MET CS 669 Database Design and Implementation for Business Concepts Assignment 5 Instructions: Identifying Index Placement

Assignment Introduction

Virtually all well-designed relational databases perform terribly without deliberate creation of indexes. This means that as a database designer and developer, you must add index creation to your list of skills. This requires a change in thinking. Logical database design deals with the structurally independent tables and relationships; you deal with these according to the soundness of their design, independent of any particular database implementation. Index placement deals with the way the database will access your data, how long that will take, and how much work the database must perform to do so. Creating indexes requires you to think in terms of implementation.

This assignment consists of two sections. The first section teaches you about indexes and where to place them, by example. In the second section, you apply what you learned in the first section with a series of exercises. You do not type SQL in this assignment, but rather identify which columns deserve indexes, the type of index it deserves, and why.

Through the first section, significant tips are denoted with this light bulb icon. 

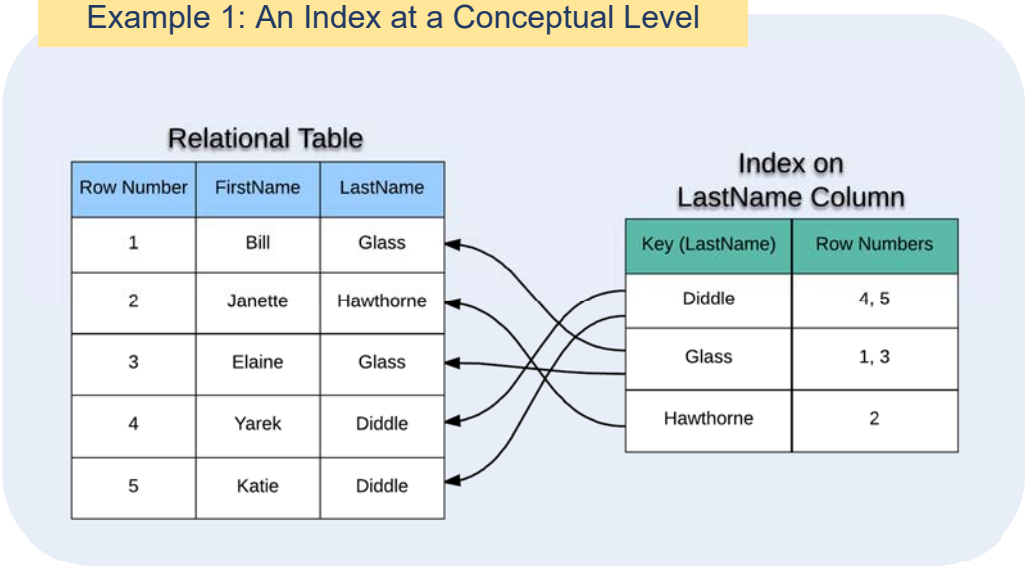
Copyright 2013, 2014, 2016-2017 Boston University. All Rights Reserved.

Section One – Learning by Example

An index is a physical construct that is used to speed up data retrieval. Adding an index to a table does not add rows, columns, or relationships to the table. From a logical design perspective, indexes are invisible. Indexes are not modeled in logical entity-relationship diagrams, because indexes do not operate at the logical level of abstraction, as do tables and table columns. Nevertheless, indexes are critical component in database performance, and database designers need be skilled with index placement, because index placement happens mostly during the design phase of a database. Excellent performance is a critical piece of database design.

Each index is assigned to a column or a set of columns in a single table, and is conceptually a lookup mechanism with two fields. The first field contains a distinct list of values that appear in the covered column or columns, and the second field contains a list of rows which have that value in the table. This is illustrated in Example 1 below.

Example 1: An Index at a Conceptual Level



In Example 1, there are two data columns in the relational table – FirstName and LastName. You may have noticed the RowNumber column as well. Every relational table has an identifier for each row created automatically by the database. For simplicity in this example, the row identifier is a sequential number. There are two columns for the index. The first column contains the unique list of last names – Diddle, Glass, and Hawthorne. The second column contains the list of rows in the relational table that have the corresponding LastName value. For example, the last name “Diddle” corresponds to rows 4 and 5 in the relational table, the last name “Glass” corresponds to rows 1 and 3, and the last name “Hawthrone” corresponds to row 2. Essentially, the index is referencing the corresponding rows in the relational table.

