MET CS 669 Database Design and Implementation for Business
Term Project Iteration 5

Contents

Authored by Warren Mansur.

Iteration Introduction

This iteration is about putting data into your database and answering questions from the data. Your design structure was implemented in Iteration 4 through creation of your tables, attributes, and constraints. In this iteration, you insert data transactionally with stored procedures, define questions useful to the organization or application, answer them with well-written queries, and add indexes to help in their performance. After your hard work, you get to see your database in action!

To help you keep a bearing on what you have left to complete, let's again look at an outline of what you created in prior iterations, and what you will be creating in this iteration.

Prior Iterations	Iteration 1	*Project Direction Overview* – You provide an overview that describes who the database will be for, what kind of data it will contain, how you envision it will be used, and most importantly, why you are interested in it. *Use Cases and Fields* – You provide use cases that enumerate steps of how the database will be typically used, also identify significant database fields needed to support the use case. *Summary and Reflection* – You concisely summarize your project and the work you have completed thus far, and additionally record your questions, concerns, and observations, so that you and your facilitator or instructor are aware of them and can communicate about them.
	Iteration 2	*Structural Database Rules* – You define structural database rules which formally specify the entities, relationships, and constraints for your database design. *Conceptual Entity Relationship Diagram (ERD)* – You create an initial ERD, the universally accepted method of modeling and visualizing database designs, to visualize the entities and relationships defined by the structural database rules.
	Iteration 3	*Conceptual Extended Entity Relationship Diagram (EERD)* – You add specialization-generalization into your conceptual ERD and structural database rules. *Initial DBMS Physical ERD* – You create an initial DBMS physical ERD, which is tied to a specific relational database vendor and version, with SQL-based constraints and datatypes.
	Iteration 4	*Full DBMS Physical ERD* – You define the attributes for your database design and add them to your DBMS Physical ERD. *Normalization* – You normalize your DBMS physical ERD to reduce or eliminate data redundancy. *Database Structure* – You create your tables, sequences, and constraints in SQL.
Current and Future Iterations	Iteration 5	*Reusable, Transaction-Oriented Store Procedures* – You create and execute reusable stored procedures that complete the steps of transactions necessary to add data to your database. *Questions and Queries* – You define questions useful to the organization or application that will use your database, then write queries to address the questions. *Index Placement and Creation* – To speed up performance, you identify columns needing indexes for your database, then create them in SQL.
	Iteration 6	*History Table* – You create a history table to track changes to values, and develop a trigger to maintain it. *Data Visualizations* – You tell effective data stories with data visualizations.

First, make any revisions to your design and scripts that you are necessary before you proceed further. The more SQL you implement, the harder it becomes to change your design later.

Transaction Driven, Reusable Stored Procedures

Recall that a stored procedure is a named code segment that can be executed by name as needed. Stored procedures can contain SQL code and business logic as needed. Stored procedures can take parameters whereby the caller specifies the values when executing the procedure. Stored procedures provide many useful features.

Also recall that a transaction is a logical unit of work consisting of a series of steps for the purpose of accomplishing a task. Organizations group work together in a transaction to transition the database from one consistent state to the next, in a way that is atomic, consistent, and durable (since transactions obey the ACIDS requirements). Transactions are the backbone of modern relational database processing.

Some organizations use a method whereby transactions are implemented in reusable stored procedures. Such a stored procedure takes one or more parameters, and completes all of the steps in the transaction using the values the caller specified in the parameters. Use of this method allows database developers to implement the transactions in the database, eliminating the need for the application to implement all of the steps. Instead, the application executes the stored procedure for the appropriate transaction with the correct parameters.

There are other advantages to using this method. For one, it's not necessary for the application to connect over the network repeatedly to execute SQL, resulting in better performance. For another, transaction steps can be updated without the need to redeploy the application. For another, I.T. staff can execute a transaction manually when necessary to correct an issue, or for other purposes, such as migrating data from another source. This method carries many advantages.

Let's look at a simple example with the Car database we've been developing in this iteration. Imagine for the Car database there is the following simple use case, for when a Ferrari is ready to be entered into this system.

Add Ferrari Use Case

1. The car reseller purchases a used Ferrari to sell.
2. The car salesperson enters the Ferrari's information into the system including its make, model, and other important information.

What kind of transaction would support this use case? Simple! We know that we have two tables – Car and Ferrari – that are relevant to Ferraris, so we need two insert statements for each of those tables in our transaction. What about tying this into a stored procedure? Again, simple! We give the stored procedure parameters for every field in both of those tables.

For Oracle, the transaction would be implemented in a stored procedure as follows.

Add Ferrari Stored Procedure (Oracle)

```
CREATE OR REPLACE PROCEDURE AddFerrari(CarID IN DECIMAL, VIN IN VARCHAR, Price IN DECIMAL,
  Color IN VARCHAR, Make IN VARCHAR, Model IN VARCHAR, Mileage IN DECIMAL)
AS
BEGIN
  INSERT INTO Car(CarID, VIN, Price, Color, Make, Model, Mileage)
  VALUES(CarID, VIN, Price, Color, Make, Model, Mileage);

  INSERT INTO Ferrari(CarID)
  VALUES(CarID);
END;
```

Notice that the stored procedure takes a parameter for every column, then inserts those values into the Car and Ferrari tables.

We would execute the stored procedure in Oracle as follows.

Execute Ferrari Stored Procedure (Oracle)

```
BEGIN
   AddFerrari(1, '1XPADB9X4TN402579', 295000, 'Giallo Modena',
  'Ferrari', 'F12berlinetta', 20000);
  COMMIT;
END;
```

We use a BEGIN/END block, then pass in the parameter values we are interested in to execute the stored procedure. After calling the stored procedure (which will implicitly begin a transaction), we issue the COMMIT keyword to commit the results of the transaction. In Oracle, the first SQL statement encountered implicitly starts the transaction, and then the COMMIT statement commits the active transaction.

I chose a fictional, but realistic, example of a Ferrari that the reseller may sale. The values are indicated below.

Parameter	Value
CarID	1
VIN	1XPADB9X4TN402579
Price	$295,000
Color	Giallo Modena
Make	Ferrari
Model	F12berlinetta
Mileage	20,000

For SQL Server, the transaction would be implemented in a stored procedure as follows.

Add Ferrari Stored Procedure (SQL Server)

```
CREATE PROCEDURE AddFerrari @CarID DECIMAL(12), @VIN VARCHAR(17), @Price DECIMAL(8,2),
@Color VARCHAR(64), @Make VARCHAR(64), @Model VARCHAR(64), @Mileage DECIMAL(7)
AS
BEGIN
  INSERT INTO Car(CarID, VIN, Price, Color, Make, Model, Mileage)
  VALUES(@CarID, @VIN, @Price, @Color, @Make, @Model, @Mileage);

  INSERT INTO Ferrari(CarID)
  VALUES(@CarID);
END;
```

Just as with Oracle, the stored procedure takes a parameter for every column, then inserts those values into the Car and Ferrari tables.

