Using

SQLite

O'REILLY®

Jay A. Kreibich

Table of Contents

CHAPTER 5

The SELECT Command

The `SELECT` command is used to extract data from the database. Any time you want to query or return user data stored in a database table, you'll need to use the `SELECT` command.

In terms of both syntax and functionality, `SELECT` is the most complex SQL command. In most applications, it is also one of the most frequently used commands. If you want to get the most out of your database (and your database designs), you'll need a solid understanding of how to properly use `SELECT`.

Generally, the returned values are derived from the contents of the database, but `SELECT` can also be used to return the value of simple expressions. `SELECT` is a read-only command, and will not modify the database (unless the `SELECT` is embedded in a different command, such as `INSERT INTO...SELECT`).

SQL Tables

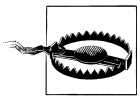
The main SQL data structure is the table. Tables are used for both storage and for data manipulation. We've seen how to define tables with the `CREATE TABLE` command, but let's look at some of the details.

A table consists of a heading and a body. The heading defines the name and type (in SQLite, the affinity) of each column. Column names must be unique within the table. The heading also defines the order of the columns, which is fixed as part of the table definition.

The table body contains all of the rows. Each row consists of one data element for each column. All of the rows in a table must have the same number of data elements, one for each column. Each element can hold exactly one data value (or a NULL).

SQL tables are allowed to hold duplicate rows. Tables can contain multiple rows where every user-defined column has an equivalent corresponding value. Duplicate rows are normally undesirable in practice, but they are allowed.

The rows in an SQL table are unordered. When a table is displayed or written down, it will have some inherent ordering, but conceptually tables have no ordering. The order of insertion has no meaning to the database.



The rows of an SQL table have no defined order.

A very common mistake is to assume a given query will always return rows in the same order. Unless you've specifically asked a query to sort the returned rows in a specific order, there is no guarantee the rows will continue to come back in the same order. Don't let your application code become dependent on the natural ordering of an unordered query. A different version of SQLite may optimize the query differently, resulting in a different row ordering. Even something as simple as adding or dropping an index can alter the row ordering of an unsorted result.

To verify your code is making no assumptions about row order, you can turn on `PRAGMA reverse_unordered_selects`. This will cause SQLite to reverse the natural row ordering of any `SELECT` statement that does not have an explicit order (an `ORDER BY` clause). See `reverse_unordered_selects` in Appendix F for more details.

The SELECT Pipeline

The `SELECT` syntax tries to represent a generic framework that is capable of expressing many different types of queries. To achieve this, `SELECT` has a large number of optional clauses, each with its own set of options and formats.

The most general format of a standalone SQLite `SELECT` statement looks like this:

```
SELECT [DISTINCT] select_heading
    FROM source_tables
    WHERE filter_expression
    GROUP BY grouping_expressions
        HAVING filter_expression
    ORDER BY ordering_expressions
    LIMIT count
        OFFSET count
```

Every `SELECT` command must have a select heading, which defines the returned values. Each additional line (`FROM`, `WHERE`, `GROUP BY`, etc.) represents an optional clause.

Each clause represents a step in the `SELECT` pipeline. Conceptually, the result of a `SELECT` statement is calculated by generating a working table, and then passing that table through the pipeline. Each step takes the working table as input, performs a specific operation or manipulation, and passes the modified table on to the next step. Manipulations operate the whole working table, similar to vector or matrix operations.

Practically, the database engine takes a few shortcuts and makes plenty of optimizations when processing a query, but the end result should always match what you would get from independently going through each step, one at a time.

The clauses in a `SELECT` statement are not evaluated in the same order they are written. Rather, their evaluation order looks something like this:

1. `FROM` *`source_tables`*

   Designates one or more source tables and combines them together into one large working table.
2. `WHERE` *`filter_expression`*

   Filters specific rows out of the working table.
3. `GROUP BY` *`grouping_expressions`*

   Groups sets of rows in the working table based off similar values.
4. `SELECT` *`select_heading`*

   Defines the result set columns and (if applicable) grouping aggregates.
5. `HAVING` *`filter_expression`*

   Filters specific rows out of the grouped table. Requires a `GROUP BY`.
6. `DISTINCT`

   Eliminates duplicate rows.
7. `ORDER BY` *`ordering_expressions`*

   Sorts the rows of the result set.
8. `OFFSET` *`count`*

   Skips over rows at the beginning of the result set. Requires a `LIMIT`.
9. `LIMIT` *`count`*

   Limits the result set output to a specific number of rows.

No matter how large or complex a `SELECT` statement may be, they all follow this basic pattern. To understand how any query works, break it down and look at each individual step. Make sure you understand what the working table looks like before each step, how that step manipulates and modifies the table, and what the working table looks like when it is passed to the next step.

FROM Clause

The `FROM` clause takes one or more source tables from the database and combines them into one large table. Source tables are usually named tables from the database, but they can also be views or subqueries (see "Subqueries" on page 76 for more details on subqueries).

Tables are combined using the `JOIN` operator. Each `JOIN` combines two tables into a larger table. Three or more tables can be joined together by stringing a series of `JOIN` operators together. `JOIN` operators are evaluated left-to-right, but there are several different types of joins, and not all of them are commutative or associative. This makes the ordering and grouping very important. If necessary, parentheses can be used to group the joins correctly.

Joins are *the* most important and most powerful database operator. Joins are the only way to bring together information stored in different tables. As we'll see in the next chapter, nearly all of database design theory assumes the user is comfortable with joins. If you can master joins, you'll be well on your way to mastering relational databases.

SQL defines three major types of joins: the `CROSS JOIN`, the `INNER JOIN`, and the `OUTER JOIN`.

CROSS JOIN

A `CROSS JOIN` matches every row of the first table with every row of the second table. If the input tables have x and y columns, respectively, the resulting table will have $x+y$ columns. If the input tables have n and m rows, respectively, the resulting table will have $n \cdot m$ rows. In mathematics, a `CROSS JOIN` is known as a Cartesian product.

The syntax for a `CROSS JOIN` is quite simple:

```
SELECT ... FROM t1 CROSS JOIN t2 ...
```

Figure 5-1 shows how a `CROSS JOIN` is calculated.

Because `CROSS JOIN`s have the potential to generate extremely large tables, care must be taken to only use them when appropriate.

