Intermediate SQL: Subqueries, Joins, and Compound Queries

Chapter 6

Michael Tsiang

Stats 167: Introduction to Databases

UCLA

Subqueries

Joins

Compound Queries

Subqueries

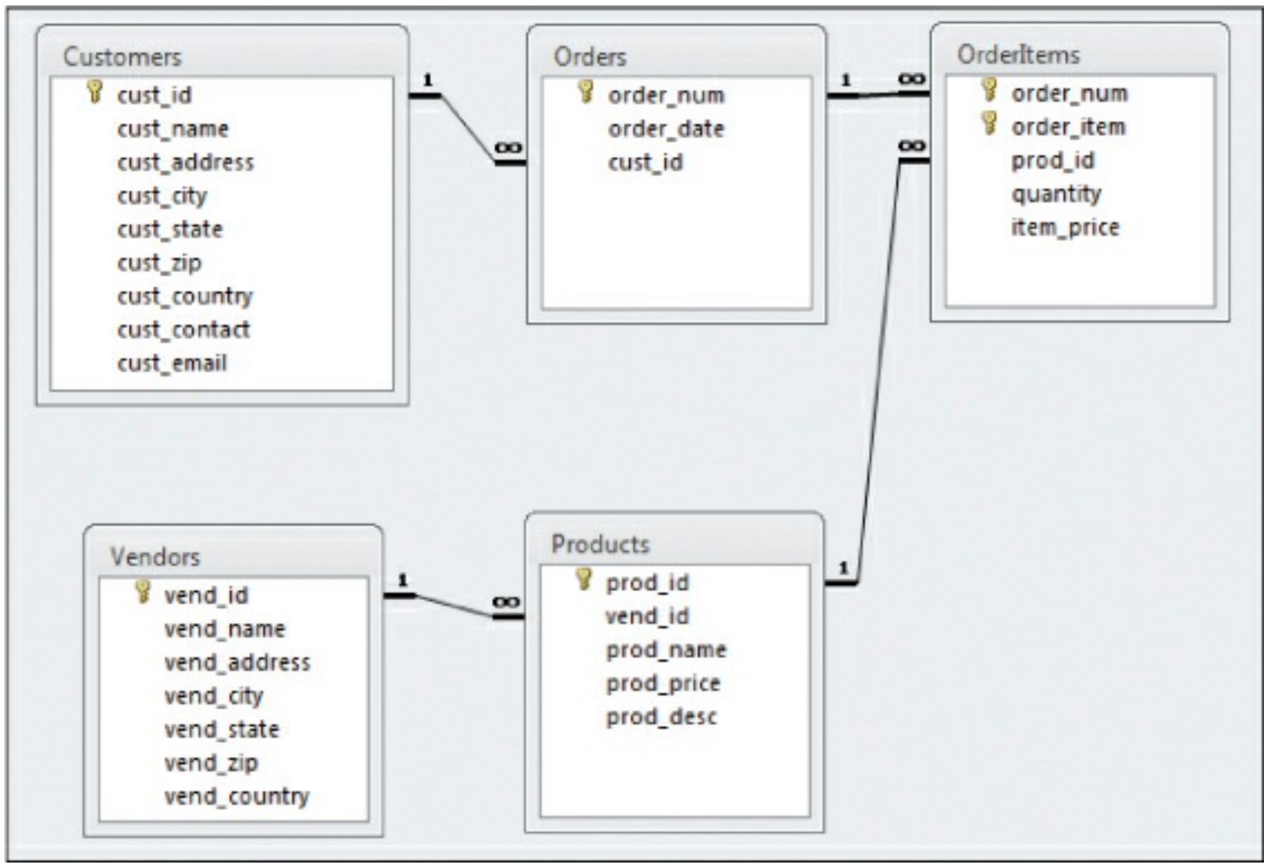
Moving Beyond Simple Queries

In the previous chapters, all of the SQL queries we considered are **simple queries**, since they only retrieved data from individual tables in a database.

We now want to consider more complex SQL queries that require accessing data from multiple tables at once.

Schema of the TYSQL Data

We will continue to use the `TYSQL.sqlite` data, which contains data for a fictitious toy store. The schema of the `TYSQL` database:



Source: Ben Forta, *SQL in 10 Minutes a Day, Sams Teach Yourself, 5th Edition*, Pearson, 2020.

