Query Design, Optimisation

Atomic queries *Molecular queries* *Query formatting* *Relational algebra* *JOIN* *WHERE* *GROUP BY* *Many-many queries* *Common query problems* *Simplify query logic* *Optimising queries* *Using EXPLAIN* *Optimising caches* *Advanced optimisations* *Query Profiler* *Optimising updates* *Optimising output* *SQL injection*

The "Q" in SQL doesn't refer just to retrieval of information from a database; it refers to any statement that retrieves *or modifies* either data or data objects like tables and schemas, or database metadata *e.g.*, from `information_schema`, so we can speak of DDL queries, INSERT queries, UPDATE queries, DELETE queries and so on.

In this chapter the main focus is on SELECT queries. If you've not yet read the *relational basics chapter*, and the Data Manipulation Language Commands section in the *syntax chapter*, especially the subsection on SELECT, now's the time!

Query design is the art of using Structured Query Language to build correct, efficient database questions. It begins with correct design of the database, which must accommodate all required queries as gracefully and efficiently as possible (*Chapter 5*).. Then given a particular retrieval or update problem, you map the logic of the problem to the database, translate that mapping to SQL, test the SQL for correctness, then test and optimise the query for performance.

Query design

SELECT queries can use JOIN, WHERE and HAVING clauses to scope results to specific rows and columns, GROUP BY to combine result rows into analytic summaries, and UNION to combine multiple query results. INSERT, DELETE and UPDATE commands may reference JOINs. INSERT … SELECT (*Chapter 6*, INSERT) inserts a query result into another table. DELETEs and UPDATEs may be scoped by WHERE clauses.

Central to queries and their design are software patterns we call *Atomic Queries* and *Molecular Queries.*

Atomic queries and the SELECT list

An *atomic query* can't be reduced to simpler ones. It performs one of the four basic DML commands—`SELECT, INSERT, DELETE` or `UPDATE`—on one table, or on no table, since the simplest possible SELECT query has nothing more than a SELECT list:

```
SELECT VERSION();   -- returns something like 6.0.10-alpha-community-log
```

ISO SQL permits queries to return scalar values, a row, or a table. If you ask MySQL for a constructed row, however, it will complain:

```
SELECT ( VERSION(), CURDATE() );
ERROR 1241 (21000): Operand should contain 1 column(s)
```

Remove the parentheses and the query runs, because a MySQL SELECT list may specify a scalar value, multiple scalars forming an implicit but not explicit row, or a table.

A common practical example of an atomic query is a SELECT statement that populates a user picklist. Suppose a user interface for the tracker database (*Chapter 5*) needs to present a dropdown list of professions rows. We could use this query:

```
SELECT professionID, name, description
FROM professions
ORDER BY name;
```

The problem we're solving may need more clauses, *e.g*, a WHERE clause to scope retrieval, a LIMIT clause to curb the retrieval rowcount. For clarity, professionals commonly place each modifying clause (*e.g.*,FROM, WHERE) on a new line. We may combine keywords and clauses to suit requirements:

```
SELECT name, companyname
FROM parties
WHERE partyID = 123;
SELECT name, companyname
FROM parties
ORDER BY companyname, name;
SELECT taskID, invoiceID, amount
FROM invoiceitems
WHERE amount > 1000
ORDER BY amount LIMIT 10;
```

Generally, for each table in its database, the usual application will require four atomic SQL statements, at a minimum (Table 9-1).

Table 9-1: Four Basic Atomic SQL Statements

SELECT	SELECT primaryKey, (columnlist) FROM tableName
INSERT	INSERT tableName (columnlist) VALUES (valuelist)
UPDATE	UPDATE tableName SET column = value WHERE PrimaryKey = keyValue
DELETE	DELETE FROM tableName WHERE PrimaryKey = keyValue

INSERT and UPDATE statements act on one table at a time, and are often atomic. Of course we may SELECT from multiple tables using JOINs and subqueries. The MySQL version of DELETE offers three syntaxes; two of these support deletion in multiple tables, so SELECT and DELETE statements will often be molecular.

