**Normalization**

You are given the following file of campaign contribution data which is a sample taken from the CA campaign for president in 2016. We are interested in fields for candidate name, contributor, contribution amount and date. We are not interested in the cmte\_id field or the last 7 fields.

CREATE TABLE campaign

(

cmte\_id varchar(12), // campaign id

cand\_id varchar(12), // candidate id

cand\_nm varchar(50), // candidate name

contbr\_nm varchar(50), // contributor name

contbr\_city varchar(40), // contributor city

contbr\_st varchar(40), // contributor state

contbr\_zip varchar(20), // contributor zipcode

contbr\_employer varchar(60), // contributor employer

contbr\_occupation varchar(40), // contributor occupation

contb\_receipt\_amt numeric(8,2), // contribution amount

contb\_receipt\_dt varchar(20), // contribution date

receipt\_desc varchar(255),

memo\_cd varchar(20),

memo\_text varchar(255),

form\_tp varchar(20),

file\_num varchar(20),

tran\_id varchar(20),

election\_tp varchar(20)

);

We want to normalize this data by splitting it into 3 tables for candidate, contributor and contribution.

Run the sql script file campaign-CA-2016.sql, which creates a campaign database with a campaign table. Check that there are 18,118 rows in the table by doing a count(\*) query.

1. Code create statements for the 3 normalized tables candidate, contributor and contribution. Table candidate should have a primary key of cand\_id. Contributor and contribution tables should have a surrogate key of int type defined as autoincrement. Contribution table should have columns for cand\_id and contbr\_id. Include your create table statement here.  
     
   CREATE TABLE CANDIDATE (

cand\_id varchar(12) PRIMARY KEY,

cand\_nm varchar(50)

);

CREATE TABLE Contributor (

contbr\_id INT NOT NULL AUTO\_INCREMENT,

contbr\_nm VARCHAR(50),

contbr\_city VARCHAR(40),

contbr\_st VARCHAR(40),

contbr\_zip VARCHAR(20),

contbr\_employer VARCHAR(60),

contbr\_occupation VARCHAR(40)

);

CREATE TABLE Contribution

(

contb\_id INT NOT NULL AUTO\_INCREMENT,

contb\_receipt\_amt DECIMAL(8,2),

contb\_receipt\_dt varchar(20)  
);

Create an index on contributor name.

create index contributor\_nm on contributor(contbr\_nm);

1. Code 3 insert statements using subselect (read “Inserting from a Query” page 185 in textbook) to select data from the campaign table and insert it into the normalized tables. You should have 22 rows in the candidate table, 14,174 rows in the contributor table, and 18,118 rows in the contribution table. Include your 3 insert statements here.   
     
   -- Insert data into Candidate table:

INSERT INTO CANDIDATE

SELECT \* from CAMPAIGN

where cand\_id in(

SELECT cand\_id FROM CAMPAIGN

);

-- Insert data into Contributor Table:

INSERT INTO CONTRIBUTOR

SELECT \* from CAMPAIGN

where contbr\_nm in(

SELECT contbr\_nm FROM CAMPAIGN

);

-- Insert data into Contribution table:

INSERT INTO CONTRIBUTION

SELECT \* from CAMPAIGN

where contb\_receipt\_amt in(

SELECT contb\_receipt\_amt FROM CAMPAIGN

);

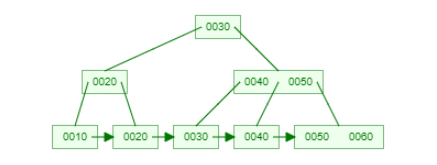
1. Alter the contribution table to add foreign key constraints for columns cand\_id and contrbr\_id. Include your alter table statement here.  
     