Motivating Example

Notice from the `TYSQL` schema that data on orders are stored in two tables, `Orders` and `OrderItems`.

The `Orders` table does not store customer information; it only stores the customer ID. The customer information is stored in the `Customers` table.

Suppose we want a list of the customers who ordered the item `RGAN01`. What do we need to do to retrieve this information?

1. Retrieve the order numbers of all orders containing `RGAN01`.
2. Retrieve the corresponding customer IDs for all of the order numbers returned in the previous step.
3. Retrieve the customer information for all the customer IDs returned in the previous step.

Hard-Coded Solution: Step 1

1. Retrieve the order numbers of all orders containing `RGAN01`.

```sql
SELECT order_num
FROM OrderItems
WHERE prod_id = 'RGAN01';
```

order_num
20007
20008

Hard-Coded Solution: Step 2

2. Retrieve the corresponding customer IDs for all of the order numbers returned in the previous step.

```
SELECT cust_id
FROM Orders
WHERE order_num IN (20007, 20008);
```

cust_id
1000000004
1000000005

Hard-Coded Solution: Step 3

3. Retrieve the customer information for all the customer IDs returned in the previous step.

```
SELECT cust_name, cust_contact
FROM Customers
WHERE cust_id IN (1000000004, 1000000005);
```

cust_name	cust_contact
Fun4All	Denise L. Stephens
The Toy Store	Kim Howard

The Issue with Hard-Coded Solutions

While each step in this solution was only a simple query, each subsequent query relied on the output column from the previous query.

- What if the number of retrieved order numbers was much larger?
- What if the number of retrieved customer IDs was much larger?

Having to copy-paste the specific values from each output table into the next query is not efficient, not generalizable, and may be prone to errors.

Is there a better way to answer this question? How can we use the results from one query in another query without observing every individual value?

Subqueries

A **subquery** is a query nested within a larger query.

Rather than copy-pasting the results of a query into a second query, we can simply use the first query as a subquery. Using subqueries, we will be able to combine all three queries into one single statement.

Note:

- Subqueries are always processed starting with the innermost `SELECT` statement and working outward, similar to how composition of functions works in mathematics and in programming languages like R or Python.
- Subquery `SELECT` statements can only retrieve a single column. Attempting to retrieve multiple columns will return an error.

Subquery Solution (Part 1)

Note that the order numbers (20007, 20008) in the second query were the output from the first query. We can thus instead nest the first query inside the second:

```
SELECT cust_id
FROM Orders
WHERE order_num IN (SELECT order_num
                    FROM OrderItems
                    WHERE prod_id = 'RGAN01');
```

cust_id
1000000004
1000000005

Subquery Solution (Part 2)

Putting all three queries together:

```
SELECT cust_name, cust_contact
FROM Customers
WHERE cust_id IN (SELECT cust_id
                  FROM Orders
                  WHERE order_num IN (SELECT order_num
                                      FROM OrderItems
                                      WHERE prod_id = 'RGAN01'));
```

cust_name	cust_contact
Fun4All	Denise L. Stephens
The Toy Store	Kim Howard

Note: Even though extra whitespace is ignored in SQL, breaking up queries over multiple lines and indenting subqueries can help make complex queries more readable.

Subqueries as Calculated Fields

Another way to use subqueries is in creating calculated fields.

Suppose we want to display the total number of orders placed by every customer in the `Customers` table. We would need to perform the following queries:

- Retrieve the list of customers from the `Customers` table.
- For each customer retrieved, count the number of associated orders in the `Orders` table.

Recall: SELECT COUNT(*)

Recall that the `SELECT COUNT(*)` statement is used to count rows in a table. By providing a `WHERE` clause to filter a specific customer ID, we can count the number of orders for that customer.

For example, the following query will count the number of orders placed by customer 1000000001:

```sql
SELECT COUNT(*) AS orders
FROM Orders
WHERE cust_id = 1000000001;
```

Using COUNT(*) as a Subquery

To perform a `COUNT(*)` calculation for each customer, we can use `COUNT(*)` as a subquery.