Molecular queries

A molecule combines atoms into a cohesive new structure with emergent properties. A *molecular query* combines atomic queries into a cohesive information request, usually via one or more JOINs and WHERE restrictions, to answer a compound question. For example if we want a list of all parties and the names of cities where they have addresses, and if we wish the list to include parties for whom no addresses are recorded, we have to retrieve information from two tables, and coordinate it into one list. We do it with:

```
SELECT name, city
FROM parties
LEFT JOIN addresses USING (partyID)
ORDER BY name;
```

which says in English: list all parties whether we have an address for them or not; if there is an address, show the city part of it. This molecular query pattern is an `OUTER JOIN`.

Most working queries are molecular. They have complexities in their SELECT lists, FROM clauses, and in JOIN, WHERE, GROUP BY and HAVING clauses.

Molecular queries and relational algebra

Table 9-2 shows how basic operations of relational algebra map to MySQL queries.

Projection SELECTs columns for the query result. *Product*, *restriction*, *partition* and *division* select rows for the query result: product via JOIN, restriction by WHERE, partition by GROUP BY, division via nested NOT EXISTS clauses. *Recursion* iterates over resultsets.

The *Cartesian product* or CROSS JOIN of tables A and B *multiplies* tables A and B to produce table AxB, which has all possible row-by-column combinations from A and B:

```
+-----+     +-----+      +-----------+
|  A  |     |  B  |      | table AxB |
+-----+     +-----+      +-----+-----+
|key_a|     |key_b|      |key_a|key_b|
+-----+     +-----+      +-----+-----+
|  2  |     |  1  |      |  2  |  1  |
|  4  |     |  7  |      |  2  |  7  |
+-----+     |  3  |      |  2  |  3  |
            +-----+      |  4  |  1  |
                         |  4  |  7  |
                         |  4  |  3  |
                         +-----+-----+
```

Division inverts multiplication, so when we divide dividend table AxB by divisor table B,

- *quotient table columns* are those in the dividend that are *not* in the divisor, and
- *quotient table rows* are the sets of quotient column values which occur in the dividend for every divisor row.

Table 9-2: Relational Operations and MySQL Query Components

Operation	***Operand***	***Definition***	***SQL statement component***
Projection	columns	Select specific columns to include in the result table	`SELECT …, …,…`
Product	rows	Return a result table comprising all combinations of all rows of joined tables, or a subset of that product	`FROM, JOIN`
Restriction	rows	Restrict row selection for the result table on some condition	`JOIN, WHERE, HAVING`
Partition	rows	Subdivide result on value ranges	`GROUP BY`
Division	rows	The inverse of product: divide a *dividend* table by a *divisor* table to produce a *quotient* or results table.	Nested `SELECT… WHERE NOT EXISTS`
Union	tables	Combine rows from 2 identically structured tables without duplication	`UNION`
Intersection	tables	Combine rows from identically structured tables	*Not implemented*
Difference	tables	List rows from 1 of 2 identically structured tables	*Not implemented*
Recursion	resultsets	Recursively retrieve from a hierarchy or network	`WITH`

Thus to manually divide table AxB by table B, list every key_a value that is paired with all key_b values in AxB, then drop all duplications in that list, then drop any key_b values from every row.

