DATA SOCIETY®

The premiere data science training for professionals

"One should look for what is and not what he thinks should be."

- Albert Einstein

Welcome to Intro to SQL

- Please confirm that the following technology is working properly:

1. Confirm that you have SQL Server software installed correctly

2. Save the following files to your desktop for this section (files previously sent to you from Data Society)
   - Claims.txt
   - crime_incidents_2013_data.csv
   - crime_incidents_2013_location.csv
   - Divisions.txt
   - Franchises.txt
   - Teams.txt

Outline

1. Overview of SQL
2. Working with data using SQL statements
3. Manipulating tables using SQL
4. Logical and mathematical operations and functions
5. SQL Server best practices

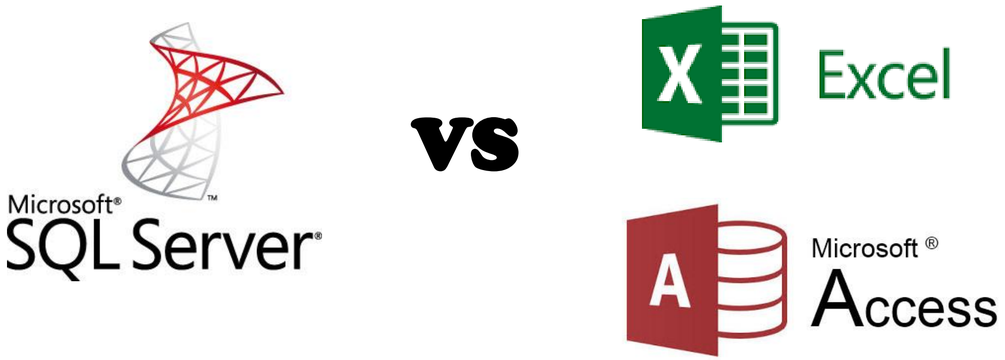
Objectives

1. Develop a basic understanding of relational databases and SQL
2. Setting up Management Studio and importing data
3. Learn foundational SQL query concepts

What is SQL?

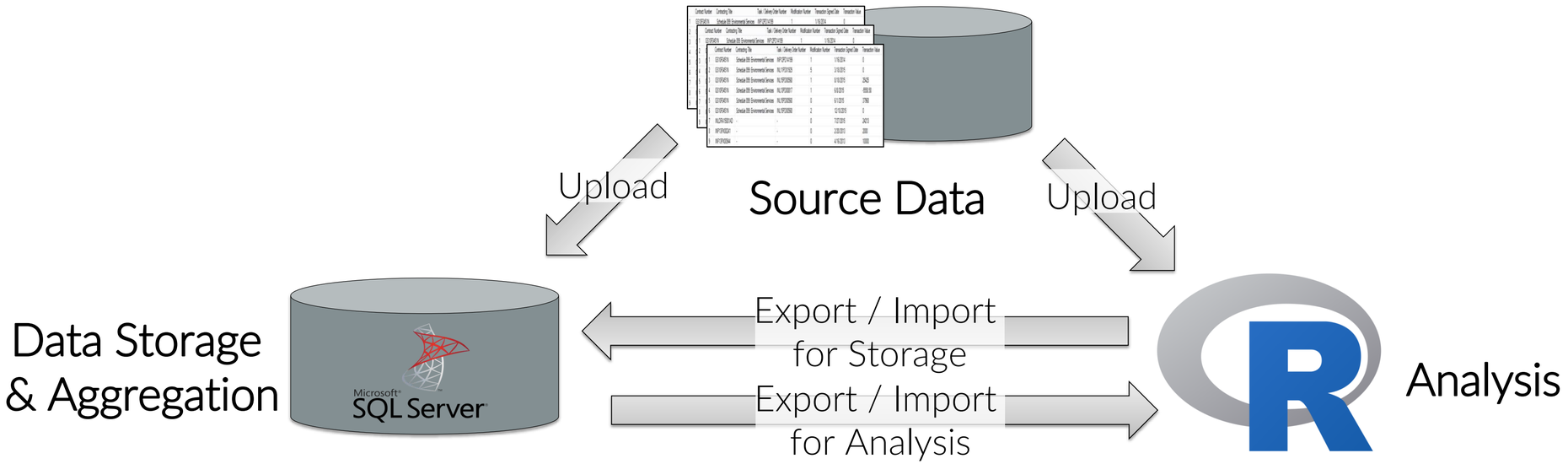
- SQL is short for Structured Query Language. It is the standard language used to communicate with most relational databases
  - SQL queries are sent to a database to ask it to perform a specific task with the data it stores

- Microsoft SQL Server is a type of relational database



Goals for Intro to SQL

- Learn to Import/Export, manipulate, combine, and aggregate data sets using SQL Management Studio
- These lessons are structured with the intent of using SQL server as an intermediary to store and aggregate data as displayed in the image below



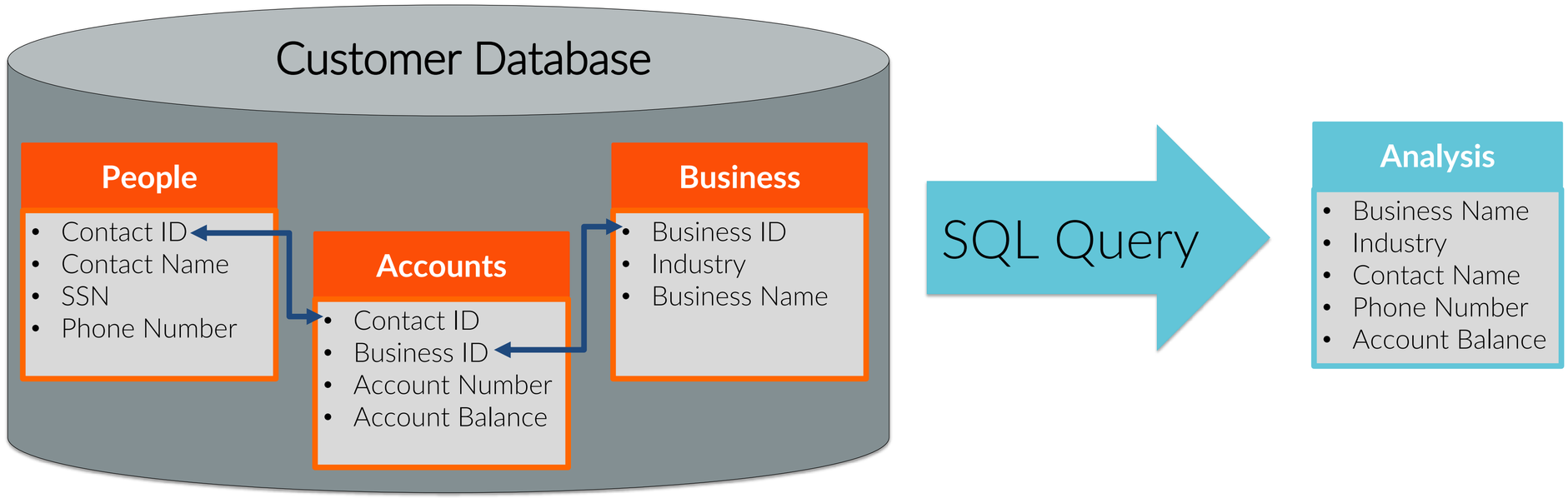
Why use SQL?

Great *Average* *Limited*

	SQL	Access	Excel
Data Size Limits	Best option for analyzing large data sets (over 1 million records)	Handles larger data sets than Excel, but can be limited by memory and space of local computers	Limited to 1,048,576 records for data sets
Manipulating Data	Multiple queries can easily be combined to coerce data from multiple data sets	Querying capabilities similar to SQL with less flexibility and capabilities	Combining data sets can be difficult and prone to manual and formula errors
Analyzing / Reporting Data	Limited built-in analysis functions and lack of built-in reporting and visualization capabilities	Built in form and report capabilities for easy reporting, but more limited set of analysis functions	Many built in analysis functions, visualizations, and formatting for easy modeling and reporting
Speed	Faster Processing	Slower processing	Slower Processing
Compatibility	Compatible with most visualization, business intelligence, and statistical analysis platforms	Compatible with many visualization, business intelligence, and statistical analysis platforms	Compatible with most visualization, business intelligence, and statistical analysis platforms
Quality Control	SQL Queries create repeatable and auditable analysis processes that can be clearly commented	Access Queries also create repeatable processes, but are often less transparent than SQL queries	Excel Analyses are harder to replicate due to manual steps that can be difficult to audit
Learning Curve	Can be easy to learn for people without a programming background	Can be easy to learn for people without a programming background	Easy to learn for any analyst

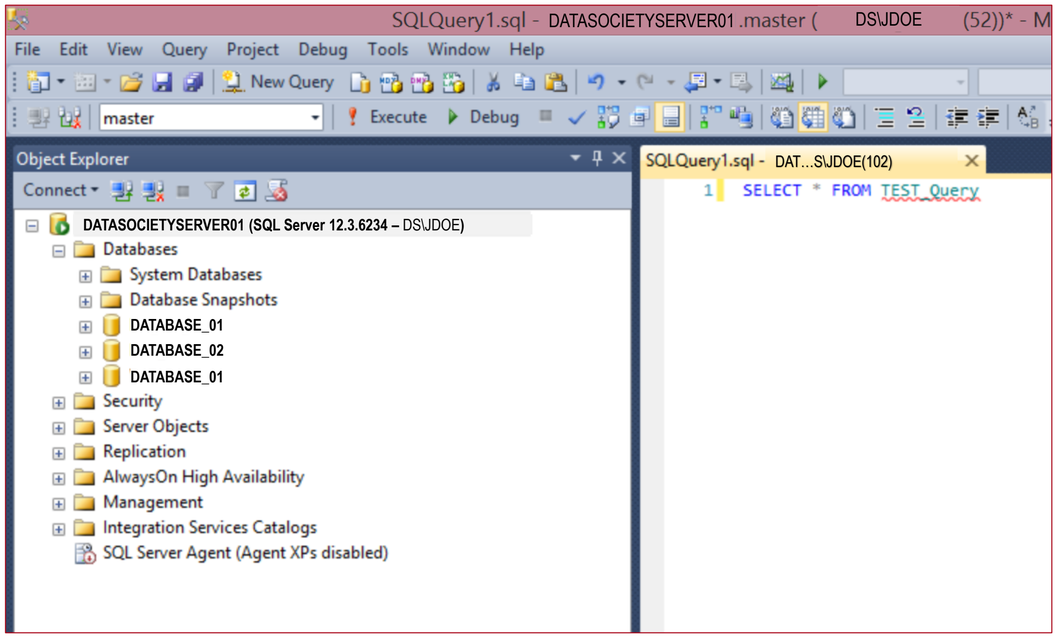
Intro to relational databases

- What is a relational database?
  - Relational databases store data in the form of tables that can be related to one another based on common attributes in the columns and rows of those tables
  - SQL Queries can leverage these relationships to rearrange the data stored in database tables



SQL Server general functionality

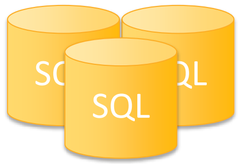
- SQL Server Management Studio is an environment used by analysts to access the capabilities of SQL Server:
  - Import/Store/Export data
  - Execute queries
    - Combine existing data
    - Update existing data
    - Extract, summarize, and/or aggregate data
    - Perform calculations on existing data
    - Create and delete data tables
  - Administrative functions
    - Database security and access control



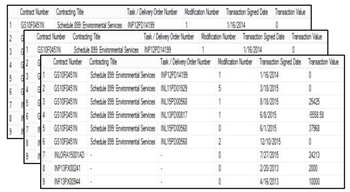
*Screen Shot from SQL Server Management Studio

SQL Server general components

- Servers

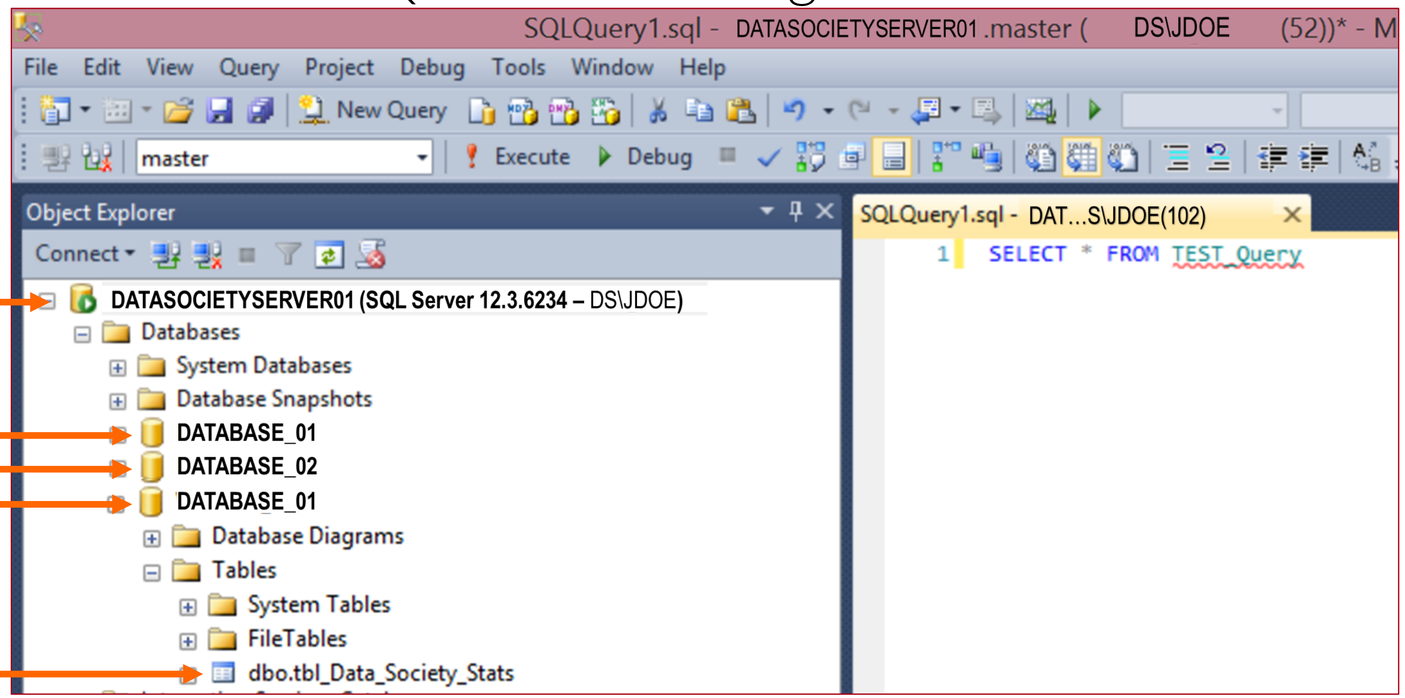


- Databases



- Tables

SQL Server Management Studio



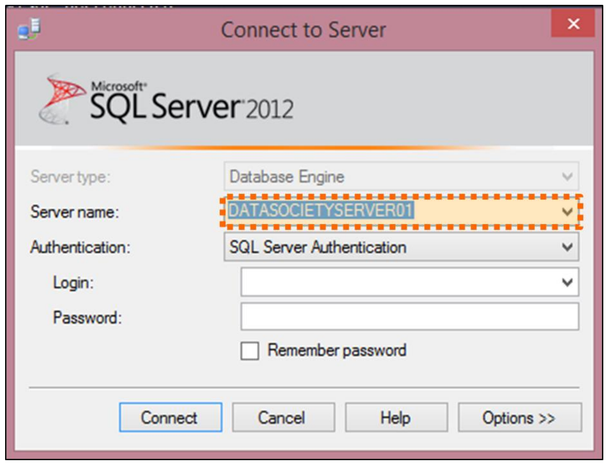
- Server – database servers are programs that provides database services to other computer programs
- Database – is a container of data/information organized into tables (and other structures) so that they can be easily managed and accessed back in same fashion.
- Table – data stored in a tabular format with rows of named columns
- SQL Server Management Studio – An application used to configure, manage, and administer components of SQL server (i.e. the user interface for accessing servers, databases, and tables to launch commands to the server)