   ALTER TABLE Contribution  
   ADD FOREIGN KEY(cand\_id)  
   REFERENCES Candidate(cand\_id);  
     
   ALTER TABLE Contribution  
   ADD FOREIGN KEY(contbr\_id)  
   REFERENCES Contributor(contbr \_id);
2. Create a view named “vcampaign” that is a join of the 3 normalized tables and has columns cand\_id, cand\_nm, contbr\_nm, contbr\_city, contbr\_st, contbr\_zip, contbr\_employer, contbr\_occupation, contb\_receipt\_amt, contb\_receipt\_dt

Do a count(\*) query using the view and verify the result is 18,118.

**B+ Tree Visualization Exercises**

Use the B+ tree simulator at <https://www.cs.usfca.edu/~galles/visualization/BPlusTree.html>

* Set MAX DEGREE = 3 Max Degree is the max number of pointers in an internal (not leaf) node. The max number of values in a node is one less than max degree. MAX DEGREE is similar to what we called in lecture FAN OUT. In the simulator we use a small value for MAX DEGREE, but remember in real databases, the FAN OUT is typically on the order of 100-200.
* Insert the values (one at a time): 10 20 30 40 50 60
* Your diagram should look like



In the diagram above, the leaf node with 0050 0060 is full, as is the parent node 0040 0050. Other nodes are not full.

A B+ tree is efficient for doing key lookup and range queries. However, when new entries have to be inserted or removed from the index due to SQL insert, update or delete statements, there are multiple reads/writes that must be done to maintain the tree nodes in the correct order and the leaf nodes in the correct linked list order.

1. Do an insert of key value 12. Draw or embed a screenshot of the updated index.  
   Diagram

   Description automatically generated
2. How many nodes were either created or modified for the insert of 12?  
   The left node with the key value 10 was modified when inserting the value 12 in it. In total there are 8 nodes.
3. Now do an insert for a key value 14. Show an updated diagram.   
   Diagram

   Description automatically generated
4. How many nodes were either created or modified for an insert of 14?  
   Another node is created, leaving key value 10 by itself and grouping value 12 and 14 together in a node. In total there is 9 nodes.
5. Do an insert of key value 52 and show an updated diagram.  
   Diagram

   Description automatically generated
6. How many nodes were either created or modified for an insert of 52?  
   With the insert of 52 a node of only 52 is created and the 50 is by itself. The insertion of 52 also became part of the same node as 60. In total there are 11 nodes.

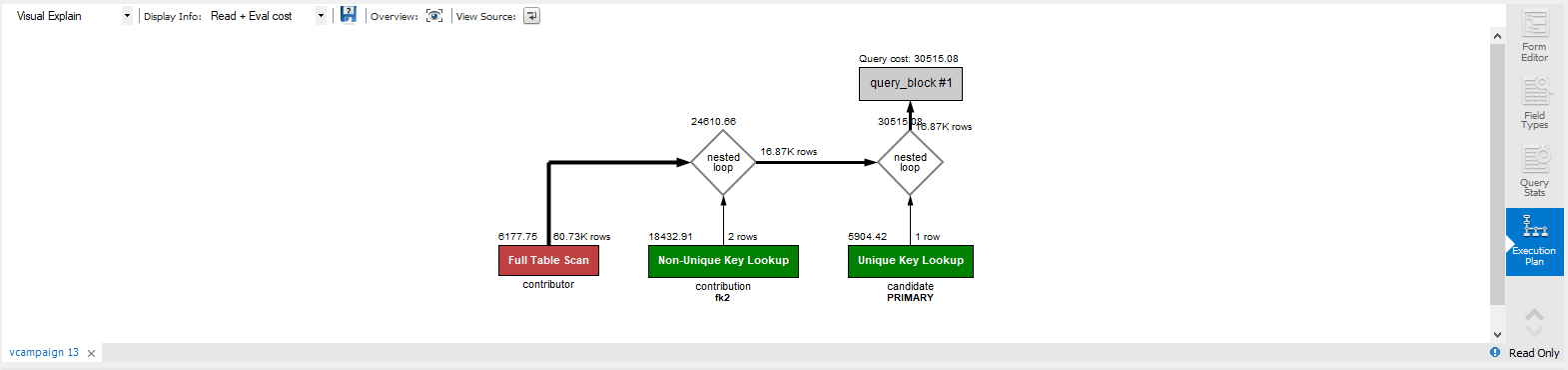
Conclusion: insert, delete of a B tree index may involve several reads/writes.

