Practicum-2 Build a Database Application- Contact Tracer

Varshitha Uppula
Saisrihitha Yadlapalli
Jing Chen
(Group 3)

Modification made to the previous table to satisfy current requirements :

Added a new attribute *age* which is derived from the attribute *DOB* in the *naiveUser* table.

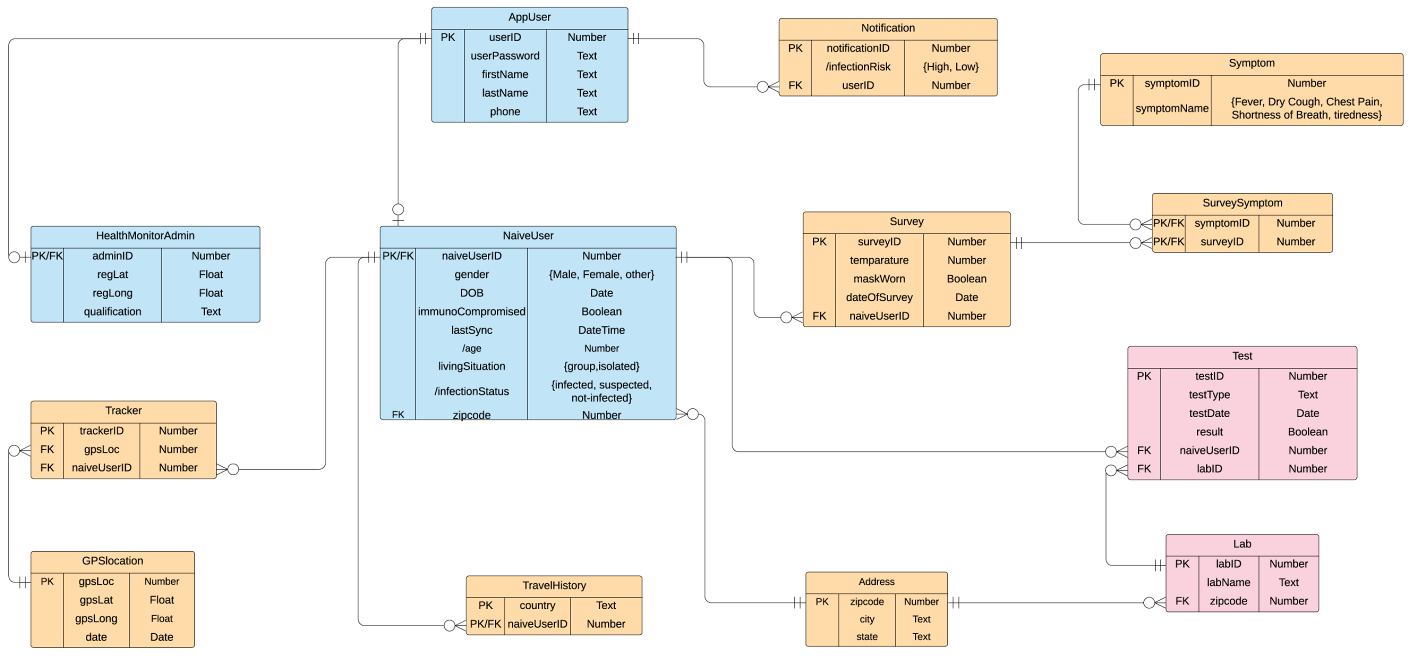


Fig : Logical Diagram

Link to Logical Model (https://app.lucidchart.com/invitations/accept/55d3e1a0-dc6e-4a04-8d6c-6d30657dec3e)

1. Create a view

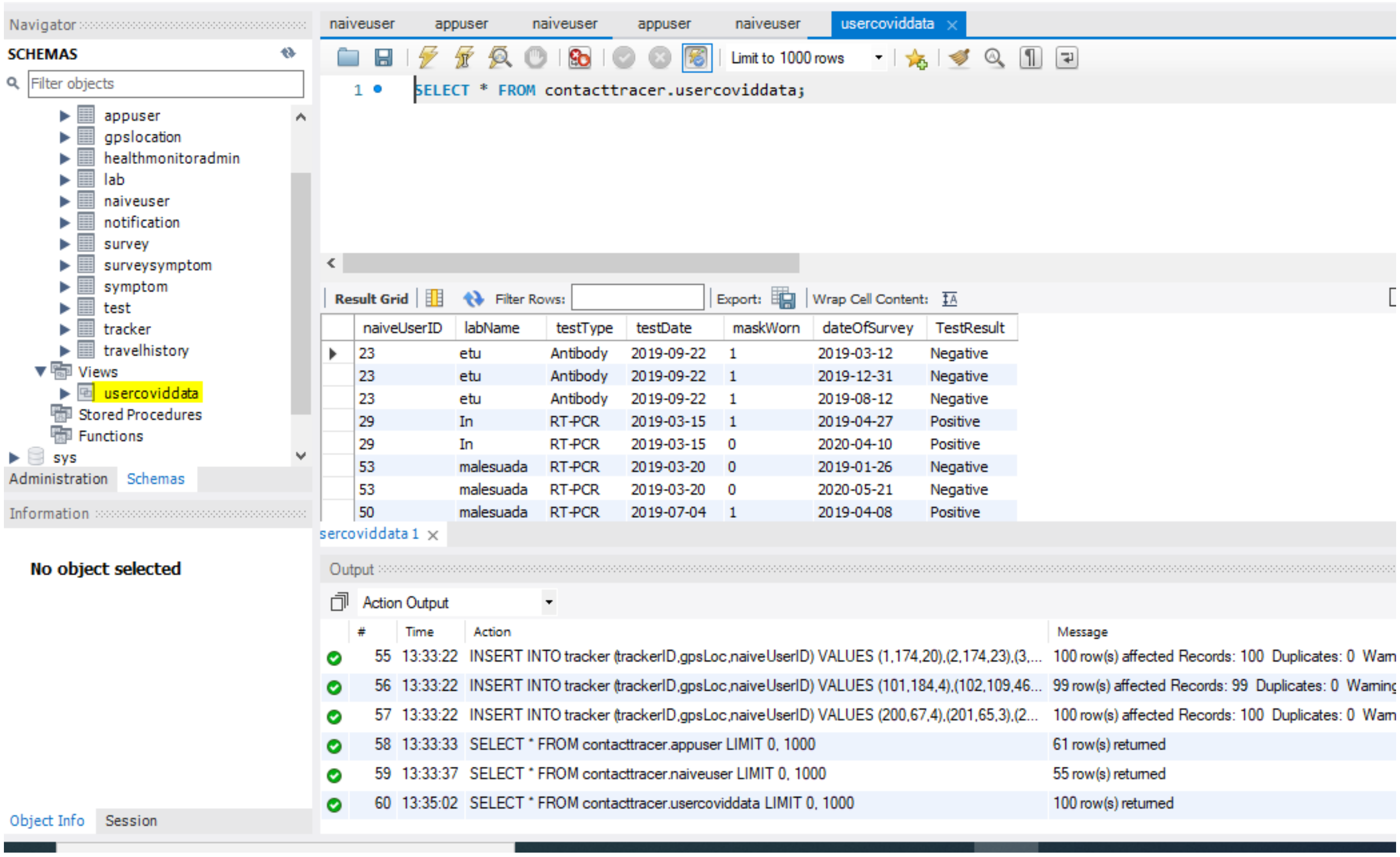
In SQL, a “view” is a virtual table. It lets you package a complex query into a single table and simplify reporting. Once the view is created it can be used in queries like any other table. Updates through views are possible in some situations but generally views are read-only database objects.
The view below is based on four tables and involves a join. Tables used for creating the view are naiveUser, test, survey and lab. It is created to help abstract a complex query.

```
dbGetQuery(contactTrace,"DROP View IF EXISTS UserCovidData;")
```

```
0 rows
```

```
query1=" CREATE VIEW UserCovidData
AS
SELECT naiveUser.naiveUserID,lab.labName, test.testType, test.testDate,survey.maskWorn,s
urvey.dateOfSurvey,
CASE WHEN test.result = FALSE THEN 'Negative'
ELSE 'Positive'
END AS TestResult
FROM naiveUser, test, lab, survey
WHERE lab.labID= test.labID AND
test.naiveUserID = naiveUser.naiveUserID AND
naiveuser.naiveUserID=survey.naiveUserId ;"

dbGetQuery(contactTrace,query1)
```

```
0 rows
```



```
SELECT * FROM contacttracer.usercoviddata;
```

naiveUserID	labName	testType	testDate	maskWorn	dateOfSurvey	TestResult
23	etu	Antibody	2019-09-22	1	2019-03-12	Negative
23	etu	Antibody	2019-09-22	1	2019-12-31	Negative
23	etu	Antibody	2019-09-22	1	2019-08-12	Negative
29	In	RT-PCR	2019-03-15	1	2019-04-27	Positive
29	In	RT-PCR	2019-03-15	0	2020-04-10	Positive
53	malesuada	RT-PCR	2019-03-20	0	2019-01-26	Negative
53	malesuada	RT-PCR	2019-03-20	0	2020-05-21	Negative
50	malesuada	RT-PCR	2019-07-04	1	2019-04-08	Positive

```
#   Time      Action                                                                        Message
55  13:33:22  INSERT INTO tracker (trackerID,gpsLoc,naiveUserID) VALUES (1,174,20),(2,174,23),(3,...   100 row(s) affected Records: 100  Duplicates: 0  Warn
56  13:33:22  INSERT INTO tracker (trackerID,gpsLoc,naiveUserID) VALUES (101,184,4),(102,109,46...    99 row(s) affected Records: 99  Duplicates: 0  Warning
57  13:33:22  INSERT INTO tracker (trackerID,gpsLoc,naiveUserID) VALUES (200,67,4),(201,65,3),(2...    100 row(s) affected Records: 100  Duplicates: 0  Warn
58  13:33:33  SELECT * FROM contacttracer.appuser LIMIT 0, 1000                              61 row(s) returned
59  13:33:37  SELECT * FROM contacttracer.naiveuser LIMIT 0, 1000                            55 row(s) returned
60  13:35:02  SELECT * FROM contacttracer.usercoviddata LIMIT 0, 1000                        100 row(s) returned
```

Fig : View created in MySQL when above statments are run

2. Query the created view

This query is performed on the view created above to find details of users who tested positive and did not wear a mask at least once in the past 10 days from when they tested positive.

