Databases Project – Spring 2021

Team No: G34

Members: Siran Li, Ke Wang, Ningwei Ma

Contents

Contents 1

Deliverable 3 2

Changes 2

ER MODEL 2

DDL 3

Query Implementation of Deliverable 2 10

Data processing 16

Deliverable 3 18

Assumptions 18

Query Implementation 18

Query Performance Analysis – Indexing 34

General Comments 45

# Deliverable 3

# Changes

Based on the feedback, we have made some changes including ER model, DDL, data processing and query implementation of Deliverable 2. We have quoted some of the feedback and have responded to it.

## **ER MODEL**

Modifications respective to Feedback on M2 (the modifications are based on the feedback on ER:

*-- ‘Yes, I think that the "condition" entity could be merged with "case". It doesn't give you any specific advantage to keep these two entities as separate.‘*

Removed the 'condition' entity and moved all its attributes to the 'case' entity.

*-- ‘For example, you can create a new entity (and the corresponding table) for "road conditions", with road condition as an attribute and the case\_id acts as a foreign key referred from the collision table. Note that in this case, both the case\_id and the road\_condition serve as the PK.’*

Changed the attributes of 'road\_en' entity into 'case\_id' and 'road\_con', the pairs of them serve as primary keys.

*-- ‘Shouldn't you have a foreign key reference of the primary keys of party into this table? Also, I don't think factors can exist without a party. Lastly, I don't think you need a separate table for the "have" relationship b/w party and other\_fac\_en, it is absorbed as a single table for 'other\_fac\_en'.’*

Changed the attributes of 'other\_fac\_en' entity into 'party\_id' and 'other\_fac', both of them serve as primary keys. Replaced the attribute 've\_num' of 'vehicle' entity by 'party\_id'.

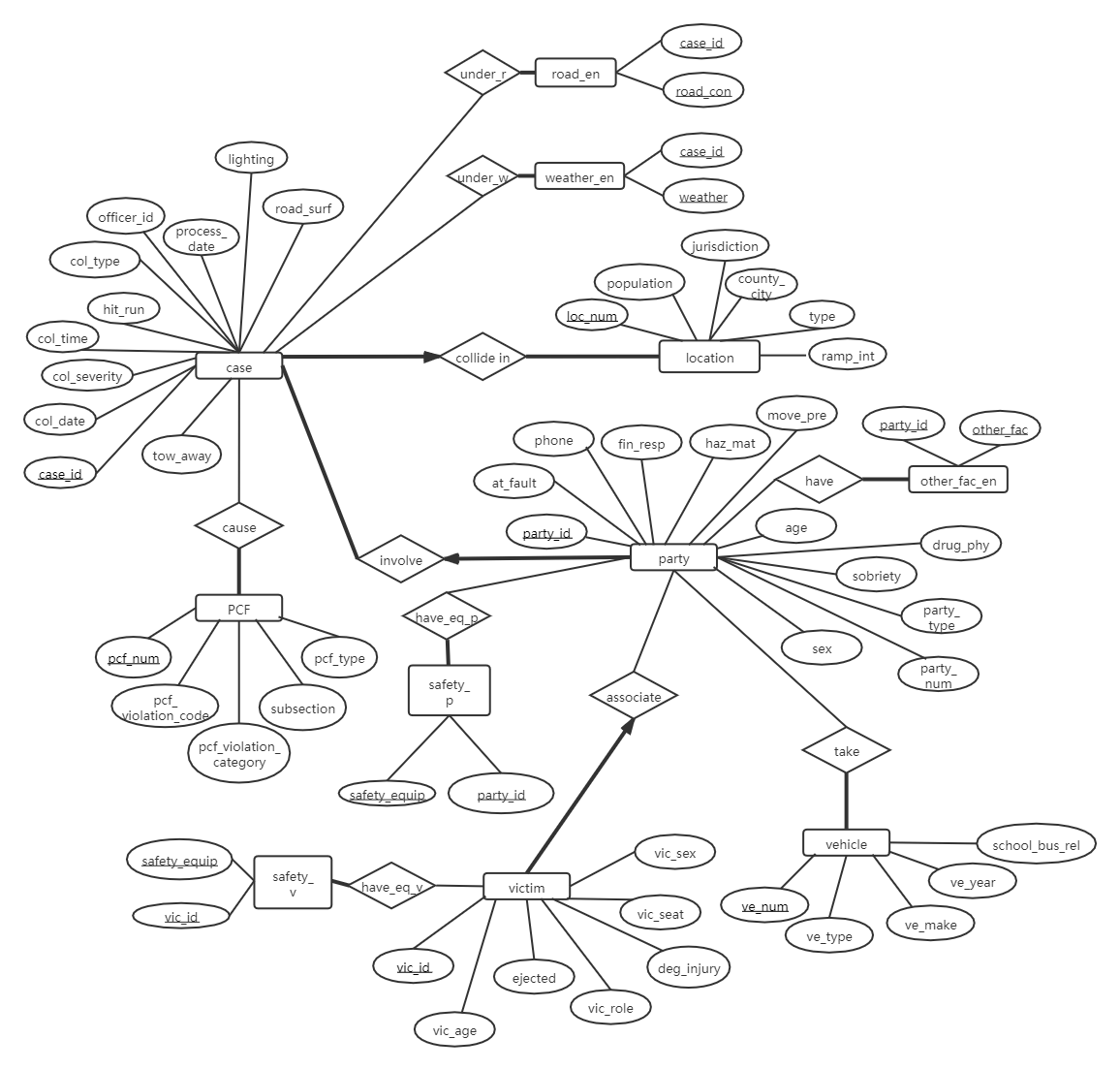
*--’It is indeed correct to have a single "safety\_equipment" entity in the ER, however, during the translation I think you would need two separate tables: safetey\_equipment for party and victim. Similar to the previous point and with a very similar explanation, I don't think you need separate tables for the relationships "have\_ps" and "have\_vs".’*

For this feedback, we decided to separate the ‘other\_fac\_en’ entity into two entities in the ER model as well.

Separated the 'safety\_equip\_en' entity into two entities 'safety\_p' (the safety equipment for party) and 'safety\_v' (the safety equipment for victim').

For 'safety\_p', its attributes are changed to 'safety\_equip' and 'party\_id', both of them serve as primary keys.

For 'safety\_v', its attributes are changed to 'safety\_equip' and 'vic\_id' both of them serve as primary keys.



## **DDL**

In response to deliverable 2's feedback, we have modified the DDL.

*--’ I don't think you need a separate table for the "have" relationship b/w party and other\_fac\_en, have\_ps, have\_vs, under\_r, under\_w, take’*

We deleted the relational tables such as under\_w, under\_r, have, have\_ps, have\_vs, and merged them to the independent entity tables including weather\_en, other\_fac\_en, safety\_equipment, road\_en, respectively.

Now in the relational table, such conditions (weather, road condition, other\_fac,  safety\_p, safety\_v, road\_con) will be like an attribute and the case\_id/party\_id/victim\_id will act as a foreign key referred from the corresponding table. We use the pairs of (xx\_id, xx\_condition) as the primary key to ensure that they are unique.

*--’other\_fac\_en should be added the ‘on delete cascade’ constraint. ’*

In these tables (other\_fac\_en, road\_en, weather\_en, safety\_p, safety\_v), we add “on delete cascade” constraint on the foreign key, because these record cannot exist without party/case/victim. Once the corresponding party/case/victim is deleted from the database, the record related to them would also be deleted.

*--’Also, shouldn't ve\_num be added as a foreign key in the party table? Lastly, similar to the above two points, I don't think you need a separate table for the "take" relationship.’*

Each case happens under exactly one location and PCF and each party drives one kind of vehicle(by the data and our way of data processing). We deleted the relational table ‘take’ between party\_involve-take-vehicle, but just added a column “ve\_num” in the party\_involve table referring to the “ve\_num” of vehicle table. We also deleted the relation ‘cause’ table between PCF-casue-case and added a foreign key “pcf\_num” to the case table referring to the “pcf\_num” of PCF table.

*--’Recall that party can/should not exist without a collision, and victim can/should not exist without a party.’*

We added “on delete cascade” constraint on the “party\_id” foreign key in the associate\_victim table, and the “case\_id” foreign key in the party\_involve table, because party is the weak entity of the case, and the victim if the weak entity of the party (by our assumption and data processing).

--’Since party is a weak entity of case, shouldn't you have a foreign key reference of the primary keys of party into this table?’

We modeled the party a weak entity of the case entity. We put the foreign key reference of the case\_id in the party entity, based on this assumption: party has exactly one participation in this relationship, but one case may be related to many parties. So, we think it might be more reasonable to put the foreign key to case in the party table. And thus we did not put foreign key refering to party in the case table, to avoid a reference loop.

*--’I just want to make sure that the duplicates are not because of the datatype issue (you are modeling case\_id as integer), and are genuine duplicates’*

We changed the type of “case\_id” to VARCHAR2, because there may be several zeros in front of the number, which is also a part of the identity. Now there is no duplicated case\_id. (We did not change the type of party\_id and vict\_id, because they do not have duplicates )

*-- ‘Yes, I think that the "condition" entity could be merged with "case". It doesn't give you any specific advantage to keep these two entities as separate.‘*

We have merged the condition table into the case table, with table condition’s two attributes becoming the attributes of case now.

We have 11 tables now, and the DDL is attached below. Modified parts are marked in yellow.

