Assignment 2

Question 1

Considering the given Course Scheduling schema, below questions are answered:

(a) What are plausible Primary Keys on each of the five relations?

Answer:

The primary keys were selected to ensure the uniqueness of rows for each table of the database:

Table	Primary Key
Instructor	instID
Course	courseID
Offering	Composite key of: courseid+ section+ termCode
Classroom	roomID
Department	deptID

(b) What are plausible Foreign Keys for the five relations?

Answer:

Table	Foreign Key	Tables Relating to
Instructor	DeptID	Department
Course	DeptID	Department
Offering	courseID	Course
	roomID	Classroom
	InstID	Instructor
Classroom	N/A	
Department	N/A	

(c) What additional constraints, if any, should be added?

Answer:

The additional constraints that are needed on this database are:

All the Primary Key and Foreign Key defined above must be **Unique** and **Not Null**.

(d) Knowing that each department is part of a faculty (deptID!faculty), that courses can have more than one prerequisite, and desiring to be able to do queries based on term (Winter, Spring, Fall) without regard to the particular year (e.g., what courses are offered in the fall term?), what **modifications to the schema**, if any, are needed to ensure that it is either **3NF** or **BCNF** (your choice)? If there are **any new of changed relations, identify them, including any changes or adjustments to primary keys and/or foreign keys, or any other constraints**. Explain your reasoning.

Answer:

The normalization process starts here with 1NF which dictates that each cell of the database (Considering the tabular format where Each Table is represented with rows and columns. Rows show data for each entry and each column shows one data for each specific entry.) has only one data in them and each table has a Primary Key.

For 1NF: we need to implement the Primary Keys and Foreign Keys described on (a) and (b). However, 1NF also dictates each cell should have only one data but from the table description, it can be seen that the Column **termCode** in table **Offering** has two data merged in it which are the information for **Year** and **Session** (e.g.: Fall, Summer, Winter). So, we decompose this data into two columns and introduce columns called **yearID** for defining the Year and **sessionCode** for denoting the session. The Primary Keys and Foreign Keys are changed as followed for this change:

Table	Primary Key	Foreign Key
Instructor	instID	DeptID
Course	courseID	DeptID
Offering	Composite key of: courseid+ section+ yearID+sessionCode	Foreign key 1: courseId Foreign key 2: roomID Foreign key 3: InstID
Classroom	roomID	
Department	deptID	

For 2NF, the requirement is when there is a composite primary key and there is a partial dependency, the table should be split. In our database, the only table that has a composite primary key is Offering and there is no partial dependency found. So, the table is already conformed to second degree normalization.

For 3NF, the requirement is eliminating any transitive dependency. However, looking into the database, we can see that there is no transitive dependency. So, we do not need to perform any additional task and the database is already conforming to the 3NF principle.

For BCNF (Boyce- Codd Normalization Form), the requirement is to decompose the table when there is a functional dependency so like A ->B where A is a **Superkey**.
Now, for BCNF, we need to analyze the following query requirements.
1) Each department is part of a faculty (deptID ->faculty):
As deptID is the Primary key of our table Department and this would make deptName dependent on the Superkey (deptID, faculty), according to BCNF, this table needs to be decomposed:
So, we decompose this table into following two tables:
Department (deptID, deptName)
DepartmentDetails (departmentDetailsID, deptID, faculty)

2) Courses can have more than one prerequisite:
This means in the Course table there can be repetition of data due to Multiple Prerequisite existence for one course. So, we decompose course table into following two tables:
Course (courseId, courseName, deptID)
CoursePrereq (coursePrereqID,courseID, prereqID)

3) Query should be able to run without regard to the particular year:

If the queries run without regard to particular year, the only table that has the Year information is Offering table. In this scenario, we need to separate the Year information from the rest of the data so this table can be decomposed into following two tables:

CourseOffering (courseOfferingID, courseID, yearCode, sessionCode)
CourseOfferingDetails (courseOfferingDetailsId, courseOfferingID, section, roomID, instID, enrollment)

