EPBI 414

Unit 7
Advanced SQL

The Recap - Unit 6

- SQL and MySQL
  - Connecting using the command line interface, MySQL Workbench
- Coding standards and conventions
- Basics of SQL
  - Selecting, filtering, inner joins
  - Symbolic logic

Unit 7 Overview

- More complex joins
  - `RIGHT` / `LEFT JOIN`, `FULL JOIN`
  - `ON` vs. `WHERE`
- `UNION`, `UNION ALL`
- Data Summary and Analysis
- Modifying Data

More complicated joins

- Last time, we covered the ***inner join***
  - Returns rows when the criteria is met
- The ***left*** / ***right outer join*** returns all records in one table, and matches from the other
  - The ***outer*** is usually omitted
- The ***full join*** (sometimes ***full outer join***) gives you all records from both tables
  - Not supported in MySQL

An example `LEFT JOIN`

```
SELECT p.patient_id,
       p.dob,
       v.visit_id,
       v.provider_id,
       v.visit_date
       FROM patients AS p
       LEFT JOIN last_visit AS v
       ON p.patient_id = v.patient_id;
```

patient_id	pcp_id	dob
1	1	12/08/1944
2	1	08/03/1999
3	2	02/07/1983

visit_id	patient_id	provider_id	visit_date
1	1	3	02/12/2016
2	3	1	09/03/2016

LEFT JOIN Results

```
SELECT p.patient_id,
       p.dob,
       v.visit_id,
       v.provider_id,
       v.visit_date
       FROM patients AS p
       LEFT JOIN last_visit AS v
       ON p.patient_id = v.patient_id;
```

patient_id	dob	visit_id	provider_id	visit_date
1	12/08/1944	1	3	02/12/2016
2	08/03/1999	NULL	NULL	NULL
3	02/07/1983	2	1	09/03/2016

ON vs. WHERE

- Left & right joins give us an opportunity to discuss the difference between `ON` and `WHERE`
- In short: `WHERE` filters results, `ON` joins rows together
- What's the difference?

WHERE filters rows

```
SELECT p.patient_id,
       p.dob,
       v.visit_id,
       v.provider_id,
       v.visit_date
       FROM patients AS p
       LEFT JOIN last_visit AS v
       ON p.patient_id = v.patient_id
       WHERE p.dob >= '1950-01-01';
```

patient_id	dob	visit_id	provider_id	visit_date
2	08/03/1999	NULL	NULL	NULL
3	02/07/1983	2	1	09/03/2016

ON joins rows together

```
SELECT p.patient_id,
       p.dob,
       v.visit_id,
       v.provider_id,
       v.visit_date
       FROM patients AS p
       LEFT JOIN last_visit AS v
       ON p.patient_id = v.patient_id
           AND p.dob >= '1950-01-01';
```

patient_id	dob	visit_id	provider_id	visit_date
1	12/08/1944	NULL	NULL	NULL
2	08/03/1999	NULL	NULL	NULL
3	02/07/1983	2	1	09/03/2016

`ON` and inner vs. outer

- Inner joins only return a row when the `ON` condition is met
  - So, `ON` and `WHERE` are equivalent!
- Outer joins always return some rows
  - This makes `ON` and `WHERE` different!
- What is the practical meaning of this?

Writing good queries

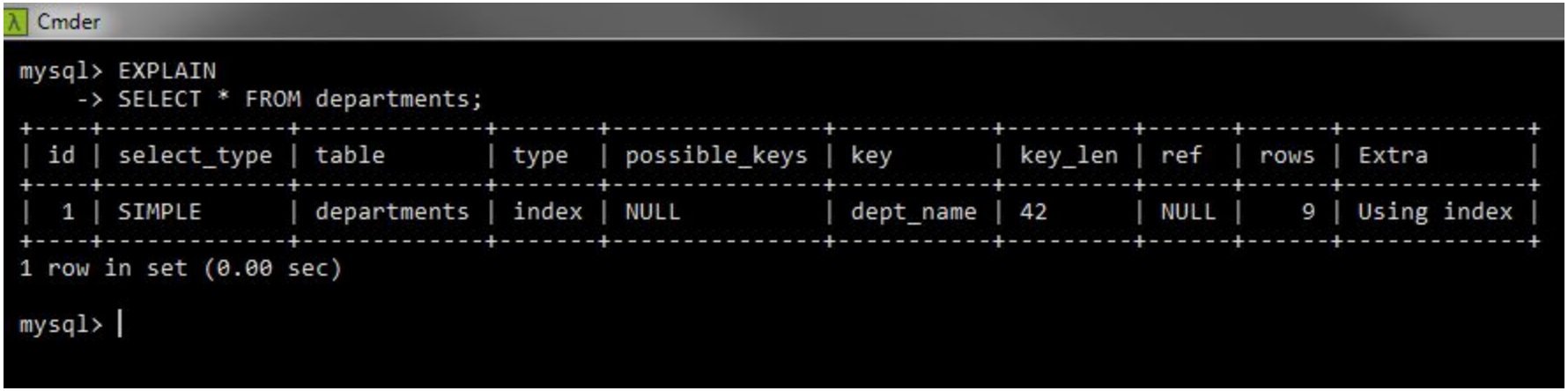
- It's tempting to think something like...
  - "If the `ON` controls which rows get included in the join at all, while the `WHERE` filters out rows after the joining is done...I should use `ON` to return smaller sets of data to start with!"
- Unfortunately, this instinct doesn't quite work
- The "sequence" isn't that simple, because...