===========================================================================================

CREATE TABLE PCF(

pcf\_num INTEGER,

pcf\_violation\_code INTEGER,

pcf\_violation\_category VARCHAR2(50),

subsection VARCHAR2(3),

pcf\_type VARCHAR2(50),

PRIMARY KEY (pcf\_num)

);

CREATE TABLE Vehicle(

ve\_num INTEGER,

ve\_type VARCHAR2(50),

ve\_make VARCHAR2(30),

ve\_year INTEGER,

school\_bus\_rel VARCHAR2(5),

PRIMARY KEY (ve\_num)

);

CREATE TABLE Location(

loc\_num INTEGER,

population INTEGER,

county\_city INTEGER,

loc\_type VARCHAR2(20),

ramp\_int VARCHAR2(10),

PRIMARY KEY (loc\_num)

);

CREATE TABLE Case(

case\_id VARCHAR2(30),

loc\_num INTEGER NOT NULL,

pcf\_num INTEGER NOT NULL,

col\_severity VARCHAR2(30),

col\_time DATE,

col\_date DATE,

hit\_run VARCHAR2(30),

jurisdiction INTEGER,

officer\_id VARCHAR2(10),

process\_date DATE,

tow\_away INTEGER,

col\_type VARCHAR2(30),

lighting VARCHAR2(50),

road\_surf VARCHAR2(10),

PRIMARY KEY (case\_id),

FOREIGN KEY (loc\_num) REFERENCES Location(loc\_num),

FOREIGN KEY(pcf\_num) REFERENCES PCF(pcf\_num)

);

CREATE TABLE Party\_involve(

party\_id INTEGER,

case\_id VARCHAR2(30) NOT NULL,

at\_fault INTEGER,

phone VARCHAR2(3),

fin\_resp VARCHAR2(3),

haz\_mat VARCHAR2(3),

move\_pre VARCHAR2(30),

age INTEGER,

drug\_phy VARCHAR2(3),

sobriety VARCHAR2(3),

party\_type VARCHAR2(15),

party\_num INTEGER,

sex VARCHAR2(6),

ve\_num INTEGER,

PRIMARY KEY (party\_id),

FOREIGN KEY (case\_id) REFERENCES Case(case\_id)

ON DELETE CASCADE,

FOREIGN KEY (ve\_num) REFERENCES Vehicle(ve\_num)

);

CREATE TABLE Associate\_victim(

vic\_id INTEGER,

party\_id INTEGER NOT NULL,

vic\_age INTEGER,

ejected INTEGER,

vic\_role INTEGER,

deg\_injury VARCHAR2(50),

vic\_seat INTEGER,

vic\_sex VARCHAR2(6),

PRIMARY KEY (vic\_id),

FOREIGN KEY (party\_id) REFERENCES Party\_involve(party\_id)

ON DELETE CASCADE

);

CREATE TABLE Other\_fac\_en(

party\_id INTEGER,

other\_fac VARCHAR2(3),

PRIMARY KEY (party\_id, other\_fac),

FOREIGN KEY (party\_id) REFERENCES Party\_involve(party\_id)

ON DELETE CASCADE

);

CREATE TABLE road\_en(

case\_id VARCHAR2(30),

road\_con VARCHAR2(20),

PRIMARY KEY (case\_id, road\_con),

FOREIGN KEY (case\_id) REFERENCES Case(case\_id)

ON DELETE CASCADE

);

CREATE TABLE Weather\_en(

case\_id VARCHAR2(30),

weather VARCHAR2(20),

PRIMARY KEY (case\_id, weather),

FOREIGN KEY (case\_id) REFERENCES Case(case\_id)

ON DELETE CASCADE

);

CREATE TABLE safety\_p(

party\_id INTEGER,

safety\_equip VARCHAR2(3),

PRIMARY KEY (party\_id, safety\_equip),

FOREIGN KEY (party\_id) REFERENCES Party\_involve(party\_id)

ON DELETE CASCADE

);

CREATE TABLE safety\_v(

vic\_id INTEGER,

safety\_equip VARCHAR2(3),

PRIMARY KEY (vic\_id, safety\_equip),

FOREIGN KEY (vic\_id) REFERENCES Associate\_victim(vic\_id)

ON DELETE CASCADE);

===========================================================================================

## **Query Implementation of Deliverable 2**

**Query 1:**

***Description of logic:***

List the year-number of collisions per year. We use “group by” to group case by year (extracted from col\_date) and count the number of cases of each year.

***SQL statement***

SELECT EXTRACT (YEAR FROM col\_date) AS YEAR, count(\*) AS N\_collisions

FROM case

GROUP BY EXTRACT (YEAR FROM col\_date)

ORDER BY YEAR ASC

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

|  |  |
| --- | --- |
| YEAR | N\_COLLISIONS |
| 2001 | 522562 |
| 2002 | 544741 |
| 2003 | 538954 |
| 2004 | 538295 |
| 2005 | 532725 |
| 2006 | 498850 |
| 2007 | 501908 |
| 2017 | 7 |
| 2018 | 21 |

**Query 2:**

***Description of logic:***

We first join table “vehicle” and “party\_involve” to know what ve\_make corresponds to a certain ve\_num, and then group it by ve\_make. To illustrate the whole row of the most popular, we fetch first 1 row only.

***SQL statement***

SELECT VE\_MAKE, COUNT(\*) AS N\_COLLISION

FROM (VEHICLE INNER JOIN PARTY\_INVOLVE ON VEHICLE.VE\_NUM = PARTY\_INVOLVE.VE\_NUM)

GROUP BY VE\_MAKE

ORDER BY N\_COLLISION DESC

FETCH FIRST 1 ROWS ONLY

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

|  |  |
| --- | --- |
| VE\_MAKE | N\_VEHICLE |
| FORD | 1129701 |

**Query 3:**

***Description of logic:***

In the lighting attribute of condition, find the description that contains “dark”, and count the fraction of cases that occur in such condition, and keep 2 significant digits..

***SQL statement***

SELECT ROUND(NOM/(SELECT COUNT(\*) FROM CASE),2) as FRACTION

FROM

(SELECT COUNT(\*) AS NOM

FROM CASE

WHERE CASE.LIGHTING LIKE '%dark%')

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

|  |
| --- |
| FRACTION |
| 0.28 |

**Query 4:**

***Description of logic:***

Find the number of collisions that have occurred under snowy weather. We count the number of entries that have weather\_con = ‘snowing’ in the table “weather\_en”

***SQL statement***

SELECT COUNT(\*)

FROM WEATHER\_EN

WHERE WEATHER LIKE '%snowing%'

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

|  |
| --- |
| N\_COLLISIONS |
| 8530 |

**Query 5:**

***Description of logic:***

Group by collisions by which day they are during a week, and count the total number of collisions of that day, then find the row of highest number of cases. We use TO\_CHAR (COL\_DATE, 'D') to extract the day of the week.

***SQL statement***

SELECT TO\_CHAR(COL\_DATE, 'D') AS WEEK\_DAY, COUNT(\*) AS N\_COLLISONS

FROM CASE

GROUP BY TO\_CHAR(COL\_DATE, 'D')

ORDER BY N\_COLLISONS DESC

FETCH FIRST 1 ROWS ONLY

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

|  |  |
| --- | --- |
| WEEK\_DAY | N\_COLLISIONS |
| 6 | 614853 |

**Query 6:**

***Description of logic:***

List all weather types and their corresponding number of collisions in ascending order of the collisions.We group cases by “weather” and list “weather” and the count number.

***SQL statement***

SELECT WEATHER, COUNT(\*) AS N\_COLLISION

FROM WEATHER\_EN

GROUP BY WEATHER

ORDER BY N\_COLLISION

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

|  |  |
| --- | --- |
| WEATHER | N\_COLLISION |
| other | 6960 |
| snowing | 8530 |
| wind | 13952 |
| fog | 21259 |
| raining | 223752 |
| cloudy | 548250 |
| clear | 2941042 |

**Query 7:**

***Description of logic:***

Count the number of parties that are at-fault, with financial responsibility and loose material. We first extract the “road\_num” of “road\_loose”, and find which parties are associated with such road condition. We filter the “party\_id” table who is at fault and with financial responsibility. Finally we count the number of the selected parties.

***SQL statement***

SELECT COUNT(\*) AS N\_PARTIES

FROM PARTY\_INVOLVE P, ROAD\_EN R

WHERE P.CASE\_ID = R.CASE\_ID AND P.AT\_FAULT = 1 AND P.FIN\_RESP = 'Y' AND R.ROAD\_CON LIKE '%loose material%'

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

|  |
| --- |
| N\_PARTIES |
| 4803 |

**Query 8:**

***Description of logic:***

Find the median victim age: we directly use the “MEDIAN” function of SQL from the associate\_victim table.

Find the most common victim seating position. We group the victims with seating position, and count the number of victims of each vic\_seat, order them in the descending order of this number and find the max.

***SQL statement***

***8.a***

SELECT MEDIAN(vic\_age) AS MEDIAN\_VIC\_AGE

FROM ASSOCIATE\_VICTIM v2;

***8.b***

SELECT VIC\_SEAT AS MOST\_COMMON\_SEAT\_POSITION

FROM

(SELECT COUNT(vic\_seat) AS count, vic\_seat

FROM associate\_victim v2

GROUP BY vic\_seat

ORDER BY count DESC)

FETCH FIRST 1 ROWS ONLY;

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

***8.a***

|  |
| --- |
| MEDIAN\_VIC\_AGE |
| 25 |

***8.b***

|  |
| --- |
| MOST\_COMMON\_SEAT\_POSITION |
| 3 |

**Query 9:**

***Description of logic:***

Fraction of all participants (victims + parties) that have been victims using a belt. All participants refer to both parties and victims, so our denominator is the sum of number of all victims and parties. We first extract the ‘vic\_id’s who use belt using table safety\_v. Then we count the unique ‘vic\_id’s and use this number as the numerator. Finally we get the fraction and keep 2 significant digits.

***SQL statement***

SELECT ROUND(A.FRACTION,3) AS fraction

FROM(SELECT DISTINCT

(SELECT COUNT(vic\_id) AS count

FROM

(SELECT h1.vic\_id as vic\_id

FROM SAFETY\_V H1

WHERE H1.SAFETY\_EQUIP like '%C%') v\_belt)/

((SELECT COUNT(party\_id) FROM party\_involve)

+(SELECT COUNT(vic\_id) FROM associate\_victim)) as fraction

FROM party\_involve) a

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

|  |
| --- |
| FRACTION |
| 0.011 |

**Query 10:**

***Description of logic:***

Compute the fraction of collisions happening for each hour of the day, and display the ratio as percentage for all the hours of the day. We first use cast(col\_time as timestamp) to extract the hour in which the case occurred. Then we group the cases by the specific hour and count the number of the cases, then order them by the number. We also directly calculate the count of total cases. Then we divide the count number of each hour by the total number to get each fraction.