We can see that all the data of the users who tested positive and have updated that they did not wear a mask at least once in the past 10 days is retrieved. For e.g, User 57 has tested positive on 2020-04-28 and has recorded that he did not wear a mask on 24-04-2020 and on 2020-04-22. This information can further be used to determine the risk of the contacts this patient has met.

```
query2<- "SELECT userCovidData.naiveUserID, userCovidData.testType, userCovidData.testDa
te, userCovidData.DateOfSurvey, testResult, maskWorn
 FROM contacttracer.usercoviddata
 WHERE userCovidData.dateOfSurvey >= DATE_SUB(userCovidData.testDate, INTERVAL 10 DAY) A
ND
userCovidData.dateOfSurvey <= userCovidData.testDate
AND userCovidData.TestResult = 'Positive'
AND userCovidData.maskWorn = FALSE;"

rs = dbSendQuery(contactTrace,query2)
result1<-dbFetch(rs)
result1
```

naiveUserID	testType	testDate	dateOfSurvey	TestResult	maskWorn
<int>	<chr>	<chr>	<chr>	<chr>	<int>
57	RT-PCR	2020-04-28	2020-04-24	Positive	0
58	RT-PCR	2020-02-23	2020-02-13	Positive	0
60	RT-PCR	2020-07-10	2020-07-07	Positive	0
61	Antibody	2020-02-12	2020-02-02	Positive	0
56	Antibody	2019-12-26	2019-12-24	Positive	0
57	RT-PCR	2020-04-28	2020-04-22	Positive	0
56	Antibody	2020-03-08	2020-03-01	Positive	0
56	RT-PCR	2020-03-09	2020-03-01	Positive	0

8 rows

3. Implement CRUD using R

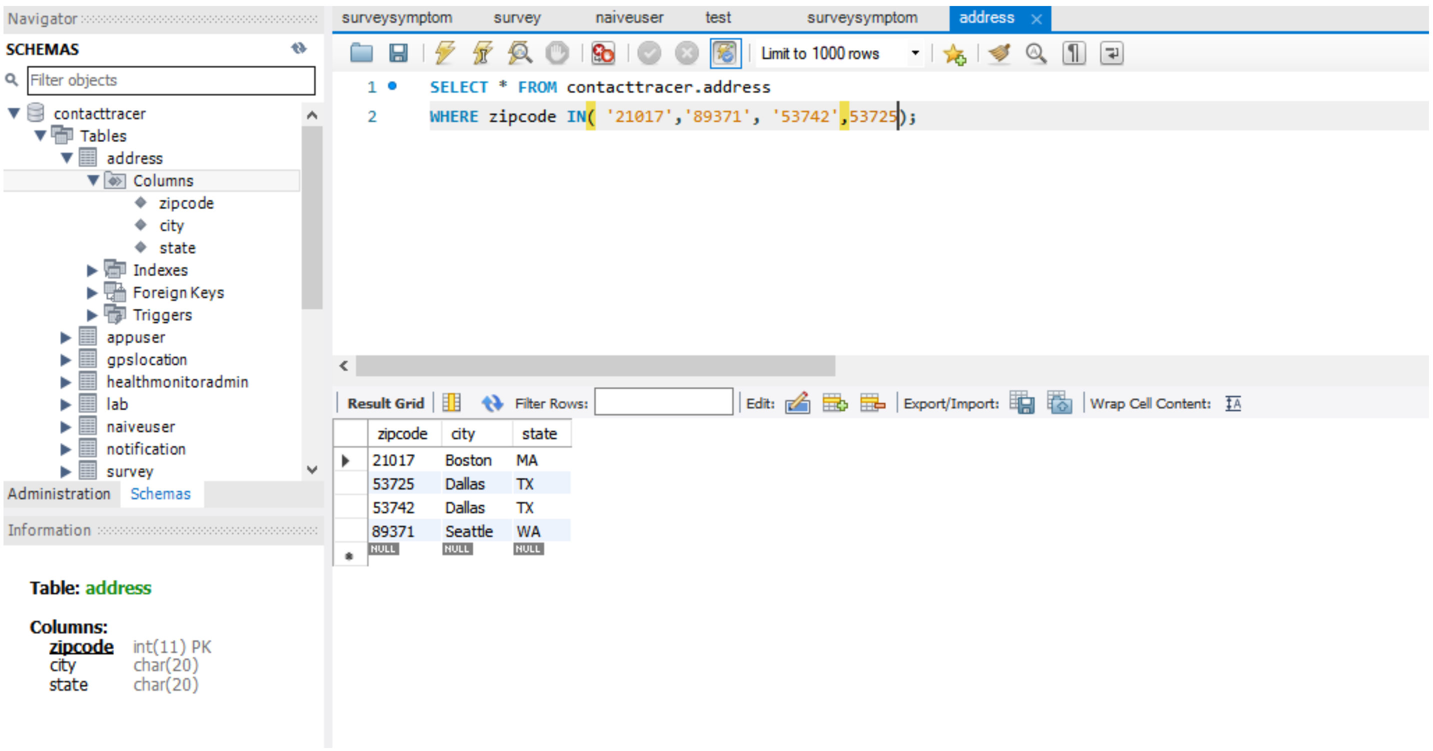
CRUD is an acronym for Create, Read, Update, and Delete. CRUD operations are basic data manipulation for the database. create (i.e. insert), read (i.e. select), update and delete operations are implemented below using functions in R

CREATE : Data is inserted into the address table

```
insertAddress <- function(addZip, addCity, addState) {

query3<-paste0("INSERT INTO address(zipcode,city,state)

VALUES (", addZip, "," ,"'",addCity,"'", "," ,"'",addState,"'" ,");")

dataInserted<- dbGetQuery(contactTrace,query3)

}
```

Call the function insertAddress() to insert data into the database.

```
insertAddress(21017,'Boston','MA')
insertAddress(89371,'Seattle','WA')
insertAddress(53742,'Dallas','TX')
insertAddress(53725,'Dallas','TX')
```



```
SELECT * FROM contacttracer.address
WHERE zipcode IN( '21017','89371', '53742',53725);
```

zipcode	city	state
21017	Boston	MA
53725	Dallas	TX
53742	Dallas	TX
89371	Seattle	WA
NULL	NULL	NULL

Table: address

Columns:

zipcode	int(11) PK
city	char(20)
state	char(20)

Fig : Data inserted into address table

RETRIEVE: Data is retrieved from the address table based on the zipcode passed

```
viewZip <- function(zip) {

query4 <- paste0("SELECT * FROM address WHERE zipCode =", zip)

print("Data Viewed")

rs<-dbGetQuery(contactTrace,query4)
rs
}
```

Call the function viewZip() to retrieve data and pass the value of a valid zipcode that exists

```
viewZip(21017)
```

```
## [1] "Data Viewed"
```

zipcode	city	state
<int>	<chr>	<chr>

zipcode <int>	city <chr>	state <chr>
21017	Boston	MA

1 row

UPDATE: The city in the address table is updated with a WHERE condition on the zipcode

```
updateAddress <- function(addrCity,addrZip) {

 query5 <- paste0("UPDATE address SET city =","'",addrCity,"'","WHERE zipcode =",addrZi
p)

 dbGetQuery(contactTrace,query5)
}
```

Call the function updateAddress to update the city of a given zipcode and then retrieve the data to check if the update happened successfully
Before Update

```
viewZip(21017)
```

```
## [1] "Data Viewed"
```

zipcode <int>	city <chr>	state <chr>
21017	Boston	MA

1 row

After Update -

```
updateAddress('Salem',21017)
```

0 rows

```
viewZip(21017)
```

```
## [1] "Data Viewed"
```

zipcode <int>	city <chr>	state <chr>
21017	Salem	MA

1 row

DELETE: Delete is performed on address table based on WHERE condition on zipcode

```
deleteAddress <- function(addrZip) {

query6 <- paste0("DELETE FROM address WHERE zipcode =",addrZip)

dbGetQuery(contactTrace,query6)
}
```

Before Delete

```
viewZip(53742)
```

```
## [1] "Data Viewed"
```

zipcode <int>	city <chr>	state <chr>
53742	Dallas	TX

1 row

After delete

```
deleteAddress(53742)
```

0 rows

```
viewZip(53742)
```

```
## [1] "Data Viewed"
```

0 rows

7. Transaction logic to support CRUD operations that span multiple tables. (Task 4 is after this)

Transaction logic to INSERT

A transaction logic to help achieve generalization is implemented here. Whenever an insert occurs in the parent class, an insert should also be made in either of the child classes.

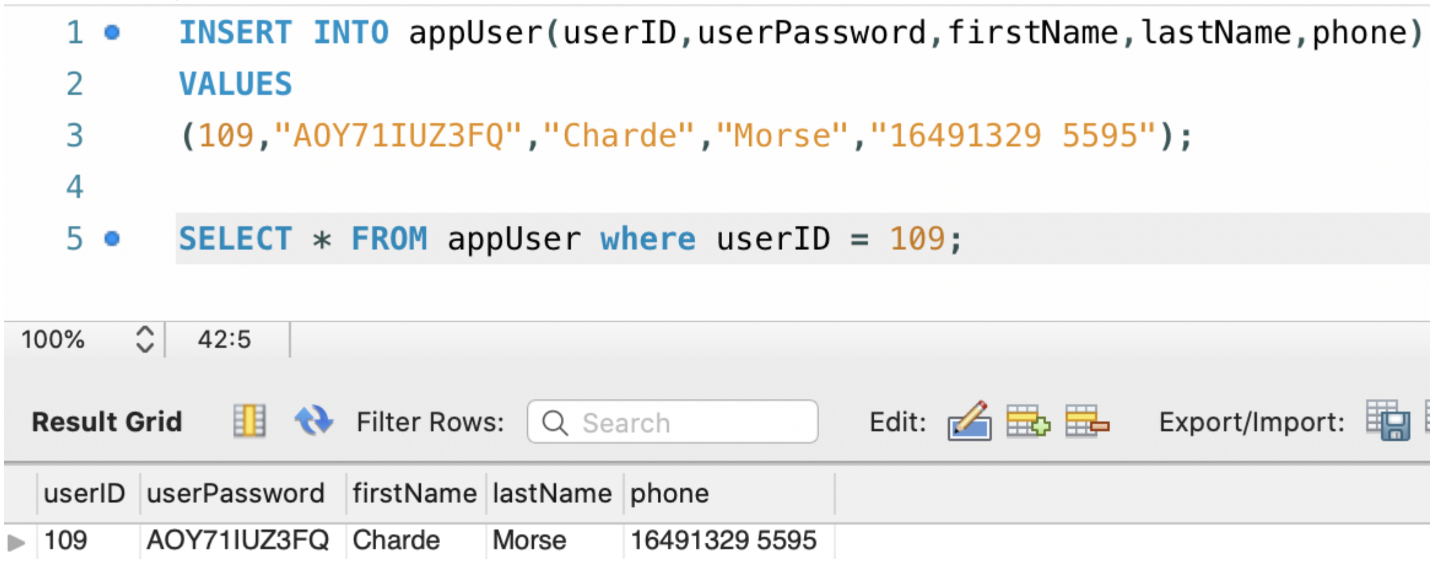
Code Explanation - We have a function which accepts the below parameters:

input : A single input which is used to determine if the transaction should run on healthMonitorAdmin and appUser or if it needs to run on naiveUser and appUser.
valuesAppUser: Vector with values that need to be inserted into the appUser table
valuesHealthAdmin:Vector with values that need to be inserted into the healthMonitorAdmin table
valuesNaiveUser:Vector with values that need to be inserted into the naiveUser table

Since the code runs as a transaction it will ensure that the parent class cannot exist by itself and the userID of the parent is inserted as the naiveUserID or adminID of the child class as well.

```
addUser<-function(input,valuesappUser, valuesHealthAdmin, valuesNaiveUser){
    if(input=="healthMonitorAdmin"){

        dbBegin(contactTrace)

        query<-paste0("INSERT INTO appUser(userID, userPassword, firstName, lastName,pho
ne)
                      VALUES (",valuesappUser[1],",'", valuesappUser[2],"','",valuesappU
ser[3],"','",valuesappUser[4],"','",valuesappUser[5],"');")
        dbGetQuery(contactTrace,query)
        query2<-paste0("INSERT INTO healthMonitorAdmin(adminID, regLat, regLong, qualifi
cation)
            VALUES (",valuesappUser[1],",'", valuesHealthAdmin[1],"','",valuesHealthAdmi
n[2],"','",valuesHealthAdmin[3],"');")
        dbGetQuery(contactTrace,query2)

        dbCommit(contactTrace)
        if(dbCommit(contactTrace) == FALSE)
        {
          dbRollback(contactTrace)
          print("Transaction has been rolled back")
        }
        else{
             print("insert for health monitor admin comitted")
        }
    }
else {
        dbBegin(contactTrace)
        query3<-paste0("INSERT INTO appUser(userID, userPassword, firstName, lastName,ph
one)
                      VALUES (",valuesappUser[1],",'", valuesappUser[2],"','",valuesappU
ser[3],"','",valuesappUser[4],"','",valuesappUser[5],"');")
        dbGetQuery(contactTrace,query3)
        query4<-paste0("INSERT INTO naiveUser(naiveUserID, gender, dob, immunoCompromise
d,lastSync,livingSituation,infectionStatus,zipcode)
             VALUES (",valuesappUser[1],",'", valuesNaiveUser[1],"','",valuesNaiveUser[2
],"',",valuesNaiveUser[3],",'",valuesNaiveUser[4],"','",valuesNaiveUser[5],"','",valuesN
aiveUser[6],"','",valuesNaiveUser[7],"');")
        dbGetQuery(contactTrace,query4)
         dbCommit(contactTrace)
         if(dbCommit(contactTrace) == FALSE)
        {
          dbRollback(contactTrace)
          print("Transaction has been rolled back")
        }
        else{
             print("insert for naive user comitted")
        }
    }
}
```

If insert happens without a transaction, we can see that an entry can exist in appUser even though the corresponding entry does not exist in naiveUser or healthMonitorAdmin.

