Gagnasafnsfræði

SQL

Hallgrímur H. Gunnarsson

Part I: Introduction

SQL

A major strengh of the relational model

Supports simple, powerful *querying* of data

Declarative language: specify *what* to return, not how

DBMS is responsible for efficient evaluation

The key: precise semantics for relational queries (relational algebra)

Allows the query optimizer to extensively re-order operations and still ensure that the answer does not change

Internal cost model drives use of indexes and choice of access path and physical operators

Basic SQL query

```
SELECT [DISTINCT] select_list
FROM tables
WHERE conditions
```

`tables`: A list of table names (possibly with a *range variable* after each name)

`select_list`: A list of column names from the given tables

`conditions`: A list of boolean expressions (e.g. `a = b`, `a > b`) combined using `AND`, `OR` and `NOT`

`DISTINCT` is an optional keyword indicating that the result should not contain duplicates

Conceptual query evaluation

```
SELECT [DISTINCT] select_list
FROM tables
WHERE conditions
```

Simplified evaluation:

1. do FROM: Compute the cross product of `tables`
2. do WHERE: Discard resulting rows if they fail the `conditions`
3. do SELECT: Filter columns that are not in the `select_list`
4. If `DISTINCT` is specified, filter duplicate rows

Probably the least efficient way to compute a query! But perhaps a useful mental model to get started

WHERE conditions

```
WHERE condition
```

`condition` is any boolean expression. Any row that does not satisfy the given condition will be eliminated from the output

A row satisfies the condition if it returns true when the actual row values are substituted for any variable references

Examples:

```
WHERE a > b
WHERE (a = b AND c = d) OR p < q;
```

SELECT list

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

The SELECT list specifies expressions that form the output rows of the SELECT statement

The expression can (and usually do) refer to columns computed in the FROM clause

Examples:

```
SELECT random();
SELECT 3 + 4;
SELECT a, b+c FROM t1;
SELECT * FROM t1;
```

Part II: Table expression

Table expression

```
SELECT select_list FROM table_expression
```

A *table expression* computes a table

The expression contains a `FROM` clause that is optionally followed by `WHERE`, `GROUP BY`, and `HAVING`

The optional `WHERE`, `GROUP BY`, and `HAVING` clauses specify a pipeline of successive transformations performed on the table derived from the `FROM` clause

Table expression: FROM

```
FROM table_reference [, table_reference [, ...]]
```

The `FROM` clause derives a table from one or more other tables given in a comma-separated table reference list

A table reference can be a table name, or a derived table such as a subquery, a table join, or complex combinations of these

If more than one table reference is listed in the `FROM` clause, they are cross-joined to form the intermediate result table

The result table is subject to transformations by the `WHERE`, `GROUP BY`, and `HAVING` clauses

Table expression: Joins in FROM

A joined table is a table derived from two other (real or derived) tables according to the rules of the particular join type

Standard join types: inner, outer and cross-joins

Table data for examples:

id	name
1	Aron
2	Bjarni
3	Davíð
4	Einar

pid	simanumer
1	515-1000
1	525-1001
2	535-2000
5	545-9000

Cross join

```
FROM T1 CROSS JOIN T2
FROM T1, T2
```

For every possible combination of rows from T1 and T2 (i.e. a Cartesian product), the joined table will contain a row of all columns from T1 followed by all columns from T2

id	name
1	Aron
2	Bjarni
3	Davíð
4	Einar

pid	simanumer
1	515-1000
1	525-1001
2	535-2000
5	545-9000

$\Rightarrow$

id	name	pid	simanumer
1	Aron	1	515-1000
1	Aron	1	525-1001
1	Aron	2	535-2000
1	Aron	5	545-9000
2	Bjarni	1	515-1000
..	..	..	..

Inner join

```
T1 [INNER] JOIN T2 ON boolean_expression
T1 [INNER] JOIN T2 USING (column1 [, ...])
```

`INNER` is optional. If you only specify `JOIN` then `INNER` is the default

For each row of `R1` of `T1`, the joined table has a row for each row in `T2` that satisfies the join condition with `R1`. Join condition is either `ON` or `USING`

The `ON` clause takes a boolean value expression of the same kind as is used in a `WHERE` clause. A pair of rows from `T1` and `T2` match if the `ON` expression evaluates to true for them.

