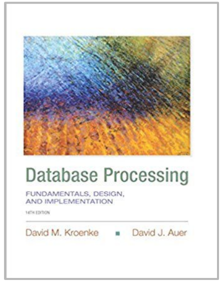
Fairfield University DOLAN School of Business



Databases for Analytics

Kroenke / Auer
Chapter 2

Learning Objectives

This is a **very LONG chapter** that we will likely have to take in two parts.

- **Skills:** You should know how to ...
  - Write basic SQL SELECT / FROM / WHERE queries
  - Calculate aggregate quantities like AVG, SUM, etc.
  - Group records using field/column selectors
  - Query multiple tables using joins and subqueries
- **Theory:** You should be able to explain ...
  - The sequence and purpose of each clause of SQL select queries (SELECT, FROM, WHERE, GROUP BY, etc.)
  - The relationship between SQL and set algebra (Cross Products, Unions, Intersections, and Complements)

Cape Codd Database

Load the Sample Database for Chapter 2

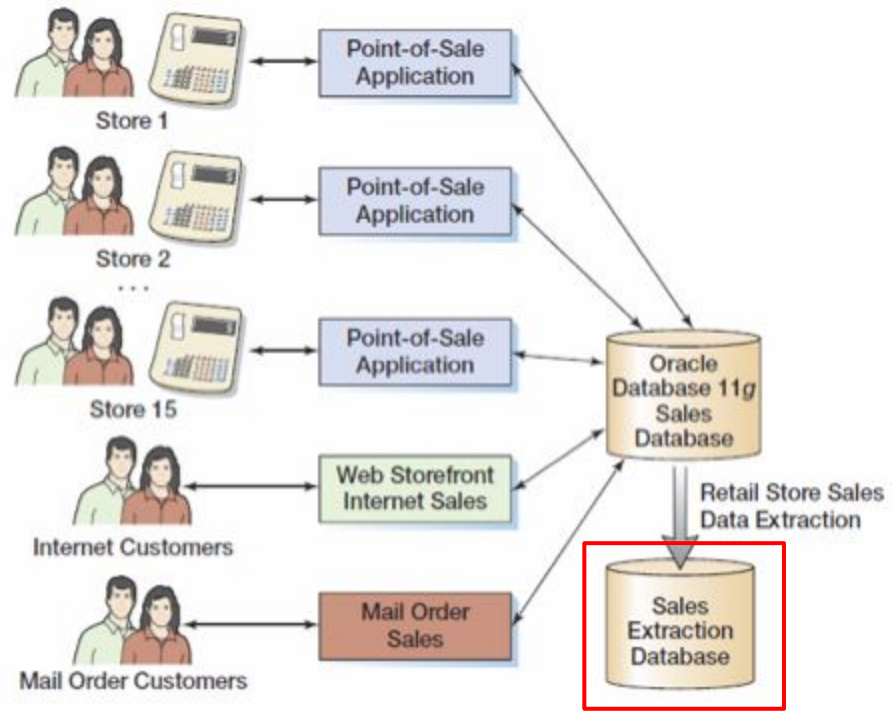
Connect to the Database

1. On today's agenda, click on the GitHub Classroom assignment for the Cape Codd database.
2. In JupyterLab, clone your copy to your workspace.
3. Open the cloned repo folder.
4. You should see a couple SQL files and a Jupyter notebook.
5. Open the notebook and follow instructions.
6. Connect URLs take the form