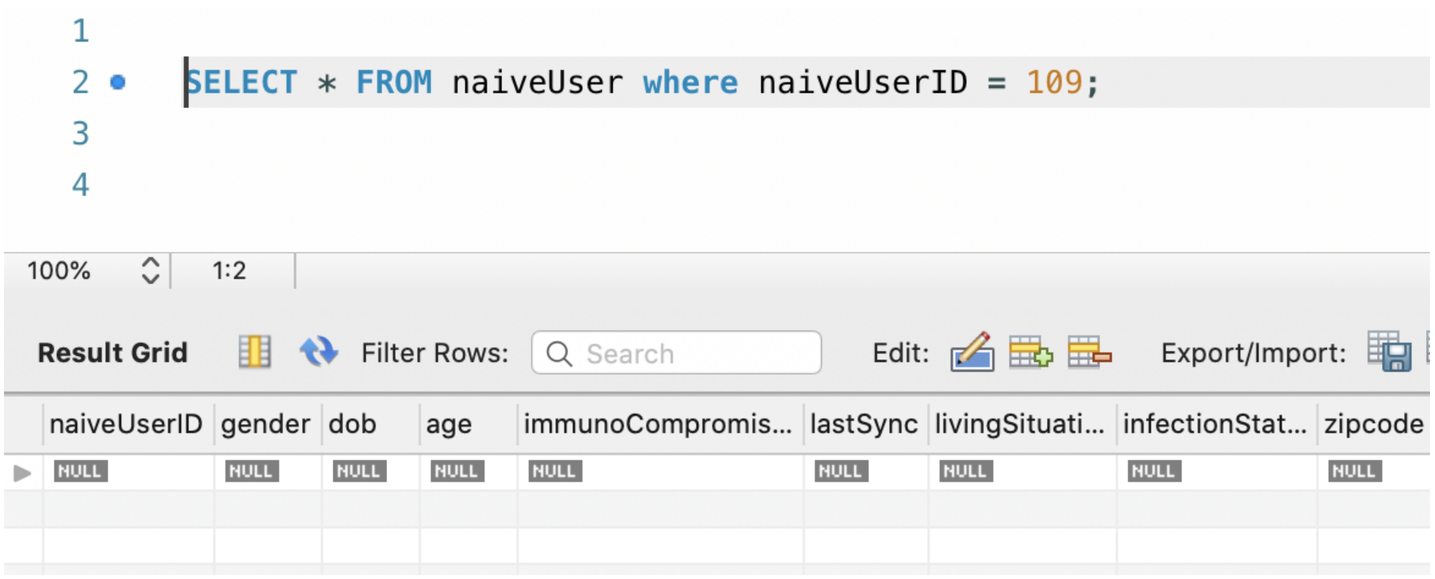


```sql
INSERT INTO appUser(userID,userPassword,firstName,lastName,phone)
VALUES
(109,"AOY71IUZ3FQ","Charde","Morse","16491329 5595");

SELECT * FROM appUser where userID = 109;
```

100% 42:5

Result Grid Filter Rows: Search Edit: Export/Import:

userID	userPassword	firstName	lastName	phone
109	AOY71IUZ3FQ	Charde	Morse	16491329 5595

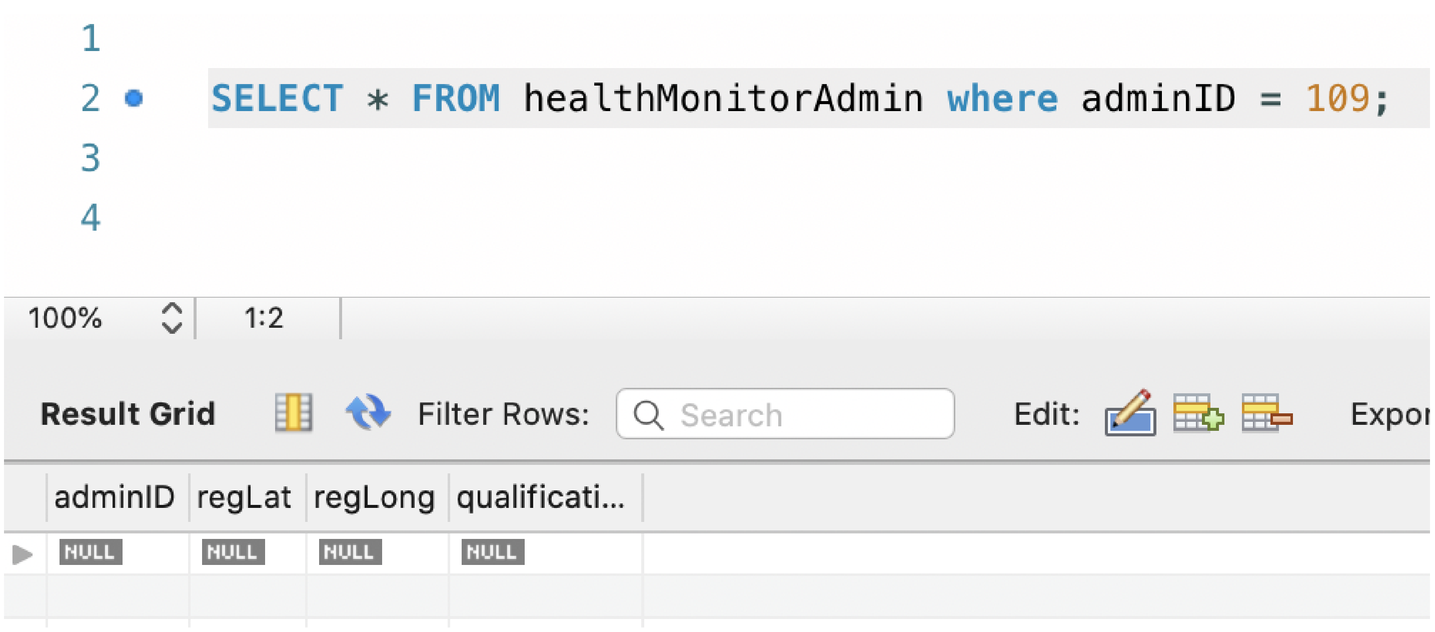


```sql
SELECT * FROM naiveUser where naiveUserID = 109;
```

100% 1:2

Result Grid Filter Rows: Search Edit: Export/Import:

naiveUserID	gender	dob	age	immunoCompromis...	lastSync	livingSituati...	infectionStat...	zipcode
NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL



```sql
SELECT * FROM healthMonitorAdmin where adminID = 109;
```

100% 1:2

Result Grid Filter Rows: Search Edit: Expor

adminID	regLat	regLong	qualificati...
NULL	NULL	NULL	NULL

If we use the above transaction logic for insert, we need to insert into both the appUser and either of the child classes simultaneously.

If we insert using transaction, we cannot perform insert only on appUser, the transaction will commit only if both the inserts into parent and child happen correctly.
Check transaction when input passed to the function is healthMonitorAdmin such that insert into healthMonitorAdmin and appUser occurs simultaneously.

```
input<-"healthMonitorAdmin"
valuesappUser<-c(112,"wergeqe","Varshitha","uppgs","15257655 5418")
valuesHealthAdmin <- c("75.31306","158.84824", "MD")
addUser(input,valuesappUser,valuesHealthAdmin,'')
```

```
## [1] "insert for health monitor admin comitted"
```

View the inserted data for healthMonitorAdmin and appUser

```
dbGetQuery(contactTrace,"SELECT * FROM appUser
WHERE userID = 112")
```

userID <int>	userPassword <chr>	firstName <chr>	lastName <chr>	phone <chr>
112	wergeqe	Varshitha	uppgs	15257655 5418

1 row

```
dbGetQuery(contactTrace,"SELECT * FROM healthMonitorAdmin
WHERE adminID = 112")
```

adminID <int>	regLat <dbl>	regLong <dbl>	qualification <chr>
112	75.31306	158.8482	MD

1 row

Check transaction when input passed to the function is naiveUser such that insert into naiveUser and appUser occurs simultaneously.

```
input<-"naiveUser"
valuesappUser<-c(66,"ehrjsfhj","Emerson","thajore","14140312 9162")
valuesNaiveUser<- c("Female","1970/01/13",TRUE,"2020-06-01 22:16:29","isolated","suspect
ed","46833")
addUser(input,valuesappUser,'',valuesNaiveUser)
```

```
## [1] "insert for naive user comitted"
```

View the inserted data for naiverUser and appUser

```
dbGetQuery(contactTrace,"SELECT * FROM appUser
WHERE userID = 66")
```

userID <int>	userPassword <chr>	firstName <chr>	lastName <chr>	phone <chr>
66	ehrjsfhj	Emerson	thajore	14140312 9162

1 row

```
dbGetQuery(contactTrace,"SELECT * FROM naiveUser
WHERE naiveuserID = 66")
```

naiveUserID <int>	gen... <chr>	dob <chr>	... <int>	immunoCompromi... <int>	lastSync <chr>	livingSituation <chr>
66	Female	1970-01-13	50	1	2020-06-01 22:16:29	isolated