Inner join USING

```
T1 [INNER] JOIN T2 ON boolean_expression
T1 [INNER] JOIN T2 USING (column1 [, ...])
```

`USING` is a shorthand notation: it takes a comma-separated list of column names, which both tables must have in common, and forms a join condition specifying equality on each of these pairs of columns

`USING(a,b,c)` is equivalent to `ON t1.a=t2.a AND t1.b=t2.b AND t1.c=t2.c`

Inner join example

```
FROM T1 INNER JOIN T2 ON T1.id=T2.pid
```

id	name
1	Aron
2	Bjarni
3	Davíð
4	Einar

pid	simanumer
1	515-1000
1	525-1001
2	535-2000
5	545-9000

$\Rightarrow$

id	name	pid	simanumer
1	Aron	1	515-1000
1	Aron	1	525-1001
2	Bjarni	2	535-2000

Outer join

```
T1 {LEFT|RIGHT|FULL} [OUTER] JOIN T2 ON boolean_expression
T1 {LEFT|RIGHT|FULL} [OUTER] JOIN T2 USING (column1 [, ...])
```

`OUTER` is optional. If you only specify `LEFT`, `RIGHT` or `FULL` then `OUTER` is implied

An outer join does not require each record in the two joined tables to have a matching record

The joined table retains each record even if no other matching record exists

Left outer join (left join)

```
T1 LEFT [OUTER] JOIN T2 ON boolean_expression
T1 LEFT [OUTER] JOIN T2 USING (column1 [, ...])
```

The result of a `LEFT OUTER JOIN` (or simply `LEFT JOIN`) for table `T1` and `T2` always contains all records of the "left" table, `T1`, even if the join condition did not find any matching record in the "right" table `T2`

This means that if the join condition matches 0 rows in `T2`, the join will still return a row in the result - but with `NULL` in each column from `T2`

Left outer join example

```
FROM T1 LEFT JOIN T2 ON T1.id=T2.pid
```

id	name
1	Aron
2	Bjarni
3	Davíð
4	Einar

pid	simanumer
1	515-1000
1	525-1001
2	535-2000
5	545-9000

$\Rightarrow$

id	name	pid	simanumer
1	Aron	1	515-1000
1	Aron	1	525-1001
2	Bjarni	2	535-2000
3	Davíð	NULL	NULL
4	Einar	NULL	NULL

Right outer join (right join)

```
T1 RIGHT [OUTER] JOIN T2 ON boolean_expression
T1 RIGHT [OUTER] JOIN T2 USING (column1 [, ...])
```

The result of a `RIGHT OUTER JOIN` (or simply `RIGHT JOIN`) closely resembles a left outer join, except with the treatment of the tables reversed

Every row from the "right" table, `T2`, will appear in the joined table at least once. If no matching row from the "left" table, `T1`, exists then NULL will appear in columns from `T1` for those record that have no match in `T2`

A right outer join returns all the values from the right table and matched values from the left table (NULL in case of no matching join condition)

Note: `T1 LEFT JOIN T2` is equivalent to `T2 RIGHT JOIN T1`

Right outer join example

```
FROM T1 RIGHT JOIN T2 ON T1.id=T2.pid
```

id	name
1	Aron
2	Bjarni
3	Davíð
4	Einar

pid	simanumer
1	515-1000
1	525-1001
2	535-2000
5	545-9000

$\Rightarrow$

id	name	pid	simanumer
1	Aron	1	515-1000
1	Aron	1	525-1001
2	Bjarni	2	535-2000
NULL	NULL	5	545-9000

Full outer join (full join)

```
T1 FULL [OUTER] JOIN T2 ON boolean_expression
T1 FULL [OUTER] JOIN T2 USING (column1 [, ...])
```

The result of a `FULL OUTER JOIN` (or simply `FULL JOIN`) combines the effect of applying both left and right outer joins

Where records in the `FULL OUTER JOIN` tables do not match, the result set will have NULL values for every column of the table that lacks a matching row

Full outer join example

```
FROM T1 FULL JOIN T2 ON T1.id=T2.pid
```

id	name
1	Aron
2	Bjarni
3	Davíð
4	Einar

pid	simanumer
1	515-1000
1	525-1001
2	535-2000
5	545-9000

$\Rightarrow$

id	name	pid	simanumer
1	Aron	1	515-1000
1	Aron	1	525-1001
2	Bjarni	2	535-2000
3	Davíð	NULL	NULL
4	Einar	NULL	NULL
NULL	NULL	5	545-9000

Self-join

A self-join is joining a table to itself.

Not really a different type of join, all the usual rules apply. It just so happens that `T1` is the same table as `T2` in this case.

