

Databases Project – Spring 2021

Team No: 24

Members: Harold Benoit, Maëlys Billon, Tom Demont

Contents

Deliverable 1	2
Change Since Last Time	2
Assumptions	2
Entity Relationship Schema	4
Schema	4
Description	4
Relational Schema	8
ER schema to Relational schema	8
DDL	11
COLLISION	11
PARTY	12
VICTIM	13
Sub-entities	13
LOCATION	13
ROAD	14
Collision_In_Road_Conditions	14
Collision_In_Weather	14
PCF	15
VEHICLE	15
Party_equipped_with	15
Victim_equipped_with	16
Associated_to	16

Deliverable 2	18
Change Since Last Time	18
Assumptions	18
Data Cleaning	18
Data Loading	19
Query Implementation	20
General Comments	28
Deliverable 3	28
Assumptions	29
Query Implementation	29
Query Performance Analysis – Indexing	42
General Comments	56

Deliverable 1

Change Since Last Time

We slightly updated the ER model to merge collision_date and collision_time together. We updated the DDL queries to be more concise. We also added primary keys to our many-to-one relations such as Collision_In_Weather.

Also, we merged the collision date and time to create a unique value TIMESTAMP that is easier to manipulate.

Assumptions

Assumptions about victims and parties:

Party are the **major players** in traffic collisions - drivers, pedestrians, bicyclists, and parked vehicles.

A **victim** is either a **party that is injured** or a person **associated with a party** (i.e. *a passenger in a car is always in the victim's table and associated with the driver of that car. The driver of that car will always be in the party's table . However the driver is in the victim's table if and only if it's injured*).

An important aspect of the relationship between the parties and victims tables is that **some parties** (i.e, drivers, bicyclists, pedestrians) will also **appear in the victim's table**. Injured parties will appear in the victim's table; uninjured parties will not. Thus, all victims need not be parties and all parties need not be victims.

Furthermore, we **cannot guarantee** that a **party** and a **victim** are the **same person** from the given data due to lack of a unique person identifier.

Constraints / Requirements:

A **collision** has **at least one party** involved.

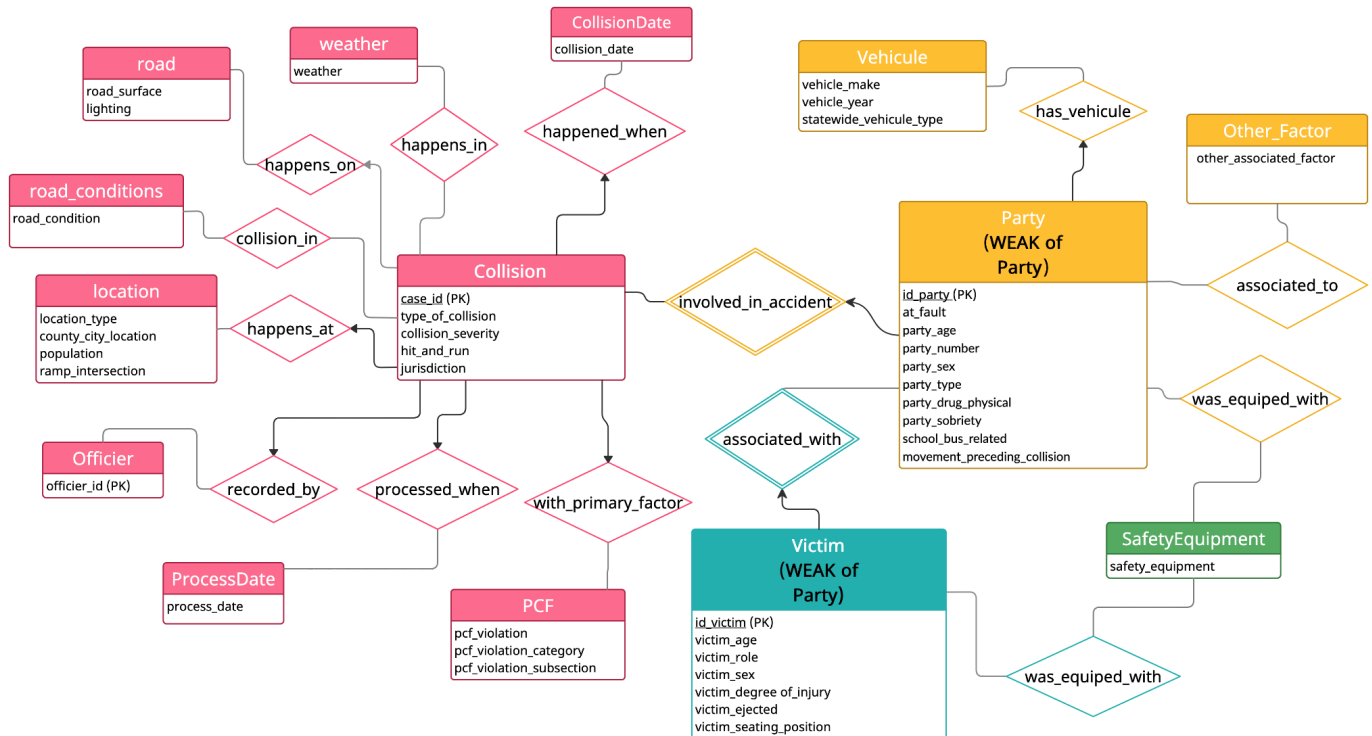
A **party** is involved in **exactly one collision**.

A **party may have no victims** associated with it.

A **victim** is part of **exactly one party**.

Entity Relationship Schema

Schema



Description

There are 3 main entity sets : **Collision**, **Victim**, and **Party**. We will describe each of them and the relations between them.

Please note that we made the choice to group related attributes of the main entities and put them in separate entities. We then link them back to the main entity via a relation. This allows us to separate the real life concepts and make a coherent ER model. It will also be an advantage considering the eventuality of adding new characteristic to a sub-entity (e.g.: *adding a new attribute to the location entity, describing also for example the environmental damages of the location, should not affect the attributes of the Collision itself*)

We may define a primary key for those entities as the entirety of their attributes.

Indeed, for example, a location exists outside of the context of a collision. But the given dataset doesn't contain a primary key for location. To uniquely identify different locations, we may define the "artificial" key as the

tuple (location_type, county_city_location, population, ramp_intersection). However, for performance reasons, we will prefer to create a unique primary key (an integer) to identify each distinct tuple.

Collision:

A collision is a traffic collision in the state of California during the year 2018.

It is described by:

- Its primary key (*case_id*)
- The main attributes listed in the box. They describe attributes that are inherently tied to the collision and don't exist outside the context of the collision.
- It happened at a given date and time (thus the relation *happened_when*).
- It happens in a certain weather (thus the relation *happens_in*).
- It happens on a certain road (thus the relation *happens_on*)
- It happens on a certain road condition (thus the relation *collision_in*).
- It happens at a certain location (thus the relation *happens_at*).
- It is recorded by a given officer (thus the relation *recorded_by*).
- It is processed at a certain date (thus the relation *processed_when*).
- It has a Primary Collision Factor (thus the relations *with_primary_factor*).

The dataset supposes that **Collision** has total participation in almost all those relations. However, we noted that some attributes (*road_condition* and *weather*) can be precised with 2 attributes. We thought it would be relevant to make the assumption that more duplicates of those attributes could be added for other data in our database. That's why we made *happens_in* and *collision_in* many-to-many relations. This allows us to qualify a collision by as many *road_conditions* and *weathers* as we might want to.

Finally, a **Collision** has **Party** related to it. It represents the link between the actors of the collision and the collision itself. This relation *involved_in_accident* is further described in the next part.

Party:

A party is a major player in traffic collisions - driver, pedestrian, bicyclist, and parked vehicles.

Inherently, **Party** is a **weak entity of Collision**. Indeed, a party does not exist / make sense outside of the context of a given collision. A party is linked to its “parent” collision by the attribute “*case_id*”.

Thus, **Party** and **Collision** are linked by the weak relationship ***involved_in_accident***.

Consequently, **Party** has total participation in this relation.

It is important to note that **Collision** has at least one party involved in every collision thus the bold line.

A party is described by :

- Its primary key (***party_id***)
- Its owner / foreign key (***case_id***)
- The main attributes listed in the box. They describe attributes that are inherently tied to a party and don't exist outside of the context of the party.
- It has a given vehicle (thus the relation ***has_vehicle***).
- It was equipped with certain safety equipment (thus the relation ***was_equipped_with***).
- It has associated factor describing its situation during the collision (thus the relation ***associated_to***)

The dataset supposes that a party is related to exactly one vehicle. It makes sense as a collision probably doesn't involve many vehicles belonging to the same party. This point would be discussed but we decided to keep that version.

On the other hand, we remarked that the dataset gives 2 precisions on the *safety_equipment* and *other_associated_factor* attribute. Similarly to *weather* and *road_condition* attributes, we made the assumption that it would be possible to have data entries with possibly more precisions on the *safety_equipment* attribute, we then decided to make those many-to-many relationships.

Finally, a party is ***associated_with*** a victim, we further describe this relation in the next part.

Victim:

A **victim** is either a **party that is injured** or a **person associated with a party**.

Inherently, **Victim** is a **weak entity of Party**. Indeed, a victim does not exist / make sense outside of the context of a given party. A victim is linked to its “parent” party by the foreign key “(case_id, party_number)”.

Thus, **Party** and **Victim** are linked by the weak relationship **associated_with**.

Consequently, **Victim** has total participation in this relationship. We may note that **Party** does not have participation constraints. Indeed, one can participate in a collision without being injured.

Because of nested weak relationships, **Victim** is technically a weak entity of **Collision** too.

A party is described by :

- Its primary key (**victim_id**)
- Its owner key ((**case_id, party_number**))
- The main attributes listed in the box. They describe attributes that are inherently tied to a victim and don't exist outside of the context of the victim.
- It was equipped with a certain safety equipment (thus the relation **was_equipped_with**).

The same reasoning as above on **safety_equipment** applies for **Victim**.

Sub-entities:

We call Sub-Entities, all the entities not described above. We consider those simpler to describe as they are directly related to at least one of the 3 main entities. They allow us to get knowledge on certain aspects of the **Collision**, **Victim** or **Party**. They are not described directly with a Primary Key, but their link to the main entities follow the [Star Schema](#) where **Collision**, **Victim** and **Party** are fact entities whereas **sub-entities** are dimension entities.

Summary:

The main points to take away are :

- The nested weak relationships from *Collision* -> *Party* -> *Victim*.
- Both victims and parties could not be described by the use of a Person entity due to lack of a unique person identifier.
- We followed a **Star Schema** logic for the description of our 3 main entities.
- *weather*, *road_condition* and *safety_equipment* are linked to their main entity through a many-to-many relationship in order to allow adding further precision on those aspects if desired.

Relational Schema

ER schema to Relational schema

We based our relational schema on the ER model. From this we defined 3 main tables : ***Collision***, ***Party*** and ***Victim***.

We recall that only ***Collision*** has a primary key that uniquely identifies a collision. The others are **weak entities** of this one.

Also, we define a table for each **sub-entity**. Those are built following the Star-Schema as explained above.

General remarks:

For every attribute, if the description of this attribute does not include the possibility “Blank or - - Not Stated”, we consider this attribute to be non-nullable.

For the relations tables ***Collision_In_Road_Conditions***, ***Collision_In_Weater***, ***Party_Equipped_With***, ***Victim_Equipped_With*** and ***Associated_To***, we made the attribute non nullable as having no stated attribute should result in having no entry in the relation table.

The weak relationships follow the ON DELETE CASCADE rule.

Also, as explained in the [Description](#), we created Integer keys to uniquely identify each tuple in our sub-entities.