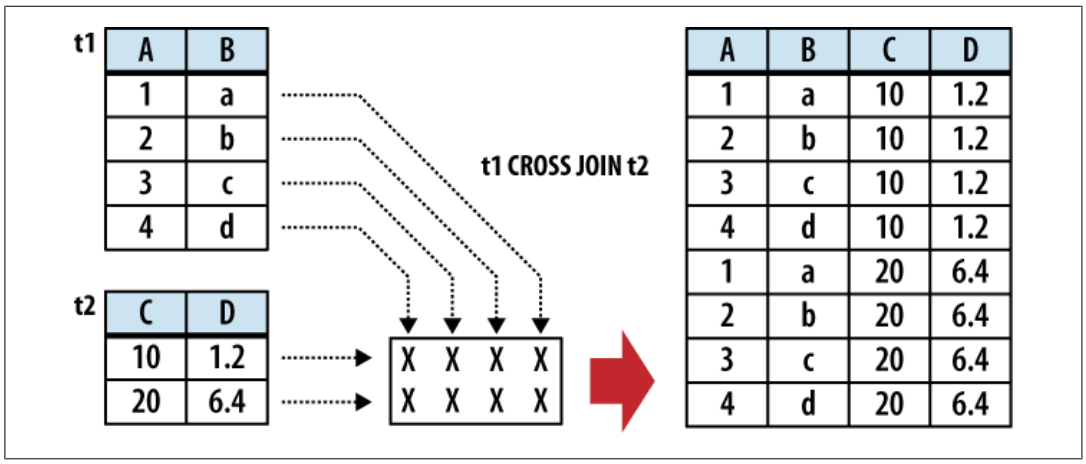


Figure 5-1. In a CROSS JOIN, each row from the first table is matched to each row in the second table.

INNER JOIN

An `INNER JOIN` is very similar to a `CROSS JOIN`, but it has a built-in condition that is used to limit the number of rows in the resulting table. The conditional is normally used to pair up or match rows from the two source tables. An `INNER JOIN` without any type of conditional expression (or one that always evaluates to true) will result in a `CROSS JOIN`. If the input tables have *x* and *y* columns, respectively, the resulting table will have no more than *x*+*y* columns (in some cases, it can have fewer). If the input tables have *n* and *m* rows, respectively, the resulting table can have anywhere from zero to *n*·*m* rows, depending on the condition. An `INNER JOIN` is the most common type of join, and is the default type of join. This makes the `INNER` keyword optional.

There are three primary ways to specify the conditional. The first is with an `ON` expression. This provides a simple expression that is evaluated for each potential row. Only those rows that evaluate to true are actually joined. A `JOIN...ON` looks like this:

```
SELECT ... FROM t1 JOIN t2 ON conditional_expression ...
```

An example of this is shown in Figure 5-2.

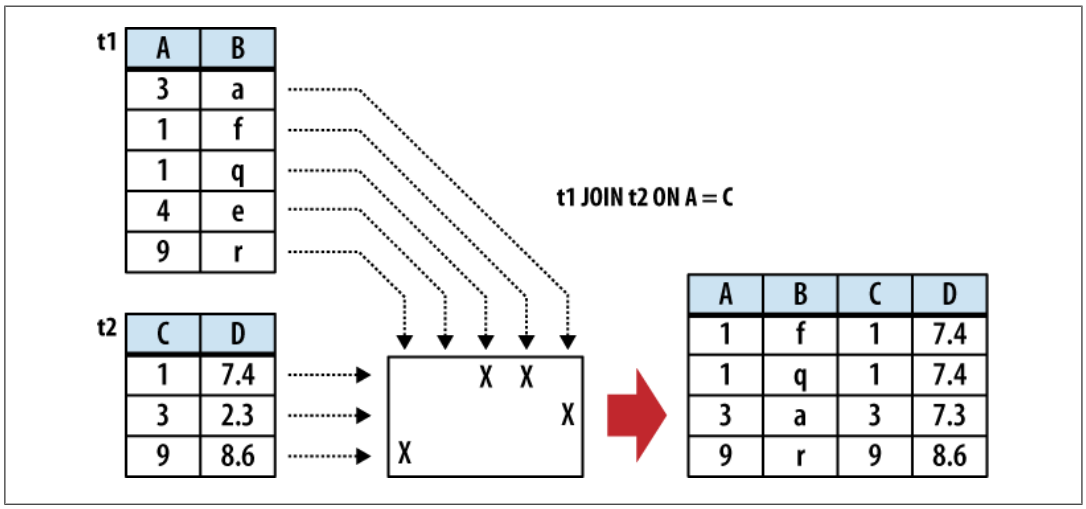


Figure 5-2. In an INNER JOIN, the rows are matched based off a condition.

If the input tables have *C* and *D* columns, respectively, a `JOIN...ON` will always result in *C*+*D* columns.

The conditional expression can be used to test for anything, but the most common type of expression tests for equality between similar columns in both tables. For example, in a business employee database, there is likely to be an `employee` table that contains (among other things) a `name` column and an `eid` column (employee ID number). Any other table that needs to associate rows to a specific employee will also have an `eid` column that acts as a pointer or reference to the correct employee. This relationship makes it very common to have queries with `ON` expressions similar to:

```
SELECT ... FROM employee JOIN resource ON employee.eid = resource.eid ...
```

This query will result in an output table where the rows from the `resource` table are correctly matched to their corresponding rows in the `employee` table.

This `JOIN` has two issues. First, that `ON` condition is a lot to type out for something so common. Second, the resulting table will have two `eid` columns, but for any given row, the values of those two columns will always be identical. To avoid redundancy and keep the phrasing shorter, inner join conditions can be declared with a `USING` expression. This expression specifies a list of one or more columns:

```
SELECT ... FROM t1 JOIN t2 USING ( col1 ,... ) ...
```

Queries from the employee database would now look something like this:

```
SELECT ... FROM employee JOIN resource USING ( eid ) ...
```

To appear in a `USING` condition, the column name must exist in both tables. For each listed column name, the `USING` condition will test for equality between the pairs of columns. The resulting table will have only one instance of each listed column.

If this wasn't concise enough, SQL provides an additional shortcut. A `NATURAL JOIN` is similar to a `JOIN...USING`, only it automatically tests for equality between the values of every column that exists in both tables:

```
SELECT ... FROM t1 NATURAL JOIN t2 ...
```

If the input tables have *x* and *y* columns, respectively, a `JOIN...USING` or a `NATURAL JOIN` will result in anywhere from *max(x,y)* to *x+y* columns.

Assuming `eid` is the only column identifier to appear in both the `employee` and `resource` table, our business query becomes extremely simple:

```
SELECT ... FROM employee NATURAL JOIN resource ...
```

`NATURAL JOIN`s are convenient, as they are very concise, and allow changes to the key structure of various tables without having to update all of the corresponding queries. They can also be a tad dangerous unless you follow some discipline in naming your columns. Because none of the columns are explicitly named, there is no error checking in the sanity of the join. For example, if no matching columns are found, the `JOIN` will automatically (and without warning) degrade to a `CROSS JOIN`, just like any other `INNER JOIN`. Similarly, if two columns accidentally end up with the same name, a `NATURAL JOIN` will automatically include them in the join condition, if you wanted it or not.

OUTER JOIN

The `OUTER JOIN` is an extension of the `INNER JOIN`. The SQL standard defines three types of `OUTER JOIN`s: `LEFT`, `RIGHT`, and `FULL`. Currently, SQLite only supports the `LEFT OUTER JOIN`.

`OUTER JOIN`s have a conditional that is identical to `INNER JOIN`s, expressed using an `ON`, `USING`, or `NATURAL` keyword. The initial results table is calculated the same way. Once the primary `JOIN` is calculated, an `OUTER` join will take any unjoined rows from one or

both tables, pad them out with NULLs, and append them to the resulting table. In the case of a `LEFT OUTER JOIN`, this is done with any unmatched rows from the first table (the table that appears to the left of the word `JOIN`).

