SECTION 1: DATABASE DESIGN AND ENTITY RELATIONSHIP DIAGRAMS (ERDS) 1

CHAPTER 1: Why is Database Design Important? 1
I. Data, Information and Knowledge 1
II. Why do we need good design? 1

CHAPTER 2: Background to ERD 4
I. What’s an ERD? 4
II. Business Rules for Glenside Bank 4
III. Components of an ERD 4
IV. ERD Styles 7
V. Keys 8
VI. Integrity Rules 11
VII. Relationship Rules 13

CHAPTER 3: Example of Creating an ERD 15
I. 1:1 Relationships 15
II. 1:M Relationships 16
III. M:N Relationship 20
IV. Final ERD 22
V. Placements of attributes in tables 23

CHAPTER 4: Invoice Example 24
I. Entities and Attributes: 24
II. Relationships: 25
III. ERD 26
IV. Variations on a theme 27

CHAPTER 5: Data Dictionary 28
I. What is a data dictionary? 28

SECTION 2: ACCESS TUTORIAL 29

CHAPTER 6: Introduction 29
I. Access 2007 Interface 29

CHAPTER 7: Tables 30
I. Introduction to Tables 30
II. Table Relationships 38

CHAPTER 8: Queries 41
I. Introduction 41
II. Select Queries 41
III. Parameter Queries 44
IV. Crosstab Queries 46
V. Action Queries 48
VI. Aggregate Functions with Group By clause 51

CHAPTER 9: Forms 57
I. Introduction to Forms 57
II. Create Form by Using Wizard 57
III. Create Form in Design View 60
IV. Adding Records Using a Form 62
V. Editing Forms 62
VI. Form Controls 64
VII. Sub Forms for 1:M Relationships 71
VIII. Sub Forms for M:M Relationships 76
IX. Multiple-Page Forms Using Tabs 79
# TABLE OF CONTENTS

## CHAPTER 10: REPORTS

I. Introduction to Reports .................................................. 81
II. Types of Reports .......................................................... 81
III. Create Report Using the Wizard ....................................... 81
IV. Create Report in Design View ......................................... 83
V. Printing Reports ............................................................ 84

## CHAPTER 11: SWITCHBOARD

I. Introduction to Switchboards .......................................... 85
II. Create a Switchboard ..................................................... 86
III. Create a Switchboard Within a Switchboard ....................... 88
IV. Display Switchboard When Database Opens ....................... 90

## CHAPTER 12: KEYBOARD SHORTCUTS CUTS

<table>
<thead>
<tr>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>92</td>
</tr>
</tbody>
</table>

## CHAPTER 13: PIVOT TABLES

I. Introduction to Pivot Tables ........................................... 93
II. Introduction to ODBC .................................................... 94
III. Connecting Oracle to Access via an ODBC Connection ........ 94
IV. Background to Connecting Access to Other Data Formats: .... 96
V. Connect Excel to Access .................................................. 97
VI. Connecting Text Files to Access ....................................... 98
VII. Creating Queries across Access and Oracle ....................... 100
VIII. Crosstab Queries: ....................................................... 101
IX. Crosstab queries in Excel 2007 ......................................... 105

## SECTION 3: TOAD

<table>
<thead>
<tr>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>107</td>
</tr>
</tbody>
</table>

## CHAPTER 14: INTRODUCTION AND LOADING TOAD

I. What is TOAD ............................................................... 107
II. Loading and Configuring TOAD ........................................ 107
III. Introduction to TOAD Interface ....................................... 109

## CHAPTER 15: TABLES IN TOAD

I. Creating a table ............................................................ 110
II. Adding constraints to a table ............................................ 111

## CHAPTER 16: SCRIPTS IN TOAD

I. Documenting Table Creation in Toad .................................. 114
II. Loading a Script into TOAD ............................................. 116

## CHAPTER 17: OTHER FUNCTIONS IN TOAD

I. Using TOAD to write SQL statements ................................ 117
II. Using TOAD to Diagram and Modify a database .................. 119
III. Exit Toad ..................................................................... 121

## SECTION 4: SQL

<table>
<thead>
<tr>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>122</td>
</tr>
</tbody>
</table>

## CHAPTER 18: WELCOME TO SQL

I. Introduction to SQL ........................................................ 122
II. Introduction to SQL*Plus ................................................. 122
III. DDL vs. DML ............................................................... 125

## CHAPTER 19: DDL – DATA DEFINITION LANGUAGE

I. Create Table ................................................................. 127
II. Drop Table ...................................................................... 128
III. Alter Table .................................................................... 128
IV. Describe ....................................................................... 129

## CHAPTER 20: CONSTRAINTS (DDL CONTINUED)

<table>
<thead>
<tr>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>130</td>
</tr>
</tbody>
</table>
# TABLE OF CONTENTS

I.  Introduction to Integrity Constraints ........................................... 130
II. Primary Key Constraints ............................................................... 130
III. Foreign Key Constraint: ............................................................... 131
IV. Null Value Constraint ................................................................. 132
V. Default Value Constraint .............................................................. 133
VI. Unique Constraint .................................................................... 134
VII. Check Constraint ...................................................................... 134

CHAPTER 21: DML - COMMIT, ROLLBACK, AND FORMATTING .................. 136
I. Commit and Rollback ................................................................. 136
II. Formatting Results ................................................................. 136

CHAPTER 22: DML - INSERTING DATA INTO TABLES ......................... 140
I. Insert Statement ........................................................................... 140
II. Querying Rows with Select Statement ........................................ 140
III. TO_DATE: ................................................................................ 141

CHAPTER 23: DML - SELECT ................................................................ 143
I. Select Query Background ............................................................ 143
II. Simple SELECT .......................................................................... 143
III. The Asterisk (*) wildcard .......................................................... 144
IV. Filtering in Select Statement (Projection) to Specific Columns: ...... 144
V. Order of Attributes in Select ....................................................... 145
VI. Distinct ..................................................................................... 145
VII. Restrict relational operator – Where Clause ............................... 146
VIII. Searching for rows .................................................................. 147
IX. Wildcard characters: ................................................................. 148
X. Comparison Operators ............................................................... 149
XI. Logical Operators ..................................................................... 152
XII. NOT operator .......................................................................... 153
XIII. IS NULL and IS NOT NULL Operator ........................................ 153
XIV. Precedence ............................................................................ 154
XV. Sorting the output – Order by .................................................. 155
XVI. Alias Operator ........................................................................ 156
XVII. Math in SQL .......................................................................... 157

CHAPTER 24: SET OPERATORS ............................................................ 159
I. Union ..................................................................................... 159
II. Intersect ................................................................................... 160

CHAPTER 25: DML AND JOINS ......................................................... 161
I. Inner Joins .............................................................................. 161
II. Outer Joins ............................................................................ 165
III. ANSI standard format for Joins ................................................ 166

CHAPTER 26: DML - UPDATE AND DELETE RECORDS ................. 169
I. Update Statement ........................................................................ 169
II. DELETE Statement ................................................................. 170

CHAPTER 27: DML - DATE FUNCTIONS ............................................. 173
I. Date Background Information: .................................................. 173
II. Sysdate .................................................................................... 174

CHAPTER 28: FUNCTIONS IN SQL .................................................. 176
I. Basics of functions ................................................................... 176
II. Common Functions .................................................................. 176
III. Date Functions ......................................................................... 177
IV. String Functions ...................................................................... 178

CHAPTER 29: AGGREGATION ................................................................. 181
# TABLE OF CONTENTS

I. Introduction to Aggregation .......................................................... 181
II. COUNT .................................................................................. 181
III. SUM .................................................................................... 181
IV. AVERAGE ............................................................................ 182
V. MIN and MAX ........................................................................ 183
VI. GROUP BY ............................................................................ 183
VII. HAVING ................................................................................ 188

CHAPTER 30: NESTED QUERY/SUBQUERY ........................................... 193
I. Introduction to Nested or Sub Queries ........................................ 193
II. Examples with Nested Queries ................................................. 193
ACCESS AND PIVOT TABLE PRACTICE EXERCISES

PRACTICE 1 – CREATE TABLES IN ACCESS: ________________________________ 36
PRACTICE 2 - CREATE AN ERD: ________________________________ 40
PRACTICE 3 - CREATE QUERIES: ________________________________ 55
PRACTICE 4 - CREATE A DATA ENTRY FORM USING FORM WIZARD: ________________ 60
PRACTICE 5 - CREATE A DATA ENTRY FORM IN DESIGN VIEW: ________________ 61
PRACTICE 6 - ADD RECORDS TO A FORM: ________________ 62
PRACTICE 7 - CREATE A COMBO BOX: ________________ 67
PRACTICE 8 - CREATE BUTTONS ON A FORM: ________________ 70
PRACTICE 9 – CREATE A FORM WITH A SUBFORM: ________________ 75
PRACTICE 10 - CREATE A M:N FORM ________________ 78
PRACTICE 11 – CREATE A FORM WITH TAB CONTROL: ________________ 80
PRACTICE 12 - CREATE A REPORT: ________________ 83
PRACTICE 13 – CREATE A SWITCHBOARD: ________________ 91
PRACTICE 14 - CREATE A SELECT QUERY WITH LINKED AND IMPORTED DATA SOURCES: ________________ 101
PRACTICE 15: CREATE CROSSTAB QUERIES IN ACCESS ________________ 104
PRACTICE 16 – CREATE A PIVOT TABLE AND CHART IN EXCEL: ________________ 106
Section 1: Database Design and Entity Relationship Diagrams (ERDs)

Chapter 1: Why is database design important?

I. Data, Information and Knowledge

Businesses collect and store data on a variety of things such as a price, a supplier, a product, or a sale. But the data doesn’t really mean anything at this point. When the organization summarizes the information, applies context it, and presents the data in a fashion which is useful to the organization, the data has been transformed into information. Information is useful and typically provides answers to "who", "what", "where", and "when" questions.

While information is vital for organizations, firms that convert information into knowledge excel in their work. Knowledge answers "how" questions.

Finally, if you move up to the top of the knowledge pyramid, you achieve wisdom or true understanding.¹

As business people, we try and move up the pyramid to move towards wisdom. Sounds like a Jedi in Star Wars, huh? So how do organizations do this? To start, organizations need good databases that facilitate the collection, storage, and dissemination of information within the organization.

II. Why do we need good design?

An organization’s success is often tied to the efficient and effective flow of information. If the data of an organization is not stored in a properly designed database, the organization cannot make good business decisions based on data. If the data is stored in a flawed manner, the user can retrieve inconsistent results and the data is said to lack integrity. The user cannot be confident with the information retrieved. If the user doesn't trust the results received, he will not be able to make a decision with certainty.

Consider the following. What if you wanted to take the Temple shuttle bus from Ambler Campus to Main Campus. You look at the schedule on Temple’s portal and it says the bus will arrive at Ambler at 8:15 AM. You arrive at the stop at 8:10 AM and wait for the bus. It finally arrives at 10:15 AM, and now you’ve missed your 9:40 AM class. It turns out that the schedule isn’t updated regularly so you can't rely on it. Sometimes the bus arrives at 8:05, sometimes at 8:17, and sometimes it is cancelled, but these changes are not reflected on the website. What will you do next time? Will you look at the online schedule or will you just go to the stop super early in the hopes that you’ll catch the bus whenever it arrives?

¹ http://www.systems-thinking.org/dikw/dikw.htm
Database Design and Entity Relationship Diagrams

The ERD is a model (a representation) of how the database is structured. The ERD is designed to allow the database designer to optimize the manner in which the data is stored to facilitate the use of the data within the organization.

A. Data Anomalies

We’ve all had situations where we’ve talked to a company and found out that the information stored isn’t correct. Maybe you’ve called a store to confirm an item you want is in stock. You are told that there are 10 items in inventory. When you get there, it turns out they are out of stock.

For example, I received a book in the mail from a publisher. The book had been sent to my old house, and I moved from over a year ago. While this seems reasonable, the publisher knew that I had moved because they’ve sent other books to my new address. Why would some books be sent to the correct address and some routed to the old location? The most common reason for these kinds of situations is that organizations have many databases and many tables within each database, and these tables hold redundant data.

What if the publisher had a faculty table that listed each faculty member to whom they send books? The publisher also has an order table that shows which books that faculty member ordered and this table includes the address to which the order was sent.

This is a perfect environment for breeding data anomalies. An anomaly is an abnormality in our data and we want to avoid abnormalities in database design. There are three types of data anomalies that plague us.

1. Insertion (add) anomaly

In an insertion anomaly a row cannot be inserted (added) to one table unless it is added somewhere else. This means there is a forced dependency between two tables that should not exist. In our publisher example it might be that a faculty member cannot be added to the database until he/she places an order. This is not an appropriate dependency. The publisher should be able to add a new faculty member, and then add orders as they are placed.

2. Deletion anomaly

A deletion anomaly is just the reverse of an insertion anomaly. In other words a table is deleted and the data from another table cannot be retrieved. For instance, if we store information about the customer such as the address in the order table and then we delete an order, the information about the customer’s account including the address will also be lost. We don’t want to lose the customer’s address. We just want to remove an order.

3. Modification (update) anomaly

In a modification anomaly, values of an attribute must be duplicated multiple times in a table. If the publisher stored my address each time an order was placed (rather than storing my address once), when I moved the publisher would need to update each row where my old address appears. An even worse scenario involves organizations which only update the address on some of the rows while others are left unchanged. This would lead to data retrievals that are inconsistent with some rows showing a new address and others showing an older location. As we’ve learned, inconsistent retrievals lead to loss of integrity in the database.

If the database had been properly designed, my address would have been stored once for each location where I receive books (work or home). When my home address changed, the organization would only need to update my home location once, and all of

the rows related to this home address would be updated appropriately. The key is to capture the value of an attribute once, store it once, and use that one data value consistently.

Typically, companies would prefer to collect the value of an attribute for a row consistently wrong, as opposed to sometimes accurately and sometimes inaccurately. This may seem odd but really makes sense if you think about it. If the information is sometimes correct you falsely assume it is always right and rely on it until you are faced with a situation where the data is clearly inaccurate. At that point, you realize you may have made decisions on poor quality information. This situation is more difficult to correct as well since you need to consider each individual row and assess whether that row’s data is correct or not.

Conversely, if the information is consistently inaccurate, it is more likely you’ll note the error earlier and initiate the process to correct it faster. Also, with a consistent error, you may be able to globally update the information, and that is much simpler than considering each row on a case by case basis.

B. We always want to avoid data anomalies and maintain data in one location which is accurate and timely. When users of the data can rely and trust the results to be updated and accurate, they feel more confident making a business decision since the veracity of the data is not questioned.
Chapter 2: Background to ERD

I. What’s an ERD?

An ERD is an Entity Relationship Diagram. Just as the name implies, an ERD shows entities and their relationships to one another. An ERD shows the data elements stored in a database and in what manner the data is stored.

II. Business Rules for Glenside Bank

One of the key aspects of an ERD is that it models the data needs of the organization as it relates to how the business operates. The policies regarding how the business works are called the “business rules” of the organization.

In this section, we will consider the case of Glenside Bank. Glenside Bank is a fictitious, local bank with just a few branches. Glenside would like to improve the way the bank stores information about customers and their accounts. The bank functions like this.

1. The bank has customers and it collects basic demographic information for each customer such as name, address, social security number, gender, date of birth, phone number.

2. The bank offers a number of accounts such as a checking account, savings account, money market account, CD, etc.

3. The bank has a number of branches. The bank needs to track where each account was opened.

4. A customer can have many accounts (you could have a checking and a savings account). For each account, the bank needs to track the type of the account (i.e. checking, savings, money market) and the date the account was opened.

5. An account can have multiple customers associated with it (a husband and wife could have a joint checking account). The bank needs to track the customers associated with each account and the date that customer was added to the account. If the customer was removed from the account, the bank needs to track the date this occurred.

6. Every time the customer interacts with one of the accounts it is considered a transaction. The bank needs to track a number of aspects of the transaction including the type (i.e. withdrawal, transfer, deposit), the amount, the date and time, the customer, and the location (ATM, check, branch, electronic fund transfer)

These business rules will be the basis for a new database which we’ll call the Account database.

III. Components of an ERD

An ERD has two main components: Entities and Relationships

A. Entity:

An entity is something about which the organization wants to collect data. We typically think of an entity as a noun (person, place, or thing). For instance, in the Glenside bank scenario, you would need to keep information about a number of people, places, and things like the customer, branch, bank account, respectively. Most people find it useful to think of an entity as table which holds rows. Each row is one row in the table or one occurrence of the entity.

1. An entity (table) is a two-dimensional structure with rows and columns

2. Column:

   a. The column in a table represents a piece of information called an attribute. Sometimes an attribute will be called a field, data element, or parameter. We’ll learn more about attributes later including what the columns marked PK and Null mean.
Database Design and Entity Relationship Diagrams

For now, let's focus on the first column in the table below contains some of the attributes in the customer table.

<table>
<thead>
<tr>
<th>Column Name</th>
<th>ID</th>
<th>Fk</th>
<th>Null?</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSTOMER_ID</td>
<td>1</td>
<td>N</td>
<td></td>
<td>NUMBER (5)</td>
</tr>
<tr>
<td>CUSTOMER_LNAME</td>
<td>2</td>
<td>N</td>
<td></td>
<td>VARCHAR2 (30 Byte)</td>
</tr>
<tr>
<td>CUSTOMER_FNAME</td>
<td>3</td>
<td>N</td>
<td></td>
<td>VARCHAR2 (30 Byte)</td>
</tr>
<tr>
<td>ADDR1</td>
<td>4</td>
<td>N</td>
<td></td>
<td>VARCHAR2 (30 Byte)</td>
</tr>
<tr>
<td>CITY</td>
<td>5</td>
<td>N</td>
<td></td>
<td>VARCHAR2 (25 Byte)</td>
</tr>
<tr>
<td>STATE</td>
<td>6</td>
<td>N</td>
<td></td>
<td>CHAR (2 Byte)</td>
</tr>
<tr>
<td>ZIP</td>
<td>7</td>
<td>N</td>
<td></td>
<td>CHAR (5 Byte)</td>
</tr>
<tr>
<td>GENDER</td>
<td>8</td>
<td>Y</td>
<td></td>
<td>CHAR (1 Byte)</td>
</tr>
<tr>
<td>DOB</td>
<td>9</td>
<td>Y</td>
<td></td>
<td>DATE</td>
</tr>
<tr>
<td>HOMEPHONE</td>
<td>10</td>
<td>Y</td>
<td></td>
<td>CHAR (10 Byte)</td>
</tr>
</tbody>
</table>

b. Column values (the information stored in an attribute) all must have the same data format (data type). We'll learn more about data types in the Access and SQL tutorials later in this book. As you can see above, all of the customer_id values must be of a number data type while all of the customer_lname values must be a varchar2.

c. An attribute has a range of values called the attribute domain. This indicates the range of acceptable values for that attribute. For instance, we could say that the domain for zip is 00001 – 99999 if there are no zip codes issued after 99999 and no negative zip codes are issued. For date of birth (DOB), you may decide to make the domain 1/1/1900 – today’s date. This would indicate that there are no customers born on or before 12/31/1899 and the youngest bank customer was born today (such as when a savings account is opened as a present for a newborn).

3. Row:
   a. An entity holds rows. Each row is one instance of an entity. As you can see below, a row holds the values for each attribute for that instance. Our first customer, Frank Sinatra, has a customer_id of 10000. He lives at 144 Woodstream Blvd in Hoboken, is male, and was born on 1/11/1943, etc.
   b. Each piece of information in an attribute in the row holds the specific value for that field for that row.

4. Row/Column Intersection:
   a. The intersection of a row and a column represents a single value for that attribute for that row.

5. Other aspects of entities:
   a. Entity names:
   b. An entity must have a unique name in a database. Let's say that Glenside likes to track the different kinds of customers including personal customers and business customers. We could not have two tables in the Account database called customer. However, there could be a customer table in different database.
Database Design and Entity Relationship Diagrams

c. An entity name should be singular. Accordingly, we would have a table called customer, not customers.
d. An entity is depicted with a box in an ERD
e. Tables must have an attribute to uniquely identify each row in the table. This attribute is called the primary key. We’ll discuss this soon.
f. The order of the rows and columns is irrelevant to the Database Management System (DBMS).

B. Relationship:

The relationship describes how entities are related to one another. A relationship is typically described as a verb. In the Glenside Bank example, the customer owns many bank accounts. The account is owned by many customers.

In an ERD, a relationship is depicted by a line that joins two (or more) entities.

There are three types of relationships: one to one, one to many, and many to many.

1. Example of a one to one relationship (abbreviated as 1:1) in Glenside Bank:
   - An account is identified by one account number.
   - An account number identified one account

2. Example of a one to many relationship (abbreviated as 1:M) in Glenside Bank:
   - An account is classified into one account type such as checking, savings, money market
   - An account type has many accounts (i.e. there are many checking accounts at Glenside Bank)

3. Example of a many to many relationship (abbreviated as M:N – yes the second letter is N) in Glenside Bank:
   - An account can be owned by many customers
   - A customer can own many accounts

C. Attributes:

Attributes are really part of an entity as we’ve already discovered. Let’s think a bit more about attributes and an important concept – Atomicity.

Let’s consider the customer at Glenside Bank. If we want to collect information about the customer, we’ll need to know the customer’s name. You might decide to create an attribute in the customer table called name and have the first and last name of the customer added to this one attribute. We’ll use me as an example. Let’s say I decide to update my account and change my last name from Marselis to Lundeen on my account from now on. It’s much more difficult to take my name which was stored in one attribute apart to update just the second name. They’d need to retrieve my name, Cindy Joy Marselis, and try and figure out which part of it is my first name, middle name, and last name. This isn’t too hard if my name is typically spaced and capitalized. But what happens if I have a suffix to my name, like the third (III) or my name was hyphenated. It would get increasingly difficult to figure out which part is which. To alleviate this problem, we store attributes in their most atomic (smallest) parts. Accordingly, we’ll keep my first name in one attribute, last name in another attribute, etc.
Database Design and Entity Relationship Diagrams

For instance, we don’t store an address in just one attribute. We’ll keep the first line of the address in one attribute, the second line (for a suite, an apartment number, etc.) in another attribute, the city in a separate attribute, and state in yet another attribute. Some folks like to keep the zip code as 2 attributes, one to store the 5 digit zip and another to store that 4 digit extension.

E. F. Codd, the father of relational databases, said the following, “values in the domains on which each relation is defined are required to be atomic with respect to the DBMS.” Codd defines an atomic value as one that "cannot be decomposed into smaller pieces by the DBMS (excluding certain special functions)."

IV. ERD Styles

There are a number of styles you can use to model a database, the most common of which are the Chen and the Crow’s Foot approaches. In addition to the Chen and the Crow’s Foot, we will also explore the infinity model (for lack of a better name) as this is the approach used by Access, our main modeling tool for this class. In general, it really doesn’t matter which style you use, but your approach should be consistent within one ERD.

A. Chen Model

1. For a 1:1 relationship, a number 1 is placed on the relationship line close to the both entity boxes
2. For a 1:M relationship, a number 1 is placed on the relationship line close to the entity which is the one side of the relationship and a letter M is placed next to the entity which is the many side of the relationship
3. For a M:N relationship, a letter M is placed next to one entity (doesn’t matter which one) and a letter N is placed next to the other entity

B. Crow’s Foot Model

1. For a 1:1 relationship, a number 1 is placed on the relationship line close to the both entity boxes
2. For a 1:M relationship, a number 1 is placed on the relationship line close to the entity which is the one side of the relationship and a little symbol that looks like a crow’s foot is placed next to the entity which is the many side of the relationship
3. For a M:N relationship, a crow’s foot is placed next to both entities

C. Infinity Model

1. For a 1:1 relationship, a number 1 is placed on the relationship line close to the both entity boxes
2. For a 1:M relationship, a number 1 is placed on the relationship line close to the entity which is the one side of the relationship and an infinity symbol (∞) is placed next to the entity which is the many side of the relationship
3. For a M:N relationship, an infinity symbol is placed next to both entities
Database Design and Entity Relationship Diagrams

The table below illustrates how relationships are represented in the three ERD Styles.

<table>
<thead>
<tr>
<th>Type</th>
<th>Example of Relationship</th>
<th>Chen</th>
<th>Crow’s Foot</th>
<th>Infinity</th>
</tr>
</thead>
</table>
| 1 to 1                | • An account is identified by one account number.  
                        • An account number identified one account | ![Chen Diagram](image) | ![Crow’s Foot Diagram](image) | ![Infinity Diagram](image) |
| 1 to Many (1:M)       | • An account is classified into one account type such as checking, savings, money market  
                        • An account type has many accounts | ![Chen Diagram](image) | ![Crow’s Foot Diagram](image) | ![Infinity Diagram](image) |
| Many to Many (M:N)    | • An account can be owned by many customers  
                        • A customer can own many accounts | ![Chen Diagram](image) | ![Crow’s Foot Diagram](image) | ![Infinity Diagram](image) |

V. Keys

A. Primary key:

A primary key is an attribute that uniquely identifies each row in a table. A primary key must satisfy three requirements:

- Unique. Each row’s primary key must be different. For instance, if you wanted to create a primary key for the student table, you might select TUID since every student has a unique identifier for Temple. It wouldn’t be wise to select the student’s first name as there could be more than one person with that name. Similarly, last name wouldn’t be a good idea.

- Not null. The value of the primary key cannot be blank. Why? Because its purpose is to identify each row. If you wanted to look a student up in Owlnet, how could you do that if the primary key (the attribute used to identify each student) was blank? So a primary key must have a value stored in it.

- Indexed. Indexing improves the speed required to find the desired row. As described in Wikipedia

3 [http://www.dbmsmag.com/9605d15.html](http://www.dbmsmag.com/9605d15.html)
The classic analogy to help you understand database indexes is the index in the back of reference books. Sure, if you wanted to find everything in the book about a particular subject you could start at the beginning and scan every page, but it is much faster to look in a smaller, alphabetized subject index that directs you to a list of pages. Then you need to scan only those pages to find information about your chosen subject. Not everything in the book is indexed, however, so if your subject is not mentioned in the index, you must still scan for it. Likewise, a database index is a look-up mechanism that helps a DBMS find the information you request faster than it could with a full scan. As with book indexes, not everything in the database is indexed, so an occasional scan may still be necessary.

The primary reason to build an index is to improve performance. But it is not the only reason to build an index. The second reason has to do with enforcing uniqueness among rows stored in a database table. Tables in a SQL database are usually designed with a primary key; that is, a set of columns with a unique value that identifies a row in the table. When a new row is inserted into a table defined with a primary key, it is up to the DBMS to ensure that the primary key value for that row is unique. Performance would be unacceptable if the DBMS had to scan the entire table each time a new row was inserted. Therefore, the accepted solution is to build a unique index on the primary-key columns and let the DBMS use that as the physical enforcement mechanism for the primary key uniqueness requirement.

1. **What makes a good primary key?**
   Any attribute or combination of attributes that satisfy the three requirements above (unique, not null, and indexed) can be a primary key. However, some attributes are more appropriate to be select as the primary key than others. For instance, if all of my students have unique first names I could make student first name as the primary key for the student table. But that doesn’t happen very often. Each semester I have a few students named John and a couple students named Michael. What about last name you might ask. Again, if I could be sure it would be unique so I never have more than one student with the last name of Patel or Smith or Jones, that would be fine. But again, that doesn’t happen most semesters and we want our primary key to ALWAYS be unique. So what can I do? We can start adding attributes together. For instance, instead of making the student’s last name the primary key, I could make a combination of first name, last name, and date of birth. It is unlikely I would have two students who have the same name and birthday.

   While it is acceptable to have a primary key which is a text or a date type or some combination thereof, it is easier to have a primary key that is an integer. As well learn later, we use primary keys and foreign keys to “join” tables (more about this in the SQL tutorial) and it is easier to join two attributes that have an integer data type.

   So what do you do if you don’t have an attribute for a table that is unique and not null and an integer? No problem. Just create a new attribute and enforce those rules.

2. **Composite key:**
   As I’ve noted, a primary key does not have to be made of just one attribute. Whenever a primary key is composed of more than one attribute it is called a composite key. As long as it meets the requirement to be unique not null and indexed, you can add as many
attributes together to make a composite key as you like. The only limitation is that it becomes increasingly difficult to join tables. We’ll talk about joins later.

B. Foreign Key:

As we’ve learned, redundant data leads to insert, delete, and modification anomalies. Wherever possible, we want to avoid redundant data. By designing a database carefully, we can do a very nice job of keeping information once and only once to ensure that if it needs to be added, changed, or deleted, we can perform the function one time, and all of the rows in a table that use that data will refer to the correctly inserted, updated, or deleted information.

We’ve also learned that we keep information about a particular item in a table, and a database can be composed of a number of tables which are related to each other in some way, all pertaining to an aspect of a business. So how do we make these tables related to each other? That’s the beauty of the foreign key.

A foreign key is an attribute in a table which is repeated in another related table. Wait, didn’t we just say we didn’t want redundant data? Yup! But, this is a form of CONTROLLED redundancy. In other words, we don’t want to keep all kinds of duplicate data but we can keep just one piece of information (i.e. the primary key) of one table, and repeat it as a foreign key in another table and now we can link the tables together.

Here’s an example. Imagine I’m designing a database that houses information about courses at a college. Here is just a piece of the database model. You can see there is a course table that has a number of attributes related to a course. There is a discipline table that holds information about the various majors (disciplines) offered, and there is a school table that describes the schools at the university.

Let’s take a look at the data in the discipline and the school tables.
We can see that the number stored in the school attribute of the discipline table matches to the school_id value in the school table. Now let’s consider row 10 in the discipline table. The school associated with SOC is 4 – or SCT-School of Communications and Theater. What would happen if there was a big reorganization at the university and all of the disciplines that were at Fox were transferred to SCT? We’d need to type SCT-School of Communications and Theater 6 times. That’s a lot of typing! What if we mistyped it on a few occasions? Can you see how easy it would be to have errors in the data whenever there is an addition, modification or deletion of a value? But if you just use a foreign key (school) in the discipline table to the school_id (primary key) of the school table, all you need to do is add, update or delete the value of 1 row. Isn’t that much easier? Sure! That’s the beauty of foreign keys!

We’ll discuss foreign keys in more detail in just a bit.

VI. Integrity Rules

There are two essential integrity rules that we follow when building ERDs

A. Entity Integrity:

Entity integrity is imposed on a database to ensure that each table’s primary key is unique and not null. Consider the following customer table which has Customer ID as the primary key:

<table>
<thead>
<tr>
<th>CustomerID</th>
<th>FirstName</th>
<th>LastName</th>
<th>Address</th>
<th>City</th>
<th>State</th>
<th>ZipCode</th>
<th>Telephone</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>David</td>
<td>Bowie</td>
<td>1515 Imom Way</td>
<td>Philadelphia</td>
<td>PA</td>
<td>19121-</td>
<td>(215) 415-3555</td>
</tr>
<tr>
<td>5</td>
<td>Cyrus</td>
<td>Alvey</td>
<td>1234 Breaty Hear</td>
<td>Philadelphia</td>
<td>PA</td>
<td>19141-</td>
<td>(215) 468-6868</td>
</tr>
<tr>
<td>6</td>
<td>John</td>
<td>Alvizures</td>
<td>55566 Guat Road</td>
<td>Cherry Hill</td>
<td>NJ</td>
<td>08002-</td>
<td>(856) 194-1858</td>
</tr>
<tr>
<td>7</td>
<td>Barack</td>
<td>Obama</td>
<td>4444 Johnson Way</td>
<td>Philadelphia</td>
<td>PA</td>
<td>12939-1999</td>
<td>(215) 606-0606</td>
</tr>
<tr>
<td>8</td>
<td>Bob</td>
<td>Smith</td>
<td>8248 Walnut Avenue</td>
<td>Trenton</td>
<td>NJ</td>
<td>08205-</td>
<td>(856) 777-7777</td>
</tr>
</tbody>
</table>
Database Design and Entity Relationship Diagrams

Do you think I can enter the following 3 rows into the database?

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>Cindy</td>
<td>Marselis</td>
<td>1810 N. 13th St</td>
<td>Phila</td>
<td>PA</td>
<td>19122</td>
<td>215.204.3077</td>
</tr>
<tr>
<td></td>
<td>Munir</td>
<td>Mandviwalla</td>
<td>1810 N. 13th St</td>
<td>Phila</td>
<td>PA</td>
<td>19122</td>
<td>215.204.8172</td>
</tr>
<tr>
<td>9</td>
<td>Bob</td>
<td>Smith</td>
<td>8248 Walnut Ave.</td>
<td>Trenton</td>
<td>NJ</td>
<td>080205</td>
<td>856.777.7777</td>
</tr>
</tbody>
</table>

1. If I try and enter the first row, I’d would violate entity integrity. Entity integrity ensures the primary key cannot be null and must be unique. Since a row already has a primary key of 8, the first row cannot be added. It doesn’t matter that the data in the row is different since entity integrity is concerned with the primary key and this is a duplicate.

2. The second row has a null primary key, so this also should not be added.

3. What about the third row? This has the same data as customerId 8. Would this be excluded as well? No. Entity integrity ensures that the primary key is not duplicated or null. Since there are no rows with a primary key of 9 and the field is populated (not null) it does not violate entity integrity. Also, isn’t it possible that there are two Bob Smiths (perhaps Bob Senior and Bob Junior) who live in the same house? It could be that this is duplicated data and that is not desirable but it also is possible that there are indeed two people with the same name and address in the database. There are methods that organizations use to try to determine if rows such as these are duplicates but that discussion is outside of the scope of this class.

B. Referential Integrity

Referential integrity ensures that a foreign key matches to a primary key. Let’s look at the following example. As we saw earlier, each discipline is in a school within Temple. Now let’s say I add a new discipline for Dentistry into the discipline table, and I want this to be added to school 7. The database will not allow this row to be stored since the foreign key (7) does not have a matching primary key in its parent table School.

Note, a primary key may match a foreign key or a null value. For example, you see that SCT in the School Table (School_ID 4) does not have any corresponding rows in its child table Discipline. That is acceptable. Think of it like this. A school can be created and then its children (disciplines) will be added to it but a child cannot be added to a non-existent parent. So a discipline cannot be added if it doesn’t have a school to go to. A row with a primary key of 7 must be created first in the school table first before it can be referenced by a child row in another table.

So the rules are:

- A foreign key must match to a primary key
- A primary key must match to a foreign key or a null value
VII. Relationship Rules

Now that we understand a bit about integrity rules, let’s explore the rules related to relationships. Relationships must be defined into one of three types (1:1, 1:M, M:N). Once it is defined, the relationship must be resolved (converted into a format that can be stored in a database). Here’s how we handle the relationship type and resolution. We will see examples of this in Chapter 3 but for now, just try and digest the concepts.

A. Determine relationship using this terminology: (i.e. relationship between student and dorm rooms)

- 1 of A is related to X (1 or many) of B
  - i.e. 1 student is assigned to 1 dorm room
- 1 of B is related to X (1 or many) of A
  - i.e. 1 dorm room is assigned to many students

The decision will be as follows:

1. 1:1
   - a. 1 of A is related to 1 of B
   - b. 1 of B is related to 1 of A

2. 1:M
   - a. 1 of A is related to many of B
   - b. 1 of B is related to 1 of A

3. M:N
   - a. 1 of A is related to many of B
   - b. 1 of B is related to many of A
B. Resolve the relationship

1. If the relationship is a 1:1, it is assumed that the entity is just another attribute for that table. Add it as another attribute to an existing entity.
   a. For instance, if you have TUID and student, a student can have only one TUID and a TUID is assigned to one student. Include the TUID as an attribute to the student table.

2. If the relationship is a 1:M, the primary key of the one side is duplicated as the foreign key on the many side. The rule is that the foreign key ALWAYS goes on the many side.
   a. The names of the primary key and the foreign key do not need to match. Only the data type needs to be the same.
   b. Of course, the values of the data stored in the field must match as well or there cannot be a join.

3. If the relationship is a M:N, resolve the M:N relationship into two 1:M relationships. To resolve the M:N relationship into two 1:M relationships:
   a. Create a new table which is an associative entity (AKA composite entity or bridge entity). The purpose of the associative entity is to function as a bridge between the two entities. This table must include the primary keys of the two entities as foreign keys. This makes sense since the associative entity is now the many side of both 1:M relationships. Since the foreign key ALWAYS goes on the many side of a relationship and the associative entity is ALWAYS the many side of the relationship, both of the foreign keys would be placed in the associative entity.
   b. If the combination of the 2 foreign keys is unique, it can be used as the primary key of the associative entity. Since the primary key will be composed of 2 primary keys, it is called a composite key.
   c. If the combination of the 2 foreign keys is not unique, leave the 2 foreign keys in the associative entity and create a new primary key for the associative entity.
Chapter 3: Example of Creating an ERD

We’ll start off with a small example. Let’s design a database for Glenside Bank. We’ll consider each of the business rules step by step. We’ll also explore some database design concepts as we go.

I. 1:1 Relationships

The bank has customers and it collects basic demographic information such as name, address, social security number, gender, date of birth, phone number, for each customer. Customer seems a good candidate for an entity as it is a noun and we are collecting information about that customer. So let’s create a bank customer table. There can be no spaces in between the words in attributes or tables names, so the name of this table will be bank_customer.

Let’s consider what information (attributes) we’ll keep about our customers.

A. Our first step is to identify a good primary key for our table. You might be tempted to use a person’s social security number, but a social security number isn’t always unique since at one point duplicate numbers were issued. Also, in the past, not everyone was issued an SS#, so the value can be null. This violates 2 of the 3 rules about a primary key. We would also consider some combination of the customer’s first name, last name, and date of birth as the primary key but this might get a bit cumbersome. To make it easier for ourselves later, let’s just create a new attribute called Customer_ID.

B. Now we’ll consider social security number, date of birth, phone number and name. The social security number is unique to one customer (hopefully) and a customer should have only one social security number (again hopefully). This is clearly a 1:1 relationship. If we review our relationship rules, if a relationship is 1:1, it is assumed that the entity is just an attribute for that table. Add it as an attribute to the bank_customer entity.

C. Derived (calculated) field:

1. What about date of birth? A customer can only have one birth date, right? But it is possible for more than one customer to have the same birthday. Is this a 1:M relationship? You could say that, but this type of data is virtually always handled as a 1:1 relationship so we’ll just add date of birth to the bank_customer table.

2. Why am I collecting the date of birth rather than the customer’s age. Unfortunately, we are all getting older every nano second, and your current age will not be the same at this moment as it will be a second, month, or year from now. However, you date of birth stays the same. We can always calculate (derive) your age by taking the current date and subtracting your birth date so it is better to keep your birthdate. Storing the birthdate has another advantage over storing age. What if we want to send a birthday message to anyone whose birthday is in July. If I store age I won’t be able to do this but if I store birthday, that’s an easy calculation.

3. You’ll find there are numerous times where you’ll be tempted to store a derived field (subtotals, totals, calculations, etc.) but it is always preferable to store the raw attribute and do the math as a calculation. In the SQL section of this book you’ll learn how to handle calculations.

D. What about address? Address is like the date of birth in that there could be multiple customers with the exact same address (such as spouses or parents and children) but it is common to keep this information within the customer table. Remember with address we want to keep the attribute atomic, so we’ll keep address 1, city, state, and zip as separate attributes.

E. We’ll handle phone number the same way as address and keep the phone in the bank_customer table. Although it is true the customer can have lots of phone numbers, from
the bank’s perspective, it really only needs one. We’ll just have one telephone number per customer.

F. We also want to keep personal information such as the person’s gender. Try to avoid calling this attribute sex as that has a number of different connotations. Rather, we’ll name this attribute gender.

G. Of note, you can add attributes to a table in any order you wish but makes more sense to group the attributes into similar areas and in a logical order. As such, we put the components of the address lumped together and place the attributes in the same order as a typical mailing address.

H. There could be lots of other pieces of information we’d like to keep about the customer but let’s go with this. Our customer table now appears as follows with customer_id as the primary key (delineated by the key symbol to the left of the attribute name):

```
bank_customer

<table>
<thead>
<tr>
<th>CUSTOMER_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSTOMER_ID</td>
</tr>
<tr>
<td>CUSTOMER_ID</td>
</tr>
<tr>
<td>CUSTOMER_ID</td>
</tr>
<tr>
<td>CUSTOMER_ID</td>
</tr>
<tr>
<td>CUSTOMER_ID</td>
</tr>
<tr>
<td>CUSTOMER_ID</td>
</tr>
<tr>
<td>CUSTOMER_ID</td>
</tr>
<tr>
<td>CUSTOMER_ID</td>
</tr>
<tr>
<td>CUSTOMER_ID</td>
</tr>
<tr>
<td>CUSTOMER_ID</td>
</tr>
<tr>
<td>CUSTOMER_ID</td>
</tr>
<tr>
<td>CUSTOMER_ID</td>
</tr>
<tr>
<td>CUSTOMER_ID</td>
</tr>
<tr>
<td>CUSTOMER_ID</td>
</tr>
<tr>
<td>CUSTOMER_ID</td>
</tr>
<tr>
<td>CUSTOMER_ID</td>
</tr>
<tr>
<td>CUSTOMER_ID</td>
</tr>
<tr>
<td>CUSTOMER_ID</td>
</tr>
<tr>
<td>CUSTOMER_ID</td>
</tr>
<tr>
<td>CUSTOMER_ID</td>
</tr>
<tr>
<td>CUSTOMER_ID</td>
</tr>
</tbody>
</table>

| ADDRESS_LINE1 |
| ADDRESS_LINE2 |
| ADDRESS_LINE3 |
| CITY |
| STATE |
| ZIP |
| GENDER |
| DOB |
| HOME_PHONE |
| SS |
```

II. 1:M Relationships

The next two business rules coincide with one another so I’m going to consider them together. The bank offers a number of accounts such as a checking account, savings account, money market account, CD, etc. The bank has a number of branches. The bank needs to track where each account was opened.

