|  |
| --- |
|  |
| Introduction To Database |
| Project Report |
| Group 3 :  Joanna Salathé |
| Eliéva Pignat  Sacha Bron |
|  |

|  |
| --- |
| This document contains the final report of the project. It contains all the subjects of the 3 deliverables |

Table des matières

[Modification of deliverable 2 2](#_Toc389575304)

[Queries 2](#_Toc389575305)

[Queries A to G 2](#_Toc389575306)

[Queries H to S 9](#_Toc389575307)

[Index performance 22](#_Toc389575308)

[Importance of indexes based 22](#_Toc389575309)

[Query Plans 22](#_Toc389575310)

[Running Time 26](#_Toc389575311)

[User Interface 27](#_Toc389575312)

[Choices 27](#_Toc389575313)

[Search Functionality 27](#_Toc389575314)

[Server sides 28](#_Toc389575315)

[Why Java 28](#_Toc389575316)

[Web server 28](#_Toc389575317)

[Database Access 29](#_Toc389575318)

[Client Size 30](#_Toc389575319)

[How to use the Interface 30](#_Toc389575320)

[Pre-written Requests 30](#_Toc389575321)

[Custom Search 30](#_Toc389575322)

[Clicable Results 30](#_Toc389575323)

[Insertion 30](#_Toc389575324)

[Deletion 30](#_Toc389575325)

[Interface Screenshots 31](#_Toc389575326)

[Importation 34](#_Toc389575327)

[ER-Model 34](#_Toc389575328)

[Constraints Explanation 35](#_Toc389575329)

[Areas 35](#_Toc389575330)

[Tracks 35](#_Toc389575331)

[Recordings 35](#_Toc389575332)

[Releases 35](#_Toc389575333)

[Table Creation 36](#_Toc389575334)

[Merge for participation constraints 36](#_Toc389575335)

[SQL Code 36](#_Toc389575336)

# Modification of deliverable 2

After the feedback, the content of the deliverable 2 was modified in the following way:

1. Some queries had to be modified and corrected. For example we add DISTINCT when we count. And we GROUP BY first by ID and after by NAME
2. The foreign key on artist of the artist\_track table was removed to be consistent with our database.
3. A short explanation of the queries A to G was added

# Queries

## Queries A to G

We test all the following queries with a dummy database that is smaller than the music database and the queries give us the expected results. But with the real database some of the queries take a lot of time to be executed (we kill the process after 5 minutes and when we let the execution run the following error can appear…). Those queries are the query B, F. Thus we think that this problem comes from a lack of optimisation of those queries (too much join or too much nested loop).



Figure 1: error when the query E is run during a long time

Query A :

--A print the name of artist from Switzerland

SELECT A.name

FROM Artist A, Area B

WHERE A.ID\_AREA = B.ID\_AREA AND B.name= 'Switzerland'



Figure 2 result of query A

Query B :

--B print the name and the number of female, male and group of the

--area that have the most female, male or group artists.

/\*The first part of the query create a table with the following column : AREA| female count| male count| group count and in

SELECT \*

FROM

(SELECT B.name AS NAME,

(SELECT COUNT(DISTINCT A.ID\_ARTIST)

        FROM ARTIST A

        WHERE A.GENDER='Female' AND B.ID\_AREA=A.ID\_AREA)

        AS COUNTF,

(SELECT COUNT(DISTINCT A.ID\_ARTIST)

        FROM ARTIST A

        WHERE A.GENDER='Male' AND B.ID\_AREA=A.ID\_AREA)

        AS COUNTM,

(SELECT COUNT(DISTINCT A.ID\_ARTIST)

        FROM ARTIST A

        WHERE A.TYPE='Group' AND B.ID\_AREA=A.ID\_AREA)

        AS COUNTG

FROM AREA B

ORDER BY COUNTF DESC)

WHERE ROWNUM=1

UNION

SELECT \*

FROM

(SELECT B.name AS NAME,

(SELECT COUNT(DISTINCT A.ID\_ARTIST)

        FROM ARTIST A

        WHERE A.GENDER='Female' AND B.ID\_AREA=A.ID\_AREA)

        AS COUNTF,

(SELECT COUNT(DISTINCT A.ID\_ARTIST)

        FROM ARTIST A

        WHERE A.GENDER='Male' AND B.ID\_AREA=A.ID\_AREA)

        AS COUNTM,

(SELECT COUNT(DISTINCT A.ID\_ARTIST)

        FROM ARTIST A

        WHERE A.TYPE='Group' AND B.ID\_AREA=A.ID\_AREA)

        AS COUNTG

FROM AREA B

ORDER BY COUNTM DESC)

WHERE ROWNUM=1

UNION

SELECT \*

FROM

(SELECT B.name AS NAME,

(SELECT COUNT(DISTINCT A.ID\_ARTIST)

        FROM ARTIST A

        WHERE A.GENDER='Female' AND B.ID\_AREA=A.ID\_AREA)

        AS COUNTF,

(SELECT COUNT(DISTINCT A.ID\_ARTIST)

        FROM ARTIST A

        WHERE A.GENDER='Male' AND B.ID\_AREA=A.ID\_AREA)

        AS COUNTM,

(SELECT COUNT(DISTINCT A.ID\_ARTIST)

        FROM ARTIST A

        WHERE A.TYPE='Group' AND B.ID\_AREA=A.ID\_AREA)

        AS COUNTG

FROM AREA B

ORDER BY COUNTG DESC)

WHERE ROWNUM=1;

Query C :

--C List the name of 10 groups with the most recorded track

/\*order by the number of tracks and select the top 10 for the group\*/

SELECT \*

FROM(SELECT A.NAME

FROM  Artist A, Artist\_Track S

WHERE A.ID\_ARTIST=S.ID\_ARTIST AND A.TYPE='Group'

GROUP BY A.ID\_ARTIST, A.NAME

ORDER BY count(S.ID\_TRACK) DESC)

WHERE ROWNUM <=10

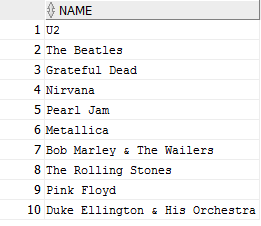


Figure 3 result of query C

Query D :

--D List the name of 10 groups with the most release

/\*Order by the number of releases and select the top 10 for the group\*/

SELECT \*

FROM(SELECT **A.NAME**

FROM Artist A, Track T, Artist\_Track S, Medium M, Release R

WHERE A.ID\_ARTIST =S.ID\_ARTIST AND T.ID\_TRACK=S.ID\_TRACK

AND T.ID\_MEDIUM=M.ID\_MEDIUM AND R.ID\_RELEASE = M.ID\_RELEASE

AND A.TYPE='Group'