A DBMS can oftentimes use an index to identify the rows that contain the requested value, rather than scanning the entire table to determine which rows have the value. Using an index is usually more efficient than scanning a table. Indexes are perhaps the most significant mechanism for speeding up data access in a relational database.

While there are various kinds of indexes, one kind is so ubiquitous – the B+ tree index – that oftentimes the use of the generic term "index" is actually referring to the B+ tree index. Indexes are categorized partly by their storage characteristics. B+ tree indexes are stored in a data structure known as the B+ tree, which is a data structure known by computer scientists to minimize the number of reads an application must perform from durable storage. If you're curious on how it works technically, you will find ample descriptions on the web (it's beyond the scope of this assignment to go into detail about B+ tree implementation). Bitmap indexes are stored as a series of bits representing the different possible data values. Function-based indexes, which are not categorized by their data storage characteristics, support the use of functions in SQL, and are actually stored in B+ trees. B+ tree indexes are the default kind of index for many modern relational databases, and it's not uncommon for large production schemas to contain only B+ tree indexes. *Please keep in mind that the generic term "index" use throughout this assignment is referring specifically to a B+ tree index and not another kind of index.* When learning about indexes, it's important to understand the kind being referred to.

The definition and categorization of indexes is not without complication. One prominent source of confusion is Microsoft's categorization of "clustered" versus "non-clustered" indexes, found throughout documentation for SQL Server, and in the metadata for schemas in SQL Server installations. A "clustered" index is not an index per se; rather, it is a row-ordering specification. For each table, SQL Server stores its rows on disk according to the one and only one clustered index on that table. No separate construct is created for a "clustered" index; rather, SQL Server can locate rows quickly by the row ordering. A "nonclustered" index is actually just an index by definition, a separate construct that speeds up data retrieval to a table. The terms "clustered" and "nonclustered" are not actually index categorizations, but rather provide a way to distinguish row-ordering specifications from actual indexes. An index is not a row-ordering specification.

Another source of confusion for indexes is the column store, a relatively new kind of way to store data in relational databases. For decades, relational databases only supported row-based storage; the data values for each column in a table row are stored together on disk. Indexes thus reference these rows and enable databases to locate the rows more quickly. Column stores group data values for the same column, across multiple rows, together on disk. Columns stores do not reference values in the underlying table, but actually store the values. Column stores can be beneficial when large numbers of rows are aggregated, as is the case with analytical databases. Some DBMS implement column stores with additional enhancements, such as implementing them fully in-memory or compressing the data values. Although column stores are distinct constructs, they are sometimes confused with indexes because column stores also help speed up data retrieval. In documentation for SQL Server, Microsoft refers to column stores within SQL Server as "column-store indexes", propagating that confusion. Column stores and indexes are two fundamentally distinct constructs that are implemented differently. The key distinction between the two is that indexes reference rows in relational tables, and column stores reference nothing; column stores store the data values themselves in a columnar format which can speed up analytic queries. Columns stores and indexes are two different constructs.