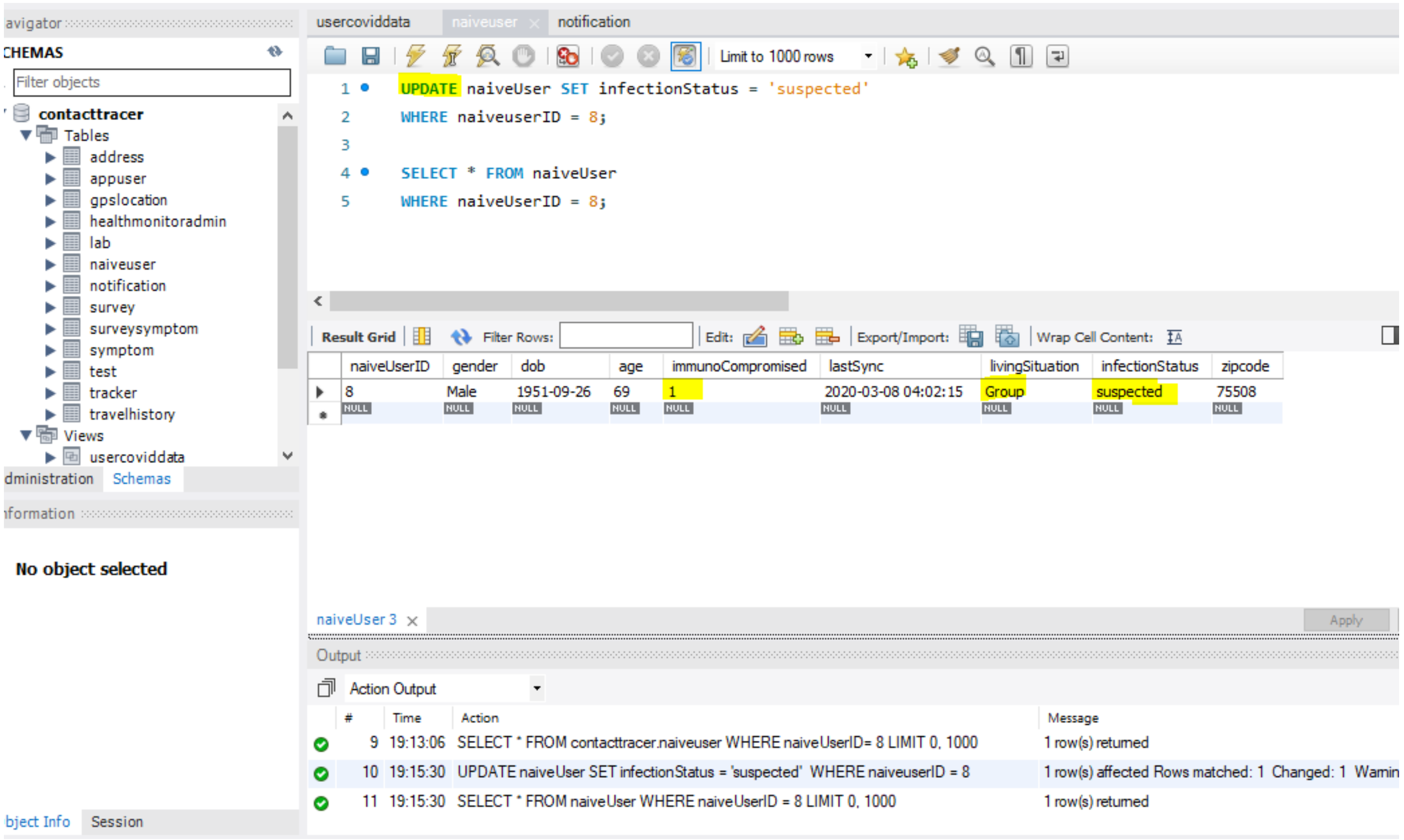
1 row | 1-7 of 9 columns

Transaction logic to UPDATE

The below transaction ensures that we update risk of a user in the notification table if we make an update to the infectionStatus in the naiveUser table.

```
updateTrans<- function(naiveUserID,infectStatus,risk){
    dbBegin(contactTrace)
    query1<-paste0("UPDATE naiveUser
SET naiveUser.infectionStatus =","'",infectStatus,"'","
WHERE naiveUserID = ", naiveUserID," AND immunoCompromised = TRUE and livingSituation=
 'group';")
    dbGetQuery(contactTrace,query1)
query2<- paste0(" UPDATE notification
SET infectionRisk = ","'",risk,"'","
WHERE userID =",naiveUserID,";")
 dbGetQuery(contactTrace,query2)
 dbCommit(contactTrace)
 if(dbCommit(contactTrace) == FALSE)
        {
          dbRollback(contactTrace)
          print("Transaction has been rolled back")
        }
        else{
              print("update committed")
        }
}
```

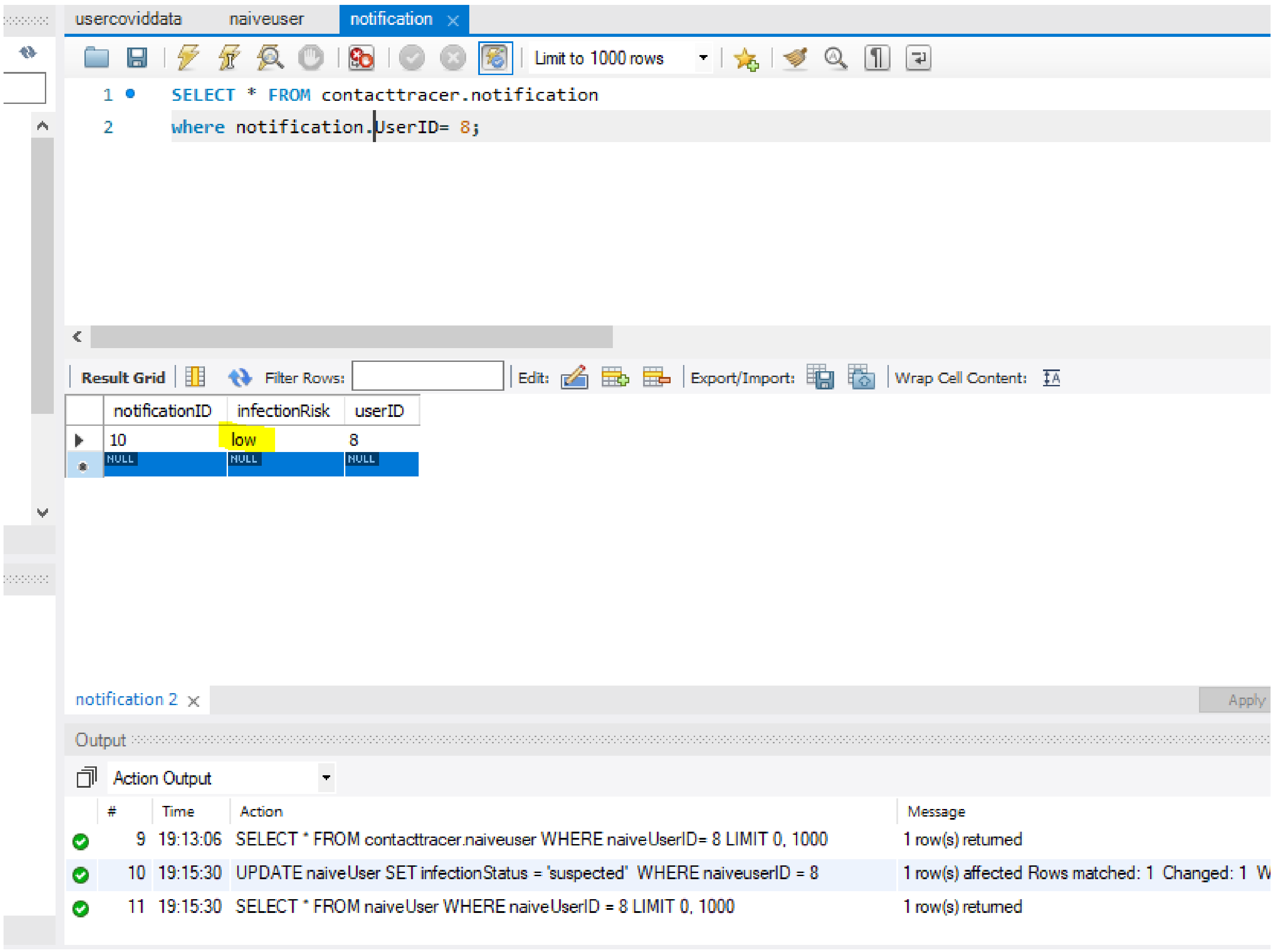
If we perform an update without this transaction, then a person's whose infection status is set as 'suspected' when he is immunocompromised and lives in a group but the risk of the person in the notification table might show inconsistent old values as its not updated simultaneously.



```
UPDATE naiveUser SET infectionStatus = 'suspected'
WHERE naiveuserID = 8;

SELECT * FROM naiveUser
WHERE naiveUserID = 8;
```

naiveUserID	gender	dob	age	immunoCompromised	lastSync	livingSituation	infectionStatus	zipcode
8	Male	1951-09-26	69	1	2020-03-08 04:02:15	Group	suspected	75508
NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

```
#   Time      Action                                                              Message
9   19:13:06  SELECT * FROM contacttracer.naiveuser WHERE naiveUserID= 8 LIMIT 0, 1000   1 row(s) returned
10  19:15:30  UPDATE naiveUser SET infectionStatus = 'suspected'  WHERE naiveuserID = 8   1 row(s) affected Rows matched: 1  Changed: 1  Warnin
11  19:15:30  SELECT * FROM naiveUser WHERE naiveUserID = 8 LIMIT 0, 1000                1 row(s) returned
```



```
SELECT * FROM contacttracer.notification
where notification.UserID= 8;
```

notificationID	infectionRisk	userID
10	low	8
NULL	NULL	NULL

```
#   Time      Action                                                              Message
9   19:13:06  SELECT * FROM contacttracer.naiveuser WHERE naiveUserID= 8 LIMIT 0, 1000   1 row(s) returned
10  19:15:30  UPDATE naiveUser SET infectionStatus = 'suspected'  WHERE naiveuserID = 8   1 row(s) affected Rows matched: 1  Changed: 1  W
11  19:15:30  SELECT * FROM naiveUser WHERE naiveUserID = 8 LIMIT 0, 1000                1 row(s) returned
```

But if we run the transaction, we can ensure that the risk of the person is up-to-date and consistent as and when the infection status is modified.

Testing:

```
updateTrans(8,'suspected','High' )
```

```
## [1] "update committed"
```

The transaction ensures that both the risk and infection status of a user are updated so that there is no inconsistency in our database.

```
dbGetQuery(contactTrace,"SELECT * FROM naiveUser
WHERE naiveUserID = 8")
```

naiveUserID <int>	gen… <chr>	dob <chr>	… <int>	immunoCompromi… <int>	lastSync <chr>	livingSituation <chr>
8	Male	1951-09-26	69	0	2020-03-08 04:02:15	Group

1 row | 1-7 of 9 columns

```
dbGetQuery(contactTrace,"SELECT * FROM notification
WHERE UserID = 8")
```

notificationID <int>	infectionRisk <chr>	userID <int>
10	High	8

1 row

Transaction logic to RETRIEVE

The transaction on select/retrieve operation ensures that both the infection status and test result that are present in different tables are executed such that there is no inconsistency. For e.g, if SELECT is performed on the infection status but some other user updates the test results, the transaction logic ensures that this cannot occur.

```
selectTransaction<-function(ID){
dbBegin(contactTrace)
query1<-paste0(
"SELECT naiveUserID, infectionStatus FROM naiveUser
WHERE naiveUserID=",ID)
query2<-paste0(
"SELECT testID, result as 'Test Result' FROM test
WHERE naiveUserID=",ID)
df1<-dbGetQuery(contactTrace,query1)
df2<-dbGetQuery(contactTrace,query2)
newlist <- list(df1,df2)
return(newlist)
dbCommit(contactTrace)
if(dbCommit(contactTrace) == FALSE)
        {
          dbRollback(contactTrace)
          print("Transaction has been rolled back")
        }
        else{
             print("retrieve transaction comitted")
        }
}
```