Looks like we need an account table. What type of information will we need in our account table? Certainly we’ll need a primary key to uniquely identify each account in the account table. Account_Number seems like a good one since it is unique and shouldn’t be null. We also need to store the state where the account was opened. In addition, we need to track the type of the account (checking, savings, money market, CD, etc.), the branch where the account was opened, and the billing cycle date (that’s the day of the month when you get your statement like the 15th of the month so the value is 15). Let’s start with this list of attributes:

- Account_number
- State_opened
- Account_type
- Branch_opened
- Billing_cycle_day

We could add an attribute for account type with a data type of varchar2 (text). In this situation, the user would need to type in the word “checking”, “savings”, “money market”, etc. each time a new row was added to the account table. But that’s quite a bit of typing! Also, what happens
when someone types Checking and someone else types checking, and a third person types check? If I want to see all of the checking account customers, I’ll only see a subset of them. Why? To a computer upper case is different than lower case. To make it easier for the user to enter information and to ensure information is entered in a consistent manner to facilitate data retrieval it would be better to give the user a drop down box which has a list of the acceptable values for that field. The user then just has to click on one and the value will be filled into the field. It’s like when you order something on the web. You don’t type your state in to the state field when you enter your address. You typically see a drop down list and you click on your state.

A. When to use a look up table?

It is wise to consider a lookup table when the possible values of the attribute are not limitless. Therefore, we don’t usually give a lookup table for birthdates, street addresses, first names or last names. However, you will typically see them for products and anything where the item is categorized or grouped into a particular type like country, region or ethnicity, student class type, etc.

B. What does a lookup table include?

A lookup table can include any attribute, but most commonly it will include 2 fields, the primary key and a description. For instance, when you pick your state when ordering something online, the state lookup table typically has a primary key which is the 2 letter state abbreviation and the state’s full name.

Here is the structure of the state table as well as a subset of some of the values stored in the table.

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>STATEID</td>
<td>DE</td>
</tr>
<tr>
<td>STATENAME</td>
<td>Delaware</td>
</tr>
<tr>
<td></td>
<td>NJ</td>
</tr>
<tr>
<td></td>
<td>New Jersey</td>
</tr>
<tr>
<td></td>
<td>NY</td>
</tr>
<tr>
<td></td>
<td>New York</td>
</tr>
<tr>
<td></td>
<td>PA</td>
</tr>
<tr>
<td></td>
<td>Pennsylvania</td>
</tr>
</tbody>
</table>

C. Building a relationship with a look up table.

A lookup table is really just an example of a 1:M relationship.

- A customer lives in one state.
- A state includes many customers.
- State is the 1 side of the relationship
- Bank_customer is the many side of the relationship

As the relationship rules tell us, in a 1:M relationship, the primary key of the one side is duplicated as the foreign key on the many side. Remember the rule is that the foreign key ALWAYS goes on the Many side. The names of the primary key and the foreign key do not need to match – only the data type needs to be the same.

To make the relationship, we’ll take the primary key of the 1 side (state) and make it a foreign key in the many side of the relationship (bank_customer). This means we’ll take stateid from the state table and add this attribute to the bank_customer table. We don’t have to call it stateid and in fact, I’ve just called it state. Now our relationship appears as follows:
When user is entering information into the bank_customer table, he will select a value from the state table. The primary key (stateid) of the state table will be duplicated in the bank_customer table as a foreign key (state).

D. Additional examples of lookup tables related to the account table.

1. Account type: We’ve seen we need to track the account type. Here’s an example of what the account type would look like in the structure as well as the values:

   Just like before, this is a 1:M relationship. An account is of one type. An account type includes many accounts. We’ll take the primary key (account_type_id) of the one side (account_type table) and make it a foreign key (account_type) on the many side of the relationship (account table).

   Here’s what the structure looks like:
Here’s what the corresponding data looks like:

<table>
<thead>
<tr>
<th>ACCOUNT_Number</th>
<th>BRANCH_ID</th>
<th>account_type</th>
<th>BILLING_CYCLE_DAY</th>
<th>STATE_OPENED</th>
</tr>
</thead>
<tbody>
<tr>
<td>10002999</td>
<td>10100</td>
<td>1</td>
<td>22</td>
<td>PA</td>
</tr>
<tr>
<td>20978983</td>
<td>10333</td>
<td>2</td>
<td>15</td>
<td>PA</td>
</tr>
<tr>
<td>23431234</td>
<td>10393</td>
<td>2</td>
<td>15</td>
<td>PA</td>
</tr>
<tr>
<td>34235254</td>
<td>20003</td>
<td>3</td>
<td>15</td>
<td>PA</td>
</tr>
<tr>
<td>77766142</td>
<td>34235</td>
<td>2</td>
<td>25</td>
<td>NJ</td>
</tr>
<tr>
<td>98746234</td>
<td>23431</td>
<td>4</td>
<td>15</td>
<td>NJ</td>
</tr>
</tbody>
</table>

You can see here that account_number 98746234 has an account_type of 4 so it must be a CD.

2. Branch: You’ll note we need to store the branch where the account was opened. The Account was opened at one branch. A branch had many accounts opened at its location.

For branch, we need to keep additional pieces of information other than just the primary key (I’ll call this branch_id) and description (branch_name). We also need to keep the branch’s address as well as the date the branch was opened. Its design will look like this:

The data stored in the table will look like this:

<table>
<thead>
<tr>
<th>BRANCH_ID</th>
<th>BRANCH_NAME</th>
<th>BRANCH_Ad</th>
<th>BRANCH_Cit</th>
<th>BRANCH_STATE</th>
<th>BRANCH_ZIP</th>
<th>DATE_OPENED</th>
</tr>
</thead>
<tbody>
<tr>
<td>10100</td>
<td>Abington</td>
<td>10101</td>
<td>Highland, Abington</td>
<td>PA</td>
<td>19007</td>
<td>1/1/2005</td>
</tr>
<tr>
<td>10333</td>
<td>Huntingdon Vall</td>
<td>1111</td>
<td>Huntingdon Huntingdon Vall</td>
<td>PA</td>
<td>19006</td>
<td>5/16/2006</td>
</tr>
<tr>
<td>10393</td>
<td>Feasterville</td>
<td>1003</td>
<td>Street Rd, Feasterville</td>
<td>PA</td>
<td>19053</td>
<td>11/12/2006</td>
</tr>
<tr>
<td>20003</td>
<td>Ambler</td>
<td>9888</td>
<td>Meetingh, Ambler</td>
<td>PA</td>
<td>19064</td>
<td>11/15/2006</td>
</tr>
<tr>
<td>34235</td>
<td>Cherry Hill</td>
<td>12495</td>
<td>Brace Rd, Cherry Hill</td>
<td>NJ</td>
<td>08002</td>
<td>5/6/2006</td>
</tr>
</tbody>
</table>

Now branch_state can use the same state lookup table that is linked to the bank_customer table. Similarly, state_opened attribute in the account table can use the stateid in the
III. M:N Relationship

We know from our business rules that a customer can have many accounts (i.e. a checking and a savings account) and an account can be associated with multiple customers (a husband and wife could have a joint checking account). The bank needs to track the customers associated with each account and the date that customer was added to the account. If the customer was removed from the account, the bank needs to track the date this occurred.

A. Resolving M:N Relationships

Clearly, we have a many to many relationship here between bank_customer and accounts. Upon review of our relationship rules:

- If you have a M:N relationship, you need to resolve it into two 1:M relationships. To do this, create an associative entity (AKA composite entity or bridge entity) which has the primary keys of the two entities as foreign keys.

Here are our two entities:
1. We need to make a new table which has the primary keys of the bank_customer table (customer_id) and the account table (account_number) as foreign keys to the new table. In addition to the foreign keys from the account and the bank_customer table, we need to add an attribute to collect the date this account was opened. Let’s call this new table account_assignment. It will look like this:

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>BANK_ACCOUNT_ID</td>
<td>Number</td>
</tr>
<tr>
<td>BANK_CUSTOMER_ID</td>
<td>Number</td>
</tr>
<tr>
<td>DATE_OPENED</td>
<td>Date/Time</td>
</tr>
</tbody>
</table>

2. Since the foreign key ALWAYS goes on the many side of a relationship and the associative entity is ALWAYS the many side of the relationship, the foreign key for both of the original entities must go on the associative entity.

B. Composite key:

1. Bank_account_id is the foreign key to the customer_id primary key in the bank_customer table. Customer_id is the foreign key to the account_number primary key in the account table. Bank_account_id and Bank_customer_id are a composite primary key in the account_assignment table. Remember a composite key is a primary key made up of more than one attributes.

2. If the combination of these attributes is unique, the combination of the foreign keys is the primary key of the associative entity. In this case, can a customer open the same account on a day? No – then it is unique. Therefore, the primary key can be the composite key of bank_account_id and bank_customer_id.

Here is the resolved M:N relationship:
3. Let's take a look at the data for these tables:

**Bank_Customer**

This is showing that the first row in the account_assignment table is for customer 10100. If we look at the bank_customer table we can see that this number corresponds with Dean Martin. He has an account which was opened at branch 10100 on 1/3/2007 in PA.

If we look at the next row, which is also for Dean Martin, we can see he also opened an account with the number of 20978983 in PA on 12/19/2006. This account is jointly owned by customer 10300 – Amanda Mobley.

**IV. Final ERD**

When we put all of the entities and their appropriate relationships together into one diagram, we develop the following ERD model of our database. Note that all of the relationships are now 1:M since all of the M:N relationships have been resolved into 1:M relationships.
V. **Placements of attributes in tables**

One area where new database designers can get confused is the placement of attributes in tables. In this scenario you'll notice I added an attribute, date_opened, into the account_assignment table. You may wonder why I put the attribute in that table.

A. When you are considering attributes you need to come back to the definition of a primary key. A primary key's purpose is to uniquely identify every row in a table and to allow the user to retrieve the values of all the attributes in that row. If I put the date_opened field in the bank_customer table, it would mean that the value of the date_opened field (i.e. what date the account was opened) is determined solely by the customer. Since a customer can have more than one account and each could be opened on different dates the value of date_opened is determined by more than the customer.

1. What if we put the date_opened attribute in the account table? This makes sense except that multiple customers can be on one account. For instance, a husband and wife can open an account, and both go and sign the paperwork on different days. In this situation, the date that the customer opened the account would be different for both customers. Therefore, date opened is not solely determined by the account.

2. Clearly, date_opened is determined by both the customer AND the account, and therefore it must be placed in the associative entity, the account_assignment table.

B. Most of the time people run into problems with the placement of attributes when working with a M:N relationship. If you find yourself confused, just consider each table independently and decide if the value of the attribute is determined by only one side of the relationship. If it is determined by both sides of the relationship, place the attribute in the bridge entity.
Chapter 4: Invoice Example

Now that we’ve seen a full ERD example, we’ll go one step farther. Organizations typically exist to make and or provide a product or a service for a customer or client. As such, companies typically need to make an invoice or a receipt which is given to the customer for payment. Since the requirement to create an invoice is basically universal in all organizations, let’s explore how to handle this through a database.

In a new scenario, what if you order books online from a publisher for next semester. When you receive the book in the mail, it will have an invoice that looks something like this:

```
Customer: Bruce Springsteen
1818 Rock and Roll Way
Rumsford, NJ 08045

Order Date: 6/3/2008
Order Clerk: 144444
Ship Date: 6/5/2008

ISBN | Item Description            | Product Type | Unit Price | Quantity | Subtotal |
-----|------------------------------|--------------|------------|----------|----------|
141414| Databases R Amazing IS      | 100.00       | 3          | 300.00   |
333333| The History of Rock and Roll Music | 200.00     | 1          | 200.00   |
455050| How to Make a Lot of Money Finance | 500.00     | 1          | 500.00   |

TOTAL 1,000.00
Tax 60.00
Grand Total 1,060.00
```

I. Entities and Attributes:

Let’s decompose each piece of the order. Invoices typically include 4 main pieces which correspond to the sections of the invoice above:

- Customer
- Shipper
- Invoice Information
- Items purchased

A. Customer Information

1. When you place an order, the company needs to know basic information about you. This might include your full name, home address, credit card information (number, type of card, expiration date, etc.) and your shipping information. Of course, the company needs to uniquely identify each customer, so it will typically issue a customer identifier.

2. Not all invoices will have customer information. For instance, imagine you go to 7-Eleven and buy some munchies for lunch. On the receipt you receive it will not include customer information since you don’t typically give any data about yourself when you shop at a store. But what about if you shop at a place that uses a frequent shopper card like a
Database Design and Entity Relationship Diagrams

super market? In that case, your customer information will probably show up on the receipt. The presence of customer information will be driven by the needs of the organization.

B. Seller Information:
An invoice typically includes basic information about the store from which you purchased the item. If it is a purchase at 7-Eleven, it might just say 7-Eleven and show the store number and address. Again, the amount of the information displayed will be driven by the needs of the organization.

C. Product Information
An invoice or receipt will normally show the products that were purchased. Typically it will include the product identifier (such as a SKU or an ISBN), a description about the product, and perhaps some information as to the type of the product.

1. In the invoice above you see information such as subtotal, total, and tax. All of these are derived or calculated fields as described earlier. These attributes would not be stored in the database, but rather would appear on a report based on a calculation you perform.

2. For product type, we probably want a lookup table since we don’t want the user to enter free text. Rather, we’d like to have the user pick the value from a list to ensure there is consistency in the information collected and to minimize data entry. As you’ll recall, when we add a lookup table it is simply a 1:M relationship. A product is of one type, and a type has many products associated with it.

D. Invoice Information
Virtually all receipts and invoices have some basic identifying information about the purchase which includes an invoice or receipt number, perhaps a barcode, an invoice date and time, and possibly the name of the person who placed the order. If the item was shipped it might include the shipping date. At a store it might include the cashier’s identifying number. All of this type of information will be housed in a table that you might call invoice, receipt, or order.

II. Relationships:
Now that we’ve identified the basic entities, we’ll move on to determining their relationships.

A. Customer to Invoice
An invoice is issued for one customer. A customer can purchase items or services from an organization on multiple occasions and each purchase is tracked through an invoice. Therefore, a customer can have many invoices but an invoice is for only 1 customer. The relationship between customer and invoice is 1 to many, with invoice as the many side of the relationship.

B. Seller to Invoice
An invoice is created by one seller. A seller creates many invoices (or they wouldn’t stay in business too long). The relationship between a seller and an invoice is also 1 to many, respectively.
C. Product to Invoice

When we buy something at a store or online we can purchase more than one item during the visit or transaction. It would be cumbersome and very time consuming if the store had to create a different invoice for each item purchased so the store adds all the products purchased during that one transaction on one invoice. For instance, in the example above, Bruce Springsteen bought three books and all appear on one invoice.

Now Bruce Springsteen is not the only person who might purchase a copy of *Databases R Amazing* or *The History of Rock and Roll*. Therefore, a product can be on multiple invoices. Accordingly, the relationship between product and invoice is M:N.

1. Be careful when thinking through these relationships. You might have thought that only one person could buy a book. It is true that an individual copy of a book could only be purchased by one person but what we are tracking in this database is a product, not an instance of a product. In other words, the bookstore has a bunch of copies of *Databases R Amazing*. It doesn’t track each individual copy of the book. Rather, it only knows there were X number of copies of the book. With RFID (radio frequency ID) you can in fact track each individual instance of an item. However, most stores still work on bar coding which just tracks a product and the number in stock.

2. Now that we know that the relationship between product and invoice is M:N, we need to review our relationship rules. As you’ll recall, if you have many to many relationships, you must resolve it into 2 one to many relationships. We resolve it by creating a new entity called a bridge or associative entity (it bridges or associates the two entities).
   a. The bridge entity must include the primary keys of the two tables as foreign keys.
   b. If the combination of these attributes is unique, the combination of the foreign keys can be used as the primary key of the bridge entity. If you use the 2 foreign keys as a primary key, the primary key is a composite key as it is composed of more than one attribute.
   c. If the combination of the foreign keys is not unique, then you need to create a new attribute to be the primary key.

3. In industry, the bridge entity between the invoice and the product table is typically called the invoice detail table. That is because it shows the line items (details) on the invoice.

III. ERD

Below is an example of what the basic entity relationship diagram for an invoice database would look like. As you’ll note, it typically includes tables pertaining to the

- Product
- Customer
- Seller
- Transaction (the invoice, receipt, or order table)
- Order details (the bridge/associative entity between invoice and product showing the invoice detail).

You’ll have a number of other tables as required by the specific needs of an organization but this is the basic design of invoicing systems.
IV. Variations on a theme

Based on the business, some of these basic tables would be called different things. For example, if you were creating a system for the Bursar’s office at Temple, the customer would be called Student. The Product would be Course. You would probably delete the Seller table since all of the services (courses) are provided from one “store” (Temple). The Invoice Detail table will not include quantity ordered or shipped since we don’t ship courses or take multiple sections of the same class.

If the business was a doctor’s office, the Product might be service (i.e. chest x-ray, physical exam, vaccination), the Customer would be called Patient, and the Seller might be Clinic since one doctor could practice in multiple locations. The Invoice table might be called Patient Bill and the Invoice_Detail table will not include quantity shipped or ordered.

Clearly, additional tables and attributes could be added to collect more information based on the organization’s needs. But in general, this invoicing structure is fairly stable across all types of firms.
Chapter 5: Data Dictionary

I. What is a data dictionary?

We’ve spent some time discussing ERDs and while the design is critical to the functionality of the database, the data dictionary is essential as well. The data dictionary is just what the name implies – a dictionary about the data housed in the database. The data dictionary holds metadata, or data about the data. Hmph, was that confusing?

Let me give you an example. I’m a new database developer at Glenside Bank. The manager comes to me and asks me to write a report of all of the accounts opened in Abington in the prior month. That seems easy enough. But, when I write the report I return no rows. Upon closer inspection of the data, I realize that Glenside stores dates in military format (i.e. MMM-DD-YYYY such as JUN-17-2008) and I had written the query assuming the date was stored as MM/DD/YYYY (such as 06/17/2008). How would I know the format of the attributes – just look it up in the data dictionary!

A. Although there is no standard for what should be contained in the metadata, you might find the following information for each table in the database. Some of these items will be explained more fully in the Access and SQL Tutorials.

- Table description (explaining the basic kind of information stored in the table)
- If the table comes from an outside source, it may include the format in which the table was received, the date of receipt, the date it was loaded into the database, and when an update may be expected.
- Attribute name, field size, data type, whether it is a primary key, and if it is required (cannot be left null), formats for the value, levels of precision, and input masks
- Information relating to relationships such as whether an attribute is a foreign key. If so, what table it references.

This is an example of a piece of a data dictionary:

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Created</th>
<th>Modified</th>
<th>Fields</th>
<th>Records</th>
</tr>
</thead>
<tbody>
<tr>
<td>Product_ID</td>
<td>2/23/07</td>
<td>4/13/07</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>Product_Name</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Product_Type</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Product_Unit_Price</td>
<td></td>
<td></td>
<td></td>
<td>&lt;0</td>
</tr>
</tbody>
</table>

B. Many database management systems are considered to be self-describing. In other words, they document the information which is stored in the data dictionary automatically when tables and attributes are added to the database. It is simply a matter of querying the data to retrieve the information.

C. A data dictionary can also be used to create an ERD of a database if one has not been provided. Since the metadata typically shows which attribute is a primary key, which attributes are foreign keys and the table referenced, you could take a dictionary and recreate the ERD.
Section 2: Access Tutorial

Chapter 6: Introduction

I. Access 2007 Interface

A. The Ribbon

Microsoft Access 2007 offers a new user interface that includes a standard area called the Ribbon, which contains groups of commands that are organized by feature and functionality. The Ribbon replaces the layers of menus and toolbars found in earlier versions of Access.

![Ribbon Image]

Use the Ribbon to locate groups of related commands faster. For example, if you need to create a form or report, use one of the commands on the Create tab. Commands are placed closer to the surface, which means that you do not need to dig for them in menus or memorize their locations.

B. Navigation Pane

The Navigation Pane lists and provides easy access to all of the objects in the currently open database. Use the Navigation Pane to organize your objects by object type, date created, date modified, related table (based on object dependencies), or in custom groups that you create. You can easily collapse the Navigation Pane so that it takes up little space, but still remains available. The Navigation Pane replaces the Database window that was used in versions of Access earlier than Access 2007.

![Navigation Pane Image]
Chapter 7: Tables

I. Introduction to Tables

Tables are grids that store information in a database similar to the way an Excel worksheet stores information in a workbook. Access provides three ways to create a table for which there are icons in the Database Window. Double-click on the icons to create a table. For our purposes, we will create a table in design view.

A. Create a Table in Design View

Design View allows you to define the fields in the table before adding any data to the datasheet. The window is divided into two parts: a top pane for entering the field name, data type, and an optional description of the field, and a bottom pane for specifying field properties.

B. Field Properties

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Field Name</td>
<td>This is the name of the field and should represent the contents of the field such as &quot;FirstName&quot;, &quot;Address&quot;, &quot;Final Grade&quot;, etc. The name cannot exceed 64 characters in length and may include spaces. Access has a few “words” that are considered reserved field names including the word “Name”. If you try and add an attribute with a reserved field name, Access will not allow you to save the attribute. If you run into this situation, simply enter a different name for that attribute.</td>
</tr>
<tr>
<td>Data Type</td>
<td>Description</td>
</tr>
<tr>
<td>Data Type</td>
<td>Description</td>
</tr>
</tbody>
</table>

A field name can be up to 64 characters long, including spaces. Press F2 for help on field names.
Data Type | Description
---|---
Text | The default type, text type allows any combination of letters and numbers up to a maximum of 255 characters per field row
Memo | A text type that stores up to 64,000 characters.
Number | Any number can be stored.
Date/Time | A date, time, or combination of both
Currency | Monetary values that can be set up to automatically include a dollar sign ($) and correct decimal and comma positions.
AutoNumber | When a new row is created, Access will automatically assign a unique integer to the row in this field. From the General options, select Increment if the numbers should be assigned in order or random if any random number should be chosen. Since every row in a datasheet must include at least one field that distinguishes it from all others, this is a useful data type to use if the existing data will not produce such values.
Yes/No | Use this option for True/False, Yes/No, On/Off, or other values that must be only one of two possible values
OLE Object | An OLE (Object Linking and Embedding) object is a sound, picture, or other object such as a Word document or Excel spreadsheet that is created in another program. Use this data type to embed an OLE object or link to the object in the database.
Hyperlink | A hyperlink will link to an Internet or Intranet site, or another location in the database. The data consists of up to four parts each separated by the pound sign (#): DisplayText#Address#SubAddress#ScreenTip. The Address is the only required part of the string. Examples:
- Internet hyperlink example: FGCU Home Page #http://www.fgcu.com#
- Database link example: C:\My Documents\database.mdb#mytable
LookupWizard | The LookupWizard isn’t really a data type but Access provides you an easy way to allow you to create foreign keys to lookup tables. Make sure you review the section on lookup tables in the ERD section of this book to gain a better understanding of what lookup tables offer.

The lookup wizard can read values from 2 main sources. You can tell the wizard to look up the values from an existing table or query OR you can type the acceptable values into a list that is stored with the field. In general, it is preferable to look the values up from a table since the user can just update that table whenever a new value needs to be added or a value needs to be changed. While the same is true if the value is typed in a list, this approach is more cumbersome and difficult to maintain.

You will need to have the lookup table created first.

As an example, let’s say I have an address table and I want the user to be able to pick a state from a lookup table (as you typically would if you were completing a form online). For our example, the table is called State and its structure looks as follows:
Here's a few rows stored in the table:

<table>
<thead>
<tr>
<th>State_abbreviation</th>
<th>State_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>DE</td>
<td>Delaware</td>
</tr>
<tr>
<td>NJ</td>
<td>New Jersey</td>
</tr>
<tr>
<td>NY</td>
<td>New York</td>
</tr>
<tr>
<td>PA</td>
<td>Pennsylvania</td>
</tr>
</tbody>
</table>

To use the lookup wizard:

1. Select LookupWizard from the data type drop down box.
2. Select “I want the lookup column to look up the values in a table or query” in the first screen of the wizard.
3. In the next screen, select the table which is the lookup table. In our case, you would select Table: State from the list of tables in the database.
4. In the next screen, select the values to appear in the lookup column. In this scenario, we would probably want to select both the abbreviation as well as the description. This will help in case the user isn’t entirely sure what is the 2 letter abbreviation for desired state (particularly useful with all those I states like Idaho, Indiana, Illinois, etc.)
5. In the following screen, you will be asked how you want the lookup values to be sorted. Typically you’ll want it in alphabetic or numeric order. You get to select whether the order should be ascending or descending. In this case, we’ll ask for it be sorted by the 2 letter abbreviation in ascending order.
6. In the next screen you can modify the appearance of the lookup columns. Note that there is a checkbox which indicates that the primary key will be hidden. This option is useful since many times the primary key doesn’t provide meaningful information to the user. For instance, if the lookup table was something like a product table, it would be difficult to remember all the different product identifiers so it would be better to show the product description. In our case, the 2 letter abbreviation is useful so we'll uncheck the box.
7. In the last screen, you get to give the column a name. We'll use state. Then click Finish.
8. Now whenever the user wants to enter a row and the user gets to the field with the data type of lookup, a drop down list will appear with the values in the lookup table in the selected order. The user will just need to click on the proper value and the value will be populated (stored) in that field.
### Field Properties:

**Description** (optional) – Allows user to enter a brief description of the contents of the field

**Field Size**

Select any pertinent properties for the field from the bottom pane of the Design View window. See Field Properties Table below.

**Field Properties:**

**Field Size**

Used to set the number of characters needed in a text or number field. The default field size for the text type is 50 or 255 characters based on how your application was configured. If the rows in the field will only have two or three characters, you can change the size of the field to save disk space or prevent data entry errors by limiting the number of characters allowed. Likewise, if the field will require more than 50 characters, enter a number up to 255. The field size is set in exact characters for Text type. The following are the options available for number type fields.

<table>
<thead>
<tr>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte</td>
<td>Positive integers between 1 and 255</td>
</tr>
<tr>
<td>Integer</td>
<td>Positive and negative integers between -32,768 and 32,768</td>
</tr>
<tr>
<td>Long Integer (default)</td>
<td>Larger positive and negative integers between -2 billion and 2 billion.</td>
</tr>
<tr>
<td>Single</td>
<td>Single-precision floating-point number</td>
</tr>
<tr>
<td>Double</td>
<td>Double-precision floating-point number</td>
</tr>
<tr>
<td>Decimal</td>
<td>Allows for Precision and Scale property control</td>
</tr>
</tbody>
</table>

**Format**

Ensures the data entered into a field is consistent for each row entered. For text and memo fields, this property has two parts that are separated by a semicolon. The first part of the property is used to apply to the field and the second applies to empty fields.

Some formats are pre-established and the user does not need further specification. For instance, in the currency format, there is a standard currency option. However, the user has the capability to change that standard to further customize the output. Examples are provided below to demonstrate how customization can be achieved.

**Text and memo format.**

<table>
<thead>
<tr>
<th>Format</th>
<th>Datasheet Entry</th>
<th>Display</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>@@-@@@</td>
<td>1234567</td>
<td>123-456</td>
<td>@ indicates a required character or space</td>
</tr>
<tr>
<td>@@-@@&amp;</td>
<td>123456</td>
<td>123-456</td>
<td>&amp; indicates an optional character or space</td>
</tr>
<tr>
<td>&lt;</td>
<td>HELLO</td>
<td>hello</td>
<td>&lt; converts characters to lowercase</td>
</tr>
<tr>
<td>&gt;</td>
<td>Hello</td>
<td>HELLO</td>
<td>&gt; converts characters to uppercase</td>
</tr>
</tbody>
</table>
| @
\     | Hello           | Hello!  | \ adds characters to the end       |

**Number format.** Select one of the preset options from the drop down menu or construct a custom format using symbols explained:
### Field Properties:

#### Number Format

<table>
<thead>
<tr>
<th>Format</th>
<th>Datasheet Entry</th>
<th>Display</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>###,##0.00</td>
<td>123456.78</td>
<td>123,456.78</td>
<td>0 is a placeholder that displays a digit or 0 if there is none.</td>
</tr>
<tr>
<td>$###,##0.00</td>
<td>0</td>
<td>$0.00</td>
<td># is a placeholder that displays a digit or nothing if there is none.</td>
</tr>
<tr>
<td>###.00%</td>
<td>.123</td>
<td>12.3%</td>
<td>% multiplies the number by 100 and added a percent sign</td>
</tr>
</tbody>
</table>

#### Currency Format:  This formatting consists of four parts separated by semicolons: format for positive numbers; format for negative numbers; format for zero values; format for Null values.

<table>
<thead>
<tr>
<th>Format</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$##0.00;($##0.00);[Red];$0.00;&quot;none&quot;</td>
<td>Positive values will be normal currency format, negative numbers will be red in parentheses, zero is entered for zero values, and &quot;none&quot; will be written for Null values.</td>
</tr>
</tbody>
</table>

#### Date format.

In the table below, the value "1/1/01" is entered into the datasheet, and the following values are displayed as a result of the different assigned formats.

<table>
<thead>
<tr>
<th>Format</th>
<th>Display</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>dddd&quot;,&quot;mmm d&quot;,&quot;yyyy</td>
<td>Monday, January 1, 2001</td>
<td>dddd, mmm, and yyyy print the full day name, month name, and year</td>
</tr>
<tr>
<td>ddd&quot;,&quot;mmm &quot;.&quot; d&quot;,&quot;&quot;yy</td>
<td>Mon, Jan. 1, '01</td>
<td>ddd, mmm, and yy print the first three day letters, first three month letters, and last two year digits</td>
</tr>
<tr>
<td>&quot;Today is &quot; dddd</td>
<td>Today is Monday</td>
<td></td>
</tr>
<tr>
<td>h:n:s: AM/PM</td>
<td>12:00:00 AM</td>
<td>&quot;n&quot; is used for minutes to avoid confusion with months</td>
</tr>
</tbody>
</table>

#### Yes/No

Fields are displayed as check boxes by default on the datasheet. To change the formatting of these fields, first click the Lookup tab and change the Display Control to a text box. Go back to the General tab choices to make formatting changes. The formatting is designated in three sections separated by semicolons. The first section does not contain anything but the semicolon must be included. The second section specifies formatting for Yes values and the third for No values.

<table>
<thead>
<tr>
<th>Format</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>;&quot;Yes&quot;[green];&quot;No&quot;[red]</td>
<td>Prints &quot;Yes&quot; in green or &quot;No&quot; in red</td>
</tr>
</tbody>
</table>

#### Default

There may be cases where the value of a field will usually be the same for all. In this case, a changeable default value can be set to prevent typing the same thing numerous times. Set the Default Value property.
**Field Properties:**

| **Primary Key** | Every row in a table must have a primary key that differentiates it from every other row in the table. In some cases, it is only necessary to designate an existing field as the primary key if you are certain that every row in the table will have a different value for that particular field. A social security number is an example of a row whose values will only appear once in a database table. By default, Access automatically names the first field ID and indicates it as the primary key field. However, you can designate a primary key field by right-clicking on the attribute and selecting **Primary Key** from the shortcut menu or selecting primary key from the Ribbon or Menu Bar. |
| **Indexe** | Creating indexes allows Access to query and sort rows faster. To set an indexed field, select a field that is commonly searched and change the Indexed property to Yes (Duplicates OK) if multiple entries of the same data value are allowed or Yes (No Duplicates) to prevent duplicates. |
| **Validation Rules** | Validation Rules specify requirements for the data entered in the datasheet. A customized message can be displayed to the user when data that violates the rule setting is entered. Click the expression builder ("...") button at the end of the Validation Rule box to write the validation rule. An example of field validation rules include <> 0 to not allow zero values in the row. |
| **Input Masks** | An input mask controls the value of a row and sets it in a specific format. They are similar to the Format property, but instead display the format on the datasheet before the data is entered. For example, a telephone number field can formatted with an input mask to accept ten digits that are automatically formatted as "(999) 888-7777". The blank field would look like (___) ____-_____. An input mask can be applied to a field by as shown below: |

1. In design view, place the cursor in the field that the input mask will be applied to.  
2. Click in the white space following **Input Mask** under the General tab.  
3. Click the "..." button to use the wizard or enter the mask, (@@@@) @@@@
Access 2007 Tutorial

Field Properties:
@#@@, into the field provided. The following symbols can be used to create an input mask from scratch:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Letter or digit</td>
</tr>
<tr>
<td>0</td>
<td>A digit 0 through 9 without a + or - sign and with blanks displayed as zeros</td>
</tr>
<tr>
<td>9</td>
<td>Same as 0 with blanks displayed as spaces</td>
</tr>
<tr>
<td>#</td>
<td>Same as 9 with +/- signs</td>
</tr>
<tr>
<td>?</td>
<td>Letter</td>
</tr>
<tr>
<td>L</td>
<td>Letter A through Z</td>
</tr>
<tr>
<td>C or &amp;</td>
<td>Character or space</td>
</tr>
<tr>
<td>&lt;</td>
<td>Convert letters to lower case</td>
</tr>
<tr>
<td>&gt;</td>
<td>Convert letters to upper case</td>
</tr>
</tbody>
</table>

Practice 1 – Create Tables in Access:

1. Download the Video Database from Blackboard.
2. Create the following tables in Design View.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>PK</th>
<th>Data type</th>
<th>Size</th>
<th>Required</th>
<th>Validation Rule</th>
<th>Validation Text</th>
<th>Format</th>
<th>Default Value</th>
<th>Caption</th>
<th>FK</th>
<th>Referenced Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>Distributor_ID</td>
<td>Y</td>
<td>AutoNumber</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Distributor ID</td>
</tr>
<tr>
<td>Distributor_Name</td>
<td>N</td>
<td>Text</td>
<td>35</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Distributor</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Attribute</th>
<th>PK</th>
<th>Data type</th>
<th>Size</th>
<th>Required</th>
<th>Validation Rule</th>
<th>Validation Text</th>
<th>Format</th>
<th>Default Value</th>
<th>Caption</th>
<th>FK</th>
<th>Referenced Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>Category_ID</td>
<td>Y</td>
<td>AutoNumber</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Category ID</td>
</tr>
<tr>
<td>Category_Name</td>
<td>N</td>
<td>Text</td>
<td>35</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Category</td>
</tr>
</tbody>
</table>
### Access 2007 Tutorial

#### VideoTitle

<table>
<thead>
<tr>
<th>Attribute</th>
<th>PK</th>
<th>Data type</th>
<th>Size</th>
<th>Required</th>
<th>Validation Rule</th>
<th>Validation Text</th>
<th>Format</th>
<th>Default Value</th>
<th>Caption</th>
<th>FK</th>
<th>Referenced Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>Title_ID</td>
<td>Y</td>
<td>AutoNumber</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Title ID</td>
</tr>
<tr>
<td>Video_Title</td>
<td>N</td>
<td>Text</td>
<td>50</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Video Title</td>
</tr>
<tr>
<td>Release_Date</td>
<td>N</td>
<td>Date/Time</td>
<td></td>
<td>Y</td>
<td>&gt;=#01/01/1900#</td>
<td>Short Date</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Release Date</td>
</tr>
<tr>
<td>Video_Duration</td>
<td>N</td>
<td>Number(Long Integer)</td>
<td>N</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Duration</td>
</tr>
<tr>
<td>Distributor_Name</td>
<td>N</td>
<td>(LookUp)</td>
<td></td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Distributor</td>
</tr>
<tr>
<td>Category_Name</td>
<td>N</td>
<td>(LookUp)</td>
<td></td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Video Category</td>
</tr>
</tbody>
</table>

3. Populate (enter the following rows into) the tables:

#### VideoCategory

<table>
<thead>
<tr>
<th>Category_ID</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Action</td>
</tr>
<tr>
<td>2</td>
<td>Horror</td>
</tr>
<tr>
<td>3</td>
<td>Thriller</td>
</tr>
<tr>
<td>4</td>
<td>Sci-Fi</td>
</tr>
<tr>
<td>5</td>
<td>Drama</td>
</tr>
</tbody>
</table>

#### VideoDistributor

<table>
<thead>
<tr>
<th>Distributor_ID</th>
<th>Distributor Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Crazy Video</td>
</tr>
<tr>
<td>2</td>
<td>SBC Video</td>
</tr>
<tr>
<td>3</td>
<td>DVDNow</td>
</tr>
<tr>
<td>4</td>
<td>East Park Media</td>
</tr>
<tr>
<td>5</td>
<td>Scary Entertainment</td>
</tr>
</tbody>
</table>

#### VideoTitles

<table>
<thead>
<tr>
<th>Title_ID</th>
<th>Video Title</th>
<th>Release Date</th>
<th>Duration</th>
<th>Distributor</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>Freedom Day</td>
<td>01/12/2006</td>
<td>140</td>
<td>Crazy Video</td>
<td>Action</td>
</tr>
<tr>
<td>4</td>
<td>Sea Trek</td>
<td>12/3/2000</td>
<td>190</td>
<td>Crazy Video</td>
<td>Sci-Fi</td>
</tr>
<tr>
<td>5</td>
<td>Gone with the Air</td>
<td>11/20/1964</td>
<td>300</td>
<td>DVDNow</td>
<td>Drama</td>
</tr>
<tr>
<td>6</td>
<td>Ex-Terminator</td>
<td>06/13/1999</td>
<td>200</td>
<td>SBC Video</td>
<td>Action</td>
</tr>
<tr>
<td>7</td>
<td>Buccaneers of the Caribbean</td>
<td>10/16/2006</td>
<td>119</td>
<td>East Park Media</td>
<td>Action</td>
</tr>
</tbody>
</table>
II. Table Relationships

A. Introduction to Table Relationships

A primary key in one table should match the foreign key in the corresponding table and the two attributes must be of the same data type.

B. Create Relationships

1. To view the relationships tool, select Database Tools → Relationships on the Ribbon or Menu Bar.

2. The Show Table Dialog should appear listing all of the tables in the database as below.
   a. If the dialog doesn’t appear, click on the show table icon.
   b. If you have created relationships with foreign keys, you will note that those relationships will automatically appear in the ERD. You’ll find that the tables I created will be related already.

3. Highlight the tables you created in Practice 1 and click on Add.

4. If the relationships do not create automatically, the user can create relationships on demand. Click on name of attribute which is primary key in a table.

5. Click and drag to corresponding foreign key in the child table.

6. The Edit relationships dialog will appear showing the tables and associated attributes selected.

7. The relationship type will appear automatically based on information entered when tables created.

8. To minimize data anomalies, enforce referential integrity for all relationships
   a. Right click on the relationship line
   b. The edit relationship dialog box will appear as shown below.

   ![Edit Relationships Dialog Box]

   c. Click on enforce referential integrity.
   d. Cascade on Update and Cascade on Delete will appear. Select both.
9. A line now connects the two fields in the Relationship window.

10. Expand the tables by clicking and dragging each table to ensure that each table is completely visible, there are no scroll bars, and relationships cross at a minimum.

11. Below are examples of the table with additional attributes not visible and then all the attributes visible (and no scroll bars).

Example with Scroll Bars:

Example with No Scroll Bars:

12. The datasheet of a relational table will provide expand and collapse indicators to view sub datasheets containing matching information from the other table. In the example below, the VideoTitle table and VideoCopy tables were related and the two can be shown simultaneously using the expand feature.

13. To expand or collapse all sub datasheets at once, from the Ribbon select Home -> Records -> More -> Subdatasheet -> Expand All or Collapse All.
Practice 2 - Create an ERD:

1. If you haven’t already done so, add the tables that you created in Practice 1 to the ERD in the Video Database.
2. Make sure that the tables that were provided in the download are shown and that all the tables are related as shown below:
3. Make sure to enforce referential integrity.
Chapter 8 Queries

I. Introduction

Queries select rows from one or more tables in a database so they can be viewed, analyzed, and sorted on a common datasheet. The resulting collection of rows is saved as a database object and can therefore be easily used in the future. This database is called a dynaset, short for dynamic subset. The query will be updated whenever the original tables are updated. There are various types of queries. The most typical is the select query that extracts data from tables based on specified values, find duplicate queries that display rows with duplicate values for one or more of the specified fields, and find unmatched queries display rows from one table that do not have corresponding values in a second table.