GROUP BY A.ID\_ARTIST, **A.NAME**

ORDER BY count(DISTINCT **R.ID\_RELEASE**) DESC)

WHERE ROWNUM <=10



Figure 4 result of query D

Query E :

--E Print the name of female artist that have the most genres

/\*Order by the number of genre of female artist and select the top 1 artist\*/

SELECT NAME

FROM (SELECT A.NAME AS NAME,

COUNT(DISTINCT G.ID\_GENRE) AS COUNT\_GENRE

FROM ARTIST\_GENRE G, ARTIST A

WHERE A.ID\_ARTIST= G.ID\_ARTIST AND A.GENDER='Female'

GROUP BY A.ID\_ARTIST, A.NAME

ORDER BY COUNT\_GENRE DESC)

WHERE ROWNUM=1



Figure 5 result of query E

Query F :

--F Print the name of the cities that have more female artist than

--male artist

/\*Compare the count of the female artist and the count of male artist for the city. And select the cites that have more female artist than male artist\*/

SELECT B.name

FROM Area B

WHERE B.type='City'

AND (SELECT Count(\*)

FROM Artist A

WHERE A.gender='Female' AND A.ID\_AREA= B.ID\_AREA)

     > (SELECT Count(\*)

FROM Artist A1

WHERE A1.gender='Male' AND A1.ID\_AREA=B.ID\_AREA)

Query G :

--G List the mediums with the most number of tracks

/\*Order the mediums by their track count and select the medium that have their track count egal to the max track count.\*/

SELECT ID, FORMAT, COUNT\_TRACK

FROM

(SELECT M.ID\_MEDIUM AS ID, M.FORMAT AS FORMAT,

COUNT(DISTINCT T.ID\_TRACK) AS COUNT\_TRACK

FROM TRACK T, MEDIUM M

WHERE M.ID\_MEDIUM = T.ID\_MEDIUM

GROUP BY M.ID\_MEDIUM, M.FORMAT)

WHERE COUNT\_TRACK = (SELECT MAX(COUNT\_TRACK)

FROM

(SELECT M.ID\_MEDIUM AS ID, M.FORMAT AS FORMAT,

COUNT(DISTINCT T.ID\_TRACK) AS COUNT\_TRACK

FROM TRACK T, MEDIUM M

WHERE M.ID\_MEDIUM=T.ID\_MEDIUM

GROUP BY M.ID\_MEDIUM, M.FORMAT)

);



Figure 6 result of query G

## Queries H to S

The queries H, M, N and S don’t give us any results. They never end (we stop after more than 1 hour). But they work (terminate and give consistant result) when we compute them on our « mini » database (a dummy database with the same schema than the project database but with less entries).

We also try to reduce the number of joins (that was really big). But reducing the multiple join introduced nested loops or/and multiple views and the queries still don’t give us any result.

For the others queries, you can find in this report the SQL codes, an explanation and (a part of) the results.

Query H :

--H :For each area that has more than 30 artists, list the male

--artist, the female artist and the group with the most tracks

--recorded.

-- The query H does not stop

/\*We do three different views. One to obtain the male artist, one for the female and the last one for the group by areas.\*/

SELECT A.NAME, M.ART\_NAME, F.ART\_NAME, G.ART\_NAME

FROM AREA A,

(SELECT AREA\_ID, ART\_NAME

FROM

(SELECT AM.ID\_AREA AS AREA\_ID, AM.NAME AS ART\_NAME,

COUNT(DISTINCT ATM.ID\_TRACK) AS TCOUNT,

ROW\_NUMBER()OVER (PARTITION BY AM.ID\_AREA

ORDER BY COUNT(DISTINCT ATM.ID\_TRACK)DESC) AS RN

FROM ARTIST AM, ARTIST\_TRACK ATM

WHERE AM.GENDER='Male' AND AM.ID\_ARTIST=ATM.ID\_ARTIST

AND AM.ID\_AREA IN (SELECT AR2.ID\_AREA

FROM AREA AR2, ARTIST A

WHERE A.ID\_AREA= AR2.ID\_AREA

GROUP BY AR2.ID\_AREA

HAVING COUNT(DISTINCT A.ID\_ARTIST)>=30)

GROUP BY AM.ID\_AREA, AM.ID\_ARTIST, AM.NAME

)WHERE RN<=1) M,

(SELECT AREA\_ID, ART\_NAME

FROM

(SELECT AM.ID\_AREA AS AREA\_ID, AM.NAME AS ART\_NAME,

COUNT(DISTINCT ATM.ID\_TRACK) AS TCOUNT,

ROW\_NUMBER()OVER (PARTITION BY AM.ID\_AREA

ORDER BY COUNT(DISTINCT ATM.ID\_TRACK)DESC) AS RN

FROM ARTIST AM, ARTIST\_TRACK ATM

WHERE AM.GENDER='Female' AND AM.ID\_ARTIST=ATM.ID\_ARTIST

AND AM.ID\_AREA IN (SELECT AR2.ID\_AREA

FROM AREA AR2, ARTIST A

WHERE A.ID\_AREA= AR2.ID\_AREA

GROUP BY AR2.ID\_AREA

HAVING COUNT(DISTINCT A.ID\_ARTIST)>=30)

GROUP BY AM.ID\_AREA, AM.ID\_ARTIST, AM.NAME

)WHERE RN<=1) F,

(SELECT AREA\_ID, ART\_NAME

FROM

(SELECT AM.ID\_AREA AS AREA\_ID, AM.NAME AS ART\_NAME,

COUNT(DISTINCT ATM.ID\_TRACK) AS TCOUNT,

ROW\_NUMBER()OVER (PARTITION BY AM.ID\_AREA

ORDER BY COUNT(DISTINCT ATM.ID\_TRACK)DESC) AS RN

FROM ARTIST AM, ARTIST\_TRACK ATM

WHERE AM.TYPE='Group' AND AM.ID\_ARTIST=ATM.ID\_ARTIST

AND AM.ID\_AREA IN (SELECT AR2.ID\_AREA

FROM AREA AR2, ARTIST A

WHERE A.ID\_AREA= AR2.ID\_AREA

GROUP BY AR2.ID\_AREA

HAVING COUNT(DISTINCT A.ID\_ARTIST)>=30)

GROUP BY AM.ID\_AREA, AM.ID\_ARTIST, AM.NAME

)WHERE RN<=1) G

WHERE A.ID\_AREA=M.AREA\_ID AND A.ID\_AREA=G.AREA\_ID

AND A.ID\_AREA=F.AREA\_ID;

Query I :

-- I: American metal group Metallica is asking its fans to choose

-- the set list for its upcoming concert in Switzerland.