Why a Star Schema:

As explained above, we decided to design a Star Schema where **Collision**, **Party** and **Victim** are the *fact entities*, *whereas sub-entities are dimension entities*. The fact that this choice reflects real life separation of our concept also brings an engineering efficiency for potential future updates of those entities' description. Indeed, if someone wants to add an attribute to a sub-entity, this can be done without having to change a huge table nesting all the attributes and potentially adding a column to Gb of Table data. Moreover, considering that the queries will be executed with conditions on few attributes, not all tables would be involved, reducing the amount of data to manipulate and optimizing this aspect of our queries.

Collision table:

For our translation, we decided to merge *recorded_by*, *processed_when* and *happened_when* relations in **Collision**. We did so because, from the data exploration, many of these attributes were almost uniquely identifying a collision. This means that their tables would've had almost as many rows as in **Collision**, reducing the interest of our dimension table. Also, considering the total participation of **Collision** in those relations, it made sense to us to merge those.

As explained above, we made **collision_in** and **happens_in** many-to-many relationships. We consequently created relation tables holding those relations: **Collision_In_Road_Conditions** and **Collision_In_Weather**.

Other sub-entities related to Collision are in "exactly one" or "at most one" relationship with **Collision** (see [Description](#)). Applying our star schema model, we store in collisions an id (a foreign key) to the attributes in the *sub-entities* tables'.

Party table:

This entity is related to **Collision** by a "weak relationship", that's why we decided to merge the relation **involved_in** and **Party** in the same table. Consequently, *case_id* is the foreign key referencing **Collision**.

Also, note even if a **Party** can be uniquely identified by the pair (*case_id*, *party_number*), we still decided to use *party_id* (the *id* attribute in victims2018.csv) as the primary key for efficiency purposes (as indexing from only an integer will be easier than from the pair of integers).

The **was_equipped_with** and **associated_to** relations were dealt with as the other many-to-many relation, yielding the **Party_equipped_With** and **Associates_To** tables.

The *has_vehicule* relation was dealt with in a Star Schema style as done for **Collision**.

Victim table:

We finally decided to build the **Victim's** table to represent the relation between **Party** and **Victim** as a **weak relationship**. The tuple (*party_number, case_id*) is the foreign key that uniquely identifies a party.

We will use *victim_id* as the primary key for the table (the *id* attribute in victims2018.csv).

The **was_equipped_with** relation is dealt with as the other many-to-many relation, yielding the **Victim_equipped_With** table.

Attributes:

Most of the attributes can be identified by an alphanumeric code. For example, lighting whose alphanumeric code is :

A - Daylight

B - Dusk - Dawn

C - Dark - Street Lights

D - Dark - No Street Lights

E - Dark - Street Lights Not Functioning

We made the choice to store those particular attributes in alphanumeric code for normalization reasons.

The translation from alphanumeric code to English can be kept in a separate table which we may call *<attribute>_translation*.

Consequently, our data is now language-agnostic, allowing us to support many languages and is also less prone to typing errors.

If the translation is needed by a query, we will simply need to join the result with a corresponding translation table.

DDL

We added constraints checks to match the data description in the handout, and make sure the database contains reliable data that is coherent with our model.

COLLISION

```
create table Collision
(
  case_id          CHAR(25)
    constraint COLLISION_PK
      primary key,
  collision_date    TIMESTAMP,
  collision_severity CHAR not null,
  county_city_location INTEGER
    constraint COLLISION_LOCATION_COUNTY_CITY_LOCATION_FK
      references LOCATION,
  hit_and_run      CHAR not null,
  jurisdiction      NUMBER(4),
  officer_id       Varchar2(20 byte),
  process_date     DATE not null,
  tow_away         INTEGER,
  type_of_collision CHAR,
  road_id          INTEGER
    constraint COLLISION_ROAD_ROAD_ID_FK
      references ROAD,
  pcf_id           INTEGER
    constraint COLLISION_PCF_PCF_ID_FK
      references PCF
);
```

PARTY

```
create table party
(
    party_id                INTEGER,
    CONSTRAINT PARTY_PK primary key (party_id),
    party_number            INTEGER,
    at_fault                NUMBER(1) default 0 not null CHECK (at_fault >=
0 AND at_fault < 2),
    cellphone_use           CHAR CHECK ( cellphone_use IN ('B', 'C', 'D')
),
    financial_responsibility CHAR CHECK ( financial_responsibility IN ('N',
'Y', 'O', 'E') ),
    hazardous_materials    CHAR CHECK ( hazardous_materials IN ('A') ),
    movement_preceding_collision CHAR CHECK (movement_preceding_collision IN
('A', 'B', 'C', 'D', 'E', 'F', 'G',
'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O',
                                'P', 'Q', 'R')),
    party_age              INTEGER CHECK ( party_age >= 0 ),
    party_drug_physical    CHAR CHECK ( party_drug_physical IN ('E', 'F',
'H', 'I', 'G') ),
    party_sex              CHAR CHECK (party_sex IN ('M', 'F')),
    party_sobriety         CHAR CHECK ( party_sobriety IN ('A', 'B', 'C',
'D', 'G', 'H')),
    party_type             CHAR CHECK (party_type IN ('1', '2', '3', '4',
'5')),
    school_bus_related     CHAR CHECK ( school_bus_related IN ('E') ),
    vehicle_id             INTEGER
        constraint PARTY_VEHICLE_VEHICLE_ID_FK
        references VEHICLE,
    case_id                CHAR(25)
        constraint PARTY_COLLISION_CASE_ID_FK
        references COLLISION
        on delete cascade
);
```

VICTIM

```
create table victim
(
    victim_id          INTEGER
        constraint VICTIM_PK
            primary key,
    victim_age          INTEGER CHECK ( victim_age >= 0 ),
    victim_degree_of_injury CHAR CHECK ( victim_degree_of_injury IN ('1', '2', '3', '4', '5', '6', '7', '0') ),
    victim_ejected      CHAR CHECK ( victim_ejected IN ('0', '1', '2', '3') ),
    victim_role         CHAR CHECK ( victim_role IN ('1', '2', '3', '4', '5', '6') ),
    victim_seating_position CHAR CHECK ( victim_seating_position IN ('A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z', '1', '2', '3', '4', '5', '6', '7', '8', '9', '0') ),
    victim_sex          CHAR CHECK ( victim_sex IN ('M', 'F') ),
    party_id            INTEGER
        constraint VICTIM_PARTY_ID_FK references PARTY on delete cascade
);
```

Sub-entities

All Sub-entities will have the same kind of organisation with their primary key, as a string encoding its attributes.

LOCATION

```
create table Location
(
    county_city_location INTEGER
        constraint LOCATION_PK
            primary key,
    population            CHAR CHECK ( population IN ('1', '2', '3', '4', '5', '6', '7', '9', '0'))
);
```

ROAD

```
create table Road
(
    road_surface      CHAR CHECK (road_surface IN ('A', 'B', 'C', 'D')),
    lighting          CHAR CHECK (lighting IN ('A', 'B', 'C', 'D', 'E')),
    location_type     CHAR CHECK (location_type IN ('H', 'I', 'R')),
    ramp_intersection CHAR CHECK (ramp_intersection IN ('1', '2', '3', '4', '5',
'6', '7', '8')),
    road_id           INTEGER
        constraint ROAD_PK
            primary key
);
```

Collision_In_Road_Conditions

```
create table Collision_In_Road_Conditions
(
    road_conditions CHAR(1) CHECK ( road_conditions IN ('A', 'B', 'C', 'D', 'E',
'F', 'G', 'H') ) NOT NULL,
    case_id         CHAR(25)
        constraint COLLISION_IN_ROAD_CONDITION_COLLISION_CASE_ID_FK
            references COLLISION
                on delete cascade,
    constraint COLLISION_IN_ROAD_CONDITION_PKEY primary key (case_id,
road_conditions)
);
```

Collision_In_Weather

```
create table Collision_In_Weather
(
    case_id CHAR(25)
        constraint COLLISION_IN_WEATHER_COLLISION_CASE_ID_FK
            references COLLISION
                on delete cascade,
    weather CHAR CHECK ( weather IN ('A', 'B', 'C', 'D', 'E', 'F', 'G') ) NOT
NULL,
```

```
constraint COLLISION_IN_WEATHER_PKEY primary key (case_id, weather)

);
```

PCF

```
create table Pcf
(
    pcf_violation          NUMBER,
    pcf_violation_category NUMBER(2) CHECK ( pcf_violation_category >= 0 AND
pcf_violation_category <= 24 ),
    pcf_violation_subsection CHAR,
    primary_collision_factor CHAR CHECK ( primary_collision_factor IN ('A', 'B',
'C', 'D', 'E') ),
    pcf_id                 INTEGER constraint PCF_PK primary key
);
```

VEHICLE

```
create table Vehicle
(
    statewide_vehicle_type CHAR(1) CHECK (statewide_vehicle_type IN
('A', 'B', 'C', 'D', 'E', 'F', 'G',
'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O')),
    vehicle_make           CHAR(30),
    vehicle_year           NUMBER(4),
    vehicle_id             INTEGER constraint VEHICLE_PK primary key
);
```

Party_equipped_with

```
create table Party_Equipped_With
(
    party_id              INTEGER
constraint PARTY_EQUIPPED_WITH_PARTY_PARTY_ID_FK
references PARTY
```

```
        on delete cascade,  
    safety_equipment CHAR CHECK ( safety_equipment IN  
                                ('A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'J',  
'K', 'L', 'M', 'N', 'P', 'Q', 'R', 'S',  
                                'T', 'U', 'V', 'W', 'X', 'Y') ) NOT NULL,  
    constraint PARTY_EQIPPED_WITH_PKEY primary key (party_id, safety_equipment)  
);
```

Victim_equipped_with

```
create table Victim_Equipped_With  
(  
    victim_id          INTEGER  
        constraint VICTIM_EQIPPED_WITH_VICTIM_VICTIM_ID_FK  
        references VICTIM  
        on delete cascade,  
    safety_equipment CHAR CHECK ( safety_equipment IN  
                                ('A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'J',  
'K', 'L', 'M', 'N', 'P', 'Q', 'R', 'S',  
                                'T', 'U', 'V', 'W', 'X', 'Y') ) NOT NULL,  
    constraint VICTIM_EQIPPED_WITH_PKEY primary key (victim_id, safety_equipment)  
);
```

Associated_to

```
create table Associated_to  
(  
    party_id          INTEGER  
        constraint  
        ASSOCIATED_TO_PARTY_PARTY_ID_FK  
        references PARTY  
        on delete cascade,  
    other_associated_factor CHAR(1) CHECK ( other_associated_factor IN ('A',  
'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T',  
'U', 'V', 'W', 'X', 'Y') ) NOT NULL,  
    constraint PARTY_ASSOCIATED_TO_PKEY primary key (party_id,  
other_associated_factor)  
);
```


General Comments

The project work was done throughout multiple discord sessions where all team members were present, and otherwise tasks were divided equally for each team member.

Deliverable 2

Change Since Last Time

We have changed and added some things since the last feedback (deliverable 2). First we have added this section : “Change since last time” , to show in this deliverable and the first the changes that have been made since the last feedback. We added the results of the queries we did not manage to run for the deadline, and screenshot the twenty first (rows) of the result.

We have unnested the queries where we could do so, some nesting is mandatory.

Query 3 : We were counting only the number of different roads where the dark light conditions were respected. So we changed the query and used an inner join.

Query 5: We selected all the day with their number of collisions, however we only need to select the one with the most collisions, this is what we did using (FETCH).

Query 7: We used less tables because we now use an inner join between party and Collision_In_Road_Condition

Query 9: Before we have counted multiple times the same collisions if they verified multiple conditions. Now with DISTINCT case_id we count them only once.