Queries are used to view, change, and analyze data in different ways. You can also use them as a source of rows for forms, reports, and data access pages (data access page: A Web page, published from Access that has a connection to a database. In a data access page, you can view, add to, edit, and manipulate the data stored in the database. A page can also include data from other sources, such as Excel).

There are several types of queries in Microsoft Access:

II. Select Queries

A select query is the most common type of query. It retrieves data from one or more tables and displays the results in a datasheet where you can update the row(s), with some restrictions. You can also use a select query to group rows and calculate sums, counts, averages, and other types of totals.

A. Wildcards:

The following table provides examples for some of the wildcard symbols and arithmetic operators that may be used. The Expression Builder can also be used to assist in writing the expressions.

<table>
<thead>
<tr>
<th>Wildcard / Operator</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>? Street</td>
<td>The question mark is a wildcard that takes the place of a single letter.</td>
</tr>
<tr>
<td>43rd*</td>
<td>The asterisk is the wildcard that represents a number of characters.</td>
</tr>
<tr>
<td>&lt;100</td>
<td>Value less than 100</td>
</tr>
<tr>
<td>&gt;=1</td>
<td>Value greater than or equal to 1</td>
</tr>
<tr>
<td>&lt;&gt;&quot;FL&quot;</td>
<td>Not equal to (all states besides Florida)</td>
</tr>
<tr>
<td>Between 1 and 10</td>
<td>Numbers between 1 and 10</td>
</tr>
<tr>
<td>Is Null</td>
<td>Finds attributes with no value</td>
</tr>
<tr>
<td>Is Not Null</td>
<td>Finds all attributes that have a value</td>
</tr>
<tr>
<td>Like &quot;a*&quot;</td>
<td>All words beginning with &quot;a&quot;</td>
</tr>
<tr>
<td>&gt;0 And &lt;=10</td>
<td>All numbers greater than 0 and less than 10</td>
</tr>
<tr>
<td>&quot;Bob&quot; Or &quot;Jane&quot;</td>
<td>Values are Bob or Jane</td>
</tr>
</tbody>
</table>
Access 2007 Tutorial

B. Example - Select query to display all videos in the database with category of drama, sorted by distributor.

1. From the Ribbon or Menu Bar, click on the Create tab then select Query Design in the Other section.

2. In the Show Table window, add the tables you created in Part 1.

3. Note the asterisk (*) in the tables you created. This is a SQL special character that tells the system that you wish to include ALL the attributes in the table to appear in the output. You can also select individual attributes to print out. The order of the attributes you select corresponds to the order that those attributes will appear in the output.

   a. For example, if you wanted to write a SQL statement which would display all the attributes in the Category table, you would write the following:

   b. SELECT * FROM VideoCategory

   c. Access actually allows you to write SQL directly without using QBE (Query by Example).

   d. To write SQL, click on the drop down next to Design ->Results->View.
e. Select SQL View

f. The SQL statement that corresponds to the query you are writing in QBE will appear

```
SELECT
FROM VideoCategory;
```

4. Add the following attributes to the lower pane from each table:

<table>
<thead>
<tr>
<th>Table</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>VideoTitle</td>
<td>TitleID, VideoTitle, ReleaseDate, VideoDuration</td>
</tr>
<tr>
<td>VideoDistributor</td>
<td>DistributorName</td>
</tr>
<tr>
<td>VideoCategory</td>
<td>CategoryName</td>
</tr>
</tbody>
</table>

5. In the DistributorName column, select Ascending for the Sort property.

6. In the CategoryName column, enter “Sci-Fi” for the Criteria. Note, you don’t need to enter the word in quotes as Access will add them automatically.
7. Under the Results section on the Ribbon or Menu Bar, click the button to run the query.

8. The results should appear as follows:

<table>
<thead>
<tr>
<th>Title ID</th>
<th>Video Title</th>
<th>Release Date</th>
<th>Duration</th>
<th>Distributor</th>
<th>Category</th>
<th>VideoDistribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>Sea Trek</td>
<td>12/3/2000</td>
<td>190</td>
<td>Crazy Video</td>
<td>Sci-Fi</td>
<td>Crazy Video</td>
</tr>
</tbody>
</table>

III. **Parameter Queries**

A parameter query is a special type of query in which the user has the ability to limit the output to specified rows which correspond to values of rows stored in the data of the table. For instance, let’s say you have a student table which has information such as first name, last name, address, status (i.e. freshman, sophomore, junior, senior). You want the user to be able, on demand, limit the output based on the status without having to write a custom query each time. Essentially, you want the user to be able to pick a value from the status attribute (a parameter) and limit the output to just those rows that have that value in the attribute. A query parameter (often called just a parameter) is a placeholder for an actual value.

Although Access’ parameter query function, it is cursory at best. Essentially when the parameter query runs, a dialog box will appear prompting the user to enter a value while will limit the rows retrieved. You can design the query to prompt you for more than one piece of information; for example, you can design it to prompt you for two dates. Access can then retrieve all rows that fall between those two dates.

Parameter queries are handy when used as the basis for forms, reports, and data access pages. For example, you can create a monthly earnings report based on a parameter query. When you print the report, Access displays a dialog box asking for the month that you want the report to cover. You enter a month and Access prints the appropriate report.

A. Example of Parameter Query:

In this example, we are going to write a query that allows the user to show the video title, release date and category name. What makes this different than a typical select query is that the user can limit the output to a specific category which can be entered as a parameter at run time. This means that the user can create a custom query without having to rewrite the select query each time the query is to run.

1. Click on Create -> Other -> Query Design.

2. In the Show Table window, add the VideoCategory and VideoTitle tables.

3. Drag the following attributes to the bottom pane of the query: VideoTitle, ReleaseDate, and CategoryName.

4. In the criteria section of the Category Name column, enter the following: [Enter Category Name].
5. Click the run button and a dialog box with the text you entered in brackets will appear.

![Enter Parameter Value dialog box]

a. When you run the query, Access sees the bracketed parameter and prompts you to enter a value in the Enter Parameter Value dialog box. The value you enter is passed to the query as the parameter. It's as if you typed the value directly into the query design grid—but you didn't have to modify the query.

b. The text you supply within the brackets of the parameter becomes the prompt that you see in the Enter Parameter Value dialog box, so you should choose your phrase carefully and make sure it clearly indicates the information that needs to be entered. The phrase also serves as the name that Access uses to identify the parameter.

6. Enter in a value for which you would like to search. For this example, enter Action and click OK.

<table>
<thead>
<tr>
<th>Video Title</th>
<th>Release Date</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Freedom Day</td>
<td>1/12/2006</td>
<td>Action</td>
</tr>
<tr>
<td>Ex-Terminator</td>
<td>6/13/1999</td>
<td>Action</td>
</tr>
<tr>
<td>Buccaneers of the Caribbe</td>
<td>10/16/2006</td>
<td>Action</td>
</tr>
</tbody>
</table>

a. After you enter the value, Access processes the query, selects the matching data, and presents the results in a datasheet.

b. Remember the value you enter must match exactly to the value stored in the table (including that the case must be identical) or no rows will return.

c. If you press ENTER without supplying a value, Access displays an empty datasheet as you are essentially telling the database to return rows with a null category.
IV. **Crosstab Queries**

You use crosstab queries to calculate and restructure data for easier analysis of your data. Crosstab queries calculate a sum, average, count, or other type of total for data that is grouped by two types of information— one down the left side of the datasheet and another across the top. When creating a crosstab query, you must specify one or more Row Heading(s) options, one Column Heading option, and one Value option.

- **Row Heading**: This crosstab option is represented vertically in your dataset. Good candidates for this grouping are product types or other categories of data you want to aggregate. You can have multiple row heading columns, so multiple column aggregations are allowed.

- **Column Heading**: This crosstab option is represented horizontally in your dataset. Good candidates for this grouping are sales quarters or other categories of data for which you want only one grouping aggregation, because only one column heading is allowed in a crosstab query.

- **Value**: This crosstab option is the data that's typically summarized in your crosstab query. It's the product of cross-referencing your Row Heading(s) and your Column Heading aggregation.

A. **Examples**:

This example will create a crosstab query that will display the total number of movies that a distributor filmed in a specific category.

1. Select Create -> Other -> Query Design.

2. In the Show Table window, add the VideoCategory, VideoTitle and the VideoDistributor tables.

3. In order to change the query into a crosstab query, under the Query Type section on the Ribbon or Menu Bar, select Crosstab.
4. Double-click the VideoTitle attribute in the VideoTitle table to add the attribute to the lower pane. Select Count for the Total property and Value for the Crosstab property.

5. Double-click the DistributorName attribute in the VideoDistributor table to add the attribute to the lower pane. Select Group By for the Total property and Column Heading for the Crosstab property.

6. Double-click the CategoryName attribute in the VideoCategory table. Select Group By for the Total property and Row Heading for the Crosstab property.

7. Under the Results section on Ribbon or Menu Bar, click the Run button to run the query. The results should appear as follows based upon the data you entered earlier:
V. **Action Queries**

An action query is a query that makes changes to or moves many rows in just one operation. There are four types of action queries:

- Append
- Make table
- Update
- Delete.

A. **Append Query**

An append query adds a group of rows from one or more tables to the end of one or more tables. For example, suppose that you acquire some new customers and a database containing a table of information on those customers. To avoid typing all this information into your own database, you'd like to append it to your Customers table.

B. **Make Table Query**

A make-table query creates a new table from all or part of the data in one or more tables. Make-table queries are helpful for creating a table to export to other Microsoft Access databases (Microsoft Access database: A collection of data and objects (such as tables, queries, or forms) that is related to a particular topic or purpose. The Microsoft Jet database engine manages the data.) or a history table that contains old rows.

Let's consider the following scenario to explore the update and delete queries. Let's say we are working on a database for a hair stylist system (note – you do not have this database. I’m just providing an example). The customer table currently holds 29 rows and the structure looks like the following.

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gender</td>
<td>Text</td>
<td>Customers' gender - Male or Female</td>
</tr>
<tr>
<td>Customer_LName</td>
<td>Text</td>
<td>Customers' last name.</td>
</tr>
<tr>
<td>Customer_FName</td>
<td>Text</td>
<td>Customers' first name.</td>
</tr>
<tr>
<td>Customer_Add1</td>
<td>Text</td>
<td>First line of customers address.</td>
</tr>
<tr>
<td>Customer_Add2</td>
<td>Text</td>
<td>Second line of customers address.</td>
</tr>
<tr>
<td>Customer_City</td>
<td>Text</td>
<td>City where customer currently resides.</td>
</tr>
<tr>
<td>Customer_State</td>
<td>Number</td>
<td>State where the customer lives from a drop down list.</td>
</tr>
<tr>
<td>Customer_Zip_Code</td>
<td>Text</td>
<td>Zip code of the town where the customer lives.</td>
</tr>
<tr>
<td>Customer_Phone_Number</td>
<td>Text</td>
<td>Customers' phone number.</td>
</tr>
<tr>
<td>Customer_E_Mail</td>
<td>Text</td>
<td>Customers email address.</td>
</tr>
<tr>
<td>Preferred_Stylist</td>
<td>Text</td>
<td>If applicable allows user to enter a customers' preferred stylist.</td>
</tr>
<tr>
<td>Status</td>
<td>Text</td>
<td>Displays whether the customer's appointment frequency is regular, occasional, or rare</td>
</tr>
</tbody>
</table>

C. **Update Queries:**

An update query makes global changes to a group of rows in one or more tables. The update statement changes the values of single rows, groups of rows, or all the rows in a table.

If you have a update statement with no criteria clause you will update all the rows in the table. Accordingly, unless you truly want to update all the rows, consider a condition or filter.

1. **Update Query with Criteria:**

As you can see, the customer table has a gender attribute. The value of the field should be male or female. However, the current value of the field is a number (1 for male, 2 for female, and 3 for other). Let’s write an update query to change the values of the field for women from a 2 to female.
Access 2007 Tutorial

a. Select Create -> Query Design to display the show table dialog box

b. Select the table(s) from which the rows are to be update. In this example, I’ll select customer.

c. Click on the update query icon on the toolbar. You’ll note that the parameters in the lower pane of the query window change to limit the functionality to selection of the update criteria.

d. Click on the attributes upon which the rows to be updated will be selected. For this update query, I’ll click on gender as I only want to update rows from the customer table if the gender is a 2.

e. In the Update To field – enter the value to which you want the current values changed. In the Criteria field – enter the current value of the field.

f. Click the Run button

g. The message below returns indicating the number of rows which are to be updated

h. Repeat steps to update rows with a value of 1 in the gender field to “Male” and a value of 3 to “Other”.

2. Update Query without Criteria:

We have a status field in our customer table which was designed to describe how frequently the customer comes to the salon. The field is currently not used. We’d like to populate (add data) to this field. Presently, our customer table is populated only with regular customers so we will update all of the rows to have a value of Regular in the status field.

All of the steps are the same to write the query except the criteria field will be left blank.
D. Delete Queries:

A delete query permanently deletes selected rows from one or more tables. For example, you could use a delete query to remove products that are discontinued or for which there are no orders. With delete queries, you delete entire rows, not just selected fields within rows.

If you have a delete statement with no criteria clause you will delete all the rows in the table. Accordingly, unless you truly want to delete all the rows in the table, it is essential to consider the condition or filter.

Remember – a delete statement only affects the rows in a table - even if you delete all the rows, the table structure still exists. In other words, the table is still there but there will be no rows in the table.

The process to write a delete query is essentially the same as with an Update Query.

1. Delete Query with Criteria

Let's say the shop decides to market solely to men so they decide to remove all customers who are woman from the database.

a. Select Create -> Query Design to display the show table dialog box

b. Select the table(s) from which the rows are to be deleted. In this example, I'll select customer.

c. Click on the delete icon on the toolbar. You'll note that the parameters in the lower pane of the query window change to limit the functionality to selection of the delete criteria.

d. Click on the attributes upon which the rows to be updated will be selected. For this update query, I'll click on gender as I only want to update rows from the customer table if the customer is female.

e. In the Criteria field, enter the appropriate criteria. I'll enter “Female”

f. Click the Run button
2. Delete Query without Criteria

Let's say you decide to delete all the rows from the gender table:

VI. Aggregate Functions with Group By clause

A Group By clause specifies that you wish to perform some type of aggregations (sum, average, count, or other type of total for data that is grouped) but the grouping may not appear as a crosstab.

A. Example of Count and Group by Clause:

This example will show the total number of movies in a given category.

1. Select Create -> Other -> Query Design.
2. In the Show Table window, add the VideoCategory and VideoTitle tables. Close the show table window.

3. Under the Show/Hide section of the Design Ribbon tab, click the button to add the Total property to the lower pane.

4. Add the following attributes to the lower pane:

<table>
<thead>
<tr>
<th>Table</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>VideoCategory</td>
<td>CategoryName</td>
</tr>
<tr>
<td>VideoTitle</td>
<td>TitleID</td>
</tr>
</tbody>
</table>

5. In the CategoryName column of the lower pane, select Group By for the Total property.
6. In the TitleID column of the lower pane, select Count for the Total Property.
Access 2007 Tutorial

7. Click -> Results -> then the button to run the query. Your result should be similar to the image below:

<table>
<thead>
<tr>
<th>Category</th>
<th>CountOfTitle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Action</td>
<td>3</td>
</tr>
<tr>
<td>Drama</td>
<td>1</td>
</tr>
<tr>
<td>Sci-Fi</td>
<td>1</td>
</tr>
</tbody>
</table>

B. Example of a Query using the SUM function with the Group By clause: This example will show the video duration total in a given category.

1. Click Create -> Other -> Query Design.

2. In the Show Table window, add the VideoCategory and the VideoTitle tables. Close the Show Table Window.

3. In Design Tab -> Show/Hide -> click the button to add the Total property to the lower pane.

4. Add the following attributes to the lower pane:

<table>
<thead>
<tr>
<th>Table</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>VideoCategory</td>
<td>CategoryName</td>
</tr>
<tr>
<td>VideoTitle</td>
<td>VideoDuration</td>
</tr>
</tbody>
</table>

5. In the CategoryName column of the lower pane, select Group By for the Total property.
6. In the VideoDuration column of the lower pane, select Sum for the Total Property.
7. Under the Results section on Ribbon or Menu Bar, click the button to run the query. Your result should be similar the image below:

<table>
<thead>
<tr>
<th>Category</th>
<th>SumOfVideoDuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Action</td>
<td>459</td>
</tr>
<tr>
<td>Drama</td>
<td>300</td>
</tr>
<tr>
<td>Sci-Fi</td>
<td>190</td>
</tr>
</tbody>
</table>

C. Example of a Query using the MAX function with the Group By clause: This example will show the max release date in a given category.

1. Click Create ->Other -> Query Design.

2. In the Show Table window, add the VideoCategory, and the VideoTitle tables. Close the Show Table Window.

3. Select ->Design-> Show/Hide->click the button to add the Total property to the lower pane.
4. Add the following attributes to the lower pane:

<table>
<thead>
<tr>
<th>Table</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>VideoCategory</td>
<td>CategoryName</td>
</tr>
<tr>
<td>VideoTitle</td>
<td>ReleaseDate</td>
</tr>
</tbody>
</table>

5. In the CategoryName column of the lower pane, select Group By for the Total property.
6. In the ReleaseDate column of the lower pane, select Max for the Total Property.

7. Click the button to run the query. Your result should be similar to the image below:

```
<table>
<thead>
<tr>
<th>Category</th>
<th>MaxOfRelease</th>
</tr>
</thead>
<tbody>
<tr>
<td>Action</td>
<td>10/16/2006</td>
</tr>
<tr>
<td>Drama</td>
<td>11/20/1964</td>
</tr>
<tr>
<td>Sci-Fi</td>
<td>12/3/2000</td>
</tr>
</tbody>
</table>
```

Practice 3 - Create Queries:

Query the tables you created above based on the following criteria below. Examples of the output are provided

1. Create select query that will display all videos that have a category of Action, sort by VideoTitle. SAVE AS qrySelect
2. Create a crosstab query that will display the total number of movies that a distributor filmed in a specific category. SAVE AS qryCrosstab

<table>
<thead>
<tr>
<th>Distributor</th>
<th>Action</th>
<th>Drama</th>
<th>Sci-Fi</th>
</tr>
</thead>
<tbody>
<tr>
<td>Crazy Video</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>SBC Video</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DVDNow</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>East Park Media</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

3. Create a select query using the COUNT function with the Group by clause to display the total number of distributors in a given category. SAVE AS qryCount

<table>
<thead>
<tr>
<th>Distributor</th>
<th>CountOfCate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Crazy Video</td>
<td>2</td>
</tr>
<tr>
<td>SBC Video</td>
<td>1</td>
</tr>
<tr>
<td>DVDNow</td>
<td>1</td>
</tr>
<tr>
<td>East Park Media</td>
<td>1</td>
</tr>
</tbody>
</table>

4. Create a query using the MIN function with the Group By clause to display the minimum release date of a video for each category. SAVE AS qryMIN

<table>
<thead>
<tr>
<th>Category</th>
<th>MinOfRelease</th>
</tr>
</thead>
<tbody>
<tr>
<td>Action</td>
<td>6/13/1999</td>
</tr>
<tr>
<td>Drama</td>
<td>11/20/1964</td>
</tr>
<tr>
<td>Sci-Fi</td>
<td>12/3/2000</td>
</tr>
</tbody>
</table>

5. Create a select query that will display the total number of videos in database. SAVE AS qryTotal

<table>
<thead>
<tr>
<th>CountOfTitle</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
</tr>
</tbody>
</table>

6. Create a parameter query for that shows the Video Title, Distributor Name and Duration and allows the user to pass the distributor’s name as a parameter. SAVE AS QryParameter.
Chapter 9: Forms

I. Introduction to Forms

A user can enter data into a table directly through the datasheet view. However, this approach is not very user friendly. Accordingly, forms are used as an alternative way to enter data into a database table. Note: Every form must have a title pertaining to the topic that is meaningful. For instance, if the data entry form is to add, edit, or search for users, you may want to name the form “Manage Users”. Also, the form should include the company’s name and logo.

II. Create Form by Using Wizard

A. In the Ribbon, click Create tab -> Forms section -> More Forms button. In the drop-down box, select Form Wizard. The Form Wizard window opens.

B. From the Tables/Queries drop-down menu, select the table or query whose datasheet the form will modify.

C. Select the fields that will be included on the form by highlighting each one of the Available Fields window and clicking the single right arrow button > to move the field to the Selected Fields window.
   1. To move all of the fields to Select Fields, click the double right arrow button >>.
   2. If you make a mistake and would like to remove a field or all of the fields from the Selected Fields window, click the left arrow < or left double arrow << buttons.

D. After the proper fields have been selected, click the Next > button to move on to the next screen.
E. On the second screen, select the layout of the form.

- **Columnar** - A single record is displayed at one time with labels and form fields listed side-by-side in columns
- **Justified** - A single row is displayed with labels and form fields are listed across the screen
- **Tabular** - Multiple rows are listed on the page at a time with fields in columns and records in rows
- **Datasheet** - Multiple rows are displayed in Datasheet View
Access 2007 Tutorial

F. Click the **Next >** button to move on to the next screen.

G. Select a visual style for the form from the next set of options and click **Next >**.

H. On the final screen, name the form in the space provided. Select "Open the form to view or enter information" to open the form in Form View or "Modify the form's design" to open it in Design View. Click **Finish** to create the form.
Practice 4 - Create a Data Entry Form Using Form Wizard:

1. Create a form that will display all the fields in the Member table by using the wizard. Save the form and call it Members.
2. When your form is complete, it should look similar to picture below:

III. Create Form in Design View

To create a form without the wizard, follow these steps. This option provides more control and flexibility.

A. Click the Create tab on the Ribbon, and then under the Forms section, click the Form Design button.

B. Notice that a new tab called Design is created on the Ribbon. This tab contains tools that can be used to design your form.
C. The Field List pane shows all of the other tables in your database, grouped into categories. To view the Field List window, on the Design tab on the Ribbon, in the Tools section, click Add Existing Fields.

1. In the Field List pane, when you click the plus sign (+) next to a table name, you see a list of all the fields available in that table.

2. To add a field to your table, drag the field that you want from the Field List pane to the form.

D. Once you have added the fields to the pane, click the button on the Ribbon to view the form.

Practice 5 - Create a Data Entry Form in Design View:

1. Create a form that will display all the fields in the VideoTitles table by using the design view. Save the form and call it Video Titles.

2. When your form is complete, it should look like this:
IV. Adding Records Using a Form
   A. To add information into the form, type the data into the appropriate fields of the form.
   B. Press the Tab key to move from field to field
   C. Create a new row by clicking Tab after the last field of the last row. A new row can also be created at any time by clicking the New Record button at the bottom of the form window.
   D. Records are automatically saved as they are entered so no additional manual saving needs to be executed.

Practice 6 - Add Records to a Form:

1. Add the following row to the VideoTitles table using the form that you created in the last section. The system will automatically assign the Title ID as the primary key is an autonumber data type.

<table>
<thead>
<tr>
<th>Video Title</th>
<th>Release Date</th>
<th>Duration</th>
<th>Distributor</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spiderwoman</td>
<td>09/12/2002</td>
<td>100</td>
<td>East Park Media</td>
<td>Action</td>
</tr>
</tbody>
</table>

V. Editing Forms
   The follow points may be helpful when modifying forms in Design View.
   A. Grid lines:
      By default, a series of lines and dots underlay the form in Design View so form elements can be easily aligned. To toggle this feature on and off click the Arrange tab on the Ribbon, in the Show/Hide section, click the button.
   B. Snap to Grid:
      Click the Arrange tab on the Ribbon, in the Control Layout section, click the Snap to Grid button to align form objects with the grid to allow easy alignment of form objects or uncheck this feature to allow objects to float freely between the grid lines and dots.
C. Resizing Objects:
Form objects can be resized by clicking and dragging the handles on the edges and corners of the element with the mouse.

D. Change form object type:
To easily change the type of form object without having to create a new one, right click on the object with the mouse and select Change To and select an available object type from the list.

E. Label/object alignment:
Each form object and its corresponding label are bounded and will move together when either one is moved with the mouse. However, to change the position of the object and label in relation to each other (to move the label closer to a text box, for example), click and drag the large handle at the top, left corner of the object or label.

F. Tab order:
Alter the tab order of the objects on the form by selecting View->Tab Order. Click the gray box before the row you would like to change in the tab order, drag it to a new location, and release the mouse button.

G. Form Appearance:
You can change the background color of the form by clicking the button on the Ribbon’s Design -> Font section. Change the color of individual form objects by highlighting one and selecting a color from the button in the Font section of the Design tab on the Ribbon. The font and size, font effect, font alignment, border around each object, the border width, and a special effect can also be modified using the Design tab.

H. Page Header and Footer:
Headers and footers added to a form only appear when the form is printed. Click Arrange->Show/Hide->, the button to turn it off and on.
I. Page numbers:

Page numbers be added to sections by selecting the Design -> Controls -> $\rightarrow$ button. A date and time can be added by selecting the Design -> Control -> $\rightarrow$ button.

VI. Form Controls

Access provides a number of controls to use on forms. Some of the common form controls including lists, combo boxes, checkboxes, option groups, and command buttons are described below.

A. List and Combo Boxes

If there are small, finite number of values for a certain field on a form, using combo or list boxes may be a quicker and easier way of entering data. These two control types differ in the number of values they display. List values are all displayed while the combo box values are not displayed until the arrow button is clicked to open it as shown in these examples.

<table>
<thead>
<tr>
<th>List Box:</th>
<th>Combo Box:</th>
<th>When arrow button clicked, options appear as below</th>
</tr>
</thead>
<tbody>
<tr>
<td>List Box appears on the form with all of the options</td>
<td>Combo box appears as a drop down box on form</td>
<td></td>
</tr>
<tr>
<td>Cherry Hill</td>
<td></td>
<td>Cherry Hill</td>
</tr>
<tr>
<td>Philadelphia</td>
<td></td>
<td>Philadelphia</td>
</tr>
<tr>
<td>Philadelphia</td>
<td></td>
<td>Philadelphia</td>
</tr>
<tr>
<td>Trenton</td>
<td></td>
<td>Philadelphia</td>
</tr>
</tbody>
</table>

By using a combo or list box, the name of the academic building does not need to be typed for every row. Instead, it simply needs to be selected from the list. Follow these steps to add a list or combo box to a form:

1. Open a form in Design View.

2. Make sure the "Control Wizards" button is pressed in by clicking on the $\rightarrow$ button in the Controls section of the Design tab on the Ribbon.
3. Click the list or combo box tool button and draw the outline on the form. The combo box wizard dialog box will appear.

4. Depending on your choice in the first dialog box, the next options will vary. In this case, we will select the choice to look up values from a table or query. The following box will be displayed.

5. Select the table or query from which the values of the combo box will come.
6. Click **Next >** and choose fields from the table or query that was selected.
7. Click **Next >** to proceed.
8. On the next dialog box, you have the option to sort the options in the combo box or list box in ascending or descending order. Click **Next**.

9. On the next dialog box, set the width of the combo box by clicking and dragging the right edge of the column.

10. If you selected the primary key field, it is recommended that you check the **Hide Key Column**. Click **Next >** to proceed to the final screen.
Practice 7 - Create a Combo Box:

1. If you haven’t done so already, convert the foreign key State field in the Members form into a combo box. The combo box should display the states names in the States table.

B. Check Boxes and Option Boxes

Use check boxes and option buttons to display yes/no, true/false, or on/off values. Only one value from a group of option buttons can be selected while any or all values from a check box group can be chosen. Typically, these controls should be used when five or fewer options are available. Combo boxes or lists should be used for long lists of options. To add a checkbox or option group:

1. Click Design-> Controls ->Option Group tool and draw the area where the group will be placed on the form with the mouse. The option group wizard dialog box will appear.

2. On the first window, enter labels for the options and click the tab key to enter additional labels. Click Next > when finished typing labels.

3. On the next window, select a default value if there is any and click Next >.
Access 2007 Tutorial

4. Select values for the options and click **Next >**.

5. Save the values in a field

6. Choose the type and style of the option group and click **Next >**.
7. Type a caption (label) for the option group and click **Finish**. The option group will appear on the form and the data will be stored in the appropriate attribute in a table.

C. **Command Buttons**

In this example, a command button beside each row is used to open another form. This is a very common approach to add buttons to forms that allow the user to add a row, save a row, delete a row, cancel an operation, and lookup a row. We’ll try it on the Member Form

1. Open the form in Design View and click on **Design -> Controls -> Control Wizard**.
2. Click **Design- > Controls- > Command button icon** and draw the button on the form. The Command Button Wizard will then appear.
3. On the first dialog window, action categories are displayed in the left list while the right list displays the actions in each category. Select an action for the command button and click **Next >**.

   ![Command Button Wizard](image)

   - a. For our example, we want to add a button to add a member.
   - b. Click on Record operations and then Add New Record

4. On the next dialog window, you have the choice to either have a picture of the action on the button or just text. For our example, we’ll add the text **Add Member**.

   ![Command Button Wizard](image)
5. On the final window, give the button a name, such as cmdAddMember and click Finish.

6. Change to form view and click on the button. You will be presented with a blank form to enter a new member.

Practice 8 - Create Buttons on a Form:

1. Create buttons on the Members form to find a row, undo a row, and close the form.
VII. **Sub Forms for 1:M Relationships**

A. **What is a Sub Form?**

A sub form is a form that is placed in a parent form, called the main form. Sub forms are particularly useful to display data from tables and queries that have one-to-many relationships. For example, in the sample below, data on the main form is drawn from an item information table while the subform contains all of the orders for that item. The item row is the "one" part of this one-to-many relationship while the orders are the "many" side of the relationship since many orders can be placed for the one item.

There are three methods to create sub forms. Each assumes that the data tables and/or queries have already been created.

B. **Create a Form and Sub Form at Once**

Use this method if neither form has already been created. A main form and sub form can be created automatically using the form wizard if table relationships are set properly or if a query involving multiple tables is selected. For example, a relationship can be set between a table containing customer information and one listing customer orders so the orders for each customer are displayed together using a main form and sub form. Follow these steps to create a sub form within a form:

1. Click the **Create** tab on the **Ribbon**, then in the **Forms** section, click the **More Forms** button. Finally, in the drop-down box, select **Form Wizard**. The Form Wizard window opens.

![Form Wizard](image)
2. From the **Tables/Queries** drop-down menu, select the first table or query from which the main form will display its data.

3. Select the fields that should appear on the form by highlighting the field names in the **Available Fields** list on the left and clicking the single arrow > button or click the double arrows >> to choose all of the fields.

4. From the same window, select another table or query from the **Tables/Queries** drop-down menu and choose the fields that should appear on the form. Click **Next** to continue after all fields have been selected.

5. Choose an arrangement for the forms by selecting **form with subform(s)** if the forms should appear on the same page or **Linked forms** if there are many controls on the main form and a sub form will not fit. Click **Next** to proceed to the next page of options.
6. Select a tabular or datasheet layout (when you click on one of the radio buttons, an example of the form layout will appear to the left) and click **Next**.

7. Select a style for the form (when you click a style, an example of the style will appear to the left) and click **Next**.

8. Enter the names for the main form and subform. Click **Finish** to create the forms. New rows can be added to either tables or queries at once by using the new combination form.

C. Create a Form and Sub Form at Once in Design View

If the main form or both forms already exist, the Subform Wizard can be used to combine the forms. Follow these steps to use the Subform Wizard:

1. Open the main form in **Design View** and make sure the "Control Wizards" button is pressed in by clicking on the button in the **Controls** section of the **Design** tab on the **Ribbon**.

2. Click the **Subform/Subreport** icon in the **Controls** section of the **Design** tab on the **Ribbon** and draw the outline of the subform on the main form. The Subform Wizard dialog box will appear when the mouse button is released.

3. If the subform has not been created yet, select "Use existing Tables and Queries". Otherwise, select the existing form that will become the subform. Click **Next** to continue.
4. The next dialog window will display the relationship between the main and the subform assumed by Access. Select one of these relationships or define your own and click Next.

5. On the final dialog box, enter the name of the subform and click Finish.

D. Drag-and-Drop Method

Use this method to create subforms from two forms that already exist. Make sure that the table relationships have already been set before proceeding with these steps.

1. Open the main form in Design View
2. Drag the form icon beside the name of the subform onto the detail section of the main form design.
Practice 9 – Create a Form with a Subform:

1. Create a form with a subform at once using the VideoDistributor and VideoTitles tables.
2. When prompted how to view your data, select by VideoDistributor.
3. Save the form and call it VideoDistributor for the main form and VideoTitles Subform for the subform.
4. When your form is complete, it should look similar to picture below:
VIII. **Sub Forms for M:M Relationships**

The process for creating sub-forms with M:M relationships to other tables is basically the same as 1:M. The only difference is that you must decide what you want to be the main form.

For instance, in the video database, the main form could be movies and the subforms could be members and rentals. This would then show for each movie all of the members who rented the movie and the rental date. Alternatively, you might prefer to see for each member all of the movies rented. In this case, the member would be the main form and the movies and rentals would the subforms. Either approach is fine – it is completely determined by the needs of the organization. In some cases, you might want to create two forms: one with the Member as the main form and one with the Movie as the main form.

There is one rule to remember with M:M relationships. The main form must always be one of the entity tables, NOT the associative entity. If you think about it, this makes sense. For instance, you wouldn't want the rental to be the main form.

Refer to Subforms for 1:M relationships to create the relationships for both foreign keys.

A. **Steps to create M:M Forms**

1. Click the **Create** tab on the **Ribbon**, then in the **Forms** section, click the **More Forms** button. Finally, in the drop-down box, select **Form Wizard**. The Form Wizard window opens.

2. Click on the drop down menu next to display the tables and queries.
Access 2007 Tutorial

3. Select the Members table from the drop down list.
   a. The attributes for that table will appear below.
   b. Click >> to move all the fields into the Selected Fields section.
   c. Repeat for the Rentals and the VideoCopies table.
   d. All of the attributes for all three tables will appear in the Selected Fields panel. Click Next.

4. The wizard will ask how the data is to be viewed – this relates to which of the two entities will be the main form. Select the VideoCopies table and click Next.

5. The next screens are the same as in the 1:M form. On the next window, you will be prompted to select a layout. For this example, choose the Datasheet layout.

6. Next select the style for the form.

7. On the next window, you will be prompted to enter titles of the both the main form and the subform. You will also be asked to open the form or to modify the form in design view.
Access 2007 Tutorial

For this example, the title of the main form will be VideoCopies and the subform Rentals Subform. Click Finish to open the form.

![Form Wizard](image)

8. Below is a screenshot of the results:

![Results Screenshot](image)

Practice 10 - Create a M:N Form

1. Create a form for M:N relationship using the VideoDistributor, VideoTitles and VideoCategory tables.
2. When prompted how to view your data, select by VideoCategory.
3. Save the form and call it VideoCategory for the main form and VideoTitles Subform 2 for the subform.
4. When your form is complete, the form should display VideoTitles and the VideoDistributor’s fields in the subform and look similar to picture below:
IX. **Multiple-Page Forms Using Tabs**

A tab control is the easiest and most effective way to create a form with multiple pages or multiple embedded subforms. With a tab control, you can build separate pages into one control. To switch pages, you click one of the tabs on the control. Create a tab control by following these steps:

A. Click the **Tab Control** icon in the **Controls** section of the **Design** tab on the **Ribbon** and draw the control on the form.

B. Add new controls to each tab page the same way that controls are added to regular form pages and click the tabs to change pages.

C. Existing form controls cannot be added to the tab page by dragging and dropping. Instead, right-click on the control and select **Cut** from the shortcut menu. Then right-click on the tab control and select **Paste**. The controls can then be repositioned on the tab control.

1. Add new tabs or delete tabs:
   a. Right-click in the tab area.
Access 2007 Tutorial

b. Choose Insert Page or Delete Page from the shortcut menu.

2. Reorder tabs:
   a. Right-click on the tab control.
   b. Select Page Order.

3. Rename tabs:
   a. Double-click on a tab
   b. Select Name property under Other tab.

Practice 11 – Create a Form with Tab Control:

1. Create a new form called Tabs. The new form should include a tab control with two tabs (one for Members and the other for Video Titles)
2. Using the Member form and the VideoTitles form that you created earlier, attach these forms to their respective tabs.
3. When your form is complete, the form should look similar to picture below:
Chapter 10: Reports

I. Introduction to Reports

Reports will organize and group the information in a table or query and provide a way to print the data in a database. Note: Every report should have a title pertaining to the topic that is meaningful. For instance, if the report is to display all employees in the Human Resources Department, you may want to name the report “Human Resource Department Employees”. Also, the report should include the company’s name and logo.

II. Types of Reports

A. Parameterized Reports - A parameterized report is a report that uses input values to complete report processing. Parameters are typically used to complete a query that selects data for the report. For instance, on online banking system may have a report that accept different parameters from the user including the bank account type (savings vs. checking) and the date range of transactions. It then returns a report of all the transactions from that account for the specified time period. The user does not typically get to decide what attributes will print on the report as this is predetermined. But the user does get to limit the output to the values entered into the parameters (the date range and the type of account).

B. Ad hoc Reports - An ad hoc (as needed) report is a customized query. The user has full control over the attributes that appear on the report, the sort order of the report, and any criteria for the report. For instance, in our online banking example, the user might want to create a custom report that allows him to print only the transaction date, merchant, and dollar amount of the transaction for all transactions made between a certain time period and only of a particular transaction type. This is a fully customized report which may only be needed once.

C. Canned Reports – a canned report is a report that runs when a user pushes a button. It does not allow the user the ability to customize the report in any manner. A typical report might be a list of all transactions for an account. The button has code attached to it which determines what information will appear in the report including the attributes to print, the sort order, and any predetermined criteria.

III. Create Report Using the Wizard

Create a report using Access' wizard by following these steps:

A. Click the Create tab on the Ribbon, then in the Reports section, click the Report Wizard button. The Report Wizard window opens.

B. Select the information source for the report by selecting a table or query from the Tables/Queries drop-down menu.

C. Select the fields that should be displayed in the report by transferring them from the Available Fields menu to the Selected Fields window using the single right arrow button.
to move fields one at a time or the double arrow button >> to move all of the fields at once. Click the **Next >** button to move to the next screen.

### D. Select fields from the list that the rows should be grouped by and click the right arrow button > to add those fields to the diagram. For instance, let’s say you want to see rows group by the member’s last name. Use the **Priority** buttons to change the order of the grouped fields if more than one field is selected. Click **Next >** to continue.

### E. If the rows should be sorted, identify a sort order here. Select the first field by which the rows should be sorted and click the A-Z sort button to choose from ascending or descending order. Click **Next >** to continue.

### F. Select a layout and page orientation for the report and click **Next >**.
G. Select a color and graphics style for the report and click **Next >**

H. On the final screen, name the report and select to open it in either Print Preview or Design View mode. Click the **Finish** button to create the report.

---

**Practice 12 - Create a Report:**

1. Create a report called Members that will display a listing of all members as well as each member’s address, phone, birth date, and date the member joined.
2. The report should display the last name in ascending order and landscape as the page layout.

---

**IV. Create Report in Design View**

A. To create a report from scratch (without a wizard), select Create-> **Reports ->Report Design** button.

B. You will be presented with a blank grid. Design the report in much the same way you would create a form. For example:
Access 2007 Tutorial

C. To view the Field List window, click Design -> Tools -> Add Existing Fields. In the Field List pane, when you click the plus sign (+) next to a table name, you see a list of all the fields available in that table. To add a field to your table, simply drag the field that you want from the Field List pane to the form.

D. Use the handles on the elements to resize them, move them to different locations, and modify the look of the report by using options on the formatting toolbar.

E. Click the View button at the top, left corner of the screen to preview the report.

V. Printing Reports

A. Select Page -> Page Layout -> Page Setup button to modify the page margins, size, orientation, and column setup.

B. After all changes have been made, print the report by selecting the button on the Ribbon and select the Print -> Print button.
Chapter 11: Switchboard

I. Introduction to Switchboards

A. A switchboard is essentially a Microsoft Access form that allows you to facilitate navigation or perform tasks within your database application. This form is basically a customized menu that contains user-defined commands; using either buttons, labels, images or hyperlinks, that invoke actions that will automatically carry out tasks for you such as opening other forms, running queries or printing reports.

B. The form will typically contain various command (buttons), which users can then click on to carry out the pre-defined actions that have been associated with these commands.

C. Using a switchboard form in your Microsoft Access database application allows you to tie together all of the other objects that you have created for your database users from a single form, and removes the need for the users to have to openly navigate from within the database window. You can also use this as the interface for your application, removing the chances of your users tampering with any of the objects that lie behind the scene of your application.

D. Using buttons on your switchboard form can also replace the many possible steps that it would take the user to navigate around the objects in the database. To simply open a form in the database, the user may need to switch to the database window, choose the Forms tab and open the form from there. Using a command on the switchboard interface, the user now only has to click one button to carry out the same action.

E. Most applications are organized into switchboards with submenus that have a button for data entry, administrative functions (manage users, manage lookup tables, etc.), and reports.

