Head First: SQL

01: Introduction to SQL

Data and tables

A computer with sticky notes on it

Imagine a jumble of sticky notes. Each note has information about one of your friends; it might look like this:

```
-----------------------
| Sticky note #1      |
| ------------------- |
| Name: Sam Dibb      |
| Address: Sunderland |
| Occupation: Sales   |
| Tel: 0191 5555 222  |
|                     |
=======================
```

It will fast become unorganised, and we can do better.

1. Look for **similar data types**, or patterns
2. Chunk your data into **categories**
3. What **label** might you give your *category*?
4. What **label** might you give your *sticky note*?

Create a table

> *A database contains tables; a table contains columns and rows.*

Once we've chunked our data, it might look something like this:

```
| database_name |

    | first_name | second_name | address    | occupation | telephone  |
    | ---------- | ----------- | ---------- | ---------- | ---------- |
    | Sam        | Dibb        | Sunderland | Sales      | 0191555222 |
    | ...        | ...         | ...        | ...        |            |
```

1. Your *categories* become *column* names
2. Your *friend* becomes a *row* in the table

Tables

- All tables in a database should be connected in some way

Columns

- A column describes the data with a label
- It should be descriptive and clearly explain the type of data
- Often referred to as *field*

Rows

- A row is a *set of columns* that describe an *object*, or *thing*
- The columns can be thought of as an *objects attributes*
- Often referred to as *record*

Creating a database

```
CREATE DATABASE database_name;
```

Rename your database

```
ALTER DATABASE database_name RENAME TO new_database_name;
```

Always end your statement with a semi-colon `;`!

Creating a table

```
CREATE TABLE table_name (
  column_one [datatype], column_two [datatype]
);
```

When creating your table and columns, it's good to ask yourself:

1. What `column_name` best describes this piece of data?
2. What type of data is your data chunk (or category)?

Data types

> *It's important to make sure your data chunk uses the correct data type*

- There's lots of standard and fancy data types to choose from
- Each datatype has different *functions* it's allowed to use

- A type acts as a validation rule for each data type
- Take care to use an appropriate limit for speed and storage space

Other things to note:

- Check your types, they might be different in your RDBMS
- See postgres datatypes for documentation

Example datatypes

Using our `create table` example above, we could add the following data types to create our friends' table from our sticky note:

```
CREATE TABLE sticky_note (
  first_name VARCHAR(20),
  second_name VARCHAR(20),
  address VARCHAR(100),
  occupation VARCHAR(30),
  telephone INT(11)
)
```

Deleting a table

You need to drop that sucker!

You can't recreate an existing table or database:

```
database_name=> CREATE TABLE my_contacts ( ... );
ERROR:  relation "my_contacts" already exists
```

Instead, you'll need to `DROP` the table:

```
DROP TABLE my_contacts;
```

And recreate it with the `CREATE TABLE` sql command

Adding data to a table

Insert your ('value') into the table!

```
INSERT INTO table_name (column_name, column_name, ...)
VALUES ('value', 'value', ...);
```

1. `'Value'` must be in the same order as their `column_name`
2. `'single quotes'` for all text and character types
3. No comma after last `column_name`
4. No comma after last `value`

You can leave out `(column_name, ...)`, but values should *exactly* match your table structure:

```
INSERT INTO table_name
VALUES ('value_of_col_1', 'value_of_col_2', ...);
```

Or, you can leave out some column `'values'`, but you *must* specify the `column_names` you *are* including

Insert multiple records (or rows)

```
INSERT INTO table_name (column_name, column_name, ...)
VALUES ('value', 'value', ...),  -- Separate with a comma
       ('value', 'value', ...);  -- Value lists must all be the same length
```

Simply add more rows in the `VALUES` statement, separated with a comma `,`.

Default values

The default value is `NULL`. If you create a record with empty column values, that's what you'll get. You can think of it as being an *undefined* entry.

```
CREATE TABLE no_empty_spaces (
  spade VARCHAR(10),            -- can be left blank
  bucket VARCHAR(10) NOT NULL   -- a bucket always has to be full!
)
```

Some columns should *always* have values. To do this, you use `NOT NULL`, which forces you to fill the column for a record.

Which should I use?

> *If the data is clean and complete it's easier for you to analyse later.*

1. Will you need to search a row by this field?
    - Use `NOT NULL`
2. Or will the data need to be filled in later?
    - Use `NULL`
3. How important is it that the data is there?

People are lazy …

We can make it easier for people to fill in an entry, by including a `DEFAULT` value:

- If the value is usually this one, set it as a default
- The default must be the same type as the column
- If it's important to get the value right, use `NULL` or `NOT NULL`

```
CREATE TABLE doughnut_list (
  doughnut_name VARCHAR(10) NOT NULL,
  doughnut_type VARCHAR(8) NOT NULL,
  doughnut_cost DECIMAL(3,2) NOT NULL DEFAULT 1.00  -- Total 3 digits: 1 full, 2 decimal
)
```

The results would look something like this:

```
doughnut_name | doughnut_type | doughnut_cost
---------------+---------------+---------------
krispy        | jammy         |          2.00
dunkin        | iced          |          1.00
robs          | dusted        |          1.00
```

02: Displaying your data

SELECT statement

```
SELECT * FROM my_contacts;  -- * is a wildcard for "all columns"
```

Displays all records from the table `my_contacts`.

Limit the results

```
SELECT first_name, location FROM my_contacts;
```

Returns only the columns you'd like to view — it also *speeds up* the query!

WHERE statement

Say we'd like to display someone specific, we need to list all the `Annes` from our contacts table. We could search through the entire table, but it's easier to narrow our search:

```sql
SELECT * FROM my_contacts
WHERE first_name = 'Anne';  -- first_name is Anne
```

- If any rows match contain `'Anne'` in `first_name`, it returns all data for that row
- If there's no match, the row isn't returned

Comparison operators

- < less than
- > greater than
- <= less than or equal to
- >= greater than or equal to
- = equal
- <> or `!=` not equal

Matching strings

You can also use comparisons for characters:

```sql
SELECT * FROM my_contacts
WHERE first_name > 'A';  -- Returns values beginning with `B`—`Z`

SELECT * FROM my_contacts
WHERE first_name >= 'C';  -- Returns values (including `C`), so `C`—`Z`
```

Combining your queries

```sql
SELECT email FROM my_contacts
WHERE profession = 'Computer programmer';
```

The `SELECT` columns are the ones displayed — you can use whatever columns you like in the `WHERE` query to refine your results: mix and match!

```sql
SELECT first_name, email FROM my_contacts
WHERE location = 'San Fran, CA'
  AND gender = 'F'
  AND first_name = 'Anne';
```

You can mix character and numeric values too …

```sql
SELECT easy_drinks FROM drinks
WHERE main = 'soda'
  AND amount1 > 1;
```

Other expressions

```sql
-- More than one expression
expression AND expression
expression OR expression
```

NULL values

You cannot select a `NULL` value directly

```
-- This WILL work ...
SELECT doughnut FROM doughnut_list
WHERE doughnut IS NULL;
-- but this WILL NOT work
SELECT doughnut FROM doughnut_list
WHERE doughnut = NULL;  -- NULL is not equal to anything
```

However, if you use `WHERE` on non-empty values, a `NULL` value will show:

```
SELECT type_of_doughnut FROM doughnut_ratings
WHERE location = 'Krispy King' OR rating > 5;
```

```
type
--------------
plain glazed
plain glazed
plain glazed

jelly
(5 rows)
```

Other options

You can display values that *are* NULL or only values that *are not* NULL.

```
-- NULL is not equal to anything
expression IS NULL
expression IS NOT NULL
```

LIKE statement

When you want to find a value that looks like something

You can search for a value `LIKE` another one. This comes in handy when:

- There might be typos or errors
- There might be many similar named values

- You might want to search *within* text

Wildcard rules

1. `%` stands for *any number of characters*
2. _ stands for *one single character*

```
SELECT first_name FROM my_contacts
WHERE first_name LIKE '%nne';
```

Returns both `Anne` and `Roseanne` ...

```
SELECT first_name FROM my_contacts
WHERE first_name LIKE '_nne';
```

But this would only return `Anne`, or other words with a single character.