***SQL statement***

SELECT DISTINCT

EXTRACT(hour from cast(col\_time as timestamp)) as hour, CONCAT( ROUND((COUNT(\*)/(SELECT COUNT(\*) FROM CASE)\*100.0),2),'%') as FRACTION

FROM CASE

GROUP BY EXTRACT(hour from cast(col\_time as timestamp))

ORDER BY hour ASC

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

|  |  |
| --- | --- |
| *HOUR* | *FRACTION* |
| *0* | *1.91%* |
| *1* | *1.83%* |
| *2* | *1.81%* |
| *3* | *1.15%* |
| *4* | *0.98%* |
| *5* | *1.45%* |
| *6* | *2.62%* |
| *7* | *5.17%* |
| *8* | *5.23%* |
| *9* | *4.09%* |
| *10* | *4.23%* |
| *11* | *4.89%* |
| *12* | *5.78%* |
| *13* | *5.78%* |
| *14* | *6.55%* |
| *15* | *7.75%* |
| *16* | *7.33%* |
| *17* | *7.91%* |
| *18* | *6.30%* |
| *19* | *4.43%* |
| *20* | *3.49%* |
| *21* | *3.28%* |
| *22* | *2.86%* |
| *23* | *2.38%* |
| *(null)* | *0.81%* |

## **Data processing**

We made modifications based on the following feedbacks:

*-- ‘How are you generated identifiers: e.g. pcf\_num, loc\_num, con\_num, etc.?‘*

For identifiers of pcf\_num, ve\_num and loc\_num (now con\_num has been deleted), we used Pandas to generate indexes on the dataframes (reset\_index(drop=True)). Thus the indexes are all natural numbers and unique. We generated these indexes to serve as keys for respective entities.

*-- ‘How are you handling attributes with missing values, if any? Especially, the attributes that are important and shouldn't/can't be simply ignored.’*

Case\_id, party\_id and victim\_id do not and should not have null values.

For road\_en, we first extracted all distinct road\_en values and dropped the null value, then joined it with the 2 columns of road condition in the *collisions2018.csv* to get the not null (case\_id, road\_con) pairs, and dropped the duplicates. In this way if a collision’s road condition is unavailable, it will not appear in the road\_en table, and all the pairs are unique. The same method also works for weather\_en, other\_fac\_en, safety\_p and safety\_v.

For vehicle, we first extracted all the distinct tuples of their attributes, and we did not drop the null values, so there will not be null value in the party\_involve table for ve\_num, but there is a row of “ve\_num | null | null | null | null” is the vehicle table. It is the same for location and PCF table.

# Deliverable 3

## Assumptions

Based on our assumptions on M1 and M2, we made the following further assumptions.

*-- ‘I just want to make sure that the duplicates are not because of the datatype issue (you are modeling case\_id as integer), and are genuine duplicates. Please reanalyze this part.’*

As stated in previous parts, we changed the type of case\_id into string now, which fixed this problem. The duplicates are indeed caused by the datatype issue, and now the problem has been fixed.

*-- (Feedback on M1) ‘"Party\_number refers to the specific party of a particular case, so party\_number + case\_id is unique for each party, playing the same role as party\_id." \* Have you verified this from the data?’*

Furthermore, after fixing this problem, it supports our earlier assumption “case\_id+party\_num” uniquely identifies ‘party\_id’.

## Query Implementation

**Query a:**

***Description of logic:***

We first select parties whose “party\_type” is ‘driver’ and the “age” is not null. Then we just project columns of “age” and “at\_fault” and classify them into different age groups. If the party is at fault, the value of “at\_fault” is 1, otherwise it is 0 (if it’s not null), so we can just calculate the sum of “at\_fault” of each age group, and then divide the sum by the count of each age group to get the ratio

***SQL statement***

SELECT

(CASE WHEN AGE<=18 THEN 'underage'

WHEN AGE BETWEEN 19 and 21 then 'young I'

when AGE BETWEEN 22 AND 24 THEN 'young II'

when AGE BETWEEN 24 AND 60 THEN 'adult'

when AGE BETWEEN 61 AND 64 THEN 'elder I'

else 'elder II' END) as AGE\_RANGE,

CONCAT(100\*ROUND(SUM(AT\_FAULT)/COUNT (AT\_FAULT), 3), '%') as ratio

FROM (SELECT AGE, AT\_FAULT FROM PARTY\_INVOLVE WHERE PARTY\_TYPE like '%driver%' and AGE IS NOT NULL and AT\_FAULT IS NOT NULL)

group by (CASE WHEN AGE<=18 THEN 'underage'

WHEN AGE BETWEEN 19 and 21 then 'young I'

when AGE BETWEEN 22 AND 24 THEN 'young II'

when AGE BETWEEN 24 AND 60 THEN 'adult'

when AGE BETWEEN 61 AND 64 THEN 'elder I'

ELSE 'elder II' END)

order by RATIO DESC

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

|  |  |
| --- | --- |
| AGE\_RANGE | RATIO |
| underage | 64.70% |
| young I | 58% |
| young II | 51.90% |
| elder II | 50.50% |
| adult | 41.00% |
| elder I | 40.10% |

We found the underage, young I and young II group has the 3 highest at\_fault ratio. If we were an insurance company, I will raise premiums for young people under 24 years old and the elder over 65 years old.

**Query b:**

***Description of logic:***

We join the party\_involve table with vehicle on ve\_num to get the vehicle details for each party, then we join it with road\_en (whose road\_con contains “holes”) on case\_id to get the road\_condition of the case that the party is involved. If in a case there are vehicles with the same type, we count 1 for how many collisions this vehicle\_type participates. Finally we group the cases based on the ve\_type and calculate the total number of cases of each group, order them by the count and fetch the top 5 type.

***SQL statement***

SELECT VE\_TYPE, COUNT(\*) AS COUNT

FROM

    (SELECT DISTINCT VEHICLE.VE\_TYPE, PARTY\_INVOLVE.CASE\_ID

    FROM PARTY\_INVOLVE, VEHICLE, ROAD\_EN

    WHERE ROAD\_EN.ROAD\_CON LIKE '%holes%'

    AND VE\_TYPE IS NOT NULL

    AND ROAD\_EN.CASE\_ID = PARTY\_INVOLVE.CASE\_ID

    AND VEHICLE.VE\_NUM = PARTY\_INVOLVE.VE\_NUM)

GROUP BY VE\_TYPE

ORDER BY COUNT DESC

FETCH FIRST 5 ROWS ONLY

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

|  |  |
| --- | --- |
| VE\_TYPE | COUNT |
| passenger car | 6940 |
| pickup or panel truck | 2017 |
| motorcycle or scooter | 432 |
| bicycle | 417 |
| truck or truck tractor with trailer | 351 |

**Query c:**

***Description of logic:***

We first select the victims whose deg\_injury is like “severe injury” or “killed”, and then join it with the table party\_involve on the party\_id and then table vehicle (select those whose ve\_make is not null) on ve\_num to get [vic\_id|ve\_make]. We group the table by ve\_make and count the number of victims of each ve\_make. Finally we order the table by the count and fetch the top-10 vehicle makes and their number of victims.

***SQL statement***

SELECT VE\_MAKE, COUNT(ASSOCIATE\_VICTIM.VIC\_ID) AS COUNT

FROM VEHICLE, PARTY\_INVOLVE, ASSOCIATE\_VICTIM

WHERE VEHICLE.VE\_NUM = PARTY\_INVOLVE.VE\_NUM

AND ASSOCIATE\_VICTIM.PARTY\_ID = PARTY\_INVOLVE.PARTY\_ID

AND (ASSOCIATE\_VICTIM.DEG\_INJURY LIKE '%severe injury%' OR ASSOCIATE\_VICTIM.DEG\_INJURY LIKE '%killed%')

AND VEHICLE.VE\_MAKE IS NOT NULL

GROUP BY VE\_MAKE

ORDER BY COUNT DESC

FETCH FIRST 10 ROWS ONLY

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

|  |  |
| --- | --- |
| VE\_MAKE | COUNT |
| FORD | 13924 |
| HONDA | 12060 |
| TOYOTA | 10639 |
| CHEVROLET | 10418 |
| NISSAN | 3860 |
| DODGE | 3641 |
| NOT STATED | 3603 |
| HARLEY-DAVIDSON | 3410 |
| SUZUKI | 2482 |
| YAMAHA | 2105 |

**Query d:**

***Description of logic:***

We assume that the fraction is calculated by: ( # victims with no injury and of one seat place) / (# victims of this seat place). We don’t consider the number of collisions because if there are 2 victims seating in same/different position in one collision, it is ambiguous to count for 1 or 2 collisions, and the collision can be of several seating positions.

We partition the table by vic\_seat, and add a column for count of victims of one kind of vic\_seat. Then we select the tuples whose deg\_injury is like “no injury”, and calculate the sum of 1/count of each vic\_seat group. The denominator is the total count and the nominator is the count of the no injured, so we get the safety\_index of each vic\_seat. To get the max and min safety\_index and their corresponding seat, we union two tables, one is for the min and one is for the max.

***SQL statement***

(SELECT vic\_seat, ROUND(SUM(1.0/denom),2) AS SAFETY\_INDEX

FROM

    (SELECT VIC\_SEAT, DEG\_INJURY, COUNT(DEG\_INJURY) OVER (PARTITION BY VIC\_SEAT) AS DENOM

    FROM ASSOCIATE\_VICTIM)

WHERE DEG\_INJURY LIKE '%no injury%'

GROUP BY VIC\_SEAT

ORDER BY SAFETY\_INDEX DESC

FETCH FIRST 1 ROWS ONLY)

UNION ALL

(SELECT vic\_seat, ROUND(SUM(1.0/denom),2) AS SAFETY\_INDEX

FROM

    (SELECT VIC\_SEAT, DEG\_INJURY, COUNT(DEG\_INJURY) OVER (PARTITION BY VIC\_SEAT) AS DENOM

    FROM ASSOCIATE\_VICTIM)

WHERE DEG\_INJURY LIKE '%no injury%'

GROUP BY VIC\_SEAT

ORDER BY SAFETY\_INDEX

FETCH FIRST 1 ROWS ONLY)

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

|  |  |
| --- | --- |
| VIC\_SEAT | SAFETY\_\_INDEX |
| 5 | 0.839 |
| 1 | 0.009 |

**Query e:**

***Description of logic:***

We assume that the query is to find ve\_types that participate in at least 10 collisions of EACH city and in at least half of the cities. We join the vehicle and party to know what type of the vehicle every party is driving, and then join it with table case and location to know in which city the collisions happen. There may be two same ve\_type in the same collision, and we count 1 for the #collision of this ve\_type, so for each [ve\_type, county\_city], we calculate the total number of collisions as CASE\_NUM. After selecting [ve\_type, county\_city] tuples where the CASE\_NUM >= 10, we group the table by ve\_type and select the number of those whose count(county\_city) >= total number of cities / 2