Query 10: We have deleted the attribute collision_time since the information is in collision_date.

Assumptions

In our queries, whenever we are asked about some properties of the data, we always ignore the rows that don't have the property i.e (NULL). An example would be for the Query 3 where we are asked “Find the fraction of total collisions that happened under dark lighting conditions' '. In this case, we do not count the collisions which do not have the information on their lighting conditions when computing the ratio.

Data Cleaning

We examined every single attribute of each csv files to see how the description corresponded to the way it was written in the files. Using this knowledge, we used Pandas to translate every single attribute from its written form (e.g. "(Vehicle) Code Violation") to its alphanumeric translation (e.g. 'A').

It's a tedious job but it enables us to write queries without being worried that our results are wrong because, for example, "Driving or Bicycling Under the Influence of Alcohol or Drug" was written as "dui" in the database.

Here's a list of interesting parts of the data cleaning where he had to make assumptions:

We saw that *party_drug_physical* had a weird G value appearing in 500'000 rows, we decided to accept it as valid and not make it NULL even though we don't know its translation as it might be valuable information. We may interpret it as G='UNKNOWN'

cellphone_use had 1,2,3 values appearing in many rows, so we made the reasonable assumption that they were a bijective mapping to the set (B,C,D) and changed the data accordingly.

Many attributes that didn't have "Blank -- or Not Stated" in their description had missing values.

They were : *jurisdiction*, *party_age*, *victim_age*, *vehicle_make*, *vehicle_year*, *pcf_violation*, *tow_away* and *collision_time*. In an effort to keep as much data as possible, we allowed these types to be nullable.

It was also remarked that *officer_id* didn't have a standardized input, so we allowed it to be nullable and a varchar.

pcf_violation_subsection is encoded in ('A','B',...) which does not correspond to the described encoding of *pcf_violation_category*. Since we don't know the mapping, we kept it as is, but we can't interpret it.

case_id, initially thought of as an INTEGER, revealed easier to deal with being a string where we could make a difference between data with only leading zeros difference. Exploring the data made us think that considering this as CHAR(25) would be enough (as the longest *case_id* is 19 digits long).

Summary:

We made a conscious effort to drop as little as possible data and we actually only dropped one row, it was a single row where *hit_and_run* was equal to D and we made the choice to drop it.

Data Loading

As we made the choice of the star schema, a lot of preprocessing had to be done to effectively create the tables from the original CSVs.

The sub-entities:

- Road(county city location, *population*)
- Location(*road_surface*, *lighting*, *location_type*, *ramp_intersection*, road_id)

- `Pcf(pcf_violation, pcf_violation_category, pcf_violation_subsection, pcf_id)`
- `Vehicule(statewide_vehicle_type, vehicule_make, vehicule_year, vehicle_id)`

were made using the following procedure:

- enumerate all possible combinations found in the collisions table.
- if there's an attribute suitable for a primary key, use it
- Otherwise, generate an integer primary key

Query Implementation

- **QUERY 1 : List the year and the number of collisions per year. Suppose there are more years than just 2018.**

First, we need to take only the year inside `collision_date`. We use the function `EXTRACT` (The `EXTRACT()` function extracts some part from a given date) with the argument `YEAR` so it would extract only the year as we want.

We then order it by descendent order (for `count(*)`, which is the number of collisions per year), so the year with more collisions will be first.

```
SELECT COUNT(*) AS number_of_collisions_per_year,  
EXTRACT(YEAR FROM C.CRASH_DATE) AS YEAR_OF_COLLISION  
FROM Collision C  
GROUP BY EXTRACT(YEAR FROM C.CRASH_DATE)  
ORDER BY COUNT(*) DESC ;
```

Query result (if the result is big, just a snippet)

	NUMBER_OF_COLLISIONS_PER_YEAR ↕	YEAR_OF_COLLISION ↕
1	540814	2002
2	534829	2003
3	533996	2004
4	527975	2005
5	519169	2001
6	497313	2007
7	494293	2006
8	29645	<null>
9	21	2018
10	7	2017

- **QUERY 2 : Find the most popular vehicle_make in the database. Also list the number of vehicles of that particular make.**

We group the vehicle by their vehicle_make arguments, and we count them by similar arguments. We have to select only the first row as they have been ordered by descending order.

SQL statement

```
SELECT V.vehicle_make, COUNT(*)
FROM Vehicle V
GROUP BY V.vehicle_make
ORDER BY COUNT(*) DESC
FETCH FIRST 1 ROWS ONLY
```

Query result (if the result is big, just a snippet)

	VEHICLE_MAKE ↕	COUNT(*) ↕
1	FORD	770

- **QUERY 3 : Find the fraction of total collisions that happened under dark lighting conditions.**

Here we use an inner join on collision and road (with the condition that C.road_id = R.road_id).

Then we will use the sum function; the sum will count all the cases where the dark collisions condition is respected. We will then divide this result into the count of all collisions.

SQL statement

```
Select  SUM(CASE WHEN R.LIGHTING IN ('C','D','E') THEN 1 ELSE 0
END)/COUNT(DISTINCT C.case_id) AS RATIO_OF_DARK_COLLISIONS
FROM COLLISION C
INNER JOIN ROAD R
ON R.road_id = C.road_id;
```

Query result (if the result is big, just a snippet)

	RATIO_OF_DARK_COLLISIONS
1	0.2798142065033161485586702997393736157792

- **QUERY 4 : Find the number of collisions that have occurred under snowy weather conditions.**

Take the collisions that happen under snowy weather ('D') and count them.

SQL statement

```
SELECT COUNT(*)
FROM Collision_In_Weather CiW
WHERE CiW.weather='D'
```

Query result (if the result is big, just a snippet)

	COLLISIONS_UNDER_SNOWY_CONDITION
1	8530

- **QUERY 5 : Compute the number of collisions per day of the week, and find the day that witnessed the highest number of collisions. List the day along with the number of collisions.**

We need to find the day of the week that corresponds to collision_date. Oracle has a function called TO_CHAR(date, 'DAY') that will return the day of the week corresponding to the date.

So we group the collisions by their weekdays, we count each collision that happens on the same weekday. We then order them by descending order to have the day that witnesses the highest number of collisions at the top, and we “FETCH” only the first row to have only the day that witnessed the highest number of collisions.

SQL statement

```
SELECT COUNT(*) AS number_of_collisions_per_weekday, TO_CHAR(C.COLLISSION_DATE,
'DAY')
FROM Collision C
GROUP BY TO_CHAR(C.COLLISSION_DATE, 'DAY')
ORDER BY COUNT(*) DESC
FETCH FIRST 1 ROWS ONLY;
```

Query result (if the result is big, just a snippet)

	NUMBER_OF_COLLISIONS_PER_WEEKDAY	WEEKDAY
1	610673	VENDREDI

- **QUERY 6** :List all weather types and their corresponding number of collisions in descending order of the number of collisions.

Group all the collisions by the weather they have happened under and order them in descending order (from the most frequent weather to the less likely).

SQL statement

```
SELECT CiW.weather, COUNT(*)
FROM Collision_In_Weather Ciw
GROUP BY CiW.weather ORDER BY COUNT(*) DESC
```

Query result (if the result is big, just a snippet):

	WEATHER	COUNT(*)
1	A	2941042
2	B	548250
3	C	223752
4	E	21259
5	G	13952
6	D	8530
7	F	6960

- **QUERY 7 : Find the number of at-fault collision parties with financial responsibility and loose material road conditions.**

We inner join Party with Collision_In_Road_Conditions (on the condition that P.case_id = CiR.case_id).

while filtering on the conditions at_fault, financial_responsibility, and road_conditions and count all the cases that check all those conditions.

SQL statement

```
SELECT COUNT(*)
FROM Party P
INNER JOIN COLLISION_IN_ROAD_CONDITIONS CIR on P.CASE_ID = CIR.CASE_ID
WHERE P.at_fault= 1 AND P.FINANCIAL_RESPONSIBILITY= 'Y'
      AND CiR.road_conditionS = 'B';
```

Query result (if the result is big, just a snippet)

	COUNT(DISTINCT CASE_ID) ▾
1	4803

- **QUERY 8 : Find the median victim age and the most common victim seating position**

We use two SQL functions : STATS_MODE that finds the most common value of a column (here victim_seating_position), and MEDIAN that computes the median value of a designated column (here victim_age).

SQL statement

```
SELECT STATS_MODE(VICTIM_SEATING_POSITION) AS MOST_COMMON_SEATING_POSITION,
MEDIAN(VICTIM_AGE) AS AGE_MEDIAN
FROM VICTIM V
```

Query result (if the result is big, just a snippet)

	MOST_COMMON_SEATING_POSITION ▾	AGE_MEDIAN ▾
1	3	25

- **QUERY 9 : What is the fraction of all participants that have been victims of collisions while using a belt?**

First, we count the number of distincts victims using a seating belt (E G and C corresponds to a victim using one). Then we divide it by the sum of the number of parties and number of victims.

We use the trunc function to make the long number look prettier

SQL statement

```
SELECT TRUNC(NUMBER_OF_VICTIM_USING_BELT/((SELECT COUNT(*) FROM PARTY P)+
(SELECT COUNT(*) FROM VICTIM V)),5) AS RATIO_OF_VICTIM_USING_BELT
FROM (SELECT COUNT(DISTINCT VW.victim_id) NUMBER_OF_VICTIM_USING_BELT
FROM VICTIM_EQIPPED_WITH VW WHERE VW.SAFETY_EQUIPMENT IN ('E','G','C'))
```

Query result (if the result is big, just a snippet)

	RATIO_OF_VICTIM_USING_BELT
1	0.2691

- **QUERY 10** : Compute the fraction of the collisions happening for each hour of the day (for example, x% at 13, where 13 means period from 13:00 to 13:59). Display the ratio as percentage for all the hours of the day.

First we group Collision by the hour at when the collision occurs (we extract the hour from collision_date thanks to the EXTRACT function). Then we order it by count(*). We divide this result by the total number of collisions and multiply it by 100 to obtain a percentage. We use the trunc function to make the big decimal number to a prettier one.

SQL statement

```
SELECT TRUNC((COUNT(*)/(SELECT COUNT(*)
                        FROM COLLISION C)) * 100,5) AS percentage,
EXTRACT (HOUR FROM C.COLLISION_DATE) AS HOUR_OF_COLLISIONS
FROM Collision C
GROUP BY EXTRACT(HOUR from c.COLLISION_DATE)
ORDER BY COUNT(*) DESC;
```

Query result (if the result is big, just a snippet)

	PERCENTAGE ↕	HOUR_OF_COLLISIONS ↕
1	7.90707	17
2	7.74804	15
3	7.33087	16
4	6.54757	14
5	6.30051	18
6	5.77554	12
7	5.77526	13
8	5.23359	8
9	5.17068	7
10	4.89138	11
11	4.42863	19
12	4.22711	10
13	4.0881	9
14	3.48963	20
15	3.28186	21
16	2.86186	22
17	2.62328	6
18	2.38451	23
19	1.90845	0
20	1.82982	1

General Comments

The project work was done throughout multiple discord sessions where all team members were present, and otherwise tasks were divided equally for each team member.

Deliverable 3

Assumptions

Assumptions related to a specific query are explained in the related explanation.

Query Implementation

- **QUERY 1:** For the drivers of age groups: underage (less and equal to 18 years), young I [19, 21], young II [22,24], adult [24,60], elder I [61,64], elder II [65 and over], find the ratio of cases where the driver was the party at fault. Show this ratio as a percentage and display it for every age group.