   <method>://<user>:<pass>@<host>/<database>

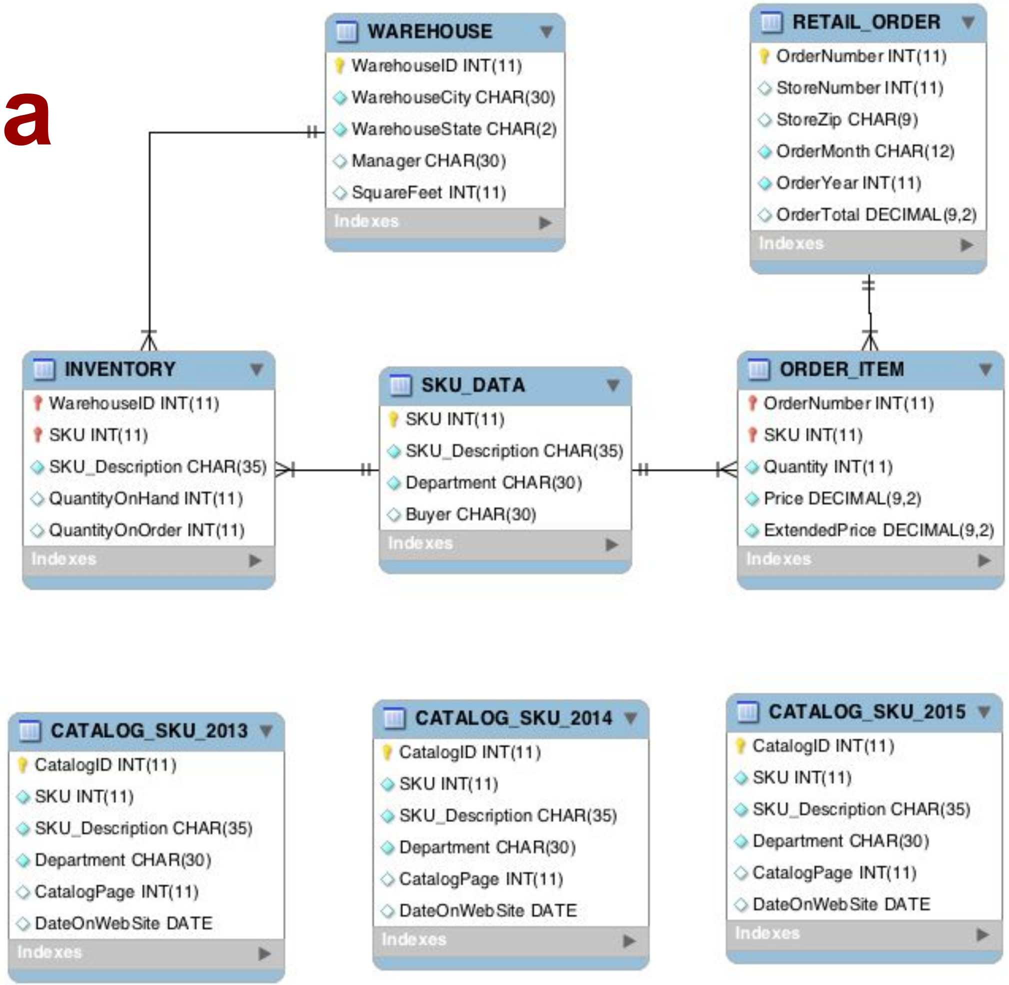
Data Source

- The database is an extract of data from a live transaction system for Cape Codd Outdoor.
- 5 connected tables plus 3 disconnected archival tables.



Database Schema

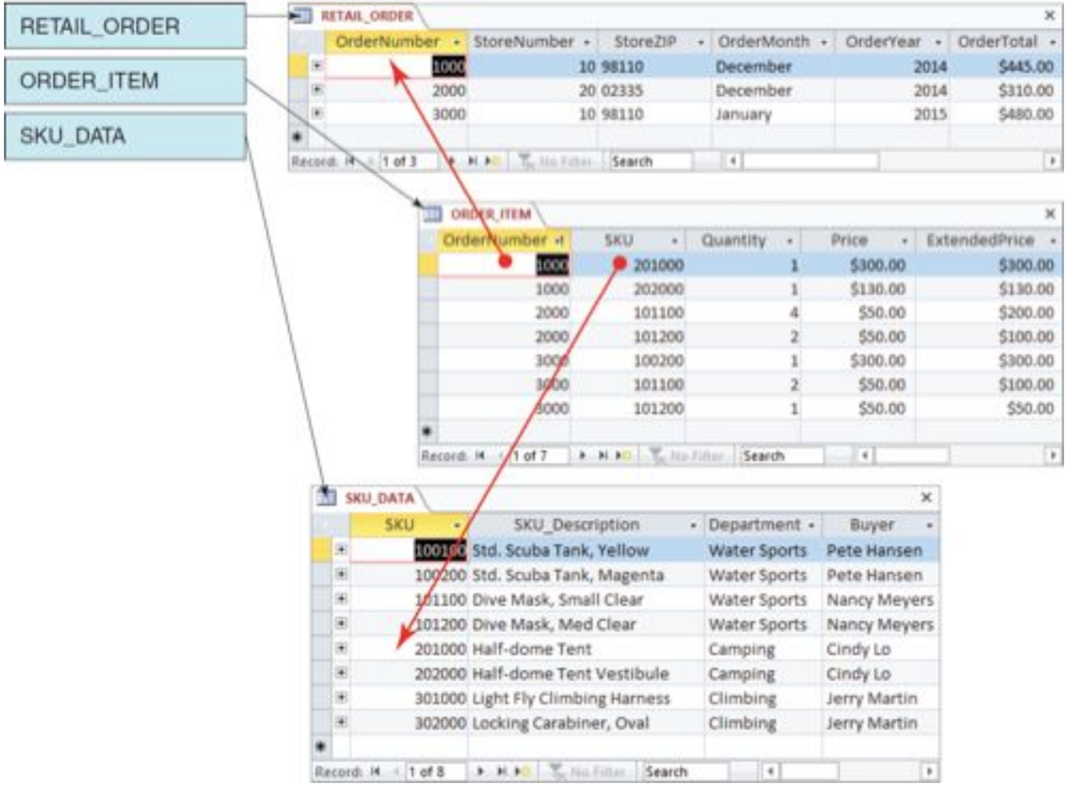
This will help us with the SELECT queries coming up.