BETWEEN statement

```
-- Order matters!
value BETWEEN low AND high      -- same as `a >= x AND a <= y`
value NOT BETWEEN low AND high  -- same as `a < x OR a > y`
```

Example with numbers:

```
-- Includes endpoint values in the range
WHERE count BETWEEN 10 AND 30;  -- 10–30
```

Example with characters:

```
-- Does not include endpoint values in the range
WHERE string BETWEEN 'r' and 'm';  -- 'r' to 'l'
```

IN statement

If you're searching for a list of strings, you could do this:

```
SELECT name FROM little_black_book
WHERE rating = 'fantastic'
   OR rating = 'hot'
   OR rating = 'smart';
```

But it's far more concise to use `IN`. You pass a list of strings, instead of multiple statements:

```
SELECT name FROM little_black_book
WHERE rating IN ('fantastic', 'hot', 'smart');
```

NOT statement

You can return values that are `NOT` the same as your statement. Combine it with `WHERE` to make the magic happen! There's two ways to write it:

1. `WHERE NOT [expression] [value]`

2. `WHERE [expression] NOT [value]`

```
SELECT * FROM drinks
WHERE NOT price > 6;
-- is the same as
SELECT * FROM drinks
WHERE price NOT > 6;
```

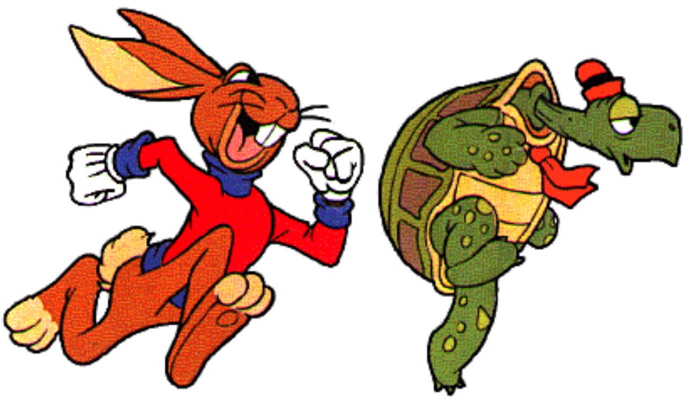
Here's some other examples:

```
-- Range
WHERE price NOT IN (5, 6, 8);
-- Not empty
WHERE price IS NOT NULL;
-- Multiple expressions
WHERE price NOT > 6
  AND name LIKE 'fiz%';
```

Be careful: It's often easier to use standard expressions rather than `NOT`!

03: Delete and update

Order! Order!



Who came first?

If you don't have a column that *describes* the order, it's impossible to tell who came first. You can't guarantee that rows are in any logical order; for that to happen you'd need to add:

- A column to describe the order (numerical)
- A column to describe the time (chronological)

So it's possible to add values that describe order. But, should you?

- Do you need historical data?
- Do you need to track over time?
- Or, do you just need the last record?

If you don't need historical data, keep it simple.

DELETE statement

You can use a `DELETE` statement to remove a single, or multiple rows.

- Use it in combination with `WHERE`
- It's similar to your `SELECT` statements!

```
-- Delete all Zippo's rows
DELETE FROM clowns_info
WHERE first_name = 'Zippo';
-- Delete only the one we want
DELETE FROM clown_info
WHERE first_name = 'Zippo'
  AND activities = 'dancing, singing';
```

Important: Make sure you always include a `WHERE` statement, or you'll delete *all* your rows!

DELETE errors

Fizzy pop rots your teeth. A bad `DELETE` statement rots your data!

It's a good idea to double (and triple!) check your `DELETE` statements before running them. Make sure you're only deleting the rows you want:

- A typo could give unintended consequences …
- Ditto for records that share values …
- Or the order you run them!

Don't rot your data!

Imagine you need to change the prices of two cans of coke:

```
-- {'pepsi': 1.00, 'coca cola': 1.50}

-- Update the pepsi price
INSERT INTO fizzy_drinks
```

```
VALUES ('pepsi', 1.50);
-- Now delete the old value
DELETE FROM fizzy_drinks
WHERE price = 1.50;
```

But wait … now we have two cans of coke the same price:

- if we delete the `1.50` drinks ..
- we'd accidentally delete both our cans of coke

SELECT-INSERT-DELETE

To avoid mistakes, use a pencil first!

If in doubt, use SELECT first

`DELETE` is final. The `WHERE` clause is exactly the same for `SELECT` *and* `DELETE` — so to make sure you don't delete anything you'll regret:

1. **Try `SELECT` first**!
2. Then `INSERT` the new record
3. Finally, `DELETE` the old record

Now you can be confident you won't mess up!

UPDATE statement

It's easier to update than do things manually

The `UPDATE` statement simply "updates" the value you want, in the place you need. It's much easier than the *SELECT-INSERT-DELETE* combo:

- You `SET` a `column_name` to (equal) a new `'value'`
- You use `WHERE` to get specific — just like `SELECT` and `DELETE`!

```
-- Original: {'pepsi': 1.00}
UPDATE fizzy_pop
   SET price = 1.50
 WHERE name = 'pepsi';

-- Result: {'pepsi': 1.50}
```

More than one column (or row)

You can set multiple columns, and multiple rows (depending on your `WHERE` statement)

```
UPDATE fizzy_pop
   SET price = 1.50,
       name = 'super pepsi'
 WHERE name = 'pepsi';
```

Remember our fizzy pop dilemma?

You also need to be careful about order in an `UPDATE` statement:

```
-- {'pepsi': 1.00, 'coca cola': 1.50}

-- Update the pepsi price
UPDATE fizzy_pop
   SET price = 1.50
 WHERE price = 1.00

-- Now update the price of coca cola
UPDATE fizzy_pop
   SET price = 2.00
 WHERE price = 1.50

-- {'pepsi': 2.00, 'coca cola': 2.00}
```

WTF happened there?