SQL has no universal quantifier, so what is the SQL instruction for this? Well, ALL X IS Y means exactly THERE IS NO X THAT IS NOT Y. SQL has a NOT EXISTS() operator, so we write relational division in SQL as a doubly nested NOT EXISTS query:

```
SELECT DISTINCT key_a
FROM AxB WHERE NOT EXISTS (
  SELECT * FROM AxB AS x1
  WHERE NOT EXISTS( SELECT * FROM AxB AS x2 WHERE x1.key_a=x2.key_a )
);
```

Union, *intersection* and *difference* combine two or more identically structured tables produced by queries using the five other relational operations. A union ORs the two query results, removing duplicates. Intersection ANDs them. Difference XORs them. MySQL supports UNION. The absence of intersection and difference need not be a problem. For example if you need a query like

```
SELECT id2 FROM tbl WHERE id1 = 1
INTERSECT
SELECT id2 FROM tbl WHERE id2 = 2;
```

since MySQL 5 you can write one of these:

```
(SELECT ID2 FROM tbl WHERE id1 = 1)
UNION DISTINCT
(SELECT id2 FROM tbl WHERE id1 = 2);
```

```
SELECT a.id2
FROM tbl AS a , tbl AS b
WHERE a.id_2=b.id2 AND a.id1 = 1 and b.id1 = 2;
```

Subqueries

A *subquery* or *inner query* is a parenthesised SELECT nested in another query. It may be …

- a *table subquery* returning a *derived table*, or
- a subquery returning one scalar value, or an implicit row of them, or
- an *expression subquery* returning a *scalar value*.

It is *uncorrelated* if it can run without its outer query. It is *correlated* if it refers to an entity in its outer query, in which case it can't run by itself, and may appear …

- in a SELECT list, where it must return a scalar value;
- in a FROM clause, where it must be a table subquery and where it serves the same purpose as the name of a base table or View;
- as an argument for IN() or EXISTS() in a WHERE or HAVING clause, where it is a *quantified predicate subquery* returning a list or row of 0 or more values;
- as an argument for a *comparison operator* in a WHERE or HAVING clause.

Practical Molecular Queries

Query formatting

The easier a query is to read, the more likely it will be correct, all other things equal, and the more likely it will be correctly understood. In SQL, as in other coding languages, there may be no one formatting style that is demonstrably best for all occasions. But you are asking for trouble if you format queries inconsistently, sloppily or ad lib. We prefer:

- SQL keywords in capital letters,
- consistent indentation,

- in complex queries start each component (FROM, JOIN, WHERE etc) on a new line,
- in complex queries use aliases freely,
- subqueries clearly set off from other clauses by indentation and parentheses, as in:

```
 SELECT DISTINCT party FROM parties p
WHERE NOT EXISTS (
  SELECT * FROM districts d
  WHERE NOT EXISTS (
    SELECT * FROM candidates c
    WHERE c.party=p.party AND c.district=d.district
  )
);
```

Use parentheses freely to disambiguate meaning, even if the query counts on precedence rules; MySQL may add even more..

The FROM clause

For FROM syntax see SELECT | FROM and JOINS in *Chapter 6*. For clarity and debugging, *explicit JOIN syntax* is, for writing or debugging or maintaing a query, far superior to *comma JOIN syntax*.

[CROSS] JOIN, COMMA JOIN, STRAIGHT_JOIN

The simple queries

```
SELECT * FROM parties, addresses;
SELECT * FROM parties [CROSS] JOIN addresses;
```

> ***Tip**: Use* **CROSS JOIN** *only when the result is to include all combinations of all rows from joined tables.*

are equivalent, producing identical lists of the tables' *Cartesian product*—all possible combinations of all rows.

STRAIGHT_JOIN, with no other JOIN spec, is alternative MySQL syntax for CROSS JOIN, With a USING or ON clause, STRAIGHT_JOIN processes tables in the order they are named.

What's the point of a CROSS JOIN? Suppose you have a billiards league from everyone listed in parties, all of whom are to visit each other exactly once for a game in the next calendar year. CROSS JOIN the parties table to itself, and exclude self-self matches …

```
SELECT p1.partyID AS Visiting, p2.partyId AS Home
FROM parties AS p1
CROSS JOIN parties AS p2
WHERE p1.partyID <> p2.partyID;
+----------+------+
| Visiting | Home |
+----------+------+
|        2 |    1 |
|        3 |    1 |
|        1 |    2 |
|        3 |    2 |
|        1 |    3 |
|        2 |    3 |
+----------+------+
```

and there's your list of required matches. All that remains is to add schedule dates.