**Query Plan Exercises**

Perform the query

select \* from vcampaign where contbr\_zip = '92653';

Then examine the query plan by scrolling down the list of icon the right side of the result panel and selecting the “Execution Plan”.



The query plan depicts how a table is accessed: either by reading the entire table (Full Table Scan Red Rectangle) or using an index (Green Rectangle with index name below the box). An index is unique if it is the primary key index or an index defined on a column that is defined as unique. The query plan also depicts how joins are done. In the diagram a scan of the contributor table is done and each row is joined first to rows in the contribution table by looking up contbr\_id using index fk2, and then join with row from candidate table looking cand\_id using the primary key index. By default, MySQL creates index on the primary key column(s) and on each foreign key column(s).

Create an index on contbr\_zip column in the contributor table

create index zip on contributor(contbr\_zip);

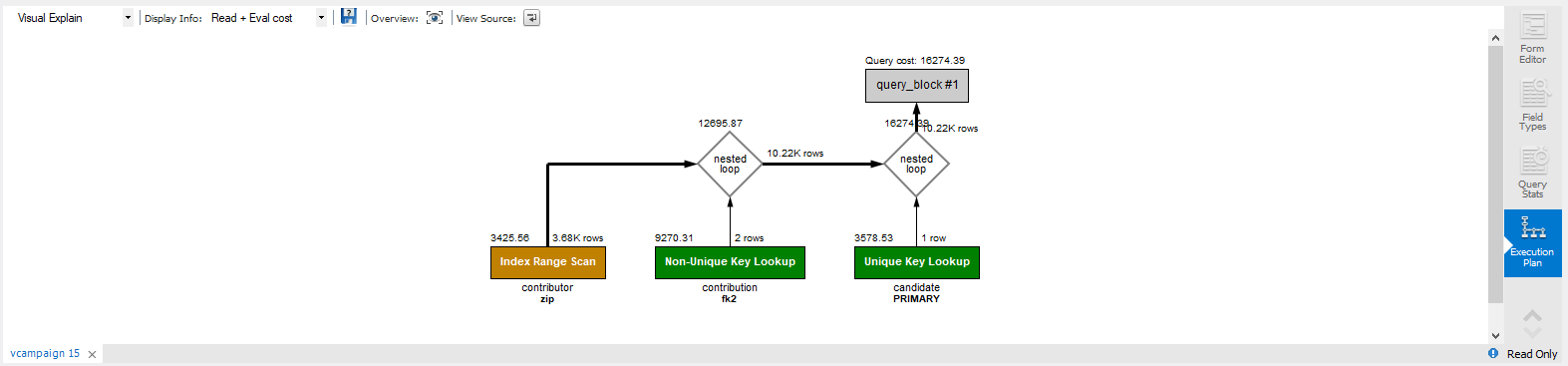
Redo the query and examine the execution plan.

select \* from vcampaign where contbr\_zip = '93933';

1. Is the new index being used? Explain in your words the execution plan.  
   The index is being used and the layout is like the previous one but this time the query will be much faster than the first query that was used.