Exercise 1 - connect to SQL

- In this activity we will ❶connect to a SQL Server, ❷create a new database, ❸open a new query window, and ❹select a SQL database to query from

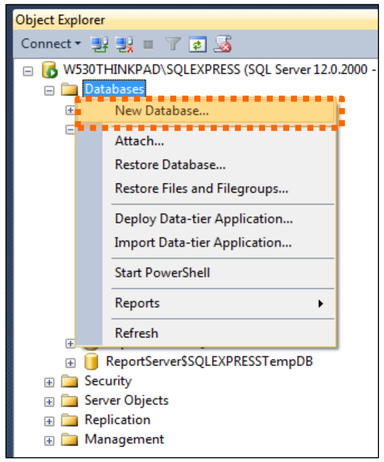
Getting Started

❶



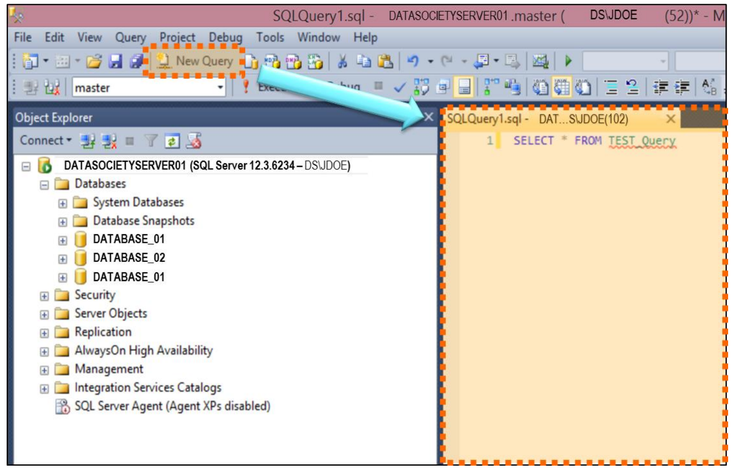
- Connect to the "XXXXXXX" Server created on your computer when SQL Express was installed

❷



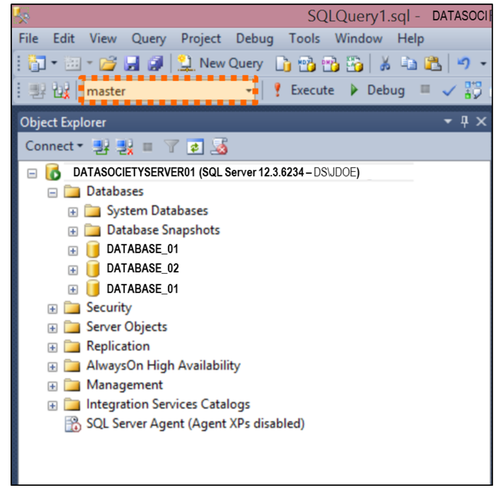
- Create a database and call it "Data_Society_SQL_Class"

❸



- Create a new SQL script by clicking on "New Query in the toolbar

❹



- Select the database for the SQL script to refer to by selecting "Data_Society_SQL_Class" in the dropdown menu

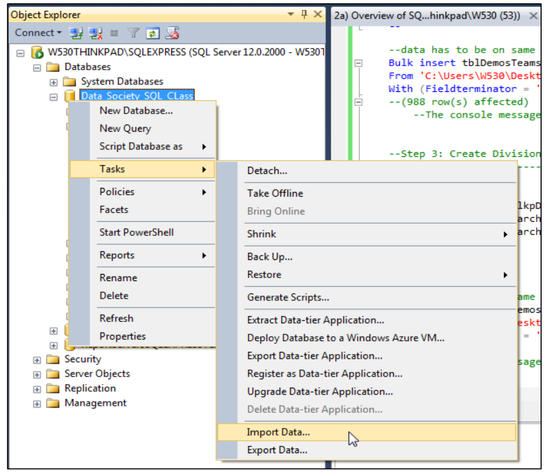
Data loading

- There are 3 methods for importing data into SQL server. This class will focus on using the Import/Export Wizard; however, it is important to understand the other methods:

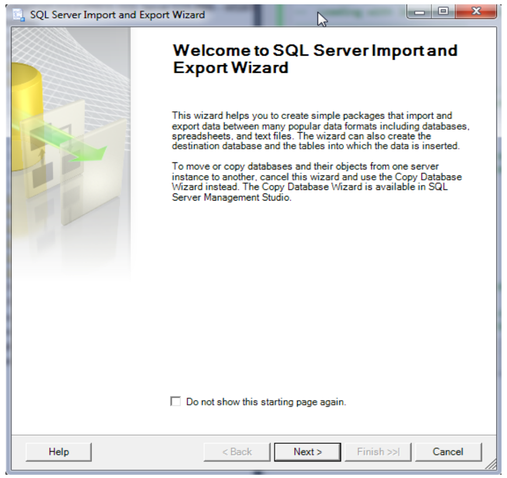
	Import/Export Wizard	Bulk Inserts	SQL Server Integration Services (SSIS)
Description	Gives users a point and click interface to upload tables into SQL server while using Management Studio	Handles larger data sets than Excel, but can be limited by memory local computers	Uses Microsoft Visual Studio to create a process flow for importing and transforming tables
Advantages	• Easy user interface • Best use is for small tables with little chance of import error: - Basic look up table containing a value and description - Sample data sets	• Creates a repeatable audit trail • Does not manipulate data to allow import to succeed	• Creates a repeatable audit trail • Dynamically import multiple files • Import similar files using loops • Perform additional transformations and queries before and after table imports
Drawbacks	• Can alter data to fit into a defined table rather than resulting in an error	• More difficult to set up and execute - relationship between the syntax, format files and data files can be hard to work with	• Requires learning Microsoft Visual Studio • More difficult to set up and has some situations requiring new syntax

Exercise 2a - loading data

- Uploading Files using the Import/Export Wizard
  - Upload 3 files using Import/Export Wizard following the directions in the SQL script accompanying these slides
    - Claims.txt
    - crime_incidents_2013_data.csv
    - crime_incidents_2013_location.csv



Open Import/Export Wizard



Follow Import Steps

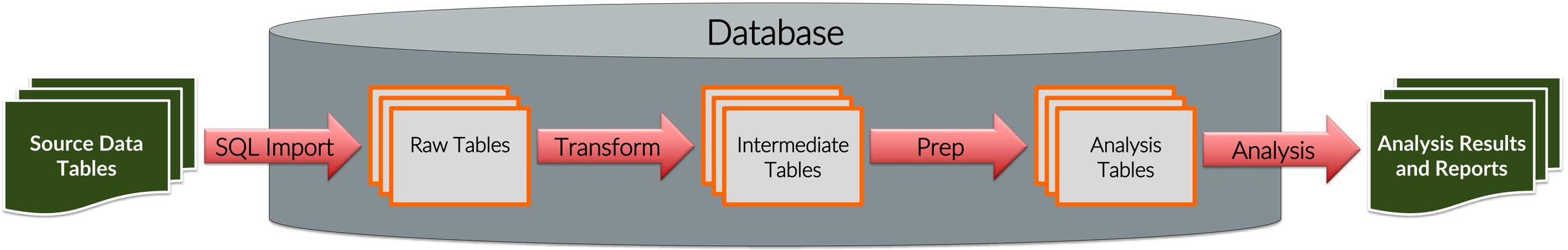
Exercise 2b - loading data

- Use code to create and import table using bulk insert
  - Create a table and import the following text files into SQL Server using the SQL script accompanying these slides
    - Franchises.txt
    - Teams.txt
    - Divisions.txt

```
--Step 1: Create Franchises Table
---------------------------------

USE [Data_Society_SQL_CLass]
GO
CREATE TABLE Data_Society_SQL_CLass.[dbo].[tblDemosFranchises](
    [FranchID] [char](3) COLLATE SQL_Latin1_General_CP1_CI_AS NULL,
    [FranchName] [varchar](50) COLLATE SQL_Latin1_General_CP1_CI_AS NULL,
    [Active] [varchar](4) COLLATE SQL_Latin1_General_CP1_CI_AS NULL
) ON [PRIMARY]


GO

--data has to be on same server copy to it first
Bulk insert dbo.tblDemosFranchises
From 'C:\Users\W530\Desktop\SQL Class\2a) Activity Files\Franchises.txt'
With (Fieldterminator = '|')
--(120 row(s) affected)
    --The console message should reflect 120 records affected if this is performed successfully

GO
```

Post import data treatment

- Often tables imported into SQL are not ready for analysis
- When preparing for an analysis think of three types of tables:
  - Raw - freshly imported into the SQL environment
  - Intermediate - mirror of raw tables with fields transformed into usable data types
    - Ex: dates might be imported as text and require transformation into a date format for analysis
  - Analysis – tables with the fields required for an analysis

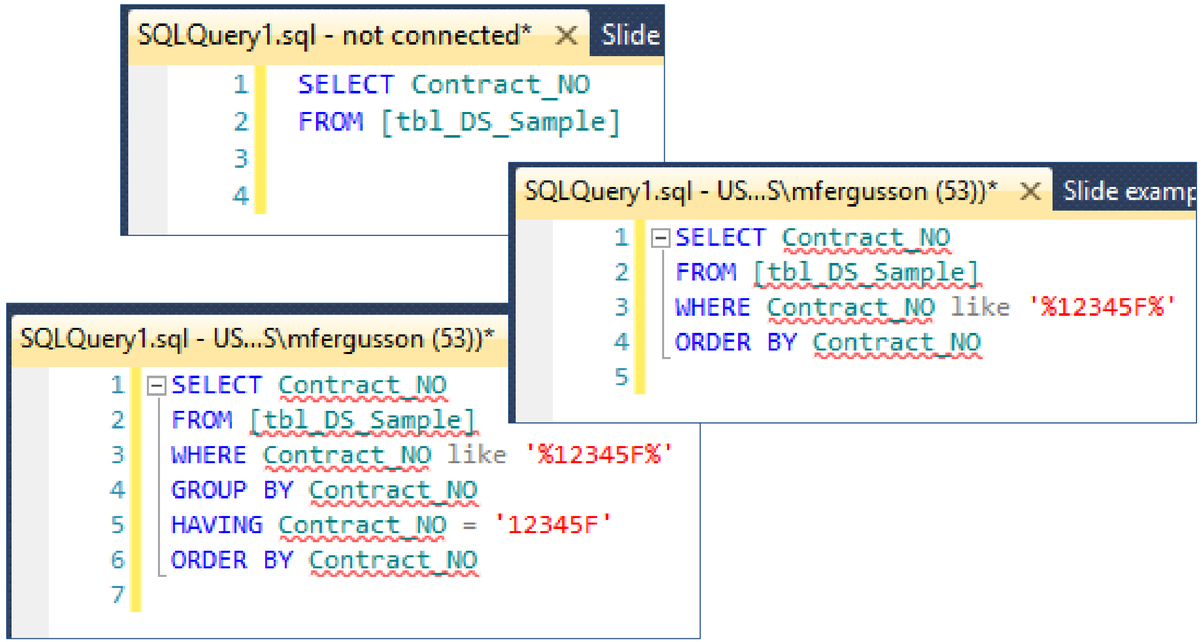


*Note: it is a best practice to name tables to distinguish these table types (Ex. tbl_Raw_Procurement_Data _2016)
- This will be covered further in the Best Practices section

Query examples and explanation

- SQL Queries follow a standard order of statements that must be followed in each query for SQL server to understand a query
  - Not all statements are required for every query, but the same order of commands must be maintained

1. SELECT
2. INTO
3. FROM
4. WHERE
5. GROUP BY
6. HAVING
7. ORDER BY



Query examples and explanation

```
SELECT Contract_NO
, Business_ID
, Sale_Amount
, Count(*) as 'Row_Count'
INTO tbl_DS_Contract_Counts
FROM tbl_DS_Sample
WHERE Contract_NO like '%123%'
GROUP BY Contract_NO
, Business_ID
, Sale_Amount
HAVING Count(*)>1
ORDER BY Contract_NO
, Sale_Amount
```

1. `SELECT` – defines the fields that will be included in the new table requested by a query from other tables and from functions
2. `INTO` – declares that a query will create a new table in the database
3. `FROM` – defines the existing tables that a query will draw data from
4. `WHERE` – filters the query results based on criteria from the original tables
5. `GROUP BY` – aggregates query results to include distinct values of the selected fields
6. `HAVING` – Similar to `WHERE`, except can contain aggregate functions. `HAVING` clauses can reference any of the items that appear in the select list
7. `ORDER BY` – sorts the query results in order by the indicated fields

Querying basics: SELECT & FROM

- The basic component of a SQL query is the `SELECT` statement which can be used to:
  1. Return text
  2. Return the results of basic math or other operations
  3. Returns 1 or more fields from a table in the `FROM` clause

`SELECT` STAR - Return 793 records from all columns

```
SELECT *
FROM tblClaims
```