We would execute the stored procedure in SQL Server as follows.

Execute Ferrari Stored Procedure (SQL Server)

```
BEGIN TRANSACTION AddFerrari;
EXECUTE AddFerrari 1, '1XPADB9X4TN402579', 295000,
  'Giallo Modena', 'Ferrari', 'F12berlinetta', 20000;
COMMIT TRANSACTION AddFerrari;
```

The BEGIN TRANSACTION block starts a transaction, named “AddFerrari”. The EXECUTE block executes the AddFerrari stored procedure and passes in the same fictional parameters previously defined. The COMMIT TRANSACTION block commits the transaction.

For Postgres, the transaction would be implemented in a function as follows.

Add Ferrari Stored Procedure (Postgres)

```
CREATE OR REPLACE PROCEDURE AddFerrari(CarID IN DECIMAL, VIN IN VARCHAR, Price IN DECIMAL,
  Color IN VARCHAR, Make IN VARCHAR, Model IN VARCHAR, Mileage IN DECIMAL)
AS
$proc$
BEGIN
  INSERT INTO Car(CarID, VIN, Price, Color, Make, Model, Mileage)
  VALUES(CarID, VIN, Price, Color, Make, Model, Mileage);

  INSERT INTO Ferrari(CarID)
  VALUES(CarID);
END;
$proc$ LANGUAGE plpgsql
```

Just as with the other databases, the stored procedure takes a parameter for every column, then inserts those values into the Car and Ferrari tables.

We would execute the stored procedure in Postgres as follows.

Execute Ferrari Stored Procedure (Postgres)

```
START TRANSACTION;
DO
 $$BEGIN
   CALL AddFerrari(1, '1XPADB9X4TN402579', 295000, 'Giallo Modena',
   'Ferrari', 'F12berlinetta', 20000);
 END$$;
COMMIT TRANSACTION;
```

The START TRANSACTION block starts a transaction. The DO command along with the CALL command executes the AddFerrari stored procedure and passes in the same fictional parameters previously defined. The COMMIT TRANSACTION block commits the transaction.

Note that the stored procedures provided in this section have the basic transaction steps, but do not implement error checking and other features that a more robust implementation would implement. *In your implementation, it would be wise to implement this more robust features.*

In summary, one methodology for implementing a transaction is to implement it in a parameterized stored procedure, execute the stored procedure with the necessary values, and surround the store procedure calls with transaction control statements to start and commit the transaction.

Implementing Transactions in your Database

Now that you know a common methodology for implementing transactions, you can take advantage of this to start populating your database with data. Select two of your use cases that involve adding data to the database, create parameterized stored procedures that implement the transactions steps, and execute the stored procedures in the context of a transaction. Provide screenshots of their creation and execution, and also attach a SQL script with them attached.

Below is a sample of one such implementation with TrackMyBuys. Note that only one example is provided in TrackMyBuys for illustrative purposes, but you are asked to create three.

TrackMyBuys Transaction

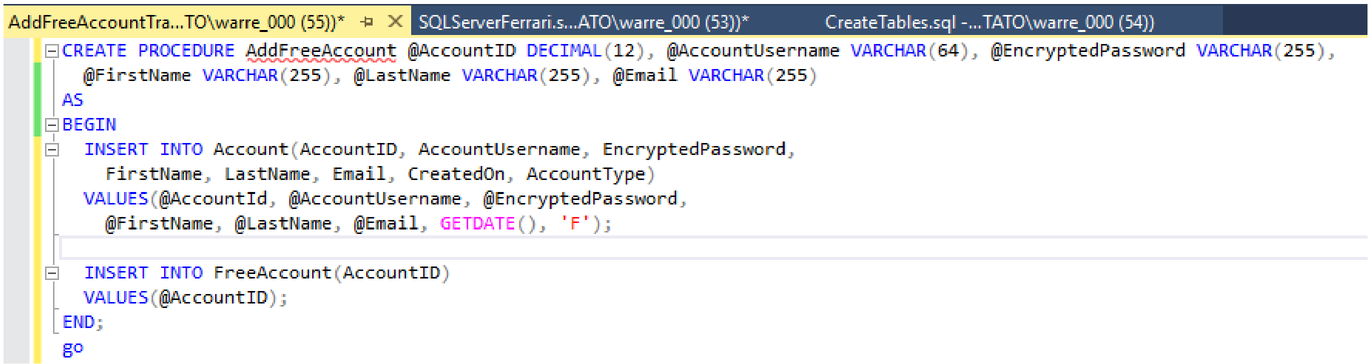
The first use case for TrackMyBuys is the account signup use case listed below.

Account Signup/Installation Use Case

1. The person visits TrackMyBuys' website or app store and installs the application.
2. The application asks them to create either a free or paid account when its first run.
3. The user selects the type of account and enters their information and the account is created in the database.
4. The application asks them to install browser plugins so that their purchases can be automatically tracked when they make them.

For this use case, I will implement a transaction that creates a free account, using SQL Server.

Here is a screenshot of my stored procedure definition.


AddFreeAccountTra...TO\warre_000 (55))*
SQLServerFerrari.s...ATO\warre_000 (53))*
CreateTables.sql -...TATO\warre_000 (54))

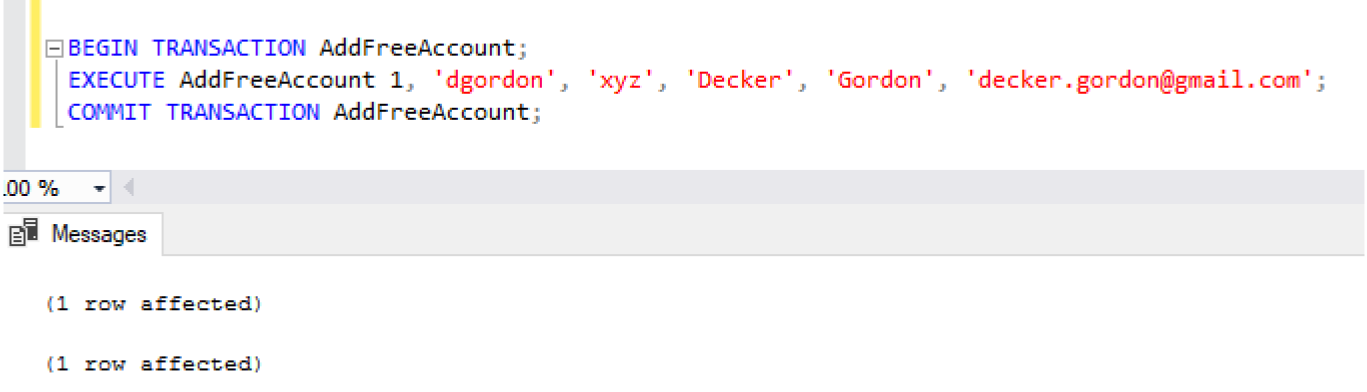
```
CREATE PROCEDURE AddFreeAccount @AccountID DECIMAL(12), @AccountUsername VARCHAR(64), @EncryptedPassword VARCHAR(255),
  @FirstName VARCHAR(255), @LastName VARCHAR(255), @Email VARCHAR(255)
AS
BEGIN
  INSERT INTO Account(AccountID, AccountUsername, EncryptedPassword,
    FirstName, LastName, Email, CreatedOn, AccountType)
  VALUES(@AccountId, @AccountUsername, @EncryptedPassword,
    @FirstName, @LastName, @Email, GETDATE(), 'F');

  INSERT INTO FreeAccount(AccountID)
  VALUES(@AccountID);
END;
go
```