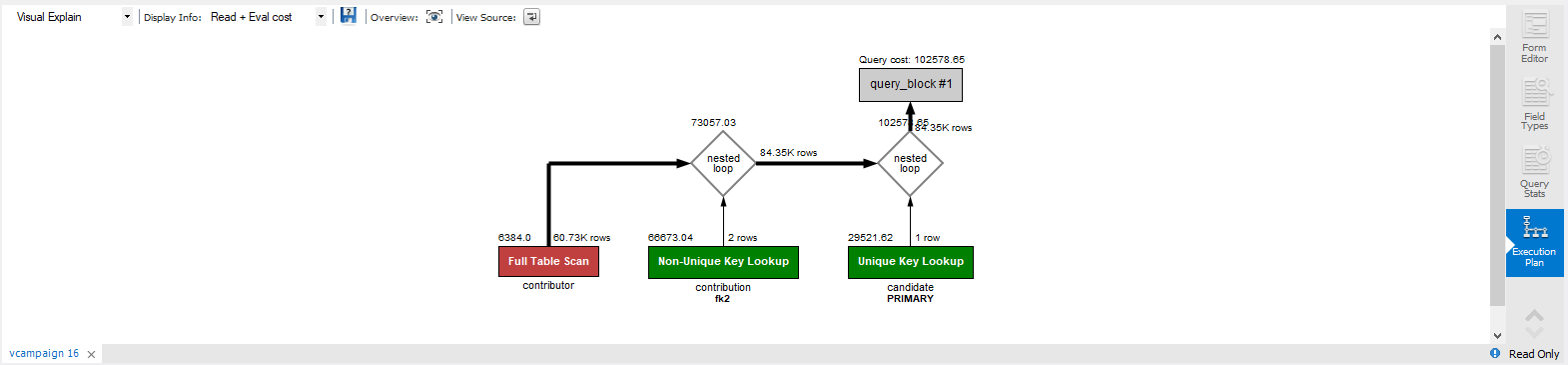
Do a query on vcampaign where contbr\_zip is between 93001 and 93599 (the zip codes in LA area)

The query plan is



showing that range scan is done using index on zip.

Change the query to zip between 00001 and 93599. The execution plan is



The zip index is not being used. Why? The MySQL query optimizer realizes that it will be faster to scan all row in contributor for zip between 00001 and 93599 rather than use index. An index is used to search when the result is expected to be a few rows. If many rows are expected, it is faster to just scan the whole table. How does the optimizer know when to use an index and when to scan? There are statistics kept about each table and each column: the number of rows, the max and min values for each column, the number of distinct values for a column. Pretty clever!

**Concurrency Exercises**

**Exclusive locking**

Observe the behavior of exclusive locking when two concurrent transactions attempt to update the same row.

For this exercise you will need two connections in the workbench that have auto commit turned off.

* Open a connection
  + menu 🡪 Query 🡪 uncheck the item “Auto Commit Transactions”
* Open a second connection.
  + To do this use the tab with the “Home” on it to return to the connection page and then open the second connection.
  + menu 🡪 Query 🡪 uncheck the item “Auto Commit Transactions”

|  |  |  |
| --- | --- | --- |
| Instance 1 | Instance 2 | Comments |
|  | use zagimore;  set autocommit = 0;  select \* from product where productid='1X1';  What is the price returned? |  |
| use zagimore;  set autocommit = 0;  select \* from product where productid='1X1';  What is the price returned? |  |  |
|  | update product set productprice=productprice+100 where productid='1X1';  select \* from product where productid='1X1';  What is the price returned? | Instance 2 has updated the price but has not committed it. Other clients cannot see uncommitted data. |
| select \* from product where productid='1X1';  What is the price returned? |  | Since the update by Instance 2 has not been committed and Instance 1 does not see the update and instead see the previously committed value. |
| update product set productprice=productprice+100 where productid='1X1';  select \* from product where productid='1X1';  **Notice the call is Running…** |  |  |
|  | commit; |  |
| The call now completes.  select \* from product where productid='1X1';  What is the price returned? |  |  |
| commit; |  |  |

**Inconsistent Writes**

Alice and Bob are both on duty. One of them may go off duty assuming that they first check that the other is still on duty.