1. When we changed `'pepsi'`, it's now the same price as `'coca cola'`
2. So our second `UPDATE` now also targets `'pepsi'`. Oops!
3. The solution is to *reverse the order* (highest price first)

There's an easier way …

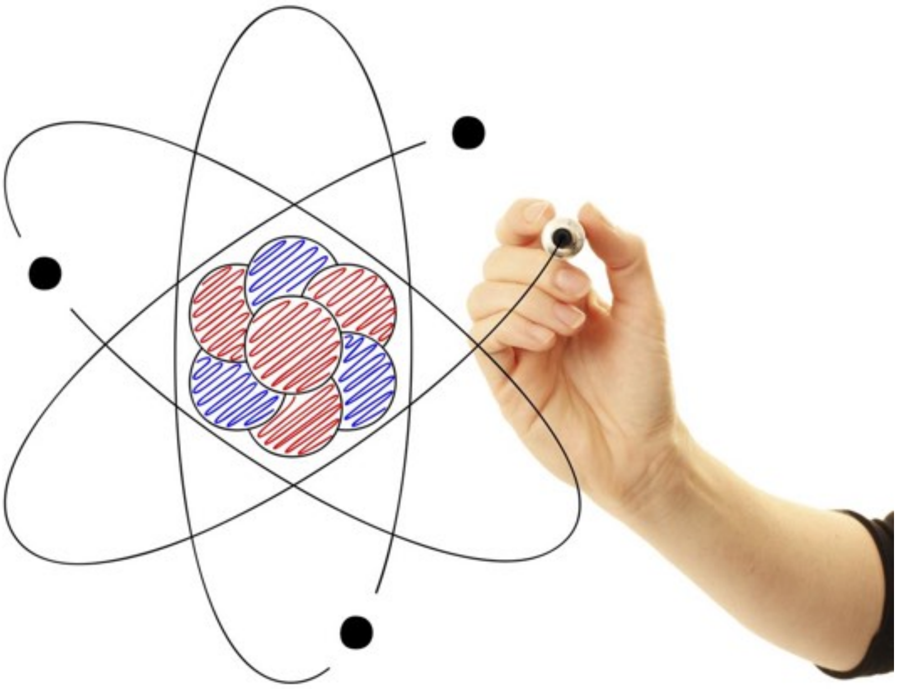
We can use simple maths on numeric values — throw in an `OR` and we can do it all in one simple `UPDATE`!

```
-- {'pepsi': 1.00, 'coca cola': 1.50}
UPDATE fizzy_pop
   SET price = price + 0.50
 WHERE name = 'pepsi'
    OR name = 'coke';
-- {'pepsi': 1.50, 'coca cola': 2.00}
```

Important: Make sure you always include a `WHERE` statement, or you'll update *all* your rows!

04: Smart table design

Atomic data



Make your data atomic for the best results!

The best database designs are **simple**, **specific**, and **fast**. In previous chapters, our table rows look something like this:

```
-[ RECORD 1 ]-------------------------------------
name       | Elsie
last_seen  | Cherry Hill Senior Center
appearance | F, red hair, green dress, huge feet
activities | balloons, little car
```

- What happens if we want to search for records with `'red hair'`?

- What happens if we want to search for records driving a `little car`?

We could use `LIKE`, but it's better to make our data *atomic*.

Break your data into smaller chunks

To make your data atomic, ask yourself the following questions:

1. What is **one** thing you want your table to describe?
2. What do you need to know about the thing? Make a list!
3. Break that down into the smallest chunks of data you need.
4. How might you label that data for your columns?
5. What type of data is it?

What's the focus? What's it's purpose? Think about the chunks of data you'll need:

1. Think about the chunks you'd need to use for your `SELECT` query
2. Think about the chunks you'd like to display as a result
3. Who's using or accessing the data?
4. What do they need? How will they search?

Only use what you need!

Smaller is better! Always build your tables in the easiest way possible, using only the information you'll need:

- Only use the chunks you actually need
- Only break them down to the smallest chunk you'll need …
    - *Don't* use anything smaller just because you can
- Throw away anything you don't need

What shall I name my atomic data?

- Use descriptive labels, like `last_seen`
- Make it short and easy to write (for your queries)

Atomic data rules

1. Each column must contain only one type of data
2. You can't have multiple columns with the same type of data

Let's fix our record

> *So, I need to find a clown who's name is Elsie, where was she last seen? What was she doing? What was her activity at the time?*

Take our example record above. Write a description like the one above — ask yourself questions! Now, chunk the main points:

- Elsie is a clown with a `name`
- We want to know where she was `last_seen`
- We want to find out by searching her `activity`

It's far easier to answer "who's driving a `'little car'`?" in a query, if we made that chunk atomic:

```
-[ RECORD 1 ]-------------------------------------
```

```
name       | Elsie
last_seen  | Cherry Hill Senior Center
appearance | F, red hair, green dress, huge feet
activities | little car
```

You can see we've changed `activities` to have one, and only one value, `'little car'`. We could have made `appearance` atomic too, but we don't need to query that column.

But what if I want to make everything atomic?

If you wanted to go crazy and atomise *all* the data, you'd have to do something like:

```
-[ RECORD 1 ]-----------------------------------
name       | Elsie
last_seen  | Cherry Hill Senior Center
gender     | F
hair_color | red
clothing   | green dress
other      | huge feet
activities | little car
```

Normalisation

Normalisation simply means working with standard rules, making it easier for you and your team to work on together.

- It makes our database easier to search
- It makes our database smaller
- It makes our queries faster!

1NF

To achieve First Normal Form (1NF), your tables must follow these rules:

1. Each row of data must contain atomic values
2. Each row of data must be unique
3. Values stored in a column should be of the same type
4. No repeating types of data across columns

5. Columns should have unique names
6. Order of data stored doesn't matter

You're going to need a primary key

Just like Harry Potter, you need to find the primary key!

To make avoid duplicate rows you'll need a unique identifier, or **primary key**.

1. A primary key *must* be unique (to it's column)
2. A primary key can't be `NULL`
3. When you `INSERT` a record, the primary key *must* be given
4. A primary key can't be changed once set

There's 3 ways to find a primary key

1. Use an existing column in your table that *you're sure* is always unique
2. Create a new column and *manually* set a number for each row
3. Create a new column and *automatically* increment a number for each row

For most cases, you'll want to generate a new column with a unique *automatically* incrementing ID.

Adding a primary key column

Method 1: From scratch (or recreate table)

1. Export your data
2. Create a new table (with a primary key column)
3. Import the old data into the new table

```
CREATE TABLE table_name (
  id SERIAL PRIMARY KEY
  ...
);
```

```
-- Leave out the `id` column: it automatically populates
INSERT INTO table_name (...)
VALUES (...)
```

Method 2: ALTER TABLE statement

```
-- Add a primary key column
ALTER TABLE table_name
ADD COLUMN id SERIAL PRIMARY KEY;

-- Or, alter an existing column
ALTER TABLE table_name
ADD PRIMARY KEY (column_name);
```

Postgres primary key notes:

- `serial` is the equivalent of `auto_increment` …
- but there are other types and methods you can use too …
- currently there's no way to alter column order with Postgres.

05: Alter table design

ALTER TABLE statement

You can alter your table anyway you like!

