case_id, v.victim_age
             FROM Party p, Victim v
             WHERE p.pid = v.pid and
                 v.victim_age IS NOT NULL and
                 v.victim_age > 0) t,
             Collision c,
             (SELECT l.county_city_loc as
                 city, l.lid, l.population
              FROM (SELECT
                  DISTINCT(county_city_loc)
                  as city, population
                  FROM Location
                  WHERE population IS NOT
                      NULL and population < 9
                  ORDER BY population DESC
                  FETCH FIRST 3 ROWS ONLY)
                      top, Location l
             WHERE l.county_city_loc =
                 top.city) t2
       WHERE c.case_id = t.case_id and c.lid
           = t2.lid))
WHERE rn <= 10;
```

7) Query 7: The subquery in the `FROM` clause selects all party ids whose victims are all over 100 years old, and computes the maximum age related to this party id. The outer query filters pedestrian collisions, joins collisions with parties and selects case ids whose parties are in the table computed in the inner query. The query takes 13,889 ms.

```
SELECT C.CASE_ID, X.MAXAGE
FROM COLLISION C, PARTY P, (
       SELECT V.PID, MAX(V.VICTIM_AGE) AS
           MAXAGE
       FROM VICTIM V
```

CITY	POPULATION	CASE_ID	AVG_AGE
109	7.0	0038669	1
109	7.0	9370011206083515536	1
109	7.0	9370010906075413725	1
109	7.0	9370010728101016001	1
109	7.0	9370010508101012906	1
109	7.0	9370010504172015128	1
109	7.0	9370010414165514162	1
109	7.0	9370010402205512484	1
109	7.0	9370010214182413534	1
109	7.0	9370010123174315201	1
1005	7.0	0059381	1
1005	7.0	9435011129173013660	1
1005	7.0	9435011129114811549	1
1005	7.0	9435011127082012588	1
1005	7.0	9435010509103214363	1
1005	7.0	3462959	1
1005	7.0	3049421	1
1005	7.0	3038724	1
1005	7.0	2933392	1
1005	7.0	2916256	1
3313	7.0	0000730	1
3313	7.0	9840011228103012075	1
3313	7.0	9840011101111508393	1
3313	7.0	9840011031193014962	1
3313	7.0	9840010927195011711	1
3313	7.0	9840010926182515360	1
3313	7.0	9840010622145015221	1
3313	7.0	9840010306075511368	1
3313	7.0	9840010130115511368	1
3313	7.0	3434304	1

TABLE V
OUTPUT OF QUERY 6

```
      WHERE V.VICTIM_AGE NOT IN (999, 998)
      GROUP BY V.PID
      -- if min age of group is > 100, then
          all victims are > 100
      HAVING MIN(V.VICTIM_AGE) > 100
   ) X
WHERE C.TYPE_COLLISION = 'pedestrian' AND
     C.CASE_ID = P.CASE_ID AND
     P.PID IN X.PID;
```

8) Query 8: We simply group by the tuple (STATEWIDE_VEHICLE_TYPE, VEHICLE_MAKE, VEHICLE_YEAR) which serves as a vehicle id and count the number of *case_id* for each of those tuples. This count corresponds to the number of collisions. We notice that the vehicles involved in the most collisions are passenger cars made by the most popular brands in the U.S : Toyota, Ford and Honda.
This query takes 22 542 ms.

```
SELECT v.STATEWIDE_VEHICLE_TYPE,
    v.VEHICLE_MAKE, v.VEHICLE_YEAR,
    COUNT(p.case_id) AS number_collisions
FROM Party p, Vehicle v
WHERE (p.pid = v.pid
      and (
         v.STATEWIDE_VEHICLE_TYPE IS NOT NULL
         AND v.VEHICLE_MAKE IS NOT NULL
         AND v.VEHICLE_YEAR IS NOT NULL
         ))
```

CASE_ID	MAX AGE
0439197	102
0868472	103
1209166	101
0784061	102
0828116	102
2472739	103
0958765	102
1373664	101
0820619	101
0851026	106
3388544	105
3485436	101
0644226	103
0815100	101
0069198	101
2531557	103
0817210	102
1347636	101
1213340	121
0036446	110
0445265	101
0566220	102
0819020	101

TABLE VI
OUTPUT QUERY 7