I name the stored procedure “AddFreeAccount”, and give it parameters that correspond to the Account and FreeAccount tables. Since CreatedDate is always the current date, I do not need a parameter for that, but instead use the GETDATE() function in SQL Server. Since this procedure is always for a free account, I do not use a parameter for AccountType, but hardcode the character “F”.

Inside the stored procedure, there are two insert statements to insert into the two respective tables.

Here is a screenshot of my stored procedure execution.



```
BEGIN TRANSACTION AddFreeAccount;
EXECUTE AddFreeAccount 1, 'dgordon', 'xyz', 'Decker', 'Gordon', 'decker.gordon@gmail.com';
COMMIT TRANSACTION AddFreeAccount;
```

```
(1 row affected)

(1 row affected)
```

I add a fictional person named Decker Gordon with other fictional but realistic information. I nested the stored procedure call between transaction control statements to ensure the transaction is committed.

Populating Tables

The stored procedures you implement for some of your use cases can be used to populate some of your tables. The remainder of the tables you access in your queries also need be populated. You may perform basic inserts, or create stored procedures, whatever works best for your project, to populate these with data. *All data tables used by your queries should have at least 5 rows in them with some variety between the rows. Domain/lookup tables should have the number of rows that naturally fit the table.* You do not need to populate tables in your schema that are not accessed by any of your queries.

For example, an Address table would be considered a data table because it is populated as needed when new addresses are added to the database. The Address table may reference a State table to identify which U.S. state the address resides in. The State table is a domain/lookup table because it is not populated as needed, but is populated ahead of time with the 50 U.S. states. This table would naturally contain 50 rows. Other domain/lookup table examples include eye color or gender, because these have a predefined list of values. So, in an operational database, domain/lookup tables are populated upfront with the full list of values they may have. Domain/lookup tables are usually a fixed size, bounded by the number of values they need. Data tables are populated as information comes in, and typically keep growing over time.

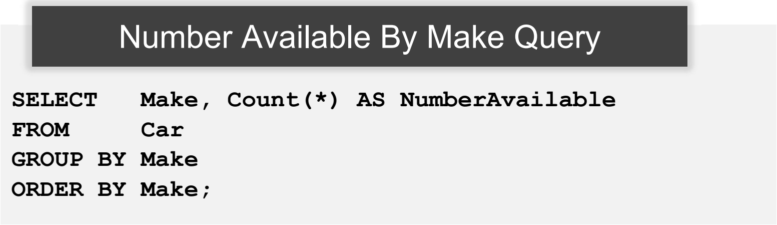
All inserts, whether through stored procedures or direct inserts, should use sequences throughout for the primary and foreign key values. No values should be hardcoded. Foreign key values can be obtained by using the current value of the appropriate sequence, or through subquery lookups.

Organization-Driven Queries

Storing the data in a database has little value if that's where it ends; what's important is using the data. Queries are king in a relational database because they are the tool for pulling out relevant information and using it. While one could write queries that are theoretically beautiful or demonstrate powerful SQL constructs, that's not a useful goal for an organization. An organization needs queries that get it the data it needs. The beauty and the power are merely tools that help accomplish the goal. Therefore, just as with all of our design work, we focus on writing queries that are based on what the organization needs and how the database will be used. "Useful to the organization" is a broad concept, but for purposes of this project, we'll create queries based upon questions we deem useful for the organization and/or application.

Let's look at an example, again from our Car database. A question that is a reasonable use of the system for the car reseller is: how many cars of each make are available? Such a question would be commonly asked because customers and staff alike will want to know the answer to this question. Perhaps a customer only wants to look at Aston Martins so wants to know the number available. Perhaps a customer hasn't decided on a make and wants to have an idea of how many the reseller has of each before visiting the showroom. Perhaps a manager wants to know the answer before purchasing more cars. It's obvious that the answer to this question has many useful applications.

To answer this question, we write a query that takes advantage of aggregation.



Number Available By Make Query

```
SELECT   Make, Count(*) AS NumberAvailable
FROM     Car
GROUP BY Make
ORDER BY Make;
```

This query selects from the Car table, groups and orders by the make column, and counts the number available for each make. The query is simple; however, we could envision many variations on this. For example, by ordering by the NumberAvailable column instead of Make, we could find out a ranking from most available to least available.

Creating Questions and Queries for your Database

Create three questions useful to your organization or application, then write queries to address the questions. Requirements for the three questions are listed below. In order to show useful results, you will need enough rows and variety in your tables. For example, if you are limiting by a certain column, some rows should match the condition, and some rows should not match. If you are using aggregates, there should be enough rows to help prove out the aggregate, and the grouped by column should have more than one distinct value. You want to help prove that your query works by providing useful data.

First Query

This query should retrieve information from at least four tables joined by associative relationships. Ensure that the question driving this query is applicable and useful for the intended use of the database.

Second Query

This query should retrieve information from the subtypes and supertype you have in your project, the entities that participate in the specialization-generalization relationship. The query should require joins to one or more of the subtypes, as well as the supertype, to fully address the question.

Ensure that the question driving this query is applicable and useful for the intended use of the database.

Third Query

Create a view that captures information that needs be accessed regularly based upon the use of your database. Utilize the view in the query. To ensure sufficient complexity, the query (or underlying view) should contain at least two of the constructs below. There should be at least one from each group.

Group 1 (choose one or more)

- joins of at least two tables.
- one or more restrictions in the WHERE clause.
- an order by statement.

Group 2 (choose one or more)

- at least one aggregate function.
- at least one subquery.
- a having clause.
- a left or right join.
- joins of four or more tables.
- a union of two queries.

If you'd like to use a SQL construct not listed here, that's great, but just run the idea by your facilitator or instructor to ensure it will meet the complexity requirements.