-- Assuming that the Metallica fans will choose the songs that have -- appeared on the highest number of mediums, list the top 25 songs.

/\*Order the recordings of Metallica by the count of their mediums apparitions in the track table and select the top 25 of them \*/

SELECT DISTINCT \*

FROM

(SELECT R.NAME, COUNT(DISTINCT T.ID\_MEDIUM)

FROM TRACK T, RECORDING R, ARTIST\_TRACK AT, ARTIST A

WHERE A.NAME = 'Metallica' AND A.ID\_ARTIST = AT.ID\_ARTIST

AND AT.ID\_TRACK = T.ID\_TRACK AND T.ID\_RECORDING = R.ID\_RECORDING

GROUP BY R.ID\_RECORDING, R.NAME

ORDER BY COUNT(DISTINCT T.ID\_MEDIUM) DESC)

WHERE ROWNUM <= 25;

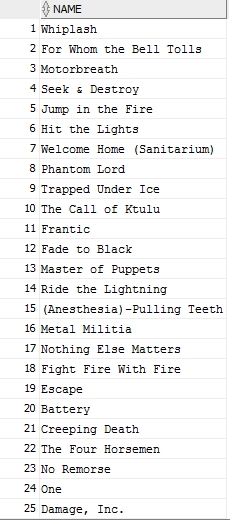


Figure 7 result of query I

Query J :

--J : For each of the 10 genres with the most artists, list the

--female artist that has recorded the highest number of tracks.

/\*First we order the genre by the count of their artists and select the top 10 of them. Then for each genre that belongs to this subset

we check if it has female artists in it. If so we order its female artists by counting the number of track they did with an over partition operator and select the first row for each area.\*/

SELECT NAME

FROM(

SELECT AG.ID\_GENRE, A.NAME AS NAME, COUNT(DISTINCT AT.ID\_TRACK),

ROW\_NUMBER()OVER (PARTITION BY AG.ID\_GENRE

ORDER BY COUNT(DISTINCT AT.ID\_TRACK) DESC) AS RN

FROM ARTIST A, ARTIST\_TRACK AT, ARTIST\_GENRE AG

WHERE A.ID\_ARTIST= AT.ID\_ARTIST AND A.GENDER='Female'

AND A.ID\_ARTIST=AG.ID\_ARTIST

AND AG.ID\_GENRE IN (SELECT \*

FROM

(SELECT G.ID\_GENRE

FROM ARTIST\_GENRE G

GROUP BY G.ID\_GENRE

ORDER BY COUNT(DISTINCT G.ID\_ARTIST) DESC)

WHERE ROWNUM <=10)

GROUP BY AG.ID\_GENRE, A.ID\_ARTIST, A.NAME

)WHERE RN=1;

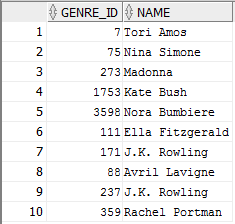


Figure 8 result of query J

Query K :

--K : List all genres that have no female artists, all genres that

--have no male artists and all genres that have no groups.

/\*Select the genres that are not present in the result of the selection of genres that have female artist.

Select the genres that are not present in the result of the selection of genres that have male artist.

Select the genres that are not present in the result of the selection of genres that have group artist.

Union the results of each of these sub-queries.\*/

SELECT G.ID\_GENRE

FROM GENRE G

WHERE G.ID\_GENRE

NOT IN (SELECT AG.ID\_GENRE

FROM ARTIST\_GENRE AG, ARTIST A

WHERE AG.ID\_ARTIST = A.ID\_ARTIST AND A.GENDER = 'Female')

UNION

SELECT G.ID\_GENRE

FROM GENRE G

WHERE G.ID\_GENRE

NOT IN (SELECT AG.ID\_GENRE

FROM ARTIST\_GENRE AG, ARTIST A

WHERE AG.ID\_ARTIST = A.ID\_ARTIST AND A.GENDER = 'Male')

UNION

SELECT G.ID\_GENRE

FROM GENRE G

WHERE G.ID\_GENRE

NOT IN (SELECT AG.ID\_GENRE

FROM ARTIST\_GENRE AG, ARTIST A

WHERE AG.ID\_ARTIST = A.ID\_ARTIST AND A.TYPE = 'Group') ;

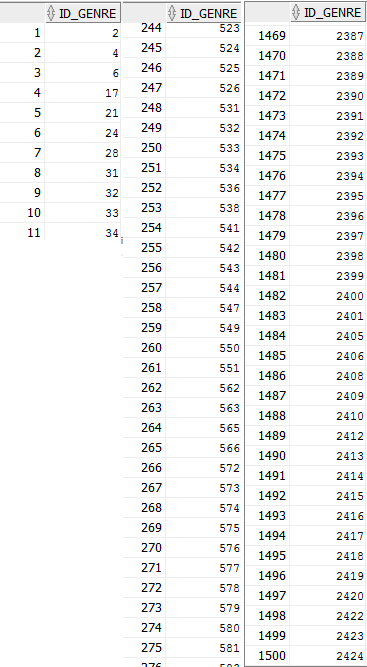


Figure 9 result of query K

Query L :

--L : For each area with more than 10 groups, list the 5 male

--artists that have recorded the highest number of tracks.

/\*Select the areas that have more than 10 groups and for each of those areas we order the artist of this area by their count record and select the 5 highest of each area \*/

SELECT AREA\_NAME, ART\_NAME

FROM

(SELECT ARE.NAME AS AREA\_NAME, A.NAME AS ART\_NAME,

COUNT(DISTINCT AT.ID\_TRACK) AS TRACKCOUNT,

ROW\_NUMBER()OVER(PARTITION BY ARE.ID\_AREA, ARE.NAME

ORDER BY COUNT(DISTINCT AT.ID\_TRACK) DESC) AS RN

FROM ARTIST A, ARTIST\_TRACK AT, AREA ARE

WHERE A.ID\_ARTIST = AT.ID\_ARTIST AND A.ID\_AREA=ARE.ID\_AREA

AND ARE.ID\_AREA IN

(SELECT AR.ID\_AREA

FROM AREA AR, ARTIST A2

WHERE AR.ID\_AREA = A2.ID\_AREA AND A2.TYPE = 'Group'

GROUP BY AR.ID\_AREA

HAVING COUNT(DISTINCT A2.ID\_ARTIST) > 10)

GROUP BY ARE.ID\_AREA, ARE.NAME, A.ID\_ARTIST, A.NAME)