***SQL statement***

SELECT COUNT(VE\_TYPE) AS NUM\_VE\_TYPE

FROM

    (SELECT VE\_TYPE, COUNT(COUNTY\_CITY) AS CITY\_NUM

    FROM

        (SELECT DISTINCT VE\_TYPE, COUNTY\_CITY, COUNT(\*) AS CASE\_NUM

        FROM VEHICLE, PARTY\_INVOLVE, CASE, LOCATION

        WHERE VEHICLE.VE\_NUM=PARTY\_INVOLVE.VE\_NUM

            AND PARTY\_INVOLVE.CASE\_ID=CASE.CASE\_ID

            AND LOCATION.LOC\_NUM=CASE.LOC\_NUM

            GROUP BY VE\_TYPE, COUNTY\_CITY)CITY

    WHERE CASE\_NUM >=10

    GROUP BY VE\_TYPE) COUNT\_CITY,

    (SELECT COUNT(DISTINCT COUNTY\_CITY) AS TOTAL\_CITY FROM LOCATION) TOTAL

WHERE CITY\_NUM>= TOTAL\_CITY/2

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

|  |
| --- |
| ***NUM\_VE\_TYPE*** |
| ***13*** |

**Query f:**

***Description of logic:***

To find the most populated cities, we use the order by function defined by ourselves to order the population level of cities descendingly, and fetch the first 3 distinct county\_city and the corresponding population. In our model, each case has a loc\_num, but several location entries can have the same county\_city, so we need to join the 3-row table with the location and the case table to get [county\_city|population|case\_id], and then join it with party\_involve and associate\_victim table to get [county\_city|population|case\_id|vic\_d|vic\_age]. We group the table over case\_id and add a column of average victim age of each case. Then we partition the table over county\_city, and add a row\_number() column in the order of victim age, and select the youngest 10.

***SQL statement***

SELECT COUNTY\_CITY, POPULATION, CASE\_ID, AVG\_AGE

FROM

(SELECT COUNTY\_CITY, POPULATION, CASE\_ID, AVG\_AGE, ROW\_NUMBER() OVER(PARTITION BY COUNTY\_CITY ORDER BY AVG\_AGE ASC) AS ROW\_NUMBER

FROM

(SELECT COL\_CITY.COUNTY\_CITY, COL\_CITY.POPULATION, CASE.CASE\_ID, AVG(VIC\_AGE) AS AVG\_AGE

FROM

(SELECT DISTINCT COUNTY\_CITY, POPULATION

FROM LOCATION

order by (case population

when 7 then 0

when 6 then 1

when 5 then 2

when 4 then 3

when 3 then 4

when 2 then 5

when 1 then 6

when 9 then 7

when 0 then 8

else 9 end)

FETCH FIRST 3 ROWS ONLY) COL\_CITY,

LOCATION, CASE, PARTY\_INVOLVE, ASSOCIATE\_VICTIM

WHERE COL\_CITY.county\_city=LOCATION.county\_city

AND CASE.LOC\_NUM=LOCATION.LOC\_NUM

AND PARTY\_INVOLVE.CASE\_ID=CASE.CASE\_ID

AND ASSOCIATE\_VICTIM.PARTY\_ID=PARTY\_INVOLVE.PARTY\_ID

AND VIC\_AGE IS NOT NULL

GROUP BY COL\_CITY.COUNTY\_CITY, COL\_CITY.POPULATION, CASE.CASE\_ID)AGE)RANK\_AGE

WHERE ROW\_NUMBER<=10

ORDER BY COUNTY\_CITY, ROW\_NUMBER

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

|  |  |  |  |
| --- | --- | --- | --- |
| COUNTY\_CITY | POPULATION | CASE\_ID | AVG\_AGE |
| 109 | 7 | 1050440 | 0 |
| 109 | 7 | 2858865 | 0 |
| 109 | 7 | 2489862 | 0 |
| 109 | 7 | 1572294 | 0 |
| 109 | 7 | 739484 | 0 |
| 109 | 7 | 491424 | 0 |
| 109 | 7 | 3553733 | 0 |
| 109 | 7 | 3486455 | 0 |
| 109 | 7 | 3012670 | 0 |
| 109 | 7 | 2856537 | 0 |
| 3801 | 7 | 984042 | 0 |
| 3801 | 7 | 2945655 | 0 |
| 3801 | 7 | 2613850 | 0 |
| 3801 | 7 | 1686308 | 0 |
| 3801 | 7 | 1783897 | 0 |
| 3801 | 7 | 353302 | 0 |
| 3801 | 7 | 2903949 | 0 |
| 3801 | 7 | 1895167 | 0 |
| 3801 | 7 | 1490755 | 0 |
| 3801 | 7 | 994429 | 0 |
| 4313 | 7 | 1702989 | 0 |
| 4313 | 7 | 2530732 | 0 |
| 4313 | 7 | 2427381 | 0 |
| 4313 | 7 | 2034278 | 0 |
| 4313 | 7 | 1585055 | 0 |
| 4313 | 7 | 3391590 | 0 |
| 4313 | 7 | 2034270 | 0 |
| 4313 | 7 | 1506715 | 0 |
| 4313 | 7 | 1322740 | 0 |
| 4313 | 7 | 1239869 | 0 |

**Query g:**

***Description of logic:***

We first select case with col\_type like “pedestrian” to get the “PE\_CASE” table, then we join PE\_CASE with party\_involve and associate\_victim table to get the corresponding victims of the case. We select the tuples that satisfies the min(vic\_age) > 100 and find the eldest victim age of each collision.

***SQL statement***

SELECT DISTINCT CASE\_ID, MAX(VIC\_AGE) OVER(PARTITION BY CASE\_ID) AS ELDEST

FROM

    (SELECT PE\_CASE.CASE\_ID, VIC\_AGE, MIN(VIC\_AGE) OVER (PARTITION BY PE\_CASE.CASE\_ID)AS MIN\_AGE

    FROM

        (SELECT CASE\_ID

        FROM CASE

        WHERE COL\_TYPE LIKE '%pedestrian%') PE\_CASE, PARTY\_INVOLVE, ASSOCIATE\_VICTIM

    WHERE PARTY\_INVOLVE.CASE\_ID=PE\_CASE.CASE\_ID

        AND ASSOCIATE\_VICTIM.PARTY\_ID=PARTY\_INVOLVE.PARTY\_ID)CASE\_AGE

WHERE MIN\_AGE>100

ORDER BY CASE\_ID

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

|  |  |
| --- | --- |
| CASE\_ID | ELDEST |
| 36446 | 110 |
| 69198 | 101 |
| 439197 | 102 |
| 445265 | 101 |
| 566220 | 102 |
| 644226 | 103 |
| 784061 | 102 |
| 817210 | 102 |
| 820619 | 101 |
| 828116 | 102 |
| 851026 | 106 |
| 868472 | 103 |
| 1209166 | 101 |
| 1213340 | 121 |
| 1347636 | 101 |
| 1373664 | 101 |
| 1548445 | 102 |
| 1847678 | 104 |
| 2472739 | 103 |
| 2531557 | 103 |

**Query h:**

***Description of logic:***

We first group the party\_involve table by the ve\_num to get [ve\_num | N\_COLLISION]  table VE\_COL, then we join the VE\_COL with the vehicle table on the ve\_num to get how many collisions a kind of vehicle has participated in. We select those who has participated in more than 9 collisions. Because we assign each vehicle with same [ve\_type, ve\_make, ve\_year] a unique ve\_num, so we finally use [ve\_type, ve\_make, ve\_year] to denote the vehicle id.

***SQL statement***

WITH

VE\_COL AS

    (

    SELECT VE\_NUM, COUNT(case\_id) AS N\_COLLISION

    FROM PARTY\_INVOLVE

    GROUP BY VE\_NUM

    )

SELECT VEHICLE.VE\_TYPE || ', ' || VEHICLE.VE\_MAKE || ', ' || VEHICLE.VE\_YEAR AS VE\_ID, VE\_COL.N\_COLLISION AS N\_COLLISION

FROM (VEHICLE INNER JOIN VE\_COL ON VE\_COL.VE\_NUM = VEHICLE.VE\_NUM)

WHERE N\_COLLISION > 9 AND VEHICLE.VE\_MAKE IS NOT NULL AND VEHICLE.VE\_TYPE IS NOT NULL AND VEHICLE.VE\_TYPE != 'pedestrain'

ORDER BY N\_COLLISION DESC

FETCH FIRST 20 ROWS ONLY

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

|  |  |
| --- | --- |
| VE\_ID | N\_COLLISION |
| passenger car, TOYOTA, 2000 | 52469 |
| passenger car, FORD, 2000 | 51915 |
| passenger car, HONDA, 2000 | 50255 |
| passenger car, FORD, 1998 | 49148 |
| passenger car, TOYOTA, 2001 | 47205 |
| passenger car, HONDA, 2001 | 45243 |
| passenger car, FORD, 2001 | 45203 |
| passenger car, TOYOTA, 1999 | 42907 |
| passenger car, HONDA, 1998 | 42063 |
| passenger car, FORD, 1999 | 41918 |
| passenger car, FORD, 1995 | 40225 |
| passenger car, HONDA, 1997 | 39182 |
| passenger car, FORD, 1997 | 38852 |
| passenger car, HONDA, 1999 | 38528 |
| passenger car, TOYOTA, 2002 | 38403 |
| passenger car, TOYOTA, 1998 | 37982 |
| passenger car, TOYOTA, 1997 | 37134 |
| passenger car, TOYOTA, 2003 | 35919 |
| passenger car, HONDA, 2002 | 35759 |
| passenger car, FORD, 2002 | 35433 |

Observation:

The vehicles participated in most collisions are all passenger cars. Besides, these vehicles are all made by large car manufactures such as Totota, Ford and Honda.

**Query i:**

***Description of logic:***

Because we assign each kind of location a loc\_num but several loc\_num have the same county\_city, so we first join location and case, then group the table by “county\_city” to count how many collisions happen in the same county\_city (if it is not null). Then we order them by number of collisions and fetch the top-10 county\_city.