Description of logic:

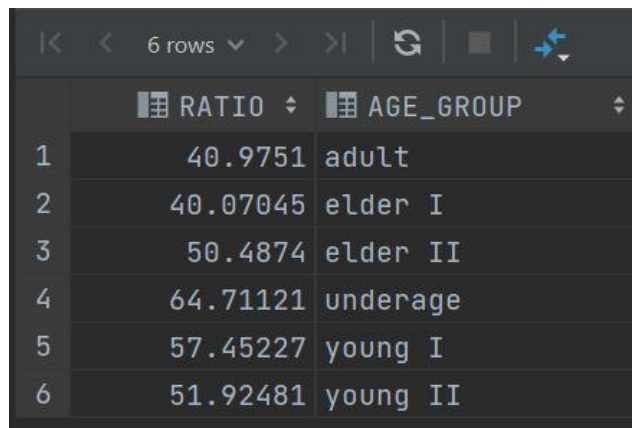
Here, the key point is that we first compute for every party its age category. We then inner join this result with the Party itself and we only keep drivers. We explicitly ask for the age group to be non null. We then group-by age categories and compute the ratio using a sum with a case statement.

SQL statement

```
select  TRUNC(((SUM(case when AT_FAULT = 1 then 1 else 0
END)/count(*))*100.0),5) as RATIO,AGE_GROUP
from    PARTY P
INNER JOIN (
  (SELECT (CASE
    WHEN P1.PARTY_AGE <= 18
    THEN 'underage'
    WHEN P1.PARTY_AGE >= 19 AND P1.PARTY_AGE <= 21
    THEN 'young I'
    WHEN P1.PARTY_AGE >= 22 AND P1.PARTY_AGE <= 24
    THEN 'young II'
    WHEN P1.PARTY_AGE >= 24 AND P1.PARTY_AGE <= 60
    THEN 'adult'
    WHEN P1.PARTY_AGE >= 61 AND P1.PARTY_AGE <= 64
    THEN 'elder I'
    WHEN P1.PARTY_AGE >=65
    THEN 'elder II'
    END) AS AGE_GROUP,
    P1.PARTY_ID AS new_id
```

```
FROM PARTY P1)) on P.PARTY_ID = new_id
where P.PARTY_TYPE = '1' AND PARTY_AGE is not null
group by AGE_GROUP
order by AGE_GROUP;
```

Query result (if the result is big, just a snippet)



	RATIO	AGE_GROUP
1	40.9751	adult
2	40.07045	elder I
3	50.4874	elder II
4	64.71121	underage
5	57.45227	young I
6	51.92481	young II

Based on this, we would increase our insurance prices for underage drivers.

- **QUERY 2: Find the top-5 vehicle types based on the number of collisions on roads with holes. List both the vehicle type and their corresponding number of collisions.**

Description of logic:

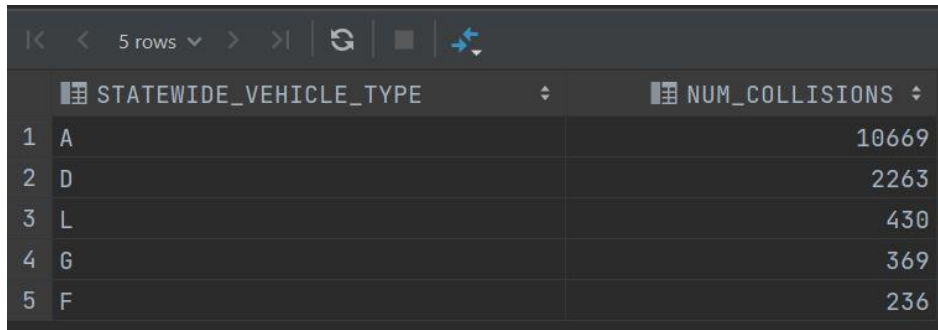
To get the collisions on roads with holes, we simply filter and join our relation Collision_In_Road_Conditions with Party on case_id. This works because CiR tells what road_conditions had a given case_id. Then we join with Vehicle and explicitly ask for the vehicle type to be non null. We then group by vehicle type and count the number of collisions. We compute count(*) and not count(distinct case_id) because there might be many vehicles involved in one collision.

SQL statement

```
SELECT V.STATEWIDE_VEHICLE_TYPE, COUNT(*) as num_collisions
FROM Vehicle V, COLLISION_IN_ROAD_CONDITIONS CiR, COLLISION C, PARTY P
WHERE CiR.road_conditions = 'A' AND CiR.case_id=P.case_id AND P.vehicle_id =
V.vehicle_id
```

```
AND V.STATEWIDE_VEHICLE_TYPE is not null
GROUP BY V.STATEWIDE_VEHICLE_TYPE
ORDER BY COUNT(*) DESC
FETCH FIRST 5 ROWS ONLY;
```

Query result (if the result is big, just a snippet)



	STATEWIDE_VEHICLE_TYPE	NUM_COLLISIONS
1	A	10669
2	D	2263
3	L	430
4	G	369
5	F	236

A = Passenger Car/Station Wagon, D = Pickup or Panel Truck, L = Bicycle, G = Truck or Truck Tractor with Trailer, F = Truck or Truck Tractor

- **QUERY 3: Find the top-10 vehicle makes based on the number of victims who suffered either a severe injury or were killed. List both the vehicle make and their corresponding number of victims.**

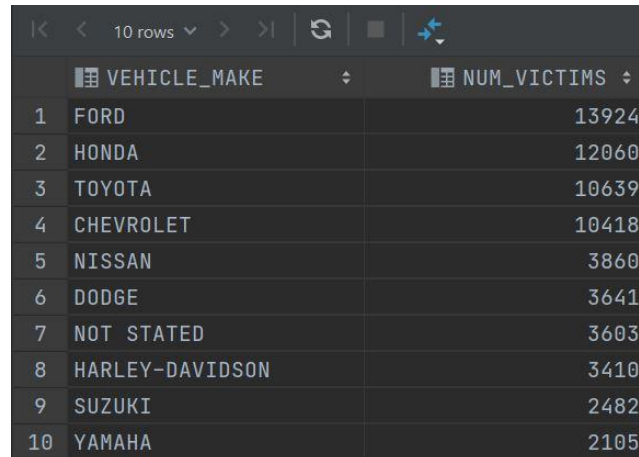
Description of logic:

This is a fairly classic query. We first join the tables party and victim to link vehicle info and victims. We explicitly filter for injuries and non null vehicle make. We then group by vehicle_make , order by count (*) and keep the first 10 rows.

SQL statement

```
SELECT Ve.vehicle_make, COUNT(*) as num_victims
FROM VICTIM V, PARTY P, VEHICLE Ve
WHERE V.party_id = P.party_id AND P.vehicle_id= Ve.vehicle_id AND
(V.victim_degree_of_injury IN ('1','2'))
AND Ve.VEHICLE_MAKE is not null
GROUP BY Ve.vehicle_make
ORDER BY COUNT(*) DESC
FETCH FIRST 10 ROWS ONLY;
```

Query result (if the result is big, just a snippet)



	VEHICLE_MAKE	NUM_VICTIMS
1	FORD	13924
2	HONDA	12060
3	TOYOTA	10639
4	CHEVROLET	10418
5	NISSAN	3860
6	DODGE	3641
7	NOT STATED	3603
8	HARLEY-DAVIDSON	3410
9	SUZUKI	2482
10	YAMAHA	2105

- **QUERY 4: Compute the safety index of each seating position as the fraction of total incidents where the victim suffered no injury. The position with the highest safety index is the safest, while the one with the lowest is the most unsafe. List the most safe and unsafe victim seating position along with its safety index.**

Description of logic:

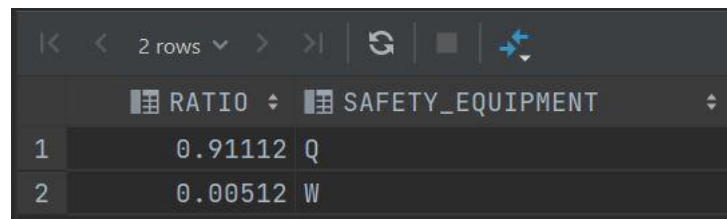
Even though the query might look nested, it's actually quite simple and the nesting is just a necessity to get only the first and last row.

To compute the safety index of each seating position, we first join Victim and Victm_equipped_with to get the info of the safety equipment worn by a victim. Then we simply group by safety_equipment and compute the ratio (# of no injuries/ total # of injuries). The upper levels of nesting are simply a trick to compute the row number and then filter by the first and last row number.

SQL statement


```
select Ratio,SAFETY_EQUIPMENT
FROM
  (select Ratio, SAFETY_EQUIPMENT, ROWNUM rn
    FROM (select trunc(sum(case when v.VICTIM_DEGREE_OF_INJURY = '0' THEN 1
ELSE 0 END)/count(*),5) AS Ratio,
          VW.SAFETY_EQUIPMENT
from VICTIM_EQIPPED_WITH VW, VICTIM V
WHERE v.VICTIM_ID = VW.VICTIM_ID
group by VW.SAFETY_EQUIPMENT
order by Ratio DESC))
WHERE rn in(1,23);
```

Query result (if the result is big, just a snippet)



	RATIO	SAFETY_EQUIPMENT
1	0.91112	Q
2	0.00512	W

Q= Child Restraint in Vehicle Used, W = Driver, Motorcycle Helmet Used.

The query seems to give us reasonable results. One might be surprised that Driver, Motorcycle Helmet Not used isn't the lowest one. After taking a look, we saw that it was indeed very close from being the lowest. Overall, we can probably conclude that driving a motorcycle is quite dangerous.

- **QUERY 5: How many vehicle types have participated in at least 10 collisions in at least half of the cities?**

Description of logic:

This one is a bit more complicated. First let's precise our interpretation of the query. A vehicle type qualifies if it has participated in at least 10 collisions in at least half of the cities, **meaning it should have participated in at least 10 collisions for each city** (in half of the cities). Now onto the explanation.

First, at the lowest level of nesting, we compute the number of collisions per vehicle type and city. This is done by joining vehicle, party, collision (with vehicle type non null) and grouping by (vehicle type,city_location) and filtering for at least 10 collisions.

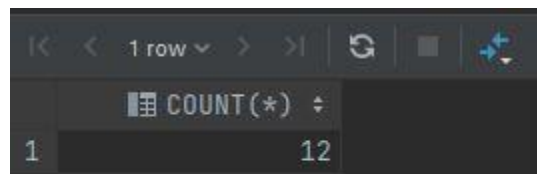
Selecting from this, we group by vehicle_type to compute the number of cities per vehicle type. We additionally add the total number of cities for the last level of nesting.

Finally, selecting from above, we count the number of vehicle types which have participated in at least 10 collisions in at least half of the cities by filtering on (num_cities_per_type >= num_cities/2).

SQL statement:

```
select count(*) FROM
  (select STATEWIDE_VEHICLE_TYPE,count(num_collisions_per_city_per_type)
   as num_cities_per_type,
   (select count(*) from LOCATION)
   as num_cities
   FROM
     (select STATEWIDE_VEHICLE_TYPE, count(*) as
num_collisions_per_city_per_type
     FROM ((VEHICLE V
      INNER JOIN PARTY P on V.VEHICLE_ID = P.VEHICLE_ID)
      INNER JOIN COLLISION C on C.CASE_ID = P.CASE_ID)
      WHERE V.STATEWIDE_VEHICLE_TYPE IS NOT NULL
      GROUP BY V.STATEWIDE_VEHICLE_TYPE,C.COUNTY_CITY_LOCATION
      HAVING count(*) >= 10)
   group by STATEWIDE_VEHICLE_TYPE)
WHERE num_cities_per_type >= num_cities/2;
```

Query result (if the result is big, just a snippet)



The screenshot shows a database query result interface. At the top, there are navigation controls: a left arrow, a right arrow, a dropdown menu showing '1 row', and a refresh button. Below these, the column header is 'COUNT(*)'. The result table has one row with the value '12'.

