INTERMEDIATE SQL

GOING BEYOND THE SELECT

Created by Brian Duffey

WHO I AM

- Brian Duffey
- 3 years consultant at michaels, ross, and cole

- 9+ years SQL user
- What have I used SQL for?

ROADMAP

- Introduction
    1. Who I am
    2. Roadmap
    3. Basic SQL Review
- Working with Data
    1. Removing Data
    2. Bringing in Data
    3. Filtering Data
    4. Transforming Data
- Working with Objects
    1. Creating Functions
    2. Creating Programs
    3. Creating Datasets
    4. Improving Performance

ROADMAP

- Introduction
  1. Who I am
  2. Roadmap
  3. Basic SQL Review
- Working with Data
  1. Removing Data
  2. Bringing in Data
  3. Filtering Data
  4. Transforming Data
- Working with Objects
  1. Creating Functions
  2. Creating Programs
  3. Creating Datasets
  4. Improving Performance

BASIC SQL REVIEW

- Data
  - Insert **- C**reate
  - Select **- R**ead
  - Update **- U**pdate
  - Delete **- D**elete
- Object
  - Create
  - Drop

BASIC SQL REVIEW - DATA

Insert

- Used to add new rows to the database

```
insert into NAMES (FIRST_NAME, LAST_NAME) values ('John', 'Smith');
```

- into NAMES - object where data is being added
- (FIRST_NAME, LAST_NAME) - fields for adding data
- values ('John', 'Smith') - values being added

BASIC SQL REVIEW - DATA

Select

- Used to query the database for data
- Read-only

```
select * from NAMES where LAST_NAME = 'Smith' order by FIRST_NAME;
```

- * - all fields, can also be a field list
- from NAMES - object data is coming from
- where LAST_NAME = 'Smith' - filtering out data
- order by FIRST_NAME - sorting data by a field(s)

BASIC SQL REVIEW - DATA

Update

- Used to modify data in one or more columns

```
update NAMES set FIRST_NAME = 'Jane' where LAST_NAME = 'Smith';
```

- NAMES - object being updated
- set FIRST_NAME = 'Jane' - updating a field(s) to a new value
- where LAST_NAME = 'Smith' - setting which rows to update

BASIC SQL REVIEW - DATA

Delete

- Used to remove rows from the database

```
delete from NAMES where LAST_NAME = 'Smith';
```

- from NAMES - object being affected
- where LAST_NAME = 'Smith' - rows to delete

BASIC SQL REVIEW - OBJECTS

Create

- Used to add a new object to the database

```
create table MONTHS (..);
```

- table - type of object to create
- MONTHS - name of object
- (..) - options for object

BASIC SQL REVIEW - OBJECTS

Drop

- Used to remove an object from the database

```
drop table MONTHS;
```

- table - type of object to remove
- MONTHS - name of object

BASIC SQL REVIEW

Labeling

- To simplify queries, you can rename parts of it
- For instance, to rename a table, I can just put some identifier after it, like below
- Fields can also be renamed, by using the AS command

```
select * from NAMES A;
```

```
select LAST_NAME as SURNAME from NAMES;
```

ROADMAP

- Introduction
  1. Who I am
  2. Roadmap
  3. Basic SQL Review
- Working with Data
  1. Removing Data
  2. Bringing in Data
  3. Filtering Data
  4. Transforming Data
- Working with Objects
  1. Creating Functions
  2. Creating Programs
  3. Creating Datasets
  4. Improving Performance

WORKING WITH DATA

REMOVING REPEATED DATA

Sometimes a data set has data that is repeated. For instance, when trying to get a list of all customers who ordered in a time period.

```
select CUSTOMER from SALES where YEAR = 2013 order by CUSTOMER;
```

The above will return every line of sales in 2013, meaning a customer could be in there zero, one, or many times!