***SQL statement***

SELECT COUNTY\_CITY, COUNT(CASE\_ID) as N\_COLLISION FROM

LOCATION, CASE

WHERE LOCATION.LOC\_NUM = CASE.LOC\_NUM and LOCATION.COUNTY\_CITY IS NOT NULL

GROUP BY COUNTY\_CITY

ORDER BY N\_COLLISION DESC

FETCH FIRST 10 ROWS ONLY

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

|  |  |
| --- | --- |
| COUNTY\_CITY | N\_COLLISION |
| 1942 | 399582 |
| 1900 | 118446 |
| 3400 | 80191 |
| 3711 | 76867 |
| 109 | 72995 |
| 3300 | 61453 |
| 3404 | 58068 |
| 4313 | 57852 |
| 1941 | 53565 |
| 3801 | 48450 |

**Query j:**

***Description of logic:***

We focus on the “group by” function. First we group by lighting of the case, there are “DAY”, ”NIGHT” and others, in the dataset, lighting conditions of  “dawn and dusk” are ambiguous, so we just distinguish them by the col\_date and col\_time, neglecting the value. If the lighting is not “DAY” or “NIGHT”, we group them by the date and time, based on different seasons. After grouping, we count the number of collisions of each lighting condition, and order them by the count.

***SQL statement***

WITH

COL\_PERIOD AS

    (

    SELECT

    CASE

    WHEN CASE.LIGHTING LIKE '%day%' THEN 'DAY'

    WHEN CASE.LIGHTING LIKE '%dark%' THEN 'NIGHT'

    ELSE

            (CASE

                WHEN EXTRACT(MONTH FROM COL\_TIME)>8 OR EXTRACT(MONTH FROM COL\_TIME)<4 THEN

                    (CASE

                    WHEN EXTRACT(HOUR FROM CAST(COL\_TIME AS TIMESTAMP))>5 AND EXTRACT(HOUR FROM CAST(COL\_TIME AS TIMESTAMP))<8 THEN 'DAWN'

                    WHEN EXTRACT(HOUR FROM CAST(COL\_TIME AS TIMESTAMP))>17 AND EXTRACT(HOUR FROM CAST(COL\_TIME AS TIMESTAMP))<20 THEN 'DUSK'

                    WHEN EXTRACT(HOUR FROM CAST(COL\_TIME AS TIMESTAMP))>7 AND EXTRACT(HOUR FROM CAST(COL\_TIME AS TIMESTAMP))<18 THEN 'DAY'

                    ELSE 'NIGHT'

                    END)

                WHEN EXTRACT(MONTH FROM COL\_TIME)>3 OR EXTRACT(MONTH FROM COL\_TIME)<9 THEN

                    (CASE

                    WHEN EXTRACT(HOUR FROM CAST(COL\_TIME AS TIMESTAMP))>3 AND EXTRACT(HOUR FROM CAST(COL\_TIME AS TIMESTAMP))<6 THEN 'DAWN'

                    WHEN EXTRACT(HOUR FROM CAST(COL\_TIME AS TIMESTAMP))>19 AND EXTRACT(HOUR FROM CAST(COL\_TIME AS TIMESTAMP))<22 THEN 'DUSK'

                    WHEN EXTRACT(HOUR FROM CAST(COL\_TIME AS TIMESTAMP))>5 AND EXTRACT(HOUR FROM CAST(COL\_TIME AS TIMESTAMP))<20 THEN 'DAY'

                    ELSE 'NIGHT'

                    END)

            ELSE NULL

            END)

    END AS PERIOD

    FROM CASE

    )

SELECT PERIOD, COUNT(\*) AS N\_COLLISION

FROM COL\_PERIOD

WHERE PERIOD IS NOT NULL

GROUP BY PERIOD

ORDER BY N\_COLLISION DESC

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

|  |  |
| --- | --- |
| PERIOD | N\_COLLISION |
| DAY | 2575153 |
| NIGHT | 1041614 |
| DAWN | 30317 |
| DUSK | 28092 |

## Query Performance Analysis – Indexing

**Query 2:**

<Initial Running time/IO: 3371ms/37073

Optimized Running time/IO:1511ms/24804

**Explain the improvement:**

**Added the following index:**

**create index ve\_case\_ix on party\_involve(case\_id, ve\_num)**

**Necessities:**

From the initial plan, we can find there are three tables joining: ROAD\_EN, VEHICLE, PARTY\_INVOLVE and each of them involves a full file scan. We only need [case\_id, ve\_num] of the party\_involve table, the whole content of the road\_en table, [ve\_num, ve\_type] of the vehicle table. Since road\_con and case\_id are both primary keys of the road\_en table, there is no need to add indexes in the road\_en tables. So we build indexes on [case\_id, ve\_num] of the party\_involve table to reduce full scan and optimize.

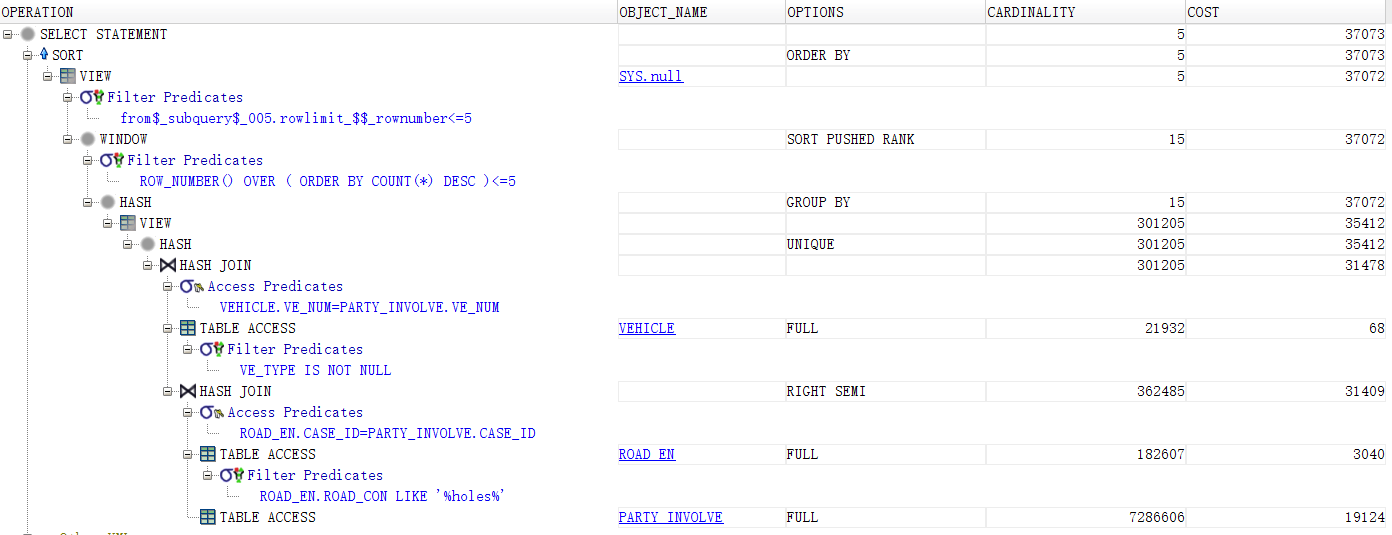
For this query, we added a joint index ‘ve\_case\_ix’ on <case\_id, ve\_num> on entity ‘party\_involve’. In this case, when doing hash join, the system can do a fast full scan on ‘party\_involve’ instead of a full scan.

In our case, the index ‘ve\_case\_ix’ is relevant when we want to group the collisions according to  the type of vehicles. So a fast full scan using a B-tree index is useful.

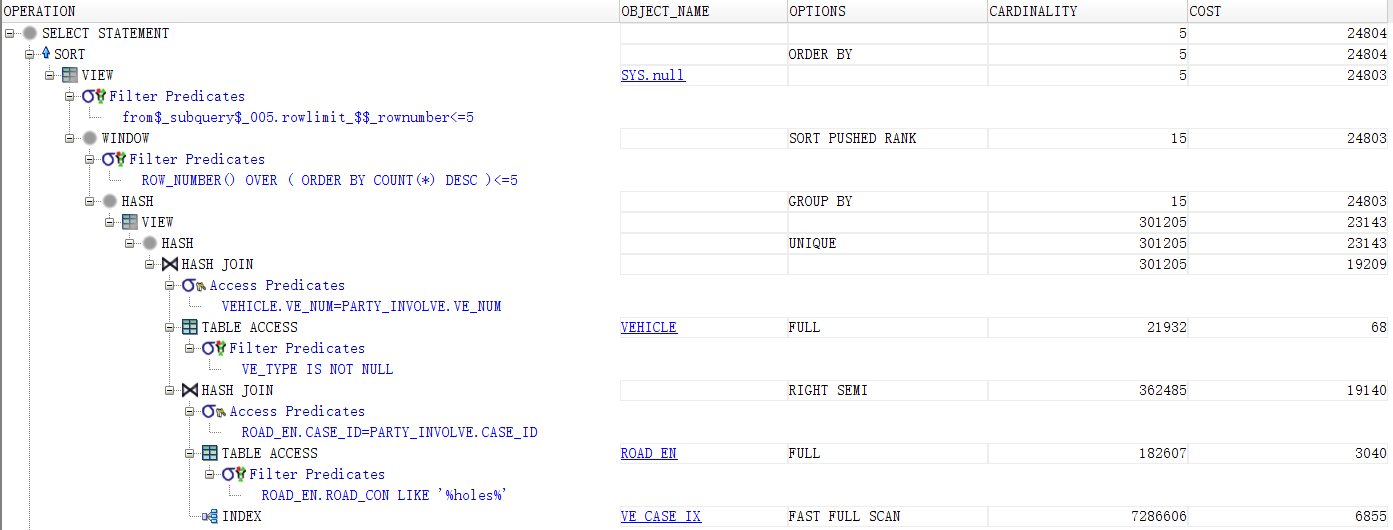
**Influence on the cost:**

After applying the two indexes, we could find that the total cost is reduced from 37073 to 24804. The cost of access of party\_involve is reduced from 19124 to 6855, and thus the hash join of party\_involve and road\_en is reduced, and then the hash join of vehicle and the party\_involve is also reduced due to the less cost of scanning the table party\_involve. Now the cost of scanning the party\_involve is still a big part of the total cost, but the cost of right semi join and the unique operation becomes the dominant part.