The query optimizer

- Every RDBMS has something called a ***query optimizer***
- Query optimization varies between database engines
  - Fairly low-level technical part of the RDBMS
- The optimizer makes "hard and fast" rules difficult to articulate

EXPLAIN

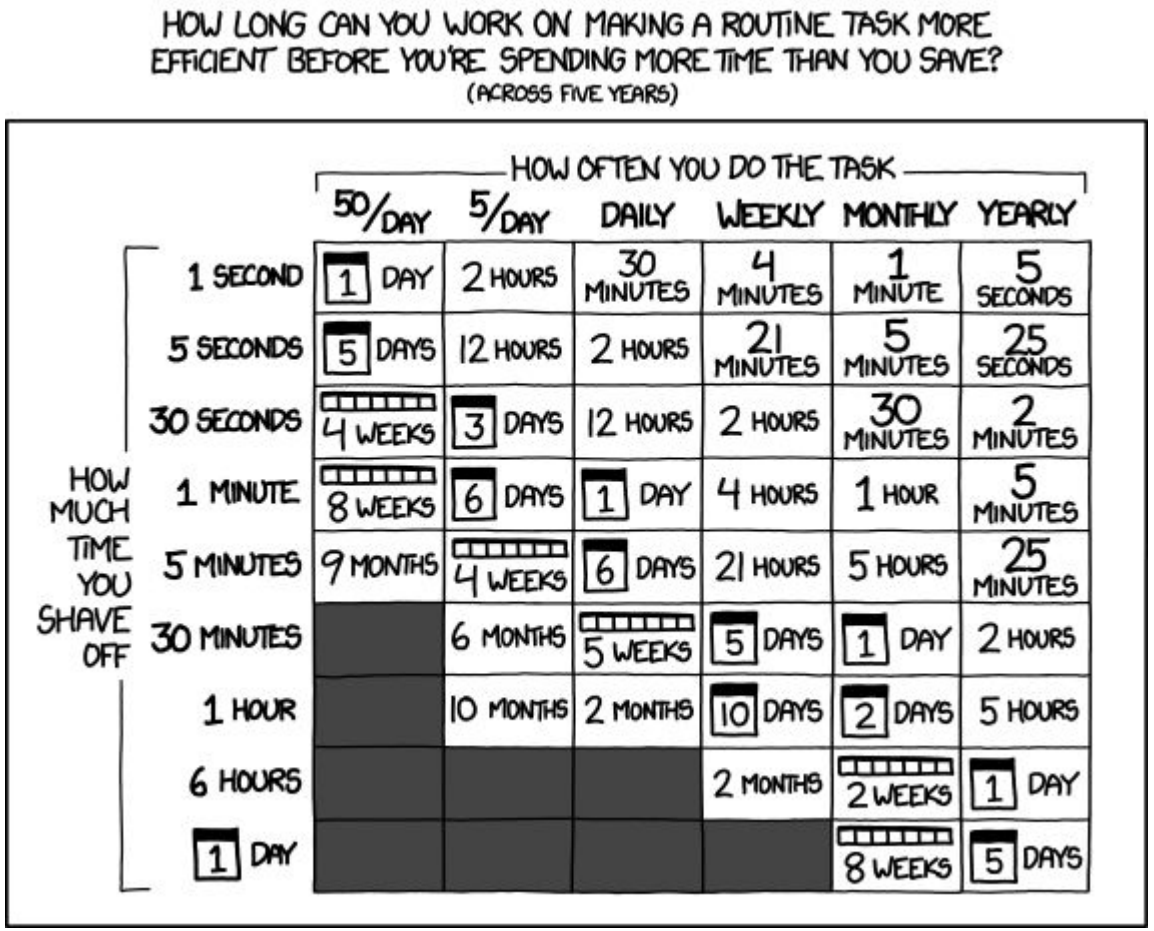
- One neat feature of most RDBMSs is the `EXPLAIN` feature
- By running `EXPLAIN <QUERY>`, you can get your RDBMS to tell you its plan
- This can let you (more likely, expert DBAs) identify bottlenecks and improve queries

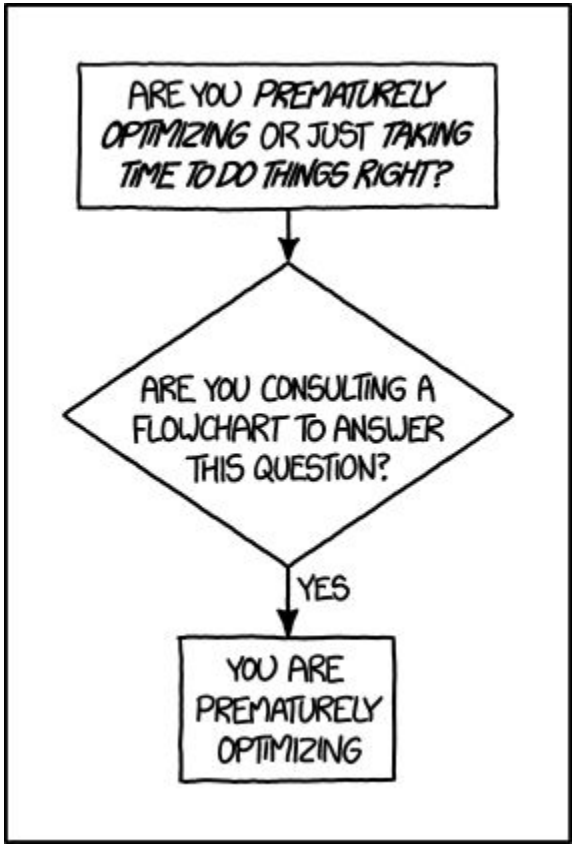


Example `EXPLAIN` Query

`EXPLAIN` and You

- `EXPLAIN` is an in-depth RDBMS feature
  - Also, very dependant on your RDBMS
- The query optimizer isn't magic
  - Can't save a terrible query from itself
- You should focus on logical queries that are efficient ***enough***
  - "Premature optimization is the root of all evil." Attributed to Donald Knuth





Two relevant xkcd comics[1,2]

A rule of thumb

- Keeping in mind that "hard and fast" rules are hard to come by...
- When you are writing your query, you can think of it like this:

`ON` ***happens before*** `WHERE`

or

Joins before filters

FULL JOIN

- As mentioned, MySQL does not support ***full outer joins*** (or just ***full joins***)
- Full joins are relatively uncommon
  - There are few circumstances where you want all the rows from both tables, regardless of match
- You can simulate a full join in MySQL if you need it
  - Not required for this class

Other join notes

- You can join a table against itself - this is called a ***self join***
  - Sometimes, it can make filtering faster - really depends on the optimizer
- Many RDBMSs support other join types
  - Natural join, cross join, implicit join...
- For your day-to-day, stick to the main ones

Many-to-many joins

- Caution is advised when joining on two columns which both contain duplicates
- SQL will produce all possible combinations from your join query
  - This may not be - probably isn't - what you want
- Many-to-many is also expensive computationally

Example many-to-many

```
SELECT  s.*,
        t.*
        FROM
        (
        SELECT  *
                FROM salaries
                WHERE emp_no = 10004
        ) AS s
        INNER JOIN
        (
        SELECT  *
                FROM titles
                WHERE emp_no = 10004
        ) AS t
        ON s.emp_no = t.emp_no;
```