You've already used the `ALTER TABLE` statement, but there's lots more it can do!!

```
ALTER TABLE table_name
-- Add a statement below to make the magic happen!
```

It goes great with …

Use it with the following statements. You can combine statements together, too:

Statement	Does this
`ADD [ COLUMN ] col_name [type]`	Adds a new column
`DROP [ COLUMN ] col_name`	Deletes a column (and *all* it's data)
`ALTER [ COLUMN ] col_name [type]`	Change a column type
`RENAME [ COLUMN ] col_name TO new_col_name`	Rename a *column* (no multiple columns)
`RENAME table_name TO new_table_name`	Rename a *table*

Tidy up names

```
----[ projekts ]----------------

     Column        |          Type          |
-------------------+------------------------+
number             | integer                |
descriptionofproj  | character varying(50)  |
contractoronjob    | character varying(10)  |
```

Let's take the following database as an example. There's lots that could be improved:

```
-- First, let's give the table a better name
ALTER TABLE projekts
RENAME TO project_list;
-- We can rename the columns too!
-- #1: Escape a keyword
ALTER TABLE project_list
RENAME COLUMN "number" TO proj_id;  -- #1
ALTER TABLE project_list
RENAME COLUMN descriptionofproj TO proj_desc;
ALTER TABLE project_list
RENAME COLUMN contractoronjob TO con_name;
-- We need more data! Let's add some columns
-- #2: Multiple statements allowed
-- #3: This is the same as `decimal(7, 2)`
ALTER TABLE project_list
ADD COLUMN con_phone varchar(11), -- #2
ADD COLUMN start_date date,
ADD COLUMN est_cost numeric;  -- #3
```

Changing data types

Before changing your data type, **watch out**!

- If your new data type *isn't* compatible, you'll get an error
- If your new data type *is* compatible and too long, it gets truncated (`'Bobby' -> 'B'`)

```
ALTER TABLE project_list
ALTER COLUMN proj_desc TYPE varchar(120);  -- Add more space
```

Adding a primary key

The wrong way

You'd think this would work …

```sql
-- Use `proj_id` (number) as auto-increment primary key
-- #1: Change type to serial
-- #2: Add the primary key
ALTER TABLE project_list
ALTER COLUMN proj_id TYPE SERIAL,
ALTER COLUMN proj_desc TYPE varchar(120),
ADD PRIMARY KEY (proj_id);
```

The right way

But it doesn't seem to. It's much easier to:

```sql
-- 1: Create `proj_id_new` column
-- 2: Make it a `serial` type
-- 3: And make it the primary key
ALTER TABLE project_list
ADD COLUMN proj_id_new serial PRIMARY KEY;
```

Removing a column

We no longer need the `start_date` and `proj_id` .. let's drop those suckers!

```sql
ALTER TABLE project_list
DROP COLUMN start_date,
DROP COLUMN proj_id;
-- Let's rename our new primary key too ..
ALTER TABLE project_list
RENAME proj_id_new TO proj_id;
```

Now our table is fully pimped!

```
 Column    |          Type          | Collation | Nullable |                     Default
-----------+------------------------+-----------+----------+---------------------------------------------------
proj_desc  | character varying(50)  |           |          | NULL::character varying
con_name   | character varying(10)  |           |          | NULL::character varying
con_phone  | character varying(11)  |           |          |
est_cost   | numeric                |           |          |
proj_id    | integer                |           | not null | nextval('project_list_proj_id_new_seq'::regclass)
```

Unfortunately, we can't do much about the order as Postgres won't let us do that easily (at the time of writing). It's possible, but fiddly and a bit risky.

String functions

String functions for Postgres

1. Look for patterns
2. Make sure the pattern is the same for all values
3. Figure out which function to use to get the job done!
4. Functions return the modified originals: they *don't change* the data

Basic functions	Does this
`left(str, n)`	Return first *n* characters in the string (see docs for `-`negative)
`right(str, n)`	Return last *n* characters in the string
`substr(string, from [, count])`	`substr('alphabet', 3, 2) -> 'ph'` (see docs)
`split_part(...)`	Split string to delimiter (see below, like MySQL `substring_index`)
`lower(string)`	Convert characters to lowercase
`upper(string)`	Convert characters to uppercase
`reverse(str)`	Return reversed string
`length(str)`	Return the length of a string (characters)
`ltrim(str)`	Return string minus empty space from left (more options in docs)
`rtrim(str)`	Return string minus empty space from right (more options in docs)

Let's get atomic again!

Let's atomise our `locations` from Greg's `my_contacts` table:

```
location
-------------------
Palo Alto, CA
San Francisco, CA
San Diego, CA
Dallas, TX
Princeton, NJ
Mountain View, CA
```

Split a string

- `split_part(string text, delimiter text, field int)`

```
-- 1. split location
-- 2. at the 1st comma
-- 3. return the first part (before 1st comma)
SELECT split_part(location, ',', 1) FROM my_contacts;
```

```
location
-------------------
Palo Alto
San Francisco
San Diego
Dallas
Princeton
Mountain View
```

Grab the county

- `right(str, n)`

```
-- 1. Grab 2 characters from the right
SELECT right(location, 2) FROM my_contacts;
```

```
location
-------------------
CA
CA
CA
TX
NJ
CA
```

Moving location

To finish off our new location we need to:

1. Combine our string Functions
2. Create two new columns, `city` and `state`
3. Use our old `location` value to `SET` our new columns
4. Delete our old `location` column

```
ALTER TABLE my_contacts
ADD COLUMN city varchar(30),
ADD COLUMN state char(2);
-- This will set ALL rows with new values
UPDATE my_contacts
SET city = split_part(location, ',', 1);
UPDATE my_contacts
SET state = right(location, 2);
-- Delete our old column
ALTER TABLE my_contacts
DROP COLUMN location;
```

06: Advanced select

CASE statement

When you want to switch genres, use CASE!

The problem

```
----[ movie_tables ]----------------

| movie_id | title | rating | drama | comedy | ...
+----------+-------+--------+-------+--------+ ...
| ...      | ...   | ...    | False | True   | ...
```

1. Currently *genre* headings are set to `True` or `False`
2. Some films belong to *more than one* genre
   - Which genre should these films be shelved?
   - Adding `True` or `False` is time consuming and error-prone

The solution

Dataville videos have decided to add a *category_type* column. You *could* add each film to the new category like this:

- If this genre is `True`, set the category_type to `category`

- `UPDATE table SET category_type = 'category' WHERE category = 'True'`

As we've seen before **order is important** — if some films have *more than one* genre set, it's `category` will be whichever `UPDATE` statement runs last.

There's a better way …

Instead of running multiple `UPDATE` statements, we can do it all at once with `CASE`:

```
UPDATE my_table
SET new_column =              -- This is the column we're changing
CASE                          -- Run the below conditions on each row
  WHEN column1 = somevalue1   -- Does current row's `column1` equal this value?
    THEN newvalue1            -- It does? Set `column1` to new value
  WHEN column2 = somevalue2   -- ...
    THEN newvalue2            -- ...
  ELSE newvalue3              -- if no conditions met, set any value (not NULL!)
END;                          -- end the case statement
```

Our new solution with CASE

```
UPDATE movie_table
SET category_type =
CASE
  WHEN drama = 'T' THEN 'drama'
  WHEN comedy = 'T' THEN 'comedy'
  ...
  ELSE 'misc'
END;
```

1. Only the first true condition will run!
2. If a film belongs to both `drama` and `comedy` …
3. The `drama` value will be set …
4. And will skip to the `END`

Be careful with your conditions!