Tips for This Section

The usefulness and complexity of the queries will be a factor in the evaluation of this section. Try to focus on the most useful questions with reasonable complexity requirements.

Add your queries to your SQL script. Each query should have an associated comment which describes the question it's answering. Also provide screenshots of the execution of each query and its results in your design document. Make sure to explain the logic of your query and how it shows the expected results.

Below I give an example of a question useful to TrackMyBuys.

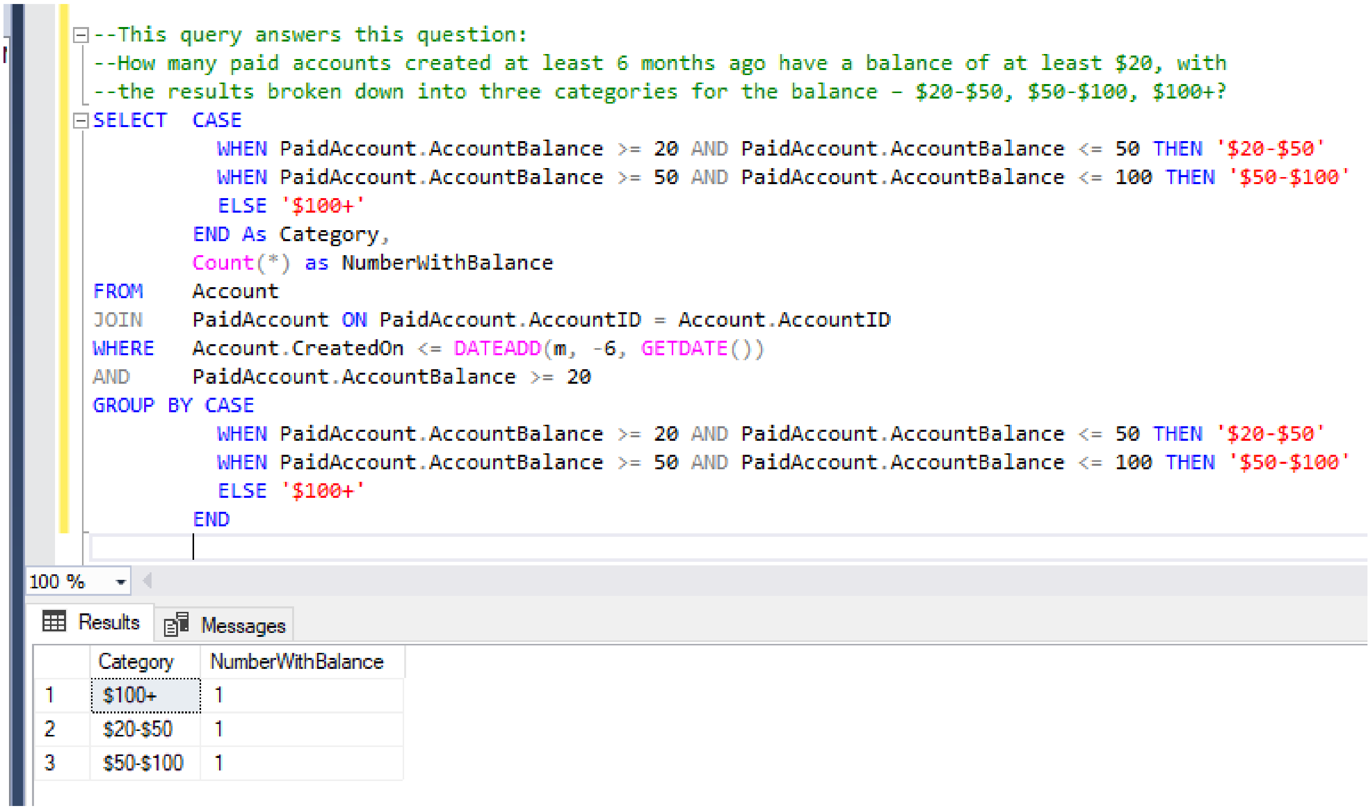
TrackMyBuys Question and Query

Paid Accounts

Here is a question useful to the core operation of TrackMyBuys: How many paid accounts created at least 6 months ago have a balance of at least $20, with the results broken down into three categories for the balance – $20-$50, $50-$100, $100+?

First, I explain why this question is useful. The answer can be used to determine how many people who have had paid accounts for a while are choosing not to pay their balance. Perhaps I want to reach out to them and encourage them to pay their balance. Perhaps I want to consider turning their account over to a credit agency. The categories help me see how much the carried balances are for, whether something small or something more egregious, so I can better make my decision as to how to handle it.

Here is a screenshot of the query I use.

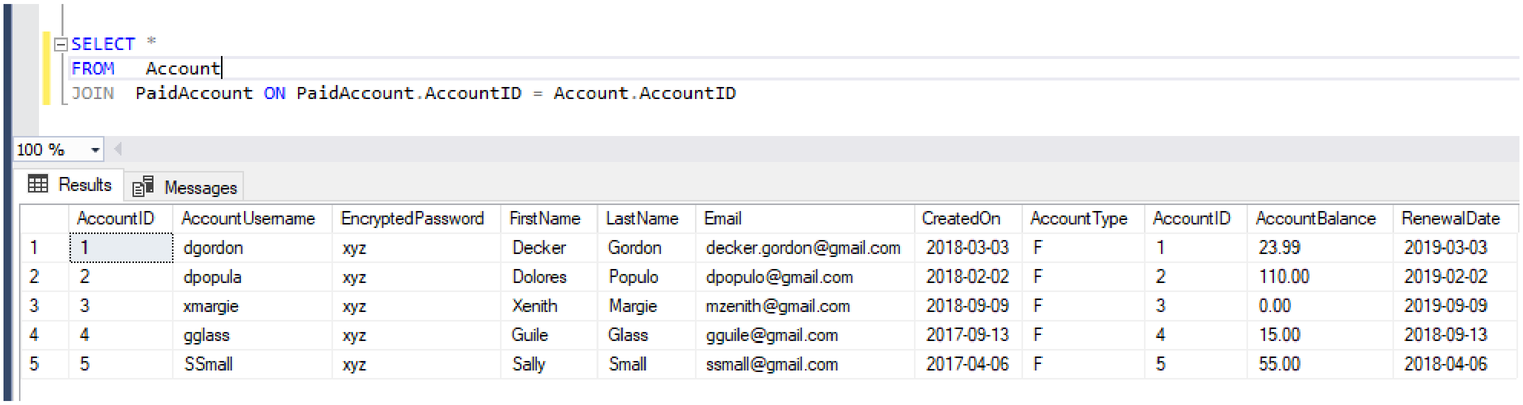


```
--This query answers this question:
--How many paid accounts created at least 6 months ago have a balance of at least $20, with
--the results broken down into three categories for the balance - $20-$50, $50-$100, $100+?
SELECT  CASE
          WHEN PaidAccount.AccountBalance >= 20 AND PaidAccount.AccountBalance <= 50 THEN '$20-$50'
          WHEN PaidAccount.AccountBalance >= 50 AND PaidAccount.AccountBalance <= 100 THEN '$50-$100'
          ELSE '$100+'
        END As Category,
        Count(*) as NumberWithBalance
FROM    Account
JOIN    PaidAccount ON PaidAccount.AccountID = Account.AccountID
WHERE   Account.CreatedOn <= DATEADD(m, -6, GETDATE())
AND     PaidAccount.AccountBalance >= 20
GROUP BY CASE
          WHEN PaidAccount.AccountBalance >= 20 AND PaidAccount.AccountBalance <= 50 THEN '$20-$50'
          WHEN PaidAccount.AccountBalance >= 50 AND PaidAccount.AccountBalance <= 100 THEN '$50-$100'
          ELSE '$100+'
        END
```