Call the function to view the transaction result

```
selectTransaction(8)
```

```
## [[1]]
##   naiveUserID infectionStatus
## 1           8    not-infected
##
## [[2]]
##   testID Test Result
## 1     38           0
```

Transaction logic to DELETE

This transaction is designed for enforcing generalizaion. An appUser is either a naiveUser or healthMonitorAdmin. Therefore, when deleting records from naiveUser table or heathMonitorAdmin table, we need to make sure that the records that have the same primary key are also deleted. The function has two inputs: ID and type. If entered type is “naiveUser”, it will delete the tuple with the entered ID from naiveUser and appUser tables. If entered type is “healthMonitorAdmin”, it will delete the tuple with the entered ID from healthMonitorAdmin and appUser.

```
deleteUser<-function(ID,type){
if(type=="naiveUser"){
dbBegin(contactTrace)
query1<-paste0(
"DELETE naiveUser FROM naiveUser
WHERE naiveUserID=",ID," ;")
query2<-paste0(
"DELETE appUser FROM appUser
WHERE userID=",ID," ;")
dbGetQuery(contactTrace,query1)
dbGetQuery(contactTrace,query2)
dbCommit(contactTrace)
if(dbCommit(contactTrace) == FALSE)
        {
          dbRollback(contactTrace)
          print("Transaction has been rolled back")
        }
        else{
              print("delete comitted")
        }}
else{
dbBegin(contactTrace)
query3<-paste0(
"DELETE healthMonitorAdmin FROM healthMonitorAdmin
WHERE adminID=",ID," ;")
query4<-paste0(
"DELETE appUser FROM appUser
WHERE userID=",ID," ;")
dbGetQuery(contactTrace,query3)
dbGetQuery(contactTrace,query4)
dbCommit(contactTrace)
if(dbCommit(contactTrace) == FALSE)
        {
          dbRollback(contactTrace)
          print("Transaction has been rolled back")
        }
        else{
              print("delete comitted")
        }
}
}
```

Before deletion

```
dbGetQuery(contactTrace,"Select * FROM naiveUser
           WHERE naiveUser.naiveUserID=3;")
```

naiveUserID <int>	gen… <chr>	dob <chr>	… <int>	immunoCompromi… <int>	lastSync <chr>	livingSituation <chr>
3	Male	1971-06-13	49	1	2020-03-28 03:06:18	Group

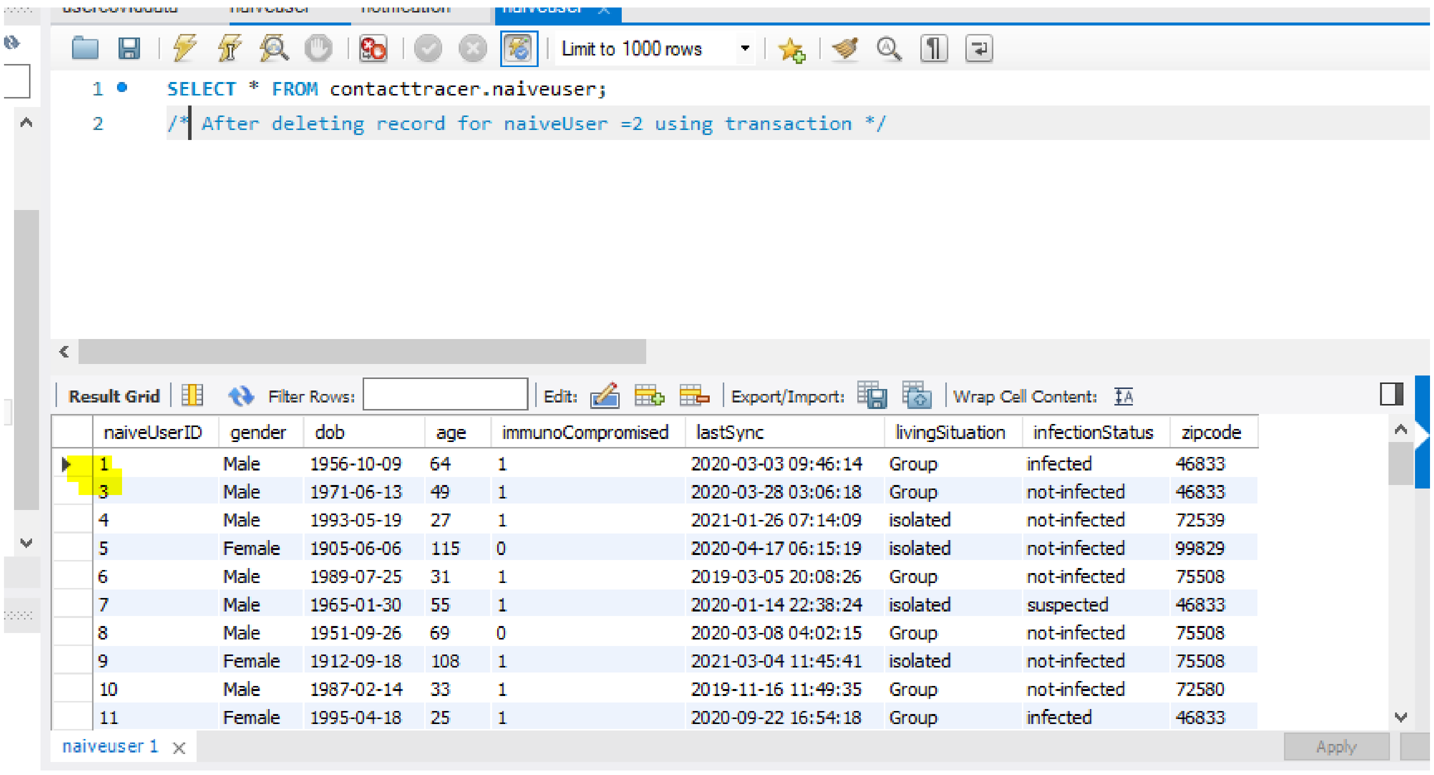
1 row | 1-7 of 9 columns

```
dbGetQuery(contactTrace,"Select * FROM appUser
           WHERE appUser.userID=3;")
```

userID <int>	userPassword <chr>	firstName <chr>	lastName <chr>	phone <chr>
3	XPO09FJL5JT	McKenzie	Hatfield	16640323 7123

1 row

After deletion



```
SELECT * FROM contacttracer.naiveuser;
/* After deleting record for naiveUser =2 using transaction */
```

naiveUserID	gender	dob	age	immunoCompromised	lastSync	livingSituation	infectionStatus	zipcode
1	Male	1956-10-09	64	1	2020-03-03 09:46:14	Group	infected	46833
3	Male	1971-06-13	49	1	2020-03-28 03:06:18	Group	not-infected	46833
4	Male	1993-05-19	27	1	2021-01-26 07:14:09	isolated	not-infected	72539
5	Female	1905-06-06	115	0	2020-04-17 06:15:19	isolated	not-infected	99829
6	Male	1989-07-25	31	1	2019-03-05 20:08:26	Group	not-infected	75508
7	Male	1965-01-30	55	1	2020-01-14 22:38:24	isolated	suspected	46833
8	Male	1951-09-26	69	0	2020-03-08 04:02:15	Group	not-infected	75508
9	Female	1912-09-18	108	1	2021-03-04 11:45:41	isolated	not-infected	75508
10	Male	1987-02-14	33	1	2019-11-16 11:49:35	Group	not-infected	72580
11	Female	1995-04-18	25	1	2020-09-22 16:54:18	Group	infected	46833

Fig : naiveUser table after deletion using transaction

```
deleteUser(3,"naiveUser")
```

```
## [1] "delete comitted"
```

```
dbGetQuery(contactTrace,"Select * FROM naiveUser
           WHERE naiveUser.naiveUserID=3;")
```

0 rows

```
dbGetQuery(contactTrace,"Select * FROM appUser
           WHERE appUser.userID=3;")
```

0 rows

4.TRIGGER to handle data integrity

TRIGGER 1

A trigger called TR_naiveUserDOB_beforeInsert is created on the table naiveUser to ensure that the date of birth cannot be greater than the current date. An alert is sent to the user prompting them to enter the right date of birth. This trigger is run before INSERT operation is performed on the table naiveUser.

```
dbGetQuery(contactTrace,"DROP TRIGGER IF EXISTS TR_naiveUserDOB_beforeInsert;")
```

```
0 rows
```

```
dbGetQuery(contactTrace,"
    CREATE TRIGGER TR_naiveUserDOB_beforeInsert
    BEFORE INSERT ON naiveuser
    FOR EACH ROW
    BEGIN
        IF (NEW.dob > CURDATE()) THEN
        SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Date of birth cannot be after curren
t date, please enter a valid Date of birth';
        END IF;
    END")
```

```
0 rows
```

Verifying that the trigger is working correctly by trying to insert a date of birth that is greater than current date using the function created in transaction logic to insert data.

```
input<-"naiveUser"
valuesappUser<-c(64,"rji5TJS3UA","Kendall","thajore","15110314 9162")
valuesNaiveUser<- c("Female","2021/01/13",TRUE,"2020-06-01 22:16:29","isolated","suspect
ed","46833")
addUser(input,valuesappUser,'',valuesNaiveUser)
```

```
Error in .local(conn, statement, ...) : could not run statement: date of birth cannot be after current date    Show Traceback
```

TRIGGER 2

A trigger called TR_naiveUserDOB_beforeUpdate is created on the Table naiveUser to ensure that the date of birth cannot be greater than the current date when an update is performed on the table. An alert is sent to the user prompting them to enter the right date of birth. This trigger is run before UPDATE operation is performed on the table naiveUser.

```
dbGetQuery(contactTrace,"DROP TRIGGER IF EXISTS TR_naiveUserDOB_beforeUpdate;")
dbGetQuery(contactTrace,"

CREATE TRIGGER TR_naiveUserDOB_beforeUpdate

BEFORE UPDATE ON naiveuser

FOR EACH ROW

BEGIN

  IF (NEW.dob > CURDATE()) THEN

    SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'date of birth cannot be after current da
te';

  END IF;

END")
```

Verifying that the trigger is working correctly by trying to update date of birth with a date that is greater than current date.

```
dbGetQuery(contactTrace,"UPDATE naiveUser
SET naiveUser.DOB = '2021/08/14'
WHERE naiveUserID= 1;")
```

```
Error in .local(conn, statement, ...) : could not run statement: date of birth cannot be
 after current date
```
Show Traceback

5. TRIGGER for derived attribute

Trigger to calculate derived attribute ‘age’ from date of birth when new record is inserted into the naiveuser table

```
dbGetQuery(contactTrace,"DROP TRIGGER IF EXISTS TR_naiveUserAge_beforeINSERT;")
```

```
0 rows
```

```
dbGetQuery(contactTrace,"
CREATE TRIGGER TR_naiveUserAge_beforeINSERT
BEFORE INSERT
ON naiveUser FOR EACH ROW
BEGIN
SET NEW.age = YEAR(CURDATE()) - YEAR(NEW.dob);
END")
```

```
0 rows
```

Pass the details of the NaiveUser date of birth in the insert statement.

```
input<-"naiveUser"
valuesappUser<-c(67,"AOY71uaZ3Fa","Sneha","Ravi","14410329 6594")
valuesNaiveUser<- c("Female","1978/04/02",TRUE,"2020-03-03 09:46:14","group","infected",
"53725")
addUser(input,valuesappUser,'',valuesNaiveUser)
```

```
## [1] "insert for naive user comitted"
```