Example:

```
SELECT a.pid, b.pid
FROM T2 a, T2 b
WHERE a.simanumer=b.simanumer
  AND a.pid != b.pid;
```

Part III: Grouping

GROUP BY

```
GROUP BY expression [, ...]
```

`GROUP BY` will collapse into a single row all selected rows that share the same values for the grouped expressions

`expression` can be an input column name, or the name of an output column (`SELECT` list item), or an arbitrary expression formed from input-column values.

Aggregate functions, e.g. `COUNT`, `MAX`, `MIN`, `SUM`, `AVG`, are computed across all rows making up each group, producing a separate value for each group.

Without `GROUP BY`, an aggregate function computes a single value across all the selected rows

HAVING

```
GROUP BY expression [, ...]
HAVING condition
```

`condition` is the same as specified for the `WHERE` clause

`HAVING` eliminates group rows that do not satisfy the condition

`HAVING` is different from `WHERE`: `WHERE` filters individual rows *before* the application of `GROUP BY`, while `HAVING` filters group rows created by `GROUP BY`

`HAVING` can only reference table columns from within aggregate functions

Part IV: Set operations

Combining queries with set operations

```
query1 UNION [ALL] query2
query1 INTERSECT [ALL] query2
query1 EXCEPT [ALL] query2
```

Queries must be "union compatible", same number of columns and compatible data types. In all cases, duplicates are eliminated unless ALL is specified

UNION appends the result of query1 to the result of query2

INTERSECT returns all rows that are in the result of both query1 and query2

EXCEPT returns all rows that are in the result of query1 but not query2

Left associative: q1 UNION q2 UNION q3 evaluated as (q1 UNION q2) UNION q3

Part V: IN and NOT IN

Using IN with values

```
expression IN (value1 [, ...])
```

is equivalent to:

```
expression = value1
OR
expression = value2
OR
...
```

Using NOT IN with values

```
expression NOT IN (value1 [, ...])
```

is equivalent to:

```
expression != value1
AND
expression != value2
AND
...
```

Note: `x NOT IN y` is equivalent to `NOT (x IN y)`

Part VI: Subquery expressions

Subquery expressions

```
EXISTS (subquery)
expression IN (subquery)
expression NOT IN (subquery)
expression operator ANY (subquery)
expression operator ALL (subquery)
```

Subquery expressions are *boolean expressions*, i.e. they return boolean (true/false) results

A subquery is an arbitrary `SELECT` statement

Subquery expression: EXISTS

```
EXISTS (subquery)
```

The subquery is evaluated to determine whether it returns any rows

If it returns at least one row, the result of `EXISTS (subquery)` is true, otherwise it is false

The subquery can refer to variables from the surrounding query

Since the result depends only on whether any rows are returned, and not on the contents of those rows, the output list of the subquery is not important

Subquery expression: IN

```
expression IN (subquery)
```

The subquery must return exactly one column. The left-hand expression is evaluated and compared to each row of the subquery result

The result is true if any equal subquery row is found

The result is false if no equal row is found (including the case where the subquery returns no rows)

Two rows are equal if all their corresponding members are non-null and equal

For more than one column: use row constructor and multi-column subquery

Subquery expression: NOT IN

```
expression NOT IN (subquery)
```

Opposite of `expression IN (subquery)`

The result is true if all rows are unequal (including the case where the subquery returns no rows)

The result is false if any equal row is found

Subquery expression: ANY

```
expression operator ANY (subquery)
```

The subquery must return exactly one column

The left-hand expression is evaluated and compared to each row of the subquery result using the given boolean `operator`, e.g. $<$, $>$

The result is true if any comparison is true

The result is false if all comparisons are false (including the case where the subquery returns no rows)

Note: `IN` is equivalent to `= ANY`

Also supports row constructor for multiple columns

Subquery expression: ALL

```
expression operator ALL (subquery)
```

The subquery must return exactly one column

The left-hand expression is evaluated and compared to each row of the subquery result using the given boolean `operator`, e.g. $<$, $>$

The result is true if all comparisons are true (including the case where the subquery returns no rows)

The result is false if any comparison is false

Note: `NOT IN` is equivalent to `!= ALL`

Also supports row constructor for multiple columns

Part VII: Summary

Evaluation of SELECT

1. do FROM: compute tables
2. do WHERE: eliminate rows that do not satisfy the given WHERE condition
3. do GROUP BY: divide output into groups of rows that match on one or more values
4. do HAVING: eliminate groups that do not satisfy given HAVING condition
5. do SELECT: calculate output rows
6. do UNION/INTERSECT/EXCEPT: combine result sets
7. do ORDER BY: sort the result set in the given order
8. do DISTINCT: eliminate duplicate rows
9. do LIMIT: take subset of result