Finally, the table schema, relations and keys change as followed because of the normalization:

Table	**Primary Key**	**Foreign Key**	**Table Relating to**
Instructor	instID	DeptID	Instructor
Department	deptID		
DepartmentDetails	departmentDetailsID	deptID	Department
Course	courseId		
CoursePrereq	coursePrereqID	courseID	Course
CourseOffering	courseOfferingID	courseID	Course
CourseOfferingDetails	courseOfferingDetailsId	courseOfferingID	CourseOffering
		roomID	Classroom
		instID	Instructor
Classroom	roomID		

(e) Considering queries for the following purposes:

- Which instructors are sessionals?
- Which instructors have taught courses over a particular timeframe?
- How many courses are taught by sessionals?
- How many students are taught by sessionals?
- Any of the above, grouped by faculty
- Any of the above, as fractions of total instructors and/or courses, as relevant?

Using "explain" and/or your own reasoning, identify what indexes would be potentially useful to help in these queries.

Answer:

For considering indexes, we first need to determine the queries:

Query 1: Which instructors are sessionals?

```
SELECT * FROM Instructor
WHERE sessional=true;
```

For this query, we do not add any additional indexes as the Condition is based on Sessional value which has only two values to begin with (true, false). Adding index here will only increase database resource usage and would do very little to nothing to optimize the query.

This can also be confirmed from the cost value of the execution plan of the queries with and without index.

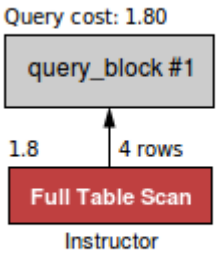

Fig: Query Without Index

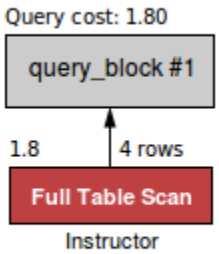

Fig: Query with index

Query 2: Which instructors have taught courses over a particular timeframe?

```
SELECT a.instID, a.instName FROM Offering a
JOIN Instructor b ON a.instID = b.instID
WHERE a.termCode = 1191
```

In this query, I am interpreting which term as particular timeframe. As termcode is a big dataset, we are adding index idx_termcode on the Termcode column of Offering table. The query can also utilize the automatic index added by the primary key on Instructor table (instID).

Query 3: How many courses are taught by sessionals?

```
SELECT count (DISTINCT a.courseID) FROM Offering a
JOIN Instructor b ON a.instID = b.instID
WHERE a.sessional = true;
```

For the same reason as Query 1 in this question, I am opting out of adding any additional index considering this query.

Query 4: How many students are taught by sessionals?

```
SELECT SUM(b.enrollment) FROM Offering b
JOIN Instructor a ON b.instID = a.instID
JOIN Department c ON a.deptID = c.deptID
WHERE a.sessional = true
```

On this query, even though the query where clause follows the same and have a similar structure to the ones before, the index that can be helpful is adding it on instID from Instructor table as this is used both in the Join condition and the sessional value is closely dependent on instID. However, as the column should be a Primary key for instructor table, this index should be automatically added, and no additional index required.

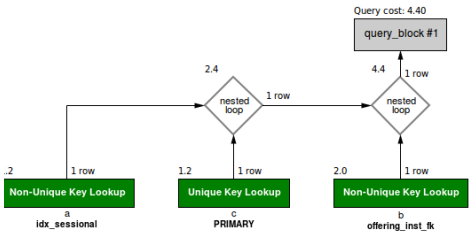

Fig: Query Execution plan

Query 5: Any of the above, grouped by faculty.

This query can have different versions, for analysis, I am choosing the one as followed:

```
SELECT SUM(a.enrollment) FROM Offering a
JOIN Instructor b ON a.instID = b.instID
JOIN Department c ON b.deptID = c.deptID
WHERE b.sessional = true
GROUP BY c.faculty;
```

For this query, I am adding index on Faculty from Department table as this is the condition for Group by. The faculty has a limited number of entries and adding index on Faculty should help to group together multiple data at once and optimize the query. Indexes instID from Instructor table (considering instID as Primary key for the table) and deptID (considering deptID as primary key of Department table) should also help optimize the query.