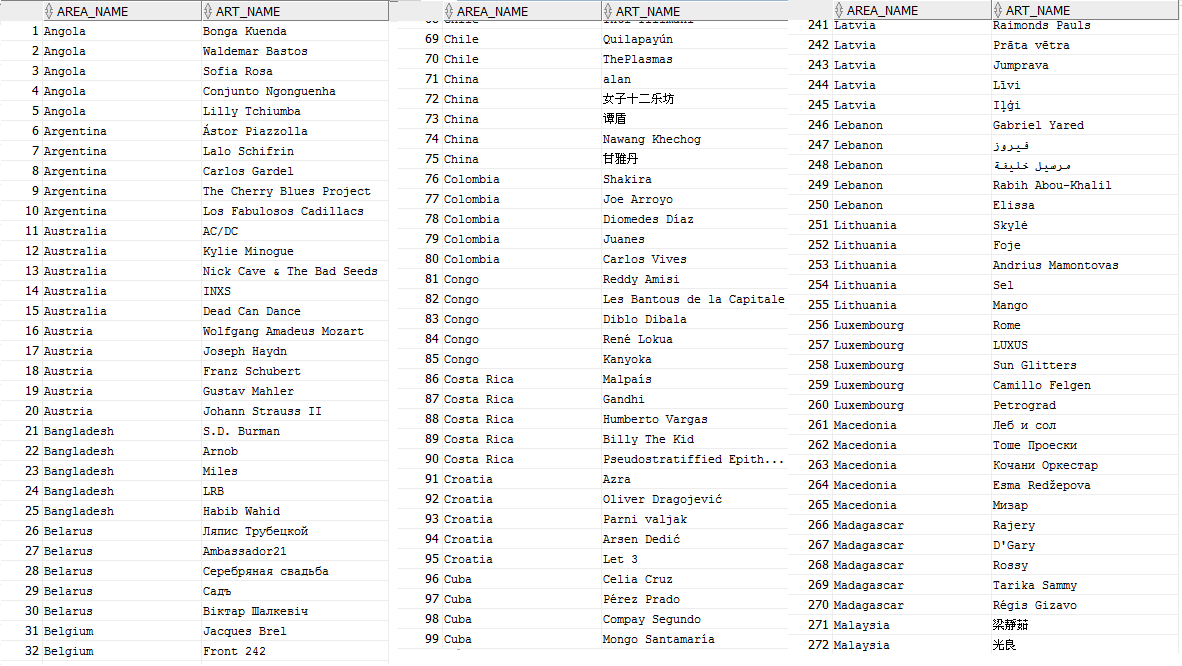
WHERE RN <= 5;

Figure 10 result of query L

Query M :

--M :List the 10 groups with the highest number of tracks that appear

--on compilations. A compilation is a medium that contains tracks

--associated with more than one artist.

--Query M does not stop

/\*Count the number of tracks of an artist if the track is on a medium that contains at least another track create by another artist. Order the artists by the number of those track and finally select the top 10\*/

SELECT \*

FROM

(SELECT A.NAME AS NAME, COUNT(DISTINCT T.ID\_TRACK)

FROM ARTIST A, ARTIST\_TRACK AT, TRACK T

WHERE A.TYPE='Group' AND A.ID\_ARTIST=AT.ID\_ARTIST

AND AT.ID\_TRACK=T.ID\_TRACK AND T.ID\_MEDIUM

IN (SELECT DISTINCT T2.ID\_MEDIUM

FROM ARTIST\_TRACK AT2, TRACK T2, ARTIST\_TRACK AT3, TRACK T3

WHERE AT2.ID\_TRACK=T2.ID\_TRACK AND AT3.ID\_TRACK =T3.ID\_TRACK

AND T2.ID\_TRACK <>T3.ID\_TRACK AND AT3.ID\_ARTIST <>AT2.ID\_ARTIST

AND T2.ID\_MEDIUM=T3.ID\_MEDIUM)

GROUP BY A.ID\_ARTIST, A.NAME

ORDER BY COUNT(DISTINCT T.ID\_TRACK)DESC

)WHERE ROWNUM<=10;

Query N :

--N :List the top 10 releases with the most collaborations, i.e.,

--releases where one artist is performing all songs and the highest

--number of different guest artists contribute to the album.

--Query N : does not give a result. There is big joins and then the

--computation is too much heavy and slow

SELECT \*

FROM

(SELECT B.RELEASE\_ID, C.COUNT\_ART

FROM

--# OF TRACKS PER ARTIST PER RELEASE

(SELECT M.ID\_RELEASE AS RELEASE\_ID,

COUNT(DISTINCT T.ID\_TRACK) AS COUNT\_TRACK

FROM ARTIST A, TRACK T, MEDIUM M, ARTIST\_TRACK AT

WHERE A.ID\_ARTIST=AT.ID\_ARTIST AND AT.ID\_TRACK=T.ID\_TRACK

AND T.ID\_MEDIUM=M.ID\_MEDIUM

GROUP BY M.ID\_RELEASE, A.ID\_ARTIST) A,

--# OF TRACK PER RELEASE

(SELECT M.ID\_RELEASE AS RELEASE\_ID,

COUNT(DISTINCT T.ID\_TRACK)AS COUNT\_TRACK

FROM MEDIUM M, TRACK T

WHERE T.ID\_MEDIUM=M.ID\_MEDIUM

GROUP BY M.ID\_RELEASE) B,

--# OF DIFFERENT ARTISTS PER RELEASE

(SELECT M.ID\_RELEASE AS RELEASE\_ID,

COUNT(DISTINCT A.ID\_ARTIST)AS COUNT\_ART,

COUNT(DISTINCT T.ID\_TRACK) AS COUNT\_TRACK

FROM ARTIST A, TRACK T, MEDIUM M, ARTIST\_TRACK AT

WHERE A.ID\_ARTIST=AT.ID\_ARTIST

AND AT.ID\_TRACK=T.ID\_TRACK AND T.ID\_MEDIUM=M.ID\_MEDIUM

GROUP BY M.ID\_RELEASE) C

WHERE B.RELEASE\_ID=A.RELEASE\_ID

AND A.COUNT\_TRACK=B.COUNT\_TRACK AND B.RELEASE\_ID=C.RELEASE\_ID

GROUP BY B.RELEASE\_ID, C.COUNT\_ART

ORDER BY C.COUNT\_ART DESC

)WHERE ROWNUM<=10;

Query O :

--O : List the release which is associated with the most mediums. If

--there are more than one such release, list all such releases.

/\*For each release we count the number of medium they are associated to and we order these releases by this count.