**Initial plan**



**Improved plan>**



**Query 5:**

<Initial Running time/IO: 21.494s/48986

Optimized Running time/IO: 14.703s /24096

**Explain the improvement:**

**Added the following index:**

**create index ve\_case\_ix on party\_involve(case\_id, ve\_num)**

**create index case\_loc\_ix on case(case\_id, loc\_num)**

**create index county\_city\_ix on location(loc\_num, county\_city)**

**Necessities:**

From the initial plan, we can find there are four tables joining: PARTY\_INVOLVE, VEHICLE, CASE, LOCATION and each of them involves a full file scan. We only need [case\_id, ve\_num] of the party\_involve table, [ve\_num, ve\_type] of the vehicle table, [case\_id, loc\_num] of the case table and [loc\_num, county\_city] of the location table. So we build indexes on [case\_id, ve\_num] of the party\_involve table, indexes on [case\_id, loc\_num] of the case table, and indexes on [loc,num, county\_city] of the location table to reduce full scan and optimize.

For this query, we added a joint index ‘ve\_case\_ix’ on <case\_id, ve\_num> on entity ‘party\_involve’, and ‘case\_loc\_ix’ on <case\_id, loc\_num> on entity ‘case’. In this case, when doing hash join, the system can do a fast full scan on ‘party\_involve’ and ‘case’ instead of a full scan.

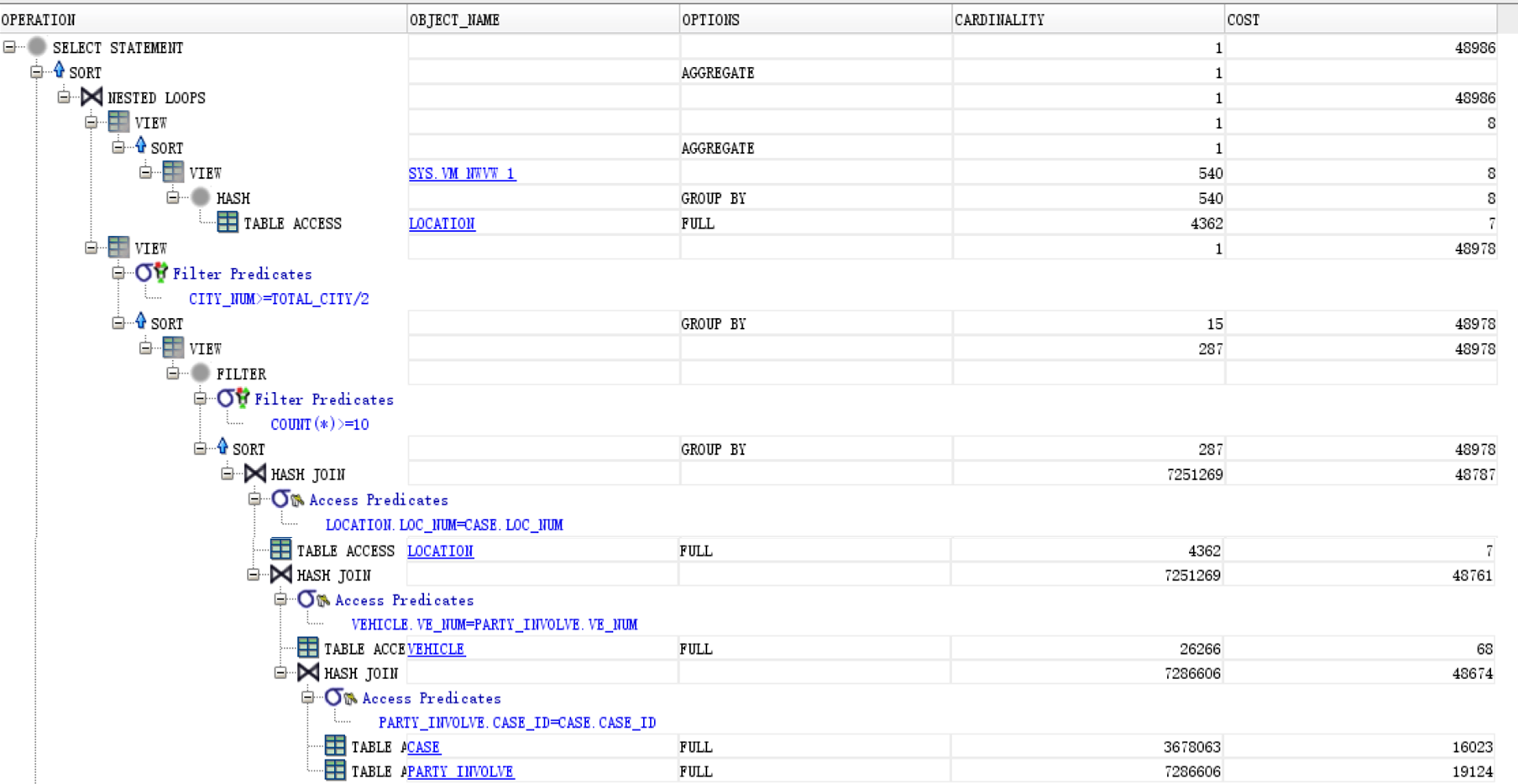
The index ‘ve\_case\_ix’ is relevant when we want to group the collisions according to the type of vehicles, and the index ‘case\_loc\_ix’ is used for the grouping of collisions according  to the county\_city code. In this case it is indeed useful to index the ‘case\_ id’  according to ‘ve\_num’ and ‘loc\_num’ as a B-tree. It could indeed search the desired result binarily.

We can see that the index ‘ve\_case\_ix’ has been used twice for a fast full scan for the matching of vehicle and case, which reduces the loss greatly, and improves the quality of the query.

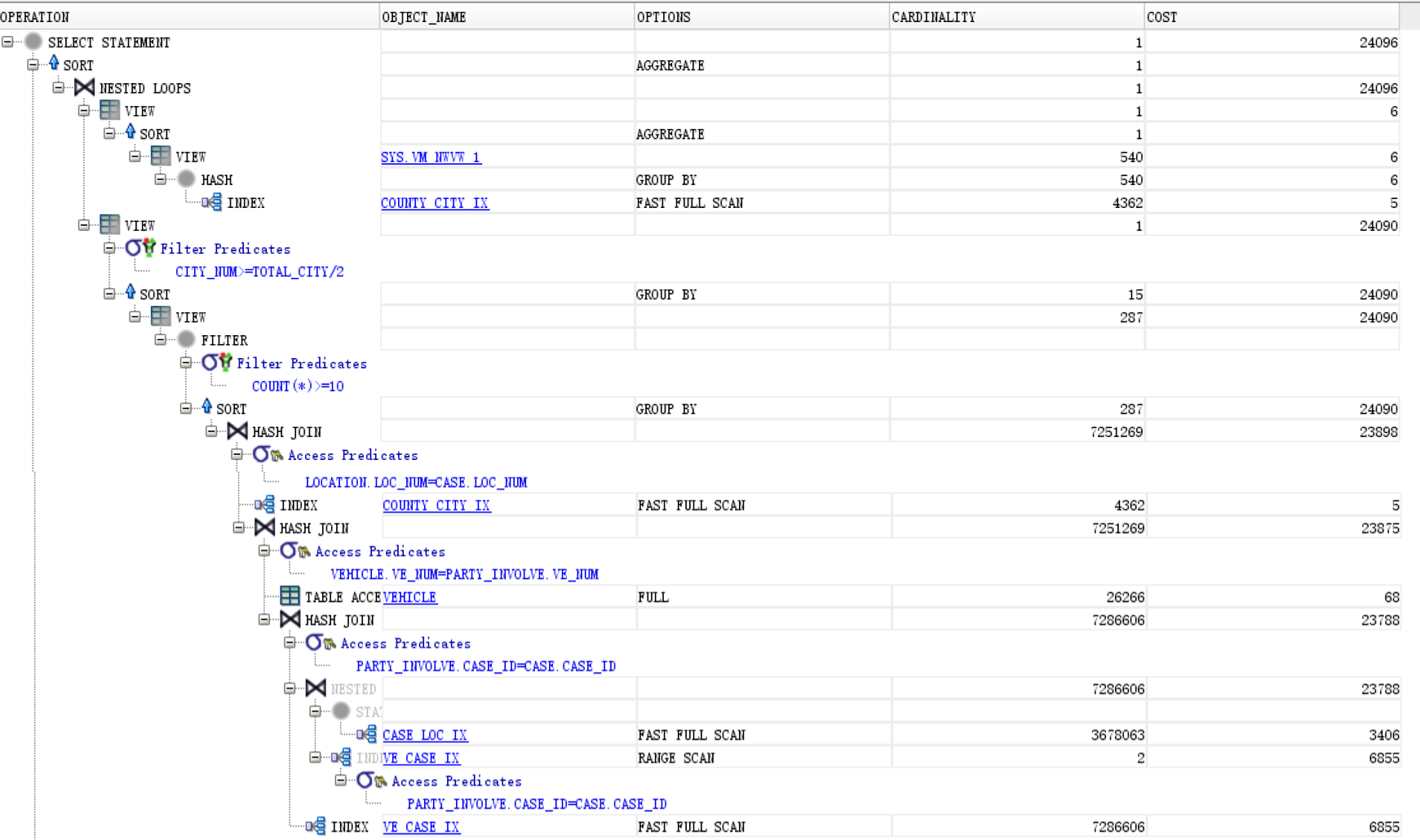
**Distribution of cost:**

Now for the optimized plan, the majority cost is spent on doing group by based on [ve\_type, county\_city] for ‘vehicle’ entity and the join of case and party\_involve table. After applying the three indexes, we could find that the total cost is reduced from 48986 to 24096. The cost of access of party\_involve is reduced from 19124 to 6855, and cost of accessing the case table is reduced from 16023 to 3406, then the cost of the join of them is also reduced. Due to the case\_loc\_ix index, the join of the location and the case is also reduced. The county\_city\_ix index plays a small role in the optimization.