COUNT(*)
12

- **QUERY 6: For each of the top-3 most populated cities, show the city location, population, and the bottom-10 collisions in terms of average victim age (show collision id and average victim age).**

Description of logic:

This query is quite complicated. There are multiple inner joins with a specific order to get the best running time possible by filtering as early as possible. The query works as follows: 1. Compute the top 3 most populated cities 2. Join it with collision on case_id to get a reduction factor of 3/5440 3. Join it with Party on case_id to get a similar reduction factor 4. Finally join it with Victim on case_id and compute the average victim age by using AVG(victim_age) over (partition by case_id) 5. Then, to be able to take the bottom-10 collisions in terms of average victim age, we select this from this and add a row_number to each row computed over (partition by city_code order by avg_age) 6. Finally, we select from this by filtering with row_num <= 10.

SQL statement

```
SELECT city_code, population, CASE_ID, avg_age
FROM (SELECT CASE_ID, city_code, avg_age, population, row_number() over
(partition by city_code order by avg_age) as rn

      FROM ( SELECT P.CASE_ID , AVG(V.VICTIM_AGE) over (partition by P.CASE_ID) as
avg_age, city_code, population
            FROM (VICTIM V
                  INNER JOIN
                    (PARTY P
                  INNER JOIN
                    (COLLISION C
                  INNER JOIN
                    (select COUNTY_CITY_LOCATION as city_code, POPULATION as population
                     FROM LOCATION L
                     order by decode(POPULATION, '7', 10, '6', 9, '5', 8,
                                     '4', 7, '3', 6, '2', 5, '1', 4, '9', 3, '0', 2, NULL,
                                     1) DESC
                     FETCH FIRST 3 rows only)
                     on C.COUNTY_CITY_LOCATION = city_code)
                     on P.CASE_ID = C.CASE_ID)
                     on V.PARTY_ID = P.PARTY_ID)))
```

```
WHERE rn <= 10
ORDER BY city_code;
```

Query result (if the result is big, just a snippet)

	CITY_CODE	POPULATION	CASE_ID	AVG_AGE
1	1005	7	0360320	0
2	1005	7	3394266	0
3	1005	7	3238620	0
4	1005	7	3238600	0
5	1005	7	3052397	0
6	1005	7	3050857	0
7	1005	7	2995098	0
8	1005	7	2981117	0
9	1005	7	2864176	0
10	1005	7	2661658	0
11	1942	7	0006509	0
12	1942	7	3554643	0
13	1942	7	3549474	0
14	1942	7	3542121	0
15	1942	7	3542121	0
16	1942	7	3523765	0
17	1942	7	3514296	0
18	1942	7	3431488	0
19	1942	7	3426044	0
20	1942	7	3414375	0
21	3711	7	0060439	0
22	3711	7	3549191	0
23	3711	7	3513429	0
24	3711	7	3468111	0
25	3711	7	3399703	0
26	3711	7	3162886	0
27	3711	7	3154613	0
28	3711	7	3098243	0
29	3711	7	3091897	0
30	3711	7	2969999	0

- **QUERY 7:** Find all collisions that satisfy the following: the collision was of type pedestrian and all victims were above 100 years old. For each of the qualifying collisions, show the collision id and the age of the eldest collision victim.

Description of logic:

In this query, we first join Collision, Party and Victim to connect collisions and victims. We obviously filter by collision of type pedestrian. The key insight here is to group by case_id and explicitly ask for the MIN(victim_age) > 100. If the minimum victim age is above 100 years old then all victims in this collision are.

We then simply output MAX(victim_age) to get the age of the eldest.

SQL statement

```
select c.CASE_ID, max(VICTIM_AGE)
FROM COLLISION C, PARTY P, VICTIM V
WHERE C.CASE_ID= P.CASE_ID AND P.PARTY_ID = V.PARTY_ID
AND C.TYPE_OF_COLLISION = 'G'
GROUP BY C.CASE_ID
HAVING min(VICTIM_AGE) > 100;
```

Query result (if the result is big, just a snippet)

	CASE_ID	MAX(VICTIM_AGE)
1	0445265	101
2	0644226	103
3	0851026	106
4	1347636	101
5	0828116	102
6	1548445	102
7	0069198	101
8	0439197	102
9	0820619	101
10	0817210	102
11	1209166	101
12	3485436	101
13	1373664	101
14	3388544	105
15	1213340	121
16	0868472	103
17	2472739	103
18	0036446	110
19	1847678	104
20	2531557	103

- **QUERY 8: Find the vehicles that have participated in at least 10 collisions. Show the vehicle id and number of collisions the vehicle has participated in, sorted according to number of collisions (descending order). What do you observe?**

Description of logic:

This query is fairly easy. The right side of the join simply groups by vehicle_id in Party, filtering on num_collisions >= 10. Note that num_collision is well defined because ,for a single collision, there might be multiple cars participating each with a different vehicle_id. Thus it is important to compute in the table Party. Finally we join with Vehicle to get the vehicle_make and year. We explicitly ask for vehicle_make and year to be non null.

SQL statement

```
SELECT V.VEHICLE_MAKE, V.VEHICLE_YEAR, num_collision
FROM
VEHICLE V
INNER JOIN
  (SELECT P.VEHICLE_ID as id, COUNT(*) AS num_collision
  FROM PARTY P
  GROUP BY P.vehicle_id
  HAVING COUNT(*) >= 10) ON V.VEHICLE_ID = id
WHERE V.VEHICLE_MAKE IS NOT NULL AND V.VEHICLE_YEAR IS NOT NULL
ORDER BY num_collision DESC;
```

Query result (if the result is big, just a snippet)

	VEHICLE_MAKE	VEHICLE_YEAR	NUMBER_OF_COLLISION
1	TOYOTA	2000	52504
2	FORD	2000	51943
3	HONDA	2000	50284
4	FORD	1998	49182
5	TOYOTA	2001	47232
6	HONDA	2001	45277
7	FORD	2001	45236
8	TOYOTA	1999	42941
9	HONDA	1998	42091
10	FORD	1999	41948
11	FORD	1995	40246
12	HONDA	1997	39210
13	FORD	1997	38885
14	HONDA	1999	38556
15	TOYOTA	2002	38427
16	TOYOTA	1998	38012
17	TOYOTA	1997	37158
18	TOYOTA	2003	35943
19	HONDA	2002	35785
20	FORD	2002	35460

We observe that TOYOTA, HONDA and FORD occupy the top 20 in terms of number of collisions. Whether that means that these cars are unsafe or simply that many people drive them cannot be inferred from that query alone.

- **QUERY 9: Find the top-10 (with respect to number of collisions) cities. For each of these cities, show the city location and number of collisions.**

Description of logic:

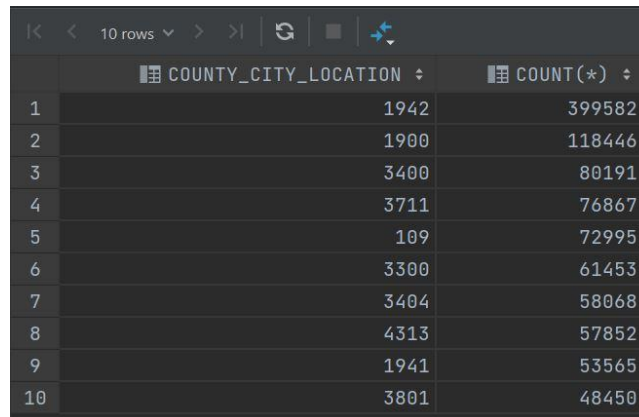
This query is very simple. We group by city_location in Collision and order by COUNT(*).

SQL statement

```
SELECT C.county_city_location, COUNT(*)
FROM COLLISION C
```

```
GROUP BY C.county_city_location
ORDER BY COUNT(*) DESC
FETCH FIRST 10 ROWS ONLY
```

Query result (if the result is big, just a snippet)



	COUNTY_CITY_LOCATION	COUNT(*)
1	1942	399582
2	1900	118446
3	3400	80191
4	3711	76867
5	109	72995
6	3300	61453
7	3404	58068
8	4313	57852
9	1941	53565
10	3801	48450

- **QUERY 10:** Are there more accidents around dawn, dusk, during the day, or during the night? In case lighting information is not available, assume the following: the dawn is between 06:00 and 07:59, and dusk between 18:00 and 19:59 in the period September 1 - March 31; and dawn between 04:00 and 06:00, and dusk between 20:00 and 21:59 in the period April 1 - August 31. The remaining corresponding times are night and day. Display the number of accidents, and to which group it belongs, and make your conclusion based on absolute number of accidents in the given 4 periods.

Description of logic:

This query looks very complicated but it is actually fairly simple. We simply first compute for each collision its time period using a case statement. We first check whether we have lighting information to decide on night or day. Since dusk/dawn is defined ambiguously by lighting, we only decide on it using the collision time. From this, we simply group it by time_period and compute count(*).

SQL statement

```
SELECT count(*) as num_collisions, time
FROM
(select (
CASE
  -- info on lighting
  WHEN R.LIGHTING = 'A' THEN 'day'
```



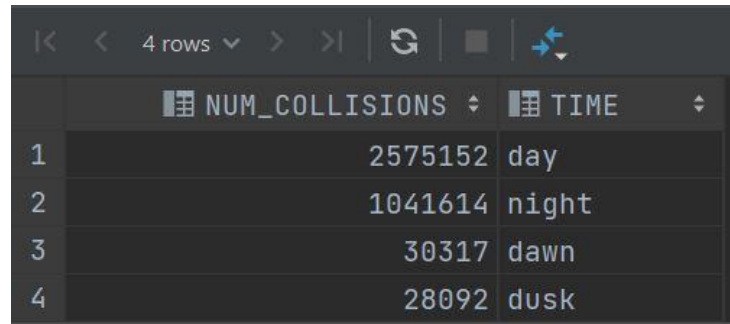
```
    WHEN R.LIGHTING IN ('C','D','E') THEN 'night'

    -- winter hour
    WHEN (extract(month from C.CRASH_DATE) >= 9 OR extract(month from
C.CRASH_DATE) <= 3)
        AND (extract(hour from C.CRASH_DATE) IN (6,7)) THEN 'dawn'
    WHEN (extract(month from C.CRASH_DATE) >= 9 OR extract(month from
C.CRASH_DATE) <= 3)
        AND (extract(hour from C.CRASH_DATE) IN (18,19)) THEN 'dusk'
    WHEN (extract(month from C.CRASH_DATE) >= 9 OR extract(month from
C.CRASH_DATE) <= 3)
        AND (extract(hour from C.CRASH_DATE) <=5 OR extract(hour from
C.CRASH_DATE) >= 20 ) THEN 'night'
    WHEN (extract(month from C.CRASH_DATE) >= 9 OR extract(month from
C.CRASH_DATE) <= 3)
        AND (extract(hour from C.CRASH_DATE) >=8 OR extract(hour from
C.CRASH_DATE) <= 17 ) THEN 'day'

    -- summer hour
    WHEN (extract(month from C.CRASH_DATE) <= 8 OR extract(month from
C.CRASH_DATE) >= 4)
        AND (extract(hour from C.CRASH_DATE) IN (4,5)) THEN 'dawn'
    WHEN (extract(month from C.CRASH_DATE) <= 8 OR extract(month from
C.CRASH_DATE) >= 4)
        AND (extract(hour from C.CRASH_DATE) IN (20,21)) THEN 'dusk'
    WHEN (extract(month from C.CRASH_DATE) <= 8 OR extract(month from
C.CRASH_DATE) >= 4)
        AND (extract(hour from C.CRASH_DATE) <=3 OR extract(hour from
C.CRASH_DATE) >= 22 ) THEN 'night'
    WHEN (extract(month from C.CRASH_DATE) <= 8 OR extract(month from
C.CRASH_DATE) >= 4)
        AND (extract(hour from C.CRASH_DATE) >=7 OR extract(hour from
C.CRASH_DATE) <= 19 ) THEN 'day'

END)
AS time, C.CASE_ID
FROM COLLISION C, ROAD R
WHERE C.ROAD_ID = R.ROAD_ID)
group by time
HAVING time IS NOT NULL
ORDER BY num_collisions DESC;
```

Query result (if the result is big, just a snippet)



	NUM_COLLISIONS	TIME
1	2575152	day
2	1041614	night
3	30317	dawn
4	28092	dusk

We may see that there are more collisions during the day, which is to be expected since people tend to drive more during the day than night even though the visibility might be better.