UNION

- The `UNION` command allows you to "stack" the results of two queries vertically
- Useful when you run two generally similar queries with different filters
- Columns need to line up, and be compatible!
  - The RDBMS can usually ***coerce*** values from one type to another (i.e. numeric to character)

`UNION` Syntax

- Unions are performed by writing `SELECT` statements, separated by `UNION`

```
SELECT * FROM tbl1
UNION
SELECT * FROM tbl2;
```

`UNION` vs. `UNION ALL`

- The `UNION` statement ***often*** (no guarantees!) removes duplicate rows
- If this is not desired behavior, `UNION ALL` will generally override it
- This tends to depend on your RDBMS

Summarizing Data

- So far, we have just accessed and filtered data using SQL
- But SQL can do much more
  - Aggregation
  - Grouping
  - Calculations
- These features are often very fast, compared to other software

SQL Analysis Tools

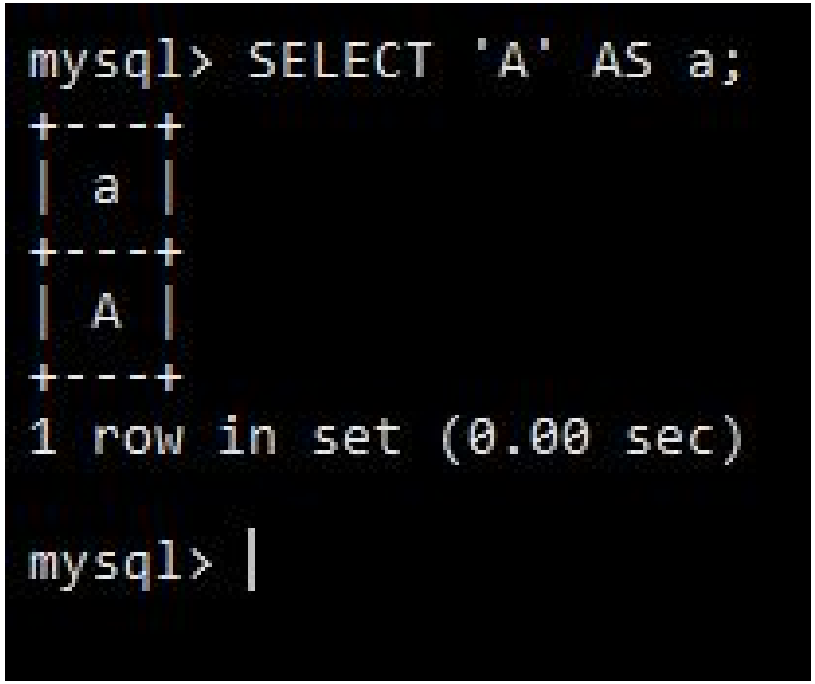
- Calculated Fields
  - SQL functions
- Selecting unique records (`DISTINCT`)
- Grouping records together (`GROUP BY`)
  - Aggregation functions

Getting calculated fields

- Most of the time, a ***field*** is synonymous with a ***column***
  - Codd never talked about ***fields*** at all
- However, most RDBMSs will let you add ***calculated fields*** to your queries
- Client apps don't see any difference between ***calculated fields*** and regular columns

Constant value example

```
SELECT 'A' AS a;
```



Naming calculations

- You can name your calculated fields using the `AS` statement
  - Just like subqueries
- Calculated field names tend not to be available within the same query
  - Can be accessed from subqueries
- Very useful if you are using `UNION`!

Using SQL Functions

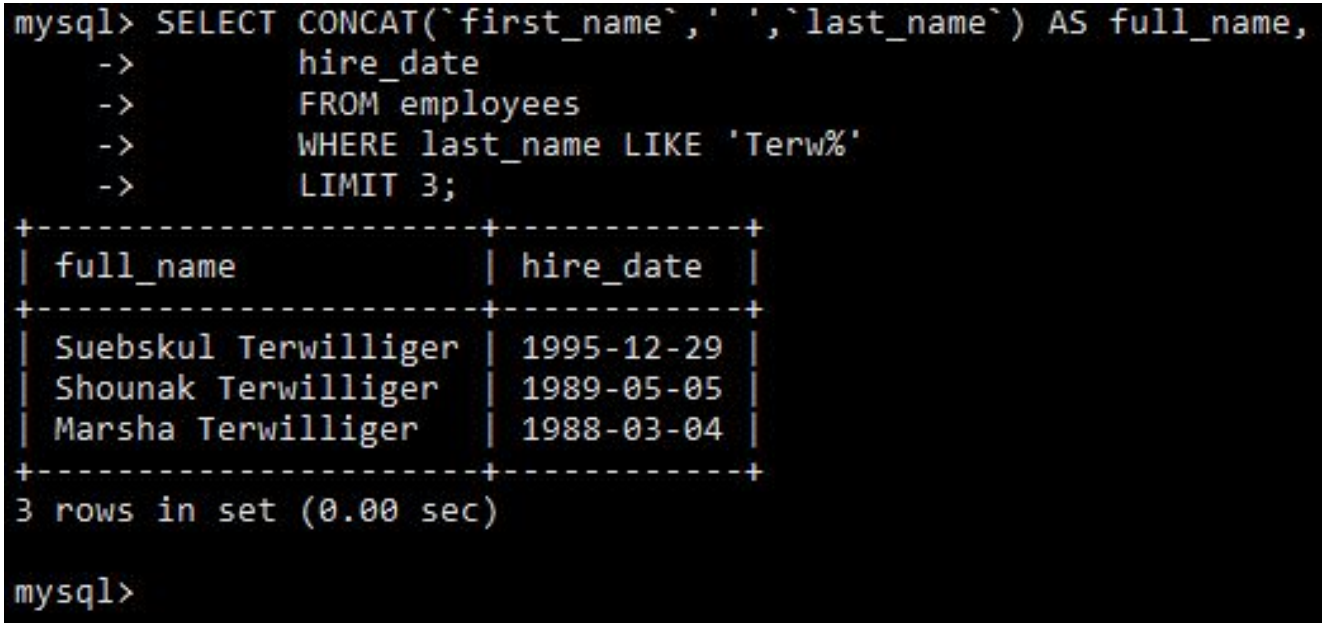
- SQL allows you to use both basic mathematics and built-in SQL functions in your calculated fields

- There are a ***lot*** of SQL functions
  - Can't possibly cover them all in class