WORKING WITH DATA

REMOVING REPEATED DATA

Instead, we can use a DISTINCT command

```
select distinct CUSTOMER from SALES where YEAR = 2013 order by CUSTOMER;
```

This returns results where no row is duplicated

All returned values are considered

```
select distinct CUSTOMER, ORDER_DATE, PRICE*AMOUNT from SALES
order by CUSTOMER;
```

WORKING WITH DATA

REMOVING REPEATED DATA

For specific values, as well as aggregation, we can use a GROUP BY command

```
select CUSTOMER from SALES group by CUSTOMER order by CUSTOMER;
```

The above will return one line per customer, just like the distinct statement

```
select CUSTOMER, max(ORDER_DATE), sum(PRICE*AMOUNT) from SALES
group by CUSTOMER order by CUSTOMER;
```

The above will still return one line per customer. Additionally it will show the last order date, the last ORDER_DATE, as well as the total sales of all orders.

Aggregation (MIN, MAX, SUM, AVG, COUNT) can be done with or without GROUP BY

WORKING WITH DATA

BRINGING IN ADDITIONAL DATA

Sometimes a data set is missing information. For instance, needing to get a customer's state

```
select * from SALES where YEAR = 2013;
```

The above will return every field in SALES, however there is no state field in this table.

WORKING WITH DATA

BRINGING IN ADDITIONAL DATA

In order to grab data from a different table, we can do a JOIN

```
select * from SALES A join CUSTOMERS B on A.CUSTOMER = B.CUSTOMER
where YEAR = 2013;
```

The above will return every field in SALES as well as CUSTOMERS

WORKING WITH DATA

BRINGING IN ADDITIONAL DATA

There are several types of joins:

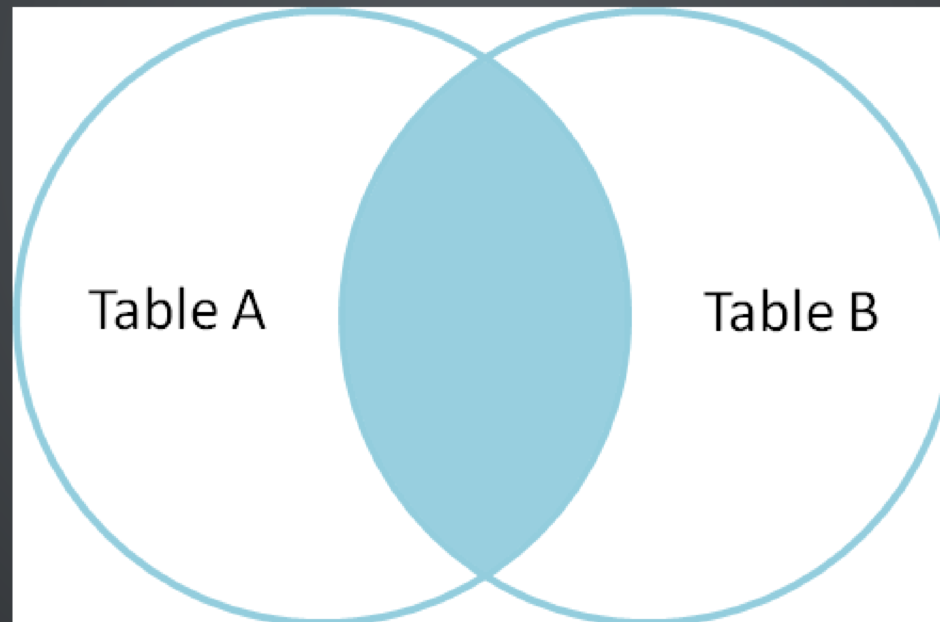
- INNER JOIN
- LEFT/RIGHT OUTER JOIN
- FULL OUTER JOIN
- CROSS JOIN
- Exception joining
- UNION (ALL)

WORKING WITH DATA

BRINGING IN ADDITIONAL DATA

There are several types of joins:

- INNER JOIN

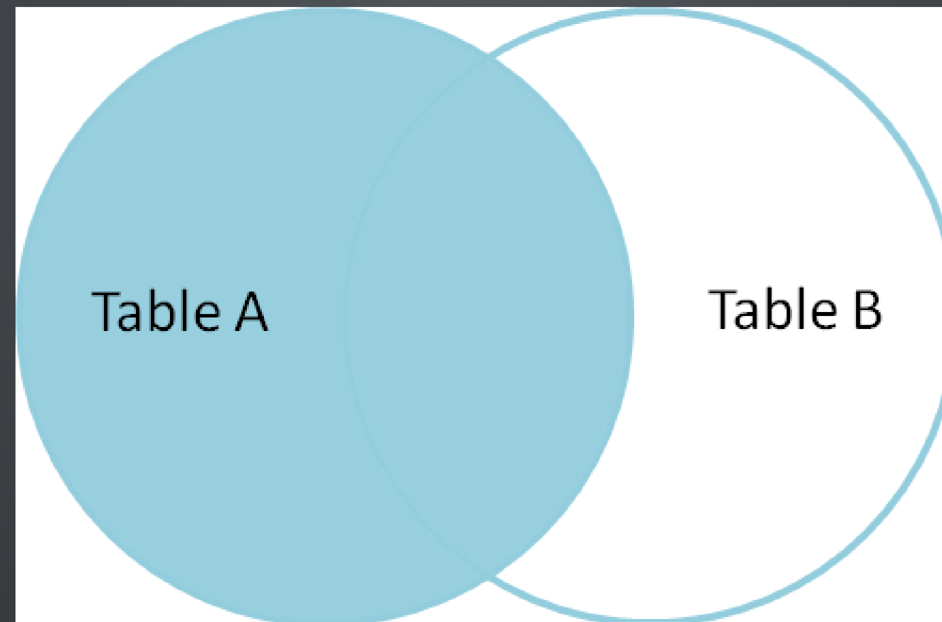


WORKING WITH DATA

BRINGING IN ADDITIONAL DATA

There are several types of joins:

- LEFT/RIGHT OUTER JOIN

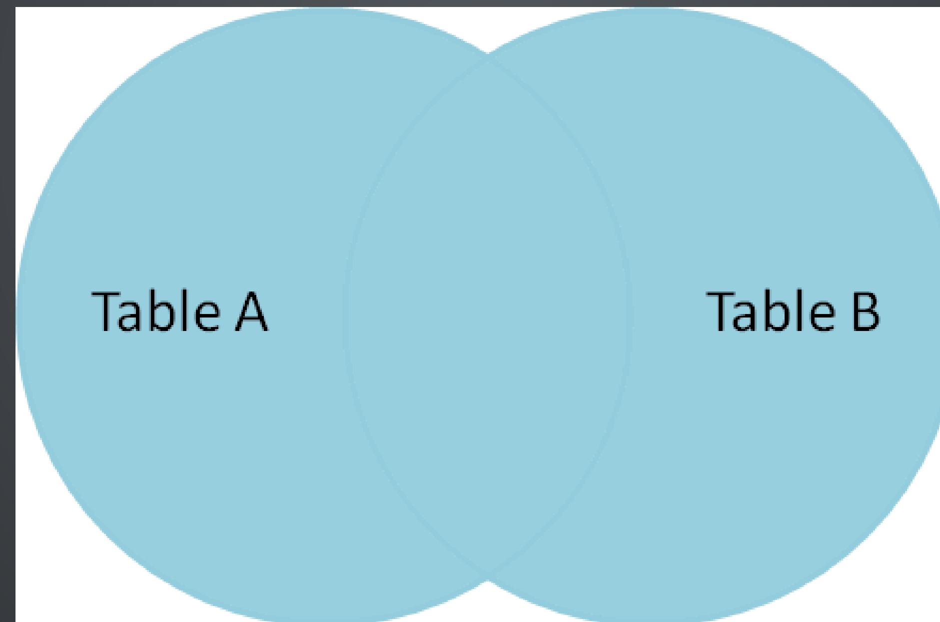


WORKING WITH DATA

BRINGING IN ADDITIONAL DATA

There are several types of joins:

- FULL OUTER JOIN

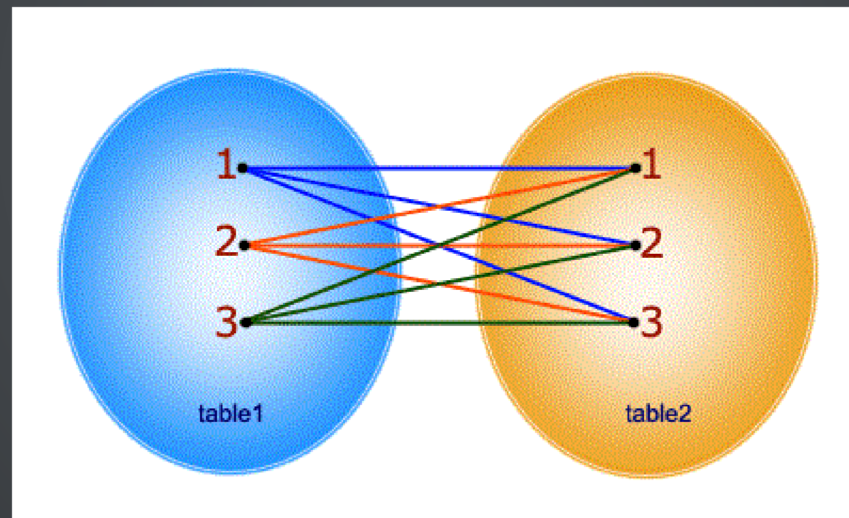


WORKING WITH DATA

BRINGING IN ADDITIONAL DATA

There are several types of joins:

- CROSS JOIN

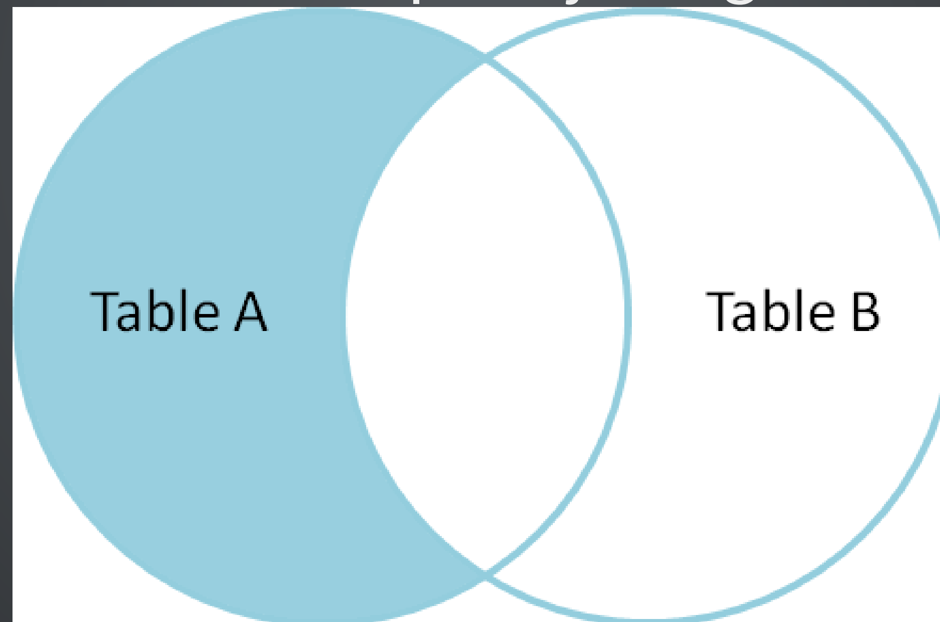


WORKING WITH DATA

BRINGING IN ADDITIONAL DATA

There are several types of joins:

- Exception joining

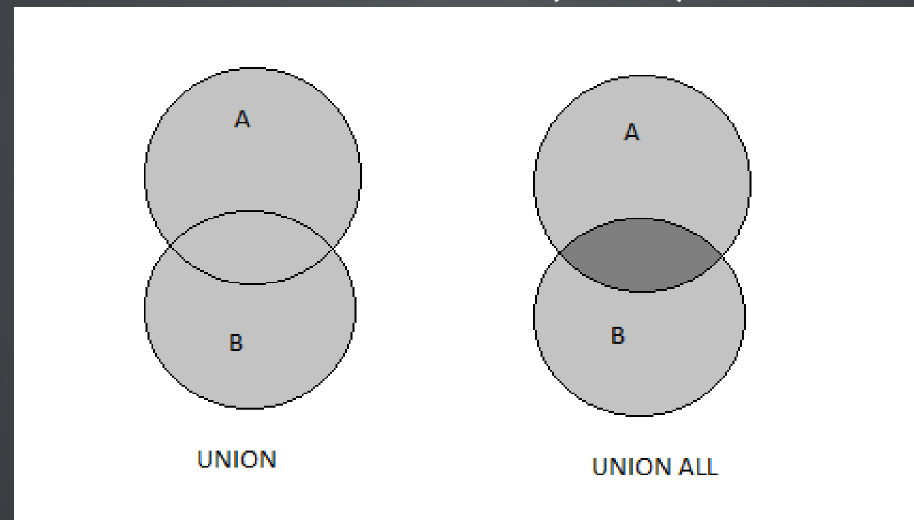


WORKING WITH DATA

BRINGING IN ADDITIONAL DATA

There are several types of joins:

- UNION (ALL)



WORKING WITH DATA

BRINGING IN ADDITIONAL DATA

What if we need data outside of the current data set? For instance, we need a breakdown of number of orders for a customer last year, plus their last order date.

```
select CUSTOMER, sum(1), max(ORDER_DATE)
from SALES
where YEAR = 2013
group by CUSTOMER;
```

The above will not work because the records are limited to 2013, meaning any orders placed in 2014 are excluded.

WORKING WITH DATA

BRINGING IN ADDITIONAL DATA

To fix, we can use a sub-query

```
select CUSTOMER, sum(1),
(select max(B.ORDER_DATE) from SALES B where B.CUSTOMER = A.CUSTOMER)
from SALES A
where YEAR = 2013
group by CUSTOMER;
```

The above will still load all orders from 2013, however the sub-query will go out and find the last order date for a customer.

WORKING WITH DATA

BRINGING IN ADDITIONAL DATA

Sub-queries are great for combining unrelated data

They can be used anywhere within the query, such as in the WHERE clause

WORKING WITH DATA

FILTERING UNWANTED DATA

The WHERE clause is very useful for selecting on the desired data

```
select * from SALES where AMOUNT < 20;
```

Filter on any field in the data set, or in a different related set (sub-query), using boolean operators:

- =
- !=, <>
- >, >=
- <, <=
- IS NULL, IS NOT NULL

WORKING WITH DATA

FILTERING UNWANTED DATA

There are also many other useful filters:

- IN

```
select * from CUSTOMERS where STATE in ('FL', 'IL');
```

- BETWEEN

```
select * from SALES where AMOUNT between 10 and 20;
```

- LIKE

```
select * from CUSTOMERS where CUSTOMER like 'A%';
```

WORKING WITH DATA

FILTERING UNWANTED DATA

- There may be times we need to filter aggregated data

```
select CUSTOMER, sum(1) from SALES
where sum(1) > 500 group by CUSTOMER;
```

- The above will fail with an error as the WHERE clause can only filter raw data, not the aggregate
- Instead, use a HAVING clause, which is performed after the GROUP BY:

```
select CUSTOMER, sum(1) from SALES
group by CUSTOMER having sum(1) > 500;
```

TRANSFORMING DATA

MAKING CONDITIONAL CHANGES

The CASE statement is very useful for changing values

```
select AMOUNT, case when AMOUNT > 20 then 'Good' else 'Bad' end from SALES
where YEAR = 2013;
```

You can have as many cases as you need, and everything is put into one column for easy reference

The above breaks the AMOUNT field down into Good or Bad values

TRANSFORMING DATA

USING DATABASE FUNCTIONS

There are many built-in functions each database supports

Some of the most common ones include things like:

- Substring - getting a section of a string
- Concatenation - joining two strings together
- Casting - changing a value from one data type to another
- Date-related functions - Getting the year of a date, number of days between two dates, etc.

ROADMAP

- Introduction
    1. Who I am
    2. Roadmap
    3. Basic SQL Review
- Working with Data
    1. Removing Data
    2. Bringing in Data
    3. Filtering Data
    4. Transforming Data
- Working with Objects
    1. Creating Functions
    2. Creating Programs
    3. Creating Datasets
    4. Improving Performance

WORKING WITH OBJECTS

USER DEFINED FUNCTIONS

- While each database has their own set of functions, there may be times when you have other needs
- The database will let you create a function to handle your own values
- You can pass in any number of values, do something with them, and then return one value

WORKING WITH OBJECTS

USER DEFINED FUNCTIONS

```
create function ADD1 (NUMBER int)
  returns int
begin
  return NUMBER + 1;
end
```

- The above creates a UDF called ADD1, which simply adds one to some number
- This function takes in one parameter, of type int(eger)
- It returns an int value
- All code happens in the begin..end section