**Initial plan**



**Improved plan>**



**Query 6:**

<Initial Running time/IO: 5 measurements shows initial average run time is 3641.6 ms / 47735

Optimized Running time/IO: 5 measurements shows optimized running time is 2980 ms / 37102

**Explain the improvement:**

**Added the following index:**

**create index loc\_in\_case on case(loc\_num)**

Necessities: From the initial plan, we can find there are five tables joining: COL\_CITY LOCATION, CASE, PARTY\_INVOLVE, ASSOCIATE\_VICTIM and each of them involves a full file scan. We only need [case\_id, loc\_num] of the case table, [[party\_id, case\_id] of the party\_involve table, [party\_id, vic\_age] of the associate\_victim table, [loc\_num, county\_city] of the location table, and we have selection predicates on these fields. So we try to build indexes on these columns to reduce full scan and optimize.

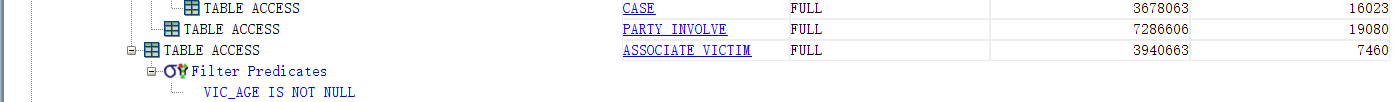
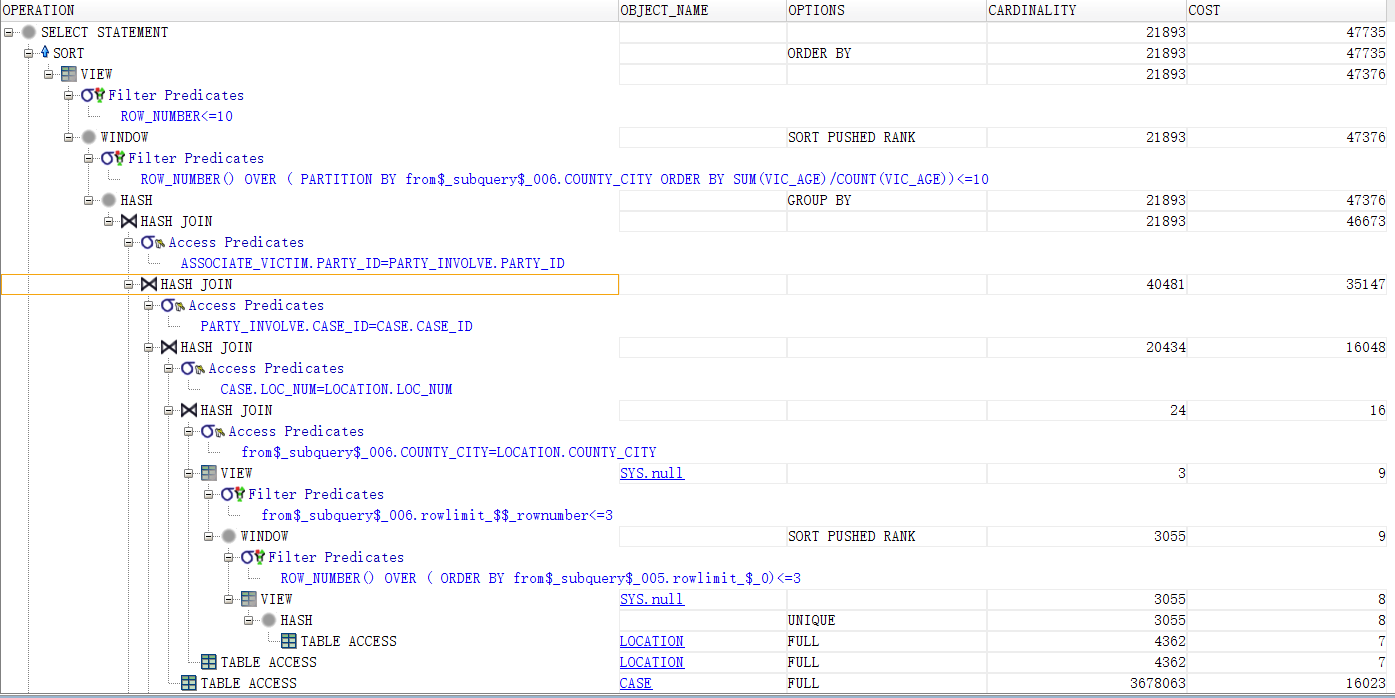
We tried several combinations of the indexes of these columns, but finally we found this one is useful in the timing, though other keys like location(county\_city) (for selection during the join), party\_involve(case\_id, loc\_num) (for utilizing the clustered index)  will also reduce the cost and change the plan by using index.

With the loc\_in\_case of the case table, the hash join of table case and location is completed by index, and it also utilizes the clustered case\_id row number as index, which is very nice . While joining and accessing the needed columns, they can just use the index range scan to check if the predicates are satisfied, thus the IO is reduced.

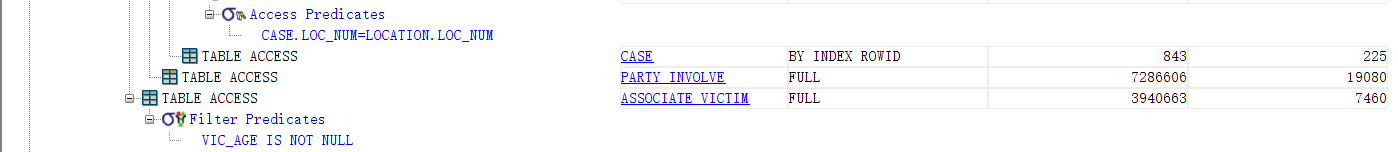
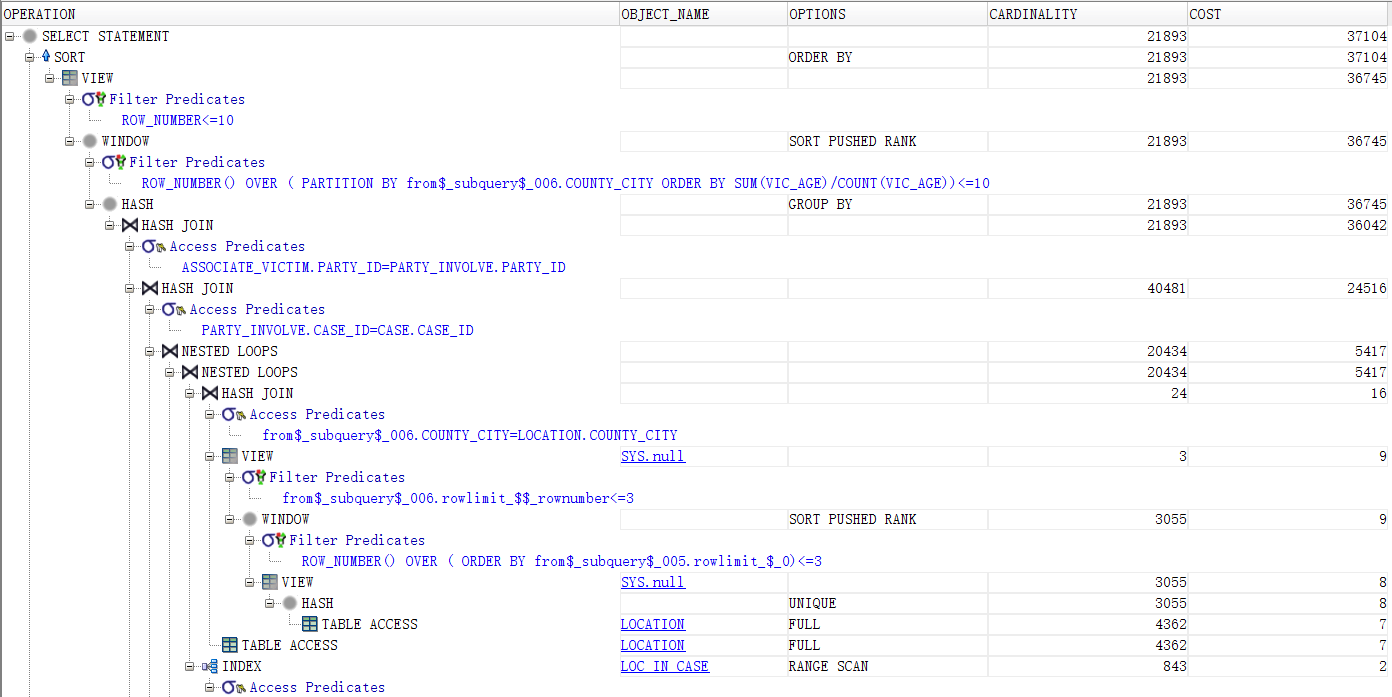
Influence on the cost: Cost is related to the CPU and IO. From the optimized plan, we can find that the total cost is reduced from 47735 to 37104, and the cost of case table access is reduced from 16023 to 225, and the cost of location access is reduced from 7 to 2, and the total cost of their join is reduced from 16048 to 5417 (this is why the total cost is reduced). There are also two hash joins becoming nested join, both because of the index scan. Maybe the optimizer chooses finds the indexed nested loop more efficient.

However, the cost of party\_involve full scan is still large and plays an important role in the total cost. We can reduce its cost by creating an index on party\_involve(party\_id, case\_id) and it indeed helps to reduce the cost, but because the running time suffers, we did not use it. From the plan we can also find that the group cost and join cost if dominated by the file access cost.

**Initial plan**



**Improved plan>**



**Query 8:**

<Initial Running time/IO: 2.478s / 19386

Optimized Running time/IO: 1.492s / 5998

**Necessity of indexing:**

This query requires us to a group by operation on the ‘ve\_num’ attribute on ‘party\_involve’ entity. However, there are no indexes for the ‘ve\_num’ attribute for ‘party\_involve’, and group by in this case can consume much cost. Therefore, it is necessary to add an index to ‘ve\_num’ for ‘party\_involve’.