`SELECT` Columns – Returns 793 records of specified columns

```
SELECT ClaimID, MemberID
FROM tblClaims
```

`SELECT TOP` X – Returns the first X Records from columns selected

```
SELECT TOP 20 *
FROM tblClaims
```

`SELECT DISTINCT` – Returns 11 unique records (no duplicates) across the columns selected

```
SELECT DISTINCT Gender
, ProviderType
FROM tblClaims
```

Querying basics: WHERE

- The `WHERE` clause is used to filter records from a table based on logical criteria defined in the WHERE clause
  - Logical operators will be discussed in more detail later, but this includes equality, ranges, and pattern matching

Where Specific Column Value – returns 3 records WHERE the diagnosis is equal to "V16.3"

```
SELECT *
FROM tblClaims
WHERE Diagnosis = 'V16.3'
```

Where value range – returns 105 records WHERE the claim payment was between $100 and $200

```
SELECT *
FROM tblClaims
WHERE CAST(Paid AS MONEY) > 100
AND CAST(Paid AS MONEY) < 200
```

Note: the Paid column was uploaded in a text formats and therefore needs to be converted to a numeric data type in order to be compared to the desired numerical range

Querying basics: GROUP BY & HAVING

- The `GROUP BY` clause is used to return unique records (no duplicates) across columns selected
  - `GROUP BY` statements are different from using `SELECT DISTINCT` because it allows aggregation functions (e.g. counts, sums, etc.)
- The `HAVING` clause can only be used in conjunction with `GROUP BY`. It acts as an additional `WHERE` clause for the results of a `GROUP BY` statement or the results of any aggregation functions used in conjunction with the grouping

GROUP BY with count – returns 11 unique records and the counts of occurrences for each combination of records in the original data set

```
SELECT Gender, ProviderType
        , COUNT(*) AS Counts
FROM tblClaims
GROUP BY Gender, ProviderType
```

GROUP BY with Count and HAVING criteria – returns 2 of the unique records from the original query based on additional filtering from the HAVING clause

```
SELECT Gender, ProviderType
, COUNT(*) AS Counts
FROM tblClaims
GROUP BY Gender, ProviderType
HAVING ProviderType = '246'
AND COUNT(*) >= 5
```

Querying basics: aggregation

- Aggregate functions are used to summarize data by rolling up a set of data items into a single item (or multiple line items when using "`GROUP BY`")
- SQL aggregation functions include:
  - `MIN`
  - `MAX`
  - `SUM`
  - `AVG`
  - `COUNT`

tbl_cash_balances

Group	Cash_Balance
A	20
B	50
B	100
D	120
E	30

```
--Aggregating on an entire table
SELECT MIN(Cash_Balance) AS 'MIN', MAX(Cash_Balance) AS 'MAX'
     , SUM(Cash_Balance) AS 'SUM', AVG(Cash_Balance) AS 'AVG'
     , COUNT(Cash_Balance) AS 'COUNT'
FROM #tbl_cash_balances

--Aggregating by groups
SELECT [GROUP]
     , MIN(Cash_Balance) AS 'MIN', MAX(Cash_Balance) AS 'MAX'
     , SUM(Cash_Balance) AS 'SUM', AVG(Cash_Balance) AS 'AVG'
     , COUNT(Cash_Balance) AS 'COUNT'
FROM #tbl_cash_balances
GROUP BY [GROUP]
```

MIN	MAX	SUM	AVG	COUNT
20	120	320	64	5

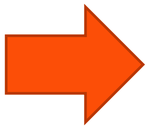
GROUP	MIN	MAX	SUM	AVG	COUNT
A	20	20	20	20	20
B	50	100	150	75	2
D	120	120	120	120	120
E	30	30	30	30	30

Querying basics: sorting data

- The `ORDER BY` query statement is used to sort query results
  - Query results can be sorted by one or more fields
  - Using `ASC` or `DESC` after a field explicitly sorts the results in ascending or descending order, respectively
    - When there is no explicit reference the sorting defaults to ascending

tbl_cash_balances

Group	Cash_Balance
A	20
B	50
B	100
D	120
E	30



```
--Sort Ascending
SELECT [GROUP], [Cash_Balance]
FROM #tbl_cash_balances
ORDER BY [Cash_Balance]
```

Group	Cash_Balance
A	20
E	30
B	50
B	100
D	120

```
--Sort Descending
SELECT [GROUP], [Cash_Balance]
FROM #tbl_cash_balances
ORDER BY [GROUP] DESC
```

Group	Cash_Balance
E	30
D	120
B	50
B	100
A	20

Note that only columns explicitly called will be sorted

Querying basics: saving results

- Saving results: If query results need to be saved for later reference or analysis there are several methods to save those results:
  - Exporting results (save results as a .csv file or copy results into a spreadsheet)
  - Using the INTO clause to create a permanent or temporary table
    - Permanent Table (use "`INTO TableName`") – Creates a permanent table that will appear in a database for all users
    - Local Temporary Table (use "`INTO #TableName`") – Visible only to their creators during the same connection to an instance of SQL Server as when the tables were first created or referenced
    - Global Temporary Table (use "`INTO ##TableName`") – Visible to any user and any connection after they are created, and are deleted when all users that are referencing the table disconnect from the instance of SQL Server

Create Permanent Table

```
SELECT *
INTO tblClaims_copy
FROM tblClaims
```

Create Local Temporary Table

```
SELECT *
INTO #tblClaims_tmp_local
FROM tblClaims
```

Create Global Temporary Table

```
SELECT *
INTO ##tblClaims_tmp_global
FROM tblClaims
```

Querying basics: deleting tables

- the `DROP TABLE` command is used to remove tables from a database
- This command will remove a table from a database permanently and this is not a reversible action
  - BE EXTREMELY CAREFUL when dropping tables in a database (especially permanent tables)
  - It is often helpful to have a statement to drop temporary tables (and sometimes permanent analysis tables) prior to the statement creating those tables in a script to streamline updates to an analysis

Dropping Tables

```
DROP TABLE tblClaims_copy
DROP TABLE #tblClaims_tmp_local
DROP TABLE ##tblClaims_tmp_global
```

Analysis Use Example

```
DROP TABLE #tblDemosFranchises
SELECT * INTO #tblDemosFranchises
FROM [dbo].[tblDemosFranchises]
--WHERE Active = 'NA'
--(50 row(s) affected) (WHERE Active = 'NA')
WHERE Active = 'Y'
--(60 row(s) affected) (WHERE Active = 'Y')
```

Querying basics: commenting code

Comments are non-executing text statements that should be used to explain queries

- Syntax:
  - In line comments – all text after "--" will not be read by SQL Server when a query is run

```
SELECT Contract_NO -- (This Text will not affect the query)
FROM [tbl_DS_Sample]
```

  - Block comments - all text between "/*" and "*/" will not be read by SQL Server when a query is run

```
SELECT Contract_NO
/* (This Text
    will not
    affect the query) */
FROM [tbl_DS_Sample]
```

Querying basics: formatting code

Formatting and commenting SQL code should be done in a consistent and repeatable manner to make your code easier to proofread and change

- Poor formatting – lines run off the screen making code hard to read

```
SELECT Contract_NO FROM [tbl_DS_Sample] WHERE Contract_NO = '54345F'
```

- Good formatting – comments and code are in logical order and easily read

```
--Step 1) Look at all Contract numbers in scope
SELECT Contract_NO
FROM [tbl_DS_Sample]
WHERE Contract_NO = '54345F'
```

Exercise 3 - writing basic SQL

- Practice writing basic SQL statements
- Remember to use proper formatting and appropriate comments

Questions?

Outline

1. Overview of SQL
2. Working with data using SQL statements
3. Manipulating tables using SQL
4. Logical and mathematical operations and functions
5. SQL Server best practices

Objectives

1. Understand, manipulate, and alter SQL Data types

2. Learn how to change data in tables

SQL data types

There are many different data types in SQL Server; however, there are 3 main data type categories:

- Numeric: contains numbers and can be used in mathematical operations
- Character: contains strings of text and can be searched for words and phrases or concatenated
- Date: contains dates and/or times that are stored as number allowing date type fields to also be used in mathematical operations

Data type examples

Numeric	Character	Date
1	A123F	1/01/2000
-2,000	Coffee is a great way to start off your day	2005-07-01 00:00:00:000
$250.35	Automobile	June 16 2013
0.0023464	Desk	Monday, January 31 2002

Examples of data types

Numeric Data Types
- Int
- Money
- Bit
- Decimal

Character Data Types
- Char
- Varchar
- Nvarchar

Date Data Types
- Datetime
- Date
- Time

NULLs

- NULL values are non-existing records in a field. They are different from "blank" or "zero-length" string values (i.e. "")
  - NULL values are excluded from aggregate functions:
    - Example: when SQL counts the number of records in the `ID_Number` column ("`COUNT(ID_Number)`") it returns a count of 2
  - NULL values do not link to one another when they are in a field being used as the relationship for combining tables

- To locate `NULL` values, use "`IS NULL`" (or "`IS NOT NULL`") in a `WHERE` clause
  - Example: "`WHERE ID_Number IS NULL`" would return only row 3 in the table below

BLANK Values

NULL Values

	ID_Number	Category	Purchase_Date
1		BMW	2002-01-01 00:00:00.000
2	1112	Mercedes	NULL
3	NULL	NULL	1970-12-01 00:00:00.000

CAST and CONVERT functions

- The `CAST` and `CONVERT` functions explicitly convert expressions of one data type into another
- Function syntax:
  - `CAST ([FIELD NAME] AS [DATA TYPE])`
  - `CONVERT ([DATA TYPE], [FIELD NAME])`

```
--Convert Text Date to a Datetime format
SELECT CONVERT(date,'1/1/2000')
```

	(No column name)
1	2000-01-01

```
--Convert Text number to a numeric format
SELECT CAST('1.0023567' AS MONEY)
```

	(No column name)
1	1.0024

Case statements

- Evaluates a list of conditions and returns one of multiple result expressions based on logical statements
- Types of logical operators
  - Simple (equality check)
  - Searched (expressions with additional logic such as >, <, `AND`, `OR`, etc.)

```
--Case Statement Syntax
CASE WHEN ... THEN
    [WHEN ... THEN]
    [ELSE]
END

--Case Statement Example
CASE WHEN Amount_1 > Amount_2 THEN '> Amount_1'
     WHEN Amount_1 <= Amount_2 THEN '<= Amount_1'
     ELSE 'N/A'
END
```

Case statement example

- Table:

#tblDemosTeams_case_ex

Yr	TeamID	Points
2001	ABC	1024
2015	ABC	910
2017	BCD	500

- Query

```
SELECT Yr
     , TeamID
     , CASE
         WHEN Points > 910 THEN 'Great Offense'
         WHEN Points < 910 THEN 'Poor Offense'
         ELSE 'Unknown'
     END AS 'Offensive Rating'
FROM #tblDemosTeams_case_ex
```

- Results

Yr	TeamID	Offensive Rating
2001	ABC	Great Offense
2015	ABC	Unknown
2017	BCD	Poor Offense

Note that the logic in this case statement does not address a score of exactly 910 which is why one result record is 'Unknown'

Exercise 4 - data types

- Using the tables you imported earlier, practice the syntax for:
  - `CASE` statements
  - `CAST` and `CONVERT` functions

Changing data

- Data tables can also be changed using specific statements that add, remove, and change data values

- These statements include:
  - `INSERT`: add records to a table
  - `DELETE`: delete records from a table
  - `TRUNCATE`: delete all records in a table
  - `UPDATE`: change values

Changing data: INSERT statement

`INSERT` statements add rows from an input source into a table

- Different syntax structures
  - Identical tables
  - Different structures
  - Values

tbl_cash_balances

Group	Cash_Balance
A	20
B	50
B	100
D	120
E	30

```
--Insert into the same columns
INSERT INTO tbl_cash_balances
VALUES('G',90)

INSERT INTO tbl_cash_balances
SELECT 'E', 90

INSERT INTO tbl_cash_balances
SELECT [GROUP], [Cash_Balance]
FROM tbl_cash_balances_insert

--Insert different columns
INSERT INTO tbl_cash_balances([GROUP])
SELECT [GROUP]
     --the [Cash_Balance] will be populated
     --with a NULL value
```

tbl_cash_balances

Group	Cash_Balance
A	20
B	50
B	100
D	120
E	30
G	90
E	90
A	20
B	50
B	100
D	120
E	30
A	*NULL*
B	*NULL*
B	*NULL*
D	*NULL*
E	*NULL*

Changing data: DELETE Statement

`DELETE` statements permanently remove rows from data tables

- Removes either an entire table or specified records

Note that `DELETE` without criteria will leave a table with columns but no records

tbl_cash_balances

Group	Cash_Balance
A	20
B	50
B	100
D	120
E	30

```
--Remove all rows
DELETE tbl_cash_balances

--Remove specific Values based on conditional logic
DELETE tbl_cash_balances
WHERE [GROUP] = 'E'
```

tbl_cash_balances

Group	Cash_Balance

tbl_cash_balances

Group	Cash_Balance
A	20
B	50
B	100
D	120

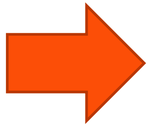
Changing data: TRUNCATE statement

`TRUNCATE` statements remove all rows from a table

- The deletions are not logged making this statement faster than the `DELETE` statement when working with large data sets

tbl_cash_balances

Group	Cash_Balance
A	20
B	50
B	100
D	120
E	30



```
--Remove all rows
TRUNCATE TABLE tbl_cash_balances
```

tbl_cash_balances

Group	Cash_Balance

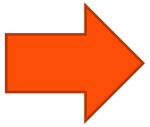
Note that `TRUNCATE` will leave a table with columns but no records similar to `DELETE` without criteria

Changing data: UPDATE statement

- Updates the values either in an entire column or based on specific values using conditional logic

tbl_cash_balances

Group	Cash_Balance
A	20
B	50
B	100
D	120
E	30



```
--Remove all rows
UPDATE tbl_cash_balances
SET [Cash_Balance] = 150

--Remove specific Values based on
--conditional logic
UPDATE tbl_cash_balances
SET [Cash_Balance] = 150
WHERE [GROUP] = 'E'
```

tbl_cash_balances

Group	Cash_Balance
A	150
B	150
B	150
D	150
E	150

tbl_cash_balances

Group	Cash_Balance
A	20
B	50
B	100
D	120
E	150

Exercise 5 - changing data

- Using the tables you imported earlier, practice the syntax for:
  - `INSERT`
  - `DELETE`
  - `TRUNCATE`
  - `UPDATE`

Questions?