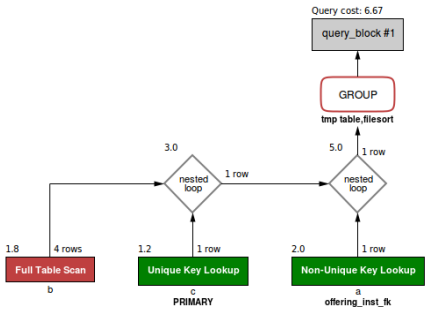

Fig: Query without Index

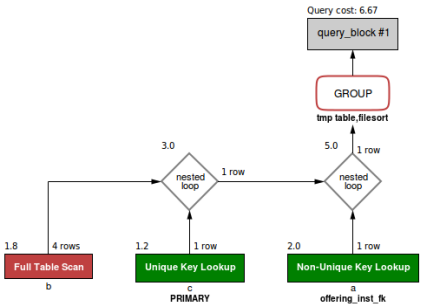

Fig: Query with index

Query 6: Any of the above, as fractions of total instructors and/or courses, as relevant?

This query can have many different interpretations, for analysis, I am choosing the two below:

a)

```
SELECT 'Instructor',
(SELECT count(instID) FROM Instructor WHERE sessional = true)
/ (SELECT count(instID) FROM Instructor) as fraction
UNION
SELECT 'Course',
(SELECT count (DISTINCT a.courseID)
```

```
FROM Offering a JOIN instructor b ON a.instID = b.instID WHERE b.sessional = true)
/ (SELECT count (DISTINCT courseID) FROM Offering) as fraction;
```

b)

```
SELECT
(SELECT count (DISTINCT a.courseID)
FROM Offering a JOIN Instructor b ON b.instID = a.instID
WHERE b.sessional = true)
/
(SELECT count (DISTINCT a.courseID)
FROM Offering a JOIN Instructor b ON a.instID = b.instID);
```

Analyzing the queries above, I am adding additional index to Sessional from Instructor table as this can be a complicated Join and Union query and sessional is being used in most of the queries. Analysis also suggests that all the columns that have indexed before is also relevant for this one (E.g.: termCode from Offering table, faculty from Department table). The queries should also be able to utilize the Indexes available on other tables (added by Primary and Foreign Key) to optimize the Join conditions.

So, finally we can say I will consider adding the below indexes on tables for the conditions provided:

Index name	**Column**	**Table**
idx_sessional	sessional	Instructor
idx_termCode	termCode	Offering
idx_faculty	faculty	Department

(f) Considering the specific query:

select count(courseID) from Course inner join Department using (deptID) where prereqID is NULL and faculty=’Math’;

Assuming no indexes, what would the execution plan be and what would be the estimated execution time for that plan if the tables are on disk, in contiguous blocks, the number of rows in Course is rc, the number of blocks in Course is bc, the number of rows in Department is rd, the

number of blocks in Department is bd, the time to find a random block on disk is Ts and the time to transfer a block from disk is Tt?

Answer:

The execution plan for the given query should be completed as followed:

$$\Pi_{count(courseID)} \left(\sigma_{prereq=NULL \cap faculty=' Math'} (\text{Course} \bowtie_{deptID} \text{Department}) \right)$$

The plan as a flow can be shown as below:

$$\Pi_{count(courseID)}$$

$$|$$

$$\sigma_{prereq=NULL \cap faculty=' Math'}$$

$$|$$

$$\bowtie_{deptID}$$

$$/ \backslash$$

$$course \quad department$$

Now, the assumption is this query is being run without any index so all the actions here will have to be linear search in contiguous blocks.
Now, if we estimate the cost from the inner-most part, first let us calculate the join action.
Here, the Course table in Joined with the Department table based on column deptID. The simplest way to join two tables without any index is Nested Loop Join. Let us assume the data here is contiguous. Now the number of rows in course is r_c and the number of block in department is b_d where as the number of blocks are b_c so the total cost of block transfer and seek transfer would be:

$$Join\ Cost = (r_c \cdot b_d + b_c) \cdot T_t + (r_d + b_c) \cdot T_s$$

As for the next part, let us move on to the select action. Again, assuming these are contiguous blocks and this is an And condition, the query will run through the whole tables for both cases with a linear search. The total cost for the select action would be the cost of these two actions together. This total cost is:

$$Selection\ Cost = (b_c \cdot T_t + T_s) + (b_d \cdot T_t + T_s)$$

As for the projection, it returns the count of rows so it does not incur any additional costs.

So, the final cost for the query is:

$$Total\ Cost = (b_c \cdot T_t + T_s) + (b_d \cdot T_t + T_s) + ((r_c \cdot b_d + b_c) \cdot T_t + (r_d + b_c) \cdot T_s)$$

(g) For the query above, identify any **indexes over one or more attributes** that might potentially improve the query performance. For each index you identify, specify the **type of index** (B+-tree or Hash or either), whether or not it is a **primary or secondary** index, if it is a secondary index identify if it is useful if it is an index extension, and justify why the query might benefit from that index.

Answer:

The query execution on 1(f) was quite costly due to unavailability of indexes. To reduce this cost, we can implement index on the select action conditions. For that, two separate indexes can be used:

1) idx_prereq : This index can be added on the **prereq** column of Course table. This should be a secondary B+ tree index.
2) idx_faculty: This index can be added on the **faculty** column of Department table. This should also be a secondary B+ tree index.

Another index that might help in cost optimization is adding index on the JOIN condition which should be added on **deptID** column from Department table. This should be a **Primary** index (primary key of Department table.) In reality though, while the indexes on WHERE did help to reduce the cost, the deptID, even though it was added as a Primary index, did not help in that regards.

However, by using Index Extension and introducing a composite index on (**deptId, faculty**) from Department table - the cost was reduced. As faculty was added as a Secondary index and it was appended to the Primary Key of Department table, this can be classified as an appropriate index extension. As the select operation works after JOIN, it was useful to add the index extension which helps the database to gather more data at once and reduce multiple efforts.

Question 2

In assignment 1 you had to compute several queries on the Sean Lahman baseball database. There were no explicit indexes on that database, though you should have added primary and foreign keys. Using explain on the queries you created for Lab 1, determine if any additional explicit indexes would help in solving those queries.

Answer:
For Assignment 1, there were total 6 queries done on Lahman baseball database. Let us consider one by one:
a) select count(*) from Master
where birthDay= 0 or birthMonth=0 or birthYear=0;

- we are not any additional index on this as the query condition is an OR clause and indexing would not help to speed up the query.

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	Master	NULL	ALL	NULL	NULL	NULL	NULL	18980	27.1	Using where

b) select
(select count(*) from Master
inner join HallOfFame
on Master.playerID= HallOfFame.playerID
where Master.deathYear =0 or Master.deathDay =0 or Master.deathMonth =0)
-
(select count(*) from Master
inner join HallOfFame
on Master.playerID= HallOfFame.playerID
where Master.deathYear !=0 or Master.deathDay !=0 or Master.deathMonth !=0) as DifferenceofAliveandDEath;

- We do not add any additional index on this. For select: the query uses or where using index is not feasible. If we could use Union and create 3 separate queries then adding index on birthday, birthmonth and birthyear would be helpful. For join: the join condition is on primary key and foreign key so no index is necessary either.