"End of the line" has an **R** rating — but! It is also a **cartoon**. Our `CASE` statement is going to make mothers angry:

```
UPDATE movie_table
SET category_type =
CASE
  WHEN cartoon = 'T' THEN 'family'
  ...
END;
```

Let's add a check for our `rating` column:

```
-- Only add a film to "family" category if it has a `G` rating!
...
CASE
  WHEN cartoon = 'T' AND rating = 'G' THEN 'family'  -- Only if
  ...
END;
```

Where can I use CASE?

You can use `CASE` with `SELECT`, `INSERT`, `DELETE` and `UPDATE`!

ORDER BY statement

Our Dataville movies are getting messy. There are many ways to organise them:

1. By title
2. By rating
3. By category
4. A mix of these

To order by title, it'd look something like:

```
SELECT title, category_type
FROM movie_list
ORDER BY title;  -- Pick a column in the SELECT
```

- Lists in `ASC`ending order [A-Z, default]
    - Films with numbers come first
    - Characters listed alphabetically
- You can also list in `DESC`ending order [Z-A]

```
SELECT title, category_type
FROM movie_list
WHERE category = 'family'   -- Limit results to a genre
ORDER BY title DESC;        -- Reverse the list order
```

You can also pick the column with a number:

```
SELECT title, category_type
...
ORDER BY 1 ASC;  -- maps to `title`
```

ORDER BY multiple columns

Dataville want to see which films are too old:

```sql
SELECT title, category_type, purchased  -- #1
FROM movie_list
ORDER BY category_type, purchased;  -- #2, #3
```

1. Grab the date it was `purchased`
2. List *all* films, ordered by category
3. Then order them by purchase date

ORDER BY sorts from left to right

You'll notice films are ordered by *purchase date*, not alphabetically.

```
A --> Sort by category --> Sort by purchased { alpha, '2008', auto, '2012' }
|
Z --> Sort by category --> Sort by purchased { zelda, '2001', zappo, '2002' }
```

Let's add `title` as the last column to sort:

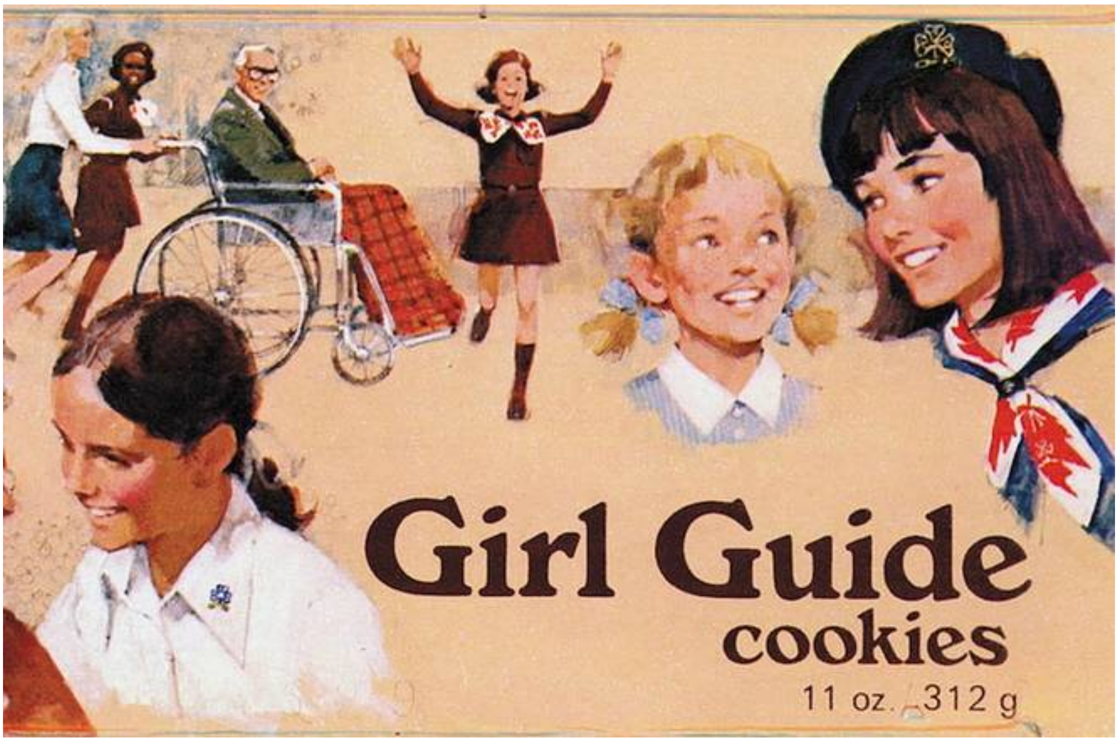
```sql
SELECT title, category_type, purchased
FROM movie_list
ORDER BY category_type, purchased, title;  -- #1, #2, #3
```

1. Order by `category_type`
2. Then by `purchase` date
3. Finally by their `title`

```
-- Categories beginning with A
-- #3                      #1              #2

| title                   | category_type | purchased |
+-------------------------+---------------+-----------+
| Greg: the untold story  | Action        | 2001      |
| Take it back            | Action        | 2003      |
| ...                     | ...           | ...       |
```

Aggregate functions



The girl guides are selling cookies. Who's in the lead?

```
id | first_name | sales | sale_date
----+------------+-------+------------
 1 | Lindsey    | 32.02 | 2007-03-12
 2 | Nicole     | 26.53 | 2007-03-12
 3 | Britney    | 11.25 | 2007-03-12
 . | ...        | ...   | ...
```

```sql
-- Find the highest individual sale
SELECT first_name, sales
FROM cookie_sales
ORDER BY sales;
```

```
--[ Results ]--

"Britney", 43.2
"Britney", 34.1
"Lindsey", 32.0
...
```

SUM

What's the total amount of sales for each girl guide?

```sql
-- We could group sales together by name ...
SELECT first_name, sales
FROM cookie_sales
ORDER BY first_name, sales;
```

```
--[ Results ]--

"Ashley", 26.8
"Ashley", 19.2
...
"Britney", 43.2
"Britney", 34.1
...
```

It's nicely organised but we have to manually calculate totals. Let's find the `sum()` of Britney's sales:

```
SELECT sum(sales)
FROM cookie_sales
WHERE first_name = 'Britney';  -- Only sum() Britney's sales
```

```
sum
---
107.91
```

SUM (using GROUP BY)

If we want to find the total of all our girl guides, we'd need to:

1. Find the `sum()` of each girl guides sales
   - By using the `sum()` function on `sales` column
2. Group each girl guide by their `first_name`
   - Which includes all her sales rows
   - Add each row for the current girl guide
   - Find her total with the `sum()` function (mentioned in step #1)
3. Order the results in reverse (largest first)
   - *You must use the same column / aggregate function!*

```
SELECT first_name, sum(sales)  -- #1
FROM cookie_sales
GROUP BY first_name            -- #2
ORDER BY sum(sales) DESC;      -- #3
```

```
-------------------------------------------[ order by sales ]----
A -> GROUP BY first_name -> Total sales -> | { "Britney", 107.91 }
|                                          | { "Nicole", 98.23 }
Z -> GROUP BY first_name -> Total sales -> | { "Ashley", 96.03 }
```