The diagram was generated using MySQL Workbench's Reverse Engineering feature. No drawing required!

With Data and Relationships

Each link is an Foreign Key (FK) match to a Primary Key (PK) column.



More Details ...

This shows the data types of the fields (columns) of the five linked tables.

Table	Column	Data Type
RETAIL_ORDER	OrderNumber	Integer
	StoreNumber	Integer
	StoreZIP	Character (9)
	OrderMonth	Character (12)
	OrderYear	Integer
	OrderTotal	Currency
ORDER_ITEM	OrderNumber	Integer
	SKU	Integer
	Quantity	Integer
	Price	Currency
	ExtendedPrice	Currency
SKU_DATA	SKU	Integer
	SKU_Description	Character (35)
	Department	Character (30)
	Buyer	Character (30)
CATALOG_SKU_20##	CatalogID	Integer
	SKU	Integer
	SKU_Description	Character (35)
	Department	Character (30)
	CatalogPage	Integer
	DateOnWebSite	Date

Structured Query Language

A bit of background from the textbook

Note about SQL Standards

- **Structured Query Language** (SQL) was developed by the IBM Corporation in the late 1970’s.
- SQL was endorsed as a U.S. national standard by the American National Standards Institute (ANSI) in 1992 [**SQL-92**].
- Newer versions exist, such as SQL:2008 and SQL:2011,and they incorporate new features including XML and some object-oriented concepts. Some of these features are discussed in this book.

SQL is a Specialized Language

- SQL is **not** a full featured programming language like C, C#, Java, Python, PHP, Javascript, …
- SQL is a **data sublanguage** for creating and processing database data and metadata.
- SQL is ubiquitous in enterprise-class DBMS products and works with just about any programming language on earth.

SQL is actually 5 languages

- SQL code can be divided into five categories:
  - **Data definition language (DDL) statements**
  - **Data manipulation language (DML) statements**
  - SQL/Persistent Stored Modules (SQL/PSM) statements
  - Transaction control language (TCL) statements
  - Data control language (DCL) statements
- **We are going to learn select DML and DDL statements.** The rest are for SQL DB Administrators/Engineers.

SQL DDL vs DML

SQL Data Definition Language (DDL) is used to **create**, **modify**, or **discard** ***metadata***:

- Tables, columns, etc.
- **Relationships, keys, etc.**

We'll come back to DDL in chapter 7.

SQL Data Manipulation Language (DML) is used to **retrieve**, **add**, **update**, or **delete data**:

- **SELECT statements retrieve data from tables**
- INSERT, UPDATE, and DELETE statements manage the data in the tables

Select / From / Where ...

Now that we are ready to go …
let's get back to the Cape Codd database

First … The Whole Enchilada

```
SELECT column-list
FROM tables
WHERE row-conditions
GROUP BY grouping-column
HAVING aggregate-conditions
ORDER BY column-sort-list;
```

- Clauses must be in the order given
- WHERE, GROUP BY, HAVING, and ORDER BY are optional
- Semicolon is required

The SELECT Clause

SELECT *column-list*

The SELECT clause is used to indicate which data **columns** we want.

Column names are comma separated.

To select all columns use the wildcard '*'.

You can even use SELECT like a calculator. Try this yourself ...

```
In [3]: %%sql
        SELECT 1+1;

        1 rows affected.

Out[3]: 1+1
          2
```

The FROM Clause

```
SELECT column-list
FROM tables;
```

The FROM clause indicates which tables to draw the columns from.

Try this ...

In [4]:
```
%%sql
SELECT *
FROM Inventory;
```

32 rows affected.

Out[4]:

WarehouseID	SKU	SKU_Description	QuantityOnHand	QuantityOnOrder
100	100100	Std. Scuba Tank, Yellow	250	0

Column Order

Columns are returned in the order they are listed in the query:

```
SELECT SKU, SKU_Description, Department, Buyer
FROM SKU_DATA;
```

is not the same as

```
SELECT SKU, Department, Buyer, SKU_Description
FROM SKU_DATA;
```

Try this yourself to see the difference.

SQL Comments

Comments are used to annotate the SQL statements for us humans. They are part of the code!

