**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 IF NOT EXISTS `campaign`.`candidate` (

`cand\_id` VARCHAR(12) NOT NULL,

`cand\_nm` VARCHAR(50),

PRIMARY KEY (`cand\_id`));

CREATE TABLE IF NOT EXISTS `campaign`.`contributor` (

`contrbr\_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),

PRIMARY KEY (`contrbr\_id`));

CREATE TABLE IF NOT EXISTS `campaign`.`contribution` (

`contbr\_id` INT NOT NULL AUTO\_INCREMENT,

`contrbr\_id` INT,

`cand\_id` VARCHAR(12),

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

`contb\_receipt\_dt` VARCHAR(20),

PRIMARY KEY (`contbr\_id`));

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 INTO candidate

SELECT DISTINCT campaign.cand\_id, campaign.cand\_nm

FROM campaign;

INSERT INTO contributor

SELECT DISTINCT 0, campaign.contbr\_nm, campaign.contbr\_city, campaign.contbr\_st, campaign.contbr\_zip, campaign.contbr\_employer, campaign.contbr\_occupation

FROM campaign;

INSERT INTO contribution

SELECT 0, contributor.contrbr\_id, campaign.cand\_id, campaign.contb\_receipt\_amt, campaign.contb\_receipt\_dt

FROM campaign, contributor

WHERE campaign.contbr\_nm = contributor.contbr\_nm AND campaign.contbr\_city = contributor.contbr\_city AND campaign.contbr\_st = contributor.contbr\_st AND campaign.contbr\_zip = contributor.contbr\_zip AND campaign.contbr\_employer = contributor.contbr\_employer AND campaign.contbr\_occupation = contributor.contbr\_occupation;

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 CONSTRAINT FK\_contrbr\_id FOREIGN KEY (contrbr\_id) REFERENCES contributor(contrbr\_id),

ADD CONSTRAINT FK\_cand\_id FOREIGN KEY (cand\_id) REFERENCES candidate(cand\_id);

1. 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

CREATE VIEW vcampaign AS

SELECT candidate.cand\_id, candidate.cand\_nm, contbr\_nm, contbr\_city, contbr\_st, contbr\_zip, contbr\_employer, contbr\_occupation, contb\_receipt\_amt, contb\_receipt\_dt

FROM candidate JOIN contribution ON candidate.cand\_id = contribution.cand\_id

JOIN contributor ON contributor.contrbr\_id = contribution.contrbr\_id;

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

**B+ Tree Visualization Exercises**

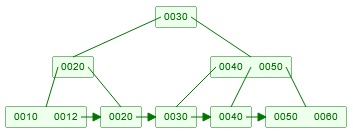
Use the B+ tree simulator at <https://www.cs.usfca.edu/~galles/visualization/BPlusTree.html>

* 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.

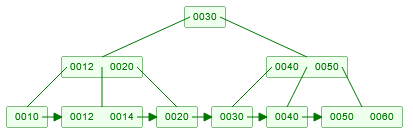
1. Do an insert of key value 12. Draw or embed a screenshot of the updated index.



1. How many nodes were either created or modified for the insert of 12?

1 node was modified/created

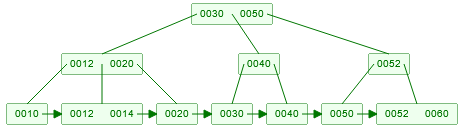
1. Now do an insert for a key value 14. Show an updated diagram.



1. How many nodes were either created or modified for an insert of 14?

2 nodes were modified/created

1. Do an insert of key value 52 and show an updated diagram.



1. How many nodes were either created or modified for an insert of 52?

5 nodes

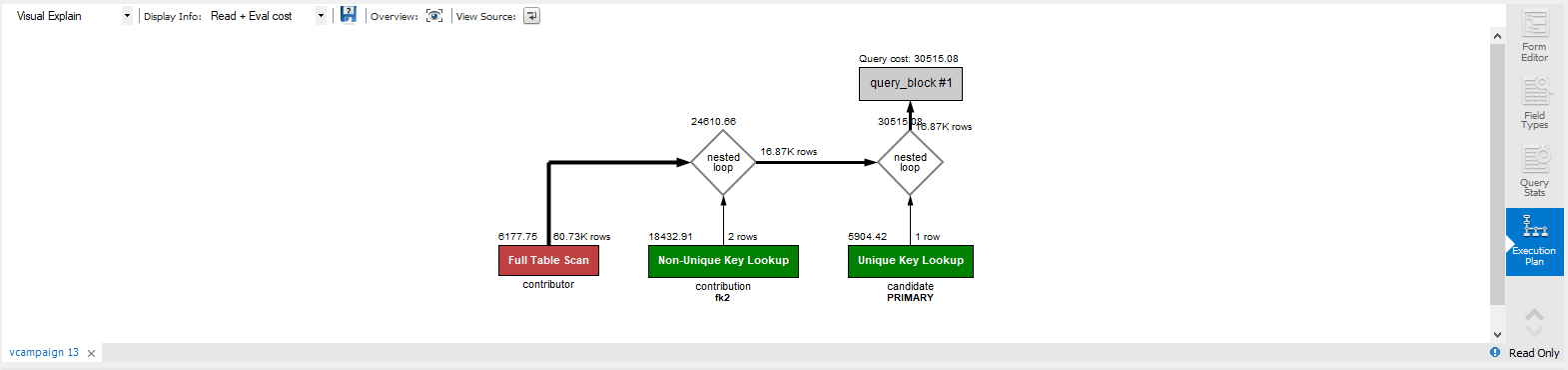
Conclusion: insert, delete of a B tree index may involve several reads/writes.