For example:

```
SELECT cust_name,
       cust_state,
       (SELECT COUNT(*)
       FROM Orders
       WHERE Orders.cust_id = Customers.cust_id) AS orders
FROM Customers
ORDER BY cust_name;
```

cust_name	cust_state	orders
Fun4All	IN	1
Fun4All	AZ	1
Kids Place	OH	0
The Toy Store	IL	1
Toy Land	NY	0
Village Toys	MI	2

Fully Qualified Column Names

Notice the syntax in the previous `WHERE` clause:

```
WHERE Orders.cust_id = Customers.cust_id
```

This clause tells SQL to compare the `cust_id` in the `Orders` table to the one currently being retrieved from the `Customers` table.

The syntax of **`table_name.column_name`** is the **fully qualified column name**, since it fully specifies the column we are referring to. When referencing more than one table, it is important to use fully qualified column names to avoid any ambiguity or unintended references (some DBMSs will throw an error).

Subquery Summary

Subqueries are a very useful way to combine simple queries into powerful and flexible SQL statements.

However, even though there is no limit to the number of subqueries that can be nested, too many nested subqueries can be slow and inefficient in practice.

We will see in the next section how to rewrite a complex statement with subqueries in a simpler way.

Joins

Motivation

Relational database design (i.e., breaking data into multiple related tables) enables more efficient and reliable storage, easier manipulation, and greater scalability.

But if data is stored in multiple tables, how can we retrieve the data with a single `SELECT` statement?

We need a way of using the relations between tables to combine the data together as needed.

Joins

One of SQL's most powerful features is the capability to join tables on the fly (i.e., during execution) within queries.

A **join** is a mechanism to combine (or *join*) tables within a single `SELECT` statement. Using join syntax, we can join multiple tables so that a single set of output is returned, and the join associates the correct rows in each table on the fly.

Note that joins do not exist in the actual database tables. A join is created by the DBMS as needed, and it persists for the duration of the query execution.

A Simple Example of a Join

To create a join, we must specify all the tables to be included and how they are related to each other.

For example:

```
SELECT vend_name, prod_name, prod_price
FROM Vendors, Products
WHERE Vendors.vend_id = Products.vend_id;
```

vend_name	prod_name	prod_price
Bears R Us	8 inch teddy bear	5.99
Bears R Us	12 inch teddy bear	8.99
Bears R Us	18 inch teddy bear	11.99
Doll House Inc.	Fish bean bag toy	3.49
Doll House Inc.	Bird bean bag toy	3.49
Doll House Inc.	Rabbit bean bag toy	3.49
Doll House Inc.	Raggedy Ann	4.99
Fun and Games	King doll	9.49
Fun and Games	Queen doll	9.49

Analyzing the Join Example

In the preceding query, notice that the columns in the `SELECT` statement refer to two different tables, so the `FROM` clause lists the two tables that are being joined.

The `WHERE` clause was used to set the join relationship between the `Vendors` and `Products` tables. This clause instructs the DBMS to match the `vend_id` in the `Vendors` table with the `vend_id` in the `Products` table.

The join condition typically relates `table1.primary_key` with `table2.foreign_key`, since primary and foreign keys are how the database is able to identify corresponding rows in different tables.

Note: You *must* use the fully qualified column names of `Vendors.vend_id` and `Products.vend_id` here. Since there are two `vend_id` columns (one in each table), just using the `vend_id` name alone is ambiguous and will throw an error.

Inner Joins

The preceding join is an example of an **inner join**, which pairs each row in one table with the matching row(s) in the other table.

The explicit (and more standard) way to specify an inner join is with the `INNER JOIN` clause along with the `ON` clause that specifies the join condition.

An equivalent way to write the previous example is with the `INNER JOIN` clause:

```
SELECT vend_name, prod_name, prod_price
FROM Vendors
INNER JOIN Products
ON Vendors.vend_id = Products.vend_id;
```

Note: An inner join where the join condition is based on testing equality (with =) is also called an **equijoin**.

Joining Multiple Tables

SQL has no limit on the number of tables that can be joined in a `SELECT` statement.