Anything inside `/*` and `*/` is a comment:

```
In [ ]: %%sql
        /* *** SQL Query-CH02-01 *** */
        SELECT SKU, SKU_Description, Department, Buyer
        FROM SKU_DATA;
```

Alternatively, use -- (two hyphens) just like # in Python.

Use comments liberally in your code to make it readable/scannable by other humans.

It's unprofessional to product unreadable code.

The WHERE Clause

SELECT *column-list*

FROM *tables*

WHERE *row-conditions;*

The WHERE clause used a boolean expression (condition) to indicate which **rows** we want.

Try this ...

In [5]:
```
%%sql
SELECT *
FROM Inventory
WHERE QuantityOnHand < 10;
```

9 rows affected.

Out[5]:

WarehouseID	SKU	SKU_Description	QuantityOnHand	QuantityOnOrder

A Few SELECT Enhancements

Don't you love puns!

The `DISTINCT` Keyword

Sometimes a query can return duplicate data, with two rows exactly the same. To prevent duplicates use the **`DISTINCT`** keyword in the **`SELECT`** clause.

```
SELECT DISTINCT Buyer, Department
FROM SKU_DATA;
```

Try this **with** and **without** the **`DISTINCT`** keyword. Notice the difference?

The `ORDER BY` Clause

```
SELECT column-list
FROM tables
WHERE column-conditions
ORDER BY column-sort-list;
```

The ORDER BY clause sorts the rows according to listed columns.

ORDER BY Examples

```
SELECT SKU, SKU_Description, Department, Buyer
FROM SKU_DATA
ORDER BY SKU DESC;
```

DESC after a column name sorts in descending order.

```
SELECT SKU, SKU_Description, Department, Buyer
FROM SKU_DATA
ORDER BY SKU_Description, SKU;
```

When multiple columns are listed, a ***lexicographic*** sort is used, starting with the first column in the list, then the second column, etc.

CREATE VIEW Statements

Views are not covered in the textbook but are included here because of their utility for business analysts.

```
CREATE VIEW SKU_TENTS
AS (SELECT SKU, SKU_Description FROM INVENTORY WHERE
SKU_Description like "%Tent%");
```

A **view** is a named query that acts like a virtual table that is always up-to-date. It can be used anywhere a table can be used:

```
SELECT *
FROM SKU_TENTS;
```

The view above might be pre-defined in the database, allowing us to query it from a Jupyter cell without so much SQL code leaking into your Python code. It's an obvious win-win.

SQL WHERE Clause Options

Because choosing what you want can be complex sometimes

Comparisons

At a base level, just about every condition in a `WHERE` clause will involve one or more **comparisons**.

```
SELECT *
FROM CATALOG_SKU_2014
WHERE
   DateOnWebsite = '01-Jan-2014';
```

SQL Comparison Operators	
Operator	Meaning
=	Is equal to
<>	Is NOT Equal to
<	Is less than
>	Is greater than
<=	Is less than OR equal to
>=	Is greater than OR equal to
IN	Is equal to one of a set of values
NOT IN	Is NOT Equal to one of a set of values
BETWEEN	Is within a range of numbers (includes the end points)
NOT BETWEEN	Is NOT within a range of numbers (includes the end points)
LIKE	Matches a set of characters
NOT LIKE	Does NOT match a set of characters
IS NULL	Is equal to NULL
IS NOT NULL	Is NOT equal to NULL

Note on SQL Expressions

The syntax of a comparison is actually like this

`<left-expression> <operator> <right-expression>`

where

- `<operator>` is a comparison operator `(=,>,<,etc.)` and
- `<left-expression>` and `<right-expression>` are *anything* that *evaluates* to a value

SQL expressions can be column names, function calls, arithmetic calculations, numbers, dates, strings, etc.

The **IN** Operator

```sql
SELECT *
FROM SKU_DATA
WHERE Buyer IN ('Nancy Meyers', 'Cindy Lo',
    'Jerry Martin');
```

The **IN** operator tests whether the column matches one of a set of values. The set is always enclosed in parentheses **( )**. The set is actually a kind of SQL expression. Why? Because IN is a comparison operator. Duh!

The `BETWEEN` Operator

```
SELECT *
FROM ORDER_ITEM
WHERE ExtendedPrice BETWEEN 100 AND 200;
```

The **`BETWEEN`** operator checks if the value is within a range. The endpoints of the range are included.

Like the set on the previous slide, the range `'100 AND 200'` is also a SQL expression.

The `LIKE` Operator

```
SELECT *
FROM SKU_DATA
WHERE Buyer LIKE 'N%';
```

> Selects all buyers whose name starts with the letter N.

The **`LIKE`** operator does a string pattern match. There are two wildcard characters we can use for the pattern:

- **%** matches any number of characters
- **_** matches exactly one character

The `IS NULL` Operator

```
SELECT *
FROM CATALOG_SKU_2015
WHERE CatalogPage IS NULL;
```

Oops! This one doesn't return anything in this case. The sample database does not match the one in the textbook. D'oh!