Outline

1. Overview of SQL
2. Working with data using SQL statements
3. Manipulating tables using SQL
4. Logical and mathematical operations and functions
5. SQL Server best practices

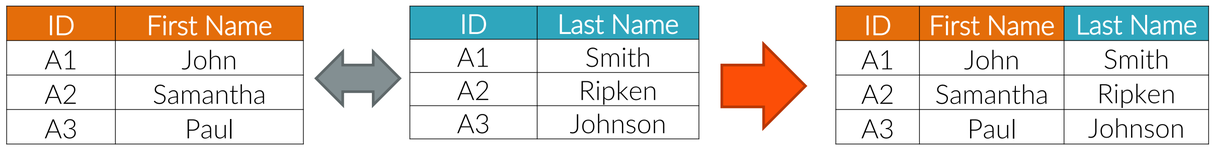
Objectives

- Understand table combination methods
- Recognize types of joins and unions
- Understand table relationships and keys
- Creating views
- Practice writing JOIN & UNION statements in SQL

SQL table combinations

SQL tables can be combined using `JOIN` or `UNION` statements to merge columns or records respectively

JOIN



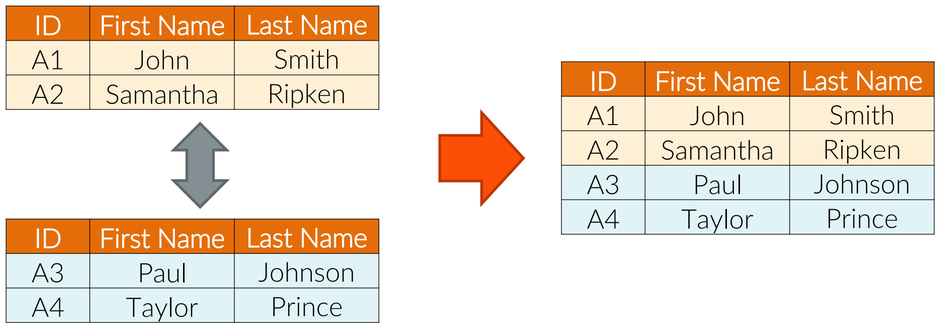
ID	First Name
A1	John
A2	Samantha
A3	Paul

ID	Last Name
A1	Smith
A2	Ripken
A3	Johnson

ID	First Name	Last Name
A1	John	Smith
A2	Samantha	Ripken
A3	Paul	Johnson

- A `JOIN` brings columns from 2 different tables into a combined table
- The combined table can have more or fewer records than its parent tables depending on both the relationship between records on the parent tables and the type of join performed

UNION



ID	First Name	Last Name
A1	John	Smith
A2	Samantha	Ripken

ID	First Name	Last Name
A3	Paul	Johnson
A4	Taylor	Prince

ID	First Name	Last Name
A1	John	Smith
A2	Samantha	Ripken
A3	Paul	Johnson
A4	Taylor	Prince

- A `UNION` appends records from 2 tables into a combined table
- The combined table will have more records than its parent tables assuming no filtering is applied to either table

Using table aliases

A table alias allows a table to be temporarily renamed within the scope of an individual query. This makes queries easier to read and limits the amount of code required for a query to execute.

Tables

[First]

ID	First Name
A1	John
A2	Samantha
A3	Paul

[Last]

ID	Last Name
A1	Smith
A3	Johnson
A4	Adler

Result

ID	First Name	ID	Last Name
A1	John	A1	Smith
A3	Paul	A3	Johnson

Code

```
SELECT First.ID, First.[First Name]
,Last.ID , Last.[Last Name]
FROM [First]
JOIN [Last]
    ON First.ID = Last.ID
```

Vs.

```
SELECT A.ID, A.[First Name]
,B.ID , B.[Last Name]
FROM [First] AS A
JOIN [Last] AS B
    ON A.ID = B.ID
```

Joins: code structure

The illustration below demonstrates how the `JOIN` code relates these tables. In more complex joins, tables are referred to as `LEFT` and `RIGHT` tables based on their order in the SQL statement, which will impact the records returned in result sets.

Code

```
SELECT A.*, B.[Last Name]
FROM [First] A
JOIN [Last] B
    ON A.ID = B.ID
```

The **LEFT** table follows **FROM**

The **RIGHT** table follows **JOIN**

Tables & Results

ID	First Name
A1	John
A2	Samantha
A3	Paul

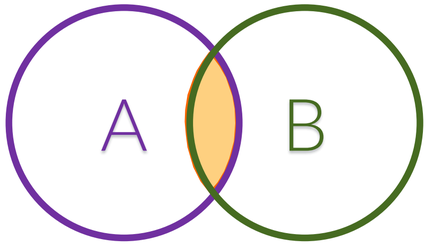
[First]
(LEFT TABLE)

ID	Last Name
A1	Smith
A3	Johnson
A4	Adler

[Last]
(LEFT TABLE)

Result

ID	First Name	Last Name
A1	John	Smith
A3	Paul	Johnson



Joins: INNER JOIN

An `INNER JOIN` between 2 tables returns the intersection between those 2 tables

Tables

[First]

ID	First Name
A1	John
A2	Samantha
A3	Paul

[Last]

ID	Last Name
A1	Smith
A3	Johnson
A4	Adler

Result

ID	First Name	Last Name
A1	John	Smith
A3	Paul	Johnson

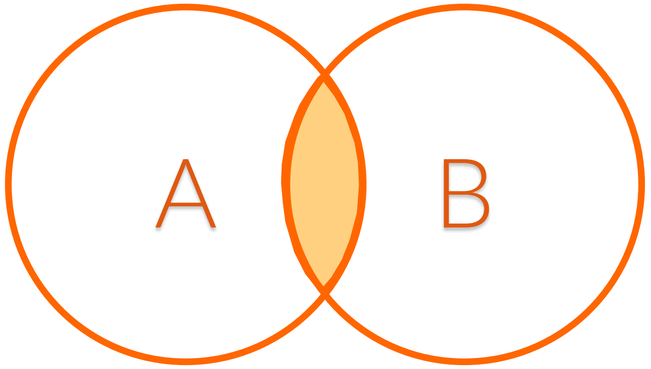
Code

```
SELECT A.*, B.[Last Name]
FROM [First] A
INNER JOIN [Last] B
    ON A.ID = B.ID
```

or

```
SELECT A.*, B.[Last Name]
FROM [First] A
JOIN [Last] B
    ON A.ID = B.ID
```

Logic



Connection

A ⟷ B

Joins: LEFT OUTER JOIN

A `LEFT OUTER JOIN` between 2 tables returns all records from the table in the initial table and the intersection between those 2 tables. Where there is no intersection NULL values are populated in columns selected from the joined table.

Tables

[First]

ID	First Name
A1	John
A2	Samantha
A3	Paul

[Last]

ID	Last Name
A1	Smith
A3	Johnson
A4	Adler

Result

ID	First Name	Last Name
A1	John	Smith
A2	Samantha	*NULL*
A3	Paul	Johnson

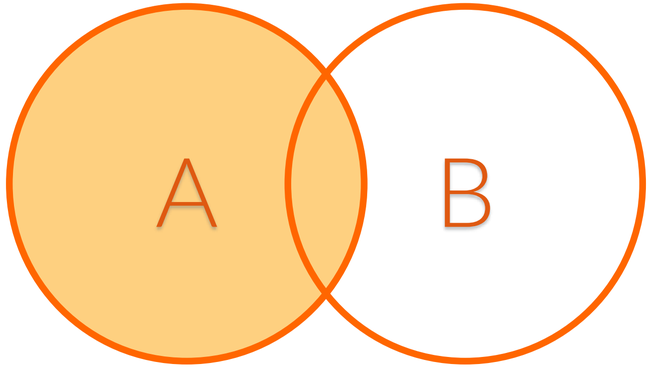
Code

```
SELECT A.*, B.[Last Name]
FROM [First] A
LEFT OUTER JOIN [Last] B
    ON A.ID = B.ID
```

or

```
SELECT A.*, B.[Last Name]
FROM [First] A
LEFT JOIN [Last] B
    ON A.ID = B.ID
```

Logic



Connection

A ⟷ B

A ⟷ NULL

Joins: LEFT OUTER JOIN (exclude)

A `LEFT OUTER JOIN` with exclusion between 2 tables returns only records from the original table with no intersection to the initial table.

Tables

[First]

ID	First Name
A1	John
A2	Samantha
A3	Paul

[Last]

ID	Last Name
A1	Smith
A3	Johnson
A4	Adler

Result

ID	First Name	Last Name
A2	Samantha	*NULL*

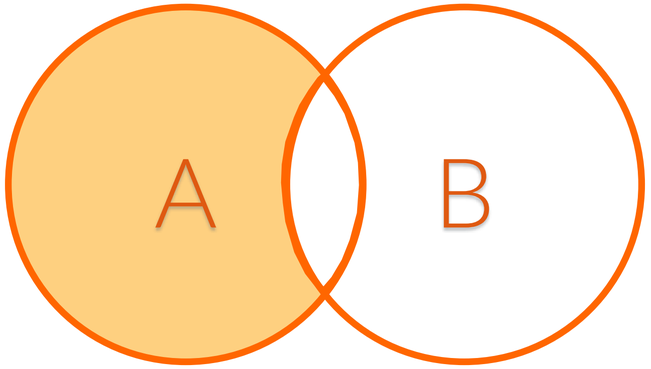
Code

```
SELECT A.*, B.[Last Name]
FROM [First] A
LEFT OUTER JOIN [Last] B
    ON A.ID = B.ID
WHERE B.ID IS NULL
```

or

```
SELECT A.*, B.[Last Name]
FROM [First] A
LEFT JOIN [Last] B
    ON A.ID = B.ID
WHERE B.ID IS NULL
```

Logic



Connection

A ⟷ NULL

Joins: RIGHT OUTER JOIN

A `RIGHT OUTER JOIN` between 2 tables returns all records from the joined table and the intersection between those 2 tables. Where there is no intersection NULL values are populated in columns selected from the table in the initial table.

Tables

[First]

ID	First Name
A1	John
A2	Samantha
A3	Paul

[Last]

ID	Last Name
A1	Smith
A3	Johnson
A4	Adler

Result

ID	First Name	Last Name
A1	John	Smith
A3	Paul	Johnson
A4	*NULL*	Adler

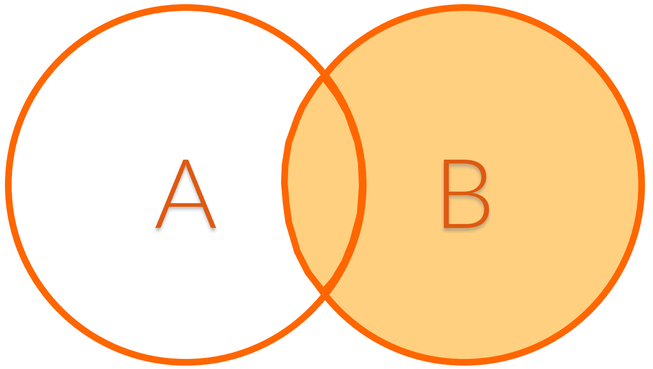
Code

```
SELECT B.ID, A.[First Name]
, B.[Last Name]
FROM [First] A
RIGHT OUTER JOIN [Last] B
    ON A.ID = B.ID
```

or

```
SELECT B.ID, A.[First Name]
, B.[Last Name]]
FROM [First] A
RIGHT JOIN [Last] B
    ON A.ID = B.ID
```

Logic



Connection

A ⟷ B

NULL ⟷ B

Joins: RIGHT OUTER JOIN (exclude)

A `RIGHT OUTER JOIN` with exclusion between 2 tables returns only records from the joined table with no intersection to the initial table.

Tables

[First]

ID	First Name
A1	John
A2	Samantha
A3	Paul

[Last]

ID	Last Name
A1	Smith
A3	Johnson
A4	Adler

Result

ID	First Name	Last Name
A4	*NULL*	Adler

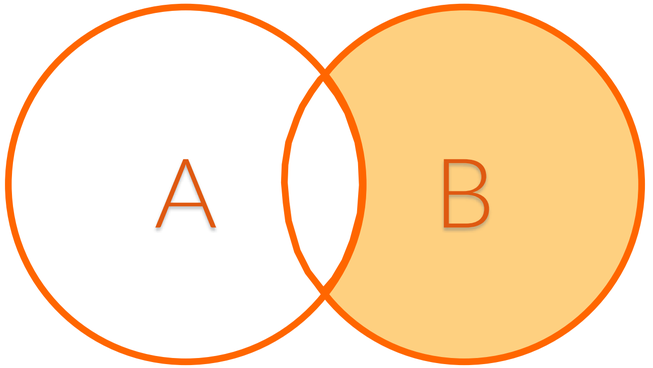
Code

```
SELECT B.ID, A.[First Name]
, B.[Last Name]
FROM [First] A
RIGHT OUTER JOIN [Last] B
    ON A.ID = B.ID
WHERE A.ID IS NULL
```

or

```
SELECT B.ID, A.[First Name]
, B.[Last Name]
FROM [First] A
RIGHT JOIN [Last] B
    ON A.ID = B.ID
WHERE A.ID IS NULL
```

Logic



Connection

NULL ⟷ B

Joins: FULL OUTER JOIN

A FULL OUTER JOIN between 2 tables returns all records from both tables including their intersection.

Tables

[First]

ID	First Name
A1	John
A2	Samantha
A3	Paul

[Last]

ID	Last Name
A1	Smith
A3	Johnson
A4	Adler

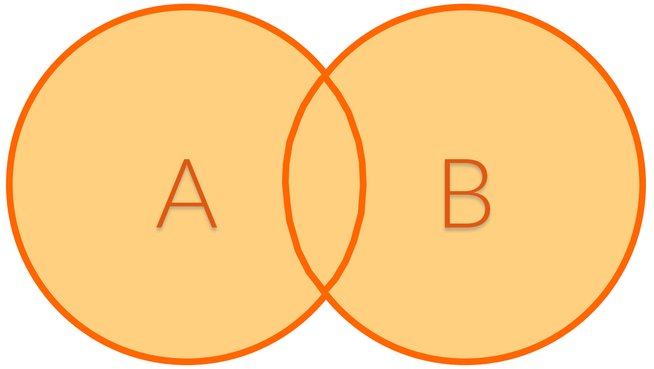
Result

ID	First Name	Last Name
A1	John	Smith
A2	Samantha	*NULL*
A3	Paul	Johnson
A4	*NULL*	Adler

Code

```
SELECT CASE WHEN A.ID IS NULL
            THEN B.ID
            ELSE A.ID
       END as ID
FROM [First] A
FULL OUTER JOIN [Last] B
    ON A.ID = B.ID
```

Logic



Connection

A ⟷ B

A ⟷ NULL

NULL ⟷ B

Joins: FULL OUTER JOIN (exclude)

A `FULL OUTER JOIN` with exclusion between 2 tables returns only records from both tables excluding their intersection.