F. Example: Below is a snapshot of a main switchboard that has four buttons: Administrative Settings, Customer Forms, Reports and Exit.
Access 2007 Tutorial

If the user selects the second option, Customer Forms, he will move to a submenu of forms pertaining to a customer as shown in the next picture. While the submenu screenshot is functional and very pretty, the user does not actually know that he is on the Customer Form menu. All submenus should have the company name and logo as well as the name of the submenu (i.e. Customer Form Menu)

II. Create a Switchboard
   A. To create a switchboard, click the Database Tools->Database Tools->Switchboard Manager. For this example we are going to add one button onto our switchboard which will allow us to update (edit) a member’s information.

   B. On the next screen, click Yes to create a new switchboard.
   C. When the Switchboard Manager window appears, click on the Edit button.

   D. Next, you will need to add buttons to the switchboard. Click on the New button.
E. The Edit Switchboard Item dialog appears and you will need to complete three sections.

1. Text:
   
   The Text field allows you to enter the text that will show as the label next to the button you are creating on the switchboard. In this example, we are entering the text that will appear next to the button. We'll call this form Edit Member. Information.

2. Command:

   The command option lets you select from a number of typical actions to be performed from a switchboard. Since we want to edit the member's information, we'll use the Open Form in Edit Mode option. The most common options are:

   a. Open Form in Add Mode: Opens a form so that the user can add a new row to the table. This option does not allow the user to modify the information that exists in a row.

   b. Open Form in Edit Mode: Opens a form so the user can edit information in a row in the table.

   c. Open Report: Allows the user to bring up a report. If the report is based on a query, the query will run when the button is selected and the report will appear with the updated data.
Access 2007 Tutorial

3. Form:

   The form option lets you select which form or report the command should be affected. The drop down will display all the forms and reports you have created in the database. In this case, we’ll select the Member form.

4. Select OK and then Close until the switchboard dialog box closes. Now you can click on the switchboard in your navigation pane and it should look similar to the image below. You can switch to design view to customize this form with images, logos, colors, etc.

III. Create a Switchboard Within a Switchboard

   This example assumes that there you have not created a switchboard for your database.

   A. To create a switchboard, click the Database Tools tab on the Ribbon, then in the Database Tools section, click Switchboard Manager.

   B. On the next screen, click Yes to create a new switchboard.
C. The Main Switchboard has already been created for you. To create another switchboard, click the New button.

D. On the next screen, name the new switchboard Data Entry Forms and click OK.

E. On the Switchboard Manager window, select the Main Switchboard and click the Edit button.
F. On the Edit Switchboard Page window, click the New button.

G. On the Edit Switchboard Item window
   1. Enter Edit Forms for the Text
   2. Select Go to Switchboard for the Command
   3. Select Data Forms for the Switchboard, Click OK and then Close twice.

IV. Display Switchboard When Database Opens
   A. To automatically display the Switchboard when the database is opened, follow these steps:
      B. From the Ribbon, click the button and then select Access Options. The Access Options window displays.
      C. Select the Current Database option.
      D. Under Application Settings, next to the Display Form label, select the Switchboard option from the drop down menu and click OK.
Practice 13 – Create a Switchboard:

1. Create a switchboard that has 3 buttons. The first button, called Queries, takes the user to another switchboard that includes buttons for all of the queries created so far. The next button, called Forms, takes the user to the M:N form created. The last button, called Exit, closes the application.

2. Make the switchboard open automatically when the user opens the database.
Chapter 12: Keyboard Shortcuts Cuts

Keyboard shortcuts can save time and the effort of constantly switching from the keyboard to the mouse to execute simple commands. Print this list of Access keyboard shortcuts and keep it by your computer for a quick reference. Note: A plus sign indicates that the keys need to be pressed at the same time.

<table>
<thead>
<tr>
<th>Action</th>
<th>Keystroke</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Document actions</strong></td>
<td></td>
</tr>
<tr>
<td>Open existing database</td>
<td>CTRL+0</td>
</tr>
<tr>
<td>Open a new database</td>
<td>CTRL+N</td>
</tr>
<tr>
<td>Save</td>
<td>CTRL+S</td>
</tr>
<tr>
<td>Print</td>
<td>CTRL+P</td>
</tr>
<tr>
<td>Undo data changes made in current field</td>
<td>ESC</td>
</tr>
<tr>
<td>Display database window</td>
<td>F11</td>
</tr>
<tr>
<td>Find and Replace</td>
<td>CTRL+F</td>
</tr>
<tr>
<td>Copy</td>
<td>CTRL+C</td>
</tr>
<tr>
<td>Cut</td>
<td>CTRL+X</td>
</tr>
<tr>
<td>Paste</td>
<td>CTRL+V</td>
</tr>
<tr>
<td>Undo</td>
<td>CTRL+Z</td>
</tr>
<tr>
<td><strong>Editing</strong></td>
<td></td>
</tr>
<tr>
<td>Toggle between editing and navigation mode</td>
<td>F2</td>
</tr>
<tr>
<td>Open window for editing large content fields</td>
<td>SHIFT+F2</td>
</tr>
<tr>
<td>Switch from current field to current row</td>
<td>ESC</td>
</tr>
<tr>
<td>Move to next field in row</td>
<td>TAB</td>
</tr>
<tr>
<td><strong>Other</strong></td>
<td></td>
</tr>
<tr>
<td>Insert line break in a memo field</td>
<td>CTRL+ENTER</td>
</tr>
<tr>
<td>Insert current date</td>
<td>CTRL+;</td>
</tr>
<tr>
<td>Insert current time</td>
<td>CTRL++;</td>
</tr>
<tr>
<td>Copy data from previous row</td>
<td>CTRL+'</td>
</tr>
<tr>
<td>Add a row</td>
<td>CTRL++</td>
</tr>
<tr>
<td>Delete a row</td>
<td>CTRL+-</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Action</th>
<th>Keystroke</th>
</tr>
</thead>
<tbody>
<tr>
<td>Formatting NOT ACCESS</td>
<td></td>
</tr>
<tr>
<td>Select all</td>
<td>CTRL+A</td>
</tr>
<tr>
<td>Copy</td>
<td>CTRL+C</td>
</tr>
<tr>
<td>Cut</td>
<td>CTRL+X</td>
</tr>
<tr>
<td>Paste</td>
<td>CTRL+V</td>
</tr>
<tr>
<td>Undo</td>
<td>CTRL+Z</td>
</tr>
<tr>
<td>Redo</td>
<td>CTRL+Y</td>
</tr>
<tr>
<td>Bold</td>
<td>CTRL+B</td>
</tr>
<tr>
<td>Italics</td>
<td>CTRL+I</td>
</tr>
<tr>
<td>Left justified</td>
<td>CTRL+L</td>
</tr>
<tr>
<td>Center justified</td>
<td>CTRL+E</td>
</tr>
<tr>
<td>Right justified</td>
<td>CTRL+R</td>
</tr>
<tr>
<td>Decrease indent</td>
<td>CTRL+SHIFT+M</td>
</tr>
<tr>
<td>Increase indent</td>
<td>CTRL+M</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Action</th>
<th>Keystroke</th>
</tr>
</thead>
<tbody>
<tr>
<td>Editing NOT ACCESS</td>
<td></td>
</tr>
<tr>
<td>Find</td>
<td>CTRL+F</td>
</tr>
<tr>
<td>Replace</td>
<td>CTRL+H</td>
</tr>
<tr>
<td>Insert hyperlink</td>
<td>CTRL+K</td>
</tr>
<tr>
<td>Spell checker</td>
<td>F7</td>
</tr>
<tr>
<td>Macros</td>
<td>ALT+F8</td>
</tr>
</tbody>
</table>

| Moving through a datasheet            |             |
| Next field                            | TAB         |
| Previous field                        | SHIFT+TAB   |
| First field of row                    | HOME        |
| Last field of row                     | END         |
| Next row                              | DOWN ARROW  |
| Previous row                          | UP ARROW    |
| First field of first row              | CTRL+HOME   |
| Last field of last row                | CTRL+END   |
Chapter 13: Pivot Tables

I. Introduction to Pivot Tables

You can download a very nice introduction to pivot tables at the following site: http://www.nacba.net/PDF_FILES/PivotTableIntroduction.pdf Here’s a really short description as provided by Wikipedia:

A. What’s a Pivot Table:

A pivot table is a data summarization tool found in data visualization programs such as spreadsheets (e.g. Microsoft Excel). Among other functions, they can automatically sort, count, and total the data stored in one table or spreadsheet and create a second table displaying the summarized data. Pivot tables are also useful for quickly creating cross tabs. The user sets up and changes the summary's structure by dragging and dropping fields graphically. This "rotation" or pivoting of the summary table gives the concept its name.

For typical data entry and storage, data is usually flat. Flat means that it consists of only columns and rows, such as in the following example:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Region</td>
<td>Gender</td>
<td>Style</td>
<td>Ship Date</td>
<td>Units</td>
<td>Price</td>
<td>Cost</td>
</tr>
<tr>
<td>2</td>
<td>East</td>
<td>Boy</td>
<td>Tee</td>
<td>1/31/2005</td>
<td>12</td>
<td>11.04</td>
<td>10.42</td>
</tr>
<tr>
<td>3</td>
<td>East</td>
<td>Boy</td>
<td>Golf</td>
<td>1/31/2005</td>
<td>12</td>
<td>11.96</td>
<td>11.74</td>
</tr>
<tr>
<td>4</td>
<td>East</td>
<td>Boy</td>
<td>Fancy</td>
<td>1/31/2005</td>
<td>12</td>
<td>11.27</td>
<td>10.96</td>
</tr>
<tr>
<td>5</td>
<td>East</td>
<td>Girl</td>
<td>Tee</td>
<td>1/31/2005</td>
<td>10</td>
<td>13.74</td>
<td>13.33</td>
</tr>
<tr>
<td>6</td>
<td>East</td>
<td>Girl</td>
<td>Golf</td>
<td>1/31/2005</td>
<td>10</td>
<td>12.12</td>
<td>11.96</td>
</tr>
<tr>
<td>7</td>
<td>East</td>
<td>Girl</td>
<td>Fancy</td>
<td>1/31/2005</td>
<td>10</td>
<td>13.42</td>
<td>13.29</td>
</tr>
<tr>
<td>8</td>
<td>West</td>
<td>Boy</td>
<td>Tee</td>
<td>1/31/2005</td>
<td>11</td>
<td>11.44</td>
<td>10.94</td>
</tr>
<tr>
<td>9</td>
<td>West</td>
<td>Boy</td>
<td>Golf</td>
<td>1/31/2005</td>
<td>11</td>
<td>12.83</td>
<td>11.73</td>
</tr>
<tr>
<td>10</td>
<td>West</td>
<td>Boy</td>
<td>Fancy</td>
<td>1/31/2005</td>
<td>11</td>
<td>12.06</td>
<td>11.51</td>
</tr>
<tr>
<td>11</td>
<td>West</td>
<td>Girl</td>
<td>Tee</td>
<td>1/31/2005</td>
<td>15</td>
<td>13.42</td>
<td>13.29</td>
</tr>
<tr>
<td>12</td>
<td>West</td>
<td>Girl</td>
<td>Golf</td>
<td>1/31/2005</td>
<td>15</td>
<td>11.48</td>
<td>10.67</td>
</tr>
</tbody>
</table>

While there is a lot of information stored in such data, it is very difficult to gather the information you want out of it. A pivot table can help you quickly summarize the flat data, giving it depth, and get the information you want. The usage of a pivot table is extremely broad and depends on the situation. The first question to ask is, "what am I looking for?". In the example here, let's ask "How many Units did we sell in each Region for every Ship Date?":

<table>
<thead>
<tr>
<th>Sum of Units</th>
<th>Ship Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>East</td>
<td>66</td>
</tr>
<tr>
<td>North</td>
<td>96</td>
</tr>
<tr>
<td>South</td>
<td>123</td>
</tr>
<tr>
<td>West</td>
<td>78</td>
</tr>
<tr>
<td>Grand Total</td>
<td>363</td>
</tr>
</tbody>
</table>

A pivot table usually consists of row, column, and data (or fact) fields. In this case, the row is Region, the column is Ship Date, and the data we would like to see is Units. These fields were dragged onto the pivot table from a list of available fields. Pivot tables also allow several kinds of aggregations including: sum, average, standard deviation, count, etc. In this case, we wanted to see the total number of units shipped, so we used a sum aggregation.
B. How a pivot table works:

Using the example above, it will find all distinct rows for Region. In this case, they are: North, South, East, West. Furthermore, it will find all distinct rows for Ship Date. Based on the aggregation type, sum, it will summarize the fact, and display them in a multidimensional chart. In the example above, the first data point is 66. This number was obtained by finding all rows where both Region was East and Ship Date was 1/31/2005, and adding the Units of that collection of rows together to get a final result.4

II. Introduction to ODBC

ODBC is an: Abbreviation of Open DataBase Connectivity, a standard database access method developed by Microsoft Corporation. The goal of ODBC is to make it possible to access any data from any application, regardless of which database management system (DBMS) is handling the data. ODBC manages this by inserting a middle layer, called a database driver, between an application and the DBMS. The purpose of this layer is to translate the application's data queries into commands that the DBMS understands. For this to work, both the application and the DBMS must be ODBC-compliant -- that is, the application must be capable of issuing ODBC commands and the DBMS must be capable of responding to them.5

What does this mean for you? All large database management systems including Oracle and SQL Server support ODBC. This means you can connect any application with an ODBC driver to the compliant DBMS. So, if you would like to attach to an Oracle DB using Access, you can query multiple databases of different types simultaneously provided they have common data.

This also means that you only need Access to connect to various ODBC client database management systems. So if you are at home but you don't have SQL Server or Oracle loaded onto your machine, you can still access them via Access!

III. Connecting Oracle to Access via an ODBC Connection

I've created a number of files and tables in a zipped file that go along with this section of the book. Please download this file from blackboard as you work on these concepts.

A. Create the ODBC Connection:
   1. Navigate to your Control Panel (In XP, Start → Control Panel)
   2. Select Administrative Tools -> Data Source (ODBC).
   3. Under the User DSN tab, click "Add." Scroll through the listbox and select either Microsoft ODBC for Oracle or Oracle ODBC Driver.
   4. In the “Data Source Name” field, type in Temple Oracle Server.
   5. In the Service Name or Server, type the host string of the server (foxora).
   6. Click OK. - The ODBC Connection to the Oracle server at Temple has now been created

B. Connect to the tables on the server from Access
   1. Click the “Office Button” -> Open

---

2. Create a new database called loan.
3. Open the loan database.
4. At the “Open” Dialog box, locate the “Files of type” dropdown. Select the file type ODBC Databases().
5. At the “Select Data Source” Dialog box, click the “Machine Data Source” tab
6. The data source you created in the ODBC Connection will appear
7. Click the connection and Click OK
8. Oracle ODBC Driver Connect login will appear as shown below with the Service name populated as “foxora”.

![Microsoft ODBC for Oracle Connect](image)

9. In the Username field, enter your schema name for Oracle (the same one you use in SQL plus)
10. In the Password field, enter your schema name for Oracle (the same one you use in SQL plus) and click OK
11. Add tables through the ODBC Connection and call the database “Loan”.
12. A list of all the tables in your personal account and will appear. Notice that the tables you will be selecting should follow this syntax B250102.Bank_Customer (replace B250102 with your own Oracle username.)
13. You will note that each of the tables connected to the oracle server have an arrow and a world icon. This indicates that the files are linked to an external source. Whenever you want to use tables from the Oracle server, you will see the login dialog box below. Enter your Oracle username and password. The server name will already be established based on the ODBC set up you created.
14. Select all the following tables. You can multi-select tables by holding the control key down while clicking on the tables below and then click OK. Save the database as Loan.
   - Bank_Customer_Pivot
   - State_Pivot
   - Bank_Account_Pivot
   - Bank_Account_Assignment_Pivot
   - Branch_Pivot
   - Loan_Type_Pivot

C. Adding Tables after ODBC Connection established
1. If you need to add a file after the initial Access database is created, open the Access database
2. Click External Data tab
3. Under Import → Click More → Click ODBC Database
4. Follow steps from above for importing tables.

D. Relinking tables after database tables have been created

1. If you open a database you have created with links to an external site and receive a message that the connection is unavailable, try the following to reestablish the links:

2. Select Database Tools tab → Linked Table Manager

3. The linked table manager appears as below.

4. Select the links to be updated (you can click on select all to pick them all)
   a. If you select a linked file that has been downloaded (such as the excel file called Rep), the wizard will prompt you to enter the new address of the file.
   b. If the file has not moved, deselect this file

5. Click on OK

6. A login dialog box will appear. Enter your oracle username and password.

7. The following message will appear and you should now be able to access your files.

IV. Background to Connecting Access to Other Data Formats:

Access was designed to function as a conduit which allows interaction between other data formats. As such, you can import and/or link data from a variety of other data sources noted below. You can find more information and instructions pertaining to working with the below data formats through the help function within Access.

- Access Files
- Text Files
- Spreadsheets
- Web Page
- Windows SharePoint Services
- SQL or another ODBC data source (described above)
Access 2007 Tutorial

- Mail programs including Microsoft Exchange and Outlook
- Another database
- Data Access Page
- XML

V. Connect Excel to Access

You can share data between Access and Excel in many ways. You can copy data from an open worksheet and paste it into an Access datasheet, import a worksheet into an Access database, or simply load an Access datasheet into Excel using the Analyze it with Excel command.

If your goal is to store some or all data in one or more Excel worksheets in Access, you will typically import the contents of the worksheet into Access database. When you import data, Access creates a copy of the data in a new or existing table, without altering the source Excel file. Of note, any changes made to an imported table will NOT be reflected in the source data. The changes will affect only the imported table.

If you do not wish to maintain a copy of the data in your Access database, you can link to the Excel worksheet. Linking allows you connect to data in Excel without importing it, so that you can view the data in Access. When you link to a worksheet, Access creates a new table that is linked to the source cells. You can update the data in the worksheet when you are working in Excel and your changes will be shown when you view the linked table in Access. Of note:

- You cannot create a link to an Access database from within Excel.
- You cannot link Excel data to an existing table in the database. When you create a link, Access creates a new table, often referred to as a linked table. The table shows the data in the source worksheet or named range, but it doesn’t actually store the data in the database.
- A database can have multiple linked tables.
- Any change that you make to the data in Excel will be automatically reflected in the linked (source) table.
- When you open an Excel worksheet in Access (In the File Open dialog box, change the Files of Type list box to Microsoft Office Excel Files, and select the file you want), Access creates a blank database, and automatically starts the Link Spreadsheet Wizard.

For the following exercise, we are going to link and import an Excel workbook to our loan database and import a text file.

A. Linking an Excel Workbook to Access:

For this practice, we’ll link the Reps worksheet of the Pivot Table workbook to Access:

1. In Access 2007, open the loan database.
2. Click -> External Data-> Import ->Excel icon
3. In the Get External Data dialog box specify the source of the data by browsing to the where you stored “Pivot table.xls,” and then double-click its icon.
4. In the Get External Data dialog box select Link to the data source by creating a link table and click OK.
5. In the Link Spreadsheet Wizard dialog:
   a. Verify “Rep” is highlighted in the wizard indicating that Rep is the worksheet to be imported and click Next
Access 2007 Tutorial

b. Verify the Check the box that says “First Row Contains Column Headings” is selected and Click Next
c. Verify the linked table is called “Rep”, click Finish
d. When the wizard is completed, you will receive a message indicating the process was successful. Click OK

B. Importing an Excel Workbook to Access

For this practice, we'll import the Transaction worksheet of the Pivot Table workbook to Access:
1. Repeat the process above to import the transaction worksheet of the pivot table.xls workbook – make sure to select the option to import select Import the source data into a new table in the current database in the Get External Data dialog box
2. Make Transaction ID the primary key
3. Save the table as Transactions
4. Don’t save the import steps.
5. Change the Customer Number data type to Text with a size of 40

VI. Connecting Text Files to Access

In this section, we will explore importing text files of different types: delimited and fixed width. When you import data from another source, the recipient (in this case Access) needs to know what information to put into which column. In a delimited file, a character (called a delimiter) is used to indicate when the values of one field end and the next column begins. In a fixed width file, all the values in the column are a set width so if the value is always 5 characters long, the characters at the sixth position would be stored in the next column. As you might guess, delimited files are typically used with files with attributes of a variable length (such as last name) while fixed width files would be used for files containing data that is always the same length (such as a phone number or a zip code). If the file has both variable length and fixed width fields, a delimited file is routinely used.

Access allows you to import files of both delimited and fixed width type through a wizard. We will see examples of both in this exercise. When you import a file, you can do the following functions. In this exercise, we will import tables.

• append data to a table that already exists in Access
• import a new table which has column headings
• import a new table without column headings.

C. Import a Fixed Width File without Row Heading.

In this section, we will see how you can import a local data file (called Loan.txt) which is a fixed width file. In the next fixed width text file but you could export a wide range of file types to a table on a server. In this example, we will import the file and it has no row headings. We will add row headings during the import process.
1. With the loans database open, on the External Data tab click on Text File
2. In the Get External Data dialog box specify the source of the data by browsing to where you stored “Loan.txt” and double-click its icon.
3. Select Import the source data into a new table in the current
4. Follow the directions in the Import Text Wizard dialog boxes:
   a. Click on the Fixed width radio button and Next
   b. In the next screen, set the field parameters as follows. You can move between the different fields by clicking on the column
   c. Select Loan_ID as the primary key
   d. Call the new table “loan” and click Finish
   e. Don’t save the import steps and click Close.

5. The loan table should be populated with the data from the file

D. Import a Delimited Width File with Row Heading

In this section, we will see how you can import a local data file which is a delimited file. Remember a delimeter is a character in a data file which indicates where the values of a column end and the values of the next column begin. You could also import a fixed width text file as we saw earlier.

Some data files have qualifiers. For instance, text information may be identified as text with quotes around the characters. When the text is imported, and the field is identified as a text field, the quotes are removed and only the characters are stored. For our practice, we’ll import the “Loan_Assignment.txt” delimited file as a new table with row headings

1. With the loans database open, on the External Data tab click on Text File
2. In the Get External Data dialog box specify the source of the data by browsing to the “Loan_Assignment.txt,” and then double-click its icon.
3. Select Import the source data into a new table in the current database
4. Follow the directions in the Import Text Wizard dialog boxes:
   a. Click on the Delimited radio button and Next
   b. Select commas as the delimeter
   c. Select “ in the Text Qualifier drop down
   d. Click on First Row Contains Row Heading and Click Next
   e. Verify that the following attributes are of the proper data types:
      i. loan_assignment and loan_id -> long integer
      ii. Customer ->text
   f. Choose Loan_Assignment_Id as the primary key
   g. Name the new table loan_assignment and click Finish
   h. If presented the option to save the import steps, click Close.
5. The loan_assignment table should be populated with the data from the space delimited file.

6. Change the Customer data type to text

VII. Creating Queries across Access and Oracle

Now that we have a database that connects various file formats, we can use this database to write queries. It will appear as though the database houses all the information and that all of the files were actually Access tables but you know that some of these are linked files, some imported, and all of various formats. You are beginning to see the beauty of ODBC!!!

If you need a refresher on creating select queries, refer to the Access Tutorial in this book.

A. Create relationships between tables

1. In the relationships window, add the tables below.

2. Create a relationship between the attributes below. Note some are linked Oracle tables, some are Access tables, and some are linked Excel files. Note, you should enforce referential integrity, cascade on update and restrict on delete wherever possible. However, you cannot enforce referential integrity on linked tables.

<table>
<thead>
<tr>
<th>Bank Account Pivot.Branch ID</th>
<th>Branch Pivot.Branch ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bank_Account_Pivot.State_Opened</td>
<td>State_Pivot.State_ID</td>
</tr>
<tr>
<td>Branch Pivot.Branch_State</td>
<td>State_Pivot.State_ID</td>
</tr>
<tr>
<td>Bank_Customer_Pivot.Cust_ID</td>
<td>Loan_Assignment.Customer</td>
</tr>
<tr>
<td>Loan_Assignment.Loan_ID</td>
<td>Loan.Loan_ID</td>
</tr>
<tr>
<td>Loan.Loan_Type</td>
<td>Loan_Type_Pivot.Loan_Type_ID</td>
</tr>
<tr>
<td>Bank_Customer_Pivot.Cust_ID</td>
<td>Transactions.Customer Number</td>
</tr>
<tr>
<td>Bank_Account_Pivot.Bank_Account_ID</td>
<td>Bank_Account_Assignment_Pivot.Bank_Account_ID</td>
</tr>
<tr>
<td>Bank_Account_Assignment_Pivot.Bank_Customer_Id</td>
<td>Bank_Customer_Pivot.Cust_ID</td>
</tr>
</tbody>
</table>
Access 2007 Tutorial

B. Create Select Query to Join Tables:

Create a select query called Customer_Loans_Accounts that displays the following attributes:
The table should include 7 tables and will return 43 rows in your query if you build it properly.

- Customer first name
- Customer last name
- Bank Account Type
- Branch Name
- Loan Amount
- Interest Rate
- Loan Type Description

Practice 14 - Create a Select Query with Linked and Imported Data Sources:

1. Create a select query called Transactions_Customer_Rep that displays the following attributes. This should include 3 tables.

- Customer last name
- Customer first name
- Customer Zip
- Transaction Id
- Week number
- Rep name
- Rep Status

VIII. Crosstab Queries:

You use crosstab queries to calculate and restructure data for easier analysis of your data. Crosstab queries calculate a sum, average, count, or other type of total for data that is grouped by two types of information— one down the left side of the datasheet and another across the top. When creating a crosstab query, you must specify one or more Row Heading(s) options, one Column Heading option, and one Value option.

- Row Heading: This crosstab option is represented vertically in your dataset. Good candidates for this grouping are product types or other categories of data you want to aggregate. You can have multiple row heading columns, so multiple column aggregations are allowed.

- Column Heading: This crosstab option is represented horizontally in your dataset. Good candidates for this grouping are sales quarters or other categories of data for which you want only one grouping aggregation, because only one column heading is allowed in a crosstab query.

- Value: This crosstab option is the data that's typically summarized in your crosstab query. It is the product of cross-referencing your Row Heading(s) and your Column Heading aggregation.

Example: Let's say we want to see a report that shows the number of each account type by branch of the bank. If you look at the customer_loans_accounts query, it is difficult to discern which branch is processing the most loans and of what type. We will create a cross tab that will display by branch, the number of loans of each type and the total number of loans for each bank. The final output will appear as below. As you will note, with the cross tab you can quickly see
that Ambler is processing the most loans. In the next steps, we'll learn how to create this cross tab.

<table>
<thead>
<tr>
<th>LOAN_TYPE_DESCRIPTION</th>
<th>Abington</th>
<th>Ambler</th>
<th>Cherry Hill</th>
<th>Feasterville</th>
<th>Huntington</th>
</tr>
</thead>
<tbody>
<tr>
<td>15 Year Fixed Mortgage</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>30 Year Fixed Mortgage</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>30 Year Variable Rate Mortgage</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>Auto Loan</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Home Equity</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>Line of Credit</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A. Click on Create -> Query Design
B. In the Show Table window, click on the Queries tab and select the customer_loans_account query you made earlier and close the dialog box.

C. In the Ribbon, click on Design -> Query Type -> Crosstab to change the query into a crosstab query.

D. To create subtotals by branch_name:
   1. Make branch_name the column heading by clicking the branch_name in the table and dragging it into the lower pane.
   2. Change the crosstab property to column heading.
   3. Since we want to group the output by branch_name, we will leave this option in the total property.

E. For the loan_type_description, we want to see the actual description as well as well as how many of that loan type was processed.
   1. Add loan_type_description to the lower pane and make it a row heading. This means that there will be a new row added for each different loan_type_description.
2. Again, we’ll leave the total property as group by which indicates that we are grouping the output by the value of the loan_type_description.

F. We don’t just want to see a listing of the branches and the different loan_types they process. Rather, we want to see how many of each (COUNT) they processed. The real value of the cross tab is the ability to summarize or aggregate data.

1. We’ll add loan_type_description to the lower pane again but this time we’ll change the crosstab property to value because we want to perform some type of aggregation to a value. All cross tab queries must perform something to a value such as count, sum, average, etc.

2. In the Total property, select Count to count the number of loans (rows) of that type. The query design will appear as below

G. Click the button on the menu bar to run the query. The following output appears

H. To answer the question, Which branch processes the most loans, we can add subtotals to the report by clicking on Home -> Records -> Totals button which adds a row at the bottom for subtotals
I. Click in a cell and a drop down box will appear.
   1. Click on the down arrow and you will see a number of choices of ways you can aggregate the data. For this example select Sum as we want to sum the counts in the rows below.

J. The final report appears as follows. It is clear that Huntingdon Valley processed the most loans.

Practice 15: Create crosstab queries in Access

1. One of the customer service representatives in the Loan division of the bank would like to know for each customer, what the total amount of the loans by loan type. This should appear as one report with the customer’s last name displayed revealing which customer has which type of loan and the amount of the loan.
   a. Create a crosstab query called Customer_Loans_Accounts_Crosstab from the Customer_Loans_Accounts query (HINT: When selecting the total amount of a loan use the “SUM” function instead of “Count”). Paste a snapshot of the query design and the output into your deliverable document.
   b. Which customer has the largest 30 year fixed mortgage and what is the amount of that mortgage?
   c. Which type of loan has the largest total loans and what is the total amount?

2. Management would also like to understand the number of transaction performed by each rep on a weekly basis.
Access 2007 Tutorial

a. Using the Transactions_Customer_Rep as input, create a cross tab called rep_transaction_week_crosstab that shows the total number of transaction processed per week for each rep.
b. The cross table should show the rep’s name as the column heading.
c. How many transactions did Brian process in Week 35? Enter your answer into your output

IX. Crosstab queries in Excel 2007

Excel’s crosstab functionality is extensive. You can apply styles, perform drill down function, click, drag and move all the attributes, include page settings, and much more. In this section, we’ll try a few.

A. Export the Customer_Loans_Accounts and query

Do not export the crosstab query – we want to work off of the original query with all the rows.

1. In the External Data tab click on Excel under Export
2. Select the location where you want the excel file to be placed.

B. Create a crosstab report in Excel of the data in Customer_Loans_Accounts that shows the total amount of the loans by branch name(column) and loan type description (row)

1. Open the Customer_Loans_Accounts excel worksheet that you just created
2. Highlight the area with data
3. Select Insert tab -> Pivot Table and accept the pivot table dialog options.
4. The pivot table wizard appears. Drag the attributes from the field list into the areas below

5. Format the report as follows:

a. Highlight the area with data
b. Click on Home -> Styles -> Cell Styles
c. In the fourth panel called Number Format select Currency[0] which indicates that Excel should format numbers with dollar signs, commas, and round to the whole dollar
Practice 16 – Create a Pivot Table and Chart in Excel:

1. Which branch had the largest overall loans and what was the amount?
2. Double click on 30 Year Variable Rate Mortgage and select Customer Last Name from the Show Detail dialog box. This will allow you to drill down on the detail summarized in the cross tab report.
3. What was Cramden’s overall loan amount?
4. Click on the – (minus) sign next to the 30 year variable amount to summarize the findings.
5. Create a Chart from pivot tables Click on Insert -> Chart -> Column and then select a chart style you like from the 3-D options
Section 3: TOAD

Chapter 14: Introduction and Loading TOAD

I. What is TOAD

TOAD (Tool for Oracle Application Developers) is an application that performs a variety of functions on relational database management systems. In the case of class, we will use the tool as the front end to our Oracle databases. It will allow us to use a graphical user interface to interact with Oracle instead of writing code at the command line using SQL*Plus. Accordingly, you can use TOAD to perform such functions as create a table, alter a table, add data to a table, and create an ERD.

II. Loading and Configuring TOAD:

A. Insert the CD with the TOAD Image onto your workstation.
B. If the files are compressed/ziped:
   1. Right click on the folder and select the option to “Extract Files”
   2. Change the destination folder to My Documents (or somewhere where you will be comfortable navigating to within your computer)
   3. Once the files have been extracted, double click on the “setup - Double Shield Icon” as shown below:

   ![Setup Icon]

C. Follow the prompts to go through the set up wizard:
   1. Click Next at the first screen and second screens
   2. At the Installation Type screen, select the first option → Local Toad Install (Full)
   3. On the next prompt for Destination Directory for Client leave the default settings and click Next
   4. At the Unix screen deselect the first option as shown below and click Next
5. Click Next on the screen that prompts you to begin the TOAD install
6. At the Knowledge Xpert for PL/SQL Installation select the “No” option and click on the Continue with Installation button
7. Select No for the SQL Tuning Installation and continue with installation
8. Select No to run the SQL Tuning Wizard
9. The wizard should now have completed installing TOAD onto your machine
D. Configure TOAD for Oracle
1. Go to Start --> All Programs --> Quest Software --> Toad for Oracle --> Toad for Oracle 8.6 OR click on the TOAD icon on your workstation.
2. Enter the following Authorization Key which is posted on blackboard
3. You will be entered to configure TOAD to link to your Oracle account your Oracle username and password.
   a. Enter the Database Name (FOXORA)
   b. Enter your Oracle Username
   c. Enter your Oracle Password
4. Click on the Make this the TOAD Default home check box
5. You should see green check marks next to the SQLNET Editor and TNSNames Editor buttons
TOAD Tutorial

6. Click on the Connect Button

III. Introduction to TOAD Interface

Columns tab:
- Allows user to modify the table design.
- Displays a list of the attributes, data types, etc.
- You may add, delete, sort the attributes, or see comments of the table design using the buttons: 🔯 🥇 🍀

Constraints tab:
- Allows user to add, delete, or modify constraints

Data tab:
- Allows user to add, delete, or modify data in the table

Referential tab:
- Shows all the parents and children of the table (the tables referenced to and from this table)
Chapter 15: Tables in TOAD

I. Creating a table

In lieu of creating a table through SQL+, you can create a table in TOAD.

A. In the Table tab, click on the New table icon

B. Adding attributes to the table

1. Enter the name you'd like to create for this table

2. Click the Add COL button to create a new attribute
II. Adding constraints to a table

1. To add a constraint, click on the constraint tab.

2. Click on the Add Constraint Button.

3. The Add Constraint Dialog box appears and you have a choice of constraint type.

3. When you add a new column, you have the ability to add the field properties for the attribute including the data type, size, precision (for a number), whether it is required (not null), and the default value.

4. Click in the field property to select the values from a drop down list.

5. Here is a snapshot of the House table. It has 6 attributes and I’ve added the field properties.
TOAD Tutorial

A. Add a check constraint:
1. Click on the check constraint radio button
2. A dialog box will appear to allow you to type in the constraint

<table>
<thead>
<tr>
<th>Constraint Text</th>
</tr>
</thead>
<tbody>
<tr>
<td>City in (Philadelphia, 'New York', 'Newark')</td>
</tr>
</tbody>
</table>

3. Click on the OK button when the text is complete.

B. Add a primary key constraint:

   1. Double click on the attribute in the table columns list to be the primary key
   2. The attribute will appear on the right hand column as a constrained column
   3. Click the OK button

C. Add a foreign key constraint:

   I want to add make the state attribute a foreign key to the stateid field in the state table. First I need to tell TOAD which attribute is the foreign key in the House table and then I need to identify which table is referenced by the foreign key and the referenced attribute (in this case State is the table and Stateid is the attribute).

   1. Double click on the attribute in the table columns list which is a foreign to an attribute in another table. Click the OK button
   2. The attribute will appear on the right hand column as a constrained column
D. View the constraints on a table

1. Click on the schema browser icon to display a listing of the objects in your schema.
2. Click on the table with the constraints to be checked.
3. On the right hand side, click on the constraints tab.
4. A listing of the constraints on the table appears.
5. The icons on the constraint tab allow you to add, enable, disable, rename and drop a constraint from this dialog box.

3. Next you need to tell Oracle what attribute in which table the foreign key is associated. To do this, click on the Referenced Table tab.
4. In the table name drop down, select the referenced table.
5. Click on the attribute in the table columns list which is a foreign to an attribute in the referenced table.
6. The attribute will appear on the right hand column as a constrained column.
7. Click the OK button.
Chapter 16: Scripts in TOAD

I. Documenting Table Creation in Toad

TOAD allows the user to create a script of the SQL commands generated during the table creation process

A. Create the table as described earlier.
B. Populate (enter the data) into the table.
C. Click on the tables tab
D. Select the table for which the script is to be created.
E. Click on the first button -> Create script.

F. On the first tab (Table Options), select the following options:

G. In the second tab (Script Options), select the following options as indicated below to include the drop statement command:
H. In the third tab (output), click on the clipboard option so the output goes to the clipboard.

I. To generate the script:
   1. Click Execute
   2. Create a new Word document.
   3. Show the contents of your clipboard by clicking on Edit -> Office Clipboard
   4. Select the contents of the clipboard from TOAD
   5. Paste the contents of the clipboard into the Word doc.
   6. Continue for the remainder of the table and contents.
II. **Loading a Script into TOAD**

Use this function if you have a script that you would like to load into Oracle through TOAD.

A. Download the script to be loaded into SQL onto your workstation
B. Click on Database -> Import -> Source Files and the screen below will appear

<table>
<thead>
<tr>
<th>Add</th>
<th>Remove</th>
<th>Load in Editor</th>
<th>Parse files</th>
<th>Execute</th>
<th>Cancel</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIlename</td>
<td>Object Type</td>
<td>Object Name</td>
<td>Status</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

C. Click Add and the File Explorer will appear
D. Navigate to the location where you filed the script and click Insert
E. Click onto the file and highlight the file

F. Click the Load in Editor Button
G. Click the Execute button
Chapter 17: Other Functions in TOAD

I. Using TOAD to write SQL statements
   A. Click on the SQL Editor button
   B. Type the SQL statements in the upper part of the split screen and click the Execute Statement Button. The results will return in the lower pane of the window.
   C. To show a listing of all the tables and attributes in a table, show the object palette
      1. From the SQL file menu select View -> Object Palette
      2. On the right side of the SQL screen, the listing will appear. When you click on a table in the upper pane, the attributes for the table will appear in the lower pane.
D. Copy the SQL statement in the top window and paste into a Word Document
   1. Make sure cursor is in SQL statement pane
   2. To copy: You can either use Ctrl C OR Highlight the statement and on the File Menu, Select Edit -> Copy
   3. To paste: You can either use Ctrl V OR From File Menu Edit -> Paste

E. Copy the output of the query
   1. Make sure cursor is in window pane
   2. From TOAD file menu, select Grid -> Save As
   3. The Save Grid Contents dialog box appears and the following options are selected:
4. In Word, make sure the clipboard is visible (Edit -> Office Clipboard)
5. Place the cursor below the SQL statement where you want the output to appear
6. Click on the copied output on the clipboard and it will be pasted into the Word document

II. Using TOAD to Diagram and Modify a database

A. Click on the schema browser button

B. A list of all the tables in your schema will appear on the left side of the screen.

C. Click on the desired table on the left side of the screen and a tab dialog will appear on the right side of the screen.

D. To create an ERD of the database, click on the Show ER diagram button
E. The create ER Diagram dialog will appear with a number of options.

1. Select which options you wish to display on the diagram
2. Click OK

3. The ERD will draw and the relationships will appear based on the constraints applied
4. You have the option to save the diagram or you can recreate it the next time you need it.

5. To return to the tables, click on the schema browser button.

III. Exit Toad

A. Select File -> Exit
Chapter 18: Welcome to SQL

I. Introduction to SQL

As described in Wikipedia:

“Structured Query Language...is a database computer language designed for the retrieval and management of data in relational database management systems (RDBMS), database schema creation and modification, and database object access control management.

SQL is a standard interactive and programming language for querying and modifying data and managing databases.... The core of SQL is formed by a command language that allows the retrieval, insertion, updating, and deletion of data, and performing management and administrative functions....”

II. Introduction to SQL*Plus

To work with an Oracle database, the user accesses the Oracle database engine located on a server. The user can communicate with an Oracle database using SQL*Plus, a command line client which allows the user to pass SQL commands to the database engine for execution. The user can create, access and maintain data structures such as tables, sequences, indexes, etc.

A. Start SQL*Plus

1. From the Start button on the Windows taskbar select Programs| Oracle| OraHome| Application Development| SQL Plus.

2. A log on dialog box appears asking to enter User Name, Password and Host String as below:

a. Your username and password are posted to gradebook. The hoststring is "foxora" (the name of the Oracle db server we use here at Temple).

3. Once you are logged in, the SQL Command Prompt SQL> is displayed.

---

6 http://en.wikipedia.org/wiki/SQL
B. Change password
   1. Enter “password” at the SQL> prompt type.
   2. At the Password prompt, enter your old password.
   3. At the confirm your new password prompt, enter your new password again.
   4. A message ‘password changed’ is displayed confirming the change of password.
   5. Make sure to record your new password - it CANNOT be retrieved.

C. Exit from SQL*Plus:
   1. Enter “exit” at SQL> prompt.

D. Get Help
   1. Enter “help” at the command prompt followed by the name of the command. For example: SQL>HELP ACCEPT.
   2. If you having difficulty with an error message in SQL, try searching on the Internet. Most error messages are well documented, including the fix.