`NULL` is a special value meaning 'nothing at all', commonly used as a missing value. The **`IS NULL`** operator checks for a `NULL` value.

We can also check for non-`NULL` values with **`IS NOT NULL`**.

Logical Expressions: OR, AND, NOT, and ( )

We can combine comparisons to check for multiple conditions:

```
SELECT *
FROM SKU_DATA
WHERE
    (SKU>200000 OR SKU_Description LIKE '%TENT%')
    AND NOT Department = 'Camping';
```

Precedence Rules:

1. Eval ( )
2. Eval NOT
3. Eval AND
4. Eval OR

Logical Operators

SQL Logical Operators	
Operator	Meaning
AND	Both arguments are TRUE
OR	One or the other or both of the arguments are TRUE
NOT	Negates the associated operator

Again, in slow motion

```
SELECT * FROM SKU_DATA
WHERE SKU>200000;

SELECT * FROM SKU_DATA
WHERE SKU>200000 OR SKU_Description LIKE '%TENT%';

SELECT * FROM SKU_DATA
WHERE (SKU>200000 OR SKU_Description LIKE '%TENT%')
    AND NOT Department = 'Camping';
```

Try all three to see how each condition affects the results

Calculations in SQL

SQL Functions and Arithmetic

Built-in Aggregate Functions

```
SELECT COUNT(SKU)
FROM SKU_DATA;

SELECT MAX(SKU)
FROM SKU_DATA;
```

SQL Built-in Aggregate Functions	
Function	**Meaning**
COUNT(*)	Count the number of rows in the table
COUNT ({Name})	Count the number of rows in the table where column {Name} IS NOT NULL
SUM	Calculate the sum of all values (numeric columns only)
AVG	Calculate the average of all values (numeric columns only)
MIN	Calculate the minimum value of all values
MAX	Calculate the maximum value of all values

The AS Keyword

```
SELECT COUNT(SKU)
FROM SKU_DATA;
```

Returns a column named 'COUNT(SKU)', which is pretty ugly and confusing. Use the **AS** keyword to give the calculation an ***alias name*** (as in "also known as").

```
SELECT COUNT(SKU) AS CountOfSKUs
FROM SKU_DATA;
```

AS always **follows** the thing being aliased.

SQL Expressions (again)

Arithmetic and other SQL expressions work a lot like function calls:

```
SELECT (Quantity * Price) AS EP
FROM ORDER_ITEM;

SELECT *
FROM ORDER_ITEM
WHERE (Quantity * Price) > 150;
```

RTFM for more …

The MySQL manual has an extensive section on various kinds of functions and operators that you can use as SQL expressions.

You should at least spend some time reading up on numeric functions like `LOG()`, `POW()`, and `ROUND()` and string operators like `CONCAT()` and `TRIM()`.

Grouping and Group-wise Aggregates

How to generate subsets, subtotals, etc.

Grouping by Column Values

This group of rows is for the **Water Sports** department

This SKU did not appear in the catalog

This group of rows is for the **Camping** department

This group of rows is for the **Climbing** department

	CatalogID	SKU	SKU_Description	Department	CatalogPage	DateOnWebSite
1	20140001	100100	Std. Scuba Tank, Yellow	Water Sports	23	2014-01-01
2	20140002	100300	Std. Scuba Tank, Light Blue	Water Sports	23	2014-01-01
3	20140003	100400	Std. Scuba Tank, Dark Blue	Water Sports	NULL	2014-08-01
4	20140004	101100	Dive Mask, Small Clear	Water Sports	26	2014-01-01
5	20140005	101200	Dive Mask, Med Clear	Water Sports	26	2014-01-01
6	20140006	201000	Half-dome Tent	Camping	46	2014-01-01
7	20140007	202000	Half-dome Tent Vestibule	Camping	46	2014-01-01
8	20140008	301000	Light Fly Climbing Harness	Climbing	77	2014-01-01
9	20140009	302000	Locking Carabiner, Oval	Climbing	79	2014-01-01

The grouping column is Department

The `GROUP BY` Clause

```
SELECT column-list
FROM tables
WHERE column-conditions
GROUP BY grouping-column-list;
```

We can group on multiple columns if needed. (Example on next slide.)