Tables

[First]

ID	First Name
A1	John
A2	Samantha
A3	Paul

[Last]

ID	Last Name
A1	Smith
A3	Johnson
A4	Adler

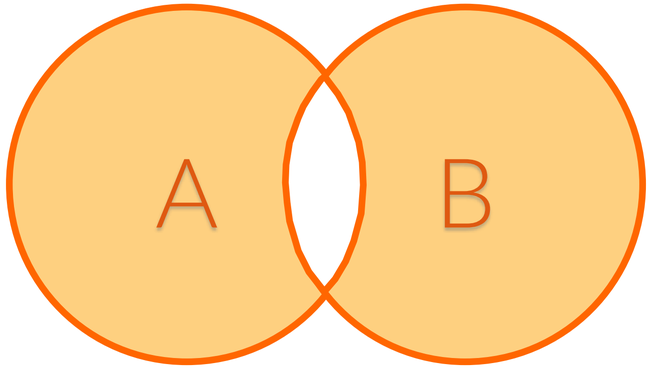
Result

ID	First Name	Last Name
A2	Samantha	*NULL*
A4	*NULL*	Adler

Code

```
SELECT CASE WHEN A.ID IS NULL
            THEN B.ID
            ELSE A.ID
       END as ID
, A.[First Name] , B.[Last Name]
FROM [First] A
FULL OUTER JOIN [Last] B
    ON A.ID = B.ID
WHERE A.ID is NULL
    OR B.ID IS NULL
```

Logic



Connection

A ⟷ NULL

NULL ⟷ B

Joins: CROSS JOIN

A `CROSS JOIN` between 2 tables returns every combination of records from one table to the other.

Tables

[First]

ID	First Name
A1	John
A2	Samantha
A3	Paul

[Last]

ID	Last Name
A1	Smith
A3	Johnson
A4	Adler

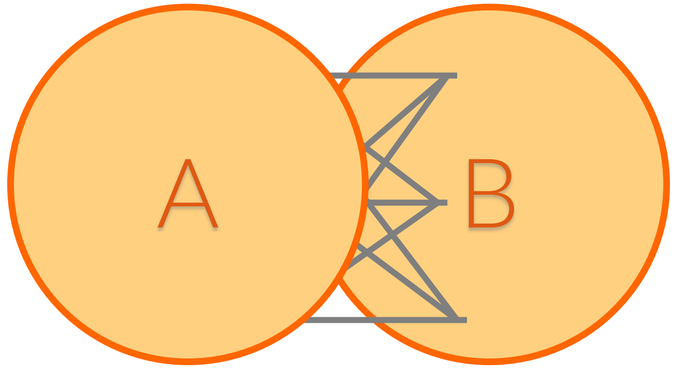
Result

First Name	Last Name
John	Smith
Samantha	Smith
Paul	Smith
John	Johnson
Samantha	Johnson
Paul	Johnson
John	Adler
Samantha	Adler
Paul	Adler

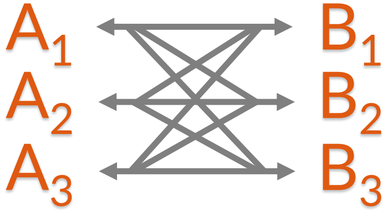
Code

```
SELECT CASE WHEN A.ID IS NULL
            THEN B.ID
            ELSE A.ID
SELECT A.[First Name]
, B.[Last Name]
FROM [First] A
CROSS JOIN [Last] B
```

Logic



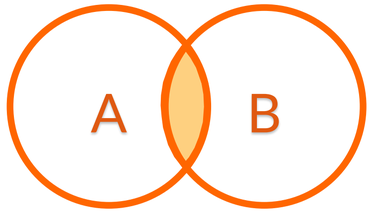
Connection



Joining logic comparison

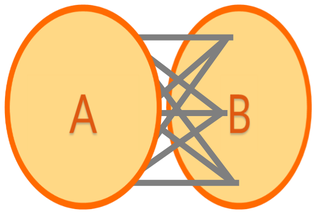
Inner Joins

Inner Join



A ⟷ B

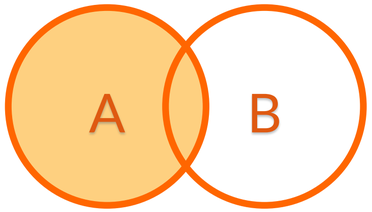
Cross Join



A_1 ⟷ B_1
A_2 ⟷ B_2
A_3 ⟷ B_3

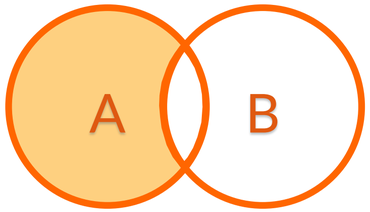
Outer Joins

Left Outer Join



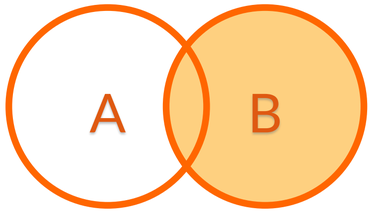
A ⟷ B
A ⟷ NULL

Left Outer Join (w/ Exclusion)



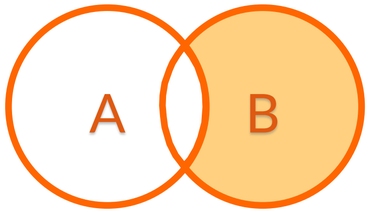
A ⟷ NULL

Right Outer Join



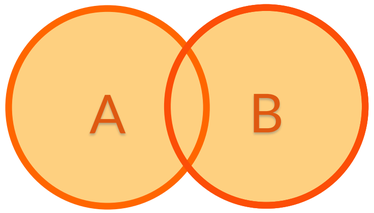
A ⟷ B
NULL ⟷ B

Right Outer Join (w/ Exclusion)



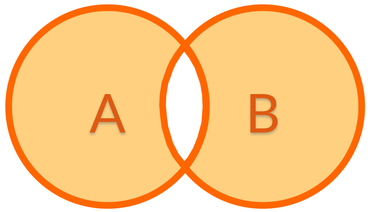
NULL ⟷ B

Full Outer Join



A ⟷ B
NULL ⟷ B
A ⟷ NULL

Full Outer Join (w/ Exclusion)



NULL ⟷ B
A ⟷ NULL

Union statements

- Unions allow records from the same fields to be appended to one another

Tables

[Dec_01_Sales]

Date	Revenue
12/10/01	$25,000
12/19/01	$120,000
12/23/01	$10,000

[Jan_02_Sales]

Date	Revenue
01/10/02	$35,000
01/19/02	$12,000
01/23/02	$110,000

Result

Date	Revenue
12/10/01	$25,000
12/19/01	$120,000
12/23/01	$10,000
01/10/01	$35,000
01/19/01	$12,000
01/23/01	$110,000

Code

```
SELECT Date, Revenue
FROM Dec_01_Sales
UNION ALL
SELECT Date, Revenue
FROM Jan_02_Sales
```

or

```
SELECT Date, Revenue
FROM Dec_01_Sales
UNION
SELECT Date, Revenue
FROM Jan_02_Sales
```

UNION vs UNION ALL

Using only `UNION` to combine tables will remove duplicate records between and within tables, whereas using `UNION ALL` will combine all records with no further alterations. Using `UNION ALL` typically runs significantly faster than a UNION unless the dataset is small and removing duplicates is necessary.

UNION

[Dec_01_Sales]

Date	Revenue
12/10/01	$25,000
12/19/01	$120,000
12/19/01	$120,000
12/19/01	$120,000
12/23/01	$10,000

[Jan_02_Sales]

Date	Revenue
01/10/02	$35,000
12/23/01	$10,000
01/19/02	$12,000
01/23/02	$110,000

Result

Date	Revenue
12/10/01	$25,000
12/19/01	$120,000
12/23/01	$10,000
01/10/01	$35,000
01/19/01	$12,000
01/23/01	$110,000

UNION ALL

[Dec_01_Sales]

Date	Revenue
12/10/01	$25,000
12/19/01	$120,000
12/19/01	$120,000
12/19/01	$120,000
12/23/01	$10,000

[Jan_02_Sales]

Date	Revenue
01/10/02	$35,000
12/23/01	$10,000
01/19/02	$12,000
01/23/02	$110,000

Result

Date	Revenue
12/10/01	$25,000
12/19/01	$120,000
12/19/01	$120,000
12/19/01	$120,000
12/23/01	$10,000
01/10/01	$35,000
12/23/01	$10,000
01/19/01	$12,000
01/23/01	$110,000

Table relationships

The previous examples use tables with one-to-one relationships; however, additional considerations should be taken when other table relationships exist. These relationships include:

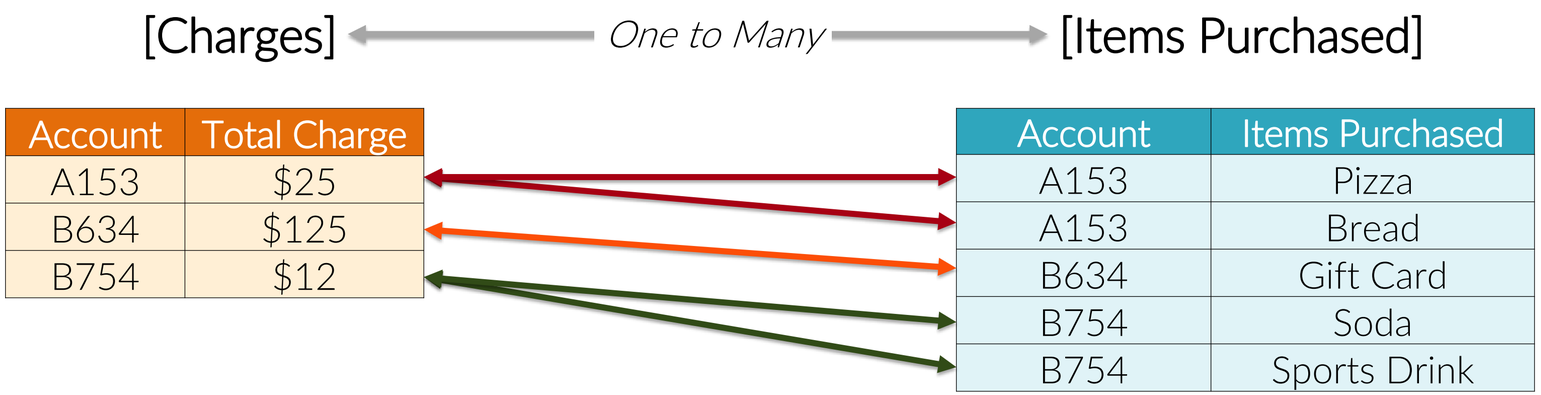
- One-to-One - each record in one table will have no more than one matching record in a second table, and vice versa.

- One-to-Many - each record in one table can have many matching records in a second table; however, each record in the second table can only have one matching record in the first table.

- Many-to-Many – records in one table can have many matching records in a second table, and vice versa.

One-to-Many relationships

One-to-Many relationships exist when each records in one table may relate to numerous records in another table.

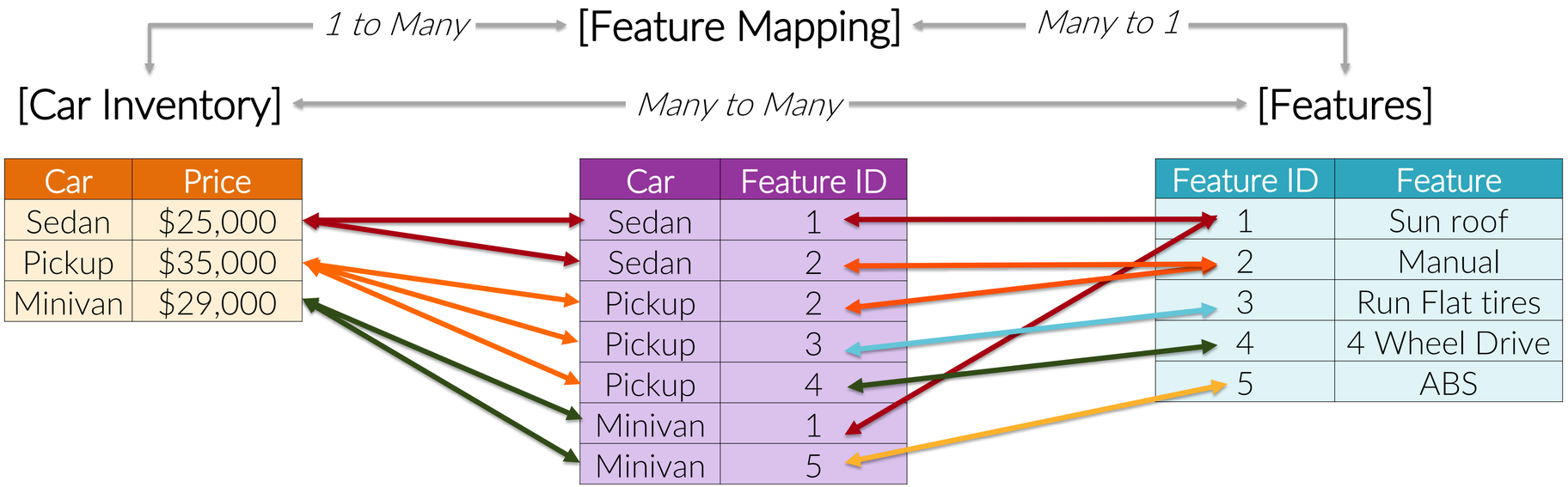


Account	Total Charge
A153	$25
B634	$125
B754	$12

Account	Items Purchased
A153	Pizza
A153	Bread
B634	Gift Card
B754	Soda
B754	Sports Drink

Many-to-Many relationships

Many-to-Many relationships exist when records in one table can have many matching records in a second table, and vice versa.



Multiple JOIN statements

- A `JOIN` statement is limited to two combining 2 tables, but a query can have multiple `JOIN` statements. Not all tables have to return a value. The order of table joins is important (see ordering example on the next slide).