- Here is an example: ***concat***enating two strings together

Example SQL function

```
SELECT CONCAT(`first_name`,' ',`last_name`) AS full_name,
       hire_date
       FROM employees
       WHERE last_name LIKE 'Terw%'
       LIMIT 3;
```



Another function

```
SELECT emp_no,
       first_name,
       last_name,
       TIMESTAMPDIFF(year,birth_date,hire_date) AS age_at_hire
       FROM employees LIMIT 10;
```

```
mysql> SELECT emp_no,
    ->        first_name,
    ->        last_name,
    ->        TIMESTAMPDIFF(year,birth_date,hire_date) AS age_at_hire
    ->        FROM employees LIMIT 10;
+--------+------------+-----------+-------------+
| emp_no | first_name | last_name | age_at_hire |
+--------+------------+-----------+-------------+
|  10001 | Georgi     | Facello   |          32 |
|  10002 | Bezalel    | Simmel    |          21 |
|  10003 | Parto      | Bamford   |          26 |
|  10004 | Chirstian  | Koblick   |          32 |
|  10005 | Kyoichi    | Maliniak  |          34 |
|  10006 | Anneke     | Preusig   |          36 |
|  10007 | Tzvetan    | Zielinski |          31 |
|  10008 | Saniya     | Kalloufi  |          36 |
|  10009 | Sumant     | Peac      |          32 |
|  10010 | Duangkaew  | Piveteau  |          26 |
+--------+------------+-----------+-------------+
10 rows in set (0.00 sec)

mysql>
```

Using field names - bad

```
SELECT emp_no,
       emp_no + 10 AS emp_no_plus,
       emp_no_plus * emp_no_plus AS emp_no_plus_squared
       FROM employees
       LIMIT 10;
```

```
mysql> SELECT emp_no,
    ->        emp_no + 10 AS emp_no_plus,
    ->        emp_no_plus * emp_no_plus AS emp_no_plus_squared
    ->        FROM employees
    ->        LIMIT 10;
ERROR 1054 (42S22): Unknown column 'emp_no_plus' in 'field list'
mysql>
```

Using field names - good

```
SELECT subq.emp_no,
       subq.emp_no_plus,
       subq.emp_no_plus * subq.emp_no_plus AS emp_no_plus_squared
       FROM
       (
       SELECT emp_no,
              emp_no + 10 AS emp_no_plus
              FROM employees
       ) AS subq
       LIMIT 10;
```

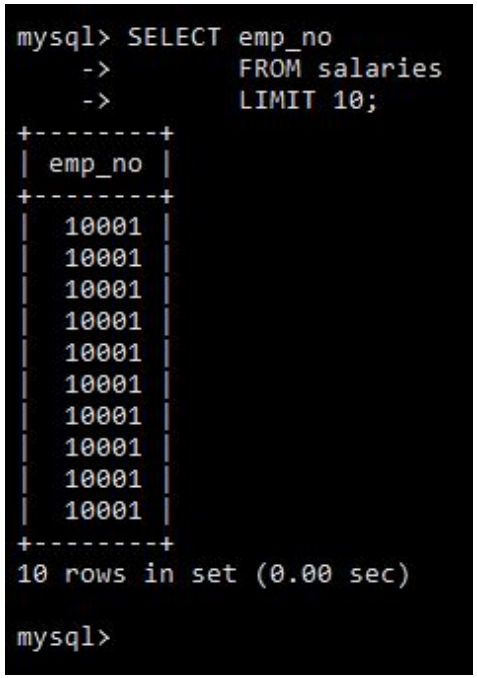
```
mysql> SELECT subq.emp_no,
    ->        subq.emp_no_plus,
    ->        subq.emp_no_plus * subq.emp_no_plus AS emp_no_plus_squared
    ->        FROM
    ->        (
    ->        SELECT emp_no,
    ->               emp_no + 10 AS emp_no_plus
    ->               FROM employees
    ->        ) AS subq
    ->        LIMIT 10;
+--------+-------------+---------------------+
| emp_no | emp_no_plus | emp_no_plus_squared |
+--------+-------------+---------------------+
|  10001 |       10011 |           100220121 |
|  10002 |       10012 |           100240144 |
|  10003 |       10013 |           100260169 |
|  10004 |       10014 |           100280196 |
|  10005 |       10015 |           100300225 |
|  10006 |       10016 |           100320256 |
|  10007 |       10017 |           100340289 |
|  10008 |       10018 |           100360324 |
|  10009 |       10019 |           100380361 |
|  10010 |       10020 |           100400400 |
+--------+-------------+---------------------+
10 rows in set (0.11 sec)

mysql>
```

`DISTINCT`

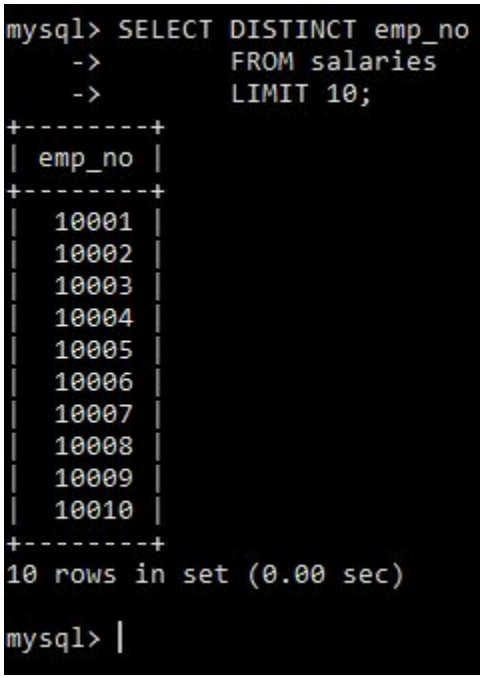
- Often, you have tables with values repeated across rows
- `DISTINCT` is a keyword that helps us obtain a unique list of values
- Often,this is useful when counting (coming up)

DISTINCT values

```
SELECT emp_no
       FROM salaries
       LIMIT 10;
```




```
SELECT DISTINCT emp_no
       FROM salaries
       LIMIT 10;
```



`DISTINCT` on rows

- You can use the `DISTINCT` keyword on multiple variables
- MySQL will return the distinct tuples from your variables
- Cannot easily return `DISTINCT` plus some non-distinct value
  - That's where grouping comes in