We usually want answers to more specific questions. Since the database is normalised, data must be combined from multiple tables, specific rows and columns must be selected, and other specific rows and columns must be excluded. The combining is done in FROM clause to specify a LEFT [OUTER], RIGHT [OUTER], or INNER JOIN, and the restrictions go into the WHERE clause, perhaps also in a HAVING clause.

OUTER JOIN

An `OUTER JOIN` returns all rows (matching the `WHERE` clause if any) on the named (`LEFT` or `RIGHT`) side of the `JOIN`, and for each of those rows shows, from the other side of the `JOIN`, matching values where such matches exist, and `NULL`s where they do not. MySQL implements `LEFT JOIN` and `RIGHT JOIN`, not `FULL OUTER JOIN`.

> ***Tip:*** *An* **OUTER JOIN** *is* **LEFT** *or* **RIGHT**. *Use it when you need to list matched rows from the named (left or right) side, and both matched and unmatched rows from the other side.*

In the sample `tracker` database, every `contractor_ client` table row defines a contractor-client relationship between two parties. by pairing two `partyID` values:

- a `contractorpartyID` value that matches the *contractor*'s `parties.partyID`,
- a `clientpartyID` matching the *client*'s `parties.partyID`.

Assume the smallest possible dataset capable of illustrating the point—three rows in the `parties` table, two of them referenced in the one and only `contractor_client` row:

```
SELECT partyID
FROM parties;
+---------+
| partyID |
+---------+
|       1 |
|       2 |
|       3 |
+---------+
```

```
SELECT contractorpartyID, clientpartyID
FROM contractor_client;
+-------------------+---------------+
| contractorpartyID | clientpartyID |
+-------------------+---------------+
|                 2 |             1 |
+-------------------+---------------+
```

Now suppose we want the list of all `partyID`s showing whether they are contractors, and if they are, then with which clients. Given our tiny dataset, the answer should be:

```
+---------+-------------------+---------------+
| partyID | contractorpartyID | clientpartyID |
+---------+-------------------+---------------+
|       1 |                   |               |
|       2 |                 2 |             1 |
|       3 |                   |               |
+---------+-------------------+---------------+
```

Begin with:

```
SELECT partyID, contractorpartyID, clientpartyID
FROM parties
LEFT JOIN contractor_client ON partyID=contractorpartyID
ORDER BY partyID;
```

which says, in English:

- List every `parties.partyID`; for each one of these…
- list all `contractor_client` rows with a matching `contractorpartyID`
- if there is no match, show `contractorpartyID` and `clientpartyID` as `NULL`.

There are three rows in `parties`, only one of which has a `contractorpartyID` match in the `contractor_client` table, and that what our query returns:

```
+---------+-------------------+---------------+
| partyID | contractorpartyID | clientpartyID |
+---------+-------------------+---------------+
|       1 |              NULL |          NULL |
|       2 |                 2 |             1 |
|       3 |              NULL |          NULL |
+---------+-------------------+---------------+
```

But the `NULL`s are unsightly. To return blank cells instead of `NULL`s for data that isn't there, use *`IFNULL()`* to transform `NULL`s to blanks:

```
SELECT partyID,IFNULL(contractorpartyID,''),IFNULL(clientpartyID,'')
FROM parties
LEFT JOIN contractor_client ON partyID=contractorpartyID
ORDER BY partyID;
+---------+-------------------+---------------+
| partyID | contractorpartyID | clientpartyID |
+---------+-------------------+---------------+
|       1 |                   |               |
|       2 |                 2 |             1 |
|       3 |                   |               |
+---------+-------------------+---------------+
```

A `RIGHT JOIN` is the exact right-left mirror image of a `LEFT JOIN`. To rewrite a LEFT JOIN as a RIGHT JOIN, reverse the table names and change `LEFT` to `RIGHT`:

```
SELECT partyID, contractorpartyID, clientpartyID
FROM contractor_client
RIGHT JOIN parties ON partyID=contractorpartyID
ORDER BY partyID;
```

Use a `LEFT JOIN` or `RIGHT JOIN` when you need not just the matches from the two tables, but all rows, *matching and not*, from the unnamed side of the join.