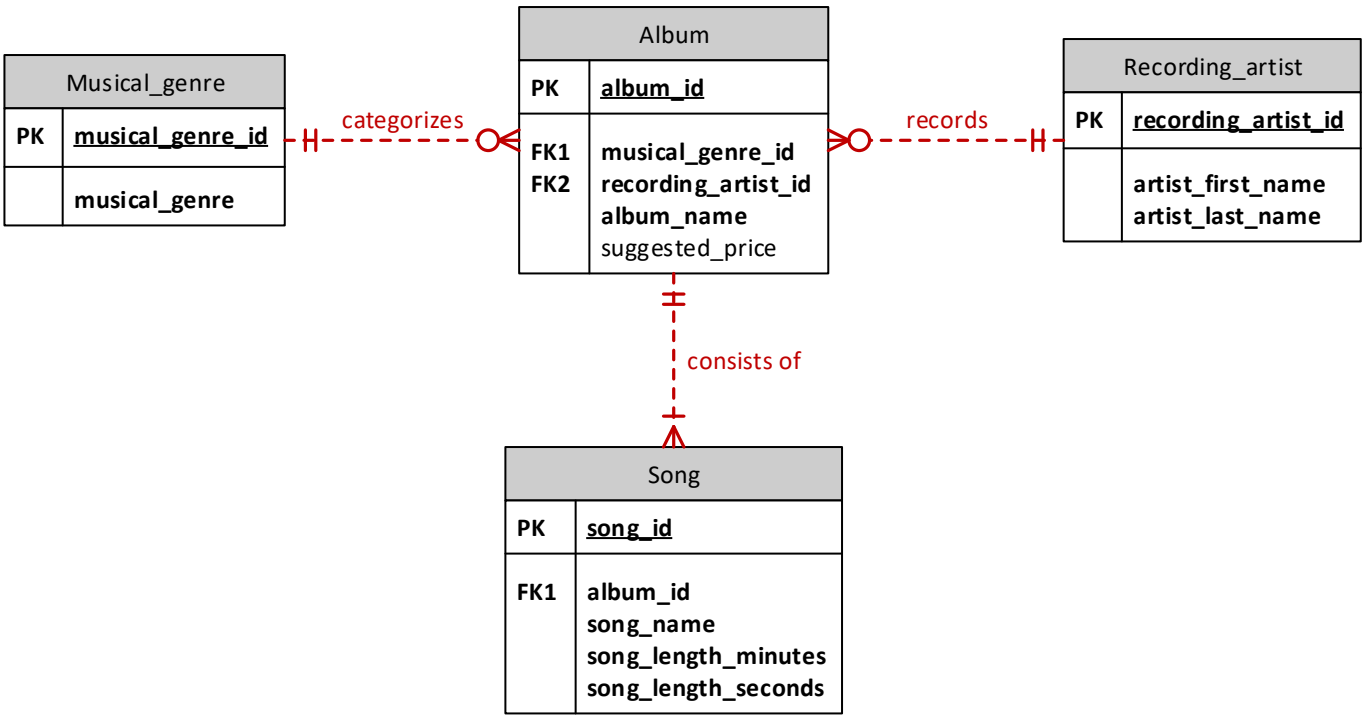
Some indexes have a secondary benefit beyond speeding up data retrieval. An index can be implemented to allow the key values to repeat, or can be implemented to disallow repeating keys. If the key values can repeat, it's a non-unique index; otherwise, it's a unique index. Thus, although not the primary purpose, indexes can be used to enforce uniqueness. In fact, many modern relational DBMS enforce uniqueness constraints through automatic creation of unique indexes. That is, the database designer creates uniqueness constraint (a logical construct), and the DBMS automatically creates a

unique index (a physical construct), if one isn't already present for the column, in order to enforce the constraint. For this reason, DBMS automatically create indexes for primary keys; a primary key constraint is really a combination of a unique constraint and a not null constraint. Enforcing uniqueness is a secondary benefit of indexes.

It's not necessary to create uniqueness constraints to define unique indexes; we can explicitly create unique indexes. Essentially, for every index we create, we decide whether an index is unique or non-unique. Unique indexes obtain better performance than non-unique indexes for some queries, because the DBMS query optimizer knows that each key requested from a unique index will at most have one value, while each key requested from a non-unique index may have many values. Therefore if it is guaranteed that values will not repeat, it is better to use unique indexes. However, adding a unique index on a column that has values that can repeat will cause erroneous transaction abortions every time a repeated value is added to the column. It is important to correctly discern which type of index is needed.

You might reasonably ask the question, "Why not simply add indexes to every column in the schema?" After all, then we would not need to concern ourselves with index placement. The primary reason is that while indexes *speed up reading* from the database, indexes *slow down writing* to the database. Indexes associated with a table slow down writes to that table, because every time data is added to, modified, or deleted from the table, the indexes referencing the data must be modified. Another reason is that indexes increase the size of our database, and that not only affects storage requirements, but also affects database performance since the buffer cache will need to handle the increased size. Yet another reason is that indexes add complexity to database maintenance, especially during structural upgrades. If we need to delete or modify columns for an upgrade, indexes on every column would make the task more complicated. Adding indexes to every column is unnecessary and can cause several issues.

We will now work through a series of examples using the album schema defined below.



Primary Keys

For indexes, you need not consider *all* columns in a table, only most of them. Many modern relational DBMS, including Oracle and SQL Server, automatically add unique indexes to table columns covered by a primary key constraint. We do not need to add indexes to primary keys, since the DBMS will create them automatically for us.