Figure 5-3 shows an example of a `LEFT OUTER JOIN`.

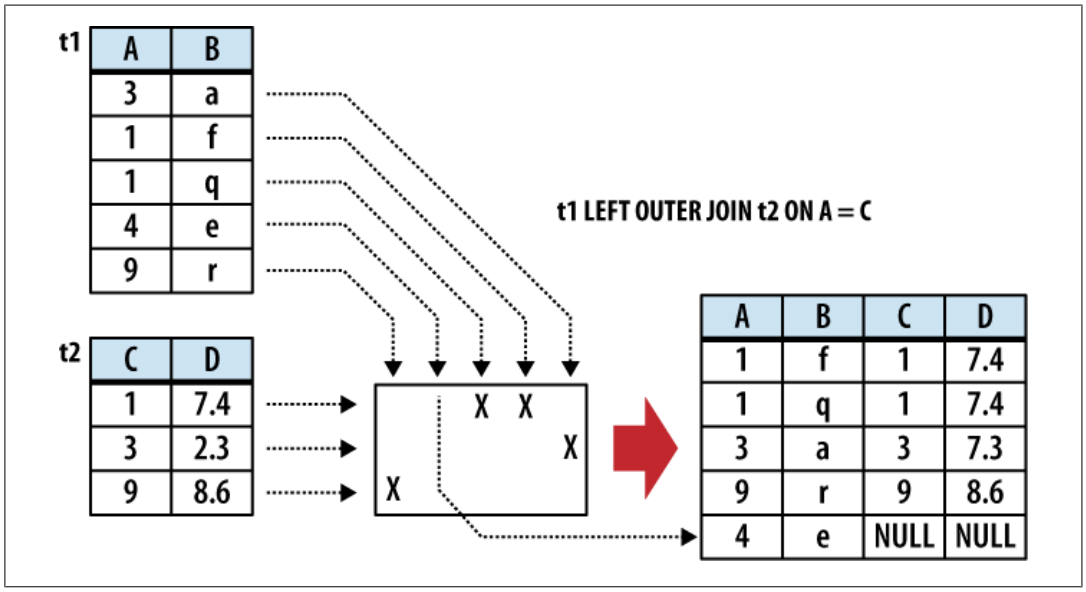


Figure 5-3. An OUTER JOIN is just like an INNER JOIN, only unmatched rows are included in the results table. This shows a LEFT OUTER JOIN, where unmatched rows from the left (t1) table are added to the results.

The result of a `LEFT OUTER JOIN` will contain at least one instance of every row from the lefthand table. If the input tables have x and y columns, respectively, the resulting table will have no more than $x+y$ columns (the exact number depends on which conditional is used). If the input tables have n and m rows, respectively, the resulting table can have anywhere from n to $n \cdot m$ rows.

Because they include unmatched rows, `OUTER JOIN`s are often specifically used to search for unresolved or "dangling" rows.

Table aliases

Because the `JOIN` operator combines the columns of different tables into one, larger table, there may be cases when the resulting working table has multiple columns with the same name. To avoid ambiguity, any part of the `SELECT` statement can qualify any column reference with a source-table name. However, there are some cases when this is still not enough. For example, there are some situations when you need to join a table to itself, resulting in the working table having two instances of the same source-table. Not only does this make every column name ambiguous, it makes it impossible to distinguish them using the source-table name. Another problem is with subqueries, as they don't have concrete source-table names.

To avoid ambiguity within the SELECT statement, any instance of a source-table, view, or subquery can be assigned an alias. This is done with the AS keyword. For example, in the cause of a self-join, we can assign a unique alias for each instance of the same table:

```
SELECT ... FROM x AS x1 JOIN x AS x2 ON x1.col1 = x2.col2 ...
```

Or, in the case of a subquery:

```
SELECT ... FROM ( SELECT ... ) AS sub ...
```

Technically, the AS keyword is optional, and each source-table name can simply be followed with an alias name. This can be quite confusing, however, so it is recommended you use the AS keyword.

If any of the subquery columns conflict with a column from a standard source table, you can now use the sub qualifier as a table name. For example, sub.col1.

Once a table alias has been assigned, the original source-table name becomes invalid and cannot be used as a column qualifier. You must use the alias instead.

WHERE Clause

The WHERE clause is used to filter rows from the working table generated by the FROM clause. It is very similar to the WHERE clause found in the UPDATE and DELETE commands. An expression is provided that is evaluated for each row. Any row that causes the expression to evaluate to false or NULL is discarded. The resulting table will have the same number of columns as the original table, but may have fewer rows. It is not considered an error if the WHERE clause eliminates every row in the working table. Figure 5-4 shows how the WHERE clause works.

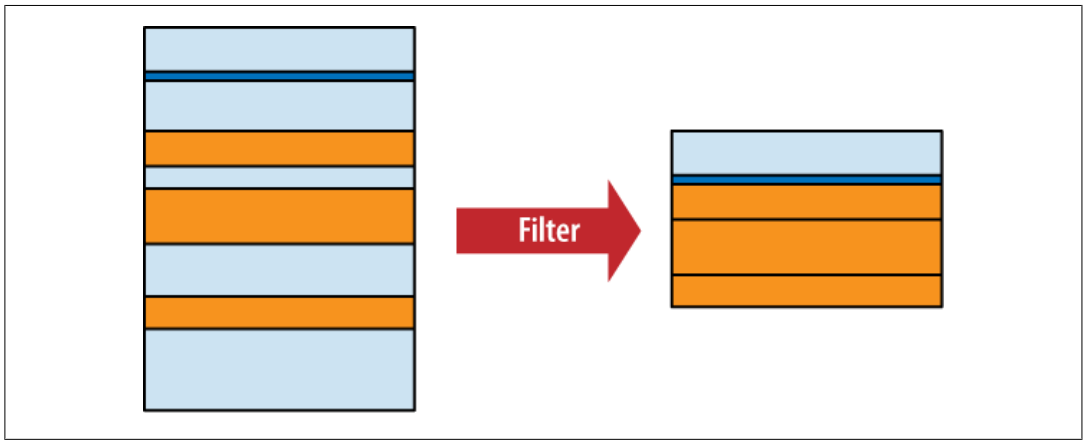


Figure 5-4. The WHERE clause filters rows based off a filter expression.

Some WHERE clauses can get quite complex, resulting in a long series of AND operators used to join sub-expressions together. Most filter for a specific row, however, searching for a specific key value.

GROUP BY Clause

The GROUP BY clause is used to collapse, or "flatten," groups of rows. Groups can be counted, averaged, or otherwise aggregated together. If you need to perform any kind of inter-row operation that requires data from more than one row, chances are you'll need a GROUP BY.

The GROUP BY clause provides a list of grouping expressions and optional collations. Very often the expressions are simple column references, but they can be arbitrary expressions. The syntax looks like this:

```
GROUP BY grouping_expression [COLLATE collation_name] [,...]
```

The grouping process has two steps. First, the GROUP BY expression list is used to arrange table rows into different groups. Once the groups are defined, the SELECT header (discussed in the next section) defines how those groups are flattened down into a single row. The resulting table will have one row for each group.