Caveats on `DISTINCT`

- General caveat: by definition, a SQL table (relation) is an unordered set
  - If you don't use an `ORDER BY`, you have no guarantee that the order will be consistent between queries
- Using `DISTINCT` on multiple columns can be very time-consuming
  - Indexes can help

Break Time

Grouping and Aggregation

- One of the most powerful tools that you have in SQL is the ability to ***group*** and ***aggregate*** data

- By using the five basic SQL aggregation functions, and a `GROUP BY` statement, you can get very useful information

Aggregation Functions

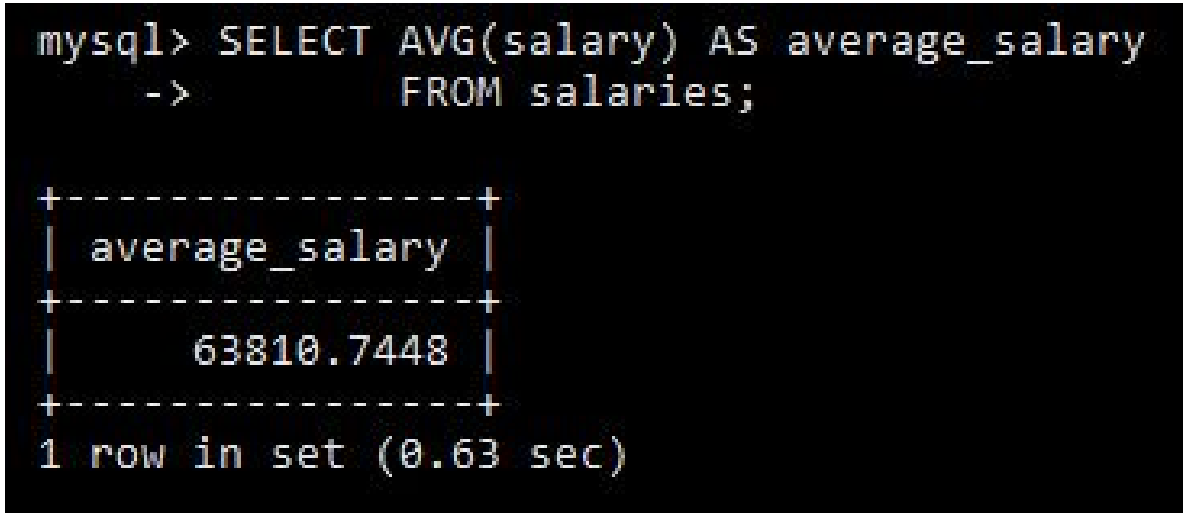
- SQL has five basic aggregation functions:
  - `AVG()`
  - `COUNT()`
  - `MAX()`
  - `MIN()`
  - `SUM()`

Notes on aggregations

- The aggregation functions only work either in ***isolation***, or as part of a ***grouping***
- You can't ask for a regular column and a aggregation without doing a group
- You might not be able to use certain aggregations on certain data types

Basic aggregation

```
SELECT AVG(salary) AS average_salary
       FROM salaries;
```



The average of all salary records in company history...(not super useful)

Basic frequencies

```
SELECT  title,
        gender,
        COUNT(*) AS n_employees FROM
        (
        SELECT  e.emp_no,
                e.gender,
                t.title
                FROM employees AS e
                INNER JOIN
                (
                SELECT  emp_no,
                        title
                        FROM titles
                        WHERE to_date = '9999-01-01'
                ) AS t
                ON e.emp_no = t.emp_no
        ) AS subq
        GROUP BY title,gender;
```

```
mysql> SELECT  title,
    ->         gender,
    ->         COUNT(*) AS n_employees FROM
    ->         (
    ->         SELECT  e.emp_no,
    ->                 e.gender,
    ->                 t.title
    ->                 FROM employees AS e
    ->                 INNER JOIN
    ->                 (
    ->                 SELECT  emp_no,
    ->                         title
    ->                         FROM titles
    ->                         WHERE to_date = '9999-01-01'
    ->                 ) AS t
    ->                 ON e.emp_no = t.emp_no
    ->         ) AS subq
    ->         GROUP BY title,gender;
+--------------------+--------+-------------+
| title              | gender | n_employees |
+--------------------+--------+-------------+
| Assistant Engineer | M      |        2148 |
| Assistant Engineer | F      |        1440 |
| Engineer           | M      |       18571 |
| Engineer           | F      |       12412 |
| Manager            | M      |           5 |
| Manager            | F      |           4 |
| Senior Engineer    | M      |       51533 |
| Senior Engineer    | F      |       34406 |
| Senior Staff       | M      |       49232 |
| Senior Staff       | F      |       32792 |
| Staff              | M      |       15436 |
| Staff              | F      |       10090 |
| Technique Leader   | M      |        7189 |
| Technique Leader   | F      |        4866 |
+--------------------+--------+-------------+
14 rows in set (0.65 sec)

mysql>
```

How the frequencies look

HAVING vs. WHERE

- When you aggregate, `WHERE` filters out rows ***before*** the calculations
- To filter on the resulting groups, you use `HAVING` instead
  - Or you can put your aggregation in a subquery...
- The performance implications are generally subtle - and relate back to `EXPLAIN`

Another rule of thumb

- Another general rule of thumb is:

  Use `WHERE` **before** `GROUP BY`

  Use `HAVING` **after** `GROUP BY`

A HAVING example

```
SELECT  title,
        gender,
        COUNT(*) AS n_employees FROM
        (
        SELECT  e.emp_no,
                e.gender,
                t.title
                FROM employees AS e
                INNER JOIN
                (
                SELECT  emp_no,
                        title
                        FROM titles
                        WHERE to_date = '9999-01-01'
                ) AS t
                ON e.emp_no = t.emp_no
        ) AS subq
        GROUP BY title,gender
        HAVING gender = 'M';
```

```
mysql> SELECT  title,
    ->         gender,
    ->         COUNT(*) AS n_employees FROM
    ->         (
    ->         SELECT  e.emp_no,
    ->                 e.gender,
    ->                 t.title
    ->                 FROM employees AS e
    ->                 INNER JOIN
    ->                 (
    ->                 SELECT  emp_no,
    ->                         title
    ->                         FROM titles
    ->                         WHERE to_date = '9999-01-01'
    ->                 ) AS t
    ->                 ON e.emp_no = t.emp_no
    ->         ) AS subq
    ->         GROUP BY title,gender
    ->         HAVING gender = 'M';
+--------------------+--------+-------------+
| title              | gender | n_employees |
+--------------------+--------+-------------+
| Assistant Engineer | M      |        2148 |
| Engineer           | M      |       18571 |
| Manager            | M      |           5 |
| Senior Engineer    | M      |       51533 |
| Senior Staff       | M      |       49232 |
| Staff              | M      |       15436 |
| Technique Leader   | M      |        7189 |
+--------------------+--------+-------------+
7 rows in set (0.67 sec)
```