INNER JOIN

An `INNER JOIN` lists only matching rows from *each side* of the `JOIN`, suppressing all non-matches.

Use an **INNER JOIN** *when the query result is to show only matching rows from both sides of the* JOIN.

To list all contractors and their clients, we need to look up two names for each contractor-client pair. So *two* joins:

```
SELECT
  clientpartyID,Cli.name AS Client,contractorpartyID,Con.name AS Contractor
FROM contractor_client AS cc
INNER JOIN parties AS Cli ON cc.clientpartyID = Cli.partyID
INNER JOIN parties AS Con ON cc.contractorpartyID = Con.partyID;
+---------------+---------------+-------------------+---------------+
| clientpartyID | Client        | contractorpartyID | Contractor    |
+---------------+---------------+-------------------+---------------+
|             1 | Arthur Fuller |                 2 | Peter Brawley |
+---------------+---------------+-------------------+---------------+
```

If it seems counter-intuitive to you to write a statement that uses an alias before the alias is defined, remember that the query engine processes the FROM clause before it processes the SELECT list.

See *Multiple Joins* below for more on compound joins.

NATURAL JOIN

A NATURAL JOIN is an OUTER JOIN on columns with identical names and types, where you tell the SQL engine to find which columns join the tables. NATURAL JOIN should fail with an error when the two joined tables have no columns of the same name and type.

Multiple JOINs

We already saw a *double INNER JOIN*. With the `parent-child-grandchild` tables of *Chapter 6* (Example 6-1) cascading `LEFT JOIN`s from `parent` to `child` to `grandchild` return all `parent.id`s having `child` and `grandchild` rows as follows:

```
SELECT parent.id AS ParentID,
       IFNULL(child.parent_id,'') AS ChildParentID,
       IFNULL(child.id,'') AS ChildID,
       IFNULL(grandchild.child_id,'') AS GrandchildChildID
FROM parent
```

```
LEFT JOIN child ON parent.id=child.parent_id
LEFT JOIN grandchild ON child.id=grandchild.child_id;
+----------+---------------+---------+-------------------+
| ParentID | ChildParentID | ChildID | GrandchildChildID |
+----------+---------------+---------+-------------------+
|        0 |               |         |                   |
|        1 | 1             | 1       | 1                 |
|        2 | 2             | 2       |                   |
|        2 | 2             | 3       |                   |
|        3 |               |         |                   |
+----------+---------------+---------+-------------------+
```

Since 5.0.12, MySQL parses …FROM a,b JOIN c… as …FROM a,(b JOIN c)…, as the SQL standard specifies. To get …FROM (a,b) JOIN c…., you must use parentheses. It is *much* better, however, to use explicit JOIN syntax, which perfectly disambiguates any series of INNER and OUTER JOINs:

```
SELECT parent.id AS ParentID,
       IFNULL(child.parent_id,'') AS ChildParentID,
       IFNULL(child.id,'') AS ChildID,
       IFNULL(grandchild.child_id,'') AS GrandchildChildID
FROM parent
INNER JOIN child ON parent.id=child.parent_id
LEFT JOIN grandchild ON child.id=grandchild.child_id;
+----------+---------------+---------+-------------------+
| ParentID | ChildParentID | ChildID | GrandchildChildID |
+----------+---------------+---------+-------------------+
|        1 | 1             | 1       | 1                 |
|        2 | 2             | 2       |                   |
|        2 | 2             | 3       |                   |
+----------+---------------+---------+-------------------+
```

Making the first join INNER instead of OUTER removed childless `parent` rows.