To split up the working table into groups, the list of expressions is evaluated across each row of the table. All of the rows that produce equivalent values are grouped together. An optional collation can be given with each expression. If the grouping expression involves text values, the collation is used to determine which values are equivalent. For more information on collations, see "ORDER BY Clause" on page 74.

Figure 5-5 shows how the rows are grouped together with the GROUP BY clause.

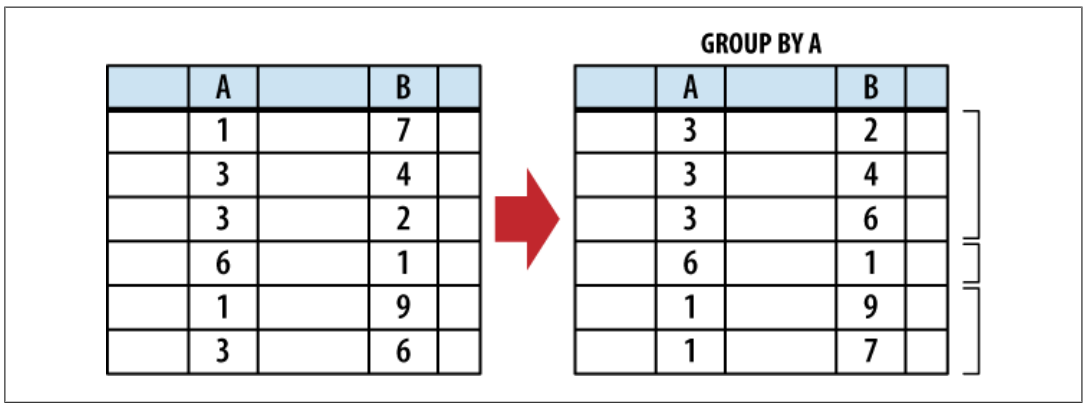


Figure 5-5. The GROUP BY clause groups rows based off a list of grouping expressions.

Once grouped together, each collection of rows is collapsed into a single row. This is typically done using aggregate functions that are defined in the SELECT heading, which is described in the next section, on page 70.

Because it is common to GROUP BY using expressions that are defined in the SELECT header, it is possible to simply reference SELECT heading expressions in the GROUP BY expression list. If a GROUP BY expression is given as a literal integer, that number is used as a column index in the result table defined by the SELECT header. Indexes start at one

with the leftmost column. A GROUP BY expression can also reference a result column alias. Result column aliases are explained in the next section.

SELECT Header

The SELECT header is used to define the format and content of the final result table. Any column you want to appear in the final results table must be defined by an expression in the SELECT header. The SELECT heading is the only required step in the SELECT command pipeline.

The format of the header is fairly simple, consisting of a list of expressions. Each expression is evaluated in the context of each row, producing the final results table. Very often the expressions are simple column references, but they can be any arbitrary expression involving column references, literal values, or SQL functions. To generate the final query result, the list of expressions is evaluated once for each row in the working table.

Additionally, you can provide a column name using the AS keyword:

```
SELECT expression [AS column_name] [,...]
```

Don't confuse the AS keyword used in the SELECT header with the one used in the FROM clause. The SELECT header uses the AS keyword to assign a column name to one of the output columns, while the FROM clauses uses the AS keyword to assign a source-table alias.

Providing an output column name is optional, but recommended. The column name assigned to a results table is not strictly defined unless the user provides an AS column alias. If your application searches for a specific column name in the query results, be sure to assign a known name using AS. Assigning a column name will also allow other parts of the SELECT statement to reference an output column by name. Steps in the SELECT pipeline that are processed before the SELECT header, such as the WHERE and GROUP BY clause, can also reference output columns by name, just as long as the column expression does not contain an aggregate function.

If there is no working table (no FROM clause), the expression list is evaluated a single time, producing a single row. This row is then used as the working table. This is useful to test and evaluate standalone expressions.

Although the SELECT header appears to filter columns from the working table, much like the WHERE clause filters rows, this isn't exactly correct. All of the columns from the original working table are still available to clauses that are processed after the SELECT header. For example, it is possible to sort the results (via ORDER BY, which is processed after the SELECT header) using a column that doesn't appear in the query output.

It would be more accurate to say that the SELECT header tags specific columns for output. Not until the whole SELECT pipeline has been processed and the results are ready to be returned, are the unused columns stripped out. Figure 5-6 illustrates this point.

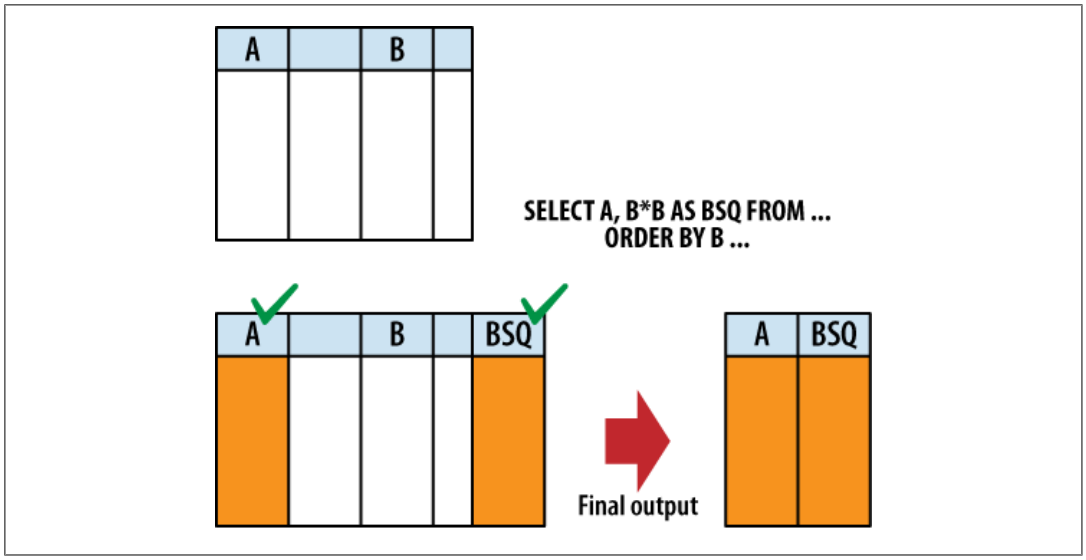


Figure 5-6. The SELECT heading tags specific columns for output. The unused columns are not removed until the query result is actually returned. Later SELECT clauses (such as ORDER BY) still have access to columns that are not part of the query result.

In addition to the standard expressions, `SELECT` supports two wildcards. A simple asterisk (`*`) will cause every user-defined column from every source table in the `FROM` clause to be output. You can also target a specific table (or table alias) using the format *`table_name`*`.*`. Although both of these wildcards are capable of returning more than one column, they can be mixed along with other expressions in the expression list. Wildcards cannot use a column alias, since they often return more than one column.