Example 2: Identifying Primary Keys

So that we know which columns would already have index on them in the album schema, we'll identify the primary key columns. We use the standardized dot notation familiar to database professionals, TableName.ColumnName.

Primary Key Column	Description
Musical_genre.musical_genre_id	This is the primary key of the Musical_genre table.
Album.album_id	This is the primary key of the Album table.
Recording_artist.recording_artist_id	This is the primary key of the Recording_artist table.
Song.song_id	This is the primary key of the Song table.

Notice that in Example 2, the indexes are implicitly unique indexes since primary key values must always be unique.

Foreign Keys

Deciding where to place indexes requires careful thought for some table columns, but there is one kind that requires no decision at all – foreign key columns. All foreign key columns should be indexed without regard to any other requirements or the SQL queries that will use them. Some DBMS, including Oracle, will sometimes escalate a row-level lock to a page-level lock when a SQL join is performed using a foreign key that has no index. The focus of this assignment is not locking, so I will not get into fine details, but suffice it to say that page-level locks are always bad for transactions because they result in deadlocks over which the database developer has no control. Another reason we index all foreign key columns is because foreign keys will almost always be used in the WHERE clauses of SQL queries that perform joins between the referencing tables and the referenced tables. The simple rule is to always index foreign key columns.

Let us look at an example of indexing foreign keys.

Example 3: Adding Foreign Key Indexes

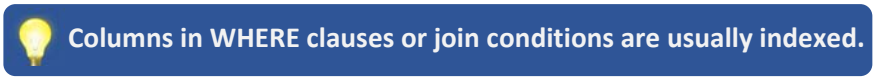
In this example, we identify all foreign key columns in the album schema. Unlike primary keys, foreign keys are not always unique, so we need to also indicate whether a unique or non-unique index is required. Below is a listing of the foreign key column indexes.

Foreign Key Column	Description
Album.musical_genre_id	This foreign key in the Album table references the Musical_genre table. The index is non-unique since many albums can be in the same genre.
Album.recording_artist_id	This foreign key in the Album table references the Recording_artist table. The index is non-unique since a recording artist can produce many albums.
Song.album_id	This foreign key in the Song table references the Album table. This index is non-unique since there are many songs in an album.

You may have noticed that all of the foreign key indexes in Example 3 are non-unique. In practice, most indexes are non-unique because most columns are not candidate keys.

Query Driven Index Placement

Columns that are considered neither primary nor foreign key columns must be evaluated on a case-by-case basis according to more complex criteria. It starts with a simple rule: *every column that will be referenced in the WHERE clause or any join condition of any query is usually indexed*. The WHERE clause and join conditions in a SQL query contain conditions that specify what rows from the tables will be present in the result set. The query optimizer makes heavy use of these conditions to ensure that the results are retrieved in a timely fashion. For example, if the underlying table has a billion rows, but a condition in the WHERE clause restricts the result set to five rows, a good plan from the query optimizer will only access a small number of rows in the underlying table as opposed to the full billion rows, even if many other tables are joined in the query. We intelligently select columns to index based upon how they are expected to be used in queries, and how we expect the database will execute those queries.



You probably noticed the word “usually” in the rule described above, hinting that the rule is not so simple after all. While we can safely exclude columns that are *not* used in WHERE clauses or join conditions, indexing columns that *are* used in those constructs is usually but not always useful. There are other factors to consider. A simple criterion drives indexing decisions: *add an index to a column if it will speed up SQL queries that use it more than a negligible amount.* If it does not meet this criterion, adding an index is not useful, and would be slightly detrimental since it increases the database size and slightly slows down writes to the table. Essentially, we need to discern whether or not the database will make use of the index. If we create the index and the database does not use it, or if the database does use it

but doing so does not increase performance, the index is not beneficial. Simple truth gives way to complexity in regards to indexing.

Table size isn't a significant factor in logical database design, but it is certainly a significant factor for index placement. Databases usually do not use indexes for small tables. If a table is small enough to fit on just a few blocks, typically no more than a few hundred rows or few thousand rows depending upon the configuration, the database often determines that it's more efficient to read the entire table into memory than to use an index on that table. After all, using an index requires that database to access one or more blocks used by the index, then additionally access the blocks needed for the table rows. This may be less efficient than reading the entire table into memory and scanning it in memory. Small lookup tables or tables that will never grow beyond a few hundred or a few thousand rows may not need an index. We must consider a table's size before adding indexes to it.