Tables

[First]

ID	First Name
A1	John
A2	Samantha
A3	Paul

[Last]

ID	Last Name
A1	Smith
A3	Johnson
A4	Adler

[PhoneNo]

ID	Phone
A1	555-111-4564
A2	555-222-6781
A3	555-333-9754
A4	555-444-3456

Result

ID	First Name	Last Name	Phone
A1	John	Smith	555-111-4564
A2	Samantha	*NULL*	555-222-6781
A3	Paul	Johnson	555-333-9754
A4	*NULL*	Adler	555-444-3456

Code

```
SELECT A.[ID],
,B.[First Name]
,C.[Last Name]
,A.[Phone]
FROM PhoneNo A
LEFT JOIN [First] B
    ON A.ID = B.ID
LEFT JOIN [Last] C
    ON A.ID = C.ID
```

Multiple JOIN statements: joins order

The order of joins will dictate what information is produced in the results. Understanding the relationships (or lack thereof) between data in tables becomes very important as tables are combined.

Tables

ClientNames

ID	Client
1	XYZ
2	ABD
5	MYR

InvoiceNo

ID	Invoice
1	H123
4	B346
5	MZ905

PhoneNo

Client	PhoneNo
XYZ	555-567-5585
ABD	555-675-1900
QAR	555-111-1801

Code

```
SELECT A.Client, B.Invoice, C.PhoneNo
FROM ClientNames A
LEFT JOIN InvoiceNo B
    ON A.ID = B.ID
LEFT JOIN PhoneNo C
    ON A.Client = C.Client
```

VS

```
SELECT B.Client, A.Invoice, C.PhoneNo
FROM InvoiceNo A
LEFT JOIN ClientNames B
    ON A.ID = B.ID
LEFT JOIN PhoneNo C
    ON B.Client = C.Client
```

Results

Client	Invoice	PhoneNo
XYZ	H123	555-567-5585
ABD	*NULL*	555-675-1900
MYR	MZ905	*NULL*

VS

Client	Invoice	PhoneNo
XYZ	H123	555-567-5585
NULL	B346	*NULL*
MYR	MZ905	*NULL*

Multi criteria joins

Joins can use more than 1 criteria as well as multiple fields to combine data from tables. The join logic can also use static values.

Tables

DS_Supply

City	State	Data Scientists
Springfield	Missouri	13
Springfield	Virginia	100
Vienna	Virginia	75
Springfield	New York	25
New York	New York	450

DS_Demand

City	State	Data Scientist Demand
Springfield	Missouri	25
Springfield	Virginia	75
Vienna	Virginia	50
Springfield	New York	20
New York	New York	500

Code

```
SELECT A.City, A.State
      , A.[Data Scientists]
      , B.[Data Scientist
Demand]
      , 1.00 * A.[Data
Scientists] / B.[Data
Scientist Demand] as 'Demand
Filled %'
FROM DS_Supply A
LEFT JOIN DS_Demand B
ON A.City = B.City
    AND A.State = B.State
    AND A.State = 'New York'
```

Results

City	State	Data Scientists	Data Scientist Demand	Demand Filled %
Springfield	Missouri	13	*NULL*	*NULL*
Springfield	Virginia	100	*NULL*	*NULL*
Vienna	Virginia	75	*NULL*	*NULL*
Springfield	New York	25	20	125%
New York	New York	450	500	90%

Non-equivalent joins

Earlier examples illustrate equivalent joins (Value 1 = Value 2), but joins can also use non equivalent logic using operators such as **>**, **<**, **>=**, and **<=.**

Tables

ArticlePages

Article	Beginning Page	Ending Page
Golf	1	5
Philosophy	6	10
Science	11	25

Excerpts

Page	Text
4	Tiger won again
8	I think therefore I am
22	The chemical is inert

Code

```
SELECT A.Article, B.Text
FROM ArticlePages A
LEFT JOIN Excerpts B
    ON A.[Beginning Page] <= B.Page
    AND A.[Ending Page] >= B.Page
```

Results

Article	Text
Golf	Tiger won again
Philosophy	I think therefore I am
Science	The chemical is inert

Self joins

SQL also allows a table to be joined to itself. This can be useful in situations such as creating cumulative sums over time.

Tables

Financials

Period	Revenue	Profit
1	$ 10,000	$ -500
2	$ 12,000	$ -1,000
3	$ 11,500	$ 200
4	$ 13,000	$ 250
5	$ 17,500	$ 400
6	$ 14,000	$ 500
7	$ 9,000	$ -100
8	$ 20,500	$ 1,000
9	$ 12,000	$ 1,200
10	$ 15,000	$ 800
11	$ 19,000	$ 950
12	$ 21,000	$ 1,250

Code

```
SELECT A.Period
, SUM(B.Revenue) as Cumulative_Revenue
, SUM(B.Profit) as Cumulative_Profit
FROM #Financials A
JOIN #Financials B
ON A.Period >= B.Period
GROUP BY A.Period
```

Results

Period	Cumulative Revenue	Cumulative Profit
1	$ 10,000	$ -500
2	$ 22,000	$ -1,500
3	$ 33,500	$ -1,300
4	$ 46,500	$ -1,050
5	$ 64,000	$ -650
6	$ 78,000	$ -150
7	$ 87,000	$ -250
8	$ 107,500	$ 750
9	$ 119,500	$ 1,950
10	$ 134,500	$ 2,750
11	$ 153,500	$ 3,700
12	$ 174,500	$ 4,950

Quality control tips for joins

- Pay attention to the row counts!
  - Is the number of rows what you expect?
    - A common problem is assuming that a field or combination of fields represents a unique value. When that is not the case you can see an increase in record count.
    - Another common issue is incorrectly setting criteria of a join or WHERE clause and excluding more records than was originally intended.
  - Use simple queries on each original table to compare the number of records with given criteria to the number in the joined table.
  - It is also helpful to use left and right joins Use LEFT and/or RIGHT joins to determine the overlap between tables and whether:
    - The fields you are using to link the original tables are appropriate.
    - The tables themselves are appropriate to combine for an analysis (2 tables can appear to be similar data from field names but actually contain little to no overlap).

SQL views

A SQL view is a virtual table that is the stored results of an underlying SQL statement. That SQL statement can be a table or query. Views can be valuable for:

1. Creating simplicity by hiding complex queries from end users of data
2. Creating security through hiding fields with private information and/or preventing changes to base tables
3. Preventing redundancy & increase consistency by providing a common source for data users

[First]

ID	First Name
A1	John
A2	Samantha
A3	Paul

[Last]

ID	Last Name
A1	Smith
A3	Johnson
A4	Adler

```
CREATE VIEW First_Last AS
SELECT A.*, B.[Last Name]
FROM First A
JOIN Last B
    ON A.ID = B.ID
```

```
INSERT INTO First
SELECT 'A4', 'Jeremy'

INSERT INTO Last
SELECT 'A2', 'Satatoga'
```

```
SELECT *
FROM First_Last
```

Initial Result

ID	First Name	Last Name
A1	John	Smith
A3	Paul	Johnson

```
SELECT *
FROM First_Last
```

Post Insert Result

ID	First Name	Last Name
A1	John	Smith
A2	Samantha	Saratoga
A3	Paul	Johnson
A4	Jeremy	Adler

Exercise 6 - practice JOINs

- Practice writing JOIN statements
- Remember to use proper formatting and appropriate comments

Questions?

Outline

1. Overview of SQL
2. Working with data using SQL statements
3. Manipulating tables using SQL
4. Logical and mathematical operations and functions
5. SQL Server best practices

Objectives

- Understand logical operators in SQL and how they are used
- Understand the use and syntax of various functions

Logical operators: comparisons

Logical operators test whether or not a condition is true. They are generally used in `CASE` Statements, `JOIN`s, `WHERE` clauses, `HAVING` clauses.

Comparison Operators:

- "`=`" and "`<>`" / "`!=`" – "Equal to" and "Not Equal to"
- "`<`" and "`<=`" / "`!>`" - "Less than" and "Less than or equal to"
- "`>`" and "`>=`" / "`!<`" - "Greater than" and "Greater than or equal to"
- `IN` - used to match values in a field to a list of values
- `BETWEEN` - used to specify an inclusive range (lower and upper values are searched as well as the values in between)
- `LIKE` - used to identify patterns in character fields
- `NOT` - used to negate conditions

Logical operators: IN

The `IN` clause is used to compare a value to a list of possible values.

Tables

[Rainbow]

Rainbow Colors	Order
Red	1
Orange	2
Yellow	3
Green	4
Blue	5
Indigo	6
Violet	7

[Favorite]

Favorite Colors
Blue
Indigo
Orange
Black

Code

Example 1 - Field to Field comparison

```
SELECT *
FROM Rainbow
WHERE [Rainbow Colors] IN ( SELECT
[Favorite Colors] FROM Favorite )
```

Example 2 - Field to list comparison

```
SELECT *
FROM Rainbow
WHERE [Rainbow Colors] IN('Blue','Violet')
```

Example 3 - CASE statement using IN

```
SELECT *, CASE WHEN [Rainbow Colors] IN (
SELECT [Favorite Colors] FROM Favorite )
THEN 'Y' ELSE 'N' END as Fav_Flag
FROM Rainbow
```

Results

Example 1

Rainbow Colors	Order
Orange	2
Blue	5
Indigo	6

Example 2

Rainbow Colors	Order
Blue	5
Violet	7

Example 3

Rainbow Colors	Order	Fav_Flag
Red	1	N
Orange	2	Y
Yellow	3	N
Green	4	N
Blue	5	Y
Indigo	6	Y
Violet	7	N

Logical operators: BETWEEN

The `BETWEEN` clause can be used to determine if a value falls between an upper and lower bound. For SQL, the `BETWEEN` statement is inclusive of the upper and lower bounds (i.e. Between is true when [lower bound] <= value <= [upper bound]).

Tables

[Sales]

Date	Amount	Category
12/1/15	150	C
12/25/15	20	A
1/15/15	35	D
1/31/16	75	B

Code

Example 1 - Text

```
SELECT * FROM Sales
WHERE Category BETWEEN 'B' AND 'D'
```

Example 2 - Numeric

```
SELECT * FROM Sales
WHERE Amount BETWEEN 25 AND 75
```

Example 3 - Dates

```
SELECT * FROM Sales
WHERE Date BETWEEN '2015-12-25'
AND '2016-01-20'
```

Results

Example 1

Date	Amount	Category
12/1/2015	150	C
1/15/2016	35	D
1/31/2016	75	B

Example 2

Date	Amount	Category
1/15/2016	35	D
1/31/2016	75	B

Example 3

Date	Amount	Category
12/25/2015	20	A
1/15/2016	35	D
1/31/2016	75	B

Logical operators: LIKE

The LIKE clause matches patterns of text to a character field. It is important to understand how to leverage special characters (“%” , “_”, “[”, “]”, and “^”) to accurately match patterns.

- Wildcard - characters that substitute for any other character in a string
  - “%” - used to represent 0 or more characters and is typically used before and/or after the part of text being searched for to look for that text anywhere in the character string
  - “_” - used to represent 1 character
- Specified patterns - Brackets “[]” can be used to specify lists or ranges of characters or numbers that should be represented by a character in a pattern
  - Using “^” after the opening bracket changes exclude the characters following it (ex. “[^m]” matches any letter other than “m”)
- Escape - The special characters listed above (“%” , “_”, “[”, and “]”) need to be treated differently than others. They either need to be included in brackets or placed after an escape character

Logical operators: LIKE examples 1

Code

Example 1 - Using '%' wildcard

```
SELECT CASE WHEN 'aabbcc' LIKE '%bb%' THEN 'T' ELSE 'F' END
SELECT CASE WHEN 'aabbcc' LIKE '%a%c%' THEN 'T' ELSE 'F' END
```

Example 2 - Using '_' wildcard

```
SELECT CASE WHEN 'fair' LIKE '_air' THEN 'T' ELSE 'F' END
SELECT CASE WHEN 'lair' LIKE '_air' THEN 'T' ELSE 'F' END
```

Example 3 - Using '[]' searching

```
SELECT CASE WHEN 'theatre 7' LIKE '%theat[re][re] [0-9]' THEN 'T' ELSE 'F' END
SELECT CASE WHEN 'theater 2' LIKE '%theat[re][re] [0-9]' THEN 'T' ELSE 'F' END
```

Example 4 - Using '[^]' searching

```
SELECT CASE WHEN 'theatre X' LIKE '%theat[re][re] [^0-9]' THEN 'T' ELSE 'F' END
SELECT CASE WHEN 'theater 9' LIKE '%theat[re][re] [^0-9]' THEN 'T' ELSE 'F' END
```

Example 5 – Using escapes for special characters

```
SELECT CASE WHEN 'a[-%_' LIKE 'a[[]-[%][_]' THEN 'T' else 'F'
SELECT CASE WHEN 'a[-%_' LIKE 'a\[-\%\_' ESCAPE '\' then 'TRUE' else 'FALSE' end
```

Results

Example 1

```
'T'
'T'
```

Example 2

```
'T'
'T'
```

Example 3

```
'T'
'T'
```

Example 4

```
'T'
'F'
```

Example 5

```
'T'
'T'
```

Logical operators: LIKE examples 2

Tables

[txtLog]

ID	Note
1	He doesn't want to give a 50% cut
2	Cut that out we need the code
3	Here is the code [code:123_tf]
4	How long the code good for?
5	123-tf won't be good for too long
6	I think 124_tf will work too

Code

Example 1 - word anywhere in string

```
SELECT * FROM txtLog
WHERE Note LIKE '%code%'
```

Example 2 - word at end of string

```
SELECT * FROM txtLog
WHERE Note LIKE '%too'
```

Example 3 - skipping multiple characters

```
SELECT * FROM txtLog
WHERE Note LIKE '%12%tf%'
```

Example 4 - skipping single characters and brackets

```
SELECT * FROM txtLog
WHERE Note LIKE '%12_[_]tf%'
```

Example 5 - skipping single characters and caret brackets

```
SELECT * FROM txtLog
WHERE Note LIKE '%12_[^_]tf%'
```

Results

Example 1

ID	Note
2	Cut that out we need the code
3	Here is the code [code:123_tf]
4	How long the code good for?

Example 2

ID	Note
6	I think 124_tf will work too

Example 3

ID	Note
3	Here is the code [code:123_tf]
5	123-tf won't be good for too long
6	I think 124_tf will work too

Example 4

ID	Note
3	Here is the code [code:123_tf]
6	I think 124_tf will work too

Example 5

ID	Note
5	123-tf won't be good for too long

Multiple logical operators

Multiple logical operators can be strung together by relating them with `AND` or `OR` statements.