```
GROUP BY v.STATEWIDE_VEHICLE_TYPE,
    v.VEHICLE_MAKE, v.VEHICLE_YEAR
HAVING COUNT(p.case_id) >= 10
ORDER BY number_collisions DESC
```

TYPE	MAKE	YEAR	# COLLISIONS
passenger car	TOYOTA	2000	49938
passenger car	FORD	2000	49288
passenger car	HONDA	2000	47789
passenger car	FORD	1998	46713
passenger car	TOYOTA	2001	44849
passenger car	HONDA	2001	42926
passenger car	FORD	2001	42897
passenger car	TOYOTA	1999	40826
passenger car	HONDA	1998	39853
passenger car	FORD	1999	39757
passenger car	FORD	1995	38180
passenger car	HONDA	1997	37255
passenger car	FORD	1997	36872
passenger car	HONDA	1999	36616
passenger car	TOYOTA	1998	36136
passenger car	TOYOTA	2002	36130
passenger car	TOYOTA	1997	35301
passenger car	HONDA	2002	33614
passenger car	FORD	1996	33456
passenger car	FORD	2002	33311

TABLE VII
OUTPUT OF QUERY 8 (FIRST 20 ROWS)

9) Query 9: For each county_city_loc which corresponds to the city location, we count the number of case_id which gives us the number of collisions. The query takes 4103 ms.

```
SELECT l.county_city_loc as city_location,
    COUNT(c.case_id) as number_collisions
FROM Location l, Collision c
WHERE c.lid = l.lid
GROUP BY l.county_city_loc
```

```
ORDER BY number_collisions DESC
FETCH FIRST 10 ROWS ONLY;
```

CITY-LOCATION	NUMBER_COLLISIONS
1942	392405
1900	115971
3400	78820
3711	75232
109	71377
3300	60182
3404	56979
4313	56840
1941	52489
3801	47738

TABLE VIII
OUTPUT OF QUERY 9

10) Query 10: For this query we use a select case approach. We first create the daylight and night groups as they can be taken without ambiguity from the lighting attribute. Then to determine the dusk and dawn groups, we had to look at the corresponding time of the day contained in the datetime variable. We couldn't use the lighting variable for this second part as the dusk or dawn attribute was stored has "dusk or dawn" and would have been ambiguous. The query takes 34275 ms.

```
SELECT CASE
        WHEN lighting LIKE '%day%' THEN
            'daylight'
        WHEN lighting LIKE '%dark%' THEN
            'night'
        WHEN (TO_CHAR(datetime,
            'mm/dd')>='09/01' OR
            TO_CHAR(datetime,
            'mm/dd')<='03/31') AND
          TO_CHAR(datetime,
             'HH24:MI')>='06:00' AND
             TO_CHAR(datetime,
             'HH24:MI')<='07:59' then 'dawn'
        WHEN (TO_CHAR(datetime,
            'mm/dd')>='09/01' OR
            TO_CHAR(datetime,
            'mm/dd')<='03/31') AND
          TO_CHAR(datetime,
             'HH24:MI')>='18:00' AND
             TO_CHAR(datetime,
             'HH24:MI')<='19:59' then 'dusk'
        WHEN (TO_CHAR(datetime,
            'mm/dd')>='04/01' AND
            TO_CHAR(datetime,
            'mm/dd')<='08/31') AND
          TO_CHAR(datetime,
             'HH24:MI')>='04:00' AND
             TO_CHAR(datetime,
             'HH24:MI')<='05:59' then 'dawn'
        WHEN (TO_CHAR(datetime,
            'mm/dd')>='09/01' OR
            TO_CHAR(datetime,
            'mm/dd')<='03/31') AND
          TO_CHAR(datetime,
             'HH24:MI')>='20:00' AND
             TO_CHAR(datetime,
             'HH24:MI')<='21:59' then 'dusk'
        ELSE 'Unknown'