E. Before starting:
   To ensure that you have all the necessary data entered correctly, a database has been developed for you. Download the script which has been posted to your student portal. This script includes drop table commands which will remove any tables created in the past with the same table names and creates those to be used in these exercises. To run the script, open the file in an editor (i.e. Notepad). Copy a piece of the script from Notepad (about 10 lines of code or so) and paste the piece of code at the command line. Continue until all of the code has been copied.

   A snapshot of the ERD is provided below to assist you in navigating through the database.
F. Working with SQL*Plus:

1. Using a Text Editor:
   SQL*Plus is a command line utility so all the commands must be typed at a SQL>prompt. This makes it difficult to make changes if you have typed something incorrectly. It is STRONGLY recommended that type your SQL statements into a text editor (such as Notepad). You can then copy (CTRL-C) the text from the text editor and paste (CTRL-V) at the SQL prompt. This also serves as a convenient to save your work and come back to use the text at a later time.

2. Typing Commands:
   At the SQL prompt, you can begin typing any SQL command. Upon hitting the Enter button, the SQL prompt will change to line number prompts. If you press Enter again, it will proceed to the next line.

3. Case Sensitivity:
   a. SQL is case insensitive with regards to SQL keywords. Accordingly, refer to a table name or attribute name or use any keyword (i.e. select, from, where, create, alter) in upper case, lower case, or mixed case, and SQL will accept it. Similarly, if you had a table name that was created in all capital letters, you reference it in lower case and SQL will still recognize it.
   b. SQL is case sensitive with regards to data. For instance, if you are entering a customer's first name as mixed case, you must use the same mixed case to query it. So if the customer's name is entered as “Bruce”, you will return no rows if you try to look at all of the customers with a first name of BRUCE.
SQL Tutorial

c. Some developers use conventions in terms of case. It is common to note that developers place all SQL keywords (i.e. select, update, where, from) in upper case and all user defined words (table names, attribute names, etc.) in lower case. For the most part, you will note this convention is used in the SQL examples in this book.

4. Spacing

SQL is a very forgiving language. With the exception of table names and attribute names, you can pretty much space your code any way you want.

G. Oracle Data Types: Here are a few of the data types that are available in Oracle.

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Data Type Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAR(n)</td>
<td>A CHAR column is a fixed length column and can contain any printable characters. Fixed length means that if the data entered into the CHAR field is less than length of the field then the field is padded to the left with spaces. The maximum length of the CHAR column is 200. Example: a state abbreviation would be a CHAR(2) since it is always 2 characters long.</td>
</tr>
<tr>
<td>VARCHAR2(n)</td>
<td>A VARCHAR2 column is a variable length column with a fixed length. If the length of the data is less than the maximum length of the field, then the field is not padded with spaces to the right. The maximum length of the column is 2000. Example: a customer’s first name might be a VARCHAR2(35) since the length of a customer’s first name is variable. Accordingly, so we should choose a length that would be reasonably large. For instance, 35 characters is sufficiently large for the average first name.</td>
</tr>
<tr>
<td>NUMBER</td>
<td>Integer and real values occupy up to 40 digits.</td>
</tr>
<tr>
<td>INTEGER</td>
<td>Same as number, but no decimals.</td>
</tr>
<tr>
<td>INTEGER (n)</td>
<td>Same as INTEGER but occupy n (where n = the maximum number of digits).</td>
</tr>
<tr>
<td>NUMBER(n,d)</td>
<td>A NUMBER column can contain a number, with or without a decimal point and a sign, and can have from 1 to 105 decimal digits of which only 38 digits are significant.</td>
</tr>
<tr>
<td>DATE</td>
<td>A DATE column may contain a date and time between the 1st of January 4712 BC to the 31st of December 4712 AD. The standard date format is DD-MMM-YY as in 01-JAN-99. If you enter data in any other format you will have to map it or use input masks.</td>
</tr>
<tr>
<td>LONG</td>
<td>A LONG column can contain any printable character and can be up to 2 Gigabytes in size.</td>
</tr>
<tr>
<td>RAW</td>
<td>A RAW column can contain data in any form, including binary.</td>
</tr>
</tbody>
</table>

III. DDL vs. DML

SQL has 2 components – the DDL(Data Definition Language) and the DML(Data Manipulation Language).

A. The DDL (Data Definition Language) deals with the management of objects themselves. DDL includes functions to allow the user to create a table, change the structure of a table, and delete a table. We will review the following statements:

- CREATE TABLE - create a new database table
- DROP TABLE – delete a database table
- ALTER TABLE - alter (changes) a database table
SQL Tutorial

B. The DML (Data Manipulation Language) deals with the data in the tables. DML allows the user to add rows to tables, query the data in a table, and change the information in a row. We will review the following statements

- INSERT INTO - insert new data into a database table
- SELECT – extract (retrieve) data from a database table
- UPDATE - update data in a database table
- DELETE - delete data from a database table
- COMMIT – permanently saves the changes to a database
- ROLLBACK – undo the uncommitted changes to a database

C. Qualifying column notation

1. When you refer to a column in a SQL statement, you specify the table first followed by a period and then the column name. This is known as qualifying the column name and uses tablename.attribute name notation.

   For instance, if there is an attribute in the student table called first_name, the attribute is called as student.first_name.

2. You can also call an attribute by just its column name (excluding the table name first) IF that attribute name is unique in the tables included in the query.

   a. For instance, if you are writing a query and the only table in the query is the student table. If you want the student’s first name to appear on the output, you can just call first_name.

   b. If, however, if you included two tables in a query (student and faculty) and both had an attribute called first name, you would need to specify the table name first before the attribute as SQL wouldn’t know which attribute you were referencing. Accordingly, if you wanted to call the first name in the student table, you’d have to write student.first_name.

3. It is always acceptable (and probably preferable) to use the full qualifying column notation. It just means more typing.
Chapter 19: DDL – Data Definition Language

I. Create Table

All the data in an Oracle database is stored in the form of 2 dimensional matrix called a table. A table consists of rows (also referred to as records) and columns (also known as fields/attributes/parameters). A table can be viewed as a container of data.

Each table has a structure. At a minimum, the table structure is defined by a table name and specific information about the attributes (columns) of the table including name, size and data type.

A. Format for Create Table:

1. The general format of the CREATE TABLE command appears below. Phrases in square brackets are optional, i.e. in this case the CONSTRAINT definitions are optional. You will learn about data declaration and constraint declaration as we go along. CONSTRAINTS can be also be defined separately after you have created a table.

   Syntax to create a table:
   CREATE TABLE <tablename>
   (<fieldname> <data declaration> …
   [CONSTRAINT <integrity constraint declaration>…
   );

2. Let’s say we want to add a bank_customer table. In this table we want to add the following attributes shown on the left. The syntax to create the table is shown on the right.

   CREATE TABLE bank_customer
   (Customer_ID NUMBER(5),
    Customer_LastName VARCHAR2(30),
    Customer_FirstName VARCHAR2(30),
    Add1 VARCHAR2(30),
    City VARCHAR2(25),
    State CHAR(2),
    Zip CHAR(5)
   );

3. To create the table, we would write the statements noted above. The CREATE TABLE statement must start with the keywords “CREATE TABLE” followed by the table name. Of note, SQL will not allow spaces between words in a table name or attribute name. If you want to have a table or attribute name that is more than one word long, separate the words with underscores as shown above.

4. When you add an attribute you need to define its data type and size.

5. Each attribute added must be followed by a comma to indicate to SQL that there are additional attributes to be added.

6. The last attribute is NOT followed by a comma (since there are no more attributes to be added).
II. Drop Table

A. To remove a table from an Oracle database we drop it. This command deletes the table from the database.

You will often see a drop table statement appear before a create table command. As you know, each table in a database must have a unique name. If you are replacing a table with another with the same name, you must drop the existing table before you can recreate it. You will often see a drop table statement before the create table statement in a script.

B. Drop Table Syntax:

```
DROP TABLE <tablename>;  DROP TABLE manager;
```

C. Drop Table Cascade Constraints

If there are any foreign keys in other tables referring to the primary key of the table being dropped, the statement will generate an error message and the table will not be dropped. So what is a foreign key constraint? We'll learn more about foreign key constraints after we learn more about the drop statement. What is happening here is that the database is telling us that the table is related to another table and that by dropping one row which is related to another, you may create a delete anomaly.

If you cascade constraints, Oracle drops all referential integrity constraints that refer to primary and foreign keys in the dropped table and then drops the table from the database.

```
DROP TABLE <tablename> CASCADE CONSTRAINTS;  DROP TABLE manager CASCADE CONSTRAINTS;
```

III. Alter Table

The ALTER table statement is used to modify the structure of a table once it has been created. For instance, you’d use an alter table statement to add a new column to a table or to change a column’s definition.

NOTE: the alter command is different from an update command. Update changes (updates) the DATA. Alter changes the table structure. We will learn more about update later.

A. Restrictions on Alter table:

One key restriction of the ALTER TABLE statement involves adding a column with a not null constraint. As we’ll learn later, a not null constraint indicates that an attribute must be populated (have data entered into it) and cannot be left blank. The database will not allow a not null column to be added to an existing table since we would automatically have null values for all rows that already exist in the table. Thus it would make it impossible for the constraint to be enforced by the DBMS. We'll explore null constraints further in a little bit.

B. Syntax:

```
ALTER TABLE <tablename>  
ADD/MODIFY….;
```

1. If you are adding something to a table (i.e. a new constraint or attribute), you will use the ADD keyword

```
Let's say we want to track a customer's date of birth and this is a newly added attribute to the table
ALTER TABLE customer ADD (customer_dob DATE);
```

2. If you are modifying something which is already in the table structure (changing a data type or size), you will use the MODIFY keyword.

```
```
IV. Describe

The describe command allows you to show the attribute names and data types of a table as noted in the snapshot below.

Of note, the describe command does not show constraints with the exception of the Null constraint. However, you can view constraints on a table using TOAD.

A. Syntax:

1. Describe <tablename>

2. You can also use the shortcut Desc as in Desc <tablename>.

```
SQL> describe bank_customer;
Name       | Null? | Type
-----------|-------|------
CUSTOMER_ID| NOT   | NUMBER(5)
CUSTOMER_LNAME| NOT   | VARCHAR2(100)
CUSTOMER_FNAME| NOT   | VARCHAR2(100)
ADDR1       | NOT   | VARCHAR2(256)
CITY        | NOT   | VARCHAR2(256)
STATE       | NOT   | CHAR(2)
ZIP         | NOT   | CHAR(5)
CUSTOMER_MN| NOT   | CHAR(1)

DESCRIBE bank_customer;
OR
DESC bank_customer;
```
Chapter 20: Constraints (DDL continued)

I. Introduction to Integrity Constraints

In addition to the data content, a relational database consists of integrity constraints, a set of conditions that must be met or satisfied by the data content at all times. There are a few basic types of constraints. The concept of an integrity constraint is discussed in Section 1 of this book.

A. Integrity Constraint Syntax

The general syntax for specifying constraints is as follows. Many developers give constraints constraint names. While this is often the preferred naming convention, it is not required.

```
[constraint <constraint_name>]  
  unique (<column> {, <column>}) |  
  not null (<column> {, <column>}) |  
  primary key (<column> {, <column>}) |  
  foreign key (<column> {, <column>})  
    references <table_name> [(<column> {, <column>})]  
    [on delete cascade]
```

II. Primary Key Constraints

A. Primary Key rules:

1. A primary key is used to uniquely identify each row and the values of its attributes in a table.
2. If a column is defined as primary key, it cannot contain duplicate values and cannot be null.
3. A primary key can be a single attribute or a combination of more than one attribute (if a primary key is made of two or more attributes, it is called a composite key).
4. You can add a primary key to a table when the table is first created as part of the create table statement
5. You can also add a primary key to a table after the table has been created using an Alter Table statement.

B. Syntax:

1. Create primary key when creating table:

```
<fieldname> data type CONSTRAINT (<fieldname> PRIMARY KEY)
```

<table>
<thead>
<tr>
<th>Create a new table called city with the following attribute:</th>
<th>Attribute</th>
<th>Data Type</th>
<th>Size</th>
<th>Primary Key</th>
</tr>
</thead>
<tbody>
<tr>
<td>city_id</td>
<td>Number</td>
<td>4</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>city_description</td>
<td>VarChar2</td>
<td>30</td>
<td>N</td>
<td></td>
</tr>
</tbody>
</table>

```
CREATE TABLE city 
  (city_ID NUMBER(4) CONSTRAINT city_ID PRIMARY KEY, 
   city_description VARCHAR2(30));
```

2. Add primary key to existing table:

```
ALTER TABLE <tablename> ADD (PRIMARY KEY (<fieldname>));
```
Add monkey_id as the primary key to the monkey table.

ALTER table monkey
ADD (PRIMARY KEY (monkey_id));

3. Create composite key when creating table:

PRIMARY KEY (<fieldname>, <fieldname>)

In the bank example, a customer can have numerous accounts at a bank and an account can have numerous customers associated with it. For instance, spouses may have a checking and a saving account at a bank and both spouses' names may be on both accounts. Let's say the bank creates a table called the Account_Assignment table. This table shows the relationship between the accounts and the account holders. The primary key for this table is a composite key which is two attributes that together make the primary key: the account_id and the customer_id.

<table>
<thead>
<tr>
<th>Create a new table called account_assignment with the following attributes</th>
<th>Attribute</th>
<th>Data Type</th>
<th>Size</th>
<th>Primary Key</th>
</tr>
</thead>
<tbody>
<tr>
<td>customer_id</td>
<td>Number</td>
<td>5</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>account_id</td>
<td>Number</td>
<td>5</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Active</td>
<td>Char</td>
<td>1</td>
<td>N</td>
<td></td>
</tr>
</tbody>
</table>

CREATE TABLE Account_Assignment
(customer_id NUMBER(5),
account_id NUMBER(5),
active CHAR(1),
PRIMARY KEY (customer_id, account_id));

4. Create composite key for existing table

ALTER TABLE <tablename> add (primary key (<fieldname>, (<fieldname>));

Make customer_id and account_id the primary key for the account_assignment table (this table already exists).

ALTER TABLE account_assignment
ADD (PRIMARY KEY (customer_id, account_id));

III. Foreign Key Constraint:

A. Foreign Key rules:

1. Foreign Keys represent relationship between tables. A foreign key is a column or group of columns whose value is derived from the primary key of another table. Foreign key constraints are used to enforce referential integrity, which means you can only place a value in table B if the value exists as a primary key in table A.

2. If a column is defined as a foreign key in a table, any inserts or updates will be rejected if a corresponding value does not exist in the primary key table. The referential integrity constraint specifies that the attribute in the referring table must refer to a primary key value that exists in the referenced table. The foreign key can also be null. (Remember a foreign MUST match to a primary key).

For example in the bank scenario, the account table has a field which refers to the Branch_ID in the Branch table. Thus referential integrity would mean that the values of Branch_ID in the Account table must already exist in the Branch table or they should be null. This prevents the Account table from referring to Branches that do not exist!
SQL Tutorial

3. The referencing column and referred column need not have the same name but MUST
be of same data type and size.

4. A foreign key can be specified when the table is first created as part of the create
statement or after the table has been created as part of an alter table statement.

5. It is often preferable to create tables without foreign keys and use an alter table
command to add the foreign key later. In this manner, if you do refer to them from other
tables, data entered satisfies the constraints. Also you cannot enter the foreign key
constraints until you have created the table to which that constraint refers.

B. Syntax

1. Create a foreign key in a new table:

   <fieldname> data type CONSTRAINT (<fieldname> REFERENCES <tablename of
   referenced table> (fieldname of referenced attribute))

   Create a new
table called
admin_office with
the following
attributes. Make
the city attribute
in the
admin_office
table a foreign
key to the city_id
attribute in the
city table.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data Type</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Admin_Office_ID</td>
<td>Number</td>
<td>5</td>
</tr>
<tr>
<td>Add1</td>
<td>VarChar2</td>
<td>30</td>
</tr>
<tr>
<td>City</td>
<td>Number</td>
<td>4</td>
</tr>
<tr>
<td>State</td>
<td>Char</td>
<td>2</td>
</tr>
<tr>
<td>Zip</td>
<td>Char</td>
<td>5</td>
</tr>
</tbody>
</table>

   CREATE TABLE Admin_Office
   (Admin_Office NUMBER(5),
   Add1 VARCHAR2(30),
   City Number(4) REFERENCES city (city_id),
   State CHAR(2),
   Zip CHAR(5));

2. Create a foreign key to an existing table

   ALTER TABLE <tablename of table with foreign key>
   ADD FOREIGN KEY <field name which is foreign key>
   REFERENCES <tablename of table with primary key to be referenced>
   (<fieldname of primary key to be reference>);

   Make admin_office.state a foreign
   key to state.stateid. Note, the
   state table already exists.

   ALTER TABLE admin_office
   ADD FOREIGN KEY (state)
   REFERENCES state(stateid);

IV. Null Value Constraint

SQL allows you to specify whether a field can be left blank (null) or must be populated (not null).
Typically you want fields to be populated (filled in) but sometimes it is appropriate for a field to be
left blank.

Null is an unknown value. When a particular row has a null value for a column it means that no
value has been entered. A null is not equal to zero. If a column is defined as not null the user
must enter data into that column for each row.

For instance, not everyone has a middle name. If you have a field for middle name you would not
want to require the user to populate the field if he does not have middle name. Accordingly, you
would not impose a not null constraint on that attribute meaning the user can leave it blank.
SQL Tutorial

A. Null Constraint Rules:

1. If you impose a not null constraint, the user **must** populate (enter data into) the field. Since a primary key cannot be null, you do not need to specify not null for the primary key.

2. You cannot add a new column to an EXISTING table with a NOT NULL constraint imposed on it. This is because the table already exists with data and by adding a column with such a constraint we would automatically have null values for all rows that already exist in the table. This makes it impossible for the constraint to be enforced by the DBMS.

B. Syntax

1. Create a column that allows null values:
   \[
   \text{<fieldname> data type } \text{NULL};
   \]

2. Create not null constraint: - CAUTION, only use in new tables.
   \[
   \text{<fieldname> data type } \text{NOT NULL};
   \]

<table>
<thead>
<tr>
<th>Create a null and not null constraints in new table called Manager</th>
<th>Attribute</th>
<th>Data Type</th>
<th>Size</th>
<th>PK</th>
<th>Nul</th>
<th>CREATE TABLE Manager (</th>
<th>PK</th>
<th>Nul</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Manager_ID</td>
<td>Number</td>
<td>5</td>
<td>Y</td>
<td>N</td>
<td>Manager_id PRIMARY KEY,</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Manager_LName</td>
<td>VarChar2</td>
<td>30</td>
<td></td>
<td></td>
<td>Manager_LName VARCHAR2(30) NOT NULL,</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Manager_FName</td>
<td>VarChar2</td>
<td>30</td>
<td>N</td>
<td></td>
<td>Manager_FName VARCHAR2(30) NOT NULL,</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Manager_MI</td>
<td>Char</td>
<td>1</td>
<td>Y</td>
<td></td>
<td>Manager_MI CHAR(1) NULL</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

V. Default Value Constraint

SQL allows you to specify a default value for an attribute. A default value is a value that the system will automatically enter for the value of an attribute unless the user decides to select another value.

When you enter the default value, it MUST match the data type and size for that field. Accordingly, if the field is state description and the data type is VarChar2 (30), the value must be a string and must be entered in quotes indicating it is a string. If the data type is a number, the default value must be entered as a number without quotes.

For instance, if a bank is in Jenkintown, PA, most of the customers will be coming from PA. To expedite data entry, we can specify that when a new customer is added to the customer table, fill 'PA' into the state column. If the customer isn’t from PA, the user can enter a different state.

A. Syntax :

1. Create a default value on an existing table
   \[
   \text{ALTER TABLE <tablename> modify <fieldname> default <default value>}
   \]

   Add a default value of PA to the Admin_Office.state:
   \[
   \text{ALTER TABLE Admin_Office MODIFY state DEFAULT 'PA';}
   \]

2. Create default value on new table.
   \[
   \text{<fieldname> data type default <default value>}
   \]
Create a new table called `bank_type` with the following attributes. Make the default bank_type_description 'MALL'.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data Type</th>
<th>Size</th>
<th>Primary Key</th>
<th>Null</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bank_type</td>
<td>Number</td>
<td>4</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>Bank_type_description</td>
<td>VarChar2</td>
<td>30</td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>

```sql
CREATE TABLE bank_type
    (Bank_type NUMBER(4) CONSTRAINT bank_type PRIMARY KEY,
    Bank_type_description VARCHAR2(30) DEFAULT 'MALL' NOT NULL);
```

VI. **Unique Constraint**

A unique constraint constrains the values in that column to be unique i.e. no two rows can have the same value. Of note, the column can have null values.

Remember that primary keys must always be unique but you do not need to specifically include this constraint. The unique constraint is automatically assumed when an attribute is designated as a primary key.

A. Syntax:

1. Create a unique constraint on existing table:
   ```sql
   <fieldname> data type unique
   ```
   Let's say you want to enforce that a social security number is unique.
   ```sql
   ss_number CHAR(5) UNIQUE
   ```

2. Create a unique constraint on new table:

   ```sql
   CREATE TABLE staff
    (staff_ID NUMBER(5) CONSTRAINT staff_id PRIMARY KEY,
    staff_LName VARCHAR2(30) NOT NULL ,
    staff_FName VARCHAR2(30) NOT NULL ,
    Staff_position_code CHAR(5) NULL UNIQUE);
   ```

VII. **Check Constraint**

Constraints can also be defined for the values that can be entered into an attribute. This type of constraint is known as a check constraint.

A check constraint is basically a list of the possible values for an attribute. The user must enter one of these values to enter information into the field.

A. Syntax:

1. Syntax to create a new check constraint:
   ```sql
   <fieldname> data type check (<fieldname> = value 1) or (<fieldname> = value 2) or (<fieldname> = value 3), etc.
   ```
2. Create check constraint to new table:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Data Type</th>
<th>Size</th>
<th>PK</th>
<th>Null</th>
</tr>
</thead>
<tbody>
<tr>
<td>animal_ID</td>
<td>NUMBER(5)</td>
<td>5</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>animal_Name</td>
<td>VARCHAR2(30)</td>
<td>30</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>species</td>
<td>VARCHAR2(15)</td>
<td>15</td>
<td>N</td>
<td></td>
</tr>
</tbody>
</table>

Let’s say the Jenkintown Bank only allows customers from specific states.

```sql
CHECK ((state = 'PA') OR (state = 'NJ') OR (state = 'DE') OR (state = 'NY'));
```

3. Create a check constraint on existing table:

```sql
ALTER TABLE animal
ADD
(CHECK (animal_name IN ('George', 'Paul', 'Ringo', 'John', 'Bruce', 'Sting')));
```
Chapter 21: DML - COMMIT, ROLLBACK, and FORMATTING

I. Commit and Rollback

Changes made to the database with INSERT, UPDATE, and DELETE statements are not committed to the database until you issue the COMMIT command.

A. ORACLE SQL provides two statements: COMMIT and ROLLBACK. Once the COMMIT command is issued, simply by typing COMMIT on the command prompt in SQL*Plus, changes to the database are made permanent.

An automatic COMMIT occurs when you exit from SQL*Plus and also when CREATE TABLE, DROP TABLE, and ALTER TABLE statements are used. That is if you are creating a table, deleting a table or modifying a table the changes are permanent.

Syntax: COMMIT;

B. If you want to undo any changes you made to the database, before you issue the COMMIT command, you can call the ROLLBACK command. The rollback reverses all the changes made to the database up until the last COMMIT statement.

Syntax: ROLLBACK;

II. Formatting Results

You will notice that in SQL*Plus the output wraps and is difficult to read. There are a number of formatting commands that help improve the look of the output:

A. COLUMN command:

1. The column command allows you to specify how specific attributes should be formatted.
   a. COL description FORMAT A# characters
      - Displays a maximum of the number of characters specified.
      - If the values in the column don’t fit, the data will wrap within the column.
      - The column heading is truncated to the specified length.

2. The format for the column persists until:
   a. You re-specify the format for the columns by clearing the format for the column through the CLEAR COLUMNS command OR
   b. You exit SQL*Plus.
Here is a snapshot of all the rows in the product table. You can see it is wrapping and hard to read:

```
<table>
<thead>
<tr>
<th>SKU</th>
<th>PRODUCT_DESCRIPTION</th>
<th>PRICE</th>
</tr>
</thead>
<tbody>
<tr>
<td>13133</td>
<td>Cordless phone</td>
<td>21.99</td>
</tr>
<tr>
<td>15135</td>
<td>Merkury boom box</td>
<td>45.87</td>
</tr>
<tr>
<td>145566</td>
<td>Sony Trinitron 30&quot; TV</td>
<td>345.76</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SKU</th>
<th>PRODUCT_DESCRIPTION</th>
<th>PRICE</th>
</tr>
</thead>
<tbody>
<tr>
<td>968767</td>
<td>Iron tiki torch</td>
<td>7.99</td>
</tr>
<tr>
<td>26666</td>
<td>Colonial black carriage light</td>
<td>58.99</td>
</tr>
<tr>
<td>269988</td>
<td>Ear Force gamer head set</td>
<td>129.99</td>
</tr>
<tr>
<td>405144</td>
<td>Extra Large Wind Chimes</td>
<td>127.54</td>
</tr>
<tr>
<td>276555</td>
<td>Deluxe Patio Set - 6 chairs + umbrella</td>
<td>899.99</td>
</tr>
</tbody>
</table>
```

If I format the columns as follows I can set it so the attributes don’t run around. Rather, the text will wrap within the column that holds the data.

```
<table>
<thead>
<tr>
<th>COL</th>
<th>SKU</th>
<th>PRODUCT_DESCRIPTION</th>
<th>PRICE</th>
</tr>
</thead>
<tbody>
<tr>
<td>product_description</td>
<td>13133</td>
<td>Cordless phone</td>
<td>21.99</td>
</tr>
<tr>
<td></td>
<td>15135</td>
<td>Merkury boom box</td>
<td>45.87</td>
</tr>
<tr>
<td></td>
<td>145566</td>
<td>Sony Trinitron 30&quot; TV</td>
<td>345.76</td>
</tr>
<tr>
<td></td>
<td>968767</td>
<td>Iron tiki torch</td>
<td>7.99</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>COL</th>
<th>SKU</th>
<th>PRODUCT_DESCRIPTION</th>
<th>PRICE</th>
</tr>
</thead>
<tbody>
<tr>
<td>producttype</td>
<td>13</td>
<td>Electronics</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>14</td>
<td>Electronics</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>45</td>
<td>Lawn and Garden</td>
<td>5</td>
</tr>
</tbody>
</table>
```

**B. Formatting Numbers**

If we are formatting a column which has a number data type, we can change the format with a format model in the COLUMN command.

1. Formatting for currency syntax:

   ```
   COL <fieldname> FORMAT $9,999.99
   ```
If I query the price from the product table, I will retrieve the following:

```sql
SQL> select price from product;

<table>
<thead>
<tr>
<th>PRICE</th>
</tr>
</thead>
<tbody>
<tr>
<td>21.99</td>
</tr>
<tr>
<td>45.87</td>
</tr>
<tr>
<td>345.76</td>
</tr>
<tr>
<td>7.99</td>
</tr>
<tr>
<td>58.99</td>
</tr>
<tr>
<td>129.99</td>
</tr>
<tr>
<td>127.54</td>
</tr>
<tr>
<td>899.99</td>
</tr>
</tbody>
</table>

8 rows selected.
```

If I query the price from the product table, I will retrieve the following:

```sql
COL price FORMAT $9,999.99

SQL> select price from product;

<table>
<thead>
<tr>
<th>PRICE</th>
</tr>
</thead>
<tbody>
<tr>
<td>$21.99</td>
</tr>
<tr>
<td>$45.87</td>
</tr>
<tr>
<td>$345.76</td>
</tr>
<tr>
<td>$7.99</td>
</tr>
<tr>
<td>$58.99</td>
</tr>
<tr>
<td>$129.99</td>
</tr>
<tr>
<td>$127.54</td>
</tr>
<tr>
<td>$899.99</td>
</tr>
</tbody>
</table>

8 rows selected.
```
### Format Mask

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>YEAR</td>
<td>Year, spelled out</td>
</tr>
<tr>
<td>YYYY</td>
<td>4-digit year</td>
</tr>
<tr>
<td>YYY</td>
<td>Last 3, 2, or 1 digit(s) of year.</td>
</tr>
<tr>
<td>YY</td>
<td>IYY</td>
</tr>
<tr>
<td>Y</td>
<td>Last 3, 2, or 1 digit(s) of ISO year.</td>
</tr>
<tr>
<td>IYY</td>
<td>IY</td>
</tr>
<tr>
<td>IY</td>
<td>I</td>
</tr>
<tr>
<td>IYYYY</td>
<td>4-digit year based on the ISO standard</td>
</tr>
<tr>
<td>RRRR</td>
<td>Accepts a 2-digit year and returns a 4-digit year. A value between 0-49 will return a 20xx year. A value between 50-99 will return a 19xx year.</td>
</tr>
<tr>
<td>Q</td>
<td>Quarter of year (1, 2, 3, 4; JAN-MAR = 1).</td>
</tr>
<tr>
<td>MM</td>
<td>Month (01-12; JAN = 01).</td>
</tr>
<tr>
<td>MON</td>
<td>Abbreviated name of month.</td>
</tr>
<tr>
<td>MONTH</td>
<td>Name of month, padded with blanks to length of 9 characters.</td>
</tr>
<tr>
<td>RM</td>
<td>Roman numeral month (I-XII; JAN = I).</td>
</tr>
<tr>
<td>WW</td>
<td>Week of year (1-53) where week 1 starts on the first day of the year and continues to the seventh day of the year.</td>
</tr>
<tr>
<td>W</td>
<td>Week of month (1-5) where week 1 starts on the first day of the month and ends on the seventh.</td>
</tr>
<tr>
<td>IW</td>
<td>Week of year (1-52 or 1-53) based on the ISO standard.</td>
</tr>
<tr>
<td>D</td>
<td>Day of week (1-7).</td>
</tr>
<tr>
<td>DAY</td>
<td>Name of day.</td>
</tr>
</tbody>
</table>

### Format Mask

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>DD</td>
<td>Day of month (1-31).</td>
</tr>
<tr>
<td>DDD</td>
<td>Day of year (1-366).</td>
</tr>
<tr>
<td>DY</td>
<td>Abbreviated name of day.</td>
</tr>
<tr>
<td>J</td>
<td>Julian day; the number of days since January 1, 4712 BC.</td>
</tr>
<tr>
<td>HH</td>
<td>Hour of day (1-12).</td>
</tr>
<tr>
<td>HH12</td>
<td>Hour of day (1-12).</td>
</tr>
<tr>
<td>HH24</td>
<td>Hour of day (0-23).</td>
</tr>
<tr>
<td>MI</td>
<td>Minute (0-59).</td>
</tr>
<tr>
<td>SS</td>
<td>Second (0-59).</td>
</tr>
<tr>
<td>SSSSS</td>
<td>Seconds past midnight (0-86399).</td>
</tr>
<tr>
<td>FF</td>
<td>Fractional seconds. Use a value from 1 to 9 after FF to indicate the number of digits in the fractional seconds. For example, ‘FF4’.</td>
</tr>
<tr>
<td>AM, A.M., PM, or P.M.</td>
<td>Meridian indicator</td>
</tr>
<tr>
<td>AD or A.D</td>
<td>AD indicator</td>
</tr>
<tr>
<td>BC or B.C.</td>
<td>BC indicator</td>
</tr>
<tr>
<td>TZD</td>
<td>Daylight savings information. For example, ‘PST’</td>
</tr>
<tr>
<td>TZH</td>
<td>Time zone hour.</td>
</tr>
<tr>
<td>TZM</td>
<td>Time zone minute.</td>
</tr>
<tr>
<td>TZR</td>
<td>Time zone region.</td>
</tr>
</tbody>
</table>
Chapter 22: DML - Inserting Data into Tables

I. Insert Statement

We add rows to a table (populate a table) using the insert command.

A. The INSERT statement starts with the keywords INSERT INTO, followed by the name of the table into which the row(s) is to be appended. This is then followed optionally by a list of columns of the table within parentheses, the keyword VALUES, and finally a list of values within parentheses.

B. The insert statement has one of the following two forms. The only difference is that if you don’t specify the column names, you must make sure to add the values in the exact same order as the columns are defined in the table. If you don’t have the columns in the proper order you take a chance of storing a value in the wrong attribute.

- INSERT INTO <tablename> [(column, {, column})] VALUES (expression {, expression});
- INSERT INTO <tablename> VALUES (expression {, expression});

<table>
<thead>
<tr>
<th>Add the following row to the customer table:</th>
<th>INSERT INTO customer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Customer_id 12345</td>
<td>(Customer_ID, Customer_lname,</td>
</tr>
<tr>
<td>Lname Springsteen</td>
<td>Customer_fname, Customer_add1,</td>
</tr>
<tr>
<td>Fname Bruce</td>
<td>Customer_city, state, zip)</td>
</tr>
<tr>
<td>Add1 1212 Awesome Way</td>
<td>VALUES (12345, ‘Springsteen’, ‘Bruce’, ‘1212</td>
</tr>
<tr>
<td>City Rumsford</td>
<td>Awesome Way’, ‘Rumsford’, ‘PA’, ‘19112’);</td>
</tr>
<tr>
<td>State PA</td>
<td>OR</td>
</tr>
<tr>
<td>Zip 119112</td>
<td>INSERT INTO customer</td>
</tr>
<tr>
<td></td>
<td>VALUES (12345, ‘Springsteen’, ‘Bruce’, ‘1212</td>
</tr>
<tr>
<td></td>
<td>Awesome Way’, ‘Rumsford’, ‘PA’, ‘19112’);</td>
</tr>
</tbody>
</table>

II. Querying Rows with Select Statement

We will learn more about querying data in future chapters. As an introduction, however, it is important to understand how to retrieve information populated into a table. To retrieve information from an Oracle table, use the Select Statement.

The SQL select statement is very English-like. We SELECT the columns that we wish to view, FROM the table that contains the columns. Optionally, we can add criteria through a WHERE clause and sort it in some ORDER. For now, we will only worry about selecting all the rows from a table.

A. SELECT *

The asterisk (*) is a shortcut so we don’t need to specify all the columns in the table. In the example below, all the attributes in the bank_customer table are displayed. SQL*Plus wraps the columns so don’t be concerned about the fact that a row appears on multiple lines. You can refer to the section on formatting if you would like to specify how the wrapping should appear.
SQL Tutorial

1. Syntax: SELECT * FROM <tablename>;

<table>
<thead>
<tr>
<th>CUSTOMER_ID</th>
<th>CUSTOMER_LNAME</th>
<th>CUSTOMER_FNAME</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ADD1</th>
<th>CITY</th>
<th>ST</th>
<th>ZIP</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>-----</td>
<td>------</td>
<td>----</td>
<td>-----</td>
<td>---</td>
</tr>
<tr>
<td>10000 Sinatra</td>
<td>Frank</td>
<td>NJ 08703 M</td>
<td></td>
<td></td>
</tr>
<tr>
<td>144 Woodstream Blvd.</td>
<td>Hoboken</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10100 Martin</td>
<td>Dean</td>
<td>PA 19115</td>
<td></td>
<td></td>
</tr>
<tr>
<td>454 St. Orange Street</td>
<td>Philadelphia</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10200 Patty</td>
<td>Davis</td>
<td>PA 19006 E</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8921 Circle Drive</td>
<td>Huntingdon Valley</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10300 Mobey</td>
<td>Amanda</td>
<td>PA 19004 J</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1716 Summerton Avenue</td>
<td>Warrington</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

III. TO_DATE:

A. Oracle expects that information entered that appears like a number will be stored as a number. Accordingly, if the user enters information such as 07/14/1979 in quotes, SQL interprets this as just a series of characters which just so happen to be numbers. SQL will store this value as a string (a series of characters). As you might have guessed, SQL will not recognize these characters as a date unless you specify it as such.

B. While SQL will allow the user to store a date as a string, you will not be able do calculations with date. Performing calculations on dates is essential in business. For instance, a company might need to calculate how many days elapsed between two dates. Unless you indicate to SQL that the string is actually a date, such a calculation could not occur. As such, if you will be doing any calculations with time, you will want to use the TO_DATE function.

C. The TO_DATE function is a function that is predefined into Oracle. It converts a string into a date. There is a complementary function called TO_CHAR which converts a date into a string which allows the user more control over the output’s appearance.

D. If you do not use the default format for date you will have to tell Oracle what format you are using with TO_DATE function.

E. Syntax:

1. The TO_DATE’s syntax is as follows:

   to_date( string1, [ format_mask ], [ nls_language ] )

   - string1 is the string that will be converted to a date.
   - format_mask is optional. This is the format that will be used to convert string1 to a date.
   - nls_language is optional. This is the nls language used to convert string1 to a date.
SQL Tutorial

2. A list of options for the \textit{format\_mask} parameter is shown below. These parameters can be used in many combinations.

\begin{center}
\begin{tabular}{l|l}
\texttt{to\_date('2003/07/09', 'yyyy/mm/dd')} & returns a date value of July 9, 2003. \\
\texttt{to\_date('070903', 'MMDDYY')} & returns a date value of July 9, 2003. \\
\texttt{to\_date('20020315', 'yyyymmdd')} & returns a date value of Mar 15, 2002. \\
\end{tabular}
\end{center}

Let's say you want to add values into a student table which was created with the following create table statement:

\begin{verbatim}
CREATE TABLE student 
( 
    sid NUMBER(5) CONSTRAINT student_sid_pk PRIMARY KEY, 
    slname VARCHAR2(30) CONSTRAINT student_slname_nn NOT NULL, 
    sfname VARCHAR2(30) CONSTRAINT student_sfname_nn NOT NULL, 
    smi CHAR(1), 
    sadd VARCHAR2(25), 
    scity VARCHAR2(20), 
    sstate CHAR(2) DEFAULT 'NJ', 
    szip VARCHAR2(9), 
    sphone VARCHAR2(10), 
    sclass CHAR(2) CONSTRAINT student_sclass_cc 
         CHECK ((sclass = 'FR') OR (sclass = 'SO') OR (sclass = 'JR') OR 
                (sclass = 'SR')), 
    sdob DATE, 
    spin NUMBER(4), 
    fid NUMBER(4) CONSTRAINT student_fid_fk 
         REFERENCES faculty(fid), 
    );
\end{verbatim}

To add a row for Sarah Miller, you would add the following insert statement which includes a \texttt{TO\_DATE} that indicates that the string of 07/14/1979 is to be converted into a date using the format of MM/DD/YYYY.

\begin{verbatim}
To add a row for Sarah Miller, you would add the following insert statement which includes a \texttt{TO\_DATE} that indicates that the string of 07/14/1979 is to be converted into a date using the format of MM/DD/YYYY.

\begin{tabular}{|l|l|}
\hline
\texttt{INSERT INTO student} & VALUES (100, 'Miller', 'Sarah', 'M', '144 Windridge Blvd.','Eau Claire', 'WI', '54703', '7155559876', 'SR', \texttt{TO\_DATE('07/14/1979', 'MM/DD/YYYY')}, 8891, 1); \\
\hline
\end{tabular}
\end{verbatim}
Chapter 23: DML - SELECT

I. Select Query Background

As we briefly discussed earlier, we use a SELECT clause to retrieve information from the database. The simple SELECT has three clauses: SELECT, FROM, and WHERE.

A. SELECT:

The SELECT clause tells which attributes should be displayed in the output.

1. The order of the attributes in the select clause corresponds with the order the attributes will appear in the output.
2. The SELECT clause is required.

B. FROM:

The FROM clause lists the tables where the data is stored and from which the data should be retrieved.

1. The order of the table names is irrelevant to the output.
2. The FROM clause is required.

C. WHERE:

The WHERE clause is used when the user wants to filter the rows returned based on specific criteria. As we’ll learn shortly, the WHERE clause is also needed when there are multiple tables in the query and we need to join the tables.

1. The order of the criteria or joins in the where clause is not significant.
2. The WHERE clause is optional and is only required if the rows are filtered based on a specific criteria AND/OR there is a join.

```sql
SELECT [distinct] <expression> {, <expression>}
FROM <tablename> [<alias>] {, <tablename> [<alias>]}
[WHERE <search_condition>]
```

II. Simple SELECT

The simple SELECT includes:

- the SELECT clause which shows the attributes to appear on the output and
- the FROM clause which shows from which tables the attributes appear:

<table>
<thead>
<tr>
<th>List all the student classes</th>
<th>SC</th>
<th>--</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT sclass FROM student;</td>
<td>SR</td>
<td>JR</td>
</tr>
<tr>
<td></td>
<td>SO</td>
<td>SO</td>
</tr>
<tr>
<td></td>
<td>FR</td>
<td></td>
</tr>
</tbody>
</table>
III. **The Asterisk (*) Wildcard**

As described earlier, the asterisk is a wild card in SQL*Plus.

The asterisk in a SELECT clause says that all the attributes (columns) in the table should appear in the output. The order of the columns in the output will correspond to the order of the columns in the table.