Be aware that the `SELECT` wildcards will not return any automatically generated `ROWID` columns. To return both the `ROWID` and the user-defined columns, simply ask for them both:

```
SELECT ROWID, * FROM table;
```

Wildcards do include any user-defined `INTEGER PRIMARY KEY` column that have replaced the standard `ROWID` column. See "Primary keys" on page 40 for more information about how `ROWID` and `INTEGER PRIMARY KEY` columns interact.

In addition to determining the columns of the query result, the `SELECT` header determines how row-groups (produced by the `GROUP BY` clause) are flattened into a single row. This is done using *aggregate functions*. An aggregate function takes a column expression as input and aggregates, or combines, all of the column values from the rows of a group and produces a single output value. Common aggregate functions include `count()`, `min()`, `max()`, and `avg()`. Appendix E provides a full list of all the built-in aggregate functions.

Any column or expression that is not passed through an aggregate function will assume whatever value was contained in the last row of the group. However, because SQL tables are unordered, and because the `SELECT` header is processed before the `ORDER BY`

clause, we don't really know which row is "last." This means the values for any unaggregated output will be taken from some essentially random row in the group. Figure 5-7 shows how this works.

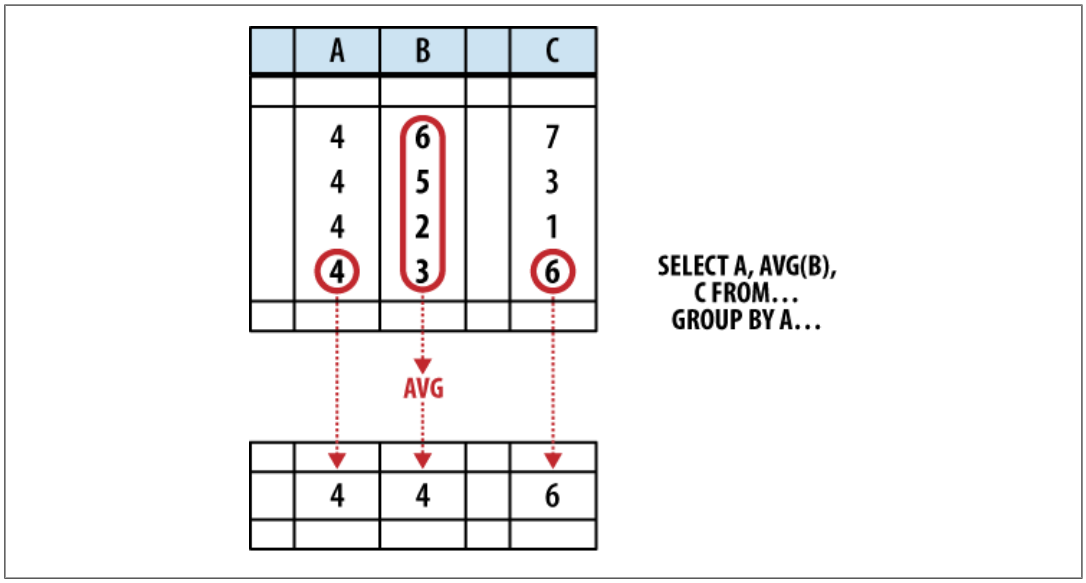


Figure 5-7. The SELECT header will flatten any row groups created by GROUP BY. This figure shows how different columns from one row-group are flattened into an output row. Any value not computed by an aggregate function comes from the last row. Because column A was used as a GROUP BY expression, all the rows are known to have the same value, and it is safe to return. Column B is run through an aggregate function, and is also safe to return. Column C is not safe to return, as the order of the rows within a group is undefined.

In some cases, picking the value from a random row is not a bad thing. For example, if a `SELECT` header expression is also used as a `GROUP BY` expression, then we know that column has an equivalent value in every row of a group. No matter which row you choose, you will always get the same value.

Where you can run into trouble is when the `SELECT` header uses column references that were not part of the `GROUP BY` clause, nor were they passed through aggregate functions. In those cases there is no deterministic way to figure out what the output value will be. To avoid this, when using a `GROUP BY` clause, `SELECT` header expressions should only use column references as aggregate function inputs, or the header expressions should match those used in the `GROUP BY` clause.

Here are some examples. In this case all of the expressions are bare column references to help make things clear:

```
SELECT col1, sum( col2 ) FROM tbl GROUP BY col1;  -- well formed
```

This is a well formed statement. The `GROUP BY` clause shows that the rows are being grouped based off the values in `col1`. That makes it safe for `col1` to appear in the `SELECT` header, since every row in a particular group will have an equivalent value in

col1. The SELECT header also references col2, but it is fed into an aggregate function. The aggregate function will take all of the col2 values from different rows in the group and produce a logical answer—in this case, a numerical summation.

The result of this statement will be two columns. The first column will have one row for each unique value from col1. Each row of the second column will have the sum of all the values in col2 that are associated with the col1 value listed in the first result column. More detailed examples can be found at the end of the chapter.

This next statement is not well formed:

```
SELECT col1, col2 FROM tbl GROUP BY col1;  -- NOT well formed
```

As before, the rows are grouped based off the value in col1, which makes it safe for col1 to appear in the SELECT header. The column col2 appears bare, however, and not as an aggregate parameter. When this statement is run, the second return column will contain random values from the original col2 column.

Although every row within a group should have an equivalent value in a column or expression that was used as a grouping key, that doesn't always mean the values are the exact same. If a collation such as NOCASE was used, different values (such as 'ABC' and 'abc') are considered equivalent. In these cases, there is no way to know the specific value that will be returned from a SELECT header. For example:

```
CREATE TABLE tbl ( t );
INSERT INTO tbl VALUES ( 'ABC' );
INSERT INTO tbl VALUES ( 'abc' );
SELECT t FROM tbl GROUP BY t COLLATE NOCASE;
```

This query will only return one row, but there is no way to know which specific value will be returned.

Finally, if the SELECT header contains an aggregate function, but the SELECT statement has no GROUP BY clause, the entire working table is treated as a single group. Since flattened groups always return one row, this will cause the query to return only one row—even if the working table contained no rows.

HAVING Clause

Functionally, the HAVING clause is identical to the WHERE clause. The HAVING clause consists of a filter expression that is evaluated for each row of the working table. Any row that evaluates to false or NULL is filtered out and removed. The resulting table will have the same number of columns, but may have fewer rows.

The main difference between the WHERE clause and the HAVING clause is where they appear in the SELECT pipeline. The HAVING clause is processed after the GROUP BY and SELECT clauses, allowing HAVING to filter rows based off the results of any GROUP BY aggregate. HAVING clauses can even have their own aggregates, allowing them to filter on aggregate results that are not part of the SELECT header.

HAVING clauses should only contain filter expressions that depend on the GROUP BY output. All other filtering should be done in the WHERE clause.

Both the HAVING and WHERE clauses can reference result column names defined in the SELECT header with the AS keyword. The main difference is that the WHERE clause can only reference expressions that do not contain aggregate functions, while the HAVING clause can reference any result column.

DISTINCT Keyword

The DISTINCT keyword will scan the result set and eliminate any duplicate rows. This ensures the returned rows constitute a proper set. Only the columns and values specified in the SELECT header are considered when determining if a row is a duplicate or not. This is one of the few cases when NULLs are considered to have "equality," and will be eliminated.