Then we inject this result table in one from of another query. In this outer query we check for all releases if the count is equal to the max count of all releases.\*/

--The code given here is not the code use for testing the index

--performance but it is a new implementation that is really more

--fast

SELECT RID

FROM

(SELECT M.ID\_RELEASE AS RID, COUNT(DISTINCT M.ID\_MEDIUM) AS COUNTM

FROM MEDIUM M

GROUP BY M.ID\_RELEASE

ORDER BY COUNT(DISTINCT M.ID\_MEDIUM) DESC)

WHERE COUNTM = (SELECT MAX(COUNTM)

FROM

(SELECT COUNT(DISTINCT M.ID\_MEDIUM) AS COUNTM

FROM MEDIUM M

GROUP BY M.ID\_RELEASE

ORDER BY COUNT(DISTINCT M.ID\_MEDIUM) DESC));



Figure 11 result of query O

Query P :

--P :List the most popular genre among the groups which are associated with at least 3 genres.

/\*We select the groups that are associated with at least 3 genres and count the number of total artist for the genre associated with the previous artists selection .\*/

SELECT GNAME

FROM

(SELECT G.NAME AS GNAME

FROM GENRE G, ARTIST\_GENRE AG

WHERE G.ID\_GENRE=AG.ID\_GENRE AND AG.ID\_ARTIST IN

(SELECT AG.ID\_ARTIST

FROM ARTIST A, ARTIST\_GENRE AG

WHERE A.TYPE = 'Group' AND AG.ID\_ARTIST = A.ID\_ARTIST

GROUP BY AG.ID\_ARTIST

HAVING COUNT(DISTINCT AG.ID\_GENRE)>=3)

GROUP BY G.ID\_GENRE, G.NAME

ORDER BY COUNT(DISTINCT AG.ID\_ARTIST) DESC

)WHERE ROWNUM <= 1;



Figure 12 result of query P

Query Q:

-- Q : List the 5 titles that are associated with the most different

--songs (recordings) along with the number of songs that share such

--title.

/\*We select the name of the titles and for each of them we count the number of recording that has this name and we

order them by this count. Finally we select the top 5 of them.\*/

SELECT \*

FROM

(SELECT R.NAME, COUNT(R.ID\_RECORDING)

FROM RECORDING R

GROUP BY R.NAME

ORDER BY COUNT(R.ID\_RECORDING) DESC)

WHERE ROWNUM <=5;

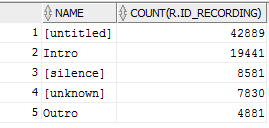


Figure 13 result of query Q

Query R :

--R : List the top 10 artists according to their track-to-release

--ratio. This ratio is computed by dividing the number of tracks an

--artist is associated with by the number of releases this artist

--as contributed a track to.

/\*We create 2 views. The first one with the count of tracks of each artist and the second one with the count of releases per artist and then compute the ratio and select the top 10.\*/

SELECT \*

FROM

(SELECT A.NAME, COUNT\_TRACK/COUNT\_RELEASE AS RATIO

FROM

(SELECT AT.ID\_ARTIST AS ART\_ID,

COUNT(DISTINCT AT.ID\_TRACK) AS COUNT\_TRACK

FROM ARTIST\_TRACK AT

GROUP BY AT.ID\_ARTIST) TR,

(SELECT AT.ID\_ARTIST AS ART\_ID,

COUNT(DISTINCT M.ID\_RELEASE) AS COUNT\_RELEASE

FROM ARTIST\_TRACK AT, TRACK T, MEDIUM M

WHERE AT.ID\_TRACK=T.ID\_TRACK AND T.ID\_MEDIUM=M.ID\_MEDIUM

GROUP BY AT.ID\_ARTIST) ME,

ARTIST A

WHERE A.ID\_ARTIST=TR.ART\_ID AND A.ID\_ARTIST =ME.ART\_ID

ORDER BY RATIO DESC

)WHERE ROWNUM <=10;

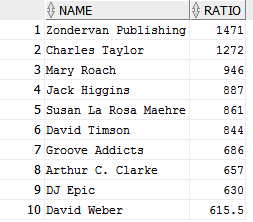


Figure 14 result of query R

Query S :

--S :The concert hit index is a measure of probability that the

--artist can attract enough fans to fill a football stadium.

--We define the “hit artist” as one that has more than 10 songs that

--appear on more than 100 mediums and measure "hit ability" as the

--average number of mediums that a top 10 song appears on.

--List all “hit artists” according to their "hit ability".

--S does not stop

/\*select songs and associated artist that appear on more than 100 medium and then compute the average for artist that have at least 10 song. This don’t works because of the big join (artist\_track, recording and track) .\*/

SELECT ART\_ID, AVG(COUNTM)

FROM

(SELECT AT.ID\_ARTIST AS ART\_ID, R.NAME AS RECORD\_NAME,

COUNT(DISTINCT T.ID\_MEDIUM) AS COUNTM

FROM ARTIST\_TRACK AT, RECORDING R, TRACK T

WHERE T.ID\_RECORDING=R.ID\_RECORDING AND AT.ID\_TRACK=T.ID\_TRACK

GROUP BY AT.ID\_ARTIST, R.NAME

HAVING COUNT(DISTINCT T.ID\_MEDIUM)>=100

)GROUP BY ART\_ID

HAVING COUNT(RECORD\_NAME)>=10;

# Index performance

## Importance of indexes based

As seen during the lessons, the indexes can considerably improve the time requested by a query. And it is more important in our case when we come with big data. Indeed, some queries take several minutes to be executed. We can win significant amount of time. The indexes are a trade-off between time and space. More precisely, we use space to stock the indexes but we win time of execution. Thus we cannot add indexes for everything. We have to choose carefully the indexes that we need.

The Oracle Optimizer determines the most efficient query plan. It chooses the plan with the lowest cost. The plan query can help to choose indexes. We can see how the time is distributed and with this information we can see how and where we can win execution time.

## Query Plans

The selected queries are the queries : I, J and O.

Plan for I :

We add an index for the artist name because it must find all Metallica’s tracks.

Here is the plan before the indexes :

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Id | Operation | Rows | Bytes | Cost (%CPU)| | Time |
| 0 | SELECT STATEMENT | 25 | 525 | 1172 (4) | 00 :00 :15 |
| 1 | HASH UNIQUE | 25 | 525 | 1172 (4) | 00 :00 :15 |
| 2 | COUNT STOPKEY |  |  |  |  |
| 3 | VIEW | 47 | 987 | 1171 (4) | 00 :00 :15 |
| 4 | SORT ORDER BY STOPKEY | 47 | 3572 | 1171 (4) | 00 :00 :15 |
| 5 | HASH GROUP BY | 47 | 3572 | 1171 (4) | 00 :00 :15 |
| 6 | NESTED LOOPS | 47 | 3572 | 1169 (4) | 00 :00 :15 |
| 7 | NESTED LOOPS | 47 | 2303 | 1075 (4) | 00 :00 :13 |
| 8 | NESTED LOOPS | 50 | 1550 | 978 (4) | 00 :00 :12 |
| 9 | TABLE ACCESS FULL | 1 | 21 | 976 (4) | 00 :00 :12 |
| 10 | INDEX RANGE SCAN | 47 | 470 | 2 (0) | 00 :00 :01 |
| 11 | TABLE ACCESS BY INDEX  ROWID | 1 | 18 | 2 (0) | 00 :00 :01 |
| 12 | INDEX UNIQUE SCAN | 1 |  | 1 (0) | 00 :00 :01 |
| 13 | TABLE ACCESS BY INDEX  ROWID | 1 | 27 | 2 (0) | 00 :00 :01 |
| 14 | INDEX UNIQUE SCAN | 1 | 1 | 1 (0) | 00 :00 :01 |