An easier way to order with aggregate functions

We can give the column (and it's aggregate function) an alias [**#1**] — similar to naming a variable. This will also rename our sales column in the output:

```
SELECT first_name, sum(sales) AS total_sales  -- #1
FROM cookie_sales
GROUP BY first_name
ORDER BY total_sales DESC  -- #1 ... Ditto!
```

```
--[ Results ]-----------

first_name | total_sales
-----------+------------
Britney    | 107.91
Nicole     | 98.23
Ashley     | 96.03
Lindsey    | 81.08
```

GROUP BY (with aggregate functions)

There's an easier way. `GROUP BY` allows us to group rows together by a column.

```
A -> GROUP BY column_name -> function(column_name) on this group -> Results
|
Z -> GROUP BY column_name -> function(column_name) on this group -> Results
```

AVG

We could also work out each girls average (over 7 days):

```
SELECT first_name, AVG(sales) AS average_sales  -- #2
FROM cookie_sales
GROUP BY first_name  -- #1
ORDER BY average_sales DESC;  -- #3
```

1. Group each girl guides sales by `first_name`
2. Calculate each girl's average sales (`sum(sales) / number of days`)
3. Order by our new `average_sales` column (highest first)

```
--[ Results ]----------------

first_name | average_sales
-----------+-----------------
Britney    | 15.4157142857143
Nicole     | 14.0328571428571
Ashley     | 13.7185714285714
Lindsey    | 11.5828571428571
```

MIN or MAX

We could also look for the *`min()`imum* or *`max()`imum* sales for each girl guide, to see which girl had the most (or least) sales on a single day:

```
SELECT first_name, MAX(sales) AS max_sale  -- Or replace with min()
FROM cookie_sales
GROUP BY first_name;
```

```
--[ Results ]----------------

first_name | max_sale
-----------+-----------------
Lindsey    | 32.02
Britney    | 43.21
Nicole     | 31.99
Ashley     | 26.82
```

COUNT

`count()` returns the number of rows in a given column:

```
SELECT COUNT(sale_date)  -- Will return number of rows (28)
FROM cookie_sales;
```

If we wanted to see how many days each girl sold cookies, we could try this:

```
SELECT first_name, COUNT(sale_date)  -- #1
FROM cookie_sales
WHERE sales > 0        -- #2
GROUP BY first_name;  -- #3
```

1. We don't have to include `first_name`, but let's display
2. Generally you'll want to remove `0` values from results
   - Only return days when `sales` were *more than* `0`
3. Grouping by `first_name` links `sale_date` rows to each girl

```
-- Daaamn girl, you still be on top
```

```
first_name | count
-----------+-------
Lindsey    |     6
Britney    |     7
Nicole     |     6
Ashley     |     6
```

COUNT and SELECT DISTINCT

We might like to know for sure how many days the girls were out selling cookies (even if they didn't make a sale!)

- We could `ORDER BY` the `sale_date`
  - But there might be tons of entries
  - And we might get the calulation wrong
- It's far easier to *select* all the *distinct* rows

```
--[ SELECT DISTINCT rows ]--

x -> | x
y -> | y
z -> | z
z -> |
y -> |
x -> |
```

```
SELECT DISTINCT sale_date  -- No duplicates!
FROM cookie_sales
ORDER BY sale_date;  -- Earliest date first!
```

```
sale_date
------------
2007-03-06
2007-03-07
2007-03-08
2007-03-09
2007-03-10
2007-03-11
2007-03-12
(7 rows)
```

We can use our `COUNT()` function to return the number of distinct rows!

```
SELECT COUNT(DISTINCT sale_date) -- DISTINCT goes inside the function
FROM cookie_sales; -- Only returns one value, no need to ORDER BY
```

Now let's add that our original count() for the girls:

```
SELECT first_name, COUNT(sale_date)
FROM cookie_sales
WHERE sales > 0
GROUP BY first_name;
```

- This will return exactly the same result as before (Britney wins again!) …
- But we can be certain it returns ONLY ONE of a particular `sale_date`
  - If a girl had two records (or rows) for the same day, one would be ignored
  - This helps us quickly see the variety of values in *any* column (ignoring duplicates)

COUNT() and DISTINCT — Another way!

For small tables, the above method works. In big tables, you're gonna have problems my friend …

```
SELECT COUNT(*)
```

```
FROM (
  SELECT DISTINCT sale_date
  FROM cookie_sales
) AS total_days;
```

This will give you exactly the same results, quicker. It's called an *inner select*.

LIMIT BY

```
--[ Total sales ]--

first_name | sales
-----------+-------
Britney    | 107.91
Nicole     | 98.23
Ashley     | 96.03
Lindsey    | 81.08
```

Fig. 1 — Displaying all girl guide's totals

As Britney has aced every test we've searched for so far, we need to find a runner-up.

```
SELECT first_name, SUM(sales) AS sales
FROM cookie_sales
GROUP BY first_name  -- #1
ORDER BY sales       -- #2
LIMIT 2;          -- #3
```

1. Remember, we have to group our rows first!
2. If we stopped here, it would give us `fig. 1` (above)
3. We only need the "Top 2" girls, our winner and runner up
    - To achieve this, we `LIMIT` the number of results we'd like
    - Giving us `fig. 2`, below

```
-[ RECORD 1 ]-------
first_name | Britney
sales      | 107.91
-[ RECORD 2 ]-------
first_name | Nicole
sales      | 98.23
```

fig. 2 — List the winner and runner up

LIMIT BY and OFFSET

We only really need our second result here, to find our runner-up girl guide. We can `OFFSET` the results to achieve this:

```
SELECT first_name, SUM(sales) AS sales
FROM cookie_sales
GROUP BY first_name
ORDER BY sales
LIMIT 1 OFFSET 1;  -- Limit to a single result, skip the first row
```

07: Multi table design

Outgrowing your table

Often, it's best to split your data into multiple tables!

It's likely you'll outgrow a single 1NF table, for any number of reasons, but the main ones being:

1. It's too hard to query specific results
2. Your queries are getting too complicated
3. You've got redundant, or repetitive data

So if you have a query like the one below — it's time to split data into more than one table!

```
-- Our contact
{
  'id': 341,
  'first_name': 'Moore',
  'interests': 'animals, horseback riding, movies',
  ...
}
-- Find him a lady with the same interests
SELECT * FROM my_contacts
WHERE gender = 'F'
AND status = 'single'
AND state = 'TX'
AND seeking LIKE '%single M%'
AND birthday > '1970-08-28'
AND birtday < '1980-08-28'
AND interests LIKE '%animals%'
AND interests LIKE '%horse%'
AND interests LIKE '%movies%';
-- Sooo many conditions :(
```

Letting your table do the work

```
--[ bad design ]--

contact | interest_1 | interest_2 | ...
--------+------------+------------+----
1       | True       | False      | ...
2       | False      | True       | ...

--[ good design ]--

contact | interest
--------+---------
1       | 5
2       | 6

          id  | interest
          ----+---------
          ... | ...
          5   | cycling
          6   | films
          ... | ...
```

In the example above, we list each contact's interests. In a real world example, each contact would have multiple interests, so we need to figure out a sane way of structuring that information — making sure we can easily `SELECT` any contact we want based on this information.