Negative (Exclusion) JOINs

In a LEFT JOIN result, missing rightsided matches show as NULLs. To show *only* those pairings, add a WHERE clause that includes only those rightsided NULLs. For example, which `parent` rows have no `child` rows?

```
SELECT parent.id
FROM parent LEFT JOIN child ON parent.id = child.parent_id
WHERE child.parent_id IS NULL;
+----+
| id |
+----+
|  0 |
|  3 |
+----+
```

The above query is faster than, but logically equivalent to …

```
SELECT parent.id AS ParentID
FROM parent
WHERE NOT EXISTS (
  SELECT parent.id
  FROM parent INNER JOIN child ON parentID = child.parent_id
);
```

Subqueries in the FROM clause

Since version 4.1, a subquery can be used in the FROM clause where a table reference is expected, if the subquery has an alias:

```
SELECT COUNT(*) FROM (
  SELECT parent.id AS ParentID
  FROM parent
  WHERE NOT EXISTS (
    SELECT parent.id
    FROM parent INNER JOIN child ON parentID = child.parent_id
  )
) AS childless;
+----------+
| COUNT(*) |
```

```
+----------+
|        2 |
+----------+
```

A subquery in the FROM clause may vastly outperform an IN() subquery in the WHERE clause. For example this query from an order-entry system (*Chapter 11*) …

```
SELECT o.orderID, discount
FROM orderdetails AS o
INNER JOIN (
  SELECT orderID
  FROM orderdetails
  GROUP BY orderID
  HAVING COUNT(1) > 1
) AS t ON o.orderID=t.orderID;
```

is, in MySQL 5.0 through 5.5, *50 times faster* than this logically equivalent query…

```
SELECT orderID, discount
FROM orderdetails
WHERE orderID IN (
  SELECT orderID FROM orderdetails
  GROUP BY orderID
  HAVING COUNT(orderID)>1
);
```

Especially since 5.7, the query engine has learnt optimisations that reduce such performance difference, but significant gaps remain.

The WHERE clause

The WHERE clause can combine *relational product* (JOIN) and *relational restriction*. Indeed JOIN syntax is a bit of a latecomer to SQL—originally, there was only WHERE. And when MySQL processes a query, it moves JOIN clauses to the WHERE clause.

So why not just combine them in the WHERE clause to start with, and forget the JOIN clause altogether? The main reason is that *product* and *restriction* are logically different, so conflating them can impede both good query design and robust query maintenance. It is usually clearer and practically preferable to formulate *product* operations in the JOIN clause, and *restriction* operations in the WHERE clause.

The basic job of the WHERE clause is to restrict the rows that are to be retrieved. This is usually accomplished by a sequence of comparison clauses, with or without subqueries:

```
… WHERE col1 <= col2 AND …
… WHERE NOT EXISTS( SELECT … ) …
… WHERE colname [NOT] IN( SELECT …) …
```

You can use any MySQL *comparison operator*, and you can link comparisons with any MySQL *logical operator*.

You are designing for two objectives: correctness and speed. Though it is preferable to settle the question of correctness entirely before you begin to tweak for performance, it is sometimes not practical to consider one without the other. Much of the rest of this chapter concerns WHERE clause logic and *optimisation*.

JOIN conditions vs WHERE conditions

Sometimes the optimiser will quietly rescue the query writer from her carelessness. For example, the output of *EXPLAIN* for the three queries

```
SELECT * FROM table1, table2 WHERE table1.id=b.id AND table1.val=2;
SELECT * FROM table1 JOIN table2 ON table1.id=table2.id WHERE table1.val=2;
SELECT * FROM table1 JOIN table2 ON table1.id=table2.id AND table1.val=2;
```

is identical:

```
+--------+--------+-----------------+---------+-----------+------+-------------+
| table  | type   | possible_keys   | key     | ref       | rows | Extra       |
+--------+--------+-----------------+---------+-----------+------+-------------+
| table1 | ref    | PRIMARY,val_idx | val_idx | const     | 6030 | Using where |
| table2 | eq_ref | PRIMARY         | PRIMARY | table1.id |    1 |             |
+--------+--------+-----------------+---------+-----------+------+-------------+
```

because with each query, the optimiser finds that

```
SELECT ... FROM table1 WHERE table1.val=2 [1]
```

yields a much smaller rowcount than

```
SELECT ... FROM table2                    [2]
```

so it uses the index on `table1.val` for `[1]`, and the index on `table2.id` for `[2]`.