**Explain the improvement:**

**Added the following index:**

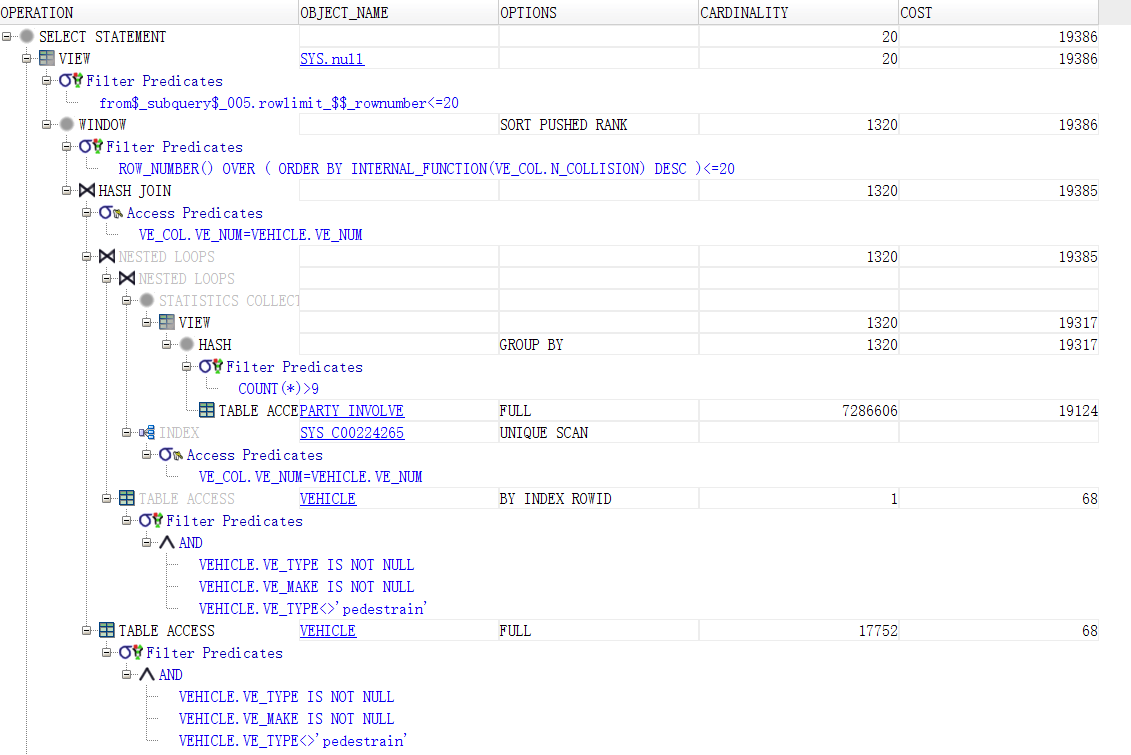
**create index ve\_party\_ix on party\_involve(party\_id, ve\_num)**

For this query, we added a joint index ‘ve\_party\_ix’ on <party\_id, ve\_num> on entity ‘party\_involve’. In this case, when doing group by, the system can use the ‘ve\_party\_ix’, which reduces the cost for group by from 19317 to 5929.

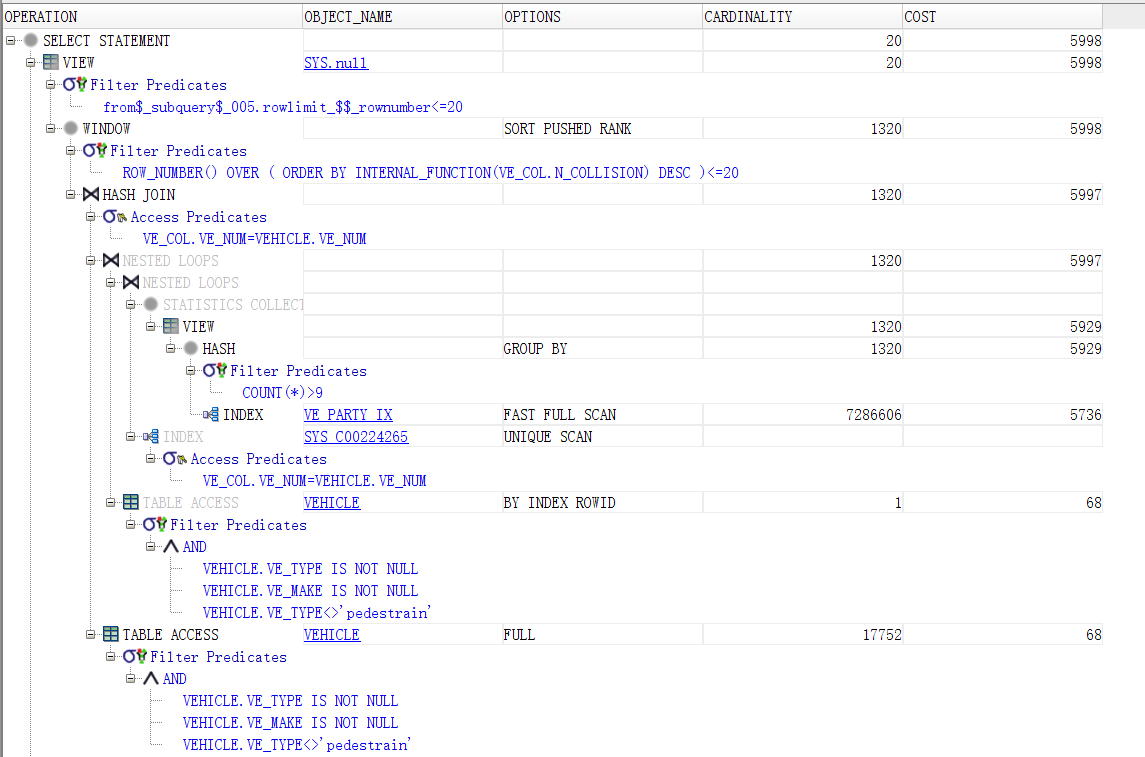
**Distribution of cost:**

Now for the optimized plan, the majority cost is spent on doing group by based on ‘ve\_num’ for ‘party\_involve’ entity. Little cost is spent on accessing the ‘vehicle’ entity (because it can access by its primary key, the index).

**Initial plan**



**Improved plan>**



**Query 9:**

<Initial Running time/IO: 4.343s / 16227

Optimized Running time/IO: 1.457s / 2260

**Necessity of indexes:**

Here in the query, we require the system to select from two different entities, given an equality condition ‘loc\_num’. Hence, the system has to do a join on the two entities. However, it only has a primary key ‘loc\_num’ for ‘location’ entity, and has no indexes for the ‘case’ entity. In this case, adding an index to ‘loc\_num’ for ‘case’ entity is necessary to reduce the costs for the join.

**Explain the improvement:**

**Added the following indexes:**

**create index loc\_in\_case on case(loc\_num)**

For this query, we added an index ‘loc\_in\_case’ on ‘loc\_num’ for ‘case’ entity.

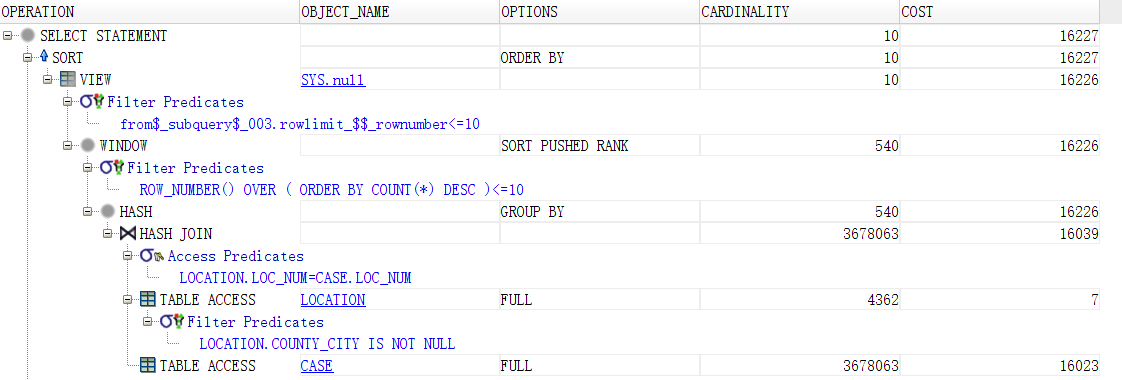
We can see from the query plan that the index ‘loc\_in\_case’ has been used twice.

Based on these indexes, when the system is doing hash join for ‘location’ entity and ‘case’ entity on ‘location.loc\_num = case.loc\_num’ (to find the name of the cities respective to the location numbers), it can do a fast full scan on ‘case’ entity, instead of a full scan, which greatly reduces the cost from 16023 to 2056.

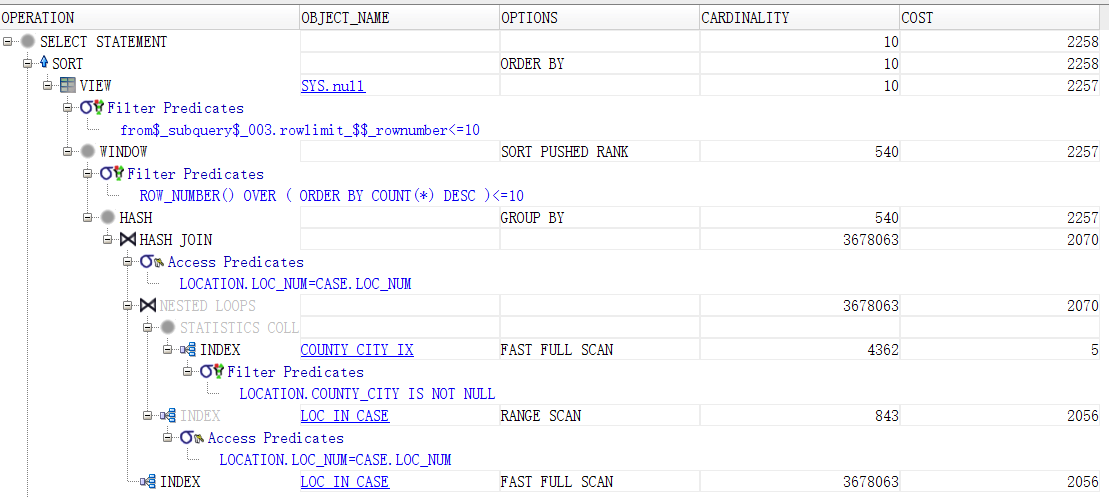
**Distribution of cost:**

From the optimized query plan, we can see very little cost is spent on scanning the table ‘location’, the majority cost is spent on doing range scan on ‘case’ tabel, fulfilling the equality constraint. Despite this, it still saves much cost compared to the original plan.

Initial plan



Improved plan>



# General Comments

In this project, we finished the design of our database system, from designing the ER diagram, preparing data to implement the queries, and optimizing the execution. We successfully compelted the tasks requried in the project, a very comprehensive and practical project.   
Many thanks for the help from the professors and TAs!