`HAVING` Version

A `WHERE` example

```
SELECT  title,
        gender,
        COUNT(*) AS n_employees FROM
        (
        SELECT  e.emp_no,
                e.gender,
                t.title
                FROM employees AS e
                INNER JOIN
                (
                SELECT  emp_no,
                        title
                        FROM titles
                        WHERE to_date = '9999-01-01'
                ) AS t
                ON e.emp_no = t.emp_no
        ) AS subq
        WHERE gender = 'M'
        GROUP BY title,gender;
```

```
mysql> SELECT  title,
    ->         gender,
    ->         COUNT(*) AS n_employees FROM
    ->         (
    ->         SELECT  e.emp_no,
    ->                 e.gender,
    ->                 t.title
    ->                 FROM employees AS e
    ->                 INNER JOIN
    ->                 (
    ->                 SELECT  emp_no,
    ->                         title
    ->                         FROM titles
    ->                         WHERE to_date = '9999-01-01'
    ->                 ) AS t
    ->                 ON e.emp_no = t.emp_no
    ->         ) AS subq
    ->         WHERE gender = 'M'
    ->         GROUP BY title,gender;
+--------------------+--------+-------------+
| title              | gender | n_employees |
+--------------------+--------+-------------+
| Assistant Engineer | M      |        2148 |
| Engineer           | M      |       18571 |
| Manager            | M      |           5 |
| Senior Engineer    | M      |       51533 |
| Senior Staff       | M      |       49232 |
| Staff              | M      |       15436 |
| Technique Leader   | M      |        7189 |
+--------------------+--------+-------------+
7 rows in set (0.61 sec)
```

`WHERE` Version

DISTINCT COUNTing

```
SELECT COUNT(DISTINCT emp_no) AS n_employees
       FROM salaries
       WHERE to_date = '9999-01-01';
```

```
mysql> SELECT COUNT(DISTINCT emp_no) AS n_employees
    ->        FROM salaries
    ->        WHERE to_date = '9999-01-01';
+-------------+
| n_employees |
+-------------+
|      240124 |
+-------------+
1 row in set (0.83 sec)

mysql>
```

Putting it together

- SQL has many individual tools that are useful
  - Grouping, aggregating, ordering, limiting, filtering, et cetera
- You can combine these tools to answer relatively specific questions!
- You will learn to "think SQL"

Modifying data

- So far, all we've discussed has been pulling data out of SQL
- To close, we'll touch upon some basics of creating and modifying your own tables
- Unlike before, you will have destructive powers now
  - Use them wisely

Note

- Though we're going to cover some basics of database administration and modifying data, the course focus is on ***consumption***
- This is a very broad survey of these topics
- Does not substitute for having experience or a good DBA

What types of powers?

- Adding, modifying, and deleting data from existing tables
- Creating, modifying, and deleting tables
  - Setting data types, learning about constraints
  - We won't get into creating and deleting schema
- Using temporary tables, if you can
  - A very useful feature!

INSERT UPDATE DELETE

- `INSERT` is used to add new records to a table
- `UPDATE` is used to change the values of some rows of a table
- `DELETE` is used to remove rows from a table

INSERTing yourself

- Data insertion is generally the least risky of operations
  - Not risk free
- Generally, the syntax is:

```
INSERT INTO sometable (somecol1,somecol2,somecol3)
   VALUES (value1,value2,value3);

INSERT INTO sometable VALUES (value1,value2,value3);
```

Making `UPDATE`s

- `UPDATE` is a more dangerous command
  - Can easily overwrite valid data (I hope you have backups)
- Because of the power of `UPDATE`, it is very crucial to use the `WHERE` clause with it
  - By default, MySQL won't let you update without it
- If you leave off the WHERE, you effectively overwrite an entire column!

UPDATE Syntax

```
UPDATE sometable
    SET somecol1=value1,
        somecol2=value2
    WHERE somefilter;
```

Example:

```
UPDATE patients
    SET withdrawn = 1
    WHERE patient_id = 12345;
```

`DELETE` Your Data

- Are you **<u>really</u>** sure you should be using this?
  - Just checking
- Even more powerful than `UPDATE`, and often tightly controlled
- The `WHERE` clause is ***REALLY*** important
  - Unless you totally want to dump every row in your table

DELETE Syntax

```
DELETE FROM sometable
    WHERE somefilter;
```

Example:

```
DELETE FROM patients
    WHERE patient_id = 1245;
```

Cascading deletions

- Referential integrity constraints, like foreign keys, often have rules about deletion
- Often, the deletion ***cascades***, or propagates through the database
- For example: you delete a row in your patient table, and all that patient's records are also deleted

Bypassing constraints

- Don't. If they exist, they're probably there for a reason.
- Most RDBMSs will let you bypass them anyway, through various methods
  - The process can be slow - they need to check many constraints, after all
- One which bears mentioning: `TRUNCATE TABLE`