Predicate Information (identified by operation id):

---------------------------------------------------

2 - filter(ROWNUM<=25)

4 - filter(ROWNUM<=25)

9 - filter("A"."NAME"='Metallica')

10 - access("A"."ID\_ARTIST"="AT"."ID\_ARTIST")

12 - access("AT"."ID\_TRACK"="T"."ID\_TRACK")

14 - access("T"."ID\_RECORDING"="R"."ID\_RECORDING")

Here is the plan of the query I after  the indexing:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Id | Operation | Rows | Bytes | Cost (%CPU)| | Time |
| 0 | SELECT STATEMENT | 25 | 525 | 201 (2) | 00 :00 :03 |
| 1 | HASH UNIQUE | 25 | 525 | 201 (2) | 00 :00 :03 |
| 2 | COUNT STOPKEY |  |  |  |  |
| 3 | VIEW | 47 | 987 | 200 (1) | 00 :00 :03 |
| 4 | SORT ORDER BY STOPKEY | 47 | 3572 | 200 (1) | 00 :00 :03 |
| 5 | HASH GROUP BY | 47 | 3572 | 200 (1) | 00 :00 :03 |
| 6 | NESTED LOOPS | 47 | 3572 | 198 (0) | 00 :00 :03 |
| 7 | NESTED LOOPS | 47 | 2303 | 104 (0) | 00 :00 :02 |
| 8 | NESTED LOOPS | 50 | 1550 | 7 (0) | 00 :00 :01 |
| 9 | TABLE ACCESS BY INDEX ROWID | 1 | 21 | 5 (0) | 00 :00 :01 |
| 10 | INDEX RANGE SCAN | 1 |  | 3 (0) | 00 :00 :01 |
| 11 | INDEX RANGE SCAN | 47 | 470 | 2 (0) | 00 :00 :01 |
| 12 | TABLE ACCESS BY INDEX  ROWID | 1 | 18 | 2 (0) | 00 :00 :01 |
| 13 | INDEX UNIQUE SCAN | 1 |  | 1 (0) | 00 :00 :01 |
| 14 | TABLE ACCESS BY INDEX  ROWID | 1 | 27 | 2 (0) | 00 :00 :01 |
| 15 | INDEX UNIQUE SCAN | 1 |  | 1 (0) | 00 :00 :01 |

Predicate Information (identified by operation id):

---------------------------------------------------

2 - filter(ROWNUM<=25)

4 - filter(ROWNUM<=25)

10 - access("A"."NAME"='Metallica')

11 - access("A"."ID\_ARTIST"="AT"."ID\_ARTIST")

13 - access("AT"."ID\_TRACK"="T"."ID\_TRACK")

15 - access("T"."ID\_RECORDING"="R"."ID\_RECORDING")

We can observe that the expected time and CPU use is really lower with the indexes.

For the queries O and J the improvement is not significant we cannot see it with the plan because it cannot use the index instead of doing full table scan.

Plan of query J :

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Id | Operation | Rows | Bytes | Cost (%CPU)| | Time |
| 0 | SELECT STATEMENT | 37800 | 18M | 3809 (4) | 00 :00 :46 |
| 1 | VIEW | 37800 | 18M | 3809 (4) | 00 :00 :46 |
| 2 | WINDOW SORT PUSHED RANK | 37800 | 2030K | 3809 (4) | 00 :00 :46 |
| 3 | SORT GROUP BY | 37800 | 2030K | 3809 (4) | 00 :00 :46 |
| 4 | NESTED LOOP | 37800 | 2030K | 2776 (4) | 00 :00 :34 |
| 5 | HASH JOIN | 797 | 35865 | 1179 (9) | 00 :00 :15 |
| 6 | HASH JOIN | 797 | 9564 | 191 (28) | 00 :00 :03 |
| 7 | VIEW | 10 | 40 | 115 (40) | 00 :00 :02 |
| 8 | COUNT STOPKEY |  |  |  |  |
| 9 | VIEW | 1725 | 6900 | 115 (40) | 00 :00 :02 |
| 10 | SORT ORDER BY STOPKEY | 1725 | 13800 | 115 (40) | 00 :00 :02 |
| 11 | SORT GROUP BY | 1725 | 13800 | 115 (40) | 00 :00 :02 |
| 12 | TABLE ACCESS FULL | 137K | 1074K | 74 (7) | 00 :00 :01 |
| 13 | TABLE ACCESS FULL | 137K | 1074K | 74 (7) | 00 :00 :01 |
| 14 | TABLE ACCESS FULL | 217K | 8759K | 982 (5) | 00 :00 :12 |
| 15 | INDEX RANGE SCAN | 47 | 470 | 2(0) | 00 :00 :01 |

Predicate Information (identified by operation id):

---------------------------------------------------

1 - filter("RN"=1)

2 - filter(ROW\_NUMBER() OVER ( PARTITION BY "AG"."ID\_GENRE" ORDER BY COUNT(DISTINCT

"AT"."ID\_TRACK") DESC )<=1)

5 - access("A"."ID\_ARTIST"="AG"."ID\_ARTIST")

6 - access("AG"."ID\_GENRE"="ID\_GENRE")

8 - filter(ROWNUM<=10)

10 - filter(ROWNUM<=10)

14 - filter("A"."GENDER"='Female')

15 - access("A"."ID\_ARTIST"="AT"."ID\_ARTIST")

The part that we would like to reduce is the full table access that is done on the artist\_track and artist tables. There is an index on the artist gender and on the artist\_genre for the id\_genre.

But the new query plan doesn’t select the new index. It may be that in this case the use of the index cost more. Even the query seems to be slightly slower. It may be because the optimizer must compute more possibilities than without indexes and finally choose the same plan than before.