Testing trigger to ensure that the data is created correctly i.e calculation of derived attribute age.

```
dbGetQuery(contactTrace," SELECT * FROM naiveUser WHERE naiveUserID = 67 ")
```

naiveUserID <int>	gen... <chr>	dob <chr>	... <int>	immunoCompromi... <int>	lastSync <chr>	livingSituation <chr>
67	Female	1978-04-02	42	1	2020-03-03 09:46:14	Group

1 row | 1-7 of 9 columns

Trigger to update the age if any modification is done to the date of birth:

```
dbGetQuery(contactTrace,"DROP TRIGGER IF EXISTS TR_naiveUserAge_beforeUpdate;")
```

```
0 rows
```

```
dbGetQuery(contactTrace,"
CREATE TRIGGER TR_naiveUserAge_beforeUpdate
BEFORE UPDATE
ON naiveUser FOR EACH ROW
BEGIN
SET NEW.age = YEAR(CURDATE()) - YEAR(NEW.dob);
END")
```

```
0 rows
```

Testing trigger to ensure that the data is created correctly

Before insert

```
dbGetQuery(contactTrace," SELECT * FROM naiveUser WHERE naiveUserID = 67 ")
```

naiveUserID <int>	gen... <chr>	dob <chr>	... <int>	immunoCompromi... <int>	lastSync <chr>	livingSituation <chr>
67	Female	1978-04-02	42	1	2020-03-03 09:46:14	Group

1 row | 1-7 of 9 columns

Update the date of birth for the user from 1985-04-02 to 1997-04-02

```
dbGetQuery(contactTrace," UPDATE naiveUser SET dob = '1997/04/02'  WHERE naiveUserID = 6
7 ")
```

```
0 rows
```

```
dbGetQuery(contactTrace," SELECT * FROM naiveUser WHERE naiveUserID = 67 ")
```

```
naiveUserID  gen...   dob         ...  immunoCompromi...  lastSync             livingSituation
      <int>  <chr>    <chr>       <int>            <int>  <chr>                <chr>
         67  Female   1997-04-02  23                   1  2020-03-03 09:46:14  Group
1 row | 1-7 of 9 columns
```

6. STORED PROCEDURE

A stored procedure is a group of SQL statements that form a logical unit and perform a particular task, and they are used to encapsulate a set of operations or queries to execute on a database server. Here we created a stored procedure to calculate the infection risk of a person based on various factors from other tables.

Infection status is calculated as below:
Infected : If the most recent test conducted on a naiveUser shows result as positive, then the user's infection status is set as Infected.
Not-Infected: If the most recent test of a person shows negative then the persons status is set as not-infected.
Suspected: If a user shows 2 or more symptoms then he is considered as a suspect.

```
 df <- dbGetQuery(contactTrace, "DROP PROCEDURE IF EXISTS infectionstatus_update;" )

storedP <- "

Create Procedure infectionstatus_update(IN ID INT)



    BEGIN



    UPDATE naiveUser



    INNER JOIN test ON naiveUser.naiveUserID = test.naiveUserID



    SET naiveUser.infectionStatus = 'infected'



    WHERE naiveUser.naiveUserID=ID AND test.result = TRUE AND (testDate IN (SELECT max(t
estDate) FROM test WHERE test.naiveUserID=ID));



    UPDATE naiveUser



    INNER JOIN test ON naiveUser.naiveUserID = test.naiveUserID



    SET naiveUser.infectionStatus = 'not-infected'



    WHERE naiveUser.naiveUserID=ID AND test.result = FALSE AND (testDate IN (SELECT max
(testDate) FROM test WHERE test.naiveUserID=ID));



    UPDATE naiveUser



    LEFT JOIN test ON naiveUser.naiveUserID=test.naiveUserID
    INNER JOIN (Select naiveUserID, survey.dateOfSurvey as dates, count(symptomID) as sy
mptomCount



        from survey



        INNER JOIN surveySymptom ON surveySymptom.surveyID=survey.surveyID



        group by survey.naiveUserID, survey.surveyID, survey.dateOfSurvey) total_count O
N naiveUser.naiveUserID=total_count.naiveUserID



    SET naiveUser.infectionStatus = 'suspected'



    WHERE naiveUser.naiveUserID=ID AND test.result IS NULL AND total_count.symptomCount>
=2 AND (dates IN (SELECT max(dateOfSurvey) FROM survey WHERE survey.naiveUserID=ID));



    END"
dbGetQuery(contactTrace,storedP)
```

```
0 rows
```

Testing if the stored procedure works as expected:
The view is created to show the results

```
dbGetQuery(contactTrace,"DROP VIEW IF exists infection_test;")
```

```
0 rows
```

```
dbGetQuery(contactTrace,"
           CREATE VIEW infection_test AS SELECT naiveUser.naiveUserID as naiveUserID, na
iveUser.infectionStatus, test.result as 'test result', test.testDate,
              survey.dateOfSurvey as dateOfSurvey, count(*) as 'Number of Symptoms' FROM
 naiveUser
 LEFT JOIN test
 ON naiveUser.naiveUserID = test.naiveUserID
 LEFT JOIN survey
 ON naiveuser.naiveUserID= survey.naiveUserID
 JOIN surveySymptom ON survey.surveyID = surveySymptom.surveyID
 WHERE naiveuser.naiveUserID in (53,30)
 Group by naiveUser.naiveUserID, test.result, survey.dateOfSurvey, test.testDate;
 ")
```

```
0 rows
```

Before running stored procedure

```
dbGetQuery(contactTrace,"SELECT * from infection_test;")
```

naiveUserID <int>	infectionStatus <chr>	test result <int>	testDate <chr>	dateOfSurvey <chr>	Number of Symptoms <dbl>
30	not-infected	NA	NA	2019-01-10	3
53	not-infected	1	2019-03-20	2019-01-26	1
53	not-infected	0	2019-01-04	2019-01-26	1
53	not-infected	1	2019-03-20	2020-05-21	3
53	not-infected	0	2019-01-04	2020-05-21	3

5 rows

1) check if infection status is set as infected if test is positive after running the stored procedure

We can see that the infection status of user 53 on running the stored procedure has become infected based on the test result of the most recent test date

```
dbGetQuery(contactTrace, "CALL infectionstatus_update(53);" )
```

```
0 rows
```

```
dbGetQuery(contactTrace,"SELECT * from infection_test;")
```

naiveUserID <int>	infectionStatus <chr>	test result <int>	testDate <chr>	dateOfSurvey <chr>	Number of Symptoms <dbl>
30	not-infected	NA	NA	2019-01-10	3
53	infected	1	2019-03-20	2019-01-26	1

naiveUserID <int>	infectionStatus <chr>	test result <int>	testDate <chr>	dateOfSurvey <chr>	Number of Symptoms <dbl>
53	infected	0	2019-01-04	2019-01-26	1
53	infected	1	2019-03-20	2020-05-21	3
53	infected	0	2019-01-04	2020-05-21	3

5 rows

Add a new entry into above user with test result as negative

```
dbGetQuery(contactTrace,"INSERT into test(testID, testType, testDate, result, naiveUserI
D, labID)
VALUES(39,'Antibody','2020/05/20',FALSE,53,5);")
```

```
0 rows
```

```
dbGetQuery(contactTrace,"SELECT * from infection_test;")
```

naiveUserID <int>	infectionStatus <chr>	test result <int>	testDate <chr>	dateOfSurvey <chr>	Number of Symptoms <dbl>
30	not-infected	NA	NA	2019-01-10	3
53	infected	1	2019-03-20	2019-01-26	1
53	infected	0	2019-01-04	2019-01-26	1
53	infected	0	2020-05-20	2019-01-26	1
53	infected	1	2019-03-20	2020-05-21	3
53	infected	0	2019-01-04	2020-05-21	3
53	infected	0	2020-05-20	2020-05-21	3

7 rows

2) check if infection status is reset as not-infected

```
dbGetQuery(contactTrace, "CALL infectionstatus_update(53);" )
```

```
0 rows
```

```
dbGetQuery(contactTrace,"SELECT * from infection_test;")
```

naiveUserID <int>	infectionStatus <chr>	test result <int>	testDate <chr>	dateOfSurvey <chr>	Number of Symptoms <dbl>
30	not-infected	NA	NA	2019-01-10	3

naiveUserID <int>	infectionStatus <chr>	test result <int>	testDate <chr>	dateOfSurvey <chr>	Number of Symptoms <dbl>
53	not-infected	1	2019-03-20	2019-01-26	1
53	not-infected	0	2019-01-04	2019-01-26	1
53	not-infected	0	2020-05-20	2019-01-26	1
53	not-infected	1	2019-03-20	2020-05-21	3
53	not-infected	0	2019-01-04	2020-05-21	3
53	not-infected	0	2020-05-20	2020-05-21	3

7 rows

3) Check if infection status is set as suspected for a user who shows two or more symptoms

```
dbGetQuery(contactTrace, "CALL infectionstatus_update(30);" )
```

```
0 rows
```

```
dbGetQuery(contactTrace, "SELECT * from infection_test;" )
```

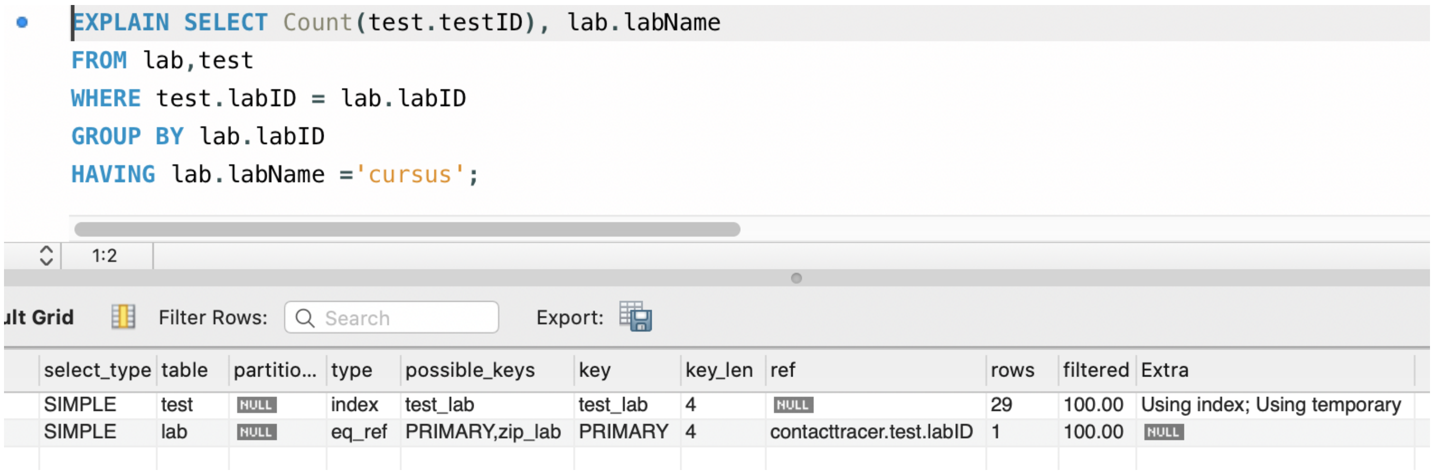
naiveUserID <int>	infectionStatus <chr>	test result <int>	testDate <chr>	dateOfSurvey <chr>	Number of Symptoms <dbl>
30	suspected	NA	NA	2019-01-10	3
53	not-infected	1	2019-03-20	2019-01-26	1
53	not-infected	0	2019-01-04	2019-01-26	1
53	not-infected	0	2020-05-20	2019-01-26	1
53	not-infected	1	2019-03-20	2020-05-21	3
53	not-infected	0	2019-01-04	2020-05-21	3
53	not-infected	0	2020-05-20	2020-05-21	3