* Open two connections as in the last problem.
* On both connections menu 🡪 Query 🡪 uncheck the item “Auto Commit Transactions”
* Create the following table and 2 rows.

create table duty (name char(5) primary key, status char(3));

insert into duty values ('Alice' ,'on'), ('Bob', 'on');

commit;

|  |  |  |
| --- | --- | --- |
| **Instance 1 “Alice”** | **Instance 2 “Bob”** |  |
| set autocommit=0;  select \* from duty; |  | Alice checks that Bob is on duty. So she updates her status to off duty. |
| update duty  set status=’off’  where name=’Alice’ |  |  |
|  | set autocommit=0;  select \* from duty; | Bob checks that Alice is on duty. So he updates his status to off duty. |
|  | update duty  set status=’off’  where name=’Bob’ |  |
| commit; |  |  |
|  | commit; |  |

What has just happened? Bob and Alice have both gone off duty even though each one checked that the other was on duty. Isn’t one of reasons to use a database is for data integrity? But how does the database this to happen? But you must understand how a database system works together with the application to guarantee data integrity.

Databases do exclusive locking on updates to the same row. But in this situation the updates are to two different based data read from two different rows.

1. Based on lecture material there are 2 ways to fix this problem. Pick one and test it out. How did you fix the problem?  
     
   The shared and exclusive locking protocol is based on acquiring and releasing locks by the transactions. There are two types of locks which may be acquired by the transactions -- Shared lock: is acquired when the transaction wants to only read the data and not write anything on it. Exclusive lock: is acquired when the transaction wants to write anything on the data. This locking method may or may not ensure serializability in the schedule. It is also not free from recoverability, cascading rollback, starvation, and deadlock.

In the question, we observe blind writes on the database. Also, since the transaction doesn't commit after write, it leads to inconsistency in the database as the other transaction reads data from the database while the data has been changed, and as such affects the integrity of the data.

We can ensure serializability as well as recoverability in this case by using exclusive lock and holding exclusive locks until commit. When Instance 1(Alice) reads and updates the data, it needs to acquire an exclusive lock for this updating. Similarly, when Instance 2(Bob) want to read and update the data, it also needs to acquire exclusive lock. Since we know that exclusive lock can't be given to a transaction when another transaction already acquired exclusive lock, if instance 1 is checking and updating the value, instance 2 will not be able to check and update until instance 1 releases the lock.

Also, as we have made the transactions hold exclusive locks until commit or abort, instance 2 can't read or write the data until instance 1 has committed and vice versa. This will ensure data integrity and prevent the data integrity violation which was occurring earlier.

**Other Exercises**

1. Consider this situation: you try to get cash at an ATM, but the ATM fails after updating your account and committing, but just before cash is dispensed.  As a system designer, how do you cope with the situation that the money has been debited from the account and committed but the cash was unable to be dispensed?  [hint:  what do you think “compensating transaction” means? do a google search.]  
   Implementing a compensatory transaction is the solution and the stages in a compensating transaction must reverse the consequences of the original operation’s actions. Compensating transaction might not be able to replace the current state with the state the system was in at the start of the operation because this approach could overwrite changes made by other concurrent instances of an application. A common approach is to use a workflow to implement an eventually consistent operation that requires compensation. As the original operation continues, the system records information about each step and how the work performed by that step can be undone. If the operation fails at any point, the workflow will rewind back through the completed steps and performs the work that reverses each step. Like how in the ATM, it fails before dispensing the cash it will have to compensate for the work performed by the steps that completed successfully before the failure.
2. Consider this situation: you try to buy an airline ticket at a web site.  The transaction commits on the server, but crashes just before the message confirming the reservation is sent to the client. As a system designer, how would you cope with the situation of a reservation was made and committed in the database, but the confirmation message was never received by the client?   
   Just like above I think implementing a compensatory transaction is a solution to this problem. And if it is unable to recover from the previous step then a manual intervention would be needed. Something to that will help would be to place a short-term timeout kind of lock on each resource that is required to complete an operation and obtaining those resources in advance. This will help increase the likelihood that overall activity will succeed.

**What to submit for this assignment?**

Edit this file with your answers to the 14 questions. Submit your answers as a PDF file to the Canvas assignment.