Plan of query O :

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Id | Operation | Rows | Bytes | Cost (%CPU)| | Time |
| 0 | SELECT STATEMENT | 923K | 22M | 15844 (4) | 00 :03 :11 |
| 1 | VIEW | 923K | 22M | 7922 (4) | 00 :01 :36 |
| 2 | SORT ORDER BY | 923K | 10M | 7922 (4) | 00 :01 :36 |
| 3 | SORT GROUP BY NOSORT | 923K | 10M | 7922 (4) | 00 :01 :36 |
| 4 | MERGE JOIN | 1047K | 11M | 7922 (4) | 00 :01 :36 |
| 5 | INDEX FULL SCAN | 1236K | 7243K | 3667 (2) | 00 :00 :45 |
| 6 | SORT JOIN | 1044K | 6121K | 4255 (6) | 00 :00 :52 |
| 7 | TABLE ACCESS FULL | 1044K | 6121K | 716 (6) | 00 :00 :09 |
| 8 | SORT AGGREGATE | 1 | 13 |  |  |
| 9 | VIEW | 923K | 11M | 7922 (4) | 00 :01 :36 |
| 10 | SORT GROUP BY | 923K | 10M | 7922 (4) | 00 :01 :36 |
| 11 | MERGE JOIN | 1047K | 11M | 7922 (4) | 00 :01 :36 |
| 12 | INDEX FULL SCAN | 1236K | 7243K | 3667 (2) | 00 :00 :45 |
| 13 | SORT JOIN | 1044K | 6121K | 4255 (6) | 00 :00 :52 |
| 14 | TABLE ACCESS FULL | 1044K | 6121K | 716 (6) | 00 :00 :09 |

Predicate Information (identified by operation id):

---------------------------------------------------

1 - filter("COUNTM"= (SELECT MAX("COUNTM") FROM (SELECT COUNT(\*) "COUNTM" FROM

"MEDIUM" "M","RELEASE" "R" WHERE "M"."ID\_RELEASE"="R"."ID\_RELEASE" GROUP BY

"R"."ID\_RELEASE") "from$\_subquery$\_004"))

6 - access("R"."ID\_RELEASE"="M"."ID\_RELEASE")

filter("R"."ID\_RELEASE"="M"."ID\_RELEASE")

13 - access("M"."ID\_RELEASE"="R"."ID\_RELEASE")

filter("M"."ID\_RELEASE"="R"."ID\_RELEASE")

We must access the full medium table twice for the id\_release. Thus we add an index on the id\_release on the medium table but the new plan does not use it. We can suppose that the use of the index cost more.

## Running Time

The running time can depend of random factors like the number of people that use the server at the same time.

A : 5.026 seconds

B : no end

C : 15 seconds

D : 568 seconds

E : 0.567 seconds

F : no end

G : 838 seconds

H : no end

I : 30.157 seconds before added indexes, 0.4 seconds after.

J : 18.64 seconds before added indexes, 19.223 seconds (no better) after.

K : 2.09 seconds

L : 2162.6 seconds

M : no end

N : no end

O : 109.714 seconds before added indexes, 108 seconds after.

The query O was reimplemented in a more performed way : now the execution time is : 23 seconds

P : 2.59 seconds

Q : 249.877 seconds

R : 5175 seconds

S : no end

# User Interface

The user interface is one of the big part of the project. We had a lot of choices to make during the development of the interface.

This section will explain how the interface work and why we made those choices.

## Choices

First of all, we choose to develop the user interface as Web App because it's an easy way to design the interface we want.

## Search Functionality

We decided to implement the search functionality as follow:

1. We chose not to allow the user to search on any column of a table. The reason is we thought all the columns were not really interesting to run queries on. For example in the GENRE table there is a count column, but we believe this is not a useful query to be run for a lambda user. That’s why the only search that can be done on our UI is about the NAME column of the table.
2. We chose to allow the user only to search in table where there is a NAME column. The reason follow from the discussion in the first point previously mentioned. Thus the available tables for search are: AREA, ARTIST, GENRE, RECORDING and RELEASE. In the UI, this table selection can be done through a drop-down list (in order to not let the user write junk as a table name).
3. We construct our queries by using the LIKE operator of SQL using the keyword the user inputted. More formally the query was designed as follow:

"SELECT \* FROM "**+**table**+**" WHERE LOWER(name) LIKE lower('%" **+** keyword **+** "%')"

1. The resulting rows of the simple search query are displayed in a HTML table. The HTML columns are the SQL columns of the table that was searched without id related columns. For example if the search was made on the ARTIST table, the HTML columns are NAME, TYPE and GENDER.
2. For the follow up queries we decided that the user can have more infos about a row by clicking on it. We decided to do so since the resulting queries are made upon the id of the row and not about a specific column.
3. The follow up queries are made using the id of the clicked row on chosen tables that seemed logical to be linked to for us.

More precisely the tables are linked like that:

|  |  |
| --- | --- |
| **ID from table** | **Linked with table(s)** |
| AREA | ARTIST |
| ARTIST | AREA, GENRE, RELEASE |
| GENRE | ARTIST |
| RECORDING | ARTIST, RELEASE |
| RELEASE | ARTIST, RECORDING |

These choices were obvious for us since we thought of it as if we were a user of the UI and we talk between us about “What more infos would want the user x if he choose to search a keyword y?”

1. Concerning the displaying we chose to use tabs. We display one of the linked tables and then the user can click on the other tabs to go to the other linked tables. When the user click on one row of one of the linked table the same behaviour occurs as if these row resulted from a simple search query.

## Server sides

The server is a simple multi-threaded HTTP server that also accesses the EPFL's Oracle Database.

## Why Java

We choose to code it in Java because it's almost the only language we all know well. Another advantage is that it handles pretty well Oracle's database with the JDBC lib. Furthermore, Java is powered by Oracle too. It had to be well supported and it is.

## Web server

The web server is pretty simple. It just handles basic HTTP GET requests.

When the server is running, a listener is called. This listener wait for a HTTP request and, when a request is caught, it spawns a worker new thread. Then, it waits for the next request.

The Worker is a little bit more complicated.

First of all, the worker will handle the stream of data coming on the port 80. It will extract the path and GET parameters from the header.

Then, it will try to open the file gave in the path and stream it to the client in the response. The MIME type is automatically detected and sent in the HTTP response headers.

If the requested path is `/do-sql`, the behaviour will be different. The worker will try to extract a SQL query from the GET parameters. Then, it will ask the Database handler to execute the SQL query and to return a JSON with the result (or an error).

## Database Access

The database access is handled by a `Database` class.

It manages the connection and can execute some queries on the Oracle database. To achieve that, we use the JDBC driver.

After executing a query, the response from the Oracle database will be transformed in a JSON.