Indexes on tables that remain small are usually not beneficial.

Large tables do not always benefit from indexes. Another factor databases use to determine whether or not to use an index is the percentage of rows pulled back by queries that access it. Even when a table is quite large, if the queries that access it read in most rows of the table, the database may decide to read in the entire table into memory, ignoring indexes. If most rows of the table will be retrieved, it's often more efficient for the database to read the entire table into memory and scan it than to access all the blocks for an index and read in most rows of the table. Typically, day-to-day business systems will access a small subset of rows in queries, and analytical systems pull back large subsets of the table to aggregate results (these rules will of course sometimes broken). So we consider the type of system, and what that system will do with the particular table, to decide whether adding an index is beneficial.

If queries always retrieve most rows in a table, indexes on that table are not usually beneficial.

If the number of distinct values in the column is very small, it's usually not beneficial to index the column, even on large tables, even when queries access a small number of rows. For example, imagine a "gender" column with two possible values on a billion-row table. If a single row was being requested, what good would it do the database to narrow down the search to 500,000 rows using the index? Not to mention, an index with a single entry that references 500,000 rows would not be efficiently accessed. An index is beneficial if it can be used to narrow down a search to a small number of rows (relative to the size of the table). If an index only cuts the table in half, or into a few partitions, it's not beneficial. Indexes must subdivide the table into enough partitions to be useful.

If most values for a column are null, it's not usually beneficial to index the column. The reason for this is actually the same as in the prior paragraph. As far as the database is concerned, a large number of null values is just another large partition identified by the index. For example, if a column for a billion-row table has 900,000 nulls, then the database could use the index, but would only narrow down the search to 900,000 rows whenever the value is null; this is not useful! An additional complication is that some DBMS do not add null entries to indexes, and some do, so some DBMS cannot take advantage of the index when a value is null regardless of the number of nulls. The percentage of null values should be taken into account when deciding which columns deserve an index.

Avoid indexing columns with a small number of distinct values or a large percentage of nulls.

In practice, one does not usually peruse the thousands of queries written against a schema to determine which columns to index. We need to decide what to index when a system is being created, before all the queries are written. We can usually spot the columns that are likely to be used in the where clause or in join conditions and provide indexes for those without viewing queries. For example, many systems would support searching on a customer's name, but would not support searching on optional information such as order comments that customers may for an order. It would be reasonable for us to index the first and last name columns, and to avoid indexing an order_comment field. Of course, you need to know the system and how queries will generally use the database fields in order to make this determination. A systematic review of all of a system's queries is not required to place many of the indexes in a database schema.

This is not to say that every index for a database schema is created at design time. Sometimes during development and maintenance of a system, a new query is written that does not perform well, and we must add an index to support the query. We must understand how to correctly place indexes given a specific query. The process of adding indexes can be categorized as adding most indexes upon design of the database, and adding some indexes iteratively over time as new queries demand them. Adding indexes is an iterative process.

You may have discerned that although certain principles are in play, index placement is not always an exact science. I will reiterate the guiding principle once again: *add an index to a column if it will speed up SQL queries that use it more than a negligible amount.* Confirming this may require some before and after testing for some indexes, or researching the system that will use it. If you apply this principle to every index you create, you will be creating useful indexes throughout your database, and improving database performance.

Let us now work through examples where specific queries are observed and indexes are created for those queries.

Example 4: Query by Song Name

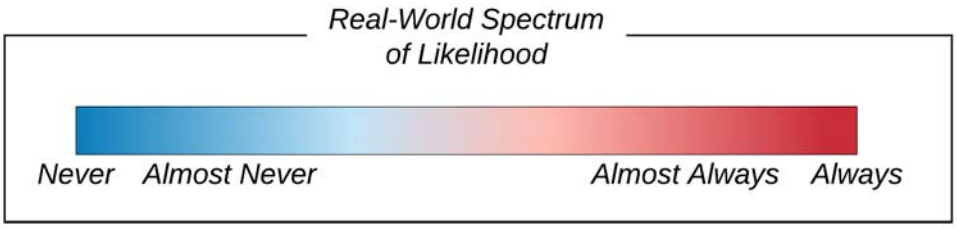
In this example, there is single table query that retrieves the minutes and seconds for the "Moods for Moderns" song in the album schema.

```
SELECT Song.song_length_minutes, Song.song_length_seconds
FROM   Song
WHERE  Song.song_name = 'Moods For Moderns'
```