Query Performance Analysis – Indexing

Remark : Whenever a running-time was reported, the query was run 5 times before reporting the time to ensure proper warmup.

- **QUERY 3: “Find the top-10 vehicle makes based on the number of victims who suffered either a severe injury or were killed. List both the vehicle make and their corresponding number of victims.”**

Initial Running time/IO: 1s 300ms / 29806 IO

Optimized Running time/IO: 700ms / 13784IO

Explain the improvement: We built an index on Victim(victim_degree_of_injury, party_id) and an index on Party(party_id, vehicle_id).

First, the query filters on victim_degree_of_injury IN ('1','2') and only 2.95 % of the Victim's rows qualify.

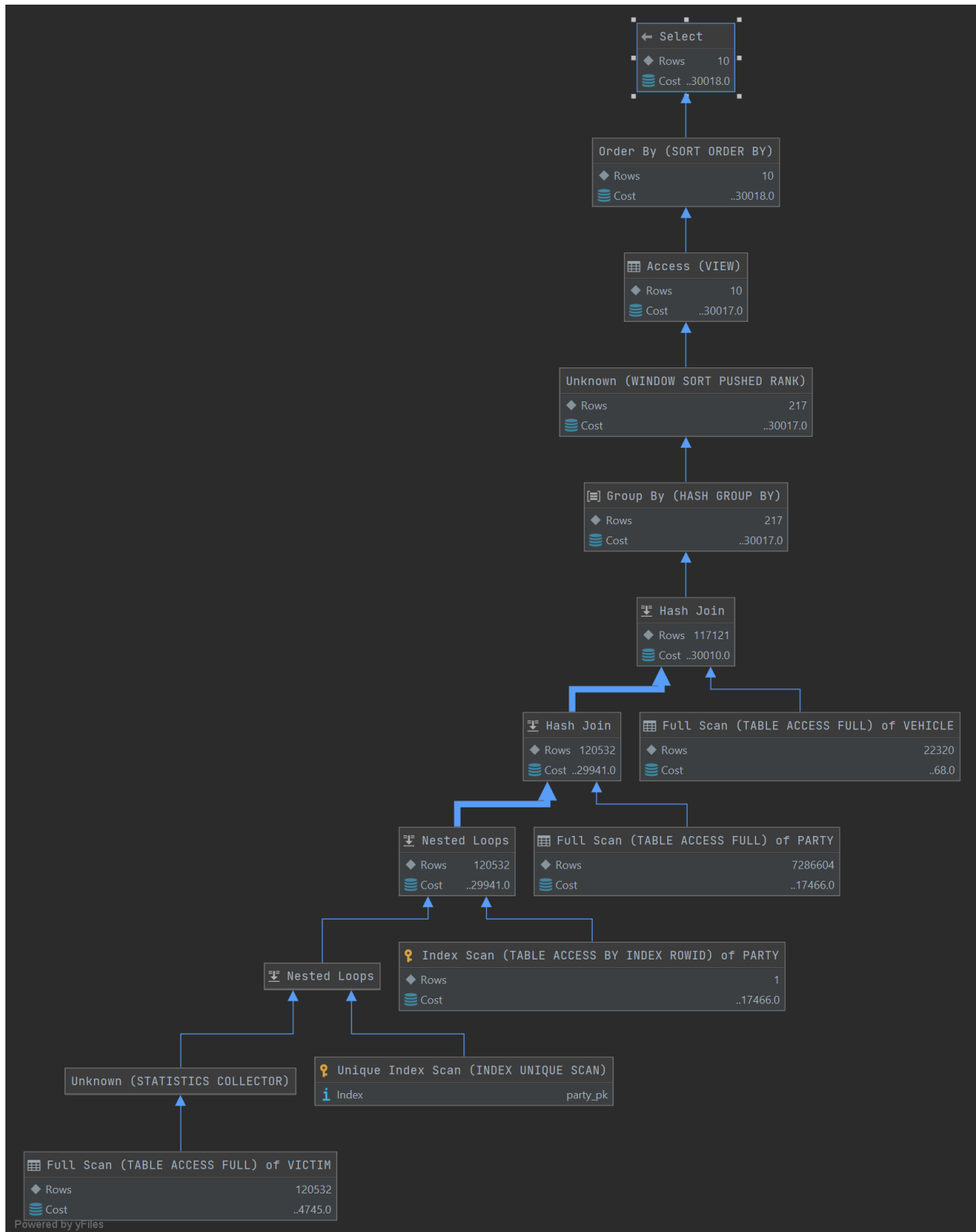
It is expected that using an unclustered index improves performance if the selectivity is < 5%. Hence using an index range scan instead of a full scan lets us improve IO even if the index isn't clustered. Furthermore we can notice that we are joining Victim and Party on party_id and filtering on victim_degree_of_injury but that the aggregate function doesn't use any attribute from the victim table.

Thus, the query can get all the information it needs inside the index `Victim(victim_degree_of_injury, party_id)` and doesn't need to read the `Victim` table

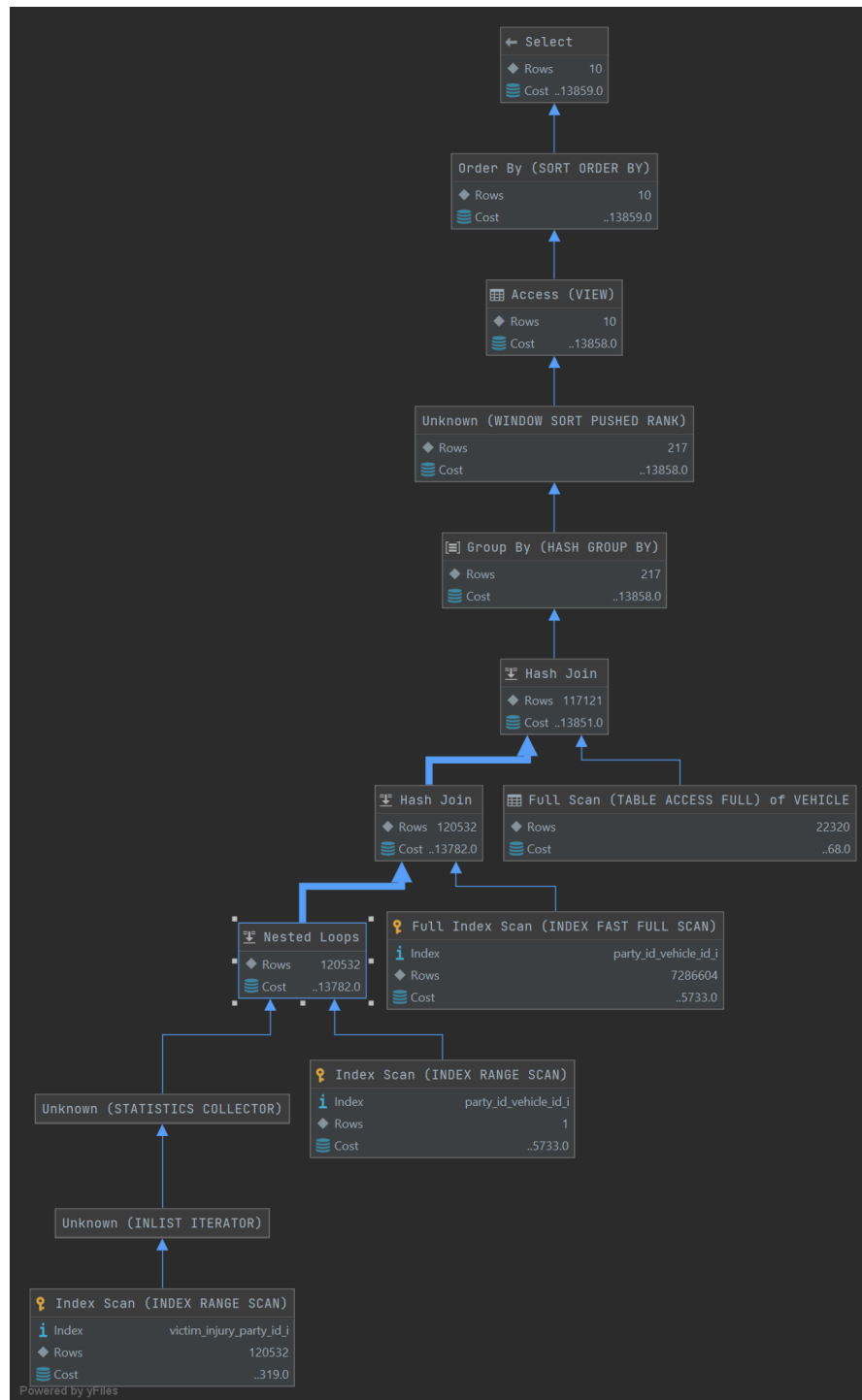
Secondly, the query also joins `Party` and `Vehicle` on `vehicle_id`. Similar to above, the query has all the info it needs inside the index, thus when joining, we can do a fast full index scan instead of a full scan of `Party`.

A fast full index scan is when you go down to the leftmost leaf and then do a sequential scan of the leaves of the B+ tree. This is the fastest way to scan an index.

Initial plan:



Improved plan:



- **QUERY 6: “For each of the top-3 most populated cities, show the city location, population, and the bottom-10 collisions in terms of average victim age (show collision id and average victim age).”**