For instance, if you search an artist called "Flume", the response sent to the client will be:

```

{

"status":"OK",

"data":[

{

"NAME":"Liza Flume",

"GENDER":"Female",

"ID\_AREA":0,

"ID\_ARTIST":1057910,

"TYPE":"Person"

},

{

"NAME":"Flume",

"GENDER":"Male",

"ID\_AREA":13,

"ID\_ARTIST":835335,

"TYPE":"Person"

},

{

"NAME":"The Flumes",

"GENDER":"Other",

"ID\_AREA":0,

"ID\_ARTIST":837196,

"TYPE":"Other"

},

{

"NAME":"DJ Michael Flume",

"GENDER":"Other",

"ID\_AREA":0,

"ID\_ARTIST":195126,

"TYPE":"Other"

}

]

}

```

## Client Size

The client is a web app. So, to access it, you must first, run the java server, and then, access `http://localhost:7123/`

## How to use the Interface

The interface is pretty simple and user-friendly. This chapter contains a description of every features of the interface. And the next chapter contains some screenshots of the interface.

### Pre-written Requests

Pre-written queries are easily runnable by clicking on buttons on the left of the interface.

Once a button is clicked, the corresponding SQL query is sent to the server and a JSON is returned as a response. This response will be show as a table in the center of the user interface.

### Custom Search

The custom search allows you to search things with a case-insensitive keyword. For instance, if you search `beetroots`, the result `The Bloody Beetroots` will appear.

The request behind it is pretty simple:

`SELECT \* FROM artist WHERE LOWER(name) LIKE lower('%beetroots%')`.

Note that characters like `%` (percent), `\_` (underscore) and `'` (single quote) are escaped.

You can refer to the Search Functionality chapter for more details on the choices of this implementation.

### Clickable Results

When a non-empty result is displayed, each row can be clicked to show more information about the chosen result.

When the row is clicked, some tabs will appear to display new results.

You can refer to the Search Functionality chapter for more details

### Insertion

For each table displayed, an insertion form will appear. This allows you to add some more data in the database.

If a field is left empty, the value will be `null`.

Note that the line will not directly appear in the already displayed table for technical reason (the ID has to be defined by the database so the row can be clickable).

### Deletion

To delete a data, just click on the red cross on the right of the row.

### Full screen Mode

Above the results, on the right, a full screen button can be clicked to display results on a bigger view. This is pretty practical when the table has a lot of row.

### Technical Details

When a button is clicked, when a search is done or when a row is clicked, the same function is called to render the table. This function will execute an `AJAX` request to the server and get a JSON containing the result. The result will then be transformed to a `HTML` table.

Note that technical information like references to other SQL tables will be hidden to the user.

When errors occur, there are logged in the `Debug Console` at the bottom of the page. This is useful if you are disconnected from the EPFL network for instance.

## Interface Screenshots

Figure 15: simple search: the user can choose in which table he wants to do the search

Figure 16 SQL queries: the left table allows the user to select a query and displays the result

****

Figure 17: the user clicked on the row of an artist and obtained different complementary information. Different information are available using the tabs

Figure 18: Mode full screen of the results

Figure 19: the user executed a search on an artist

# Importation

We were confronted to some problems during the import part. The « real » data which we have at our disposal are far from the idealized schema that we made for the first part. The next chapter presents the ER-Model and the SQL code for the table creation. Some choices don’t seem logic but are adapted to the available data. An explanation is given for those choices.

# ER-Model

This schema fits best with the data structures..

## Constraints Explanation

There is some complex constraints between entities because the musical universe is also complex: collaborations (multiple artists per track), compilations (multiple artists per album), etc.

But because of the reality of the data (for instance information is missed). A lot of constraints cannot be strong. Some constraints had to be weakened by deleting some FOREIGN KEY. Indeed, some constraints that stand on the schema were finally not well adapted to the available data.

For example, in the artist table the FOREIGN KEY on the area ID had to be suppressed because some artists are associated with an area ID that does not exist in the area table. For the same reason the FOREIGN KEY on the artist ID in the artist\_genre table had to be suppressed, as well as the FOREIGN KEY on the track ID in the artist\_track table and the FOREIGN KEY on the Medium ID in the Track table.

### Areas

An artist can have at most one area. This constraint is given by the available data. Indeed, instinctively we may think that one artist have to come from one or more than one area (for example: New-York (*City*) and America (*Continent*)). But in the given data, the situation is different: an artist can have only one area or no area if the data is missing.

### Tracks

Each track is unique. It contains only one recording and is on one medium exactly.

### Recordings

A recording must be on a track but can be on several different tracks. It makes no sense to have a recording that is not on a track (a logical song need a support, so participate to the song relationship).

### Releases

Because some data is missed, release can be associated with one or zero medium.

## Table Creation

### Merge for participation constraints

To capture the constraints that an artist is associated with at most one area, the artist\_area table is merged with the artist table. The same is made for the physical\_song and the medium.

### SQL Code

**CREATE** **TABLE** Area(

ID\_Area INT,

Name VARCHAR2(1000),

**Type** VARCHAR2(60),

**PRIMARY** **KEY**(ID\_Area)

);

**CREATE** **TABLE** Genre

(ID\_Genre INT,

Name VARCHAR2(1000),

**Count** INT,

**PRIMARY** **KEY**(ID\_Genre));

**CREATE** **TABLE** Recording

(ID\_Recording INT,

Name VARCHAR2(2000),

**Length** INT,

**PRIMARY** **KEY**(ID\_Recording));

**CREATE** **TABLE** Release

(ID\_Release INT,

Name VARCHAR2(1000),

**PRIMARY** **KEY**(ID\_Release));

**CREATE** **TABLE** Artist

(ID\_Artist INT,

Name VARCHAR2(1000),

**Type** VARCHAR2(60),

Gender VARCHAR2(20),

ID\_Area INT,

**PRIMARY** **KEY**(ID\_Artist)

);

**CREATE** **TABLE** Medium

(ID\_Medium INT,

Format VARCHAR2(60),

ID\_Release INT,

**PRIMARY** **KEY**(ID\_Medium)

);

**CREATE** **TABLE** Track

(ID\_Track INT,

**Position** INT,

ID\_Medium INT,

ID\_Recording INT,

**PRIMARY** **KEY**(ID\_Track),

**FOREIGN** **KEY**(ID\_Recording) **REFERENCES** Recording

**ON DELETE CASCADE**);

**CREATE** **TABLE** Artist\_Genre

(ID\_Artist INT,

ID\_Genre INT,

**PRIMARY** **KEY**(ID\_Artist, ID\_Genre),

**FOREIGN** **KEY**(ID\_Genre) **REFERENCES** Genre

**ON DELETE CASCADE)**;

**CREATE** **TABLE** Artist\_Track

(ID\_Artist INT,

ID\_Track INT,

**PRIMARY** **KEY**(ID\_Artist, ID\_Track),

**FOREIGN** **KEY**(ID\_Artist) **REFERENCES** Artist

**ON DELETE CASCADE)**;