	Category	NumberWithBalance
1	$100+	1
2	$20-$50	1
3	$50-$100	1

To get the results, I join the Account to the PaidAccount table, limit the results to those with balances of at least $20, and then use the DATEADD function to additionally limit the results to accounts created at least 6 months ago. I use a case statement to categorize the balances into $20-$50, $50-$100, and $100+.

To help prove that the query is working properly, I show the full contents of the Account and PaidAccount tables with a simple query.



```
SELECT *
FROM    Account
JOIN  PaidAccount ON PaidAccount.AccountID = Account.AccountID
```

	AccountID	AccountUsername	EncryptedPassword	FirstName	LastName	Email	CreatedOn	AccountType	AccountID	AccountBalance	RenewalDate
1	1	dgordon	xyz	Decker	Gordon	decker.gordon@gmail.com	2018-03-03	F	1	23.99	2019-03-03
2	2	dpopula	xyz	Dolores	Populo	dpopulo@gmail.com	2018-02-02	F	2	110.00	2019-02-02
3	3	xmargie	xyz	Xenith	Margie	mzenith@gmail.com	2018-09-09	F	3	0.00	2019-09-09
4	4	gglass	xyz	Guile	Glass	gguile@gmail.com	2017-09-13	F	4	15.00	2018-09-13
5	5	SSmall	xyz	Sally	Small	ssmall@gmail.com	2017-04-06	F	5	55.00	2018-04-06

Upon inspection, you see that there are 5 paid accounts in my database. Only four of them were created at least 6 months ago (assuming the date in questions is September 28, 2018). Xenith Margie's account was created on September 9th, 2018, so it is excluded. Out of those four, only three of them have a balance of at least $20. Guile Glass's account only has a balance $15, so that is excluded. Last, you can see an even distribution across categories. Decker Gordon's account has a balance of $23.99, so falls into the $20-$50 category. Dolores Populo's account has a balance of $110, so falls into the $100+ category. Sally Small's balance is $55, so falls into the $50-$100 category. So as is demonstrated, the query appears to be returning the correct results based upon the question.

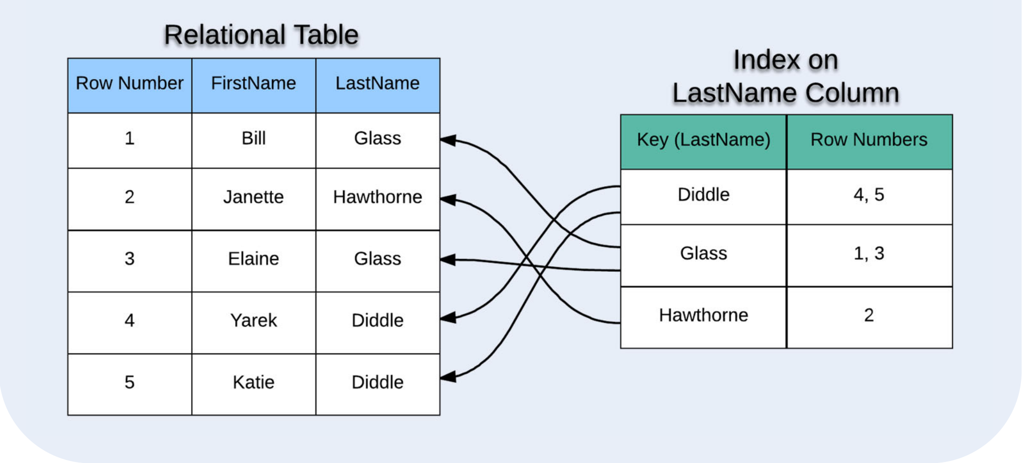
Indexing Databases

Virtually all relational databases, even well-designed ones, 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.

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



Relational Table

Row Number	FirstName	LastName
1	Bill	Glass
2	Janette	Hawthorne
3	Elaine	Glass
4	Yarek	Diddle
5	Katie	Diddle

Index on LastName Column

Key (LastName)	Row Numbers
Diddle	4, 5
Glass	1, 3
Hawthorne	2

In the example above, 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.