Because SELECT DISTINCT must compare every row against every other row, it is an expensive operation. In a well-designed database, it is also rarely required. Therefore, its usage is somewhat unusual.

ORDER BY Clause

The ORDER BY clause is used to sort, or order, the rows of the results table. A list of one or more sort expressions is provided. The first expression is used to sort the table. The second expression is used to sort any equivalent rows from the first sort, and so on. Each expression can be sorted in ascending or descending order.

The basic format of the ORDER BY clause looks like this:

```
ORDER BY expression [COLLATE collation_name] [ASC|DESC] [,...]
```

The expression is evaluated for each row. Very often the expression is a simple column reference, but it can be any expression. The resulting value is then compared against those values generated by other rows. If given, the named collation is used to sort the values. A collation defines a specific sorting order for text values. The ASC or DESC keywords can be used to force the sort in an ascending or descending order. By default, values are sorted in an ascending order using the default collation.

An ORDER BY expression can utilize any source column, including those that do not appear in the query result. Like GROUP BY, if an ORDER BY expression consists of a literal integer, it is assumed to be a column index. Column indexes start on the left with 1, so the phrase ORDER BY 2 will sort the results table by its second column.

Because SQLite allows different datatypes to be stored in the same column, sorting can get a bit more interesting. When a mixed-type column is sorted, NULLs will be sorted to the top. Next, integer and real values will be mixed together in proper numeric order. The numbers will be followed by text values, with BLOB values at the end. There will

be no attempt to convert types. For example, a text value holding a string representation of a number will be sorted in with the other text values, and not with the numeric values.

In the case of numeric values, the natural sort order is well defined. Text values are sorted by the active collation, while BLOB values are always sorted using the `BINARY` collation. SQLite comes with three built-in collation functions. You can also use the API to define your own collation functions. The three built-in collations are:

`BINARY`
: Text values are sorted according to the semantics of the POSIX `memcmp()` call. The encoding of a text value is *not* taken into account, essentially treating it as a large binary string. BLOB values are always sorted with this collation. This is the default collation.

`NOCASE`
: Same as `BINARY`, only ASCII uppercase characters are converted to lowercase before the comparison is done. The case-conversion is strictly done on 7-bit ASCII values. The normal SQLite distribution does not support UTF-aware collations.

`RTRIM`
: Same as `BINARY`, only trailing (righthand) whitespace is ignored.

While `ORDER BY` is extremely useful, it should only be used when it is actually needed—especially with very large result tables. Although SQLite can sometimes make use of an index to calculate the query results in order, in many cases SQLite must first calculate the entire result set and then sort it, before rows are returned. In that case, the intermediate results table must be held in memory or on disk until it is fully computed and can then be sorted.

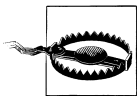
Overall, there are plenty of situations where `ORDER BY` is justified, if not required. Just be aware there can be some significant costs involved in its use, and you shouldn't get in the habit of tacking it on to every query "just because."

LIMIT and OFFSET Clauses

The `LIMIT` and `OFFSET` clauses allow you to extract a specific subset of rows from the final results table. `LIMIT` defines the maximum number of rows that will be returned, while `OFFSET` defines the number of rows to skip before returning the first row. If no `OFFSET` is provided, the `LIMIT` is applied to the top of the table. If a negative `LIMIT` is provided, the `LIMIT` is removed and will return the whole table.

There are three ways to define a `LIMIT` and `OFFSET`:

```
LIMIT limit_count
LIMIT limit_count OFFSET offset_count
LIMIT offset_count, limit_count
```



Note that if both a limit and offset are given using the third format, the order of the numbers is reversed.

Here are some examples. Notice that the `OFFSET` value defines how many rows are skipped, not the position of the first row:

```
LIMIT 10            -- returns the first 10 rows (rows 1 - 10)
LIMIT 10 OFFSET 3   -- returns rows  4 - 13
LIMIT 3  OFFSET 20  -- returns rows 21 - 23
LIMIT 3, 20         -- returns rows  4 - 23 (different from above!)
```

Although it is not strictly required, you usually want to define an `ORDER BY` if you're using a `LIMIT`. Without an `ORDER BY`, there is no well-defined order to the result, making the limit and offset somewhat meaningless.

Advanced Techniques

Beyond the basic `SELECT` syntax, there are a few advanced techniques for expressing more complex queries.

Subqueries

The `SELECT` command provides a great deal of flexibility, but there are times when a single `SELECT` command cannot fully express a query. To help with these situations, SQL supports *subqueries*. A subquery is nothing more than a `SELECT` statement that is embedded in another `SELECT` statement. Subqueries are also known as sub-selects.

Subqueries are most commonly found in the `FROM` clause, where they act as a computed source table. This type of subquery can return any number of rows or columns, and is similar to creating a view or running the query, recording the results into a temporary table, and then referencing that table in the main query. The main advantage of using an in-line subquery is that the query optimizer is able to merge the subquery into the main `SELECT` statement and look at the whole problem, often leading to a more efficient query plan.

To use a subquery in the FROM clause, simply enclose it in parentheses. The following two statements will produce the same output:

```
SELECT * FROM TblA AS a JOIN TblB AS b;
SELECT * FROM TblA AS a JOIN (SELECT * FROM TblB) AS b;
```

Subqueries can show up in other places, including general expressions used in any SQL command. The `EXISTS` and `IN` operators both utilize subqueries. In fact, you can use a subquery any place an expression expects a list of literal values (a subquery cannot be used to generate a list of identifiers, however). See Appendix D for more details on SQL expressions.

Compound SELECT Statements

In addition to subqueries, multiple `SELECT` statements can be combined together to form a *compound `SELECT`*. Compound `SELECT`s use set operators on the rows generated by a series of `SELECT` statements.

In order to combine correctly, each `SELECT` statement must generate the same number of columns. The column names from the first `SELECT` statement will be used for the overall result. Only the last `SELECT` statement can have an `ORDER BY`, `LIMIT` or `OFFSET` clause, which get applied to the full compound result table. The syntax for a compound `SELECT` looks like this:

```
SELECT ... FROM ... WHERE ... GROUP BY ... HAVING ...

compound_operator

SELECT ... FROM ... WHERE ... GROUP BY ... HAVING ...

[...]

ORDER BY ... LIMIT ... OFFSET ...
```

Multiple compound operators can be used to include additional `SELECT` statements.

`UNION ALL`
: The `UNION ALL` operator concatenates all of the rows returned by each `SELECT` into one large table. If the two `SELECT` blocks generate *N* and *M* rows, respectively, the resulting table will have *N*+*M* rows.

`UNION`
: The `UNION` operator is very similar to the `UNION ALL` operator, but it will eliminate any duplicate rows, including duplicates that came from the same `SELECT` block. If the two `SELECT` blocks generate *N* and *M* rows, respectively, the resulting table can have anywhere from 1 to *N*+*M* rows.

`INTERSECT`
: The `INTERSECT` operator will return one instance of any row that is found (one or more times) in both `SELECT` blocks. If the two `SELECT` blocks generate *N* and *M* rows, respectively, the resulting table can have anywhere from 0 to `MIN(`*N*,*M*`)` rows.