7 rows

8. Query plan

Query alternative 1

EXPLAIN SELECT Count(test.testID), lab.labName
FROM lab,test
WHERE test.labID = lab.labID
GROUP BY lab.labID
HAVING lab.labName =‘cursus’;



```
EXPLAIN SELECT Count(test.testID), lab.labName
FROM lab,test
WHERE test.labID = lab.labID
GROUP BY lab.labID
HAVING lab.labName ='cursus';
```

1:2

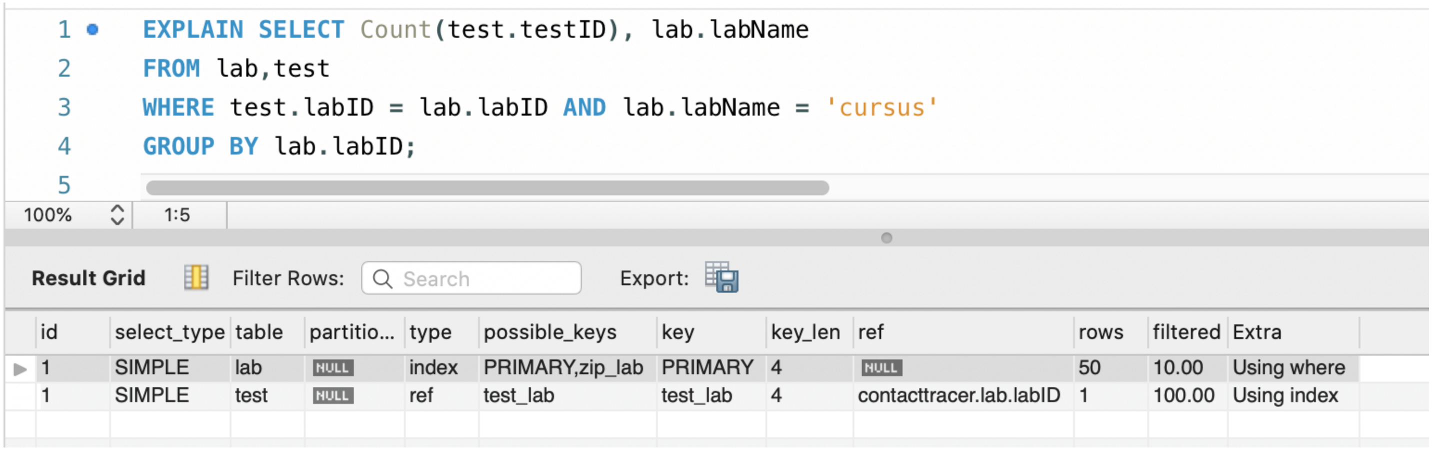
ult Grid Filter Rows: Search Export:

select_type	table	partitio...	type	possible_keys	key	key_len	ref	rows	filtered	Extra
SIMPLE	test	NULL	index	test_lab	test_lab	4	NULL	29	100.00	Using index; Using temporary
SIMPLE	lab	NULL	eq_ref	PRIMARY,zip_lab	PRIMARY	4	contacttracer.test.labID	1	100.00	NULL

Fig : Query evaluation of query1 using EXPLAIN

Query alternative 2

EXPLAIN SELECT Count(test.testID), lab.labName
FROM lab,test
WHERE test.labID = lab.labID AND lab.labName = ‘cursus’
GROUP BY lab.labID;



```
EXPLAIN SELECT Count(test.testID), lab.labName
FROM lab,test
WHERE test.labID = lab.labID AND lab.labName = 'cursus'
GROUP BY lab.labID;
```

100% 1:5

Result Grid Filter Rows: Search Export:

id	select_type	table	partitio...	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	lab	NULL	index	PRIMARY,zip_lab	PRIMARY	4	NULL	50	10.00	Using where
1	SIMPLE	test	NULL	ref	test_lab	test_lab	4	contacttracer.lab.labID	1	100.00	Using index

Fig : Query evaluation of query2 using EXPLAIN

Each of the rows in the EXPLAIN contains the following fields:

id - In most cases, the id field will present a sequential number of the SELECT query this row belongs to. The queries above contains no subqueries nor unions, so therefore the id for both rows is 1, as there is actually only 1 query.

select_type - The type of SELECT query. In our case, both are SIMPLE queries as they contain no subqueries or unions.

table - the table name

type - defines how the tables are accessed / joined.

possible_keys - The optional indexes MySQL can choose from, to look up for rows in the table. Some of the indexes in this list can be actually irrelevant, as a result of the execution order MySQL chose. In general, MySQL can use indexes to join tables. Said that, it won’t use an index on the first table’s join column, as it will go through all of its rows anyway (except rows filtered by the WHERE clause).

key - This column indicates the actual index MySQL decided to use.

key_len - This is one of the important columns in the explain output. It indicates the length of the key that MySQL decided to use, in bytes. In the EXPLAIN outputs above, MySQL uses the entire PRIMARY index (4 bytes). Unfortunately, there is no easier way to figure out which part of the index is used by MySQL, other than aggregating the length of all columns in the index and comparing that to the key_len value.

rows - Indicates the number of rows MySQL believes it must examine from this table, to execute the query. This is only an estimation. Usually, high row counts mean there is room for query optimization.

filtered - The filtered column indicates an estimated percentage of table rows that will be filtered by the table condition. Rows × filtered / 100 shows the number of rows that will be joined with previous tables.

Comparing the plans for the above two queries using the Visual explain plan

The Visual Explain feature generates and displays a visual representation of the MySQL EXPLAIN statement by using extended information available in the extended JSON format.

Visual explain plan for query 1

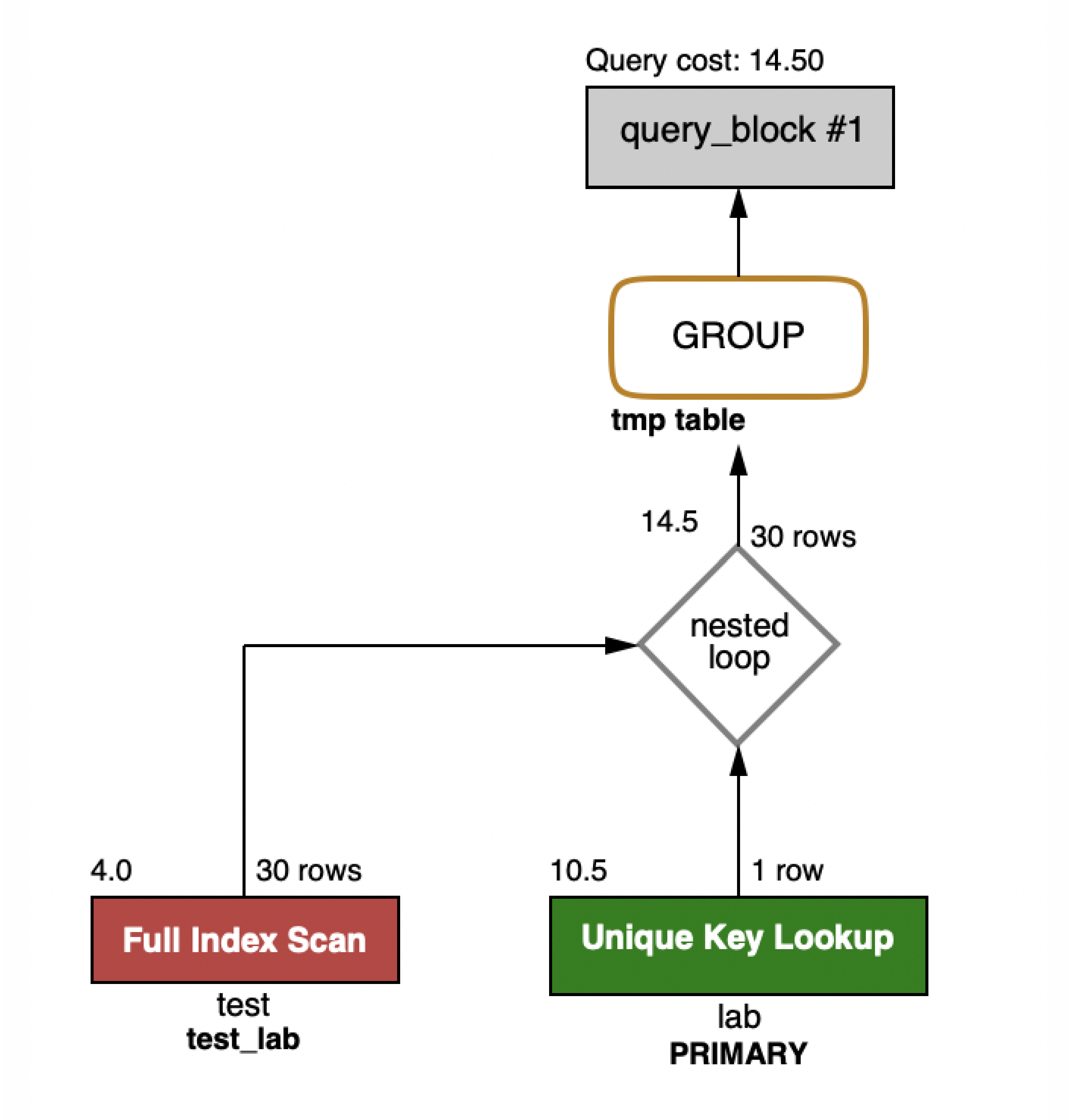


Fig : Query evaluation of query1 Visually

For the query alternative 1, MySQL believes it must examine 30 rows from this table to execute the query. Among these 30 rows, 100% of them would be filtered by the table condition using index. For query 1, the full test table index scan is estimated to costs 4 milliseconds and Unique Key Lookup, which finds an index that can be used to retrieve the required records, is estimated to cost 10.5 milliseconds. The estimated query cost is 14.5 milliseconds.