A few general rules:

- The `column-list` in the SELECT clause can ***only*** include grouping columns and aggregate functions
- The `WHERE` clause is executed **before** the grouping

GROUP BY Examples

```
/* Grouping by a single column */
SELECT SKU, Count(SKU), AVG(ExtendedPrice)
FROM ORDER_ITEM
GROUP BY SKU;

/* Grouping by multiple columns */
SELECT SKU, SKU_Description, SUM(QuantityOnHand)
FROM INVENTORY
GROUP BY SKU, SKU_Description;
```

A Common GROUP BY Error

```
/* The following does not work! */
SELECT SKU, SKU_Description, SUM(QuantityOnHand)
FROM INVENTORY
GROUP BY SKU;
```

Problem: **`SKU_Description`** is neither a grouping-column nor an aggregate.

Solution: Add **`SKU_Description`** to the **`GROUP BY`** clause.

The `HAVING` Clause

The group-wise equivalent of `WHERE` is the **`HAVING`** clause:

SELECT *column-list*
FROM *tables*
WHERE *row-conditions*
GROUP BY *grouping-column-list*
HAVING *aggregate-conditions*

The `HAVING` clause applies conditions to the groups formed by the `GROUP BY` clause.

HAVING Example

```
SELECT SKU, SUM(QuantityOnHand) AS QuantOnHand
FROM INVENTORY
GROUP BY SKU
HAVING QuantOnHand < 1000;
```

`WHERE` vs `HAVING`

It's important to keep in mind the subtle difference between these two seemingly similar clauses:

- **`WHERE`** chooses which **rows** to include.
- **`HAVING`** chooses which **groups** to include.

So, don't try to do group-wise conditions in the **`WHERE`** clause or row-wise conditions in the **`HAVING`** clause.

ORDER BY **with Groups**

If the statement has a `GROUP BY` clause then the `ORDER BY` clause applies to the groups, not the rows.

```
SELECT SKU, SUM(QuantityOnHand) AS QuantOnHand
FROM INVENTORY
GROUP BY SKU
ORDER BY QuantOnHand ASC;
```

Joins

Querying multiple tables using foreign key references

Three Ways to Query Multiple Tables

- **Implicit Joins** with multiple tables listed in the **`FROM`** clause with match conditions given in the **`WHERE`** clause.
- **Explicit Joins** with SQL **`JOIN ON`** or **`JOIN USING`** operators in the **`FROM`** clause.
- **Subqueries** with a subquery in the **`WHERE`** clause that matches records from the *parent* query to records in the *child* subquery.

More about this later ...

Implicit Joins -- Cross Products

What happens if we try this?

```
SELECT *
FROM RETAIL_ORDER, ORDER_ITEM;
```

Each record (all columns) from the first table is merged with each record (again, all columns) from the second table. This is called a **cartesian cross product**.

To stick with the Join theme, a cartesian product in the `FROM` clause is sometimes called a **Cross Join**.

Implicit Joins -- `WHERE` Clause

Once we have a Cross Product, we can then use the **`WHERE`** clause to narrow the results down to just the rows we want (i.e., the ones where the keys match):

```
SELECT *
FROM RETAIL_ORDER, ORDER_ITEM
WHERE
RETAIL_ORDER.OrderNumber = ORDER_ITEM.OrderNumber;
```

Cross Product

Key match using TABLE.Column notation to refer to columns with identicals names on different tables

Equijoins and Theta Joins

Looking for a key match is called an **equijoin** because the keys on either side of the match have to be equal (identical).

A **theta join** relaxes this a bit to allow inexact matches and even inequalities:

```
SELECT *
FROM SKU_DATA as SD1, SKU_DATA as SD2
WHERE (SD1.SKU - SD2.SKU) BETWEEN 0 and 1000;
```

Useful when dealing with floating point numbers and times.

Imprecise match!

Explicit Joins: SQL `JOIN ON` Syntax

SQL Joins (typically) use some variation of

`table-left` **`JOIN`** `table-right` **`ON`** `(join-condition)`

where *`join-condition`* is usually a key match.

Three variations:

- *`table-left`* **`INNER`** `JOIN` *`table-right`* `ON …`
- *`table-left`* **`LEFT`** `JOIN` *`table-right`* `ON …`
- *`table-left`* **`RIGHT`** `JOIN` *`table-right`* `ON …`

We'll take them one at a time.

Inner Joins

table-left **INNER JOIN** *table-right* **ON** …

```
SELECT *
FROM RETAIL_ORDER INNER JOIN ORDER_ITEM ON
(RETAIL_ORDER.OrderNumber = ORDER_ITEM.OrderNumber);
```

- Inner Joins are the *default* join type
- Only rows from both tables where the join condition is satisfied are included in the results

Outer Joins

Outer Joins are used when we might want to show records in table A that have no matching record in table B.