WORKING WITH OBJECTS

USER DEFINED FUNCTIONS

```
create function GETSTATE (CUST char(25))
  returns char(2)
begin
  declare ST char(2);
  select STATE into ST from CUSTOMERS where CUSTOMER = CUST;
  return ST;
end
```

- The above creates a UDF called GETSTATE
- This function takes in one parameter, the customer name
- It returns the state
- This function uses a query to grab the state for a customer

WORKING WITH OBJECTS

USER DEFINED FUNCTIONS

Use these functions like a field value

The returned value is displayed

```
select AMOUNT, ADD1(AMOUNT) from SALES;
```

```
select CUSTOMER, STATE, GETSTATE(CUSTOMER) from CUSTOMERS;
```

WORKING WITH OBJECTS

STORED PROCEDURES

- There will be times when you need a program
- Stored procedures differ from functions in a couple ways:
  - They do not return any value
  - They cannot be called from a query
  - Their parameters can be modified

WORKING WITH OBJECTS

STORED PROCEDURES

```
create procedure CHANGEAMOUNT (in VAL int)
begin
  update SALES set AMOUNT = AMOUNT + VAL;
end
```

- The above program simply adds some amount to the AMOUNT field
- Parameters can be IN, OUT, or INOUT

WORKING WITH OBJECTS

STORED PROCEDURES

- Procedures are called
- Usually this is done from some program, i.e. Java
- Can also be done from the database/command line:

```
call CHANGEAMOUNT(1);
```

WORKING WITH OBJECTS

CUSTOM DATA SETS

- When selecting data, the FROM clause is generally a table
- However, you can use a sub-query to SELECT from:

```
select CUSTOMER, CUSTOMER_NUMBER, STATE from CUSTOMERS
where CUSTOMER like 'A%';
```

- The above returns a data set of all customers that have a name starting with 'A'
- We can now further select within this data set:

```
select * from (
select CUSTOMER, CUSTOMER_NUMBER, STATE from CUSTOMERS
where CUSTOMER like 'A%'
) A where STATE = 'OK';
```

WORKING WITH OBJECTS

CUSTOM DATA SETS

- Alternatively, we can make this data set more "permanent"
- Views are dynamic data sets based upon some query

```
create view A_CUSTOMERS as
select CUSTOMER, CUSTOMER_NUMBER, STATE from CUSTOMERS
where CUSTOMER like 'A%';
```

- The above creates an object that stores all rows in CUSTOMERS that have a name starting with 'A'
- This can then be used like a table:

```
select * from A_CUSTOMERS where STATE = 'OK';
```

WORKING WITH OBJECTS

IMPROVING PERFORMANCE

- The less rows/columns selected, the quicker the query will run
- Use WHERE and HAVING clauses to limit irrelevant data
- Use INNER JOINs to only select matching data
- Don't use * when you only need a few fields

WORKING WITH OBJECTS

IMPROVING PERFORMANCE

- Second, after first optimizing your query, try indexes
- Indexes are like a table of contents for your database
- Types of indexes:
  - UNIQUE
  - Covering
  - Clustered
- Sample index:

```
create index MY_INDEX on NAMES(FIRST_NAME, LAST_NAME);
```

- Covering:

```
select FIRST_NAME, LAST_NAME from NAMES;
```

- Clustered:

```
select FIRST_NAME, LAST_NAME, AGE from NAMES;
```

WORKING WITH OBJECTS

IMPROVING PERFORMANCE

So, why not create a bunch of indexes?

- Most tables won't have every column selected on
- All non-read statements become much slower, i.e. insert/update/delete
- Indexes take up disk space and memory

Instead, use database tools like EXPLAIN to help you optimize your query and build the proper indexes

LINKS

- My information: www.mrc-productivity.com/Services/Brian_Duffey.html
- Slides: www.mrc-productivity.com/Duffey/slides/IntermediateSQL.html
- Other resources: www.mrc-productivity.com/Duffey/COMMON14.html
  - MySQL
  - DBVisualizer

CREDITS

- http://www.dbvis.com/
- http://blog.codinghorror.com/a-visual-explanation-of-sql-joins/
- http://www.sitepoint.com/using-explain-to-write-better-mysql-queries/