Bad table design	Good table design
complex `selects`	easy `selects`
bad matches	correct matches
search too complex	it just works
hacks/workarounds	

Database schema

Atomic data (again)

Remember our atomic data rules?

- Only use the chunks you actually need
- Only break them down to the smallest chunk you'll need …
  - *Don't* use anything smaller just because you can
- Throw away anything you don't need

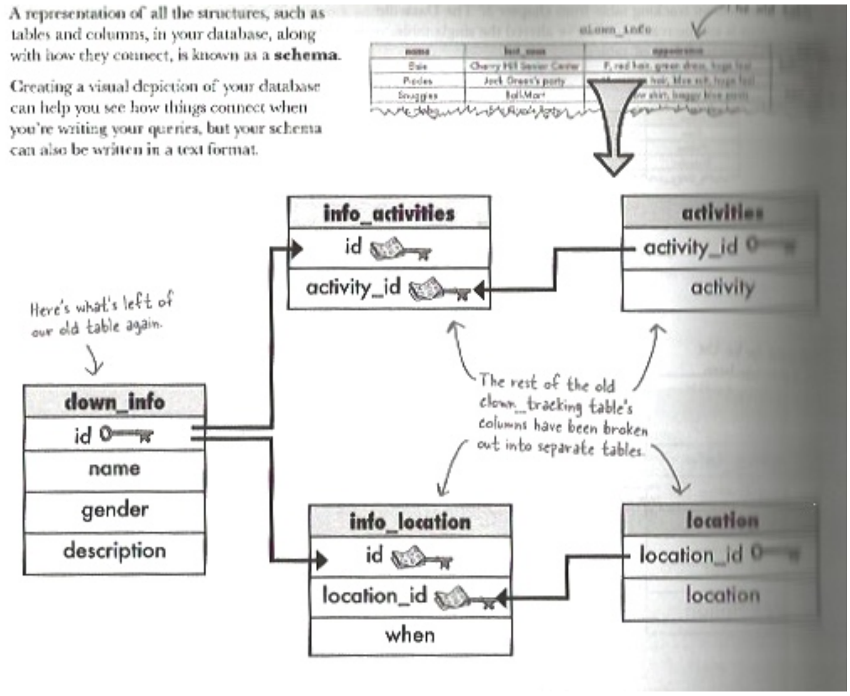
Well, our needs have outgrown our simple atomic data, so we'll need to change this:

```
--[ Clown info ]--

id | attributes
---+-----------
1  | red hair, green dress, huge feet
```

Into a properly formatted table, further atomising `attributes` data into *rows*.

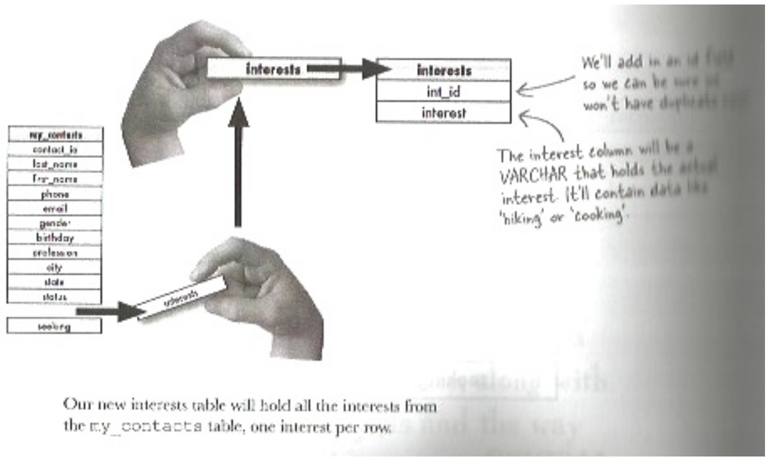
What's a schema?



A **schema** helps us understand the relationships between the data:

1. A description of the data (columns, tables)
2. How the objects relate to each other
3. How the columns and tables connect together

Database diagram



You can view your database as objects (tables), and lists (columns)

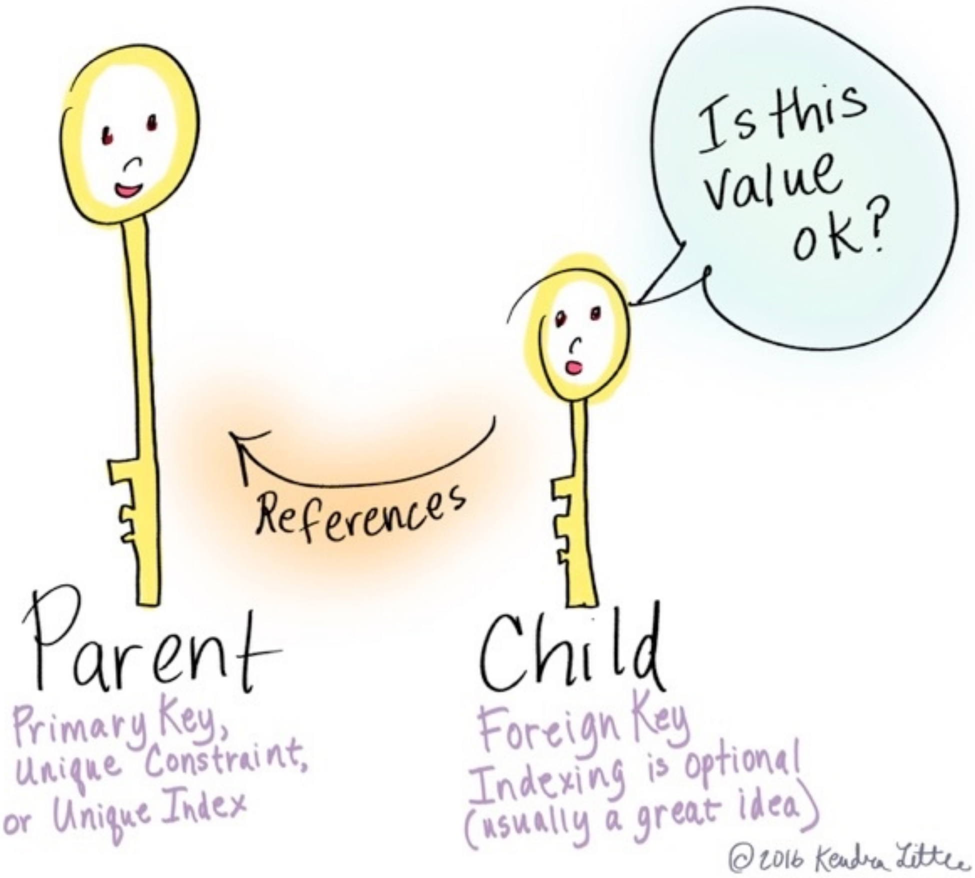
1. Select the non-atomic data you'd like to split out
2. Create a new object (table)
3. Create attributes for the table (columns)
4. Make sure you add an `id` field for your attribute rows

Let's take `gregs_list` as an example. Once we've created our new table, we need to find a way to link the interests to the person.

- It's generally best to use the **id** (in this case, *primary key*)
- This will link our `interests` table with a person's `contact_id`
- The column you link to should be **unique**
- This unique link is called a **foreign key**

A *foreign key* is a column that links a *primary key* of another table. This primary key is now the *parent key* of the `interests` table. The `my_contacts` table is the *parent table*.