Which table is table A depends on which kind of outer join we are using:

- Left Outer Joins treat the first table as table A
- Right Outer Joins treat the second table as table A

Left Outer Joins

table-left **LEFT JOIN** *table-right* **ON** …

```
SELECT *
FROM RETAIL_ORDER LEFT JOIN ORDER_ITEM ON
(RETAIL_ORDER.OrderNumber = ORDER_ITEM.OrderNumber);
```

- Left Joins include every record from *table-left* but only records from *table-right* where the join condition is satisfied.

Right Outer Joins

table-left **RIGHT JOIN** *table-right* **ON** …

```
SELECT *
FROM RETAIL_ORDER RIGHT JOIN ORDER_ITEM ON
(RETAIL_ORDER.OrderNumber = ORDER_ITEM.OrderNumber);
```

- Right Joins include every record from *table-right* but only records from *table-left* where the join condition is satisfied.

Explicit Equijoins: `JOIN … USING` Syntax

`JOIN … USING` is a handy shorthand for **`JOIN … ON`** that works for equijoins:

    table-left **JOIN** *table-right* **USING** (*equi-columns*)

where *equi-columns* is a list of columns to match.

- The *equi-columns* have to exist in both tables.
- Since the *equi-columns* have identical values on either side of the join, they are collapsed into a single column.

A Better Join Example

STUDENT

StudentPK	StudentName	LockerFK
1	Adams	NULL
2	Buchanan	NULL
3	Carter	10
4	Ford	20
5	Hoover	30
6	Kennedy	40
7	Roosevelt	50
8	Truman	60

LOCKER

LockerPK	LockerType
10	Full
20	Full
30	Half
40	Full
50	Full
60	Half
70	Full
80	Full
90	Half

(a) The STUDENT and LOCKER Tables Aligned to Show Row Relationships

INNER JOIN

Only the rows where LockerFK=LockerPK are shown—Note that some StudentPK and some LockerPK values are not in the results

StudentPK	StudentName	LockerFK	LockerPK	LockerType
3	Carter	10	10	Full
4	Ford	20	20	Full
5	Hoover	30	30	Half
6	Kennedy	40	40	Full
7	Roosevelt	50	50	Full
8	Truman	60	60	Half

(b) INNER JOIN of the STUDENT and LOCKER Tables

Outer Join Example

LEFT JOIN

All rows from STUDENT are shown, even where there is no matching LockerFK=LockerPK value

StudentPK	StudentName	LockerFK	LockerPK	LockerType
1	Adams	NULL	NULL	NULL
2	Buchanan	NULL	NULL	NULL
3	Carter	10	10	Full
4	Ford	20	20	Full
5	Hoover	30	30	Half
6	Kennedy	40	40	Full
7	Roosevelt	50	50	Full
8	Truman	60	60	Half

(c) LEFT OUTER JOIN of the STUDENT and LOCKER Tables

NULLs where there is no key match

RIGHT JOIN

All rows from LOCKER are shown, even where there is no matching LockerFK=LockerPK value

StudentPK	StudentName	LockerFK	LockerPK	LockerType
3	Carter	10	10	Full
4	Ford	20	20	Full
5	Hoover	30	30	Half
6	Kennedy	40	40	Full
7	Roosevelt	50	50	Full
8	Truman	60	60	Half
NULL	NULL	NULL	70	Full
NULL	NULL	NULL	80	Full
NULL	NULL	NULL	90	Half

(d) RIGHT OUTER JOIN of the STUDENT and LOCKER Tables

Chained Joins

Since a **`JOIN`** always returns a table-like result set, we can join the result with another table, then another, etc.

```
SELECT *
FROM RETAIL_ORDER
    JOIN ORDER_ITEM ON (RETAIL_ORDER.OrderNumber =
    ORDER_ITEM.OrderNumber)
    JOIN SKU_DATA ON (SKU_DATA.SKU = ORDER_ITEM.SKU);
```

Subqueries

Using a SELECT query as a SQL expression inside another query … sort of like the movie "Inception"

SELECT queries as SQL Expressions

Any SELECT query can be made into a SQL expression by wrapping it inside parentheses:

```
(SELECT SKU, SUM(QuantityOnHand) AS QuantOnHand
 FROM INVENTORY GROUP BY SKU)
```

When used inside another query this kind of expression is called a **subquery**.

Subqueries in the `WHERE` Clause

```
SELECT SUM(ExtendedPrice) AS WaterSportRevenue
FROM ORDER_ITEM
WHERE SKU IN
   (SELECT SKU FROM SKU_DATA WHERE
   Department='Water Sports');
```

- The child subquery is run **before** the parent query
- The subquery returns a set of keys that can be matched between the tables, just like a join

Aggregates in the `WHERE` Clause

```
SELECT *
FROM ORDER_ITEM
WHERE ExtendedPrice >
   (SELECT AVG(ExtendedPrice) FROM ORDER_ITEM);
```

This selects any `ORDER_ITEM` with an above average `ExtendedPrice`.