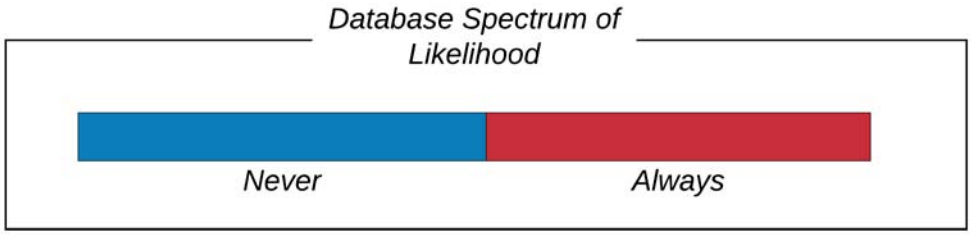
Three columns are accessed in this query – s*ong_length_minutes, song_length_seconds, and song_name* – but only *song_name* is in the WHERE clause. Therefore we would index the *song_name* column. The database can use *song_name* to locate the correct rows in the table, and once the rows are located, can retrieve additional columns including *song_length_minutes* and *song_length_seconds* without searching again. Adding indexes for the other two columns would not benefit this query.

We would create this index as a non-unique index, because it is possible that a song name can repeat.

Because song names are almost always unique (artists usually give each new song a unique title), one might lean toward creating a unique index in Example 4. In the real-world "almost always" is considered quite close to "always" in spectrum of likelihood of occurrence, demonstrated in the figure below.



However in the database world the two are fundamentally different, because we are dealing with absolutes. This is illustrated in the figure below.



If something can happen even once, then we must accommodate for it. For example, several artists have sung the song "Amazing Grace", so if we add a unique index to *song_name*, we would prevent these songs from being input into our database. It is important not to apply shades of gray to the choice of indexes, because a unique index requires 100% conformity.

Now for another example.

Example 5: Query by Album Name

In this example, the query retrieves the artist name for the "Power Play" album.

```
SELECT Recording_artist.artist_first_name, Recording_artist.artist_last_name
FROM   Album
JOIN   Recording_artist
ON     Album.recording_artist_id = Recording_artist.recording_artist_id
WHERE  Album.album_name = 'Power Play'
```

*Album.recording_artist_*id and *Recording_artist.recording_artist_id* are both used in a join condition so are candidates for indexing. However, the former is a foreign key, and the latter is a primary key; we already marked these columns for indexing in Example 1 and Example 2. This also demonstrates that foreign keys are typically used in join conditions.

Album.album_name appears in the WHERE clause (this how the query limits the results to the "Power Play" album), so we will want to index it. Similar to song names, album names are usually unique, but not always, so we make the index non-unique.

Queries with subqueries can be complex, but subqueries affect indexing in a predictable way. The WHERE clause and join conditions for the outer query, and for all subqueries, collectively determine what rows from the underlying tables in the schema will be retrieved. Hence all columns in the WHERE clause and join conditions for the subqueries are candidates for indexing, in addition to those in the outer query. In terms of indexing, subqueries add more layers, but do not change the strategy.

Now for a more complex example of a query that contains a subquery.

Example 6: Albums by Song Length

This query lists the names of all albums that have songs that are less than four minutes in length.

```
SELECT Album.album_name
FROM   Album
WHERE  Album.album_id
       IN (SELECT Song.album_id
           FROM   Song
           WHERE  Song.song_length_minutes < 4)
```

Because *song_length_minutes* is in the WHERE clause of the subquery, we add an index to that column. We use the same strategy for the subquery as with the outer query. *Album.album_id* is the primary key of album, and so was previously indexed in Example 1.

Even though the less-than sign "<" is used in Example 6 rather than equality, the database can still take advantage of an index on the column. The reason is that the index is sorted, and the database can take advantage of the sorting to efficiently locate all songs less than 4 minutes in length.

Now, on the second section, where you can apply what you have learned.