For example:

```
SELECT prod_name, vend_name, prod_price, quantity
FROM Products
INNER JOIN OrderItems
ON OrderItems.prod_id = Products.prod_id
INNER JOIN Vendors
ON Products.vend_id = Vendors.vend_id
WHERE order_num = 20007;
```

prod_name	vend_name	prod_price	quantity
18 inch teddy bear	Bears R Us	11.99	50
Fish bean bag toy	Doll House Inc.	3.49	100
Bird bean bag toy	Doll House Inc.	3.49	100
Rabbit bean bag toy	Doll House Inc.	3.49	100
Raggedy Ann	Doll House Inc.	4.99	50

Implicit Mulitple Inner Join

An implicit way to write the previous join using the `WHERE` clause:

```
SELECT prod_name, vend_name, prod_price, quantity
FROM OrderItems, Products, Vendors
WHERE Products.vend_id = Vendors.vend_id
   AND OrderItems.prod_id = Products.prod_id
   AND order_num = 20007;
```

prod_name	vend_name	prod_price	quantity
18 inch teddy bear	Bears R Us	11.99	50
Fish bean bag toy	Doll House Inc.	3.49	100
Bird bean bag toy	Doll House Inc.	3.49	100
Rabbit bean bag toy	Doll House Inc.	3.49	100
Raggedy Ann	Doll House Inc.	4.99	50

Rewriting Complex Subqueries as Joins

Recall the nested subqueries below:

```
SELECT cust_name, cust_contact
FROM Customers
WHERE cust_id IN (SELECT cust_id
                  FROM Orders
                  WHERE order_num IN (SELECT order_num
                                      FROM OrderItems
                                      WHERE prod_id = 'RGAN01'));
```

cust_name	cust_contact
Fun4All	Denise L. Stephens
The Toy Store	Kim Howard

Since subqueries are not always the most efficient way to perform complex queries, how can we rewrite this using joins?

Complex Subqueries as Joins (Explicit Join Solution)

As an explicit multiple inner join:

```
SELECT cust_name, cust_contact
FROM Customers
INNER JOIN Orders
ON Customers.cust_id = Orders.cust_id
INNER JOIN OrderItems
ON Orders.order_num = OrderItems.order_num
WHERE prod_id = 'RGAN01';
```

cust_name	cust_contact
Fun4All	Denise L. Stephens
The Toy Store	Kim Howard

Complex Subqueries as Joins (Implicit Join Solution)

As an implicit multiple inner join:

```
SELECT cust_name, cust_contact
FROM Customers, Orders, OrderItems
WHERE Customers.cust_id = Orders.cust_id
   AND Orders.order_num = OrderItems.order_num
   AND prod_id = 'RGAN01';
```

cust_name	cust_contact
Fun4All	Denise L. Stephens
The Toy Store	Kim Howard

Types of Joins

There are actually many types of joins:

- Inner Joins
- Cross Joins
- Self Joins
- Natural Joins
- Outer Joins

We will consider examples of all of them.

Cross Joins

A **cross join** pairs every row in one table with every row in the second table. A cross join is also called a **Cartesian product**.

The `CROSS JOIN` clause is used to create cross joins.

```
SELECT vend_name, prod_name, prod_price
FROM Vendors
CROSS JOIN Products;
```

An implicit cross join can be made by referencing multiple tables in the `SELECT` and `FROM` clauses without specifying a join condition:

```
SELECT vend_name, prod_name, prod_price
FROM Vendors, Products;
```

Self Joins

A **self join** is used to join a table to itself.

For example, suppose we want to send an advertisement for our toy store to all the customer contacts who work for the same company that Jim Jones works for. We would need to perform the following queries:

- Retrieve which company Jim Jones works for.
- Retrieve which customers work for that company.

Both queries refer to the same table. How can we write this as a single `SELECT` statement?

Subquery Solution

One way to find all the customer contacts who work for the same company that Jim Jones works for is through a subquery:

```
SELECT cust_id, cust_name, cust_contact
FROM Customers
WHERE cust_name = (SELECT cust_name
                   FROM Customers
                   WHERE cust_contact = 'Jim Jones');
```

cust_id	cust_name	cust_contact
1000000003	Fun4All	Jim Jones
1000000004	Fun4All	Denise L. Stephens

How can we write this as a (self) join?

Self Join Solution

In order to avoid confusion when referring to the same table in multiple ways within a single `SELECT` statement, we must create table **aliases** with the `AS` clause. Without using aliases, this query will throw an error.

```
SELECT c1.cust_id, c1.cust_name, c1.cust_contact
FROM Customers AS c1, Customers AS c2
WHERE c1.cust_name = c2.cust_name
   AND c2.cust_contact = 'Jim Jones';
```

cust_id	cust_name	cust_contact
1000000003	Fun4All	Jim Jones
1000000004	Fun4All	Denise L. Stephens

Side Note: The Oracle DBMS does not support the `AS` keyword when aliasing tables. In Oracle, simply omit the `AS` (e.g., write `Customers c1` instead of `Customers AS c1`).