The optimiser can use all the help you can give it. `Table1 STRAIGHT_JOIN table2` tells the optimiser that `table2` depends on `table1` and not vice-versa, so the optimiser can ignore the possibility of `table1` depending on `table2`. If the optimiser can guess the dependency incorrectly, or can spend significant time testing for various dependencies, `STRAIGHT_JOIN` may speed up query performance. `Table1 LEFT JOIN table2` likewise tells the optimiser that `table2` depends on `table1`.

For performance, it does not matter whether you impose other `table1` conditions in the `JOIN` clause or in the `WHERE` clause, because in each case they will be applied before the `JOIN`. *It can matter a lot, however, for additional conditions on `table2`*, especially if `table2` is large, because in …

```
SELECT ...
FROM table1 LEFT JOIN table2 ON table1.id=table2.id
WHERE table2.val='xyz'
```

the optimiser collects, in its internal temp table, all the rows that meet the `JOIN` condition, *then* discards rows which fail to meet the `WHERE` condition. But in …

```
SELECT ...
FROM table1 LEFT JOIN table2 ON table1.id=table2.id AND table2.val='xyz'
```

the optimiser does *not* have to add rows to its internal temporary table where the `table2.val='xyz'` condition fails, so performance is faster, especially if there is not enough dynamic memory for the optimiser's entire internal temporary table.

Here is a more complicated example using the `nwib` order entry database (Chapters *11* and *15*). To list products and quantities (0 and up) shipped and sold via a given courier for a given six-month period, we need a JOIN from `products` to `orderdetails` to capture products and sale quantities, then a JOIN to the `orders` table to capture the range of order dates. It seems intuitive to write the shipping and date conditions in the WHERE clause …

```
SELECT p.productID, p.productname,
       SUM(IF(oi.quantity IS NULL,0,oi.quantity)) AS qty
FROM products AS p
LEFT JOIN orderdetails AS oi ON (p.productid = oi.productid)
LEFT JOIN orders as o ON (o.orderid = oi.orderid)
WHERE (shipvia=2 AND o.orderdate BETWEEN '1997-01-01' AND '1997-06-30')
   OR o.orderid IS NULL
GROUP BY p.productid, p.productname
ORDER BY qty DESC;
```

but the query runs much faster if we make the joins INNER, and move the conditions to the second JOIN clause:

```
SELECT
  p.productID, p.productname, SUM(IF(o.orderid IS NULL,0,oi.quantity)) AS Qty
```

```
FROM products AS p
INNER JOIN orderdetails AS oi ON (p.productid = oi.productid)
INNER JOIN orders AS o
  ON (o.orderid = oi.orderid)
  AND o.shipvia=2
  AND (o.orderdate BETWEEN '1997-01-01' AND '1997-06-30')
GROUP BY p.productid, p.productname
ORDER BY qty DESC;
```

Why? As running *EXPLAIN* on the query shows, when we use LEFT JOINs and put scoping conditions in the WHERE clause, the query engine has to read and sort more rows…

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	o	ref	PRIMARY,ShipVia,OrderDate	ShipVia	4	const	326	Using where; Using temporary; Using filesort
1	SIMPLE	oi	ref	PRIMARY,ProductID	PRIMARY	4	nwib.o.OrderID	2	
1	SIMPLE	p	eq_ref	PRIMARY	PRIMARY	4	nwib.oi.ProductID	1	

…than when we write INNER JOINs and attach the conditions to the second join:

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	p	index	*NULL*	ProductName	40	*NULL*	78	Using index; Using temporary; Using filesort
1	SIMPLE	oi	ref	ProductID	ProductID	4	nwib.p.ProductID	13	
1	SIMPLE	o	eq_ref	PRIMARY	PRIMARY	4	nwib.oi.OrderID	1	Using where

Subqueries in the WHERE clause

Subqueries can test the (non-)existence of some condition in a table using a [NOT] EXISTS clause. In the following *simple schema*,

```
CREATE TABLE IF NOT EXISTS ridings (
  riding CHAR(10) PRIMARY KEY
);
CREATE TABLE IF NOT EXISTS parties (
  party CHAR(12) PRIMARY KEY
);
CREATE TABLE IF NOT EXISTS candidates (
  id INT PRIMARY KEY,
  name CHAR(10),
  riding CHAR(10),
  party CHAR( 10 )
);
INSERT INTO ridings VALUES ('Essex'),('Malton'),('Riverdale'),('East York');
INSERT INTO parties VALUES ('Liberal'),('Conservative'),('Socialist');
INSERT INTO candidates VALUES (1,'Anne Jones','Essex','Liberal'),
 (2,'Mary Smith','Malton','Liberal'),  (3,'Sara Black','Riverdale','Liberal'),
 (4,'Paul Jones','Essex','Socialist'), (5,'Ed While','Essex','Conservative'),
 (6,'Jim Kelly','Malton','Liberal'),   (7,'Fred Price','Riverdale','Socialist');
```

what are the ridings for which there are candidates? Here is one way to answer:

```
SELECT DISTINCT riding FROM ridings
WHERE EXISTS(SELECT * FROM candidates WHERE candidates.riding=ridings.riding);
+-----------+
| riding    |
+-----------+
| Essex     |
| Malton    |
| Riverdale |
+-----------+
```

To find which ridings have *no* candidates, insert NOT before EXISTS. Note that the same results can be had without subqueries …

```
SELECT DISTINCT ridings.riding FROM ridings
LEFT JOIN candidates ON ridings.riding = candidates.riding;

SELECT DISTINCT ridings.riding FROM ridings
```

```
LEFT JOIN candidates ON ridings.riding = candidates.riding
WHERE candidates.riding IS NULL;
```

with better performance because, generally before 5.7, IN(…) subqueries do not optimise index use as well as they should. What if we wish to know *which parties have candidates in all ridings*? No simple SQL quantifier will deliver the result, but a peculiar looking, double nesting of NOT EXISTS will do the job in one query:

```
SELECT DISTINCT party FROM parties
WHERE NOT EXISTS (
  SELECT * FROM ridings
  WHERE NOT EXISTS (
    SELECT * FROM candidates
    WHERE candidates.party=parties.party
      AND candidates.riding=ridings.riding
  )
);
+---------+
| party   |
+---------+
| Liberal |
+---------+
```

The query asks for the `parties`, for which there is no riding, for which there is no candidate. This is *relational division* again: the `parties` which have candidates in all `ridings` are the parties for which there is no riding with no candidate.

To write this relational division query with joins rather than subqueries, start with the inner query, finding ridings which are missing a candidate from a party:

```
SELECT DISTINCT p.party
FROM ridings r
CROSS JOIN parties p
LEFT JOIN candidates c ON r.riding=c.riding AND p.party=c.party
WHERE c.riding IS NULL;
```

Now write a left exclusion join from `parties` to the above query:

```
SELECT a.party
FROM parties a
LEFT JOIN (
  SELECT p.party
  FROM ridings r
  CROSS JOIN parties p
  LEFT JOIN candidates c ON r.riding=c.riding AND p.party=c.party
  WHERE c.riding IS NULL
) b ON a.party=b.party
WHERE b.party IS NULL;
+---------+
| party   |
+---------+
| Liberal |
+---------+
```


TOC ***Previous*** ***Next***