`EXCEPT`
: The `EXCEPT` operator will return all of the rows in the first `SELECT` block that are *not* found in the second `SELECT` block. It is essentially a subtraction operator. If there are duplicate rows in the first block, they will all be eliminated by a single, matching row in the second block. If the two `SELECT` blocks generate *N* and *M* rows, respectively, the resulting table can have anywhere from 0 to *N* rows.

SQLite supports the `UNION`, `UNION ALL`, `INTERSECT`, and `EXCEPT` compound operators. Figure 5-8 shows the result of each operator.

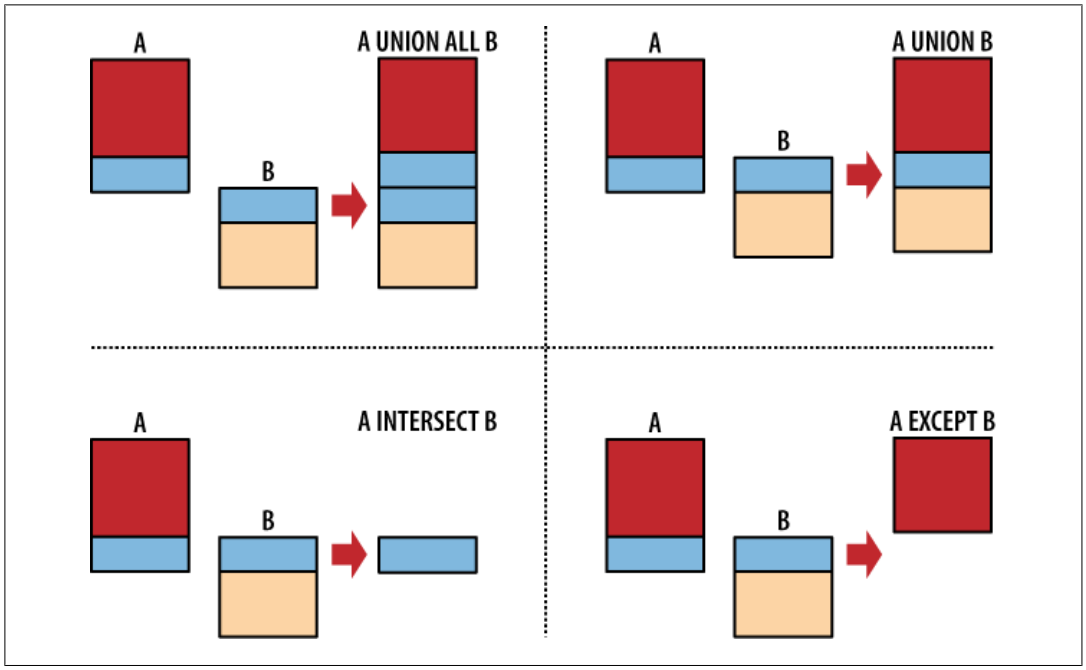


Figure 5-8. The compound operators UNION ALL, UNION, INTERSECT and EXCEPT.

Once all the compound operators have been combined, any trailing `ORDER BY`, `LIMIT`, and `OFFSET` is applied to the final result table. In the case of compound `SELECT` statements, the expressions present in any `ORDER BY` clause must be exactly match one of the result columns, or use a column index.

Alternate JOIN Notation

There are two styles of join notation. The style shown earlier in this chapter is known as *explicit join notation*. It is named such because it uses the keyword `JOIN` to explicitly describe how each table is joined to the next. The explicit join notation is also known as *ANSI join notation*, as it was introduced when SQL went through the standardization process.

The older, original join notation is known as *implicit join notation*. Using this notation, the `FROM` clause is simply a comma-separated list of tables. The tables in the list are combined using a Cartesian product and the relevant rows are extracted with additional `WHERE` conditions. In effect, it degrades every join to a `CROSS JOIN` and then moves the join conditions out of the `FROM` clause and into the `WHERE` clause.

This first statement uses the explicit join notation we learned earlier in the chapter:

```
SELECT ...
   FROM employee JOIN resource ON ( employee.eid = resource.eid )
   WHERE ...
```

This is the same statement written with the implicit join notation:

```
SELECT ...
   FROM employee, resource
   WHERE employee.eid = resource.eid AND ...
```

There is no performance difference between the two notations: it is purely a matter of syntax.

In general, the explicit notion (the first one) has become the standard way of doing things. Most people find the explicit notation easier to read, making the intent of the query more transparent and easier to understand. I've always felt the explicit notation is a bit cleaner, as it puts the complete join specification into the `FROM` clause, leaving the `WHERE` clause free for query-specific filters. Using the explicit notation, the `FROM` clause (and the `FROM` clause alone) fully and independently specifies what you're selecting "from."

The explicit notation also lets you be much more specific about the type and order of each `JOIN`. In SQLite, you must use the explicit notation if you want an `OUTER JOIN`—the implicit notation can only be used to indicate a `CROSS JOIN` or `INNER JOIN`.

If you're learning SQL for the first time, I would strongly suggest you become comfortable with the explicit notation. Just be aware that there is a great deal of SQL code out there (including older books and tutorials) using the older, implicit notation.

SELECT Examples

The `SELECT` command is very complex, and it can be difficult to see how these different clauses can be fit together into something useful. Some of this will become more obvious in the next chapter, when we look at standard database design practices, but to get you started, we're going to look at several examples.

All of these examples will use this data:

```
CREATE TABLE x ( a, b );
INSERT INTO x VALUES ( 1, 'Alice' );
INSERT INTO x VALUES ( 2, 'Bob' );
INSERT INTO x VALUES ( 3, 'Charlie' );

CREATE TABLE y ( c, d );
INSERT INTO y VALUES ( 1, 3.14159 );
INSERT INTO y VALUES ( 1, 2.71828 );
INSERT INTO y VALUES ( 2, 1.61803 );

CREATE TABLE z ( a, e );
INSERT INTO z VALUES ( 1, 100 );
INSERT INTO z VALUES ( 1, 150 );
INSERT INTO z VALUES ( 3, 300 );
INSERT INTO z VALUES ( 9, 900 );
```

These examples show the `sqlite3` command-line tool. The following dot-commands were issued to make the output easier to understand. The last command will cause `sqlite3` to print the string `[NULL]` whenever a NULL is encountered. Normally, a NULL will produce a blank output that is indistinguishable from an empty string:

```
.headers on
.mode column
.nullvalue [NULL]
```

This dataset is available on the book's download page on the O'Reilly website, as both an SQL file and an SQLite database. I suggest you sit down with a copy of `sqlite3` and try these commands out. Try experimenting with different variations.

If one of these examples doesn't quite make sense, just break the `SELECT` statement down into its individual parts and step through them bit by bit.