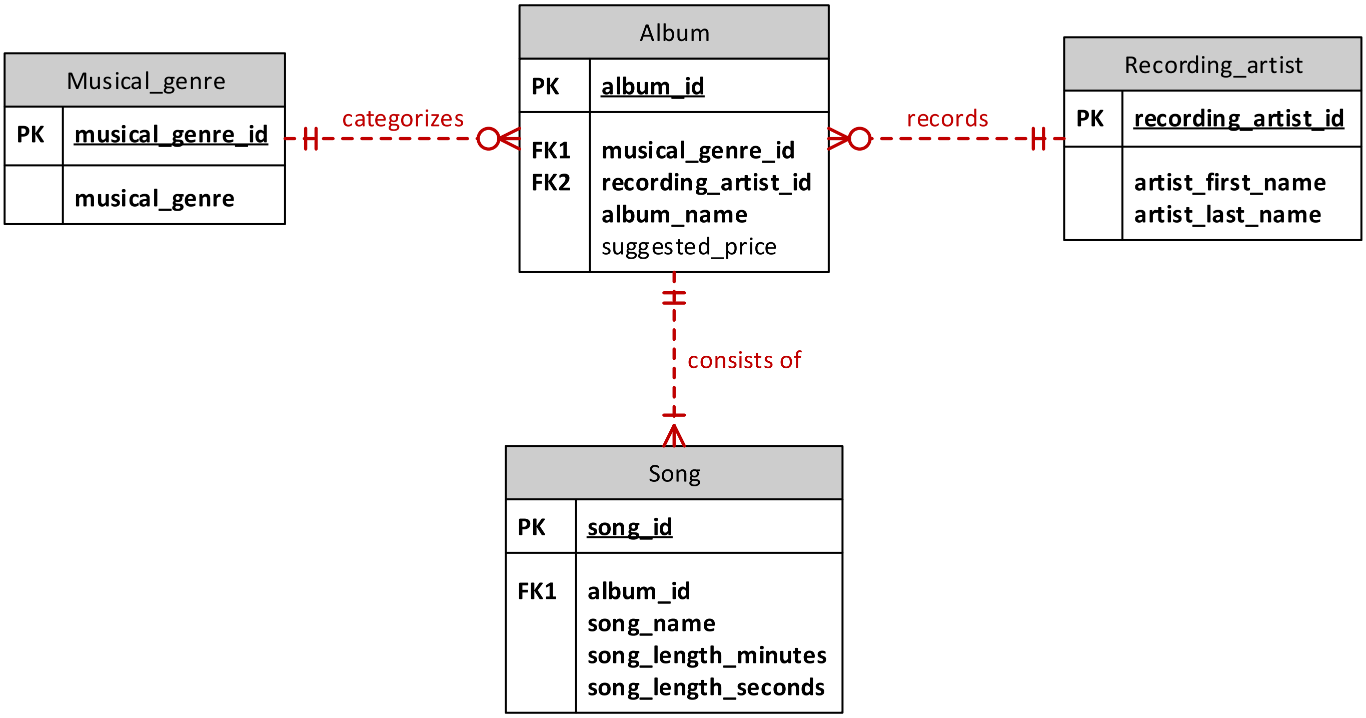
Some indexes have a secondary benefit beyond speeding up data retrieval. An index can be implemented to allow the key values to repeat, or can be implemented to disallow repeating keys. If the key values can repeat, it’s a non-unique index; otherwise, it’s a unique index. Thus, although not the primary 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.

Deciding Which Columns Deserve Indexes

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.

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 this example, 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.

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.

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 be 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 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.

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.

Query by Song Name

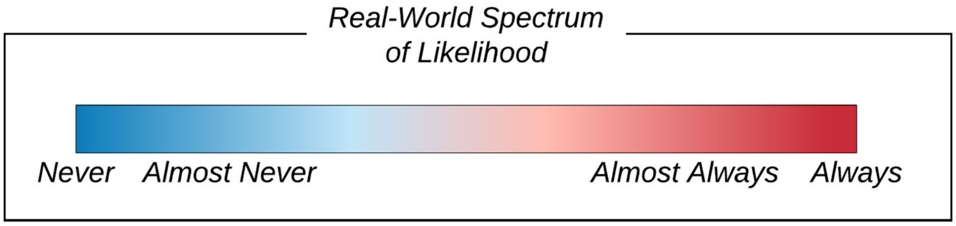
In this example, there is single table query that retrieves the minutes and seconds for the “Moods for Moderns” song in the album schema.

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

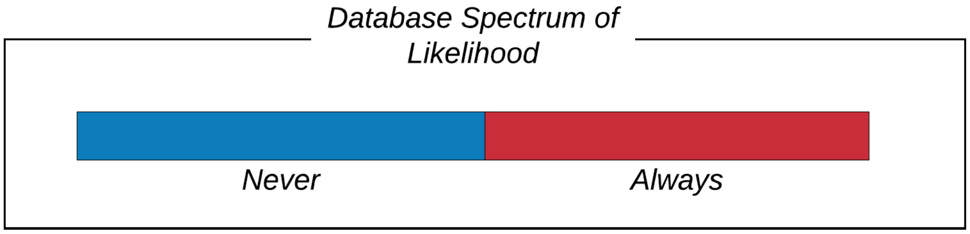
Three columns are accessed in this query – *song_length_minutes, song_length_seconds, and song_name* – but only *song_name* is in the WHERE clause. Therefore we would index the *song_name* column. The database can use *song_name* to locate the correct rows in the table, and once the rows are located, 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 the above example. 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.

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.

Query for 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.

Indexing Summary

You learned many principles as to how we determine which columns deserve indexes. To help you remember them, let’s summarize them now.

✓ Modern databases automatically index primary key columns.

✓ Foreign key columns should always be indexed.

✓ Columns in WHERE clauses or join conditions are usually indexed.

✓ Indexes on tables that remain small are usually not beneficial.

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

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

Identifying Columns Needing Indexes for your Database

You now know enough to identify columns needing indexes for your database. As noted in the prior section, there can be many indexes discovered both at design time and implementation time. So, we’ll limit the amount you need to identify to the below:

1. Identify all primary keys so you know which columns are already indexed.
2. Identify all foreign keys so you know which columns must have an index without further analysis.
3. Identify three query driven indexes that are based upon the queries you defined in this iteration. These indexes will not be placed on foreign or primary keys, but on other columns because the queries use them.

Use the standard TableName.ColumnName format when identifying columns for indexing. For example, if a table is named “Person” and a column for indexing is named “LastName”, then it would be identified as “Person.LastName”. For each index, explain why you selected it, and indicate whether it’s a unique or non-unique index, and why.

Below are the columns I identified for TrackMyBuys.

TrackMyBuys Indexing

As far as primary keys which are already indexed, here is the list.

Account.AccountId
FreeAccount.AccountId
PaidAccount.AccountId
Purchase.PurchaseId
OnlinePurchase.PurchaseId
FaceToFacePurchase.PurchaseId
Address.AddressId
State.StateId

Store.StoreId
Product.ProductId
ProductPurchaseLink.ProductPurchaseLinkId

As far as foreign keys, I know all of them need an index. Below is a table identifying each foreign key column, whether or not the index should be unique or not, and why.

Column	Unique?	Description
Purchase.StoreID	Not unique	The foreign key in Purchase referencing Store is not unique because there can be many purchases in the same store.
Purchase.AccountID	Not unique	The foreign key in Purchase referencing Account is not unique because there can be many purchases from the same account.
FaceToFacePurchase.AddressID	Not unique	The foreign key in FaceToFacePurchase referencing Address is not unique because there can be many purchases at the same store location.
Address.StateID	Not unique	The foreign key in Address referencing State is not unique because there can be many addresses in the same state.
ProductPurchaseLink.ProductID	Not unique	The foreign key in ProductPurchaseLink referencing Product is not unique because the same product can be purchased many times.
ProductPurchaseLink.PurchaseID	Not unique	The foreign key in ProductPurchaseLink referencing Purchase is not unique because a single purchase can be linked to many products.

As far as the three query driven indexes, I spotted three fairly easily by predicting what columns will commonly be queried. For example, it's reasonable that there will be many queries that limit by account balances, to see what accounts are over or under a certain. So I select PaidAccount.AccountBalance to be indexed. This would be a non-unique index because many accounts could have the same balance.