Self Join Solution (Explicit Inner Join)

The self join can also be written as an explicit inner join:

```sql
SELECT c1.cust_id, c1.cust_name, c1.cust_contact
FROM Customers AS c1
INNER JOIN Customers AS c2
ON c1.cust_name = c2.cust_name
WHERE c2.cust_contact = 'Jim Jones';
```

cust_id	cust_name	cust_contact
1000000003	Fun4All	Jim Jones
1000000004	Fun4All	Denise L. Stephens

Natural Joins

A **natural join** is an equijoin with an implicit join condition based on comparing all columns in both tables that have the same column names. The output table contains only one column for each pair of equally named columns.

If the tables have no column names in common, the output is a cross join.

```
SELECT *
FROM Orders
NATURAL JOIN OrderItems
LIMIT 10;
```

order_num	order_date	cust_id	order_item	prod_id	quantity	item_price
20005	2020-05-01	1000000001	1	BR01	100	5.49
20005	2020-05-01	1000000001	2	BR03	100	10.99
20006	2020-01-12	1000000003	1	BR01	20	5.99
20006	2020-01-12	1000000003	2	BR02	10	8.99
20006	2020-01-12	1000000003	3	BR03	10	11.99
20007	2020-01-30	1000000004	1	BR03	50	11.49
20007	2020-01-30	1000000004	2	BNBG01	100	2.99
20007	2020-01-30	1000000004	3	BNBG02	100	2.99
20007	2020-01-30	1000000004	4	BNBG03	100	2.99
20007	2020-01-30	1000000004	5	RGAN01	50	4.49

Natural Joins Versus Inner Joins

Notice the difference with the explicit join condition in the inner join:

```
SELECT *
FROM Orders
INNER JOIN OrderItems
ON OrderItems.order_num = Orders.order_num
LIMIT 10;
```

order_num	order_date	cust_id	order_num	order_item	prod_id	quantity	item_price
20005	2020-05-01	1000000001	20005	1	BR01	100	5.49
20005	2020-05-01	1000000001	20005	2	BR03	100	10.99
20006	2020-01-12	1000000003	20006	1	BR01	20	5.99
20006	2020-01-12	1000000003	20006	2	BR02	10	8.99
20006	2020-01-12	1000000003	20006	3	BR03	10	11.99
20007	2020-01-30	1000000004	20007	1	BR03	50	11.49
20007	2020-01-30	1000000004	20007	2	BNBG01	100	2.99
20007	2020-01-30	1000000004	20007	3	BNBG02	100	2.99
20007	2020-01-30	1000000004	20007	4	BNBG03	100	2.99
20007	2020-01-30	1000000004	20007	5	RGAN01	50	4.49

The `order_num` column is shown twice in the explicit inner join, but only appears once in the natural join.

Notes on Natural Joins

Natural joins are a special type of inner[1] join where the join condition does not need to be specified explicitly, so natural joins are shorter, and they (by default) remove repeated column names.

Discussion of natural joins is included here for completeness. However, in practice, it is usually better, safer, and less prone to unintended errors to explicitly specify the join condition.

Note: Not all DBMSs support natural joins.

[1] There are natural outer joins too, but the default is an inner join.

Outer Joins

Most joins relate rows in one table with rows in another, but occasionally we want to include rows that have no related rows.

An **outer join** is a join that includes rows that have no associated rows in the related table.

Note: The syntax to create outer joins can vary based on different SQL implementations. Refer to the documentation for the DBMS you are using to verify its syntax.

Left and Right Outer Joins

A outer join is created with the `OUTER JOIN` keywords.

Unlike inner joins, which relate rows in both tables, outer joins also include rows with no related rows. Thus, we must also include the `LEFT` or `RIGHT` keywords to specify the table from which to include all rows. The syntax is as follows:

```
SELECT *
FROM left_table
LEFT/RIGHT OUTER JOIN right_table
ON join_condition;
```

The left/right directions refer to the two sides of the `JOIN` keyword (if `FROM left_table LEFT/RIGHT OUTER JOIN right_table` was written on one line).

A **left outer join** (or just **left join**) includes all rows from the table on the left (the one specified in the `FROM` clause).

A **right outer join** (or just **right join**) includes all rows from the table on the right (the one specified in the `OUTER JOIN` clause).