Simple SELECTs

Let's start with a simple select that returns all of the columns and rows in table `x`. The `SELECT *` syntax returns all columns by default:

```
sqlite> SELECT * FROM x;

a           b
----------  ----------
1           Alice
2           Bob
3           Charlie
```

We can also return expressions, rather than just columns:

```
sqlite> SELECT d, d*d AS dSquared FROM y;

d           dSquared
----------  ------------
3.14159     9.8695877281
2.71828     7.3890461584
1.61803     2.6180210809
```

Simple JOINs

Now some joins. By default, the bare keyword `JOIN` indicates an `INNER JOIN`. However, when no additional condition is put on the `JOIN`, it reverts to a `CROSS JOIN`. As a result, all three of these queries produce the same results. The last line uses the implicit join notation.

```
sqlite> SELECT * FROM x JOIN y;
sqlite> SELECT * FROM x CROSS JOIN y;
sqlite> SELECT * FROM x, y;

a           b           c           d
----------  ----------  ----------  ----------
1           Alice       1           3.14159
1           Alice       1           2.71828
1           Alice       2           1.61803
2           Bob         1           3.14159
2           Bob         1           2.71828
2           Bob         2           1.61803
3           Charlie     1           3.14159
3           Charlie     1           2.71828
3           Charlie     2           1.61803
```

In the case of a cross join, every row in table `a` is matched to every row in table `y`. Since both tables had three rows and two columns, the result set has nine rows (3·3) and four columns (2+2).

JOIN...ON

Next, a fairly simple inner join using a basic `ON` join condition:

```
sqlite> SELECT * FROM x JOIN y ON a = c;

a           b           c           d
----------  ----------  ----------  ----------
1           Alice       1           3.14159
1           Alice       1           2.71828
2           Bob         2           1.61803
```

This query still generates four columns, but only those rows that fulfill the join condition are included in the result set.

The following statement requires the columns to be qualified, since both table `x` and table `z` have an `a` column. Notice that two different `a` columns are returned, one from each source table:

```
sqlite> SELECT * FROM x JOIN z ON x.a = z.a;

a           b           a           e
----------  ----------  ----------  ----------
1           Alice       1           100
1           Alice       1           150
3           Charlie     3           300
```

JOIN...USING, NATURAL JOIN

If we use a `NATURAL JOIN` or the `USING` syntax, the duplicate column will be eliminated. Since both table `x` and table `z` have only column `a` in common, both of these statements produce the same output:

```
sqlite> SELECT * FROM x JOIN z USING ( a );
sqlite> SELECT * FROM x NATURAL JOIN z;

a           b           e
----------  ----------  ----------
1           Alice       100
1           Alice       150
3           Charlie     300
```

OUTER JOIN

A `LEFT OUTER JOIN` will return the same results as an `INNER JOIN`, but will also include rows from table `x` (the left/first table) that were not matched:

```
sqlite> SELECT * FROM x LEFT OUTER JOIN z USING ( a );

a           b           e
----------  ----------  ----------
1           Alice       100
1           Alice       150
2           Bob         [NULL]
3           Charlie     300
```

In this case, the `Bob` row from table `x` has no matching row in table `z`. Those column values normally provided by table `z` are padded out with NULL, and the row is then included in the result set.

Compound JOIN

It is also possible to `JOIN` multiple tables together. In this case we join table `x` to table `y`, and then join the result to table `z`:

```
sqlite> SELECT * FROM x JOIN y ON x.a = y.c LEFT OUTER JOIN z ON y.c = z.a;

a           b           c           d           a           e
----------  ----------  ----------  ----------  ----------  ----------
1           Alice       1           3.14159     1           100
1           Alice       1           3.14159     1           150
1           Alice       1           2.71828     1           100
1           Alice       1           2.71828     1           150
2           Bob         2           1.61803     [NULL]      [NULL]
```

If you don't see what is going on here, work through the joins one at a time. First look at what `FROM x JOIN y ON x.a = y.c` will produce (shown in one of the previous examples). Then look at how this result set would combine with table `z` using a `LEFT OUTER JOIN`.

Self JOIN

Our last join example shows a self-join, where a table is joined against itself. This creates two unique instances of the same table and necessitates the use of table aliases:

```
sqlite> SELECT * FROM x AS x1 JOIN x AS x2 ON x1.a + 1 = x2.a;

a           b           a           b
----------  ----------  ----------  ----------
1           Alice       2           Bob
2           Bob         3           Charlie
```

Also, notice that the join condition is a more arbitrary expression, rather than being a simple test of column references.

WHERE Examples

Moving on, the `WHERE` clause is used to filter rows. We can pick out a specific row:

```
sqlite> SELECT * FROM x WHERE b = 'Alice';

a           b
----------  ----------
1           Alice
```

Or a range of values:

```
sqlite> SELECT * FROM y WHERE d BETWEEN 1.0 AND 3.0;

c           d
----------  ----------
1           2.71828
2           1.61803
```

In this case, the `WHERE` expression references the output column by its assigned name:

```
sqlite> SELECT c, d, c+d AS sum FROM y WHERE sum < 4.0;

c           d           sum
----------  ----------  ----------
1           2.71828     3.71828
2           1.61803     3.61803
```

GROUP BY Examples

Now let's look at a few `GROUP BY` statements. Here we group table `z` by the `a` column. Since there are three unique values in `z.a`, the output has three rows. Only the grouping `a=1` has more than one row, however. We can see this in the `count()` values returned by the second column:

```
sqlite> SELECT a, count(a) AS count FROM z GROUP BY a;

a           count
----------  ----------
1           2
3           1
9           1
```

This is a similar query, only now the second output column represents the sum of all the `z.e` values in each group:

```
sqlite> SELECT a, sum(e) AS total FROM z GROUP BY a;

a           total
----------  ----------
1           250
3           300
9           900
```

We can even compute our own average and compare that to the `avg()` aggregate:

```
sqlite> SELECT a, sum(e), count(e),
   ...>    sum(e)/count(e) AS expr, avg(e) AS agg
   ...>   FROM z GROUP BY a;

a           sum(e)      count(e)    expr        agg
----------  ----------  ----------  ----------  ----------
1           250         2           125         125.0
3           300         1           300         300.0
9           900         1           900         900.0
```

A `HAVING` clause can be used to filter rows based off the results of the `sum()` aggregation:

```
sqlite> SELECT a, sum(e) AS total FROM z GROUP BY a HAVING total > 500;

a           total
----------  ----------
9           900
```

ORDER BY Examples

The output can also be sorted. Most of these examples already look sorted, but that's mostly by chance. The `ORDER BY` clause enforced a specific order:

```
sqlite> SELECT * FROM y ORDER BY d;

c           d
----------  ----------
2           1.61803
1           2.71828
1           3.14159
```

Limits and offsets can also be applied to pick out specific rows from an ordered result. Conceptually, these are fairly simple, however.

These tables and queries are available as part of the book download. Feel free to load the data into `sqlite3` and try out different queries. Don't worry about creating `SELECT` statements that use every available clause. Start with simple queries where you can understand all the steps, then start to combine clauses to build larger and more complex queries.

What's Next

Now that we've had a much closer look at the `SELECT` command, you've seen most of the core SQL language. The next chapter will take a more detailed look at database design. This will help you lay out your tables and data in a way that reinforces the strengths of the database system. Understanding some of the core design ideas should also make it more clear why `SELECT` works the way it does.