<table>
<thead>
<tr>
<th>List all the locations in the location table</th>
<th>LOCID BLDG_CODE ROOM   CAPACITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT * FROM location;</td>
<td>11 SP 101 150</td>
</tr>
<tr>
<td></td>
<td>12 SP 202 40</td>
</tr>
<tr>
<td></td>
<td>13 SP 103 35</td>
</tr>
<tr>
<td></td>
<td>14 SP 105 35</td>
</tr>
<tr>
<td></td>
<td>15 BUS 105 42</td>
</tr>
<tr>
<td></td>
<td>16 BUS 404 35</td>
</tr>
<tr>
<td></td>
<td>17 BUS 421 35</td>
</tr>
<tr>
<td></td>
<td>18 BUS 211 55</td>
</tr>
<tr>
<td></td>
<td>19 BUS 424 1</td>
</tr>
<tr>
<td></td>
<td>20 BUS 402 1</td>
</tr>
<tr>
<td></td>
<td>21 BUS 433 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LOCID BLDG_CODE ROOM   CAPACITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>22 LIB 217 1</td>
</tr>
<tr>
<td>23 LIB 222 1</td>
</tr>
</tbody>
</table>

If I describe the location table to see the tables, structure, you’ll note the order of the attributes in the output above (locid, bldg_code, room, capacity) corresponds with the order of the attributes in the table.

IV. **Filtering in Select Statement (Projection) to Specific Columns:**

Up until this point, we have not had the ability to modify the order of the columns in the output of our queries. Moreover, all of the attributes were displayed in every instance. However, a user can determine which attributes (columns) appear as well as the order that they appear in the output. By listing the specific columns to appear, the user is projecting (displaying) only certain columns of the table and an order specified by the user.

Let’s say I want a report that shows student id, first name and last name for all the students and I want the output in that order. As we saw earlier, if I output all of the data in the student table, it will appear as follows:

<table>
<thead>
<tr>
<th>SID</th>
<th>LNAME</th>
<th>SNAME</th>
<th>M</th>
<th>SADD</th>
<th>SCITY</th>
<th>SSTATE</th>
<th>ZIP</th>
<th>SPHONE</th>
<th>SCCLASS</th>
<th>SDOB</th>
<th>SPIN</th>
<th>FID</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>Pitt</td>
<td>Brad</td>
<td>M</td>
<td>144 Windridge Blvd.</td>
<td>Cherry Hill</td>
<td>NJ</td>
<td>54703</td>
<td>7155595676</td>
<td>SR</td>
<td>7/14/1979</td>
<td>8891</td>
<td>2</td>
</tr>
<tr>
<td>101</td>
<td>Julie</td>
<td>Angeline</td>
<td>D</td>
<td>454 St. John's Street</td>
<td>Wilmington</td>
<td>DE</td>
<td>54702</td>
<td>7155523245</td>
<td>SR</td>
<td>8/19/1979</td>
<td>1230</td>
<td>1</td>
</tr>
<tr>
<td>102</td>
<td>Damon</td>
<td>Matt</td>
<td></td>
<td>821 Circle Drive</td>
<td>Philadelphia</td>
<td>PA</td>
<td>54715</td>
<td>7155558907</td>
<td>JR</td>
<td>10/10/1977</td>
<td>1613</td>
<td>1</td>
</tr>
<tr>
<td>103</td>
<td>Clooney</td>
<td>George</td>
<td>J</td>
<td>1716 Summit St.</td>
<td>Eau Claire</td>
<td>NJ</td>
<td>54703</td>
<td>7155556902</td>
<td>SO</td>
<td>9/24/1978</td>
<td>1841</td>
<td>5</td>
</tr>
<tr>
<td>104</td>
<td>Bloom</td>
<td>Orlando</td>
<td>R</td>
<td>1780 Broad Street</td>
<td>Philadelphia</td>
<td>ID</td>
<td>54701</td>
<td>7155558889</td>
<td>SO</td>
<td>11/20/1977</td>
<td>4420</td>
<td>4</td>
</tr>
<tr>
<td>105</td>
<td>Depp</td>
<td>Johnny</td>
<td>S</td>
<td>1818 Silver Street</td>
<td>Elk Mound</td>
<td>ID</td>
<td>54712</td>
<td>7155554944</td>
<td>FR</td>
<td>12/4/1977</td>
<td>9188</td>
<td>3</td>
</tr>
</tbody>
</table>

This would be fine except there is more information than I need. I can determine the exact output I want with the following query:
V. **Order of Attributes in Select**

The order you list the attributes in the `SELECT` clause determines the order of the columns in your output. Each attribute is separated by a comma.

<table>
<thead>
<tr>
<th>FLNAME</th>
<th>FFNAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>Knightly</td>
<td>Keira</td>
</tr>
<tr>
<td>Streep</td>
<td>Meryl</td>
</tr>
<tr>
<td>Redford</td>
<td>Robert</td>
</tr>
<tr>
<td>Newman</td>
<td>Paul</td>
</tr>
<tr>
<td>Dench</td>
<td>Judith</td>
</tr>
<tr>
<td>Binoche</td>
<td>Juliette</td>
</tr>
</tbody>
</table>

6 rows selected.

To show the first name first and the last name last, revise the query as follows:

<table>
<thead>
<tr>
<th>FFNAME</th>
<th>FLNAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>Keira</td>
<td>Knightly</td>
</tr>
<tr>
<td>Meryl</td>
<td>Streep</td>
</tr>
<tr>
<td>Robert</td>
<td>Redford</td>
</tr>
<tr>
<td>Paul</td>
<td>Newman</td>
</tr>
<tr>
<td>Judith</td>
<td>Dench</td>
</tr>
<tr>
<td>Juliette</td>
<td>Binoche</td>
</tr>
</tbody>
</table>

6 rows selected.

VI. **Distinct**

The following query displays the sclass from the student table. You will note SR (for senior) appears twice and SO (for sophomore) appears twice as well. There are a total of 6 rows selected.

<table>
<thead>
<tr>
<th>SC</th>
</tr>
</thead>
<tbody>
<tr>
<td>SR</td>
</tr>
<tr>
<td>SR</td>
</tr>
<tr>
<td>JR</td>
</tr>
<tr>
<td>SO</td>
</tr>
<tr>
<td>SO</td>
</tr>
<tr>
<td>FR</td>
</tr>
</tbody>
</table>

6 rows selected.
This makes sense if you look at the student table:

<table>
<thead>
<tr>
<th>SID</th>
<th>SLNAME</th>
<th>SNAME</th>
<th>SMI</th>
<th>SADD</th>
<th>SCITY</th>
<th>SSTATE</th>
<th>SZIP</th>
<th>SPHONE</th>
<th>SCLASS</th>
<th>SDOB</th>
<th>SPIN</th>
<th>FID</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>Pitt</td>
<td>Brad</td>
<td>M</td>
<td>144 Windridge Blvd.</td>
<td>Cherry Hill</td>
<td>NJ</td>
<td>54703</td>
<td>7155559876</td>
<td>SR</td>
<td>7/14/1979</td>
<td>8891</td>
<td>2</td>
</tr>
<tr>
<td>101</td>
<td>Jolie</td>
<td>Angelina</td>
<td>D</td>
<td>454 St. John's Street</td>
<td>Wilmington</td>
<td>DE</td>
<td>54702</td>
<td>7155552345</td>
<td>SR</td>
<td>8/19/1979</td>
<td>1230</td>
<td>1</td>
</tr>
<tr>
<td>102</td>
<td>Damon</td>
<td>Matt</td>
<td>M</td>
<td>8921 Circle Drive</td>
<td>Philadelphia</td>
<td>PA</td>
<td>54715</td>
<td>7155555967</td>
<td>JR</td>
<td>10/10/1977</td>
<td>1613</td>
<td>1</td>
</tr>
<tr>
<td>103</td>
<td>Clooney</td>
<td>George</td>
<td>J</td>
<td>1716 Summit St.</td>
<td>Eau Claire</td>
<td>NJ</td>
<td>54703</td>
<td>7155556682</td>
<td>SO</td>
<td>9/24/1978</td>
<td>1841</td>
<td>5</td>
</tr>
<tr>
<td>104</td>
<td>Bloom</td>
<td>Orlando</td>
<td>R</td>
<td>1780 Broad Street</td>
<td>Philadelphia</td>
<td>ID</td>
<td>54701</td>
<td>7155558899</td>
<td>SO</td>
<td>11/20/1977</td>
<td>4420</td>
<td>4</td>
</tr>
<tr>
<td>105</td>
<td>Depp</td>
<td>Johnny</td>
<td>S</td>
<td>1816 Silver Street</td>
<td>Elk Mound</td>
<td>ID</td>
<td>54712</td>
<td>7155554944</td>
<td>FR</td>
<td>12/4/1977</td>
<td>9188</td>
<td>3</td>
</tr>
</tbody>
</table>

It is clear that there are 6 rows which correspond to the row count in the prior query. You can also see that Brad Pitt and Angelina Jolie are the two seniors while George Clooney and Orlando Bloom are the two sophomores.

What if you just want to see the list of states in the table. If you write a query that shows the states, there would be many duplicates and it would be quite lengthy. Fortunately, SQL has a command that solves this problem. To display only the unique values, in other words, to suppress duplication of the values, use the DISTINCT keyword to return unduplicated results.

A. Syntax:

```
Distinct <fieldname>
```

Show a listing of the possible types of student class. This will show the values in the sclass attribute but a value will appear only once regardless of how many rows are in the table with that value.

```
SELECT DISTINCT sclass FROM student;
```

VII. **Restrict relational operator – Where Clause**

So far we have been able to query all the rows of a table. However, most times we only want to see rows in a table that meet certain criteria.

With a where clause, the user can limit the results to only those rows that satisfy a condition. The condition is indicated in the WHERE clause.

A. Syntax: … WHERE <fieldname> = criteria

1. Let's say instead of seeing all the students in the student table we want to see the the first and last names of students whose faculty advisor (FID) is 1. To write this query, we need to start with our SELECT clause. Remember, every select query has to start with a SELECT clause. In this case, we will only select the first and last name and we will put the attributes in that order, as well.

   SELECT fname, lname

2. Next we need to tell SQL which table holds this data. We do that in the FROM clause. Every SELECT query must have a FROM clause.

   FROM student

3. Finally, since we only want the rows in the table with a faculty advisor of 1, we need to restrict the results using a WHERE clause.

   WHERE fid = 1

If we put it all together, the query looks like:

```
SELECT sfname, slname
FROM student
WHERE fid = 1;
```
B. It is often advisable to quality check the query by adding the criteria into the SELECT clause to verify it ran correctly. For instance, in this case, we can add fid into the SELECT clause so it appears on the output. Then we can be sure that the rows met our criteria.

```sql
SELECT sfname, slname, fid
FROM student
WHERE fid = 1;
```

<table>
<thead>
<tr>
<th>SFNAME</th>
<th>SLNAME</th>
<th>FID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Angelina</td>
<td>Julie</td>
<td>1</td>
</tr>
<tr>
<td>Matt</td>
<td>Damon</td>
<td>1</td>
</tr>
</tbody>
</table>

VIII. Searching for rows

A. String Searches:

If you are searching for a string (char, varchar, or dates), the letters must be entered in single quotes. For instance, suppose you want to identify all students who are sophomores. Since the data in the sclass field is a varchar data type, when we search for a specific value, the value must be entered in quotes.

```sql
SELECT sfname, slname
FROM student
WHERE SCLASS = 'SO';
```

<table>
<thead>
<tr>
<th>SFNAME</th>
<th>SLNAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>George</td>
<td>Clooney</td>
</tr>
<tr>
<td>Orlando</td>
<td>Bloom</td>
</tr>
</tbody>
</table>

B. The criteria entered in the where clause must match exactly to the way the value is stored in the database. Accordingly, the values are case sensitive! As you’ll note, attribute names and SQL keywords and operators (such as select, where, and, etc.) are NOT case sensitive.

```sql
SELECT sfname, slname
FROM student
WHERE SCLASS = 'Freshman';
```

no rows selected

Why won’t any rows return? Well, let’s start by describing the table. As you can see, sclass is only 2 characters long. It clearly can’t hold a string as long as “Freshman”.

<table>
<thead>
<tr>
<th>NAME</th>
<th>NULL?</th>
<th>TYPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>SID</td>
<td>NOT NULL</td>
<td>NUMBER(5)</td>
</tr>
<tr>
<td>SLNAME</td>
<td>NOT NULL</td>
<td>VARCHAR2(30)</td>
</tr>
<tr>
<td>SFNAME</td>
<td>NOT NULL</td>
<td>VARCHAR2(30)</td>
</tr>
<tr>
<td>SMI</td>
<td>CHAR(1)</td>
<td></td>
</tr>
<tr>
<td>SADD</td>
<td>VARCHAR2(25)</td>
<td></td>
</tr>
<tr>
<td>SCTY</td>
<td>VARCHAR2(20)</td>
<td></td>
</tr>
<tr>
<td>SSTATE</td>
<td>CHAR(2)</td>
<td></td>
</tr>
<tr>
<td>SZIP</td>
<td>VARCHAR2(9)</td>
<td></td>
</tr>
<tr>
<td>SPHONE</td>
<td>VARCHAR2(10)</td>
<td></td>
</tr>
<tr>
<td>SCLASS</td>
<td>CHAR(2)</td>
<td></td>
</tr>
<tr>
<td>SDOB</td>
<td>DATE</td>
<td></td>
</tr>
<tr>
<td>SPIN</td>
<td>NOT NULL NUMBER(4)</td>
<td></td>
</tr>
<tr>
<td>FID</td>
<td>NUMBER(4)</td>
<td></td>
</tr>
</tbody>
</table>
To see what values are stored in this attribute, write the distinct clause as follows:

```
SELECT DISTINCT sclass
FROM student;
```

<table>
<thead>
<tr>
<th>sclass</th>
</tr>
</thead>
<tbody>
<tr>
<td>SC</td>
</tr>
<tr>
<td>FR</td>
</tr>
<tr>
<td>JR</td>
</tr>
<tr>
<td>SO</td>
</tr>
<tr>
<td>SR</td>
</tr>
</tbody>
</table>

The distinct clause allows the user to see that sclass stores only a 2 letter abbreviation. Moreover, the abbreviation for “Freshman” is “FR”. Of note, if the user searches for “FR” in small case, no rows will return because to computer ‘FR’ is quite different than ‘fr’;

```
SELECT sfname, slname
FROM student
WHERE sclass = 'fr';
```

<table>
<thead>
<tr>
<th>SFNAME</th>
<th>SLNAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mickey</td>
<td>Connoly</td>
</tr>
</tbody>
</table>

Once the query is corrected to search for all students with a sclass of ‘FR’, the proper results will return:

```
SELECT sfname, slname
FROM student
WHERE sclass = 'FR';
```

<table>
<thead>
<tr>
<th>SFNAME</th>
<th>SLNAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mickey</td>
<td>Connoly</td>
</tr>
</tbody>
</table>

Because the values of data must match precisely in order to retrieve the desired rows from a database, data quality is critical. If data is entered into an attribute in different cases or with abbreviations or typographical errors, the user will not be able to retrieve all the rows. Accordingly, we typically suggest adding a lookup table whenever an attribute has a discrete number of values rather than permitting the user to enter free text.

IX. **Wildcard characters:**

Sometimes you are searching for a string (word or list of characters) and you can't locate it. In such an occasion the use of a wildcard can be tremendously helpful. Wildcards can be used in query expressions to expand word searches into pattern searches.

Typically, instead of using the equal sign (=), we use the like operator to indicate that we are doing a string search.

The wildcard characters are:

<table>
<thead>
<tr>
<th>Wildcard Character</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>%</td>
<td>The percent wildcard specifies that any characters can appear in multiple positions represented by the wildcard.</td>
</tr>
<tr>
<td>_</td>
<td>The underscore wildcard specifies a single position in which any character can occur.</td>
</tr>
</tbody>
</table>
SQL Tutorial

A. Syntax: `<fieldname>% OR <fieldname>_`

Show the students who have a last name that starts with P:

```
SELECT sfname, slname
FROM student
WHERE slname LIKE 'P%';
```

Show all the students with an O in their last name:

```
SELECT sfname, slname
FROM student
WHERE slname LIKE '%o%';
```

I can’t recall if Matthew Damon spells his nickname with one t or 2:

```
SELECT sfname, slname
FROM student
WHERE sfname LIKE 'Mat_';
```

<table>
<thead>
<tr>
<th>SFNAME</th>
<th>SLNAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brad</td>
<td>Pitt</td>
</tr>
<tr>
<td>Angelina</td>
<td>Jolie</td>
</tr>
<tr>
<td>Matt</td>
<td>Damon</td>
</tr>
<tr>
<td>George</td>
<td>Clooney</td>
</tr>
<tr>
<td>Orlando</td>
<td>Bloom</td>
</tr>
<tr>
<td>Matt</td>
<td>Damon</td>
</tr>
</tbody>
</table>

X. Comparison Operators

In addition to looking to see if an attribute is equal to a specific value, we might want to know if the attribute is greater than (or greater than or equal to), less than, (or less than or equal to), or not equal to a value. Fortunately, SQL allows for all of these combinations with comparison operators.

The comparison operators are typically used in the WHERE clause to show the relationship between a variable and some value. A comparison operator can be used in combination of with another comparison operator to further filter the records.

The comparison operators include:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;</td>
<td>Greater Than</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater than or equal to</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less Than</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less than or equal to</td>
</tr>
<tr>
<td>!=, &lt;&gt;</td>
<td>Not equal to</td>
</tr>
<tr>
<td>IN</td>
<td>Looks for records with a value which is equal to one of the values in a list</td>
</tr>
<tr>
<td>BETWEEN</td>
<td>Looks for records with a value between two different values</td>
</tr>
</tbody>
</table>

Show the room numbers and capacity of the room of all rooms that have a capacity greater than or equal to 40:

```
SELECT Room, capacity
FROM location
WHERE capacity >= 40;
```

<table>
<thead>
<tr>
<th>ROOM</th>
<th>CAPACITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>150</td>
</tr>
<tr>
<td>202</td>
<td>40</td>
</tr>
<tr>
<td>105</td>
<td>42</td>
</tr>
<tr>
<td>211</td>
<td>55</td>
</tr>
</tbody>
</table>
Show the room numbers and capacity of the room of all rooms that have a capacity greater than or equal to 40 and less than or equal to 70.

```
SELECT Room, capacity
FROM location
WHERE capacity >= 40 and capacity <= 70;
```

<table>
<thead>
<tr>
<th>ROOM</th>
<th>CAPACITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>202</td>
<td>40</td>
</tr>
<tr>
<td>105</td>
<td>42</td>
</tr>
<tr>
<td>211</td>
<td>55</td>
</tr>
</tbody>
</table>

Show the building code and capacity of all the buildings except SPK. Show a building code only once.

```
SELECT distinct bldg_code, capacity
FROM location
WHERE bldg_code !='SPK';
```

OR

```
SELECT distinct bldg_code, capacity
FROM location
WHERE bldg_code <> 'SPK';
```

A. IN Operator

The IN operator gives you the ability to allow for multiple conditions in one SQL statement. The list of values enclosed in the parentheses is called an *inlist*.

**Note:** the IN operator assumes that you are always including the OR comparison operator.

1. syntax: `variable in (value, value, value,...)`

Select the student’s last and first name and state of all the students that live in New Jersey, Pennsylvania, or New York.

If you were to write this query using a typical comparison operator, it would appear as:

```
SELECT slname, sfname, sstate
FROM student
WHERE sstate = 'NY' or sstate = 'NJ' or sstate = 'PA';
```

Alternatively, you could use the in operator to write it as follows. This saves quite a bit of typing since you do not need to repeat the attribute name and the OR operator each time. In addition, it adds clarity for others looking at your code.

```
SELECT slname, sfname, sstate
FROM student
WHERE sstate in ('NY', 'NJ', 'PA');
```
B. BETWEEN Operator

The BETWEEN operator gives the user the ability to look for something BETWEEN two values. The between can be used in lieu of creating a query that looks for records with a value greater than X and less than Y, inclusive. For instance, earlier in this section we explored how to write this query:

Show the room numbers and capacity of the room of all rooms that have a capacity greater than or equal to 40 and less than or equal to 70.

This same query can be rewritten using the BETWEEN operator as shown below.

1. Syntax: variable between value and value

<table>
<thead>
<tr>
<th>ROOM</th>
<th>CAPACITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>202</td>
<td>40</td>
</tr>
<tr>
<td>105</td>
<td>42</td>
</tr>
<tr>
<td>211</td>
<td>55</td>
</tr>
</tbody>
</table>

Earlier we saw that this query could be written as:

SELECT Room, capacity
FROM location
WHERE capacity >= 40 and capacity <= 70;

Using the BETWEEN operator, we can write the query as:

SELECT Room, capacity
FROM location
WHERE capacity between 40 and 70;
XI. **Logical Operators**

Logical operators test for the truth of a condition. You probably remember Venn Diagrams. The diagram below shows 2 groups: those people who live in Pennsylvania and those people who attend Temple. The section in the middle, the intersection, represents the population that satisfies BOTH conditions. In other words, that group lives in Pennsylvania AND attends Temple.

Conversely, the union of PA and Temple is the entire diagram. This means that the person either lives in PA OR attends Temple.

A. With the AND operator, the output displays values that match both conditions in the query. For instance, the condition meets both PA and Temple

```
<table>
<thead>
<tr>
<th>Room</th>
<th>BldgCode</th>
<th>Capacity</th>
</tr>
</thead>
<tbody>
<tr>
<td>105</td>
<td>BUS</td>
<td>42</td>
</tr>
<tr>
<td>211</td>
<td>BUS</td>
<td>55</td>
</tr>
</tbody>
</table>
```

As you can see above, both rows have a building code of BUS and have a capacity greater than or equal to 40.

B. With the OR operator only one of the two conditions needs to be true for the entire condition to be true.

```
<table>
<thead>
<tr>
<th>Room</th>
<th>BldgCode</th>
<th>Capacity</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>SP</td>
<td>150</td>
</tr>
<tr>
<td>202</td>
<td>SP</td>
<td>40</td>
</tr>
<tr>
<td>105</td>
<td>BUS</td>
<td>42</td>
</tr>
<tr>
<td>404</td>
<td>BUS</td>
<td>35</td>
</tr>
<tr>
<td>421</td>
<td>BUS</td>
<td>35</td>
</tr>
<tr>
<td>211</td>
<td>BUS</td>
<td>55</td>
</tr>
<tr>
<td>424</td>
<td>BUS</td>
<td>1</td>
</tr>
<tr>
<td>402</td>
<td>BUS</td>
<td>1</td>
</tr>
<tr>
<td>433</td>
<td>BUS</td>
<td>1</td>
</tr>
</tbody>
</table>
```
SQL Tutorial

Here you can see that because only one condition needed to be satisfied, more rows met the criteria including rows that were in SP if the capacity was at least 40.

C. Multiple logical operators

A query can include multiple logical conditions. In each case, the syntax must appear as:

Variable comparison operator value logical operator….

| List the ids, names, and student class of students who are freshman, junior or sophomores. |
| Select sid, slname, sfname, sclass From student WHERE sclass = ‘FR’ OR sclass = ‘JR’ OR sclass = ‘SO’; |
| List the ids, names, and student class of students who are sophomores or freshman and live in Idaho. An explanation of why I’ve added the parentheses is coming up next – stay tuned… |
| SELECT sid, sfname, slname, sclass, sstate FROM student WHERE (sclass = 'SO' or sclass = 'FR') AND sstate = 'ID'; |

XII. NOT operator

Sometimes when we write a query we are trying to find values that do NOT satisfy a condition. For instance, you might want to write a query that looks for all students who attend Temple with the exception of those students living in 19111. Rather than try to figure out all of the possible zip codes that could exist with the exception of 19111, you can use the NOT operator. The not operator allows you to apply a condition to each row and the NOT essentially reverses (i.e. converts a true to false and vice versa) the outcome thus selecting only those rows that do not satisfy the condition.

A. Syntax: not (fieldname) operator value

| List the first and last name and class of every student who is not a senior. |
| SELECT sfname, slname, sclass FROM student WHERE NOT (sclass = 'SR'); |

As you can see, only the rows for those students who are not seniors returned.

XIII. IS NULL and IS NOT NULL Operator

A. IS NULL Operator

There are times when we need to find if a field has not been populated, in other words when the user left the field blank. To do this, we use the null operator. This operator checks to see if there is some entry stored in the field. It is important to note that a null value is not the same as a 0 (zero) or a space. A zero implies that someone actually typed in the number...
zero. Similarly, a space indicates someone entered hit the space key in the field and stored that as the value.

1. Syntax: `<fieldname> IS NULL

   NOTE: the syntax for null includes the word is. No other operator such as = or != will work.

```sql
SHOW all enrollment rows where a grade has not been entered.
Select * FROM enrollment
WHERE grade IS NULL;
```

In this case, every row in the enrollment table has a grade so no rows will return.

B. IS NOT NULL

The converse of the null operator is the not null operator. This operator looks to see that a field has been populated. Any field that has a zero, spaces, or any other value will return with this operator.

1. Syntax: `<fieldname> IS NOT NULL

```sql
SHOW all enrollment rows where a grade has not been entered.
Select * FROM enrollment
WHERE grade IS NOT NULL;
```

```
<table>
<thead>
<tr>
<th>SID</th>
<th>CSECID</th>
<th>NUMBER GRADE</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>1008</td>
<td>95</td>
</tr>
<tr>
<td>100</td>
<td>1005</td>
<td>96</td>
</tr>
<tr>
<td>100</td>
<td>1008</td>
<td>87</td>
</tr>
<tr>
<td>100</td>
<td>1005</td>
<td>88</td>
</tr>
<tr>
<td>101</td>
<td>1006</td>
<td>76</td>
</tr>
<tr>
<td>101</td>
<td>1005</td>
<td>84</td>
</tr>
<tr>
<td>101</td>
<td>1005</td>
<td>93</td>
</tr>
<tr>
<td>101</td>
<td>1008</td>
<td>85</td>
</tr>
<tr>
<td>102</td>
<td>1006</td>
<td>75</td>
</tr>
<tr>
<td>102</td>
<td>1011</td>
<td>66</td>
</tr>
<tr>
<td>102</td>
<td>1012</td>
<td>55</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SID</th>
<th>CSECID</th>
<th>NUMBER GRADE</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>1010</td>
<td>99</td>
</tr>
<tr>
<td>100</td>
<td>1008</td>
<td>87</td>
</tr>
<tr>
<td>100</td>
<td>1008</td>
<td>76</td>
</tr>
<tr>
<td>100</td>
<td>1008</td>
<td>77</td>
</tr>
<tr>
<td>104</td>
<td>1012</td>
<td>69</td>
</tr>
<tr>
<td>104</td>
<td>1010</td>
<td>32</td>
</tr>
<tr>
<td>105</td>
<td>1010</td>
<td>94</td>
</tr>
<tr>
<td>105</td>
<td>1008</td>
<td>89</td>
</tr>
</tbody>
</table>
```

20 rows selected.

XIV. **Precedence**

You might have noticed in the prior example that parentheses were added to the statement. Just as you learned in grade school mathematics, you can add parentheses to ensure that an equation is evaluated in the order you wish as opposed to the order of precedence.

Precedence is the order in which the equation is evaluated by default or in this case, how Oracle evaluates different operators in the same expression. When evaluating an expression containing multiple operators, Oracle evaluates operators with higher precedence before evaluating those with lower precedence. Oracle evaluates operators with equal precedence from left to right within a SQL statement. The table shows the operators from highest precedence to lowest. Conditions listed on the same line have the same precedence. As the table indicates, Oracle evaluates operators before conditions.
SQL Tutorial

A. Conditional Precedence

<table>
<thead>
<tr>
<th>Type of Condition</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL operators are evaluated before SQL conditions</td>
<td>See “Operator Precedence”</td>
</tr>
<tr>
<td>=, !=, &lt;, &gt;, &lt;=, &gt;=,</td>
<td>comparison</td>
</tr>
<tr>
<td>IS [NOT] NULL, LIKE, [NOT] BETWEEN, [NOT] IN, EXISTS, IS OF type</td>
<td>comparison</td>
</tr>
<tr>
<td>NOT</td>
<td>exponentiation, logical negation</td>
</tr>
<tr>
<td>AND</td>
<td>conjunction</td>
</tr>
<tr>
<td>OR</td>
<td>disjunction</td>
</tr>
</tbody>
</table>

Select all of rows in the location table for every room in the BUS or CR building with a capacity greater than 35:

```
SELECT * FROM location
WHERE bldg_code = 'BUS' OR bldg_code = 'CR' AND capacity > 35;
```

<table>
<thead>
<tr>
<th>LOCID</th>
<th>BLDG_CODE</th>
<th>ROOM</th>
<th>CAPACITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>BUS</td>
<td>105</td>
<td>42</td>
</tr>
<tr>
<td>17</td>
<td>BUS</td>
<td>211</td>
<td>55</td>
</tr>
<tr>
<td>18</td>
<td>BUS</td>
<td>424</td>
<td>1</td>
</tr>
<tr>
<td>19</td>
<td>BUS</td>
<td>402</td>
<td>1</td>
</tr>
<tr>
<td>20</td>
<td>BUS</td>
<td>433</td>
<td>1</td>
</tr>
</tbody>
</table>

Is the output to the right the correct response to the query? Why are rooms with capacity less than 35 present in the output? SQL first evaluates the AND condition (i.e. the condition BLDG_CODE = ’CR’ and CAPACITY > 35). What will be the result of these two conditions applied to the LOCATION table?

B. Parentheses

As you saw earlier, you can use parentheses in an expression to override operator precedence. Oracle evaluates expressions inside parentheses before evaluating those outside.

To ensure the previous query (Select all of rows in the location table for every room in the BUS or CR building with a capacity greater than 35) is evaluated properly, add parentheses.

```
SELECT * FROM location
WHERE (bldg_code= 'BUS'
    OR bldg_code= 'CR')
    AND CAPACITY > 35;
```

<table>
<thead>
<tr>
<th>LOCID</th>
<th>BLDG_CODE</th>
<th>ROOM</th>
<th>CAPACITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>BUS</td>
<td>105</td>
<td>42</td>
</tr>
<tr>
<td>17</td>
<td>BUS</td>
<td>211</td>
<td>55</td>
</tr>
</tbody>
</table>

XV. Sorting the Output – Order by

By now you might have noticed that there doesn’t seem to be any order to the results of the above queries. That’s because we haven’t told SQL in what order we want the rows to be returned. However, a user can specify exactly how the rows should be sorted using the ORDER BY operator.
SQL Tutorial

Syntax: Order by <fieldname> ASC (default) or DESC

<table>
<thead>
<tr>
<th>BLDG_CODE</th>
<th>ROOM</th>
<th>CAPACITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>SP</td>
<td>202</td>
<td>40</td>
</tr>
<tr>
<td>BUS</td>
<td>105</td>
<td>42</td>
</tr>
<tr>
<td>BUS</td>
<td>211</td>
<td>55</td>
</tr>
<tr>
<td>SP</td>
<td>101</td>
<td>150</td>
</tr>
</tbody>
</table>

Create a report of the building code, room number, and capacity for every room with a capacity greater than equal to 40, sorted in ascending order by capacity.

SELECT bldg_code, room, capacity
FROM location
WHERE capacity >= 40
ORDER BY capacity ASC;

A. The default order is ascending (ASC) so this does not need to be specified. You can change the order to descending by adding the keyword DESC after the column name in the ORDER by clause. You can sort the output on the basis of multiple columns which is called nesting the order by clause. There is no limit to the number of nests you can request.

1. Syntax:

   ORDER BY <fieldname> ASC (default) or DESC, <fieldname> ASC (default) or DESC,
   <fieldname> ASC (default) or DESC…

Change the prior example to sort the results in descending order by capacity and then ascending order by building code.

<table>
<thead>
<tr>
<th>BLDG_CODE</th>
<th>ROOM</th>
<th>CAPACITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>SP</td>
<td>101</td>
<td>150</td>
</tr>
<tr>
<td>BUS</td>
<td>211</td>
<td>55</td>
</tr>
<tr>
<td>BUS</td>
<td>105</td>
<td>42</td>
</tr>
<tr>
<td>SP</td>
<td>202</td>
<td>40</td>
</tr>
</tbody>
</table>

XVI. Alias Operator

A. Alias for attribute names:

By default, SQL will return an attribute’s name as the column heading for that attribute. For example, in the query above, the output shows the first column as BLDG_CODE. Although that is indeed the name of the attribute, it isn’t particularly user friendly on a report. We’d probably prefer to see something a bit more English-like, such as the word “Building Code”. Essentially, we’d like to make an alias for the attribute’s name.

To create the alias, enter the column name followed by AS and then the alias you wish to assign to the column. If the alias is to be more than one word with spaces in between, you must put the alias in quotes. If the alias does not have spaces, you may type it without quotes.

Syntax: <fieldname> as “alias name”

<table>
<thead>
<tr>
<th>Student First Name</th>
<th>Student Last Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brad</td>
<td>Pitt</td>
</tr>
<tr>
<td>Angelina</td>
<td>Jolie</td>
</tr>
<tr>
<td>Matt</td>
<td>Damon</td>
</tr>
<tr>
<td>George</td>
<td>Clooney</td>
</tr>
<tr>
<td>Orlando</td>
<td>Bloom</td>
</tr>
<tr>
<td>Johnny</td>
<td>Depp</td>
</tr>
</tbody>
</table>

List all of the students’ first and last names with the column heading of Student First Name and Student Last Name.

SELECT sfname AS “Student First Name”, slname AS “Student Last Name”
FROM student;
B. Alias for table name

We’ve seen that you can create an alias for a field. You can also create an alias for a table. But why would you want to do that? Some table names are really long. If we need to write the table name repeatedly in a query, it can get a little tiresome and also opens the doors for typos in our query. When you being joining tables and there are a number of tables, sometimes aliases save a bit of time.

Syntax: When aliases are used for table names, we do not include the AS like we do with aliases for attribute names. Just type the alias you want to use after the table’s actual name, separated by a space.

1. Syntax: <tablename> alias

<table>
<thead>
<tr>
<th>BLDG_CODE</th>
<th>ROOM</th>
<th>CAPACITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>SP</td>
<td>101</td>
<td>150</td>
</tr>
<tr>
<td>SP</td>
<td>202</td>
<td>40</td>
</tr>
<tr>
<td>BUS</td>
<td>105</td>
<td>42</td>
</tr>
<tr>
<td>BUS</td>
<td>211</td>
<td>55</td>
</tr>
</tbody>
</table>

Give the location table the alias of loc. For now, this won’t seem too useful but soon it will prove most helpful.

```
SELECT loc.bldg_code, loc.room, loc.capacity
FROM location loc
WHERE capacity >= 40;
```

2. Aliases are only valid within the query in which they are defined. In other words, you can’t refer to the location table as “loc” in another query unless you add the alias again.

XVII. Math in SQL

There are a wide range of math functions included in SQL ranging from basic mathematical functions to highly complex statistical functions. The most basic mathematical operations are: Addition (+), Subtraction (-), Multiplication (*), and Division. To write a query, use the fieldnames in lieu of the values.

In a prior example, we saw that you can double the capacity in a room by multiplying capacity by 2. Let’s say administration decides to redesign the rooms in Speakman Hall and increase the capacity of each room 4 fold. However, 10% of the room is dedicated to smart technology so that will reduce the capacity.

Show the current and new capacity for each room.

Take the current capacity * 4. Take that total and multiply it by 90% since only 90% of the room capacity is available.

```
SELECT room, capacity, (capacity * 4)* .90
FROM location
WHERE bldg_code= 'SPK';
```

We could improve the appearance of the output with some aliases.

```
SELECT room, capacity AS "Original Capacity", (capacity * 4)* .90 AS "New Capacity"
FROM location
WHERE bldg_code= 'SPK';
```

<table>
<thead>
<tr>
<th>ROOM</th>
<th>ORIGINAL CAPACITY</th>
<th>NEW CAPACITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>150</td>
<td>540</td>
</tr>
<tr>
<td>202</td>
<td>40</td>
<td>144</td>
</tr>
<tr>
<td>103</td>
<td>35</td>
<td>126</td>
</tr>
<tr>
<td>105</td>
<td>35</td>
<td>126</td>
</tr>
</tbody>
</table>
A. SQL Operator Precedence

SQL is a powerful language that allows the user to make exceptionally complex queries which include mathematical computations. When you were in grade school, you learned the Please Excuse My Dear Aunt Sally mnemonic aid which helped you to learn the order of precedence of mathematical functions if parentheses are not used. These same rules apply here with SQL.

The levels of precedence among SQL operators are shown below from high to low.

- Parentheses
- Exponents
- Multiplication and Division
- Addition and Subtraction

***Operators listed on the same line have the same precedence.

<table>
<thead>
<tr>
<th>Double the building capacity of room 101 in Speakman Hall and then divide by 2 + 3.</th>
</tr>
</thead>
<tbody>
<tr>
<td>The result of the calculation will be different according to whether and where you use parentheses.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>If we don't use any parentheses, the query will look as follows:</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT capacity * 2/2 +3</td>
</tr>
<tr>
<td>FROM location</td>
</tr>
<tr>
<td>WHERE bldg_code = 'SPK' AND room = 101;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CAPACITY*2/2+3</th>
</tr>
</thead>
<tbody>
<tr>
<td>153</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>If we use parentheses to be sure that the 2+3 gets evaluated first, the output is dramatically different. Addition has a lower order of precedence then multiplication or division. To force that part of the equation to be evaluated first, add parentheses.</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT capacity * 2/(2 +3)</td>
</tr>
<tr>
<td>FROM location</td>
</tr>
<tr>
<td>WHERE bldg_code = 'SPK' AND room = 101;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CAPACITY*2/(2+3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>60</td>
</tr>
</tbody>
</table>
Chapter 24: Set Operators

“Set operators combine the results of two component queries into a single result. Queries containing set operators are called compound queries.”

Consider the following tables:

<table>
<thead>
<tr>
<th>State:</th>
<th>State_Pivot</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>STATEID</strong></td>
<td><strong>STATENAME</strong></td>
</tr>
<tr>
<td>PA</td>
<td>Pennsylvania</td>
</tr>
<tr>
<td>NJ</td>
<td>New Jersey</td>
</tr>
<tr>
<td>DE</td>
<td>Delaware</td>
</tr>
<tr>
<td>NY</td>
<td>New York</td>
</tr>
<tr>
<td>VA</td>
<td>Virginia</td>
</tr>
</tbody>
</table>

I. **Union**

A. **UNION operator:**

   The Union operator takes the results of two individual SQL queries and combines them into an output which is displayed as a single table of all matching rows. The two queries must have the same number of columns and compatible data types in order for the union to be successful. Duplicate rows are automatically removed from the results of a union.

B. **Union All**

   The UNION ALL operator does not remove the duplicate rows in the output.

Show all the rows in the state and state_pivot tables. Remove duplicates.

```
SELECT * FROM state
UNION
SELECT * FROM state_pivot;
```
II. **Intersect**

The INTERSECT operator takes the results of two queries and returns only duplicated rows that appear in both result sets.

The INTERSECT operator does not distinguish between NULLs when removing duplicate rows.

<table>
<thead>
<tr>
<th>STATEID</th>
<th>STATENAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>PA</td>
<td>Pennsylvania</td>
</tr>
<tr>
<td>NJ</td>
<td>New Jersey</td>
</tr>
<tr>
<td>DE</td>
<td>Delaware</td>
</tr>
<tr>
<td>NY</td>
<td>New York</td>
</tr>
<tr>
<td>PA</td>
<td>Pennsylvania</td>
</tr>
</tbody>
</table>

Show the intersection of state and state_pivot tables. Remove duplicates.

```sql
SELECT * FROM state
INTERSECT
SELECT * FROM state_pivot;
```
Chapter 25: DML and Joins

The power of relational databases really comes from the ability to join tables. As we’ve seen, data is stored in a number of tables which are related to each other through some common pieces of information. Specifically an attribute in one table is repeated in another table as a foreign key. So far, we have only had to write queries from one table but that is clearly not the norm in a normalized database (no pun intended).

When we use joins in queries, we take advantage of the ‘controlled redundancy’ of relational database tables, by using the foreign key references. It should also become apparent why relational databases require referential integrity. If referential integrity is not enforced, we will not be able to join tables. In most instances, when we make a join between two tables, we join the foreign key from one table and the primary key of the corresponding table.

I. Inner Joins

An inner join is also known as an equijoin.

The general format of a SELECT statement with a join operation is:

```
SELECT <table1.column1,…table2.column,…>
FROM <table1, table2 >
WHERE <table1.joincolumn_name> = <table2.joincolumn_name>;
```

Rules for joins:

- the attributes must have the same data type.
- the attributes must be of the same size.
- the attributes do NOT need to have the same attribute name.

A. The join condition specifies attributes which are common between two tables. Essentially, the join condition will contain the foreign key reference in one table, and the corresponding primary key in the other table. So what does this mean? Let’s take a look at the data entered into our student and faculty tables.