Initial Running time/IO: 3s 400ms / 37411 IO

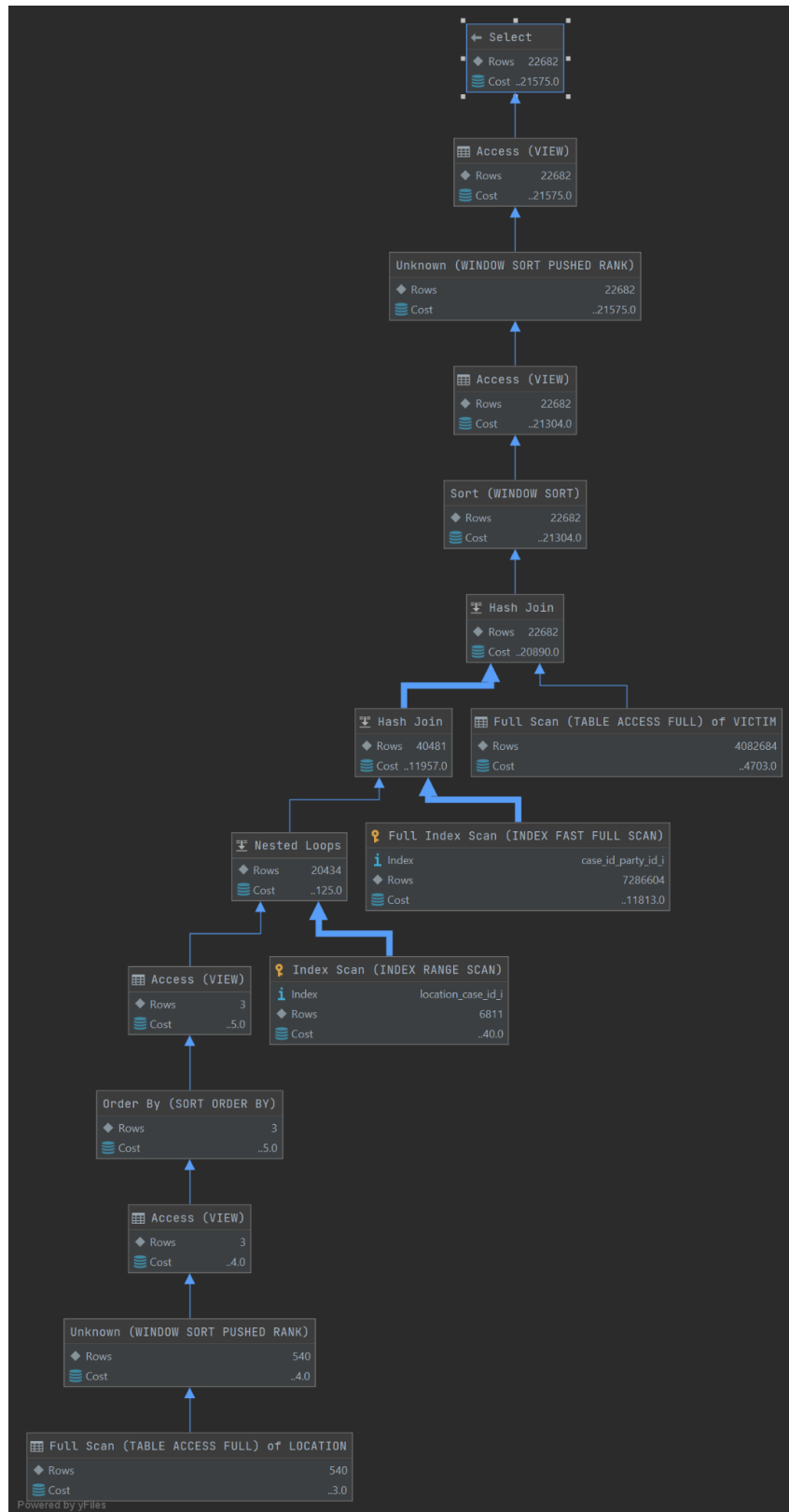
Optimized Running time/IO: 3s 300ms / 21465 IO

Explain the “improvement”: We built an index on Collision(county_city_location, case_id) and Party(case_id,party_id).

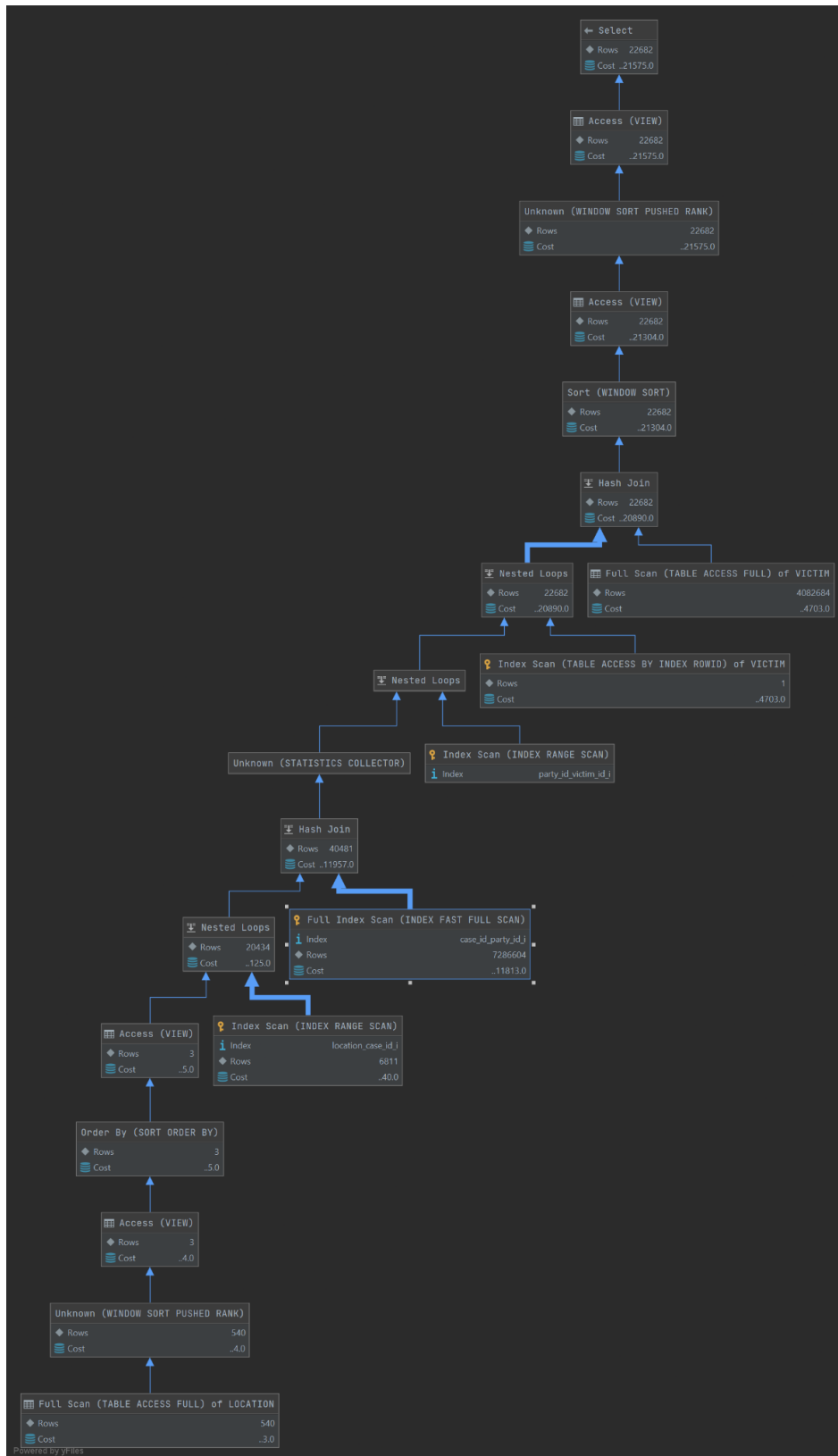
Thanks to the structure of our query, our first join is between the top-3 most populated cities and Collision. Thus, we can do an index range scan on these 3 locations to find all the corresponding case_ids.

Then we join the filtered Collision with Party on case_id . Since we don’t use any attribute of Party and Collision other than case_id in our aggregate functions, we can join using an index fast full scan.

Initial plan:



Improved plan:



- **QUERY 7: “Find all collisions that satisfy the following: the collision was of type pedestrian and all victims were above 100 years old. For each of the qualifying collisions, show the collision id and the age of the eldest collision victim.”**

Initial Running time/IO: 1s 500ms / 52747 IO

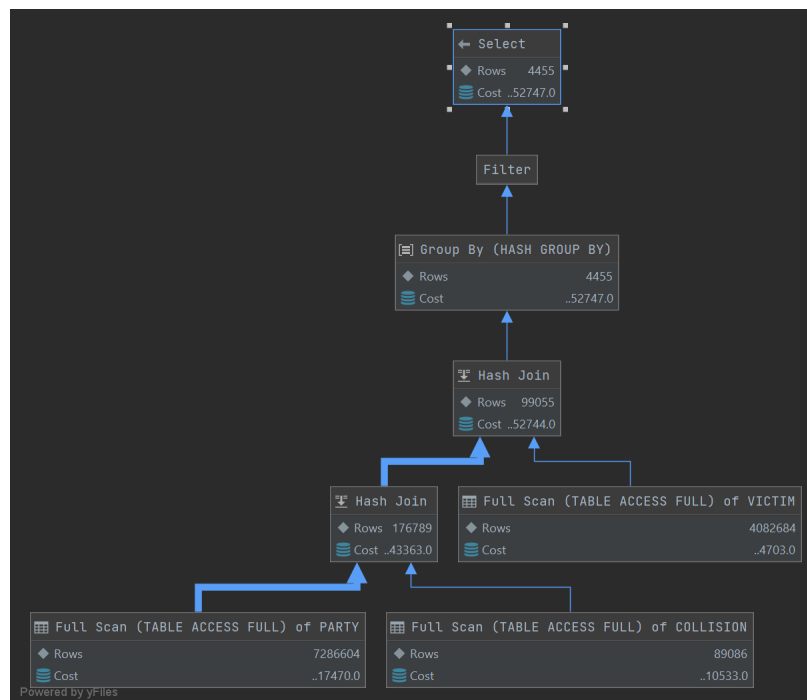
Optimized Running time/IO: 1s 200 ms / 43467 IO

Explain the “improvement”: We built an index on Collision(type_of_collision) and Party(party_id,case_id).

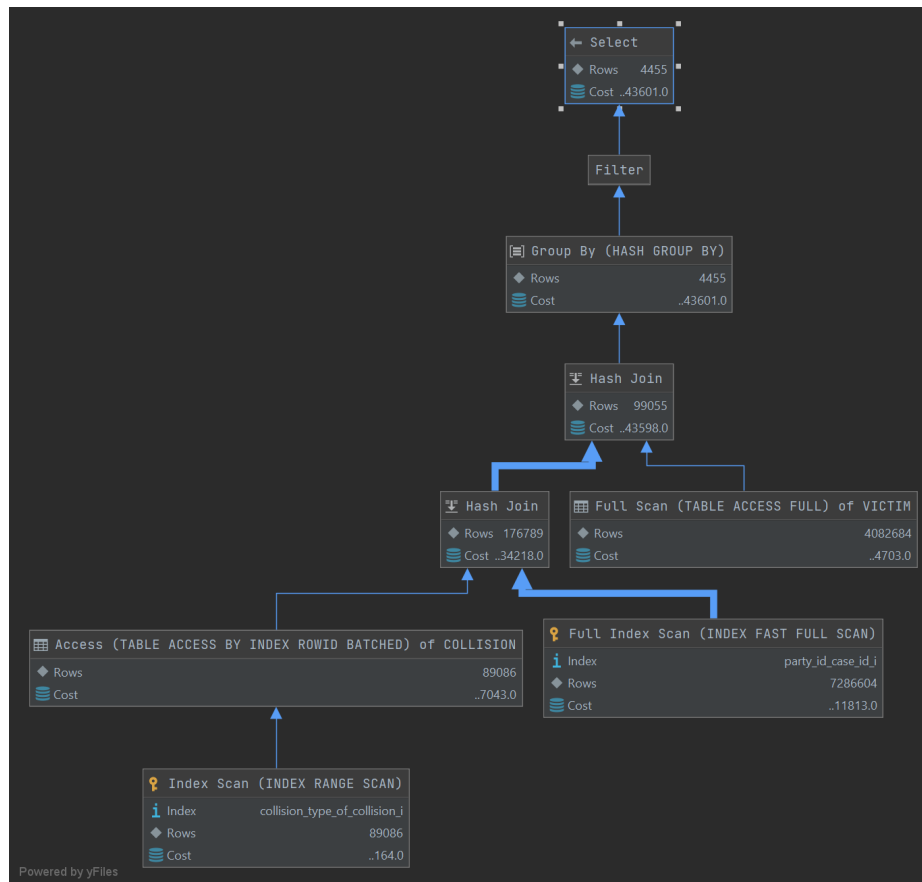
Similar to a previous argument, the query filters on collision of type pedestrian. 2.42 % of the rows of Collision qualify. This is selective enough for an unclustered index to be useful. Thus, we can first filter Collision using an index range scan.

Secondly, because we are joining Collision and Party on case_id and not using anything else from those two tables in our aggregate functions, then the query can get enough information from a fast full index scan instead of a full scan.