**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.

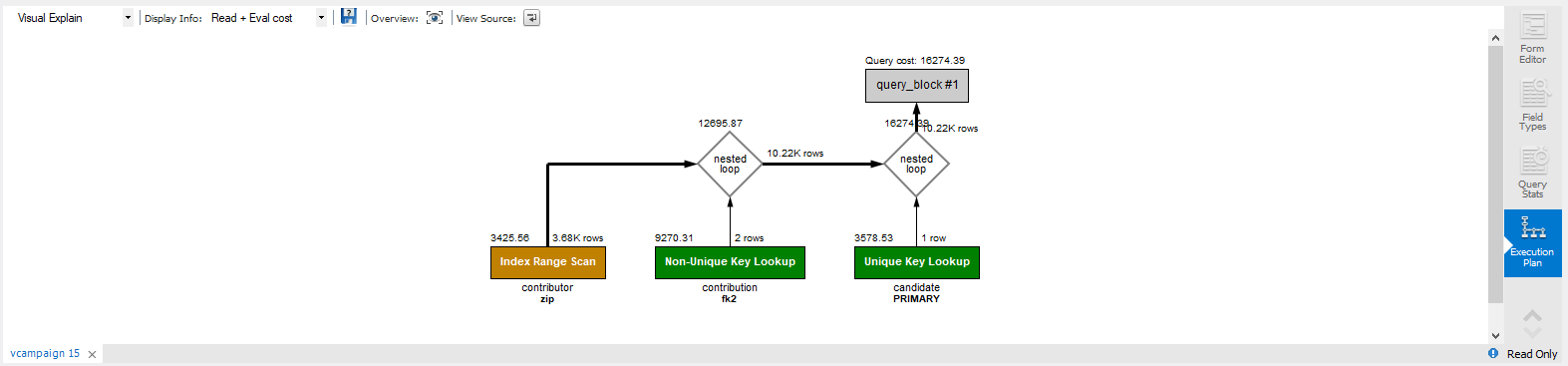
select \* from vcampaign where contbr\_zip = '93933;

1. Is the new index being used? Explain in your words the execution plan.

Yes the new index is being used. The index makes searching for information faster. It provides a point where the information can begin to be searched for directly instead of searching the entire database.

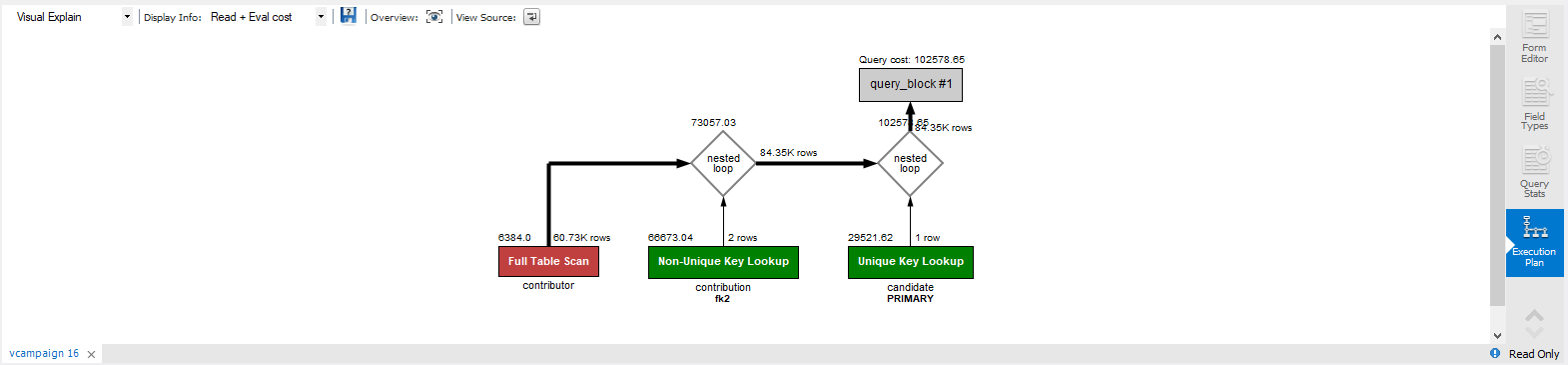
Do a query on vcampaign where contbr\_zip is between 93001 and 93599 (the zip codes in LA area)

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?  Price returned was 100.00 |  |
| use zagimore;  set autocommit = 0;  select \* from product where productid='1X1';  What is the price returned?  Price returned was 100.00 |  | Nothing was returned |
|  | update product set productprice=productprice+100 where productid='1X1';  select \* from product where productid='1X1';  What is the price returned?  Price returned was 200.00 | 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?  Price returned was 100.00 |  | 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?  Price returned was 300.00 |  |  |
| 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?

To fix the Alice and Bob problem I would use serializing on transactions. Serializability helps preserve the consistency and concurrency of a database.

**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.]

Compensating transaction is a set of database operations that perform a logical undo of a failed transaction. A method would be the way the book describes it, using, the backup and recovery methods, using a recovery log as described in the book, that is used to protect against this loss of data (and money, in this case). The transaction can be rolled back. Using this idea, I would set up the ATM to undo the operation since the money was unable to be dispensed. Since the transaction was committed and the money was taken out of the account, I would roll back the transaction and send the user an email stating that the money will be put back into their account in a matter of time.

1. 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?

I would keep a check in the system sends the confirmation message to the email that was used to create the transaction. If the email was sent then the database would consider that as being a good transaction. The user may not get a confirmation message displayed but they will get an email confirmation.

**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.