Section Two – Learning by Doing

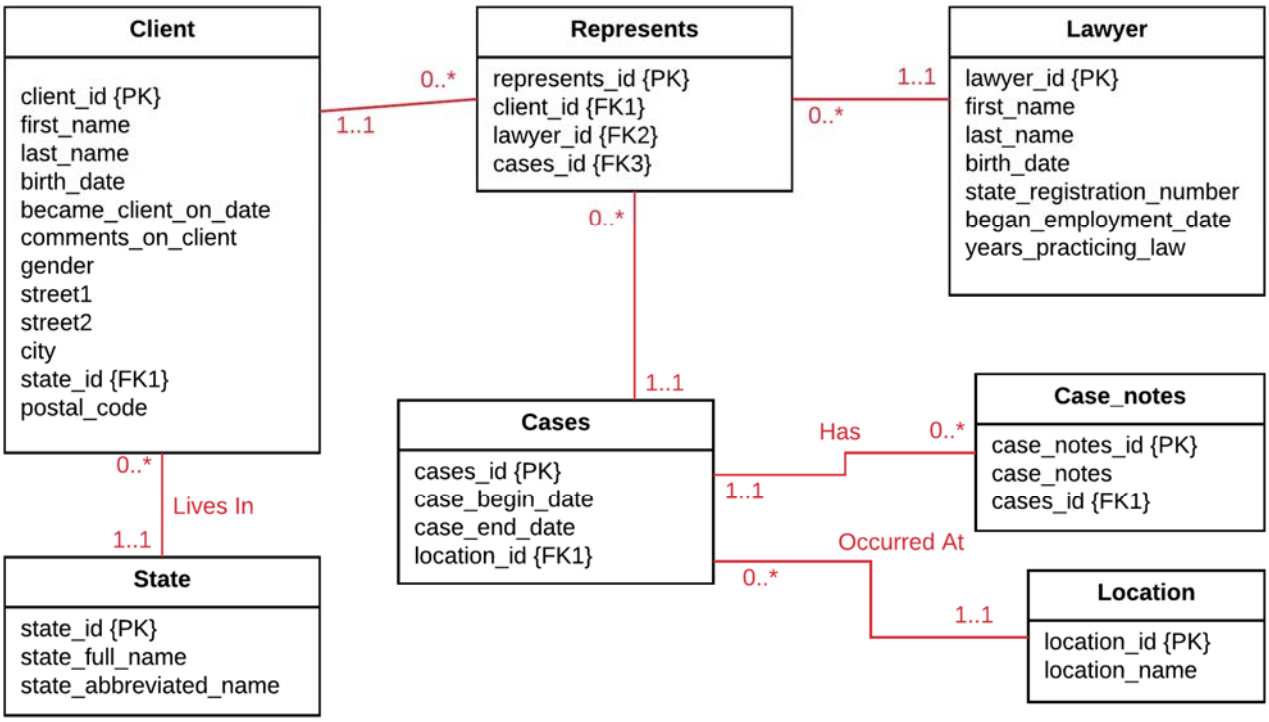
In this section, you apply what you learned in the first section with a series of exercises. You will be deciding index placement for a large law firm's relational schema and associated application.

Overview

The law firm represents each of its client's cases with one or more of its lawyers. A client's cases are tracked, with notes entered by the firm for the case as needed. The law firm handles cases in a large variety of locations, including court rooms throughout the state, and other locations as needed. The firm's clients all do business in the state, but live all over the country.

Entity Relationship Diagram

The schema's ERD is listed below (in a modified UML class diagram).



The *Client* table contains data for the clients represented by the law firm, including the client's name, birth date, gender, address, the date the client started working with the law firm, any special comments the law firm records about the client. The *State* table contains both a full name and an abbreviated name for the states in which a client can reside (some clients live out of state but do business in the state). The *Lawyer* table contains data for the lawyers employed by the firm, including their name and birth date, their registration number for the state, the number of years they have practiced law, and when they began employment with the firm.

The *Cases* table contains data for the cases handled by the law firm, including when the case began and ended (case_end_date will be null until the case ends), and the location the case is held. The *Location* table contains any location where a case can be held, including names of courts and other locations. The *Case_notes* table is important to the firm because allows the firm to record any number of notes

(comments) about a case, which is useful for strategizing about the case, or recording special adjustments or areas that need special attention. Lastly, the *Represents* table keeps track of which lawyers represent which clients for which cases.