Initial plan:



Improved plan:



- **QUERY 8: “Find the vehicles that have participated in at least 10 collisions. Show the vehicle id and number of collisions the vehicle has participated in, sorted according to number of collisions (descending order). What do you observe?”**

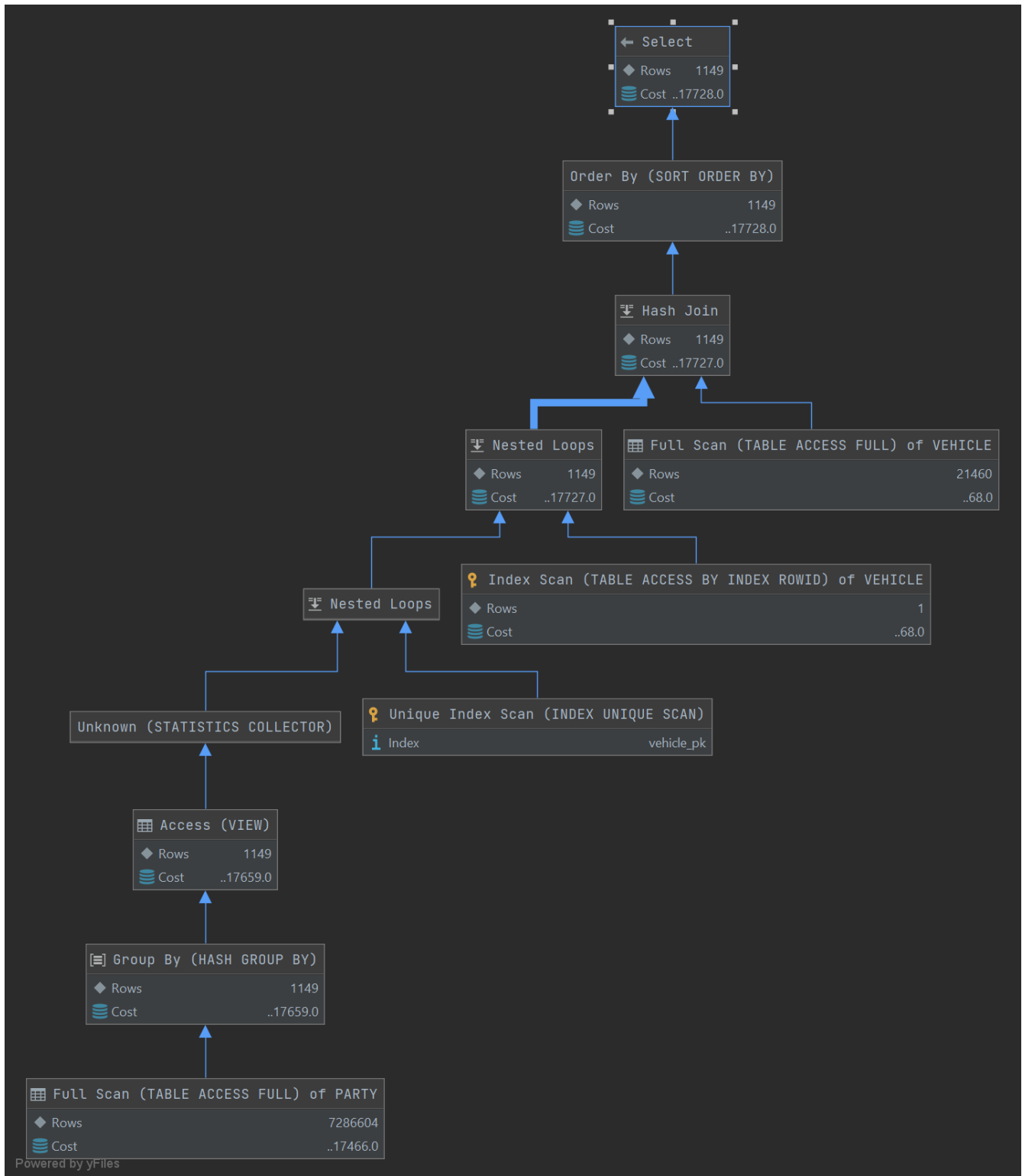
Initial Running time/IO: 1 s 200ms / 17446 IO

Optimized Running time/IO: 1s 200ms / 5775 IO

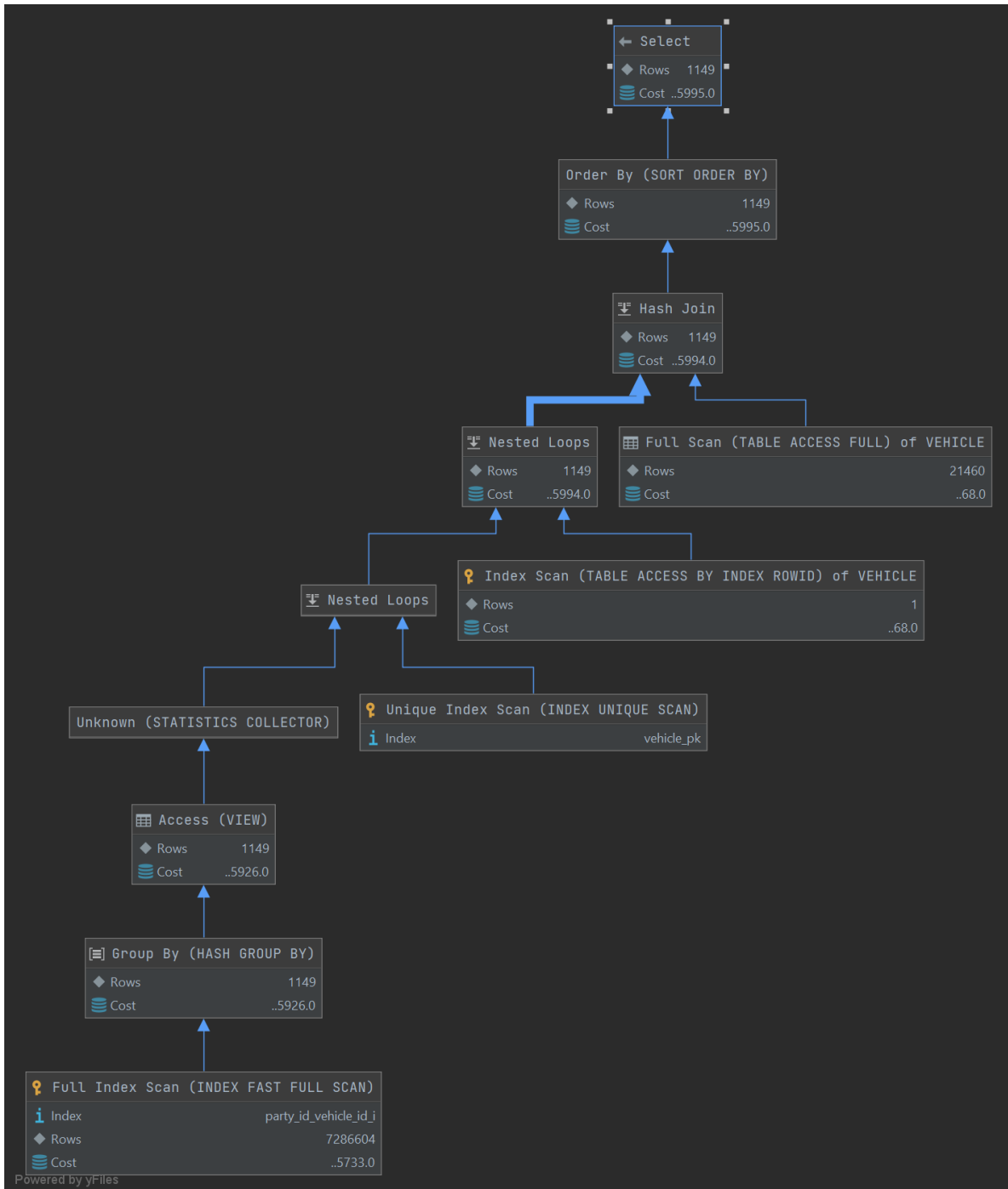
Explain the “improvement”: We built an index on Party(party_id, vehicle_id). The main IO cost in the query comes from grouping Party by vehicle_id. Without an index, a group by operation does a full scan.

But if we build an index on the primary key and the group key, then it suffices to go down to the left of the B+ tree and sequentially read the leaves (index fast full scan). This is possible because our aggregate function only counts the number of rows per vehicle_id.

Initial plan:



Improved plan:



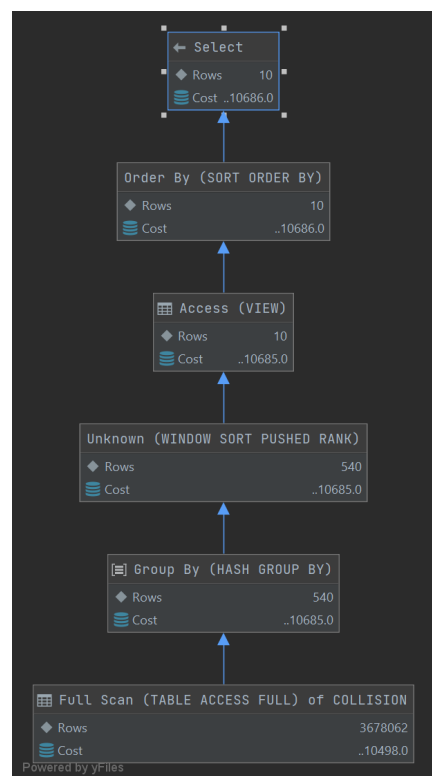
- **QUERY 9: “Find the top-10 (with respect to number of collisions) cities. For each of these cities, show the city location and number of collisions.”**

Initial Running time/IO: 550ms / 10471 IO

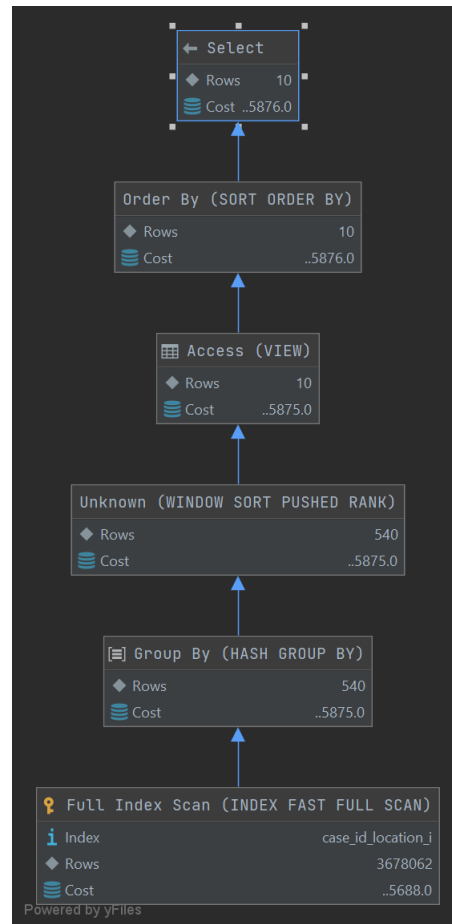
Optimized Running time/IO: 550 ms / 5673IO

Explain the “improvement”: We built an index on Collision(case_id,county_city_location). This query is very simple and is only composed of a group-by operation. Because our aggregate function only counts the number of rows, we can simply create an index on the primary key and group key such that we only need a full index scan instead of a full relation scan.

Initial plan



Improved plan:



General Comments

The project work was done throughout multiple discord sessions where all team members were present, and otherwise tasks were divided equally for each team member.

Also, we thank our TA Georgia for her help and her exceptional availability at any time to answer us as fast and precisely as possible, this was precious all along this project.