Student Table:

<table>
<thead>
<tr>
<th>ID</th>
<th>SLNAME</th>
<th>SFNAME</th>
<th>SMI</th>
<th>SADD</th>
<th>SCITY</th>
<th>SSTATE</th>
<th>SZIP</th>
<th>SPHONE</th>
<th>SCLASS</th>
<th>SDOB</th>
<th>SPIN</th>
<th>FID</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>Pitt</td>
<td>Brad</td>
<td>M</td>
<td>144 Winding Rd,</td>
<td>Cherry Hill</td>
<td>NJ</td>
<td>54703</td>
<td>7155558876</td>
<td>SR</td>
<td>7/14/1978</td>
<td>8861</td>
<td>2</td>
</tr>
<tr>
<td>101</td>
<td>Julie</td>
<td>Angelina</td>
<td>D</td>
<td>454 St. John's Street</td>
<td>Wilmington</td>
<td>DE</td>
<td>54702</td>
<td>7155552345</td>
<td>SR</td>
<td>8/19/1979</td>
<td>1230</td>
<td>1</td>
</tr>
<tr>
<td>102</td>
<td>Damon</td>
<td>Matt</td>
<td>892 Circle Drive</td>
<td>Philadelphia</td>
<td>PA</td>
<td>54715</td>
<td>7155559697</td>
<td>JR</td>
<td>10/10/1977</td>
<td>1613</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>103</td>
<td>Cleeney</td>
<td>George</td>
<td>J</td>
<td>1716 Summit St</td>
<td>Eau Claire</td>
<td>NJ</td>
<td>54702</td>
<td>7155556902</td>
<td>SO</td>
<td>9/24/1978</td>
<td>1841</td>
<td>5</td>
</tr>
<tr>
<td>104</td>
<td>Blaun</td>
<td>Orlando</td>
<td>R</td>
<td>1880 Broad Street</td>
<td>Philadelphia</td>
<td>ID</td>
<td>54701</td>
<td>7155558899</td>
<td>SO</td>
<td>11/20/1977</td>
<td>4420</td>
<td>4</td>
</tr>
<tr>
<td>105</td>
<td>Depp</td>
<td>Johnny</td>
<td>S</td>
<td>1818 Silver Street</td>
<td>Elk Mound</td>
<td>ID</td>
<td>54712</td>
<td>7155554944</td>
<td>FR</td>
<td>12/4/1977</td>
<td>9188</td>
<td>3</td>
</tr>
</tbody>
</table>

Faculty Table:

<table>
<thead>
<tr>
<th>FID</th>
<th>FLNAME</th>
<th>FFNAME</th>
<th>FMI</th>
<th>LOGID</th>
<th>FPHONE</th>
<th>FRANK</th>
<th>FPIN</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Knightly</td>
<td>Keira</td>
<td>J</td>
<td>12</td>
<td>7155551234</td>
<td>ASSO</td>
<td>1181</td>
</tr>
<tr>
<td>2</td>
<td>Streep</td>
<td>Meryl</td>
<td>R</td>
<td>11</td>
<td>7155559087</td>
<td>FULL</td>
<td>1075</td>
</tr>
<tr>
<td>3</td>
<td>Redford</td>
<td>Robert</td>
<td>F</td>
<td>13</td>
<td>7155555412</td>
<td>ASST</td>
<td>8531</td>
</tr>
<tr>
<td>4</td>
<td>Newman</td>
<td>Paul</td>
<td>M</td>
<td>18</td>
<td>7155558409</td>
<td>INST</td>
<td>1690</td>
</tr>
<tr>
<td>5</td>
<td>Danch</td>
<td>Judith</td>
<td>E</td>
<td>22</td>
<td>7155558082</td>
<td>ASSO</td>
<td>9899</td>
</tr>
</tbody>
</table>

Brad Pitt’s faculty advisor (fid) is 2. As you can see, fid 2 in the faculty table corresponds to Meryl Streep. This means that the value of Student.fid is equal to Faculty.FID. When we write a join, we are literally saying to SQL that the value of one attribute is equal to the value of another attribute (foreign key = primary key).
SQL Tutorial

B. There is no limit to the number of tables that can be added to a query. To add a table to be the query:
   - Add the table to the FROM clause. You need to separate the tables with commas.
   - Add the join to the WHERE clause. Separate the joins with an AND.

C. Order in the WHERE Clause
SQL has no rules about order in the WHERE clause. This means:

1. It doesn’t matter if you put the foreign key first and then the primary key or the reverse in a join.
2. The order of the joins is irrelevant
3. The order of the conditions in the WHERE clause is irrelevant
4. You can mix joins and conditions in your WHERE clause.

D. Rule of Thumb - number of joins per query
There is typically 1 less join than there are tables. If you are including 3 tables in your query, you will probably have 2 joins.

E. Qualifying columns
We learned earlier that in order to accurately specify the column to which the table belongs the table name should be qualified through tablename.attribute name notation. For instance, if you want to add the student’s last name to a select query, you would write:

   SELECT Student.sname

While this is the proper notation, we typically just refer to the attribute name without the table name as a short cut. This has worked for us so far. Now however, we are going to learn to write queries that involve more than one table in a database. Accordingly, we will begin to encounter situations where an attribute name may appear in multiple tables. Remember, while an attribute name must be unique within a table, an attribute name can appear in different tables within the same database.

Why would we do that? Let’s consider the attribute First_Name. In our database we could have an attribute in the faculty table called First_Name which refers to the first name of the faculty member. Similarly, we could also have an attribute of First_Name in the student table for the student’s first name. If we write a query which includes both the student and faculty tables and we add First_Name in the SELECT clause, how would SQL know we want to include the first name of the student or the faculty member? This situation actually has a special name – natural join.

So when do you need to qualify an attribute? You need to qualify the attribute with its table name when:
   - You are writing a query that includes more than one table
   - The attribute name you are using is not unique in the tables used in the query

F. If you forget a join in a multi-table query, you will return every possible combination of rows in the tables included in the query. Whenever you return many more rows than you expect in a query which includes more than one table, check to see if you forgot a join.
List the student’s first, last name, and last name of the student’s faculty advisor.

In this query, we need to find information from two tables: the student table and the faculty table. To make the join, find an attribute that is common between the two tables. Since FID in the student key is a foreign key to the primary key of FID in the faculty table, we can join the tables.

Select student.sfname, student.slname, faculty.fname
FROM student, faculty
WHERE student.fid = faculty.FID;

G. Let’s change the previous query. In this case, we’d like to see the faculty advisor’s location added to the query.

To add the location, we need to add another table. Whenever we add a table to a query we need to do a few things:

1. If an attribute in the newly added table needs to appear in the output, add the attribute name in the appropriate location in the select clause. Remember to separate the attributes with commas. In this case, we will add bldg_code to the select clause.

2. Add the table name to the FROM clause. Remember to separate the table names with commas. In this case, we will add location to the from clause.

3. Add a join between the new table and its related table in the query. To add the join, make the foreign key in the first table equal to the primary key in the second table. Remember to add an AND to separate the joins and the conditions. In this case, we’ll add a join between faculty.locid (the foreign key) and location.locid (the primary key).

So to add the location to the above query, we’ll modify the query as follows:

4. Here’s another example with more tables. For each student, we want to list his first and last name as well as the name of courses in which he was enrolled and the grades he received.

In this query, we need information from 4 tables. Why 4?

- We need to retrieve the student’s first and last name from the student table.
- We find the student’s grade from the enrollment table. As we can see in the ERD, enrollment.sid is a foreign key to student.sid so we can join the student and the enrollment table.
- We get the name of the course in the course table. When we make a join, it must be between two tables that have a common value. But, there is no common attribute between the course table and the enrollment table. Why?
SQL Tutorial

Because there is a M:N relationship between course and enrollment which is resolved with the bridge entity called course_section.

NOTE: Often times you will need to navigate through a few tables to connect two that have the information you want. In this case, we need to navigate through the bridge entity to connect the enrollment and the course table.

If we follow the relationships between the foreign keys and the primary keys in the ERD below, it is easy to see how to make the joins.

SELECT sfname, slname, cname, grade
FROM student, enrollment, course, course_section
WHERE student.sid = enrollment.sid
AND enrollment.csecid = course_section.csecid
AND course.cid = course_section.cid;

<table>
<thead>
<tr>
<th>SFNAME</th>
<th>SLNAME</th>
<th>CNAME</th>
<th>GRADE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brad</td>
<td>Pitt</td>
<td>DBMS</td>
<td>A</td>
</tr>
<tr>
<td>Brad</td>
<td>Pitt</td>
<td>Principles of MIS</td>
<td>A</td>
</tr>
<tr>
<td>Brad</td>
<td>Pitt</td>
<td>Database Management</td>
<td>B</td>
</tr>
<tr>
<td>Brad</td>
<td>Pitt</td>
<td>Project Management</td>
<td>B</td>
</tr>
<tr>
<td>Angelina</td>
<td>Jolie</td>
<td>DBMS</td>
<td>C</td>
</tr>
<tr>
<td>Angelina</td>
<td>Jolie</td>
<td>Principles of MIS</td>
<td>B</td>
</tr>
<tr>
<td>Angelina</td>
<td>Jolie</td>
<td>Database Management</td>
<td>A</td>
</tr>
<tr>
<td>Angelina</td>
<td>Jolie</td>
<td>Project Management</td>
<td>B</td>
</tr>
<tr>
<td>Matt</td>
<td>Damon</td>
<td>DBMS</td>
<td>C</td>
</tr>
<tr>
<td>Matt</td>
<td>Damon</td>
<td>Principles of MIS</td>
<td>D</td>
</tr>
<tr>
<td>Matt</td>
<td>Damon</td>
<td>Database Management</td>
<td>F</td>
</tr>
<tr>
<td>George</td>
<td>Cleavage</td>
<td>DBMS</td>
<td>B</td>
</tr>
<tr>
<td>George</td>
<td>Cleavage</td>
<td>Principles of MIS</td>
<td>A</td>
</tr>
<tr>
<td>Orlando</td>
<td>Bloom</td>
<td>DBMS</td>
<td>B</td>
</tr>
<tr>
<td>Orlando</td>
<td>Bloom</td>
<td>Principles of MIS</td>
<td>C</td>
</tr>
<tr>
<td>Orlando</td>
<td>Bloom</td>
<td>Project Management</td>
<td>C</td>
</tr>
<tr>
<td>Orlando</td>
<td>Bloom</td>
<td>Database Management</td>
<td>D</td>
</tr>
<tr>
<td>Orlando</td>
<td>Bloom</td>
<td>DBMS</td>
<td>F</td>
</tr>
<tr>
<td>Johnny</td>
<td>Depp</td>
<td>DBMS</td>
<td>A</td>
</tr>
<tr>
<td>Johnny</td>
<td>Depp</td>
<td>Principles of MIS</td>
<td>B</td>
</tr>
</tbody>
</table>
H. One last example with conditions and joins. Now let's say we want to see the same information but only if the student's grade is a B or an A. All we have to do to write this query is add the condition to the WHERE clause as follows:

```
SELECT sfname, slname, cname, grade
FROM student, enrollment, course, course_section
WHERE student.sid = enrollment.sid
  AND enrollment.csecid = course_section.csecid
  AND course.cid = course_section.cid
  AND (grade = 'B' OR grade = 'A');
```

II. **Outer Joins**

Sometimes we want to find rows in one table that do not have corresponding rows in another table. Consider this situation: A student hasn't been assigned an advisor yet. In addition, an advisor is teaching abroad and has no students to advise this semester. If we write an inner join
SQL Tutorial

that shows students and their associated advisors, these two rows will be excluded. Fortunately, we have a slick little feature called an outer join which is designed to solve just this dilemma.

The general format of a SELECT statement with an outer join operation is identical to an inner join except that a plus sign in parentheses (+) is placed after the optional attribute in the join.¹¹

```
SELECT <table1.column1,…table2.column,…>
FROM <table1, table2>
WHERE <table1.joincolumn_name> = <table2.joincolumn_name>(+);
```

Just like in an inner join:

- the attributes must have the same data type.
- the attributes must be of the same size.
- the attributes do NOT need to have the same attribute name.

<table>
<thead>
<tr>
<th>SFNAME</th>
<th>SLNAME</th>
<th>FNAME</th>
<th>FLNAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brad</td>
<td>Pitt</td>
<td>Meryl</td>
<td>Streep</td>
</tr>
<tr>
<td>Angelina</td>
<td>Jolie</td>
<td>Keira</td>
<td>Knightly</td>
</tr>
<tr>
<td>Matt</td>
<td>Damon</td>
<td>Keira</td>
<td>Knightly</td>
</tr>
<tr>
<td>George</td>
<td>Clooney</td>
<td>Judith</td>
<td>Dench</td>
</tr>
<tr>
<td>Orlando</td>
<td>Bloom</td>
<td>Paul</td>
<td>Newman</td>
</tr>
<tr>
<td>Johnny</td>
<td>Depp</td>
<td>Robert</td>
<td>Redford</td>
</tr>
<tr>
<td>Antonio</td>
<td>Banderas</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

As you can see, Anthonio Banderas has no faculty advisor.

<table>
<thead>
<tr>
<th>SFNAME</th>
<th>SLNAME</th>
<th>FNAME</th>
<th>FLNAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>Angelina</td>
<td>Jolie</td>
<td>Keira</td>
<td>Knightly</td>
</tr>
<tr>
<td>Matt</td>
<td>Damon</td>
<td>Keira</td>
<td>Knightly</td>
</tr>
<tr>
<td>Brad</td>
<td>Pitt</td>
<td>Meryl</td>
<td>Streep</td>
</tr>
<tr>
<td>Johnny</td>
<td>Depp</td>
<td>Robert</td>
<td>Redford</td>
</tr>
<tr>
<td>Orlando</td>
<td>Bloom</td>
<td>Paul</td>
<td>Newman</td>
</tr>
<tr>
<td>George</td>
<td>Clooney</td>
<td>Judith</td>
<td>Dench</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Juliette</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Binoche</td>
</tr>
</tbody>
</table>

You can see here that Juliette Binoche doesn’t have any students assigned to her.

III. ANSI standard format for Joins

The syntax that we’ve used to develop joins in which you specify the tables in the FROM clause and then join in the WHERE clause is the traditional format for joins. However, this syntax is not ANSI (American National Standard Institute) compliant. Although Oracle and Access still accept it, you may run into some issues if you try to port your code to other DBMS such as SQL Server.

¹¹ http://oreilly.com/catalog/orsqlpluspr2/chapter/ch01.html
SQL Tutorial

In general, this format seems to run more easily if you use table name aliases.

Of note, according to some comments on some database developer sites, the new ANSI standard format for joins can improve the performance of queries.

A. INNER JOIN

If you need to specify the joins more clearly, you can use the following syntax:

```sql
SELECT <table1.column1,…table2.column,…> 
FROM <table1> 
INNER JOIN <table2> ON <table1.column> = <table2.column> … ;
```

<table>
<thead>
<tr>
<th>Desired Query</th>
<th>Traditional Join</th>
<th>ANSI Join</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Show each student’s name, grade and course section number.</td>
<td><strong>SELECT</strong> sfname, slname, fnname, grade, csecid <strong>FROM</strong> student, faculty, enrollment <strong>WHERE</strong> faculty.fid = student.fid <strong>AND</strong> enrollment.sid = student.sid <strong>AND</strong> student.sid = enrollment.sid;</td>
<td><strong>SELECT</strong> sfname, slname, fnname, grade, csecid <strong>FROM</strong> student INNER JOIN faculty ON faculty.fid = student.fid INNER JOIN enrollment ON enrollment.sid = student.sid;</td>
<td>sfname slname fnname grade csecid</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Brad Pitt Streep A 1000</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Brad Pitt Streep A 1003</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Brad Pitt Streep B 1005</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Brad Pitt Streep B 1008</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Angelina Jolie Knightly C 1000</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Angelina Jolie Knightly B 1004</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Angelina Jolie Knightly A 1005</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Angelina Jolie Knightly B 1008</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Matt Damon Knightly C 1000</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Matt Damon Knightly D 1011</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Matt Damon Knightly F 1012</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>George Clooney Dench A 1010</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>George Clooney Dench A 1011</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Orlando Bloom Newman B 1000</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Orlando Bloom Newman C 1004</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Orlando Bloom Newman C 1008</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Orlando Bloom Newman D 1012</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Orlando Bloom Newman F 1010</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Johnny Dapp Redford A 1010</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Johnny Dapp Redford B 1011</td>
</tr>
<tr>
<td>Show each student’s name, grade and course section number if the student received an A.</td>
<td><strong>SELECT</strong> sfname, slname, fnname, grade, csecid <strong>FROM</strong> student, faculty, enrollment <strong>WHERE</strong> faculty.fid = student.fid <strong>AND</strong> enrollment.sid = student.sid <strong>AND</strong> student.sid = enrollment.sid;</td>
<td><strong>SELECT</strong> sfname, slname, fnname, grade, csecid <strong>FROM</strong> student INNER JOIN faculty ON faculty.fid = student.fid INNER JOIN enrollment ON enrollment.sid = student.sid WHERE grade = 'A';</td>
<td>sfname slname fnname grade csecid</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Brad Pitt Streep A 1000</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Brad Pitt Streep A 1003</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Angelina Jolie Knightly A 1005</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>George Clooney Dench A 1011</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Johnny Dapp Redford A 1010</td>
</tr>
</tbody>
</table>

C J Marselis
### B. OUTER JOINS (Right and Left)

<table>
<thead>
<tr>
<th>Desired Query</th>
<th>Traditional Join</th>
<th>ANSI Join</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Left Outer Join:</strong> List the student’s first and last name and the advisor’s first and last name regardless if the student has a faculty advisor</td>
<td><code>SELECT sfname, slname, ffname, flname FROM student, faculty WHERE student.fid = faculty.fid(+);</code></td>
<td><code>SELECT sfname, slname, ffname, flname FROM student LEFT OUTER JOIN faculty ON (student.fid = faculty.fid);</code></td>
<td></td>
</tr>
<tr>
<td><strong>Result</strong></td>
<td></td>
<td></td>
<td>SFNAME</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Matt</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Angelina</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Brad</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Johnny</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Orlando</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>George</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Antonio</td>
</tr>
</tbody>
</table>

**Right Outer Join:** Show faculty members and their associated students.

<table>
<thead>
<tr>
<th>Desired Query</th>
<th>Traditional Join</th>
<th>ANSI Join</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Right Outer Join:</strong> Show faculty members and their associated students.</td>
<td><code>SELECT sfname, ffname, slname, ffname FROM student, faculty WHERE student.fid(+) = faculty.fid;</code></td>
<td><code>SELECT sfname, ffname, slname, ffname FROM student RIGHT OUTER JOIN faculty ON (student.fid = faculty.fid);</code></td>
<td></td>
</tr>
<tr>
<td><strong>Result</strong></td>
<td></td>
<td></td>
<td>SFNAME</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Brad</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Angelina</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Matt</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>George</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Orlando</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Johnny</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Antonio</td>
</tr>
</tbody>
</table>

Notice Antonio Banderas does not appear as this output since he has no faculty member who whom he matches. Similarly, Juliette Binoche does not appear on the example above as she has no student with whom she matches.

### C. FULL OUTER JOIN

What if you want to create a query which show rows which match, regardless if it is a left or right outer join. There is no way to do this with a traditional join except with a UNION. However, in ANSI standard join, you can perform a FULL OUTER JOIN.

<table>
<thead>
<tr>
<th>Desired Query</th>
<th>Traditional Join</th>
<th>ANSI Join</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Full Outer Join:</strong> List the student’s first and last name and the advisor’s first and last name regardless if the student has a faculty advisor or the faculty has students assigned.</td>
<td><code>SELECT sfname, slname, ffname, flname FROM student, faculty WHERE student.fid = faculty.fid(+) UNION SELECT sfname, slname, ffname, flname FROM student, faculty WHERE student.fid(+) = faculty.fid;</code></td>
<td><code>SELECT sfname, slname, ffname, flname FROM student FULL OUTER JOIN faculty ON (student.fid = faculty.fid);</code></td>
<td></td>
</tr>
<tr>
<td><strong>Result</strong></td>
<td></td>
<td></td>
<td>SFNAME</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Angelina</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Antonio</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Brad</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>George</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Johnny</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Orlando</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Juliette</td>
</tr>
</tbody>
</table>
Chapter 26: DML - Update and Delete Records

I. Update Statement

We’ve seen that we can change a table’s structure using the alter statement in DDL. But what if we want to change the data entered into a row? The UPDATE statement allows you to modify the data which was entered into a field.

An UPDATE statement changes the data while the ALTER statement changes the structure of the table. The UPDATE statement changes the values of single rows, groups of rows, or all the rows in a table.

A. Syntax:

```
UPDATE tablename
SET attribute = X
WHERE attribute = X;
```

University 1 is being relocated to Mars. Update the city information accordingly.

```
UPDATE university
SET university_city = 'Mars'
WHERE university_id = 1;
```

B. An UPDATE statement with no WHERE clause

If you have an UPDATE statement with no where clause, the update will affect all rows in the table being modified. Accordingly, unless you truly want to modify the entire table it is essential to consider the condition or filter. The WHERE clause specifies which rows are to be updated.
Change the University city to be blank for all universities.

UPDATE university
SET university_city = NULL

You can also use computed column values in an update. This example doubles all prices in the titles table:

UPDATE book
SET price = price * 2

II. DELETE Statement

We’ve seen that we can delete a table using the DROP table statement in DDL. But what if we want to delete a row in a table? The DROP table statement removes the entire table from the database. Conversely, the DELETE statement removes rows from a table. The DELETE statement may delete single rows, groups of rows, or all the rows in a table.

A. Syntax:

DELETE
FROM <tablename>
WHERE attribute = X;
SQL Tutorial

Delete the Willow Grove row from the city table

DELETE FROM city
WHERE city_id = 'WG';

Original data:
<table>
<thead>
<tr>
<th>CITY_ID</th>
<th>CITY_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>PH</td>
<td>Philadelphia</td>
</tr>
<tr>
<td>HV</td>
<td>Huntingdon Valley</td>
</tr>
<tr>
<td>WG</td>
<td>Willow Grove</td>
</tr>
<tr>
<td>NY</td>
<td>New York</td>
</tr>
</tbody>
</table>

Revised data:
<table>
<thead>
<tr>
<th>CITY_ID</th>
<th>CITY_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>PH</td>
<td>Philadelphia</td>
</tr>
<tr>
<td>HV</td>
<td>Huntingdon Valley</td>
</tr>
<tr>
<td>NY</td>
<td>New York</td>
</tr>
</tbody>
</table>

B. DELETE statement with no WHERE clause

Just like in the update statement, if you have a DELETE statement with no WHERE clause, you will delete all rows in the table. Accordingly, unless you truly want to delete all the rows in the table, it is essential to consider the condition or filter. The WHERE clause specifies which rows are to be deleted.

Remember a DELETE statement only affects the rows in a table. Even if you delete all the rows, the table structure still exists. In other words, the table is still there but there will be no rows in the table. To delete the table structure, you must use the DROP TABLE DDL command.

Delete the rows from the city table

DELETE FROM city;

Original data:
<table>
<thead>
<tr>
<th>CITY_ID</th>
<th>CITY_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>PH</td>
<td>Philadelphia</td>
</tr>
<tr>
<td>HV</td>
<td>Huntingdon Valley</td>
</tr>
<tr>
<td>WG</td>
<td>Willow Grove</td>
</tr>
<tr>
<td>NY</td>
<td>New York</td>
</tr>
</tbody>
</table>

Revised data:
<table>
<thead>
<tr>
<th>CITY_ID</th>
<th>CITY_NAME</th>
</tr>
</thead>
</table>

Delete the entire city table:

DROP TABLE city CASCADE CONSTRAINTS.
C. Deleting Foreign Primary Key with Foreign Key

There may be an occasion when the following error will appear when attempting to delete a row.

![TOAD Error]

This message is indicating that the row cannot be deleted as it is a foreign key to a row in another table. SQL will not allow the rows to be deleted as it will result in an integrity error.

In order to delete the rows, you must first delete the corresponding row in the associated table. Then you can delete the original row.
Chapter 27: DML- Date Functions

I. Date Background Information:

The date data type in Oracle comes with a powerful set of operators and functions to manipulate the date field. Within the database, the date data is stored in a special format that includes date (day, month, and year) and the time (hours, minutes and seconds). In fact the date value is stored in seven bytes, one each for century, year, month, day, hours, minutes, and seconds. Some of the commonly used format masks are described below:

<table>
<thead>
<tr>
<th>Format</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>MM</td>
<td>Month number</td>
<td>3</td>
</tr>
<tr>
<td>MON</td>
<td>3-letter abbreviation of month</td>
<td>JAN</td>
</tr>
<tr>
<td>MONTH</td>
<td>Fully spelled out month</td>
<td>MARCH</td>
</tr>
<tr>
<td>D</td>
<td>Number of days in the week</td>
<td>3</td>
</tr>
<tr>
<td>DD</td>
<td>Number of days in the month</td>
<td>16</td>
</tr>
<tr>
<td>DDD</td>
<td>Number of days in the year</td>
<td>234</td>
</tr>
<tr>
<td>DY</td>
<td>3-letter abbreviation of day of week</td>
<td>WED</td>
</tr>
<tr>
<td>DAY</td>
<td>Fully spelled-out day of the week</td>
<td>FRIDAY</td>
</tr>
<tr>
<td>Y</td>
<td>Last digit of year</td>
<td>8</td>
</tr>
<tr>
<td>YY</td>
<td>Last two digits of year</td>
<td>00</td>
</tr>
<tr>
<td>YYYY</td>
<td>Last three digits of year</td>
<td>000</td>
</tr>
<tr>
<td>YYYYY</td>
<td>Full four-digit year</td>
<td>2000</td>
</tr>
<tr>
<td>HH12</td>
<td>Hours of the day (1 to 12)</td>
<td>9</td>
</tr>
<tr>
<td>HH24</td>
<td>Hours of the day (0-23)</td>
<td>18</td>
</tr>
<tr>
<td>MI</td>
<td>Minutes of the hour</td>
<td>34</td>
</tr>
<tr>
<td>SS</td>
<td>Seconds of minute</td>
<td>29</td>
</tr>
<tr>
<td>AM</td>
<td>Display AM or PM depending on time</td>
<td>PM</td>
</tr>
</tbody>
</table>

A. How do Computers Understand Time?

Sometimes, the easiest way to understand time with computers is to consider how dates are handled in Microsoft Excel. Microsoft selected an arbitrary start date. Essentially this is just a point in time which happens to be January 1, 1900. This date is assigned number 1. Each day after the start date is assigned a sequential number in increments of 1. In other words, 1/2/1900 is equal to 2, 1/3/1900 is equal to 3, 1/4/1900 is equal to 4, etc.

This may seem complicated but it makes perfect sense when you consider that computers can be really handy at calculating the difference between two numbers. If you can assign each date a number, you can calculate how much time has elapsed between dates. For instance, let’s say I bought something on July 12, 2008 (which corresponds to 39641) and the company wants to invoice me in 30 days. The developer can easily write some code that says invoice date = purchase date + 30. In this case, I should be invoiced on 39671 (8/11/2008).

The dates that we store, such as May 1, 2008, really don’t mean anything to the computer until we convert them into something that is meaningful such as this start date in Excel. Frankly, the computer sees May 1, 2008 or 5/1/2008 as just another list of characters, or string. This is similar to any other word we might write such as a first name, city, or company name. As you know, you can’t do math on strings. For instance, I can’t add 30 to my name.
or the city I live in or to Temple University. How then can we do math with a date like May 1, 2008 so we can tell when it is time to invoice someone? The good folks who brought us SQL have thought up nifty ways so that we can tell the computer that this string is actually a date.

II. Sysdate

One of the most important concepts in Oracle in terms of dates and time is the function sysdate. Sysdate (aka system date) is a function which comes predefined in Oracle and it tells the current date and time stored in the computer. If you know the current date stored in SYSDATE, you can use it to calculate the amount of time that has elapsed between a given date and the current date.

A. Calculations using sysdate.

Since you are now a database guru, you are well aware that in good database design we avoid storing attributes that are derived or calculated fields. As such, we don’t store a person’s age in a database since this should actually be calculated based on the person’s date of birth. So how do we actually do this?

If someone tells you his birth date you could quickly calculate how old he is by subtracting the date from today’s date. That is precisely how we do it in Oracle. We can always tell what the current date and time is with sysdate. So, if we subtract the person’s birthdate from SYSDATE, it will return the person’s age.

<table>
<thead>
<tr>
<th>SFNAME</th>
<th>SLNAME</th>
<th>SYSDATE - SDOB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brad</td>
<td>Pitt</td>
<td>10548.8283564815</td>
</tr>
<tr>
<td>Angelina</td>
<td>Jolie</td>
<td>10512.8283564815</td>
</tr>
<tr>
<td>Matt</td>
<td>Damon</td>
<td>11190.8283564815</td>
</tr>
<tr>
<td>George</td>
<td>Clooney</td>
<td>10841.8283564815</td>
</tr>
<tr>
<td>Orlando</td>
<td>Bloom</td>
<td>11149.8283564815</td>
</tr>
<tr>
<td>Johnny</td>
<td>Depp</td>
<td>11135.8283564815</td>
</tr>
</tbody>
</table>

What the heck? Is Brad Pitt really 10,548 plus years old? Something must be wrong. Let’s add some attributes to our query to try and understand what’s going on. Let’s add the student’s date of birth and the sysdate to the query.

<table>
<thead>
<tr>
<th>SFNAME</th>
<th>SLNAME</th>
<th>SYSDATE</th>
<th>DOB</th>
<th>SYSDATE - DOB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brad</td>
<td>Pitt</td>
<td>5/30/2008 7:55:55 PM</td>
<td>7/14/1979</td>
<td>10548.8304976852</td>
</tr>
<tr>
<td>Angelina</td>
<td>Jolie</td>
<td>5/30/2008 7:55:55 PM</td>
<td>8/19/1979</td>
<td>10512.8304976852</td>
</tr>
<tr>
<td>Matt</td>
<td>Damon</td>
<td>5/30/2008 7:55:55 PM</td>
<td>10/10/1977</td>
<td>11190.8304976852</td>
</tr>
</tbody>
</table>

OK so those dates of birth seem reasonable and the system date looks valid so why is the calculation so screwy? The default result in Oracle in a date calculation is in days. Basically Oracle is returning how old the students are in days. We can easily convert days to years by dividing the number by 365.25 (for leap year) days/years as follows. We might also add an alias of age for the column heading for the calculation. Remember, because of My Dear Aunt Sally order of precedence, you really need to make sure to add the parentheses or the student’s age will be really weird.
SELECT sfname, slname, SYSDATE, sDOB, (SYSDATE - sDOB)/365.25 AS "Age" FROM student;

<table>
<thead>
<tr>
<th>SF NAME</th>
<th>SL NAME</th>
<th>SYSDATE</th>
<th>SDOB</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brad</td>
<td>Pitt</td>
<td>5/30/2008 8:00:59 PM</td>
<td>7/14/1979</td>
<td>28.8811335145692</td>
</tr>
<tr>
<td>Angelina</td>
<td>Jolie</td>
<td>5/30/2008 8:00:59 PM</td>
<td>8/19/1979</td>
<td>28.7825708862524</td>
</tr>
<tr>
<td>Matt</td>
<td>Damon</td>
<td>5/30/2008 8:00:59 PM</td>
<td>10/10/1977</td>
<td>30.638833719528</td>
</tr>
<tr>
<td>George</td>
<td>Clooney</td>
<td>5/30/2008 8:00:59 PM</td>
<td>9/24/1978</td>
<td>29.68323237952189</td>
</tr>
<tr>
<td>Orlando</td>
<td>Bloom</td>
<td>5/30/2008 8:00:59 PM</td>
<td>11/20/1977</td>
<td>30.5266818376566</td>
</tr>
<tr>
<td>Johnny</td>
<td>Depp</td>
<td>5/30/2008 8:00:59 PM</td>
<td>12/4/1977</td>
<td>30.4882519266357</td>
</tr>
</tbody>
</table>
Chapter 28: Functions in SQL

I. Basics of functions

SQL has a wide range of built-in functions that help database folks do our daily jobs. If a function isn’t immediately available, it is always advisable to surf the net. Developers are a generous sort and typically like to share useful functions they have created. This makes your life much easier. If you can’t find a function, you can always write one yourself and post it for others!

In terms of the last example with age, the output looked much better when we gave the column heading an alias and presented age in years as opposed to days. But, but do we really want all those decimal places after the number of years? Honestly, don’t we want to round down rather than up when it comes to age?!!! Lucky for us, SQL has some useful mathematical functions that can help here.

There are a few things to understand when working with functions.

- A function typically takes an argument. An argument is a piece of information you “pass” to the function. This will make more sense with our first example.
- The parameters passed to the function are enclosed in parentheses.

II. Common Functions

A. TRUNC (n, precision)

Trunc takes a number (n) and a level of precision (number of digits after the decimal point) as arguments and returns only the truncated portion of the number.

1. For instance, in the last example, we don’t want to round the students’ ages up or down. We just want to only show the age as a whole number. To do this:
   - Truncate the age (age is the number here - the n – which is the first argument for the function).
   - The level of precision is 0 (the second argument in the function) since we want no digits after the decimal place.

   ```sql
   SELECT sfname, slname, SYSDATE, sdob, TRUNC((SYSDATE - sdob)/365.25, 0) AS "Age"
   FROM student;
   ```

B. ROUND (n, precision)

The round function takes a number and the desired level of precision (digits after the decimal point) as arguments and returns the number rounded up or down.
Show the student’s last name with the column heading of “Student Last Name” and the student’s age with the column heading of age. Round age to 2 decimal places.

```
SELECT sfname, slname, SYSDATE, sdob, ROUND((SYSDATE - sdob)/365.25, 2) AS "Age"
FROM student;
```

### III. Date Functions

#### A. TO_DATE(s, format)

As we’ve been learning, the TO_DATE is a critical function in SQL. The to_date takes a string (s) and a format for the string and returns a date corresponding to the string s based on the format. This means a user can enter a string format for a date such as May 31, 2008 and it will be converted to a date that can be used in calculations. This function allows users to enter dates in English-like and readable formats without sacrificing the ability to do calculations on the dates as if they were numbers.

The format argument is essential to this function. This argument tells SQL in what format the data was stored in a field. For instance, was it MM/DD/YYYY or DD/MM/YYYY.

In this example, we are taking two strings and converting them to dates. Once we do this, we can look for values between the dates.

```
List the students and their birthdays if the student was born during October of 1977
SELECT sid, slname, sfname, SDOB
FROM student
WHERE SDOB BETWEEN TO_DATE('10/01/1977', 'MM/DD/YYYY') AND TO_DATE('10/31/1977', 'MM/DD/YYYY');
```

#### B. ADD_MONTHS (d, count)

This function takes a date d and a number count as inputs and return a new date that is count months after d.

```
List the students and their birthdays if the student was born within a month of October 1, 1977
SELECT sid, slname, sfname, SDOB
FROM student
WHERE SDOB BETWEEN ADD_MONTHS(TO_DATE('10/01/1977', 'MM/DD/YYYY'), -1) AND ADD_MONTHS(TO_DATE('10/01/1977', 'MM/DD/YYYY'), +1);
```
C. Other Useful Date Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NEXT_DAY(d,day))</td>
<td>Function takes as input a date d and a string day that represents a day of</td>
</tr>
<tr>
<td></td>
<td>the week, and returns as output the next date after d whose day of the week</td>
</tr>
<tr>
<td></td>
<td>is the same as day.</td>
</tr>
<tr>
<td>LAST_DAY(d)</td>
<td>Function takes as input a date d and returns the date corresponding to the</td>
</tr>
<tr>
<td></td>
<td>last day of the month in which d belongs.</td>
</tr>
<tr>
<td>LEAST(d1, d2, .....dn)</td>
<td>Given a list of dates, this function returns the earliest date.</td>
</tr>
<tr>
<td>GREATEST(d1, d2,.....dn)</td>
<td>Given a list of dates, this function returns the latest date.</td>
</tr>
<tr>
<td>TRUNC(d)</td>
<td>Given date d, this function returns the same date with time reset to 12:00A</td>
</tr>
<tr>
<td></td>
<td>M midnight.</td>
</tr>
<tr>
<td>ROUND(d)</td>
<td>Given date d, this function returns the same date with time reset to 12:00A</td>
</tr>
<tr>
<td></td>
<td>M midnight if d is before noon and rounds the date up to the next day (again</td>
</tr>
<tr>
<td></td>
<td>12:00 A.M. midnight) if d is after noon.</td>
</tr>
<tr>
<td>TO_CHAR(d, format)</td>
<td>Given date d and a format as a string, this function returns a character</td>
</tr>
<tr>
<td></td>
<td>string equivalent of the date based on the format. The contents of the</td>
</tr>
<tr>
<td></td>
<td>format are shown below.</td>
</tr>
</tbody>
</table>

IV. String Functions

This is just a subset of some of the common string functions in SQL. These functions do not actually change the way that the value is stored in the field. Rather, the functions just change the way that the value is displayed for the purposes of the query.

A. Concatenation (||)

Two or more strings can be concatenated (stuck together) using the || operator. You can make the first name and last name appear as though they are stored in one field by using the concatenation function. Add the concatenation symbol (2 pipes or the character above the forward slash key) between the attribute names to be added together.

Output the students’ names from the student table to appear as one attribute.

SELECT sfname || slname FROM student;

To make the output more English-like, you can add a space between the first and last name fields and give an alias for the column heading. The concatenation function is really useful as it allows you to enter any characters between single quotes. The characters will appear exactly as entered.

The syntax is: attribute name|| ‘ text to appear’ || attribute name
### SQL Tutorial

<table>
<thead>
<tr>
<th>SQL Query</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>`SELECT sfname</td>
<td></td>
</tr>
<tr>
<td><strong>We can use this function to show the students’ names formatted as Last Name, First Name.</strong>&lt;br&gt;`SELECT slname</td>
<td></td>
</tr>
</tbody>
</table>

### B. LOWER

This function converts all the characters of the string input to lowercase.

**Syntax:** LOWER(attribute)

With this function, we pass the argument of the attribute which is to be converted to lowercase.

```
SELECT LOWER(flname) FROM faculty;
```

### C. UPPER

This function converts all the characters of the string input to uppercase.

**Syntax:** UPPER (attribute)

With this function, we pass the argument of the attribute which is to be converted to uppercase.

```
SELECT UPPER(flname) FROM faculty;
```

### D. Initcap

This function converts all the characters of the string input to mixed (proper) case.
SQL Tutorial

Syntax: initcap(string)

With this function, we pass the argument of the attribute which is to be converted to mixed case.

Show the states in the student table in mixed case format.

<table>
<thead>
<tr>
<th>INITCAP(SSTATE)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nj</td>
</tr>
<tr>
<td>De</td>
</tr>
<tr>
<td>Pa</td>
</tr>
<tr>
<td>Nj</td>
</tr>
<tr>
<td>Id</td>
</tr>
<tr>
<td>Id</td>
</tr>
</tbody>
</table>

SELECT INITCAP(sstate)
FROM student;

E. LTRIM

The LTRIM function removes from the left of string any chars until it finds one that is not in chars. This is a really useful little function. Many times companies start invoice numbers with a specific letter combination and then have a sequential numbering system afterwards. If you strip off the leading characters with the Ltrim, you can do all kinds of things with the remaining characters like sort them different ways.

Syntax: (string, ['chars'])

With this function, we pass the argument of the attribute which is to be converted to upper case

1. Example: LTRIM (‘abracadabra’, ‘abr’) will result in ‘acadbra’

F. RTRIM

This is the same as LTRIM except it trims on the right of the string.

Syntax: RTRIM(string, ['chars'])

With this function, we pass the argument of the attribute which is to be converted to upper case

1. Example: RTRIM (‘abracadabra’, ‘bra’) will result in ‘abracada’

G. LENGTH(string)

The LENGTH function takes a string as an argument and returns the number of characters in the string.

Syntax: LENGTH(string)

With this function, we pass the string whose length is to be counted. Returns the length of the string input

<table>
<thead>
<tr>
<th>FLNAME</th>
<th>LENGTH(FLNAME)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Knightly</td>
<td>8</td>
</tr>
<tr>
<td>Streep</td>
<td>6</td>
</tr>
<tr>
<td>Redford</td>
<td>7</td>
</tr>
<tr>
<td>Newman</td>
<td>6</td>
</tr>
<tr>
<td>Dench</td>
<td>5</td>
</tr>
<tr>
<td>Binoche</td>
<td>7</td>
</tr>
</tbody>
</table>

Display the last name and the length of the last names of the faculty members

SELECT fnname, LENGTH(fnname)
FROM faculty;
Chapter 29: Aggregation

I. Introduction to Aggregation

Although detail level data is great, many times we actually want to see aggregated or summarized data. The result of an aggregate function is just one row of data. Oracle's SQL supports five aggregate functions: count, sum, avg, max, and min.

For example, let's say you have a product table and one of the attributes is price. What if you want to know what is the average price of the products in the table. Or, perhaps you want to know what are the highest (max) and lowest (min) priced products sold. Maybe you want to know how many (count) products we sell. Finally, you might want to add up (sum) the value of all the products. All of this can be accomplished with Oracle’s aggregation functions.