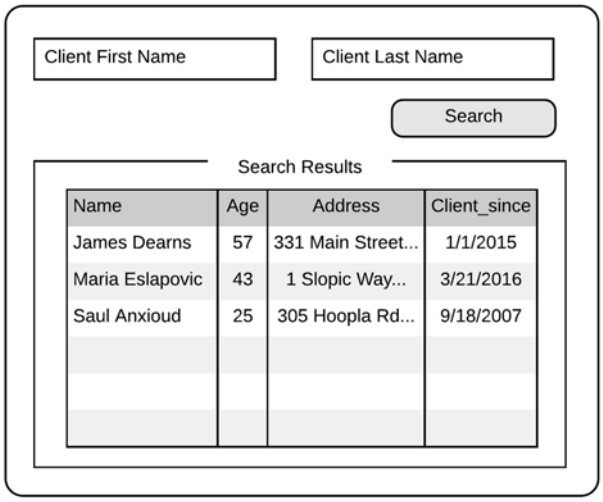
Application

The application developed for the firm, and which uses this schema, is named LawTrax. It is a day-to-day business system that allows the firm to track clients, their cases, and the lawyers that represent them. The data is updated throughout each day with a series of data entry screens as new clients hire the firm, as cases progress, and as lawyers join or leave the firm. Other than supporting the aforementioned data entry screens, LawTrax also supports searching for clients, cases, and lawyers.

The application development team was able to pull out two queries to help you. The first is one of the queries used by the search screen that allows LawTrax users to search for specific clients by name.

```
SELECT *
FROM   Client
WHERE  Client.last_name = ?
AND    Client.first_name = ?;
```

Note that the question marks (?) in the query are parameters that are passed to the query dynamically at runtime. They are populated with whatever values the user enters on the search screen. The team managed to capture a screenshot for the screen as well, which is the following:



The team also pulls out a query LawTrax uses to list out the most recent cases that have not yet been closed, along with the lawyers that represent the case.

```
SELECT *
FROM   Cases
JOIN   Case_notes ON Case_notes.cases_id = Cases.cases_id
JOIN   Location   ON Location.location_id = Cases.location_id
JOIN   Represents ON Represents.cases_id = Cases.cases_id
JOIN   Lawyer     ON Lawyer.lawyer_id = Represents.lawyer_id
WHERE  Cases.case_begin_date > ?
AND    Cases.case_end_date IS NULL;
```

Again, the question mark (?) is used by LawTrax to dynamically put in a date at runtime, rather than hardcoding one specific date.

When the system first launched, the performance of LawTrax was good, because not much data was present. As time has gone on, however, system performance has gradually decreased to the point where it is now unbearably slow, and the system has become mostly unusable. The law firm has hired you as a consultant to urgently help fix the performance issues. To your horror, a quick scan of the database's metadata reveals that the database has no indexes except the ones automatically created by the database. The database designer created no indexes at all! You quickly get to work deciding where to place the indexes.

1. To get started, list out all of the primary key columns in the schema. These have already been indexed by the database. Identify them using the standard tablename.columnname format as used in Examples 2 and 3 in the first section.

2. Next, you need to identify all of the foreign key columns in the schema. As described in the first section, these all need to be indexed. You will need to decide whether to make them unique indexes, or non-unique indexes. List out all foreign key columns in tablename.columnname format, indicate for each whether a unique or non-unique index is needed, and explain your choice.

3. You have been provided with some queries by the development team; they are a good resource to continue your index work. Indicate which columns would need to be indexed for the given queries using the tablename.columname format. Make sure to indicate whether the indexes should be unique or non-unique, and to explain your choice.

4. At this point, you have seen a couple of queries used by the system, and have read a description of what the system does. Your next step is to identify all remaining columns that should be indexed based upon this information. You may recall that the original database designer did not index any columns, so all remaining columns should be considered. In order to accomplish this, you will need to make reasonable assumptions about what kinds of queries the system uses in order to do its work. State your assumptions, then list all additional columns (in tablename.columnname format) that need an index. Explain why the index is beneficial, taking into account the factors described in the first section. Also identify whether the index should be unique or non-unique, and justify that choice.

Congratulations! You have learned how to determine what columns need indexes, and this skill is extremely important for relational database design. Database developers and designers regularly apply the techniques you just learned while developing and maintaining databases.