This can also be supported by EXPLAIN on the query:

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	PRIMARY	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	No tables used
3	SUBQUERY	HallOfFame	NULL	index	PRIMARY	PRIMARY	1538	NULL	4156	100	Using index
3	SUBQUERY	Master	NULL	eq_ref	PRIMARY	PRIMARY	767	lahman2016.HallOfFame.playerID	1	99.9	Using where
2	SUBQUERY	HallOfFame	NULL	index	PRIMARY	PRIMARY	1538	NULL	4156	100	Using index
2	SUBQUERY	Master	NULL	eq_ref	PRIMARY	PRIMARY	767	lahman2016.HallOfFame.playerID	1	27.1	Using where

c) select concat(B.nameGiven,' ',B.nameLast), C.HighestSalary from
(select A.playerID,(SUM(A.salary)) HIghestSalary from Salaries A group by A.playerID) C
inner join Master B on B.playerID = C.PlayerID
order by C.HighestSalary desc limit 1;

- We do not add any additional index as the order condition is descending and for this scenario, indexing is not advised.

d) select avg(HR) as BattingHR from Batting;

- We added an additional index called idx_HR on HR column from Batting table as avg is an aggregated function on a non-prime attribute and it should help to reduce the cost of the query.

e) select avg(HR) as BattingHR from Batting where HR>=1;

- We do not add any additional index for this query. We tested with adding index on HR and it actually increased the cost of the query since the range for the WHERE clause is too big and it contains more than 1/3rd of the whole data.

f) select count(distinct a.playerID)
from (select playerID from Batting
    group by playerID
    having avg(HR) > (select avg(HR) from Batting)) as a
inner join
    (select playerID from Pitching
    group by playerID
    having avg(SHO) > (select avg(SHO) from Pitching)) as b
on b.playerID=a.playerID;

- We do not add any additional index for this query. We tested with adding index on HR column from Batting table and SHO column from Pitching table however the system ends up using the primary keys instead as it is a complicated query with multiple actions.

This is explained from the Execution plan below too:

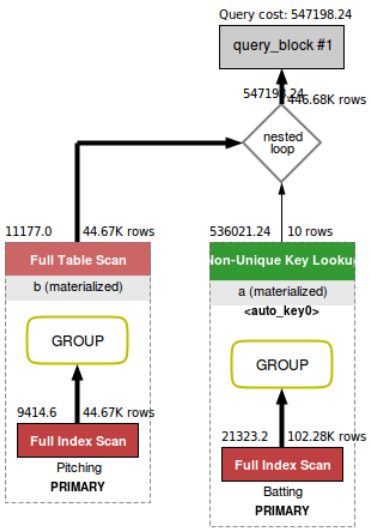

Fig: Query without index

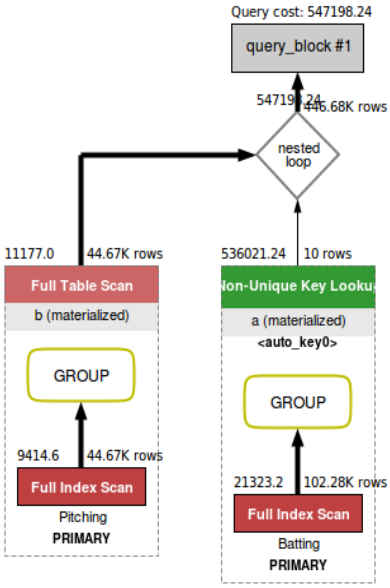

Fig: Query with index

Question 3:

Likewise, you had to compute several queries on the Yelp database. Again, using explain on the queries you created for Lab 1, determine what indexes would help in solving those queries.

Answer:

Same as the previous question, this part was consisted of 6 different queries as well. Let us consider one by one.

a) select name,review_count from user group by user_id,name
order by review_count desc limit 1;

- For this query, we do not need any additional index as the Group By is occurring on user_id which is already a Primary Key and the Order is a descending condition which does not get any benefits from index.