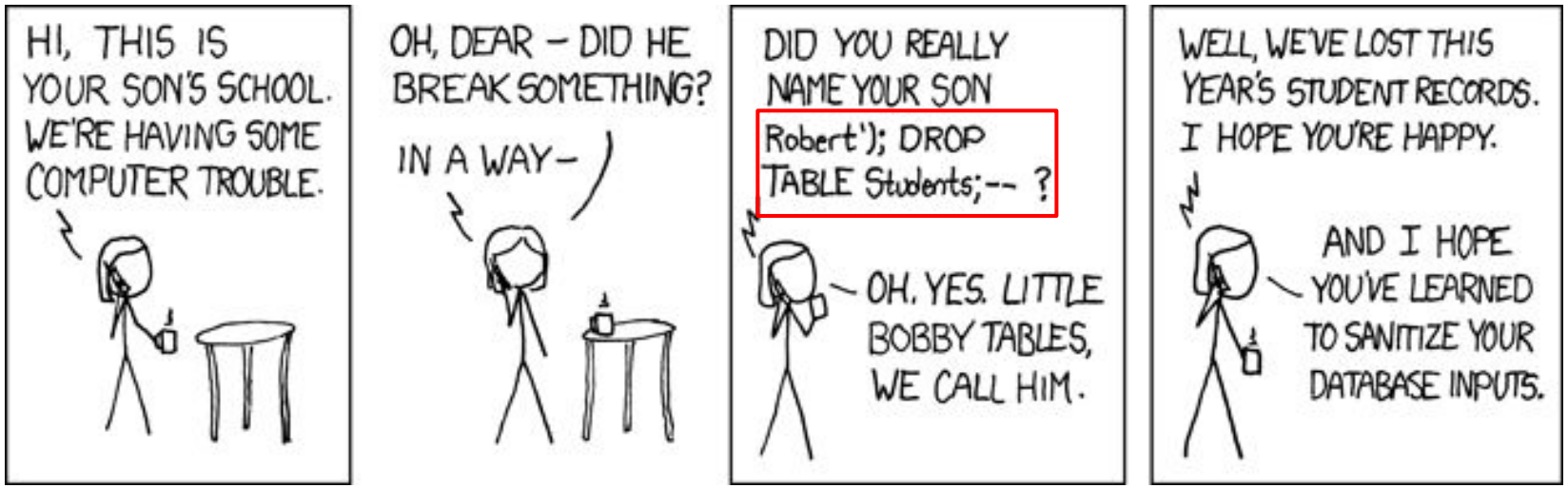
TRUNCATE TABLE

- For those tables that you really hate
- `TRUNCATE TABLE` essentially drops every row of the table instantly
  - Generally, it bypasses checking referential integrity constraints
- Useful if your table does not have any constraints anyway - often quite snappy

CREATE ALTER DROP

- `CREATE TABLE` makes a new table
- `ALTER TABLE` changes an existing table
- `DROP TABLE`, well, drops a table
  - Sounds familiar...

Revisiting the xkcd comic



You should now know a little bit about why this is funny!

Notes on tables

- We will not go into great depth on creating, manipulating, or deleting tables
  - Fairly straightforward in documentation
- Most of this relates to making good data structures, applying constraints, or rules for propagating updates and deletes
- But one thing that is useful...***temporary tables***

Temporary tables

- A ***temporary table*** is a table which only exists for the duration of your session
  - Disappears when you log out
- You can use these to stage data, store the results of queries, or otherwise manipulate data
  - Can help to make your code shorter!

Creating your temp table

- Temporary tables can be defined using the same basic syntax as regular tables
- Instead of `CREATE TABLE`, you use `CREATE TEMPORARY TABLE`
- This can be combined with `AS` to make easy new temporary tables...

Example Temp Table

```
CREATE TEMPORARY TABLE empno_10004_salaries
    AS
    (
    SELECT *
           FROM salaries
           WHERE emp_no = 10004
    );
```

```
mysql> SELECT * FROM salaries WHERE emp_no = 10004;
+--------+--------+------------+------------+
| emp_no | salary | from_date  | to_date    |
+--------+--------+------------+------------+
|  10004 |  40054 | 1986-12-01 | 1987-12-01 |
|  10004 |  42283 | 1987-12-01 | 1988-11-30 |
|  10004 |  42542 | 1988-11-30 | 1989-11-30 |
|  10004 |  46065 | 1989-11-30 | 1990-11-30 |
|  10004 |  48271 | 1990-11-30 | 1991-11-30 |
|  10004 |  50594 | 1991-11-30 | 1992-11-29 |
|  10004 |  52119 | 1992-11-29 | 1993-11-29 |
|  10004 |  54693 | 1993-11-29 | 1994-11-29 |
|  10004 |  58326 | 1994-11-29 | 1995-11-29 |
|  10004 |  60770 | 1995-11-29 | 1996-11-28 |
|  10004 |  62566 | 1996-11-28 | 1997-11-28 |
|  10004 |  64340 | 1997-11-28 | 1998-11-28 |
|  10004 |  67096 | 1998-11-28 | 1999-11-28 |
|  10004 |  69722 | 1999-11-28 | 2000-11-27 |
|  10004 |  70698 | 2000-11-27 | 2001-11-27 |
|  10004 |  74057 | 2001-11-27 | 9999-01-01 |
+--------+--------+------------+------------+
16 rows in set (0.00 sec)
```

```
mysql> CREATE TEMPORARY TABLE empno_10004_salaries AS (SELECT * FROM salaries WHERE emp_no = 10004);
Query OK, 16 rows affected (0.00 sec)
Records: 16  Duplicates: 0  Warnings: 0

mysql> SELECT * FROM empno_10004_salaries;
+--------+--------+------------+------------+
| emp_no | salary | from_date  | to_date    |
+--------+--------+------------+------------+
|  10004 |  40054 | 1986-12-01 | 1987-12-01 |
|  10004 |  42283 | 1987-12-01 | 1988-11-30 |
|  10004 |  42542 | 1988-11-30 | 1989-11-30 |
|  10004 |  46065 | 1989-11-30 | 1990-11-30 |
|  10004 |  48271 | 1990-11-30 | 1991-11-30 |
|  10004 |  50594 | 1991-11-30 | 1992-11-29 |
|  10004 |  52119 | 1992-11-29 | 1993-11-29 |
|  10004 |  54693 | 1993-11-29 | 1994-11-29 |
|  10004 |  58326 | 1994-11-29 | 1995-11-29 |
|  10004 |  60770 | 1995-11-29 | 1996-11-28 |
|  10004 |  62566 | 1996-11-28 | 1997-11-28 |
|  10004 |  64340 | 1997-11-28 | 1998-11-28 |
|  10004 |  67096 | 1998-11-28 | 1999-11-28 |
|  10004 |  69722 | 1999-11-28 | 2000-11-27 |
|  10004 |  70698 | 2000-11-27 | 2001-11-27 |
|  10004 |  74057 | 2001-11-27 | 9999-01-01 |
+--------+--------+------------+------------+
16 rows in set (0.00 sec)
```

Example Temp Table Results

The power of `CREATE + AS`

- There are many options when creating tables
  - SQL wants you to define data types, constraints, et cetera...
- `CREATE TABLE foo AS` ... lets you bypass a lot of those steps
  - Useful when you just need a place to stash stuff
- Does not transfer constraints from tables