       END AS lighting,
       COUNT(*)/(SELECT COUNT(*) FROM
           Collision) AS ratio
FROM Collision C, Conditions Cd
WHERE C.cid = Cd.cid
GROUP BY CASE
        WHEN lighting LIKE '%day%' THEN
            'daylight'
        WHEN lighting LIKE '%dark%' THEN
            'night'
        WHEN (TO_CHAR(datetime,
            'mm/dd')>='09/01' OR
            TO_CHAR(datetime,
            'mm/dd')<='03/31') AND
          TO_CHAR(datetime,
             'HH24:MI')>='06:00' AND
             TO_CHAR(datetime,
             'HH24:MI')<='07:59' then 'dawn'
        WHEN (TO_CHAR(datetime,
            'mm/dd')>='09/01' OR
            TO_CHAR(datetime,
            'mm/dd')<='03/31') AND
          TO_CHAR(datetime,
             'HH24:MI')>='18:00' AND
             TO_CHAR(datetime,
             'HH24:MI')<='19:59' then 'dusk'
        WHEN (TO_CHAR(datetime,
            'mm/dd')>='04/01' AND
            TO_CHAR(datetime,
            'mm/dd')<='08/31') AND
          TO_CHAR(datetime,
             'HH24:MI')>='04:00' AND
             TO_CHAR(datetime,
             'HH24:MI')<='05:59' then 'dawn'
        WHEN (TO_CHAR(datetime,
            'mm/dd')>='09/01' OR
            TO_CHAR(datetime,
            'mm/dd')<='03/31') AND
          TO_CHAR(datetime,
             'HH24:MI')>='20:00' AND
             TO_CHAR(datetime,
             'HH24:MI')<='21:59' then 'dusk'
        ELSE 'Unknown'
       END;
```

LIGHTING	RATIO
dusk	0.005851565789685377437782557535080268669948
dawn	0.008272080749666159780616906767178416307491
night	0.2814285532000117613837156985069145285821
daylight	0.6771455787460700612313170142285003847415
Unknown	0.027302221514566640166567822962326401699

TABLE IX
OUTPUT OF QUERY 10

C. *Query optimization*

1) Query 1: The plan is:

```
---------------------------------------------
| Id  | Operation              | Name          |
---------------------------------------------
|   0 | SELECT STATEMENT       |               |
|   1 |  SORT AGGREGATE        |               |
|   2 |   INDEX FAST FULL SCAN| SYS_C00223262 |
|   3 |  HASH GROUP BY         |               |
|   4 |   TABLE ACCESS FULL    | PARTY         |
---------------------------------------------
```

However, the query can be performed by an index-only scan. We thus create an index (a B+ Tree by default) on age:

```sql
CREATE INDEX IDX_PARTYAGE ON PARTY(PARTY_AGE);
```

which modifies the plan with an index-only scan on the new index:

```
---------------------------------------------
| Id  | Operation              | Name          |
---------------------------------------------
|   0 | SELECT STATEMENT       |               |
|   1 |  SORT AGGREGATE        |               |
|   2 |   INDEX FAST FULL SCAN| SYS_C00223262 |
|   3 |  HASH GROUP BY         |               |
|   4 |   INDEX FAST FULL SCAN| IDX_PARTYAGE  |
---------------------------------------------
```

2) Query 2: The plan for the query is

```
--------------------------------------------------------
| Id  | Operation                          | Name          |
--------------------------------------------------------
|   0 | SELECT STATEMENT                   |               |
|   1 |  SORT ORDER BY                     |               |
|   2 |   VIEW                             |               |
|   3 |    WINDOW SORT PUSHED RANK         |               |
|   4 |     HASH GROUP BY                  |               |
|   5 |      NESTED LOOPS                  |               |
|   6 |       NESTED LOOPS                 |               |
|   7 |        VIEW                        | VW_NSO_1      |
|   8 |         HASH UNIQUE                |               |
|   9 |          HASH JOIN                 |               |
|  10 |           TABLE ACCESS FULL        | ROADCONDITION |
|  11 |           TABLE ACCESS FULL        | PARTY         |
|  12 |        INDEX UNIQUE SCAN           | SYS_C00223746 |
|  13 |       TABLE ACCESS BY INDEX ROWID| VEHICLE       |
--------------------------------------------------------
```

We create an index an road_cond to avoid using a full access on the Road_condition table which is not necessary for our query.

```sql
CREATE INDEX IDX_ROADCOND ON ROADCONDITION(ROAD_COND);
```

We also create an index on `IDX_PARTY_CASEID`:

```sql
CREATE INDEX IDX_PARTY_CASEID ON PARTY(CASE_ID);
```

We obtain a new plan which outputs the result in 108ms (2076 ms before)

```
-----------------------------------------------------------------------
```

```
| Id  | Operation                                    | Name              |
-------------------------------------------------------------------------
|   0 | SELECT STATEMENT                             |                   |
|   1 |  SORT ORDER BY                               |                   |
|   2 |   VIEW                                       |                   |
|   3 |    WINDOW SORT PUSHED RANK                   |                   |
|   4 |     HASH GROUP BY                            |                   |
|   5 |      NESTED LOOPS                            |                   |
|   6 |       NESTED LOOPS                           |                   |
|   7 |        VIEW                                  | VW_NSO_1          |
|   8 |         HASH UNIQUE                          |                   |
|   9 |          NESTED LOOPS                        |                   |
|  10 |           NESTED LOOPS                       |                   |
|  11 |            TABLE ACCESS BY INDEX ROWID BATCHED| ROADCONDITION    |
|  12 |             INDEX RANGE SCAN                 | IDX_ROADCOND      |
|  13 |            INDEX RANGE SCAN                  | IDX_PARTY_CASEID  |
|  14 |           TABLE ACCESS BY INDEX ROWID        | PARTY             |
|  15 |        INDEX UNIQUE SCAN                     | SYS_C00223746     |
|  16 |       TABLE ACCESS BY INDEX ROWID            | VEHICLE           |
-------------------------------------------------------------------------
```

3) Query 4: The query plan is