Visual explain plan for query 2

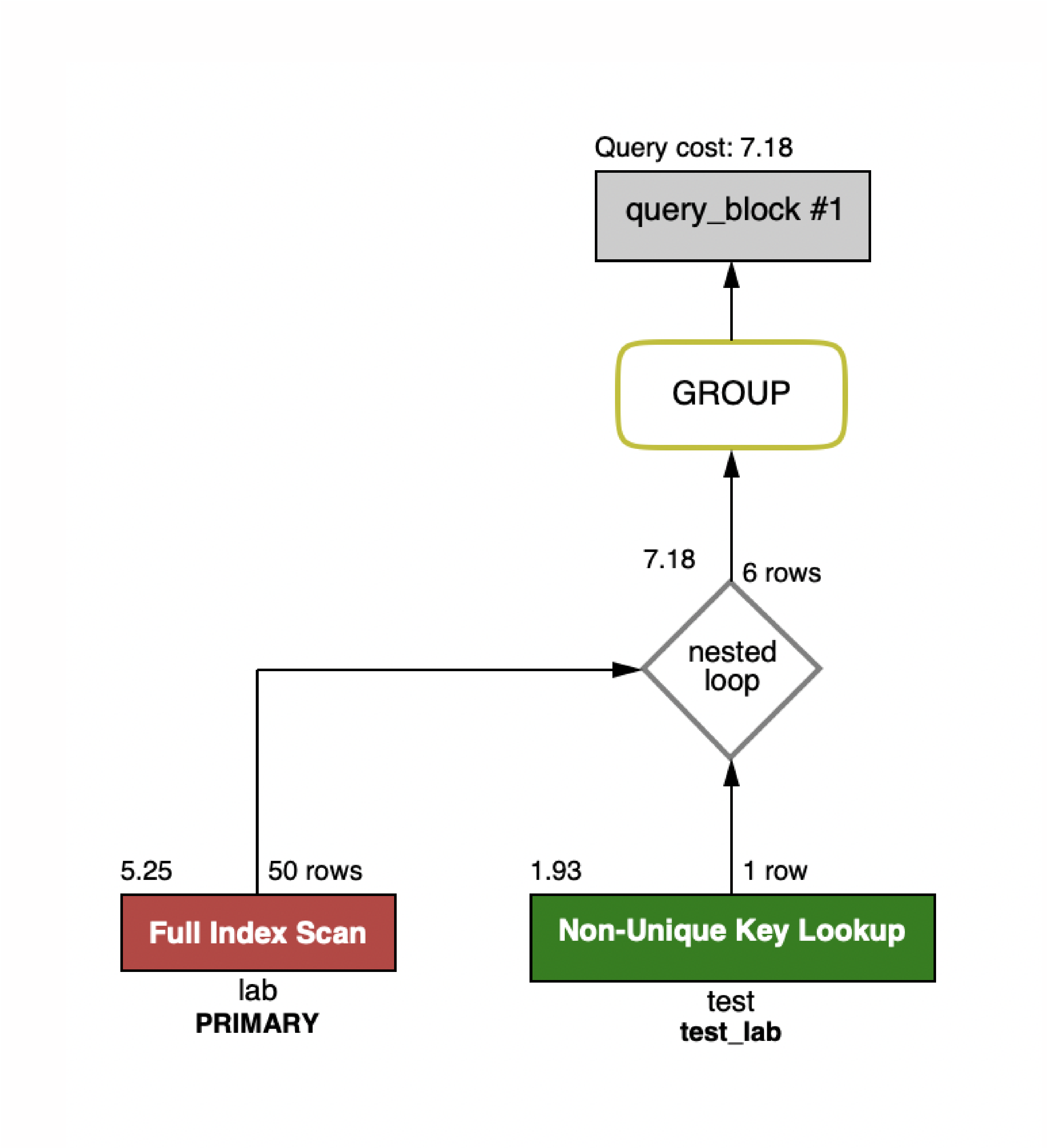


Fig : Query evaluation of query2 Visually

In query 2, the Where clause acts as a pre filter where as Having as a post filter. MySQL believes it must examine 50 rows from this table to execute the query. Among these 50 rows, 10% of them would be filtered by Where statement. The full table index scan for lab table with 50 rows is estimated to costs 5.25 milliseconds and The unique key lookup in the test table is estimated to cost only 1.93 milliseconds. In total, the estimated query cost is 7.18 milliseconds for query 2.

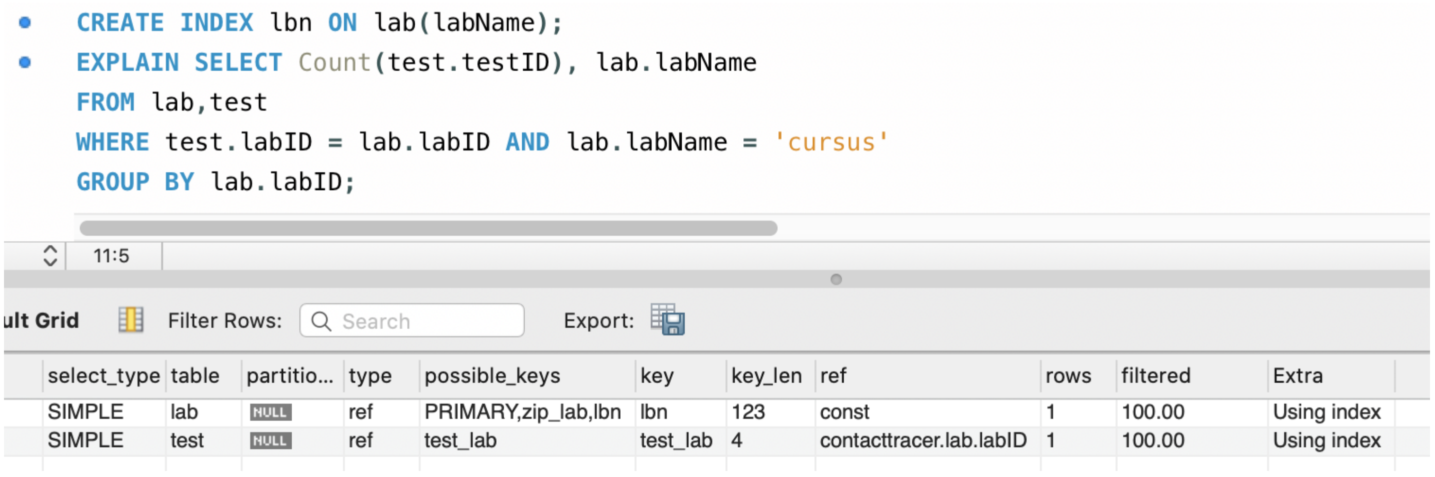
Comparision:
If filtering can be done without aggregate function (HAVING) then it must be done using the WHERE clause as it improves the performance since counting and sorting will be done on a much smaller set. If the same rows are filtered after grouping, you unnecessarily bear the cost of sorting, which is not being used. Hence, even with 20 more rows query 2 has a lower query cost than query one, thereby increasing the performance and is more optimized. **Therefore we have chosen query 2 over query 1.**

Note: WHERE restricts the result set before returning rows and HAVING restricts the result set after bringing all the rows. Therefore, WHERE is faster. MySQL for example, applies HAVING almost last in the chain, meaning there is almost no room for optimization so one needs to avoid the HAVING clause, whenever possible as it filters the selected tuples only after all the tuples have been fetched.

9. Index is created on labName column for query 2.

Query alternative 2 with index

CREATE INDEX lbn ON lab(labName);
SELECT Count(test.testID), lab.labName
FROM lab,test
WHERE test.labID = lab.labID
AND lab.labName = ‘cursus’
GROUP BY lab.labID;



```
CREATE INDEX lbn ON lab(labName);
EXPLAIN SELECT Count(test.testID), lab.labName
FROM lab,test
WHERE test.labID = lab.labID AND lab.labName = 'cursus'
GROUP BY lab.labID;
```

select_type	table	partitio...	type	possible_keys	key	key_len	ref	rows	filtered	Extra
SIMPLE	lab	NULL	ref	PRIMARY,zip_lab,lbn	lbn	123	const	1	100.00	Using index
SIMPLE	test	NULL	ref	test_lab	test_lab	4	contacttracer.lab.labID	1	100.00	Using index

Fig : Query evaluation of query with index

Visual explain plan with index

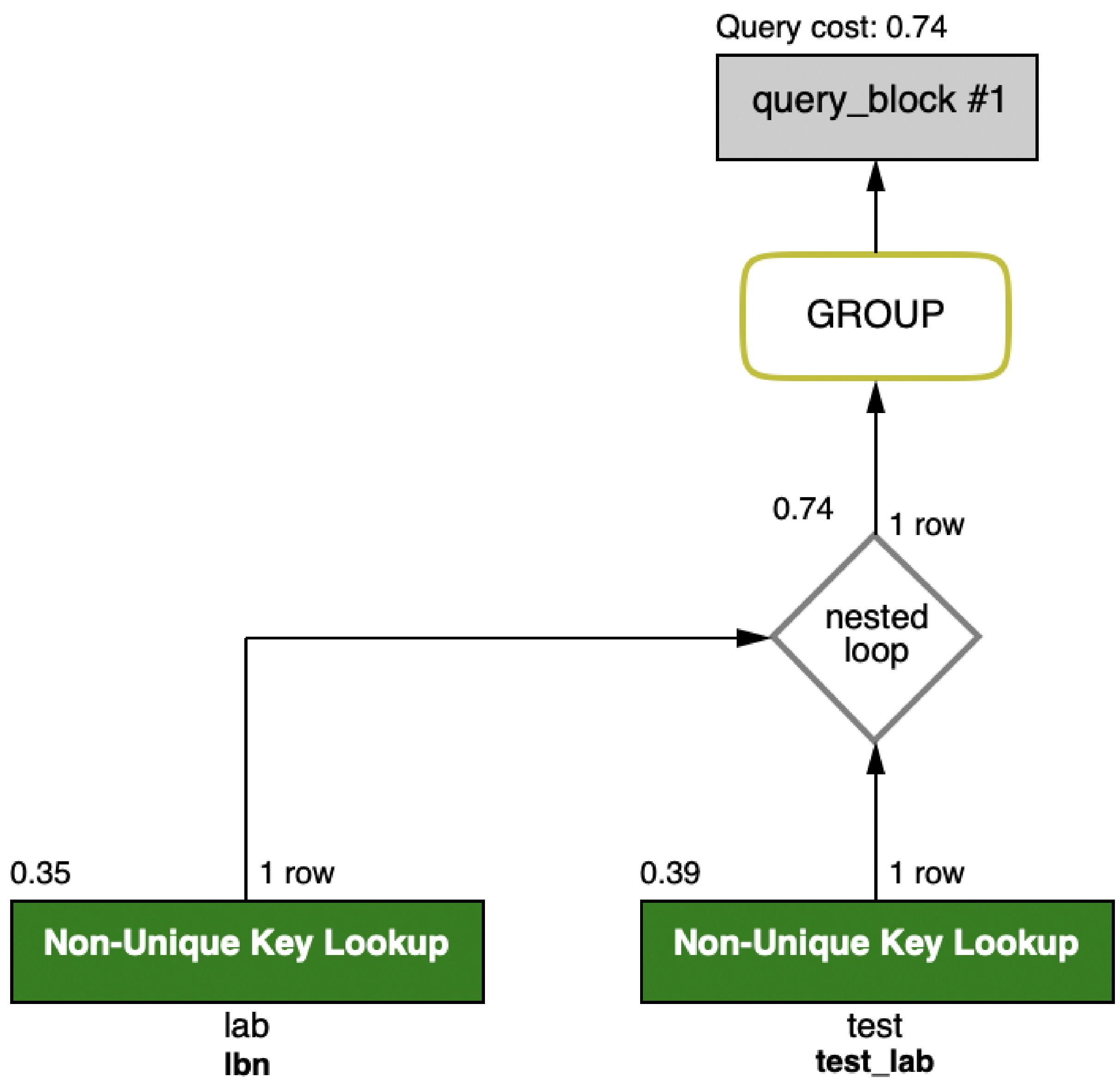


Fig : Query evaluation of query2 Visually

As seen in the above visual explain plan, the number of rows to be examined has reduced from 50 rows to 1 row on using an index and in total the estimated query cost is only 0.74 milliseconds which is very low when compared to the query that does not use an index.
This is because indexing makes columns faster to query by creating pointers to where data is stored within a database. Without index, to get records with the *labName* as ‘cursus’ from the database, there will be a need to look through every row until it finds it. An index is a copy of information from a table that speeds up retrieval of rows from the table or view. Indexes speed up performance by ordering the data on disk so it’s quicker to find your result or telling the SQL engine where to go to find your data.