It's also reasonable that the date of purchase will be a limiting column in queries, because reports and analysts will commonly want to limit their analysis by date range, such as a particular year, quarter, month, or day. So I select Purchase.PurchaseDate to index. This would be a non-unique index because many purchases can happen on the same day.

Lastly, it's reasonable that the date the account was created will be a limiting column for some queries, such as queries that want to see how many accounts were created in a certain time period. So I select Account.CreatedOn for an index, which would be non-unique index because many accounts could be created on the same day.

Creating Indexes in SQL

Identifying the columns that deserve indexes is an important step, but alone does not improve performance on a database. Of course, the indexes need to be created in SQL for the database to take advantage of them. Thankfully, the same command can be used across Oracle, SQL Server, and Postgres. The command to create a non-unique index (which are most common) is illustrated below.

Creating a Non-Unique Index

```
CREATE INDEX IndexName
ON TableName (ColumnList);
```

The keywords CREATE INDEX are followed by a name given to an index. It's a good idea to choose a useful name that will help explain what the index is for, which might include part or all of the tablename, part or all the column name, and perhaps even a suffix such as "idx" to indicate it's an index. For example, if an index was placed on the FirstName column in a Person table, the index could be named "PersonFirstNameIdx". The reason we care about the name is that we sometimes see them come up in error messages, and a useful name will help us track down any issues without the need to investigate the schema's metadata.

Next, the ON keyword is followed by the name of the table which will contain the index. Last, a comma-separated list of columns in the table are listed. It is quite common for an index to contain only one column. It is possible, however, to have an index contain multiple columns. Such composite indexes are useful if queries commonly access all of those columns together.

Creating a unique index is very similar, as illustrated below.

Creating a Unique Index

```
CREATE UNIQUE INDEX IndexName
ON TableName (ColumnList);
```

The syntax is mostly identical to the non-unique version, the exception being the presence of the UNIQUE keyword. When a unique index is created, the database enforces that the column (or columns) in the list have unique values. If a SQL statement makes a change that would violate the uniqueness, the SQL statement is rejected.

Let's take a look again at the Car example that was in a prior section. As a reminder, the SQL that creates the Car table is listed below.

Car CREATE TABLE Statement

```
CREATE TABLE Car (
CarID   DECIMAL(12) NOT NULL PRIMARY KEY,
VIN     VARCHAR(17) NOT NULL,
Price   DECIMAL(8,2) NOT NULL,
Color   VARCHAR(64) NOT NULL,
Make    VARCHAR(64) NOT NULL,
Model   VARCHAR(64) NOT NULL,
Mileage DECIMAL(7) NOT NULL);
```

We can explore adding both a unique and a non-unique index for this table. Let's start with a non-unique index. In truth, all of these attributes – Price, Color, Make, Model, and Mileage – deserve a non-unique index because the car reseller could be interested in asking questions about any one of these fields. How many cars are in stock that are greater than a certain price? How many cars are in stock with a particular color? How many cars in stock have a specific make and model? How many cars in stock have fewer than a specific mileage? These are all reasonable questions for a car reseller.

For illustrative purposes, I will add an index for Price, illustrated below.

Car Price Index Creation

```
CREATE INDEX CarPriceIdx
ON Car(Price);
```

Notice that I created an index named "CarPriceIdx" to help us know later what it would be used for. Further notice that the index identifies the Price column in the Car table.

The VIN column deserves a unique index, for two reasons. First, every car must have a unique VIN. If any person tries to enter two different cars with the same VIN, it should not be permitted at least at the database level, and even better at the application level. Second, it is quite reasonable that a person would want to lookup a car by its VIN number, and establishing an index on the column will support an efficient lookup. We can create a unique index to accomplish both of these goals. Such an index is illustrated below.

Car VIN Index Creation

```
CREATE UNIQUE INDEX CarVINIdx
ON Car(VIN);
```

Notice the UNIQUE keyword here makes the index unique. The name "CarVINIdx" helps describe what the index is for. Otherwise this index creation is quite similar to the creation of the Price index.

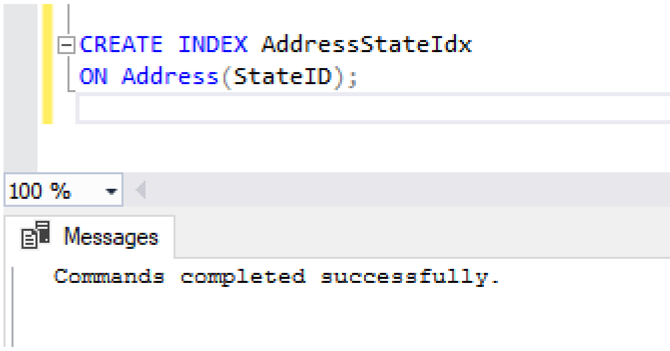
Creating Indexes in your Database

Go ahead and create the indexes for your database that you identified. You'll want to add these commands to your SQL script after your table creations. Comment the section so your instructor or facilitator knows what the SQL code is for. Provide a screenshot of creating the indexes. *Please note that you must create the SQL by hand; SQL generated from a tool will not be accepted.*

Below are a sample of the index creations for TrackMyBuys. Note that only one foreign key index and one query-driven index is demonstrated for TrackMyBuys, but you should include all that are requested in the instructions.

TrackMyBuys Index Creations

Here is a screenshot demonstrating creation of a foreign key index for TrackMyBuys, for the foreign key between Address and State.

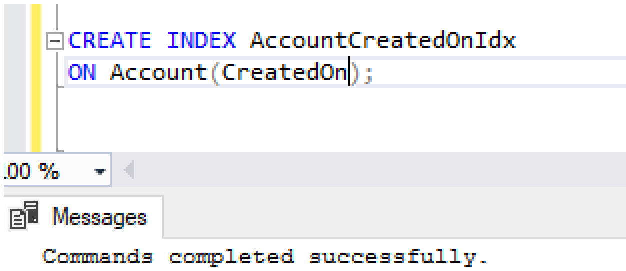


```
CREATE INDEX AddressStateIdx
ON Address(StateID);
```

```
Commands completed successfully.
```

I named the index “AddressStateIdx” to help identify what it’s for, and placed the non-unique index on the StateID table in Address.

Here is a screenshot demonstrating creation of a query-driven index, an index on the CreatedOn attribute in Account.



```
CREATE INDEX AccountCreatedOnIdx
ON Account(CreatedOn);
```

```
Commands completed successfully.
```

I named the index “AccountCreatedOnIdx” to help identify what it’s for, and put it on the CreatedOn column in Account.

Summary and Reflection

Take a moment to reflect on all you will have accomplished in your project when you complete this iteration. You formally designed your own database from scratch using universally understood language and diagrams. You created your database structure in SQL, the universal language for relational databases. You will have written transactions against it to accomplish useful tasks for your organization or application, and will have indexed your database to help it perform well. Excellent progress!