Using Explain on the query also supports the approach:

```
+----+-------------+-------+------+---------------+------+---------+------+---------+----------------+
| id | select_type | table | type | possible_keys | key  | key_len | ref  | rows    | Extra          |
+----+-------------+-------+------+---------------+------+---------+------+---------+----------------+
|  1 | SIMPLE      | user  | ALL  | NULL          | NULL | NULL    | NULL | 1021667 | Using filesort |
+----+-------------+-------+------+---------------+------+---------+------+---------+----------------+
```

b) select business_id, name,review_count from business
group by business_id
order by review_count desc limit 1;

- Using Explain, gives us the following result:

```
+----+-------------+----------+------+---------------+------+---------+------+--------+----------------+
| id | select_type | table    | type | possible_keys | key  | key_len | ref  | rows   | Extra          |
+----+-------------+----------+------+---------------+------+---------+------+--------+----------------+
|  1 | SIMPLE      | business | ALL  | PRIMARY       | NULL | NULL    | NULL | 142527 | Using filesort |
```

+----+-------------+----------+------+---------------+------+---------+------+--------+----------------+

For the same reason as (a), we do not add any additional index for this query either.

c) select avg(review_count) from user;

- Explain produces the following for the query above:

+----+-------------+-------+------+---------------+------+---------+------+---------+-------+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows | Extra |
+----+-------------+-------+------+---------------+------+---------+------+---------+-------+
| 1 | SIMPLE | user | ALL | NULL | NULL | NULL | NULL | 1021667 | NULL |
+----+-------------+-------+------+---------------+------+---------+------+---------+-------+

For this query, we can see that there is no possible keys or particular method used. We would like to add additional index named idx_review_count on the column review_count from table User. This is used for an aggregate function so it should optimize the query speed.

d) select count(*) from (select user_id,average_stars as given_avg from user) as A
inner join (select user_id,avg(stars)as computed_avg from review group by user_id) as B
on B.user_id=A.user_id
where A.given_avg-B.computed_avg>0.5 or B.computed_avg-A.given_avg>0.5;

- From the Explanation of the above query, below result is produced:

+----+-------------+------------+------+---------------+-------------+---------+-----------+---------+-----------------------------+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows | Extra |
+----+-------------+------------+------+---------------+-------------+---------+-----------+---------+-----------------------------+
1	PRIMARY	<derived2>	ALL	NULL	NULL	NULL	NULL	1021667	NULL
1	PRIMARY	<derived3>	ref	<auto_key0>	<auto_key0>	22	A.user_id	10	Using where
3	DERIVED	review	ALL	NULL	NULL	NULL	NULL	1655155	Using temporary; Using filesort
2	DERIVED	user	ALL	NULL	NULL	NULL	NULL	1021667	NULL

+----+-------------+------------+------+---------------+-------------+---------+-----------+---------+--------
-------------------------+

For this query, we would like to create an additional index called idx_stars on the column Stars from review. From explain, we can see the aggregate function is not using any method or Key so this should be helpful to speed up the query.

e) select((select count(user_id) from user where review_count>10)
/
(select count(user_id) from user))

- Explain on this query produces the result below:

+----+-------------+-------+-------+---------------+---------+---------+------+---------+----------------+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows | Extra |
+----+-------------+-------+-------+---------------+---------+---------+------+---------+----------------+
1	PRIMARY	NULL	NULL	NULL	NULL	NULL	NULL	NULL	No tables used
3	SUBQUERY	user	index	NULL	PRIMARY	22	NULL	1021667	Using index
2	SUBQUERY	user	ALL	NULL	NULL	NULL	NULL	1021667	Using where

From the above we can see that the second subquery is not using any index. So, we can add an additional index on review_count column from User table as this is a non-prime attribute used for where condition.

f) . select (select sum(length(text)))/
(select count(review_id)) from review;

+----+--------------------+--------+------+---------------+------+---------+------+---------+----------------
+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows | Extra |
+----+--------------------+--------+------+---------------+------+---------+------+---------+----------------
+
| 1 | PRIMARY | review | ALL | NULL | NULL | NULL | NULL | 1655155 |
NULL |

| 3 | DEPENDENT SUBQUERY | NULL | NULL | NULL | NULL | NULL | NULL | NULL | No tables used |
| 2 | DEPENDENT SUBQUERY | NULL | NULL | NULL | NULL | NULL | NULL | NULL | No tables used |
+----+--------------------+--------+------+---------------+------+---------+------+---------+----------------

- The query is done on a VARCHAR type column which is not supported by indexing. So, we proceed without adding any additional index.