```
--------------------------------------
| Id  | Operation            | Name   |
--------------------------------------
|   0 | SELECT STATEMENT     |        |
|   1 |  SORT ORDER BY       |        |
|   2 |   HASH GROUP BY      |        |
|   3 |    TABLE ACCESS FULL| VICTIM |
--------------------------------------
```

We only need to access the victim_seating_position column from the victim table, thus me made an indexe on that column.

```
CREATE INDEX IDX_VIC_SIT_POS ON VICTIM(victim_seating_position);
```

The query now outputs in 1569ms instead of 5267ms before optimization. We now obtain the following plan:

```
    --------------------------------------------------
| Id  | Operation              | Name              |
--------------------------------------------------
|   0 | SELECT STATEMENT       |                  |
|   1 |  SORT ORDER BY         |                  |
|   2 |   HASH GROUP BY        |                  |
|   3 |    INDEX FAST FULL SCAN| IDX_VIC_SIT_POS |
--------------------------------------------------
```

4) Query 7: The plan for the query is

```
--------------------------------------------
| Id  | Operation              | Name      |
--------------------------------------------
|   0 | SELECT STATEMENT       |           |
|   1 |  HASH JOIN             |           |
|   2 |   TABLE ACCESS FULL    | COLLISION |
|   3 |   HASH JOIN            |           |
|   4 |    VIEW                |           |
|   5 |     FILTER             |           |
|   6 |      HASH GROUP BY     |           |
|   7 |       TABLE ACCESS FULL| VICTIM    |
```

```
|   8 |     TABLE ACCESS FULL    | PARTY      |
--------------------------------------------
```

The inner query (on victims) can be performed by an index-only scan, so we create an index on pid and age:

```sql
CREATE INDEX IDX_VIC_PIDAGE ON VICTIM(PID, VICTIM_AGE);
```

The new plan below computes now the query in 9,972 ms (13,889 ms before):

```
--------------------------------------------------
| Id  | Operation                 | Name           |
--------------------------------------------------
|   0 | SELECT STATEMENT          |                |
|   1 |  FILTER                   |                |
|   2 |   HASH GROUP BY           |                |
|   3 |    HASH JOIN              |                |
|   4 |     HASH JOIN             |                |
|   5 |      TABLE ACCESS FULL    | COLLISION      |
|   6 |      TABLE ACCESS FULL    | PARTY          |
|   7 |     INDEX FAST FULL SCAN| IDX_VIC_PIDAGE |
--------------------------------------------------
```

5) *Query 10:*

```
-----------------------------------------------
| Id  | Operation               | Name          |
-----------------------------------------------
|   0 | SELECT STATEMENT        |               |
|   1 |  SORT AGGREGATE         |               |
|   2 |   INDEX FAST FULL SCAN| SYS_C00223218 |
|   3 |  HASH GROUP BY          |               |
|   4 |   HASH JOIN             |               |
|   5 |    TABLE ACCESS FULL    | CONDITIONS    |
|   6 |    TABLE ACCESS FULL    | COLLISION     |
-----------------------------------------------
```

We create index on lighting, a full access on the conditions table is not necessary for our query which could be resolved using an index full scan.

```sql
CREATE INDEX IDX_light ON Conditions(lighting);
```

New plan computes the query in 2984ms(34275ms before)

```
---------------------------------------------------------------
| Id  | Operation                                | Name          |
---------------------------------------------------------------
|   0 | SELECT STATEMENT                         |               |
|   1 |  SORT AGGREGATE                          |               |
|   2 |   INDEX FAST FULL SCAN                   | SYS_C00223218 |
|   3 |  HASH GROUP BY                           |               |
|   4 |   HASH JOIN                              |               |
|   5 |    TABLE ACCESS BY INDEX ROWID BATCHED| CONDITIONS    |
|   6 |     INDEX FULL SCAN                      | IDX_LIGHT     |
|   7 |    TABLE ACCESS FULL                     | COLLISION     |
---------------------------------------------------------------
```