Update your project summary to reflect your new work. Write down your reflections on your database, as well as any questions or concerns you may have.

Here is an updated summary as well as some reflections I have about TrackMyBuys.

TrackMyBuys Reflection

My database is for a mobile app named TrackMyBuys which records purchases made across all stores, making it the one stop for any purchase history. Typically, when a person purchases something, they can only see their purchase history with that same vendor, and TrackMyBuys seeks to provide a single interface for all purchases. The database must support a person entering, searching, and even analyzing their purchases across all stores.

The structural database rules and conceptual ERD for my database design contain the important entities of Store, Product, Purchase, and Account, as well as relationships between them. The design contains a hierarchy of Purchase/FaceToFacePurchase and Purchase/OnlinePurchase to reflect the two primary ways people purchase products. The design also contains a hierarchy of Account/PaidAccount and Account/FreeAccount to reflect the fact that people can signup for a free account or a paid account for TrackMyBuys. The DBMS physical ERD contains the same entities and relationships, uses the best practice of synthetic keys, and contains the important attributes needed by the database to support the application.

The SQL script that contains all table creations that follows the specification from the DBMS physical ERD exactly. Important indexes have been created to help speed up access to my database and are also available in an index script. Stored procedures have been created and executed transactionally to populate some of my database with data. Some questions useful to TrackMyBuys have been identified, and implemented with SQL queries.

As I reflect on the database, it is amazing to see a real database in action that can be hooked up to the mobile application TrackMyBuys. I can envision many of the screens already, and should I choose to make this a real Android or IPhone application, a good portion of the database is already implemented.

Items to Submit

In summary, for this iteration, you revise your design, create stored procedures that are executed to transactionally add data to your database, populate the rest of your tables with data, identify questions useful to the organization or application, answer them with queries, and add indexes to make your database perform better. Make sure to use the template provided with this iteration to ensure you are submitting all necessary items.

Evaluation

Your iteration will be reviewed by your facilitator or instructor with the criteria outlined in the table below. Note that the grading process:

- involves the grader assigning an appropriate letter grade to each criterion.
- uses the following letter-to-number grade mapping – A+=100,A=96,A-=92,B+=88,B=85,B-=82,C+=88,C=85,C-=82,D=67,F=0.
- provides an overall grade for the submission based upon the grade and weight assigned to each criterion.
- allows the grader to apply additional deductions or adjustments as appropriate for the submission.
- applies equally to every student in the course.

Aspect	What is Measured	A+ Excellent	B Good	C Fair/Satisfactory	D Insufficient	F Failure
Stored Procedures (20%)	This is a measure of the quality of the construction and correctness of the syntax of the stored procedures, and the number of stored procedures created. Excellent stored procedures implement useful and complete transactions, are reusable through use of parameters, are robust with the inclusion of parameter validation, and compile and are executable. Excellent solutions define at least two stored procedures.	Excellent construction Entirely correct syntax At least two stored procedures	Good construction Mostly correct syntax At least two stored procedures	Satisfactory construction Somewhat correct syntax At least one stored procedure	Insufficient construction Mostly incorrect syntax At least one stored procedure	No stored procedures defined *or* Unacceptable construction Entirely incorrect syntax
Questions (10%)	This is a measure of how essential the questions are to your database, as well as the number of questions provided. Excellent solutions have at least three questions defined that represent very important use of the data, where the answers to the questions are critical to those using the database.	Entirely essential At least three questions	Mostly essential At least three questions	Useful but secondary At least two questions	Mostly unnecessary At least one question	No questions given *or* Entirely unnecessary

Queries (20%)	This is a measure of how correctly the queries answer the questions and conform to the project requirements, the correctness of the syntax, and the number of queries defined. Excellent solutions exhibit all of the following properties. At least three queries produce results that completely answer the questions. Each query meets the requirements given in the project instructions. The columns selected give all needed information for those asking the question. No unneeded columns and rows are present. The syntax of the queries is entirely correct and could be executed in a modern relational database without modification.	Entirely correct answers and conformity Entirely correct syntax At least three queries	Mostly correct answers and conformity Mostly correct syntax At least three queries	Somewhat correct answers and conformity Somewhat correct syntax At least two queries	Mostly incorrect answers and conformity Mostly incorrect syntax At least one query	No queries defined *or* Entirely incorrect answers and conformity Entirely incorrect syntax
Index Identifications (20%)	This is a measure of how beneficial the indexes are to your database, how accurate the uniqueness choices are, and the coverage of the indexes. Excellent solutions exhibit all of the following properties. The query driven fields selected would benefit significantly with indexes. The unique/not unique choices are entirely accurate for all indexes. All primary and foreign key indexes have been identified, and at least three query driven indexes have been identified.	Entirely beneficial Entirely accurate Full coverage	Mostly beneficial Mostly accurate Good coverage	Somewhat beneficial Somewhat accurate Partial coverage	Mostly unhelpful Mostly inaccurate Insufficient coverage	No indexes identified *or* Entirely unhelpful Entirely inaccurate Unacceptable coverage
SQL Script (10%)	This is measure of how accurately the SQL script implements the designed database structure, and the correctness of the SQL syntax. Excellent scripts exhibit all of the following properties. The SQL script includes the table names, attribute names, attribute datatypes, and constraints exactly as defined in the DBMS physical ERD, as well as the identified sequences and indexes. Only designed elements are present in the script. The syntax of the SQL script is entirely correct and could be executed in a modern relational database without modification.	Entirely accurate Entirely correct	Mostly accurate Mostly correct	Somewhat accurate Somewhat correct	Mostly inaccurate Mostly incorrect	The SQL script is missing *or* The SQL script is generated by a tool without significant modification *or* Entirely inaccurate Entirely incorrect

Overall Presentation (10%)	This is a measure of how well your design and implementation choices are supported with explanations, as well as the quality of your documentation organization and presentation.	Excellent support Well organized and presented	Good support Organized and presentable	Partial support Somewhat organized and presented	Mostly unsupported Mostly disorganized presentation	No explanations Entirely disorganized presentation
Prior Work Soundness (10%)	This measures how well any issues from prior iterations have been improved in order to provide a frame of reference for this iteration.	Completely improved *or* No improvement necessary	Mostly improved	Somewhat improved	Mostly not improved	No improvements
Preliminary Grade:		**Entities Deduction:** At least 10 required at the conceptual level At least two subtypes required 3 point deduction for each missing		**Lateness Deduction:** 5 points per day 4 days maximum Contact your facilitator for any exceptions		**Iteration Grade:**

Use the **Ask the Teaching Team Forum** if you have any questions regarding how to approach this iteration. Make sure to include your name in the filename and submit it in the *Assignments* section of the course.