Getting DDLs

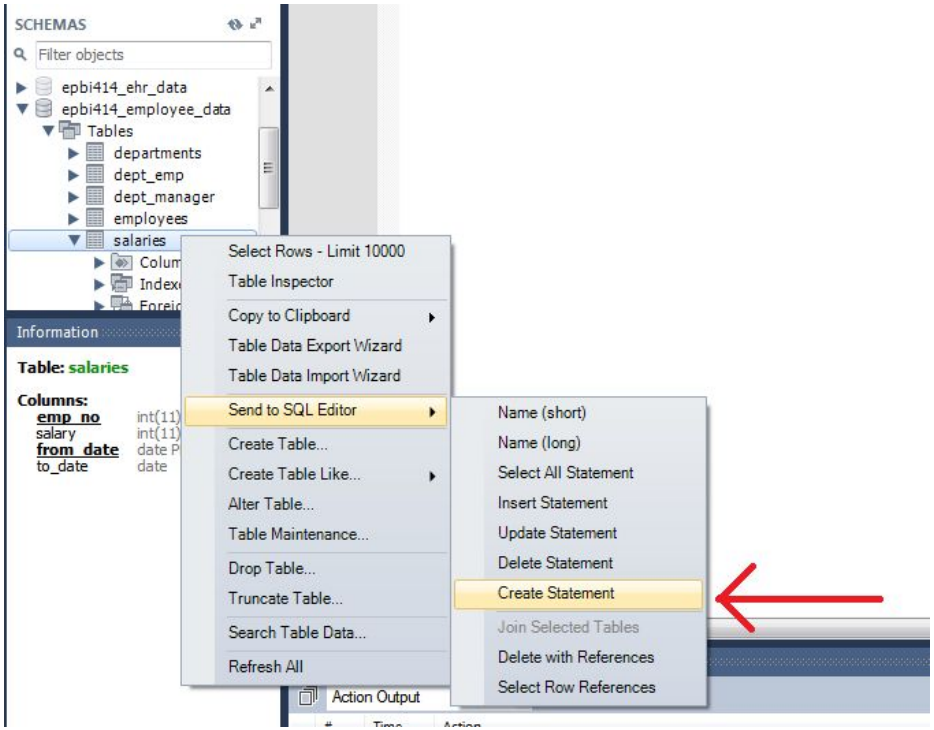
- One of the great things about RDBMSs is that structure is stored with the data
- Data structures are created using something called a ***DDL***, or ***data definition language***
  - In our case, it's just part of SQL
- You can retrieve the DDL statement used to create an object in the database

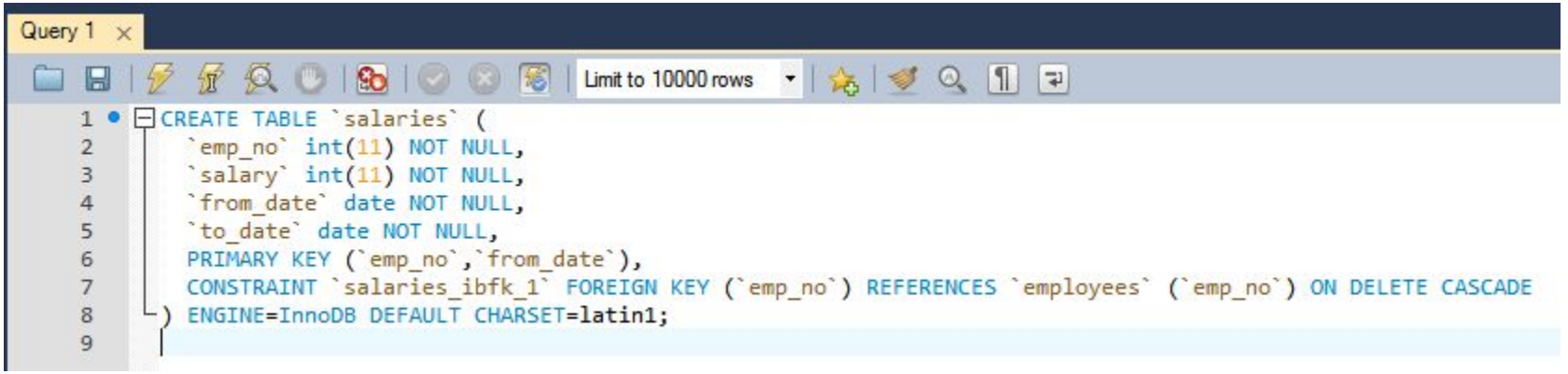
Two methods

- Here are two methods to get the DDL statement used to create an object:
  - `SHOW CREATE TABLE` in command line (this might be MySQL specific)
  - Using MySQL Workbench

```
mysql> SHOW CREATE TABLE salaries;
+----------+--------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------+
| Table    | Create Table
                                                                                              |
+----------+--------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------+
| salaries | CREATE TABLE `salaries` (
  `emp_no` int(11) NOT NULL,
  `salary` int(11) NOT NULL,
  `from_date` date NOT NULL,
  `to_date` date NOT NULL,
  PRIMARY KEY (`emp_no`,`from_date`),
  CONSTRAINT `salaries_ibfk_1` FOREIGN KEY (`emp_no`) REFERENCES `employees` (`emp_no`) ON DELETE CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=latin1 |
+----------+--------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------+
1 row in set (0.00 sec)

mysql>
```

Using the command line





Using MySQL Workbench

Closing notes, 1

- We are just brushing the tip of what you can do in SQL
  - MySQL is only one RDBMS too...
  - Consider PostgreSQL for a more sophisticated platform
- Your careers will largely start with consuming data in SQL
  - But you never know where you will end up

Closing notes, 2

- SQL can be very fast!
  - Sometimes, it is a lot faster to do work in SQL than it would be in other languages
- SQL is not perfect
  - Just because you can do it in SQL doesn't ***necessarily*** mean you should
  - Time tradeoff, maintenance tradeoff

Attributions

1. "Is It Worth the Time?" xkcd. https://xkcd.com/1205/.
   Licensed under Creative Commons Attribution Non-Commerical 2.5. Randall Munroe, author.

2. "Optimization" xkcd. https://xkcd.com/1691/.
   Licensed under Creative Commons Attribution Non-Commerical 2.5. Randall Munroe, author.

3. "Exploits of a Mom" xkcd. https://xkcd.com/327/.
   Licensed under Creative Commons Attribution Non-Commerical 2.5. Randall Munroe, author.