This is a special case where we don't want to use `GROUP BY`. Instead, We use a subquery because we need to compute the `AVG()` **over all rows** (i.e., not a subgroup) before we can compare it with the ExtendedPrice for the current row.

Subqueries are not Joins

The textbooks tends to conflate the two a bit, but **a subquery is not the same thing as a join**.

- Subqueries are not nearly as efficient, generating a new result-set each time instead of just matching keys directly.
- However, subqueries can *sometimes* do things that can't be done with joins.

Subqueries in the `FROM` clause

```
SELECT *
FROM (SELECT SKU, SUM(QuantityOnHand) AS
QuantOnHand FROM INVENTORY GROUP BY SKU) AS SQ;
```

Tips:

- This usage is not very common
- The subquery must be aliased with `AS` in order to use it here.
- We sometimes use this to "decorate" the results of a subquery by adding columns to the `SELECT` clause.

SQL Views (again)

You can't do this one on the server. You don't have permission ot create views.

```
CREATE VIEW SKU_QUANT_ON_HAND
AS (SELECT SKU, SUM(QuantityOnHand) AS QuantOnHand
FROM INVENTORY GROUP BY SKU);
```

While *technically* not a subquery, **a view can be used anywhere you can use a subquery**. They are also more efficient.

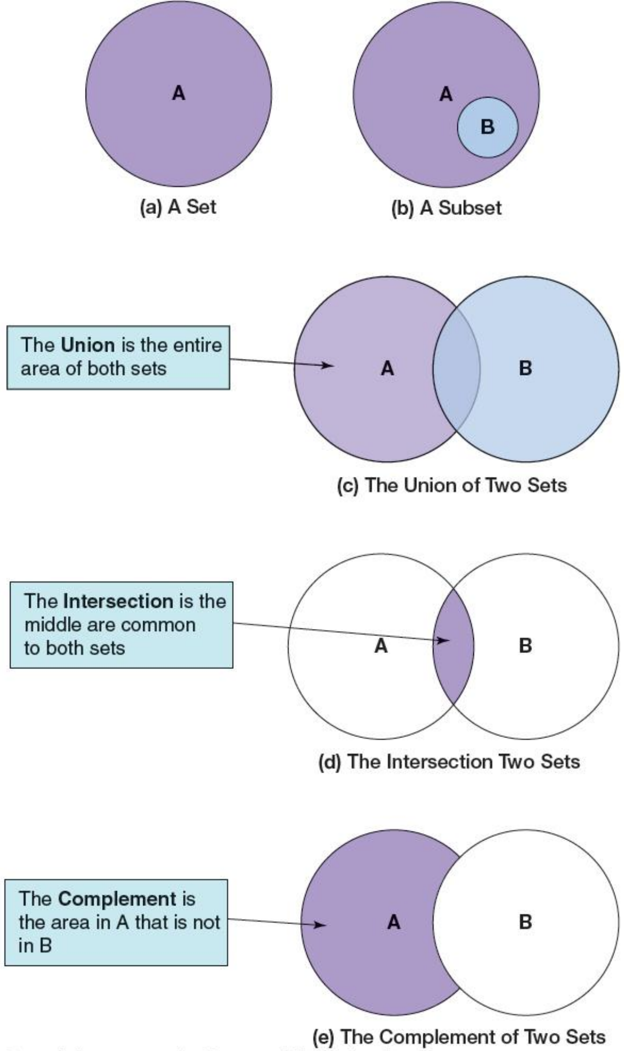
Set Operators

When Joins and Subqueries are still not enough

Set Algebraic Features

SQL is a pretty full-featured tool for set algebra, where the sets the records inside the tables.

SQL Set Operators	
Operator	**Meaning**
UNION	The result is all the row values in one or both tables
INTERSECT	The result is all the row values common to both tables
EXCEPT	The result is all the row values in the first table but not the second



SQL `UNION` Queries

Whereas a join merges the columns of two tables horizontally (i.e., "wider"), a union merges the tables vertically (i.e., "taller"), stacking the rows of one SELECT query on top of the rows of another.

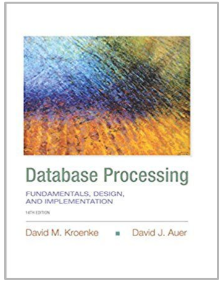
```
SELECT * FROM CATALOG_SKU_2013
UNION
SELECT * FROM CATALOG_SKU_2014
```

The columns in both tables have to be compatible. Same column order, data types, etc.

Intersection and Except Operators

These are rarely used in practice. They are included in SQL for completeness, not practicality.

So … let's move on to the next chapter.



Databases for Analytics

Kroenke / Auer
Chapter 2