- `AND` - used to connect two or more conditions and only returns those rows meeting all conditions

- `OR` - used to connect two or more conditions and returns any rows that meet any of these conditions

Multiple logical operators examples

DS_Supply

City	State	Data Scientists
Springfield	Missouri	13
Springfield	Virginia	100
Vienna	Virginia	75
Springfield	New York	25
New York	New York	450

AND

```
SELECT *
FROM #DS_Supply
WHERE City = 'Vienna'
AND State = 'Virginia'
```

City	State	Data Scientists
Vienna	Virginia	75

OR

```
SELECT *
FROM #DS_Supply
WHERE City = 'Vienna'
OR State = 'Missouri'
```

City	State	Data Scientists
Springfield	Missouri	13
Vienna	Virginia	75

Combined

```
SELECT *
FROM #DS_Supply
WHERE State = 'New York'
AND [Data Scientists] > 50
OR State = 'Missouri'
```

City	State	Data Scientists
Springfield	Missouri	13
New York	New York	450

Functions

- Functions in SQL Server are stored programs that can be passed parameters and return a value or values. In SQL most functions are structured to work within the context of a query.
- SQL has functions that are designed to work with each and/or multiple data types and also operate either across columns in a single row or across rows in a single column.

Code

```
--Aggregating on an entire table
SELECT MIN(Cash_Balance) AS 'MIN', MAX(Cash_Balance) AS 'MAX'
     , SUM(Cash_Balance) AS 'SUM', AVG(Cash_Balance) AS 'AVG'
     , COUNT(Cash_Balance) AS 'COUNT'
FROM tbl_cash_balances
```

Function

Parameter

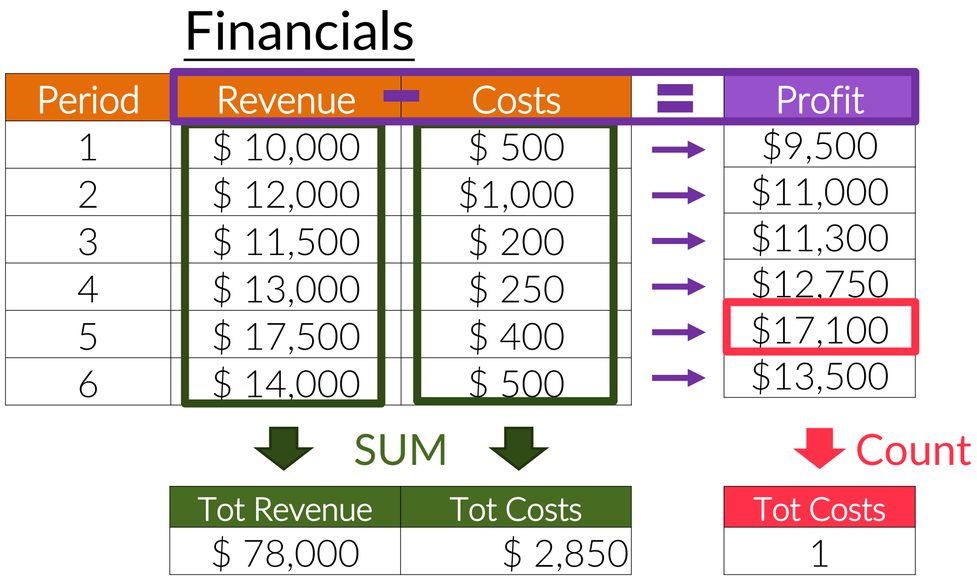
Returned value

Results

	MIN	MAX	SUM	AVG	COUNT
1	20	120	320	64	5

Functions: cross column vs. aggregate

- Cross column functions combine data in the same record from different columns
- Aggregate functions combine data in multiple rows from the same columns
- The two function types can be used together; however, the syntax needs to be precise




```
--Cross Column function: Periodic Profit
SELECT Revenue - Costs AS 'Profit'
FROM Financials

--Aggregate function
SELECT SUM(Revenue) AS 'Tot Revenue'
     , SUM(Costs) AS ' Tot Costs'
FROM Financials

--Cross Column: count if Profit > 15k
SELECT SUM(CASE WHEN Revenue - Costs > 15000
                THEN 1 ELSE 0 END) AS 'Periods GT 15K'
FROM Financials
```

Functions: math

- Cross column functions:
  - Numeric data types support all basic math functions such as addition (“+”), subtraction (“-”), multiplication (“*”), division (“/”), and exponents (“POWER()”)
  - Cross-column also includes mathematical expressions such as absolute value (“ABS()”), rounding (“ROUND()”) and random number generation (“RAND()”)
- Aggregate Functions:
  - SQL also supports aggregate math functions such as counts (“COUNT()”), sums (“SUM()”), maximum values (“MAX()”), and averages (“AVG()”)

Functions: text

- Cross-column functions:
  - Text data types support a variety of functions to manipulate text strings including:
    - Extract part of a string: LEFT(), RIGHT(), SUBSTRING()
    - Finding the position of text and length of string: PATINDEX(), CHARINDEX(), LEN()
    - Removing spaces from the ends of strings: LTRIM(), RTRIM(), TRIM()
    - Combine strings: CONCAT(), "+"
- Aggregate Functions:
  - Text data only supports several aggregate functions including counts ("COUNT()"), maximum alphanumeric values ("MAX()"), and minimum alphanumeric values ("MIN()")

Functions: dates

- Cross-column functions:
  - Date data types support a variety of functions to manipulate dates including:
    - Get current date and/or time: CURRENT_TIMESTAMP, GETDATE(), SYSDATETIME()
    - Return date and/or time parts: DAY(), MONTH(), YEAR(), DATENAME(), DATEPART()
    - Create date and/or time from parts: DATEFROMPARTS(), TIMEFROMPARTS()
    - Date math: DATEDIFF(), DATEADD()
  - Date data only supports several aggregate functions including counts ("COUNT()"), maximum alphanumeric values ("MAX()"), and minimum alphanumeric values ("MIN()")
    - This limitation can be avoided because dates can be converted to a numeric data type for additional calculations. After calculations (averages, variances, etc.) are performed on a numeric date they can be converted back knowing that Microsoft's base date (numerically zero) is 1/1/1900.

Functions: NULLs

- There are some useful functions that assist with dealing with incomplete data sets that contain NULL values:
  - ISNULL() - replaces NULL with the specified replacement value
  - COALESCE() - returns the first non-null expression among its arguments

Tables

[tblSales]

Date	SalesAmt1	SalesAmt2
12/1/15	150	75
12/25/15	NULL	50
1/15/16	35	NULL
1/31/16	NULL	65

Code

```
--Manually substitute for NULL values
SELECT Date, ISNULL(SalesAmt1, 0) AS SalesAmt
FROM tblSales

--Take first non-NULL values
SELECT Date, COALESCE(SalesAmt1, SalesAmt2) AS SalesAmt
FROM tblSales
```

Results

ISNULL

Date	SalesAmt
12/1/15	150
12/25/15	0
1/15/15	35
1/31/16	0

COALESCE

Date	SalesAmt
12/1/15	150
12/25/15	50
1/15/15	35
1/31/16	65

Exercise 7 - logic & functions

- Using the tables you imported earlier, practice the syntax for:
  - Logical operators
  - Cross-column and aggregate functions
    - Math
    - Text
    - Date

Questions?

Outline

1. Overview of SQL
2. Working with data using SQL statements
3. Manipulating tables using SQL
4. Logical and mathematical operations and functions
5. SQL Server best practices

Objectives

- Understand common data problems
- Review Quality Control (QC) tips
- Review Organizational tips
- Understand ways to improve performance and efficiency

common data problems

- Unreliable or unusable data
  - Incomplete data
  - Duplicate data
  - Inconsistent formats
- Inaccurate interpretations
  - What do the values in OFFENSE and METHOD really mean?
- Join issues
  - NULLs not accounted for
  - Wrong columns used
  - Table relationships not considered

CCN	REPORTEDDATE	REPORTEDTIME	SHIFT	OFFENSE	METHOD
13009976	1/23/2013	NULL	DAY	NULL	OTHERS
13009981	1/23/2013	9:46:00 AM	DAY	MOTOR VEHICLE THEFT	OTHERS
13009981	1/23/2013	9:46:00 AM	DAY	MOTOR VEHICLE THEFT	OTHERS
13009981	1/23/2013	9:46:00 AM	DAY	MOTOR VEHICLE THEFT	OTHERS
NULL	13009986	1/23/2013	11:25:00 AM	DAY	THEFT F/AUTO
NULL	13009987	1/23/2013	12:30:00 PM	DAY	THEFT/OTHER
13009990	01232013	10:54:00 AM	DAY	MOTOR VEHICLE THEFT	OTHERS
13009991	2013-1-23	23:57:00	DAY	BURGLARY	OTHERS
13009994	2013-1-23	12:13:00 PM	DAY	THEFT/OTHER	KNIFE
13010012	1/23/2013	18:34:00	DAY	THEFT F/AUTO	OTHERS
13010013	1/23/2013	12:31:00 PM	DAY	THEFT/OTHER	KNIFE
13010014	1/23/2013	12:25:00 PM	DAY	THEFT/OTHER	OTHERS

QC: preventing data problems

The following are quality control (QC) steps to detect and prevent common data problems:

- Reconcile data source totals
  - Compare record counts and sum totals of data sets used to source documents
  - Reconcile with other source documents
  - Compare detail and summary data sets
  - Account for all records

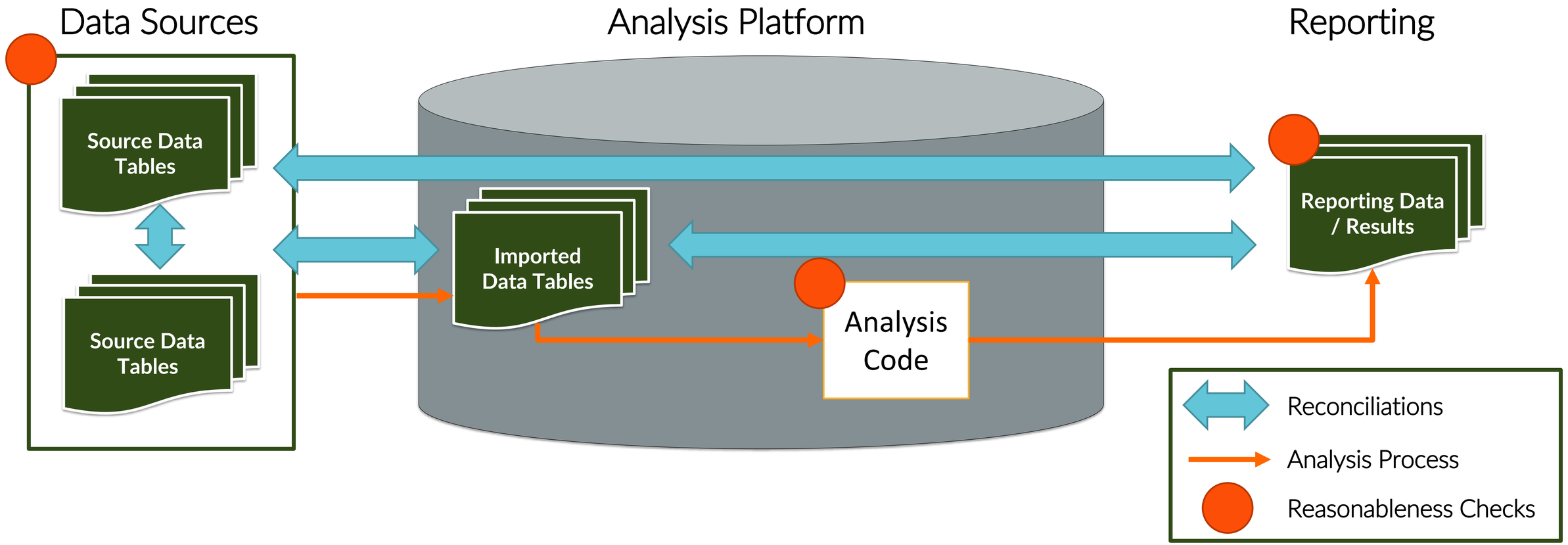
- Perform frequency distributions on the data
  - Check for incorrectly excluded or included data
  - Check for invalid entries
  - Distinguish between NULL and blank fields

QC: preventing data problems

- Check for reasonableness
  - Check that fields provide the correct data for your analysis
  - Check that amounts fields are appropriate (i.e. 5000 could be $5,000 or $50 if there is an implied decimal)
  - Ensure that fields are logically related to each other and carefully compare fields for `JOIN`s
  - In transactional data check for adjustments/corrections or reverse values that may impact totals or the meaning of the data
- Lookup table verification
  - Determine **valid values list** for lookup tables
  - **Compare valid values to data values** used
  - **Identify missing values**
- Query result tests
  - Evaluate whether or not JOIN results meet expectations
  - Verify the type of join used was correct
  - Determine if results contained too many or few records

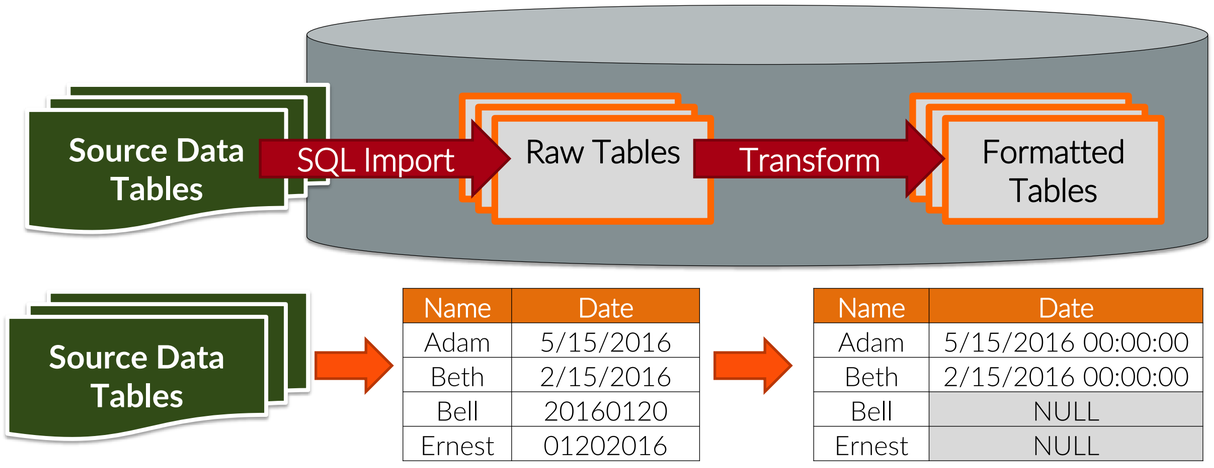
QC: building quality into analysis

The diagram below illustrates that quality control (QC) must be built into an analysis from beginning to end.