To help illustrate some of the concepts in aggregation, we'll introduce a new table called the product table. Here is the table structure:

<table>
<thead>
<tr>
<th>Column Name</th>
<th>ID</th>
<th>Pk</th>
<th>Nu?</th>
<th>Data Type</th>
<th>Default</th>
<th>Histogram</th>
</tr>
</thead>
<tbody>
<tr>
<td>SKU</td>
<td>1</td>
<td>N</td>
<td>Y</td>
<td>INTEGER</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>PRODUCT_DESCRIPTION</td>
<td>2</td>
<td>N</td>
<td>Y</td>
<td>VARCHAR2 (50 Byte)</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>PRICE</td>
<td>3</td>
<td>N</td>
<td>Y</td>
<td>NUMBER</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>QOH</td>
<td>4</td>
<td>N</td>
<td>Y</td>
<td>INTEGER</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>PRODUCTTYPE</td>
<td>5</td>
<td>Y</td>
<td>Y</td>
<td>VARCHAR2 (35 Byte)</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>MANUFACTURER</td>
<td>6</td>
<td>Y</td>
<td>Y</td>
<td>INTEGER</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

II. COUNT

The COUNT function lets you count the values (or rows) in a table.

A. COUNT (*) vs. COUNT (fieldname)

COUNT is quite useful when considering if a table has data in it or not. If you write a query using COUNT (*) and it returns 0 rows, it means that there are no rows but the table exists.

The following is very important to note:

1. COUNT (*) counts rows that contain null values
2. COUNT (fieldname) only counts rows that contain values so it excludes rows with null values.

B. Syntax: COUNT (fieldname)

COUNT is unique from some of the other aggregate functions as it accepts a column in a table of any data type and can even accept an * (asterisk) wildcard.

Show how many products are in the product table.

SELECT COUNT(sku) FROM product;
SQL Tutorial

SELECT COUNT(ProductType) FROM product;

<table>
<thead>
<tr>
<th>COUNTPRODUCTTYPE</th>
<th>Product Type Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td>6</td>
</tr>
</tbody>
</table>

But we’re not trying to find this out. Now if we change the query as follows, we can see that there are only 6 different product types represented in the product table. We'll learn a bit more about how to find out exactly how many products are in which product type in an upcoming section. Add an alias to improve the usability of the query.

SELECT COUNT (DISTINCT ProductType) AS “Product Type Count” FROM product;

III. SUM

The sum function allows you to add up the values in an attribute. As we'll note, SUM is different than COUNT. COUNT literally counts how many rows exist in a table. SUM performs addition on the values of a field to come up with a total. Accordingly, SUM only works on a field that has a data type of a number. Also, SUM will ignore values which are null.

A. Syntax: SUM (fieldname)

What is the total value of all of the products in the product table?

SELECT SUM(price) FROM product;

While this query works, it actually isn't really accurate as it assumes that there is only 1 of each item. What we really want to do is figure out what is the value of all the items in inventory.

To do this, we need to do some basic math. We know how many of each item is in inventory through the QOH (Quantity on Hand) attribute. If we multiply the quantity on hand of each item by its price and THEN sum it all up, we will find the value of the inventory.

SELECT SUM(price * QOH) AS “Total Value of Inventory” FROM product;

<table>
<thead>
<tr>
<th>Total Value of Inventory</th>
</tr>
</thead>
<tbody>
<tr>
<td>19589.41</td>
</tr>
</tbody>
</table>

IV. AVERAGE

The AVERAGE function allows you to calculate the average of the values in an attribute. Just like SUM, AVERAGE only works on a field that has a data type of a number and will ignore values which are null.

A. Syntax: AVERAGE(fieldname)

What is the average price of the products in the product table? Add an alias to improve the readability of the output.

SELECT AVG(price) AS “Average Price” FROM product;

<table>
<thead>
<tr>
<th>Average Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>111.119411764706</td>
</tr>
</tbody>
</table>
SQL Tutorial

We typically show currency at 2 decimal points so we will either want to truncate this output or round it to 2 decimal points. Let’s truncate it.

```
SELECT TRUNC(AVG(price),2) AS "Average Price"
FROM product;
```

### V. MIN and MAX

The minimum and maximum functions show the smallest value and largest value, respectively, in a set of rows. MIN and MAX work with numeric data types as well as VARCHAR2.

#### A. ASCII character set

When you are using the MIN and MAX functions with character strings it is important to remember that SQL considers characters according to the ASCII character set. ASCII converts characters into number equivalents. Capital letter "A" is equal to ASCII 65, “B” is 66, etc. Lower letters start at 97 for “a” and work up sequentially. Accordingly, if you look for the minimum value in a field, something with a capital “C” will be show up before a small case “a”.

#### B. Syntax:

- **MIN(fieldname)**
- **MAX(fieldname)**

<table>
<thead>
<tr>
<th>Show the cheapest price of the products.</th>
<th>Least Expensive Item</th>
</tr>
</thead>
</table>
| SELECT MIN(price) AS "Least Expensive Item"
FROM product;                                    | $6.99                |

<table>
<thead>
<tr>
<th>Show the most expensive price of the products</th>
<th>Most Expensive Item</th>
</tr>
</thead>
</table>
| SELECT MAX(price) AS "Most Expensive Item"
FROM product;                                    | $999.99               |

<table>
<thead>
<tr>
<th>Show the first alphabetic value of the products</th>
<th>First item</th>
</tr>
</thead>
</table>
| SELECT MIN(product_description) AS "First item"
FROM product;                                    | Roy Friend Jeans |

<table>
<thead>
<tr>
<th>Show the last alphabetic value of the products</th>
<th>First item</th>
</tr>
</thead>
</table>
| SELECT max(product_description) as "First item"
FROM product;                                    | Union Bay Tiki Flower Shorts |

### VI. GROUP BY

So far when we’ve written a query with an aggregation function we have just retrieved one row of data that shows the aggregation for that field’s value across the entire table. For instance, we retrieved the average price of all of the products in the product table. But what if we want to break down the output? Let’s say we want to see the average price of the products in the product table by product type. Here are the distinct product types in the product type table:
With a GROUP BY, we can find out what is the average price for all of the china items versus the average price of all of the electronic items. If we don’t use the GROUP BY, we’ll just return the average price for all of the items lumped together in the product table. SQL allows you to see aggregated results “GROUPED BY” some attribute using a GROUP BY clause.

The GROUP BY clause literally tells SQL to group the output by something. It might help to think of it in this way - a GROUP BY clause tells SQL to create a distribution or a little table. Conceptually, this is basically how it works. First SQL reads through all the rows and groups them into categories based on the attribute in the group by clause. Then, once the grouping has occurred, it calculates the summary function (MIN, MAX, SUM, COUNT, OR AVERAGE).

A. Syntax: GROUP BY <fieldname>

- SELECT
- FROM
- WHERE
- GROUP BY
- ORDER BY

---

**Show the average price of items in the product table by product type**

```
SELECT AVG(price) FROM product GROUP BY producttype;
```

<table>
<thead>
<tr>
<th>AVG(PRICE)</th>
</tr>
</thead>
<tbody>
<tr>
<td>29.99</td>
</tr>
<tr>
<td>135.9025</td>
</tr>
<tr>
<td>34.99</td>
</tr>
<tr>
<td>220.3</td>
</tr>
<tr>
<td>23.99</td>
</tr>
<tr>
<td>32.74</td>
</tr>
</tbody>
</table>

Hmm, we can’t tell from this output what is the product type. Let’s add product type to the select clause as follows:

```
SELECT producttype, AVG(price) FROM product GROUP BY producttype;
```

<table>
<thead>
<tr>
<th>PRODUCTTYPE</th>
<th>AVG(PRICE)</th>
</tr>
</thead>
<tbody>
<tr>
<td>China</td>
<td>29.99</td>
</tr>
<tr>
<td>Electronics</td>
<td>135.9025</td>
</tr>
<tr>
<td>Housewares</td>
<td>34.99</td>
</tr>
<tr>
<td>Lawn and Garden</td>
<td>220.3</td>
</tr>
<tr>
<td>Men’s Clothes</td>
<td>23.99</td>
</tr>
<tr>
<td>Women's Clothes</td>
<td>32.74</td>
</tr>
</tbody>
</table>
Let's add an alias and format the number:

```sql
SELECT producttype AS "Product Type",
       ROUND(AVG(price),2) AS "Average Product Price"
FROM product
GROUP BY producttype;
```

<table>
<thead>
<tr>
<th>Product Type</th>
<th>Average Product Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>China</td>
<td>29.99</td>
</tr>
<tr>
<td>Electronics</td>
<td>135.9</td>
</tr>
<tr>
<td>Housewares</td>
<td>34.99</td>
</tr>
<tr>
<td>Lawn and Garden</td>
<td>220.3</td>
</tr>
<tr>
<td>Men's Clothes</td>
<td>23.99</td>
</tr>
<tr>
<td>Women's Clothes</td>
<td>32.74</td>
</tr>
</tbody>
</table>

B. Multiple forms of aggregation in one query.

SQL allows you to put multiple forms of summary data in one query.

```sql
SELECT producttype,
       COUNT(SKU) AS "Number of Items",
       ROUND(AVG(price),2) AS "Average Price",
       MIN(price) AS "Cheapest Item",
       MAX(price) AS "Most Expensive Item"
FROM product
GROUP BY producttype;
```

<table>
<thead>
<tr>
<th>PRODUCTTYPE</th>
<th>Number of Items</th>
<th>Average Price</th>
<th>Cheapest Item</th>
<th>Most Expensive Item</th>
</tr>
</thead>
<tbody>
<tr>
<td>China</td>
<td>1</td>
<td>29.99</td>
<td>29.99</td>
<td>29.99</td>
</tr>
<tr>
<td>Electronics</td>
<td>4</td>
<td>135.9</td>
<td>21.99</td>
<td>21.99</td>
</tr>
<tr>
<td>Housewares</td>
<td>1</td>
<td>34.99</td>
<td>34.99</td>
<td>34.99</td>
</tr>
<tr>
<td>Lawn and Garden</td>
<td>5</td>
<td>220.3</td>
<td>6.99</td>
<td>89.99</td>
</tr>
<tr>
<td>Men's Clothes</td>
<td>2</td>
<td>23.99</td>
<td>12.99</td>
<td>23.99</td>
</tr>
<tr>
<td>Women's Clothes</td>
<td>4</td>
<td>32.74</td>
<td>24.99</td>
<td>40.66</td>
</tr>
</tbody>
</table>

C. Adding more than one attribute (nest) the GROUP BY clause

You can nest the attributes in your GROUP BY clause.

You can use up to 10 fields to group rows, with the order of field names determining the group levels from highest to lowest.

```sql
SELECT producttype, manufacturer,
       COUNT(SKU) AS "Number of Items"
FROM product
GROUP BY producttype, manufacturer;
```

<table>
<thead>
<tr>
<th>PRODUCTTYPE</th>
<th>MANUFACTURER</th>
<th>Number of Items</th>
</tr>
</thead>
<tbody>
<tr>
<td>China</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>Housewares</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Electronics</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>Electronics</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>Electronics</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>Men's Clothes</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>Men's Clothes</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>Lawn and Garden</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>Lawn and Garden</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>Lawn and Garden</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>Lawn and Garden</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>Women's Clothes</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>Women's Clothes</td>
<td>7</td>
</tr>
</tbody>
</table>

D. Adding criteria with GROUP BY clause

You can add joins and criteria to your query.

In the first query, we'll just show the average product price by product type.
### SQL Tutorial

**SELECT producttype AS “Product Type”,**
**ROUND(AVG(price),2) AS “Average Product Price”**
**FROM product**
**GROUP BY producttype;**

<table>
<thead>
<tr>
<th>Product Type</th>
<th>Average Product Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>China</td>
<td>29.99</td>
</tr>
<tr>
<td>Electronics</td>
<td>135.9</td>
</tr>
<tr>
<td>Housewares</td>
<td>34.99</td>
</tr>
<tr>
<td>Lawn and Garden</td>
<td>220.3</td>
</tr>
<tr>
<td>Men’s Clothes</td>
<td>23.99</td>
</tr>
<tr>
<td>Women’s Clothes</td>
<td>32.74</td>
</tr>
</tbody>
</table>

No consider if I restrict the output to SKUs > 400000, we can see that the average amount of some product types has changed (i.e. Lawn and Garden’s average price went from $220.30 to $47.51) and some product types are completely excluded from the output (i.e. Electronics) since a number of items have been excluded.

Show the average price of items in the product table by product type if the SKU is greater than 400000.

**SELECT producttype AS “Product Type”,**
**ROUND(AVG(price),2) AS “Average Product Price”**
**FROM product**
**WHERE SKU > 400000**
**GROUP BY producttype;**

### Sorting the results of the GROUP BY clause

Normally, the output of an aggregation query is sorted in ascending order by the attribute in the group by clause. For instance, you’ll note in the above example that the output is alphabetically arranged in ascending order by product type. You can override this by an ORDER BY clause.

<table>
<thead>
<tr>
<th>Product Type</th>
<th>Number of Items</th>
<th>Average Price</th>
<th>Cheapest Item</th>
<th>Most Expensive Item</th>
</tr>
</thead>
<tbody>
<tr>
<td>Women’s Clothes</td>
<td>4</td>
<td>32.74</td>
<td>24.99</td>
<td>46.99</td>
</tr>
<tr>
<td>Men’s Clothes</td>
<td>2</td>
<td>20.99</td>
<td>10.99</td>
<td>34.99</td>
</tr>
<tr>
<td>Lawn and Garden</td>
<td>5</td>
<td>220.3</td>
<td>6.99</td>
<td>899.99</td>
</tr>
<tr>
<td>Housewares</td>
<td>1</td>
<td>34.99</td>
<td>34.99</td>
<td>34.99</td>
</tr>
<tr>
<td>Electronics</td>
<td>4</td>
<td>135.9</td>
<td>21.99</td>
<td>345.76</td>
</tr>
<tr>
<td>China</td>
<td>1</td>
<td>29.99</td>
<td>29.99</td>
<td>29.99</td>
</tr>
</tbody>
</table>

Sort the example output above by descending product type.

**SELECT producttype,**
**COUNT(SKU) AS “Number of Items”,**
**ROUND(AVG(price),2) AS “Average Price”,**
**MIN(price) AS “Cheapest Item”,**
**MAX(price) AS “Most Expensive Item”**
**FROM product**
**GROUP BY producttype**
**ORDER BY producttype DESC;**
Sort the above output by ascending maximum price:

```sql
SELECT producttype, COUNT(SKU) AS "Number of Items", ROUND(AVG(price),2) AS "Average Price", MIN(price) AS "Cheapest Item", MAX(price) AS "Most Expensive Item" FROM product GROUP BY producttype ORDER BY MAX(price) ASC;
```

<table>
<thead>
<tr>
<th>PRODUCTTYPE</th>
<th>Number of Items</th>
<th>Average Price</th>
<th>Cheapest Item</th>
<th>Most Expensive Item</th>
</tr>
</thead>
<tbody>
<tr>
<td>Things</td>
<td>1</td>
<td>29.99</td>
<td>29.99</td>
<td></td>
</tr>
<tr>
<td>Housewares</td>
<td>1</td>
<td>34.99</td>
<td>34.99</td>
<td></td>
</tr>
<tr>
<td>Men's Clothes</td>
<td>2</td>
<td>23.99</td>
<td>12.99</td>
<td></td>
</tr>
<tr>
<td>Women's Clothes</td>
<td>4</td>
<td>32.74</td>
<td>24.99</td>
<td></td>
</tr>
<tr>
<td>Electronics</td>
<td>4</td>
<td>135.9</td>
<td>21.99</td>
<td></td>
</tr>
<tr>
<td>Lawn and Garden</td>
<td>5</td>
<td>220.3</td>
<td>6.99</td>
<td></td>
</tr>
</tbody>
</table>

**F. Displaying non-aggregate attributes in aggregation query**

Let’s say we want to add the SKU to this output:

```sql
SELECT SKU, producttype AS "Product Type", ROUND(AVG(price),2) AS "Average Product Price" FROM product GROUP BY producttype;
```

Wait – we get this error. Why?

1. Remember that in aggregation you are showing summary data, not row level data. If you try and show row level data such as each individual SKU, SQL will tell you it isn’t a group by expression. You can only show attributes in the select statement that are being summarized OR are the attribute on which you are grouping your data through a group by clause. For instance, let’s revisit one of our queries from the prior section.

```sql
SELECT AVG(price) AS "Average Price" FROM product;
```

2. If we try to add producttype to this query as shown, we’ll get a single-group group function error as displayed below. This is SQL’s way of saying you are trying to show single level data but you aren’t grouping your output in any way. You either have to add the producttype to a group by clause or take it out of the select clause.

```sql
SELECT producttype, AVG(price) AS "Average Price" FROM product;
```
G. Rules of thumb:
1. Put all of the grouping columns in the SELECT list.
2. Only columns may be used for grouping, not expressions.
3. Don’t use GROUP BY to get multiple levels of summary value.
4. Don’t use GROUP BY without aggregates—DISTINCT gives the same effect and the meaning is clearer.

VII. HAVING
We have seen that you can place criteria or filters on queries using a WHERE clause. But SQL has another feature that really is useful. You can filter on aggregation. Previously we wrote a query that displayed the average product price by product type.

<table>
<thead>
<tr>
<th>Product Type</th>
<th>Average Product Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>China</td>
<td>29.99</td>
</tr>
<tr>
<td>Electronics</td>
<td>135.9</td>
</tr>
<tr>
<td>Housewares</td>
<td>34.99</td>
</tr>
<tr>
<td>Lawn and Garden</td>
<td>220.3</td>
</tr>
<tr>
<td>Men’s Clothes</td>
<td>23.99</td>
</tr>
<tr>
<td>Women’s Clothes</td>
<td>32.74</td>
</tr>
</tbody>
</table>

What if a client asked us to write the same query for those product types where the average is above $33. Look what happens if we try do this with a WHERE clause.

SELECT producttype AS "Product Type", round(avg(price),2) AS "Average Product Price"
FROM product
WHERE avg(price) > 33
GROUP BY producttype;
SQL Tutorial

So what do we do? SQL has a special feature called a HAVING clause. HAVING is used to restrict an output based on an aggregated condition.

A. Syntax: HAVING aggregate function(condition)

<table>
<thead>
<tr>
<th>Product Type</th>
<th>Average Product Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>Electronics</td>
<td>135.9</td>
</tr>
<tr>
<td>Housewares</td>
<td>34.99</td>
</tr>
<tr>
<td>Lawn and Garden</td>
<td>220.3</td>
</tr>
</tbody>
</table>

Show the average product price by product for all products with an average price greater than $33

SELECT producttype AS "Product Type", ROUND(AVG(price),2) AS "Average Product Price" FROM product GROUP BY producttype HAVING AVG(price) > 33;

B. HAVING and WHERE Clause

1. Filtering on Aggregated and Unaggregated Data in the Same Query:

There are times when you will want to filter on aggregated data AND un-aggregated data. In these situations you will have both a WHERE and a HAVING clause

For example, let’s say we want to know the average product price by product for all products with a minimum price greater than $20 and the manufacturer is more than 2. Since we are looking for a minimum price which is an aggregate function, we will need to use a having for this piece of the query. We are also looking to filter on un-aggregated data since we want to restrict to manufacturers with a code > 2. We will use a WHERE clause to filter on this condition. The query will look as follows:

<table>
<thead>
<tr>
<th>Product Type</th>
<th>Average Product Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>Electronics</td>
<td>135.9</td>
</tr>
<tr>
<td>Women's Clothes</td>
<td>32.74</td>
</tr>
</tbody>
</table>

SELECT producttype AS "Product Type", ROUND(AVG(price),2) AS "Average Product Price" FROM product WHERE manufacturer >2 GROUP BY producttype HAVING MIN(price) > 20;

2. Filtering on Aggregated Data and a Join in the Same Query:

If you want to filter on aggregated data and your query includes more than one table, you will have both a WHERE and a HAVING clause.

For instance, we want to show the average product price by manufacturer for all products with a minimum price greater than $20. Since we are looking for an average price which is an aggregate function, we will need to use a having for this piece of the query. Since the manufacturer’s name is in the manufacturer table, we need to include 2 tables into this query. Whenever we have more than one table, we need a join. We will use a WHERE clause to join the tables and a HAVING clause to filter on the condition.
3. If you want to filter on aggregated data, your query includes more than one table, AND you want to filter on unaggregated data you will have both a WHERE and a HAVING clause.

For example, show the average product price by manufacturer for all products made by manufacturer's with names beginning with D or after in the alphabet AND with a minimum price greater than $20.

a. Since we are looking for a average price which is an aggregate function, we will need to use a HAVING for this piece of the query.

b. Since the manufacturer's name is in the manufacturer table, we need to include 2 tables into this query. Whenever we have more than one table, we need a join.

c. We also are filtering on aggregated data as we are limiting the output to just those Manufacturer's with names >D. We will use a WHERE clause to join the tables and a HAVING clause to filter on the condition.

4. Filtering on Aggregated and a Join in the Same Query with Sort:

Let's say we want to write the same query as above but this time we want the output to be sorted by descending average product price. All we need to do is add an ORDER BY clause and we are good to go.
SELECT manufacturername AS "Manufacturer", ROUND(AVG(price),2) AS "Average Product Price"
FROM product, manufacturer
WHERE manufacturer = manufacturer_id
AND MANUFACTURERNAME > 'D'
GROUP BY manufacturername
HAVING MIN(price) > 20
ORDER BY ROUND(AVG(price),2)
DESC;

<table>
<thead>
<tr>
<th>Manufacturer</th>
<th>Average Product Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sunbeam</td>
<td>899.99</td>
</tr>
<tr>
<td>Sony</td>
<td>345.76</td>
</tr>
<tr>
<td>Katie and Friends</td>
<td>127.54</td>
</tr>
<tr>
<td>Lucky Brand</td>
<td>46.99</td>
</tr>
<tr>
<td>Denali Brand</td>
<td>34.99</td>
</tr>
<tr>
<td>Goebel</td>
<td>34.99</td>
</tr>
<tr>
<td>Intersect</td>
<td>33.93</td>
</tr>
<tr>
<td>Mikasa</td>
<td>29.99</td>
</tr>
<tr>
<td>Union Bay</td>
<td>27.99</td>
</tr>
</tbody>
</table>

C. The Difference Between the HAVING and WHERE Clauses in a SQL Query

Although the HAVING clause specifies a condition that is similar to the purpose of a WHERE clause, the two clauses are not interchangeable. Listed below are some differences to help distinguish between the two:

1. The WHERE clause specifies the criteria which individual rows must meet to be selected by a query. It can be used without the GROUP BY clause. The HAVING clause cannot be used without the GROUP BY clause.

2. The WHERE clause selects rows before grouping. The HAVING clause selects rows after grouping.

3. The WHERE clause cannot contain aggregate functions. The HAVING clause can contain aggregate functions.

Paragon Corporation created this output which has a nice summary of the components of an aggregate query.\(^9\)

## SQL Tutorial

<table>
<thead>
<tr>
<th>Clause</th>
<th>Purpose</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT</td>
<td>Used to specify what fields will be included in the query result. This clause is always found in SQL query statements.</td>
<td>SELECT AVERAGE(Price) AS &quot;Average Price&quot;</td>
</tr>
<tr>
<td>FROM</td>
<td>Specifies what tables data will come from. This exists in all SQL query statements.</td>
<td>SELECT SUM(Amount) AS Total FROM Sales WHERE CustomerID='ABC'</td>
</tr>
<tr>
<td>WHERE</td>
<td>Specifies what subset of the data will be used (always has non-aggregate conditions). This clause is almost always found in SQL query statements.</td>
<td>SELECT Sum(Amount) AS Total FROM Sales WHERE CustomerID='ABC'</td>
</tr>
<tr>
<td>JOIN</td>
<td>The join clause is used to link more than one table together. This is often found in more complex queries that require retrieving data from more than one table. There are several formats (INNER JOIN, OUTER JOIN, LEFT JOIN, RIGHT JOIN, ..OUTER is usually optional</td>
<td>SELECT Customers.CustomerName, Sales.CustomerID, Sum(Sales.Amount) AS Total FROM Sales INNER JOIN Customers ON Customers.CustomerID = Sales.CustomerID WHERE Sales.CustomerID='ABC'</td>
</tr>
<tr>
<td>GROUP BY</td>
<td>Used to specify about what fields data should be aggregated. In this example, we group by CustomerID so that we get a summary of the total purchases per customer</td>
<td>SELECT CustomerID, SUM(Amount) AS Total FROM Sales GROUP BY CustomerID</td>
</tr>
<tr>
<td>HAVING</td>
<td>HAVING is very similar to WHERE except the statements within it are of an aggregate nature. Note in this example - we are only returning summaries for customers who have purchased more than 60,000 worth of items</td>
<td>SELECT CustomerID, SUM(Amount) AS Total FROM Sales GROUP BY CustomerID HAVING Sum(Amount) &gt; 60000</td>
</tr>
</tbody>
</table>

### Aggregation Functions
- **SUM**, **COUNT**, **AVG**

Aggregate functions are used to summarize data by rolling up a set of data items into a single item. There are a few basic ones that exist in most systems that support SQL, and many are specific to certain DBMS. An important thing to note is that if a column in the resultset is not an aggregate field, then it must be included in the GROUP BY clause.

SELECT CustomerID, SUM(Amount) AS Total, COUNT(*) As SaleCount, AVG(Amount) AS AverageOrder FROM Sales GROUP BY CustomerID
Chapter 30: Nested Query/SubQuery

I. Introduction to Nested or Sub Queries

A nested query, also known as a sub-query, is a query within a query. Nested queries can be used to accomplish a number of things. For instance, you can use a nested query to write queries with multiple tables and avoid writing joins. One of the really great things about a nested query is that it allows you to write aggregate queries which include non-aggregated data.

As we saw in the prior section, you cannot have aggregated and non-aggregated data in the same query. But a nested query is really a query within a query. This allows you to write one query, the inner query, which calculates the aggregated part of the query. The second query uses the output of the inner query as an input to the outer query, the un-aggregated query. Sufficiently confused?! Hold on, it will make sense!

II. Examples with Nested Queries

A. Let's say you have a product table. You want to display the product information for those products which cost less than the average price of all the products. If we tried to write the query as follows you will get an error message as shown below.

```
SELECT SKU, product_description, price, QOH, producttype, manufacturer 
FROM product 
WHERE price > AVG(price);
```

B. To generate the desired output, we essentially need to write 2 queries.

Query 1:
```
SELECT AVG(price) 
FROM product;
```

Now that we know what the average price is, we can write another query that retrieves the un-aggregated information as follows:

**QUERY 2:**
```
SELECT sku, product_description, price, QOH, producttype, manufacturer 
FROM product 
WHERE price > 111.12;
```

1. We can see that all of the items' prices were greater than the average price of all of the products. But this means you need to write two queries each time. Because you need to type the information from the first query, there is always room for error. That's where the nesting part comes in. Let's take the two queries and make them into one.
2. We already learned that anything inside a parenthesis is evaluated first so we need to make sure that the inner query (or nested query) is the aggregated part. This will look identical to the aggregated query we already wrote – Query 1. The outer query or the un-aggregated part will look the same as Query 2 except that the numeric output of query 1 is actually replaced with the query.

```sql
SELECT sku, product_description, price, QOH, producttype, manufacturer
FROM product
WHERE price > (SELECT AVG(price) FROM product);
```

<table>
<thead>
<tr>
<th>SKU</th>
<th>PRODUCT_DESCRIPTION</th>
<th>PRICE</th>
<th>QOH</th>
<th>PRODUCTTYPE</th>
<th>MANUFACTURER</th>
</tr>
</thead>
<tbody>
<tr>
<td>145666</td>
<td>Sony Trinitron 30” TV</td>
<td>345.76</td>
<td>14</td>
<td>Electronics</td>
<td>Sony</td>
</tr>
<tr>
<td>269886</td>
<td>Ear Force gamer head set</td>
<td>129.99</td>
<td>15</td>
<td>Electronics</td>
<td>Bøgje</td>
</tr>
<tr>
<td>4653446</td>
<td>Extra Large Wind Chimes</td>
<td>127.54</td>
<td>3</td>
<td>Lawn and Garden</td>
<td>Katie and Friends</td>
</tr>
<tr>
<td>276655</td>
<td>Deluxe Patio Set - 6 chairs + umbrella</td>
<td>899.99</td>
<td>5</td>
<td>Lawn and Garden</td>
<td>Sunbeanz</td>
</tr>
</tbody>
</table>

C. Adding a join to a nested query

The prior query is great but the manufacturer id isn’t all that useful. We’d like to see the manufacturer’s name. Let’s add a join here to get the information from the manufacturer table.

```sql
SELECT sku, product_description, price, QOH, producttype, manufacturername
FROM product, manufacturer
WHERE price > (SELECT AVG(price) FROM product) AND manufacturer = manufacturer_id;
```

<table>
<thead>
<tr>
<th>SKU</th>
<th>PRODUCT_DESCRIPTION</th>
<th>PRICE</th>
<th>QOH</th>
<th>PRODUCTTYPE</th>
<th>MANUFACTURERNAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>145666</td>
<td>Sony Trinitron 30” TV</td>
<td>345.76</td>
<td>14</td>
<td>Electronics</td>
<td>Sony</td>
</tr>
<tr>
<td>269886</td>
<td>Ear Force gamer head set</td>
<td>129.99</td>
<td>15</td>
<td>Electronics</td>
<td>Bøgje</td>
</tr>
<tr>
<td>4653446</td>
<td>Extra Large Wind Chimes</td>
<td>127.54</td>
<td>3</td>
<td>Lawn and Garden</td>
<td>Katie and Friends</td>
</tr>
<tr>
<td>276655</td>
<td>Deluxe Patio Set - 6 chairs + umbrella</td>
<td>899.99</td>
<td>5</td>
<td>Lawn and Garden</td>
<td>Sunbeanz</td>
</tr>
</tbody>
</table>

D. Adding multiple criteria to a query

Limit the above query to so that it only displays electronic products

```sql
SELECT sku, product_description, price, QOH, producttype, manufacturername
FROM product, manufacturer
WHERE producttype = 'Electronics' and price > (SELECT AVG(price) FROM product) AND manufacturer = manufacturer_id;
```

<table>
<thead>
<tr>
<th>SKU</th>
<th>PRODUCT_DESCRIPTION</th>
<th>PRICE</th>
<th>QOH</th>
<th>PRODUCTTYPE</th>
<th>MANUFACTURERNAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>145666</td>
<td>Sony Trinitron 30” TV</td>
<td>345.76</td>
<td>14</td>
<td>Electronics</td>
<td>Sony</td>
</tr>
<tr>
<td>269886</td>
<td>Ear Force gamer head set</td>
<td>129.99</td>
<td>15</td>
<td>Electronics</td>
<td>Bøgje</td>
</tr>
</tbody>
</table>
E. Mathematical functions and nested queries

Let’s add the value of the inventory (QOH * Price) to the query and improve the headings

<table>
<thead>
<tr>
<th>SKU</th>
<th>Product</th>
<th>Price</th>
<th>QOH</th>
<th>Inventory Value</th>
<th>Product Type</th>
<th>Manufacturer</th>
</tr>
</thead>
<tbody>
<tr>
<td>145566</td>
<td>Sony Trinitron 36&quot; TV</td>
<td>345.76</td>
<td>14</td>
<td>4840.64</td>
<td>Electronics</td>
<td>Sony</td>
</tr>
<tr>
<td>209880</td>
<td>Ear Force gamer head set</td>
<td>129.99</td>
<td>15</td>
<td>1949.85</td>
<td>Electronics</td>
<td>Bose</td>
</tr>
</tbody>
</table>

```sql
SELECT sku, product_description AS "Product", price, QOH, (price * QOH) AS "Inventory Value", producttype AS "Product Type", manufacturername AS "Manufacturer"
FROM product, manufacturer
WHERE producttype = 'Electronics' and price > (SELECT avg(price) FROM product) AND manufacturer = manufacturer_id;
```

F. Sort the output

Sort the output by ascending manufacturer’s name

<table>
<thead>
<tr>
<th>SKU</th>
<th>Product</th>
<th>Price</th>
<th>QOH</th>
<th>Inventory Value</th>
<th>Product Type</th>
<th>Manufacturer</th>
</tr>
</thead>
<tbody>
<tr>
<td>209880</td>
<td>Ear Force gamer head set</td>
<td>129.99</td>
<td>15</td>
<td>1949.85</td>
<td>Electronics</td>
<td>Bose</td>
</tr>
<tr>
<td>145566</td>
<td>Sony Trinitron 36&quot; TV</td>
<td>345.76</td>
<td>14</td>
<td>4840.64</td>
<td>Electronics</td>
<td>Sony</td>
</tr>
</tbody>
</table>

```sql
SELECT sku, product_description AS "Product", price, QOH, (price * QOH) AS "Inventory Value", producttype AS "Product Type", manufacturername AS "Manufacturer"
FROM product, manufacturer
WHERE producttype = 'Electronics' and price > (SELECT AVG(price) FROM product) AND manufacturer = manufacturer_id
ORDER BY manufacturername ASC;
```
%  Wildcard, 148
* , 144
-  Wildcard, 148
1  1 Relationship Example, 15
    M Relationship Example, 16
Access  Corsstab Query, 101
       Create Table - Design View, 30
       Field Properties, 30
       Field Name, 30
       Forms, 57
       Navigation Pane, 29
       Ribbon, 29
       Tables, 30
Access Data Type  
       Currency, 31
Access OLE Object, 31
Access Queries  
       Aggregate Function  
           Count, 51
Access Yes/No, 31
Access 1  
    M Subform  
        Create, 71
Access Action Queries, 48
Access Aggregate Function, 51
Access AutoNumber, 31
Access Boolean, 31
Access Byte, 33
Access Check Box, 67
Access Combo Box, 64
Access Command Button, 69
Access Controls  
       Combo Box, 64
       Check Box, 67
       Command Button, 69
       Option Box, 67
Access Controls, 64
       List Box, 64
Access Count, 51
Access Create Form in Design View, 60
Access Crosstab Query, 46
Access Currency Format, 34
Access Data Type  
       AutoNumber, 31
       Date/Time, 31
       Hyperlink, 31
       OLE Object, 31
       Yes/No, 31
Access Data Type, 31
       Memo, 31
       Number, 31
       Text, 31
Access Date Format, 34
Access Date/Time, 31
Access Decimal, 33
Access Double, 33
Access Field Properties  
       Currency Format, 34
       Date Format, 34
       Field Size, 33
       Field Validation Rules, 35
       Format  
           Text and memo format, 33
       Input Masks, 35
       Number format, 33
       Primary Key, 35
       Text and memo format, 33
       Yes/No, 34
Access Field Properties, 30
       Data Type, 31
       Field Name, 30
Access Field Properties
       Index, 35
Access Field Size  
       Byte, 33
       Decimal, 33
       Double, 33
       Integer, 33
       Long Integer, 33
       Single, 33
Access Form  
       Subform 1
       M, 71
Access Form Controls, 64
Access Format, 33
Access Forms  
       Multipage using Tab, 79
       Subforms M
       M, 76
Access Forms  
       Create Form Using Wizard, 57
       Edit Form, 62
Access From
       Add Records to Form, 62
Access Hyperlink, 31
Access Index, 35
Access Integer, 33
Access Keyboard
       Shortcuts, 92
Access List Box, 64
Access Long Integer, 33
Access Max, 54
Access Memo, 31
Access Multipage Form
       Tab, 79
Access Number, 31
Access Number Format, 33
Access Object Linking and Embedding, 31
Index

Access Option Box, 67
Access Parameter Queries, 44
Access Primary Key, 35
Access Queries, 41
  Action Queries, 48
  Aggregate Function
    Max, 54
    Sum, 53
  Aggregate Functions, 51
  Crosstab Query, 46
  Parameter, 44
  Select Queries, 41
Access Query
  Query Wildcards, 41
Access Reports, 81
  Parameterized, 81
  Adhoc, 81
  Canned, 81
Access Select Queries, 41
Access Shortcuts, 92
Access Single, 33
Access Subforms M
  M, 76
Access Sum, 53
Access Switchboard, 85
  Create, 86
  View when Database Opens, 90
Access Switchboard within Switchboard, 88
Access Tab Form, 79
Access Tables
  Introduction to Tables, 38
Access Text, 31
Access Validation Rules, 35
Access Wildcards, 41
Add_Months, 177
Aggregation, 181
  Average, 182
  Count and Distinct, 181
  Group By, 183
  Having, 188
  Max, 183
  Min, 183
  Sum, 182
Alias Operator, 156
Alter Table, 128
AND Operator, 152
ANSI standard Join, 166
Asterisk, 144
Attribute
  Placement, 23
  Attribute Order in Select, 145
Attributes, 6
Average, 182
BETWEEN Operator, 151
Calculated Field, 15
Cascade Constraints, 128
Case Sensitivity, 124
Char DataType, 125
Check Constraint, 134
Chen, 7

Clause
  Where, 146
Column, 4
Commit, 136
Comparison Operator
  >, >=, !=, <>, >, <=, 149
Comparison Operators, 149
  BETWEEN Operator, 151
  IN Operator, 150
Composite Key, 9, 21
Concatenation, 178
Constraint
  Foreign Key, 131
Constraints, 130
  Check, 134
  Default Value, 133
  Null Value, 132
  Primary Key, 130
  Unique, 134
Controlled Redundancy, 10
Count and Distinct, 181
Create Table. See Create Table, 86
Cross Tab Query, 101
Crosstab Query in Excel, 105
Crow's Foot, 7
Currency, 31
Data Definition Language, 127
Data Dictionary, 28
Data Manipulation Language, 140
Data Type
  Date, 125
    Description, 32
      Integer, 125
      Number, 125
      Raw, 125
      VarChar2, 125
  Data Type Long, 125
  Data Types, 125
  DataType
    Char, 125
Date
  Sysdate, 174
Date Data Type, 125
Date Functions, 173
  Add_Months, 177
  Greatest, 178
  Last Day, 178
  Least, 178
  Months_between, 177
  Next_day, 178
  Round, 178
  to_char, 178
  To_Date, 177
  Trunc, 178
Dates
  Entering Date Data, 141
  DDL, 127
  DDL vs. DML, 125
  Default Value Constraint, 133
  Delete Statement, 170
| Derived Field, 15 |
| Describe, 129 |
| Distinct, 145 |
| DML, 140 |
| DML vs. DDL, 125 |
| Drop Table, 128 |
| Entity, 4 |
| Entity Names, 5 |
| Entity Relationship Diagram, 4 |
| equijoin, 161 |
| ERD, 4 |
| Field Properties, 33 |
| Currency format, 34 |
| Field Properties XE “Field Size” \t “See Field Properties: Field Size” Field Sir, 33 |
| Field Size. See Field Properties: Field Size |
| Foreign Key, 10, 17 |
| Foreign Key Constraints, 131 |
| Format Masks, 139 |
| Formatting Numbers, 137 |
| Formatting Query Results, 136 |
| Functions, 176 |
| Date, 173 |
| Greater Than (or Equal to), 149 |
| Greatest Date Function, 178 |
| Group By, 183 |
| Having, 188 |
| I Access Input Masks, 35 |
| IN Operator, 150 |
| Infinity Model, 7 |
| Initcap, 179 |
| Inner Join, 161 |
| Insert Data, 140 |
| Integer Data Type, 125 |
| Integrity Constraints, 130 |
| Intersect, 160 |
| Invoice Example, 24 |
| Is Not Null, 153 |
| Is Null, 153 |
| Join |
| ANSI Standard, 166 |
| Inner, 161 |
| Left, 168 |
| Outer, 165, 168 |
| Right, 168 |
| Join On, 167 |
| Last Day, 178 |
| Least Date Function, 178 |
| Left Join, 168 |
| Logical Operator |
| AND, 152 |
| Is Null, Is Not Null, 153 |
| NOT, 153 |
| OR, 152 |
| Logical Operators, 152 |
| Long |
| Data Type, 125 |
| Lookup Table, 17 |
| Lower, 179 |
Index

Select Statement, 140
Set Operators, 159
    Intersect, 160
    Union, 159
    Union All, 159
Simple Select, 143
Sort, 155
SQL Plus
    Logging In, 122
SQL+
    Exit, 123
    Help, 123
    Introduction, 122
    Password, 123
String Functions
    Concatenation, 178
    Initcap, 179
    Lower, 179
    Rtrim, 180
    Upper, 179
String Searches, 147
    Wildcard, 148
Sub Query, 193
Sum, 182
Sysdate, 174
To_Char, 178

To_Date, 141, 177
TOAD, 107
    Check Constraint, 112
    Configure TOAD, 107
    Constraints, 111
    Create Table, 110
    Entity Relationship Diagram, 119
    ERD, 119
    Exit, 121
    Foreign Key, 112
    Load Script, 116
    Load TOAD, 107
    Primary Key, 112
    SQL Statements, 117
    Table Creation Script, 114
Trunc, 176, 178
Union, 159
Union All, 159
Unique Constraint, 134
Update Statement, 169
Upper, 179
Varchar2 Data Type, 125
Where Clause, 146
Wildcard Characters, 148
Wildcard String Search, 144