Left Outer Join Example

For example, the **left outer join** below retrieves a list of all customers and their orders:

```
SELECT Customers.cust_id, Orders.order_num
FROM Customers
LEFT OUTER JOIN Orders
ON Customers.cust_id = Orders.cust_id;
```

cust_id	order_num
1000000001	20005
1000000001	20009
1000000002	NA
1000000003	20006
1000000004	20007
1000000005	20008
1000000006	NA

Right Join Functionality

Note: SQLite supports `LEFT OUTER JOIN` but does not support `RIGHT OUTER JOIN`. Using `RIGHT OUTER JOIN` will throw an error in SQLite.

The only difference between left outer joins and right outer joins is the order of the tables they are relating. In other words, a left outer join can be turned into a right outer join by just reversing the order of the tables in the `FROM` or `WHERE` clause.

Your Left or My Left?

For example, the left outer join below is equivalent to a right outer join with `Customers` and `Orders` switched:

```
SELECT Customers.cust_id, Orders.order_num
FROM Orders
LEFT OUTER JOIN Customers
ON Customers.cust_id = Orders.cust_id;
```

cust_id	order_num
1000000001	20005
1000000003	20006
1000000004	20007
1000000005	20008
1000000001	20009

Full Outer Joins

Another variant of the outer join is the **full outer join**, which includes unrelated rows from both tables. The `FULL OUTER JOIN` keywords create a full outer join.

An example of the syntax for a full outer join:

```
SELECT Customers.cust_id, Orders.order_num
FROM Customers
FULL OUTER JOIN Orders
ON Customers.cust_id = Orders.cust_id;
```

Note: The `FULL OUTER JOIN` syntax is not supported by MariaDB, MySQL, or SQLite.

Using Joins with Aggregate Functions

We can use aggregate functions with joins just as we did with queries involving single tables.

For example, if we want to retrieve a list of all customers and the number of orders that each has placed, we can use the following query:

```
SELECT Customers.cust_id,
       COUNT(Orders.order_num) AS num_ord
FROM Customers
LEFT OUTER JOIN Orders
ON Customers.cust_id = Orders.cust_id
GROUP BY Customers.cust_id;
```

cust_id	num_ord
1000000001	2
1000000002	0
1000000003	1
1000000004	1
1000000005	1
1000000006	0

Compound Queries

Compound Queries

Most SQL queries contain a single `SELECT` statement that returns data from one or more tables. SQL also allows us to perform multiple queries and return the results as a single query result set. These combined queries are called **compound queries**.

Compound queries can sometimes be simpler to write than a single query with complicated clauses.

SQL queries are combined using the **set operators** `UNION`, `INTERSECT`, and `EXCEPT` that are (respectively) analogous to the mathematical set operators union, intersection, and set difference.

The UNION Operator

The `UNION` operator combines the results of two or more `SELECT` statements. A few rules:

- A union must be composed of two or more `SELECT` statements, each separated by the `UNION` keyword.
- Each query in a union must contain the same columns, expressions, or aggregate functions. (Some DBMSs require that columns be listed in the same order.)
- Though the column names do not need to be the same, column datatypes must be compatible (they should be the same or can be implicitly coerced).

Intuitively, a union in SQL is similar to the `rbind()` function on data frames in R. The retrieved tables to be combined need to be conformable.

Note: These rules for `UNION` also apply to `INTERSECT` and `EXCEPT`.

Union Example: Separate Statements

```
SELECT cust_name, cust_contact, cust_email
FROM Customers
WHERE cust_state IN ('IL', 'IN', 'MI');
```

cust_name	cust_contact	cust_email
Village Toys	John Smith	sales@villagetoys.com
Fun4All	Jim Jones	jjones@fun4all.com
The Toy Store	Kim Howard	kim@thetoystore.com

```
SELECT cust_name, cust_contact, cust_email
FROM Customers
WHERE cust_name = 'Fun4All';
```

cust_name	cust_contact	cust_email
Fun4All	Jim Jones	jjones@fun4all.com
Fun4All	Denise L. Stephens	dstephens@fun4all.com

Union Example: Creating a Union

```sql
SELECT cust_name, cust_contact, cust_email
FROM Customers
WHERE cust_state IN ('IL', 'IN', 'MI')
UNION
SELECT cust_name, cust_contact, cust_email
FROM Customers
WHERE cust_name = 'Fun4All';
```

cust_name	cust_contact	cust_email
Fun4All	Denise L. Stephens	dstephens@fun4all.com
Fun4All	Jim Jones	jjones@fun4all.com
The Toy Store	Kim Howard	kim@thetoystore.com
Village Toys	John Smith	sales@villagetoys.com

The UNION ALL Operator

By default, the `UNION` operator will automatically remove any duplicate rows from the query result set.