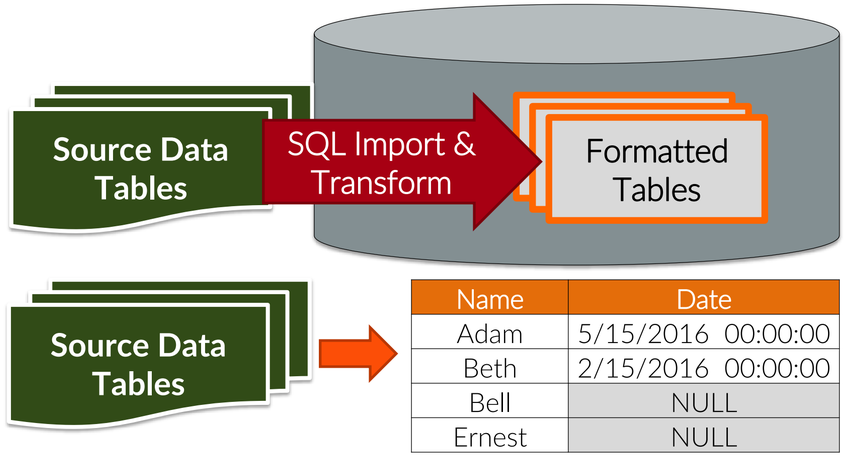


QC: import/export processes

- When importing data into SQL from files or other databases, there is a significant risk of losing data through conversion errors
- It is a good practice to import a raw table with fields in a text format first, and create a SQL script to convert the data types in a new table to assist with reconciling to source data inputs



VS.



Note: Importing and then transforming data types allows simple reconciliation facilitates correction of conversion errors

Note: without a complete import there is no straight forward way to know if NULLs in a data set are true NULLs or conversion errors

Organization: commenting

- Code should always be commented to the extent that a non-SQL user can understand the analysis process, and for SQL users to repeat the process

- Syntax:
  - In-line comments – all text after "--" will not be read by SQL Server when a query is run

```
-- Selecting all contract numbers for comparison
SELECT Contract_NO FROM [tbl_DS_Sample]
```

  - Block comments - all text between "/*" and "*/" will not be read by SQL Server when a query is run

```
SELECT Contract_NO
/* These contract numbers represent the population
   of non- performing loans */
FROM [tbl_DS_Sample]
```

Organization: formatting

Consistent code formatting makes it easier to read and edit. This saves hours of time spent trying to understand what is happening if the code is reviewed later. Here are a few suggestions for SQL formatting:

- Put keywords in UPPER CASE
  - SELECT, FROM
  - SUM, AVG
  - CASE, CAST, CONVERT
- Use tabs to align blocks of code
- Put commas at the beginning of a line

```
--Create Greater than 2000 flag
  SELECT A.Contract_NO
  , CASE WHEN Origination_Year > 2000 THEN 'Y'
         ELSE 'N' END AS GT_2000_Flag
  INTO #TEST
  FROM [tbl_DS_Sample] A
  LEFT JOIN [tbl_DS_Sample2] B
    ON A.Contract_NO = B. Contract_NO
  WHERE A.Contract_NO IS NOT NULL

--View records from #Test
  SELECT *
  FROM #TEST
```

Organization: naming conventions

- Use consistent naming conventions for SQL tables, views, and other objects

- Be consistent when renaming fields within a table/project
  - If there are many analyses relying on a field in a table and the name changes, it could severely interrupt other processes before the proper changes are made across a team or organization

- Do not use spaces in names
  - When spaces are used in a table or column names, it requires brackets ("[Field Name]") around the name to be properly read in SQL

Organization: saving SQL scripts

- Follow your office or team rules for creating and organizing scripts
- Each script should be modular and relate to one analysis or function
- Version control is also very important and adding version descriptions and dates to scripts can be very helpful when an analysis needs to be revisited
- For example a project named Surfside may have files saved like:
  - Surfside_Data_Prep_v01_20170225.sql = scripts related to preparing a table or data set for analysis
  - Surfside_Review_v01_20170315.sql = scripts such as frequency distributions that describe the data
  - Surfside_Data_Analysis_v01_20170322.sql = scripts related to the project deliverable

Performance: argument order

Fastest

Slowest

=

>,>=,<,<=

LIKE

<>

- Try to use a leading character with `LIKE` (ex. `LIKE 'm%'` instead of `LIKE '%m'`)
- Use LIKE instead of SUBSTRING with =

OR
AND

- Be careful with `OR`
- If multiple `AND`s, put least likely condition first
- If equally likely, put least complex condition first

Performance: query structure

- Restrict result sets by using WHERE or only selecting the columns needed
- Use WHERE with HAVING when appropriate
- ORDER BY is inefficient; sort results in a separate step

Performance: space efficiency

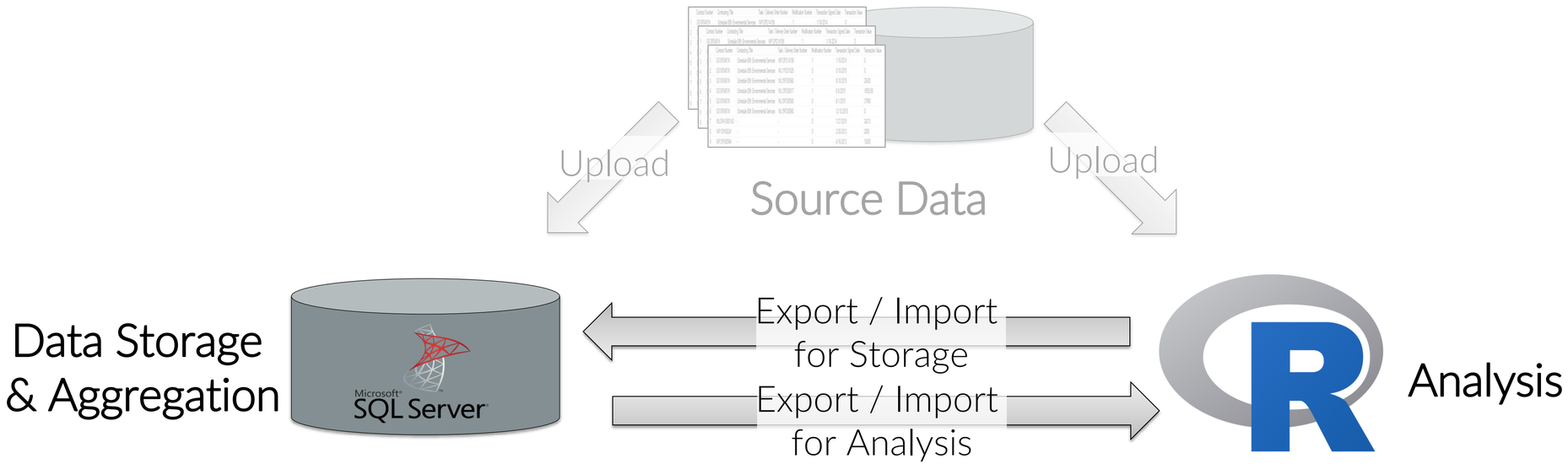
- Backup your database, but don't store excessive copies of the backup
- Choose between temp tables and views
- Use appropriate field types (avoid using NVARCHAR)

Questions?

Appendix:
Bridging data between SQL and R

Bridging R and SQL

- R can be used to extract and write data to a SQL database
- This allows analysis results sets and interim data sets to easily accessed from a central location



ODBC Connection

- R can connect to SQL using an Open Database Connectivity (ODBC) connection
  - ODBC is a protocol that allows for a connection between data sources such as Microsoft SQL Server.
- The RODBC package uses the odbcDriverConnect() function to create an ODBC connection that R can use to query a database


```
SQL_Con = odbcDriverConnect(paste('driver={SQL Server}'
                  ,';server=W530THINKPAD\\SQLEXPRESS'
                  ,';database=Data_Society_SQL_CLass'
                  ,';trusted_connection=true', sep = ''))
```

driver={SQL Server} indicates that we are connecting to a MS SQL server

The server you would like to connect to should be entered here after "server="

The database within the server that you would like to connect to should be entered here after "database="

true indicates that windows authentication will be used for server access permissions

Create a list of SQL tables

- Using the ODBC connection and the sqlQuery() function from the RODBC package, we can execute the SQL command SP_tables which will store a dataframe containing all SQL tables and views in a database.

Execute and store query

```
SQL_Table_List = sqlQuery(SQL_Con
                          ,'SP_tables')
```

SQL_Con is the ODBC connection previously defined

SP_tables is a function in SQL server that returns all database tables and Views (system and otherwise)

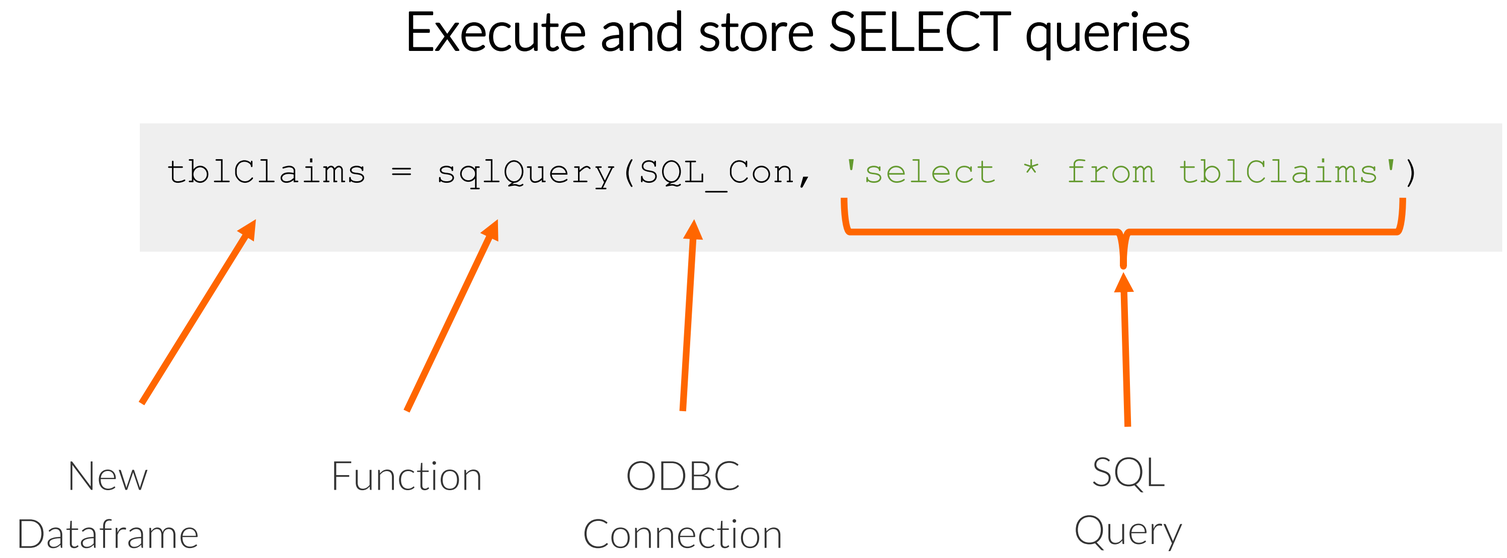
Subset the data set

```
SQL_Table_List = SQL_Table_List[SQL_Table_List$TABLE_OWNER != 'sys'
& SQL_Table_List$TABLE_OWNER != 'INFORMATION_SCHEMA',]
```

Remove System tables and Information schemas that would not typically be part of your analysis

Import SQL tables (individual)

- Using the ODBC connection and the sqlQuery() function from the RODBC package, we can execute a SELECT query which will store a dataframe containing all the results of that query.



Import SQL tables (bulk)

- Using a for loop the ODBC connection and the sqlQuery() function from the RODBC package, we can execute multiple SELECT queries to create dataframes for all (or a selected list) of tables and views in a database.

Import all tables

```
for(i in 1:nrow(SQL_Table_List)){
    schema = as.character(SQL_Table_List$TABLE_OWNER[i])
    tbl_name = as.character(SQL_Table_List$TABLE_NAME [i])
    tbl_import_loop = sqlQuery(SQL_Con
                               , paste('select * from '
                                       , schema
                                       , '.'
                                       , tbl_name
                               ,sep = '') )
    assign(paste( schema, '_'
                  , tbl_name ,sep = '')
                  ,tbl_import_loop )
    print(paste(schema, '_', tbl_name ,sep = ''))
}
```

- get the schema name for each record to ensure the right object is being selected from the database
- get the table name for querying each table in the database for import
- code to import SQL table into R
- code to assign each table imported an appropriate variable name based on both the schema and the table name from the SQL database
- Print tables imported in the console to track progress while loop runs

Export data to SQL

- Using the ODBC connection and either the sqlSave() function or the sqlQuery() function from the RODBC package, we can save a data into a new SQL table or append the data to an existing data table from R.

Export dataframe to SQL table

```
sqlSave(SQL_Con
        , tbl_crime_incidents_2013
        , tablename = "tbl_crime_incidents_2013_R_export"
        , rownames = FALSE
        , append = TRUE)
```

- SQL connection created earlier in the code
- R dataframe being exported
- The name of the table being created and/or appended to
- excludes rownames from being exported as a column in the new SQL table
- TRUE - if the table already exists then the records from this data frame will be appended to the existing table

Export values to SQL table

```
DivisionCode = "'C'"
DivisionName = "'Central'"
sqlQuery(SQL_Con
         , paste('INSERT INTO tlkpDemosDivisions_R_export
         SELECT ',DivisionCode,',',DivisionName, sep = ''))
```

- Define values
- SQL connection created earlier in the code
- Use INSERT statement with variable values to append data to a SQL table

Close ODBC connections

- It is a best practice to close an ODBC connection (or connections) after you are done accessing the databases to help maintain the performance and availability of your SQL database. We use the functions odbcClose() and odbcCloseAll() from the RODBC package to close individual or all connections respectively.

Close connections

```
odbcClose(SQL_Con)
odbcCloseAll()
```

- Close a specific ODBC connection
- Close all ODBC connection

Questions?