To include all occurrences of all matches, we can use the **`UNION ALL`** operator:

```
SELECT cust_name, cust_contact, cust_email
FROM Customers
WHERE cust_state IN ('IL', 'IN', 'MI')
UNION ALL
SELECT cust_name, cust_contact, cust_email
FROM Customers
WHERE cust_name = 'Fun4All';
```

cust_name	cust_contact	cust_email
Village Toys	John Smith	sales@villagetoys.com
Fun4All	Jim Jones	jjones@fun4all.com
The Toy Store	Kim Howard	kim@thetoystore.com
Fun4All	Jim Jones	jjones@fun4all.com
Fun4All	Denise L. Stephens	dstephens@fun4all.com

The INTERSECT Operator

The `INTERSECT` operator combines the results of two `SELECT` statements, but it returns only the rows from the first `SELECT` statement that are identical to rows in the second `SELECT` statement.

```
SELECT cust_name, cust_contact, cust_email
FROM Customers
WHERE cust_state IN ('IL', 'IN', 'MI')
INTERSECT
SELECT cust_name, cust_contact, cust_email
FROM Customers
WHERE cust_name = 'Fun4All';
```

cust_name	cust_contact	cust_email
Fun4All	Jim Jones	jjones@fun4all.com

The EXCEPT Operator

The `EXCEPT` operator combines two `SELECT` statements and returns the rows from the first `SELECT` statement that are *not* returned by the second `SELECT` statement.

Note: In Oracle, the `EXCEPT` operator is called `MINUS`.

```
SELECT cust_name, cust_contact, cust_email
FROM Customers
WHERE cust_state IN ('IL', 'IN', 'MI')
EXCEPT
SELECT cust_name, cust_contact, cust_email
FROM Customers
WHERE cust_name = 'Fun4All';
```

cust_name	cust_contact	cust_email
The Toy Store	Kim Howard	kim@thetoystore.com
Village Toys	John Smith	sales@villagetoys.com

Sorting Compound Query Results

Recall: The `ORDER BY` clause is used to sort output from `SELECT` queries.

When combining queries with a set operator, only one `ORDER BY` clause is allowed, and it must occur after the final `SELECT` statement. Multiple `ORDER BY` clauses are not allowed.

Note: The `ORDER BY` clause usually references the name or alias of a column in the query result, but `ORDER BY` also allows referencing the column number.

Example: Sorting Combined Results (by Column Name)

```sql
SELECT cust_name, cust_contact, cust_email
FROM Customers
WHERE cust_state IN ('IL', 'IN', 'MI')
UNION
SELECT cust_name, cust_contact, cust_email
FROM Customers
WHERE cust_name = 'Fun4All'
ORDER BY cust_name, cust_contact;
```

cust_name	cust_contact	cust_email
Fun4All	Denise L. Stephens	dstephens@fun4all.com
Fun4All	Jim Jones	jjones@fun4all.com
The Toy Store	Kim Howard	kim@thetoystore.com
Village Toys	John Smith	sales@villagetoys.com

Example: Sorting Combined Results (by Column Number)

```sql
SELECT cust_name, cust_contact, cust_email
FROM Customers
WHERE cust_state IN ('IL', 'IN', 'MI')
UNION
SELECT cust_name, cust_contact, cust_email
FROM Customers
WHERE cust_name = 'Fun4All'
ORDER BY 1, 2;
```

cust_name	cust_contact	cust_email
Fun4All	Denise L. Stephens	dstephens@fun4all.com
Fun4All	Jim Jones	jjones@fun4all.com
The Toy Store	Kim Howard	kim@thetoystore.com
Village Toys	John Smith	sales@villagetoys.com

Working with Multiple Tables

To focus on introducing the set operators, all the examples in this section are simple compound queries that combine multiple queries on the same table.

In practice, compound queries are particularly useful when combining data from multiple tables, even tables with mismatched column names.

If tables do not have consistent column names, the resulting output will inherit the column names from the first `SELECT` statement.