Constraint



You're going to need to constrain your child!

A constraint **only** allows you to add values that exist in the parent table. It's also called *referential integrity*, which helps you avoid breaking your table, avoids "dead" data, keeping your database speedy.

- The value you add into your *foreign key* must already exist in your *primary key*

Create our table with a foreign key

```
CREATE TABLE interests (
  int_id SERIAL PRIMARY KEY,
  interest varchar(50) NOT NULL,
  contact_id int NOT NULL              -- #1
  REFERENCES my_contacts (contact_id)  -- #2
);
```

1. Create a field for our foreign key
2. Link it to our primary key table
    - `(contact_id)` is optional

The `FOREIGN KEY` statement is only needed in Postgres when creating a foreign key with more than one column.

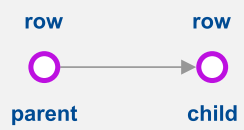
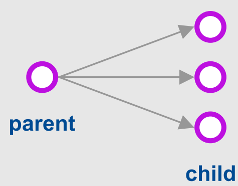
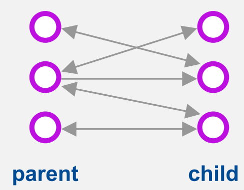
Deleting a row

If you try to delete a foreign key row, within a parent table, you're going to have problems my friend.

- You *must* delete it's linked child table rows first.

Table relationships

How do your tables relate to each other?

one-to-one	one-to-many	many-to-many
		
Exactly one row of a parent table is related to one row of a child table	A parent record has many matching child records; a child table can only match *one* parent record	Many of these records, match many of those
e.g social security number	**e.g** many people share the same profession	**e.g** many women own the same pair of shoes, many shoes are owned by the same women

Which data pattern to use?

One-to-one

You won't use *one to one* data very often, as it generally makes sense to leave it in your primary table. It may make sense if:

1. You want to write faster queries; you only need the search data in the child table
2. You have an object with values you don't yet know. Splitting it out into a separate table avoids polluting your primary table with `NULL` values
3. Restrict data that's sensitive (you don't want other querying)
4. A large piece of data, splitting out can speed up queries

One-to-many

You'll use this regularly. Any data that is specifically linked to one entry in your main table:

1. Customer address
2. A profession
3. …

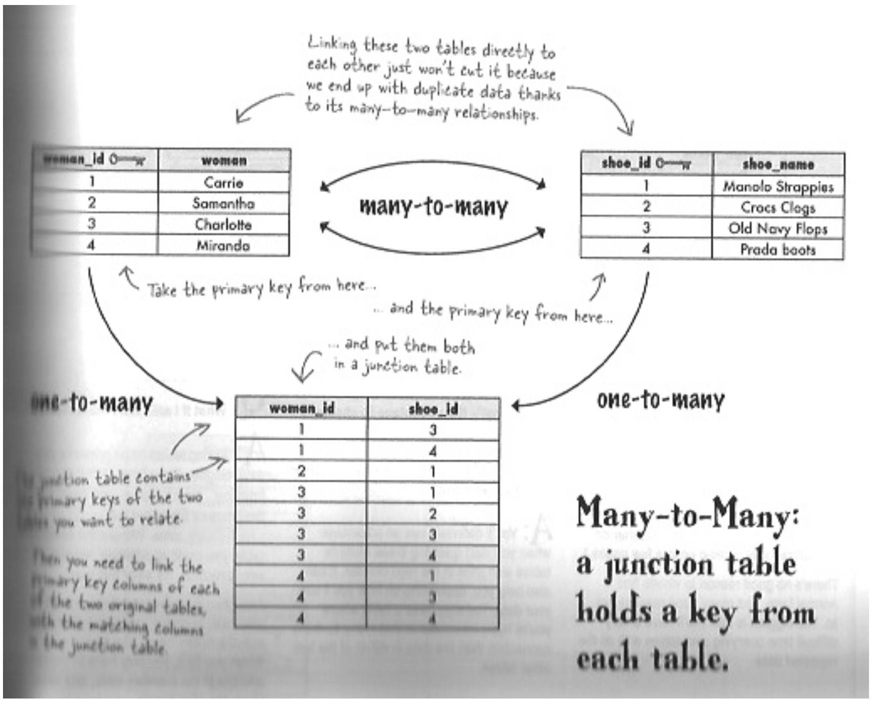
Many-to-many

This can be useful for relating two-way relationships:

1. Books and their authors
2. Friendships
3. Ingredients and recipes
4. Customers and products

You'll probably need a junction …

A junction table holds a key from each table. This way you can link a primary table and a child table, while avoiding duplicate entries.



- It helps stick to 1NF
- Helps with data integrity

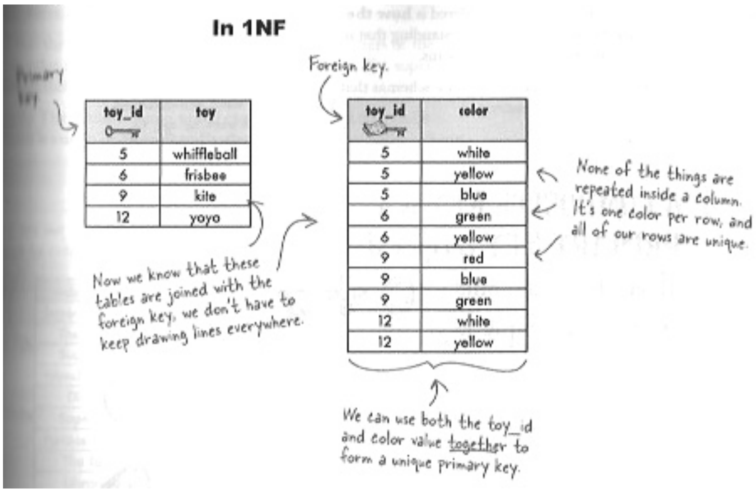
- Avoids duplicates
- Change entries in *one* place

1NF

We first need to make sure we're in *first normal form*

1. Each row of data must contain atomic values
2. Each row of data must be unique
3. Values stored in a column should be of the same type
4. No repeating types of data across columns
5. Columns should have unique names
6. Order of data stored doesn't matter

Composite key



In this example, we've made the data atomic and split the tables — however, the second column needs a little work. It has duplicate data in the rows. To fix this, we can use a **composite key**

- A composite key is a primary key using multiple columns

So, if we combine the `toy_id` and it's `color`, we'll have our *composite key* and each row will be unique!

Functional dependency

```
--[ Superheroes ]-----------
name            | Super Guy
power           | Flies
...             | ...
initials        | SG
arch_enemy_id   | 4
arch_enemy_city | Kansas City
```

1. **Composite key**: `name` and `power`
2. **Functional dependency**: `initials`

`initials` are *functionally dependent* on `name`; they'll change if our superhero `name` does!

> *A dependent column is one containing data that could change if another column changes*

Partial functional dependency

The `initials` column is *partially* dependent on `name`. Partial dependency means one *non-key* column is reliant on some, but not all, of the columns in a composite primary key.

Transitive functional dependency

How does each *non-key* column relate to others? If our `arch_enemy_id` changes, it *could* potentially change the `arch_enemy_city` field. If it could potentially change one of the other (non-key) columns, it's called a *transitive functional dependency*.

Independent

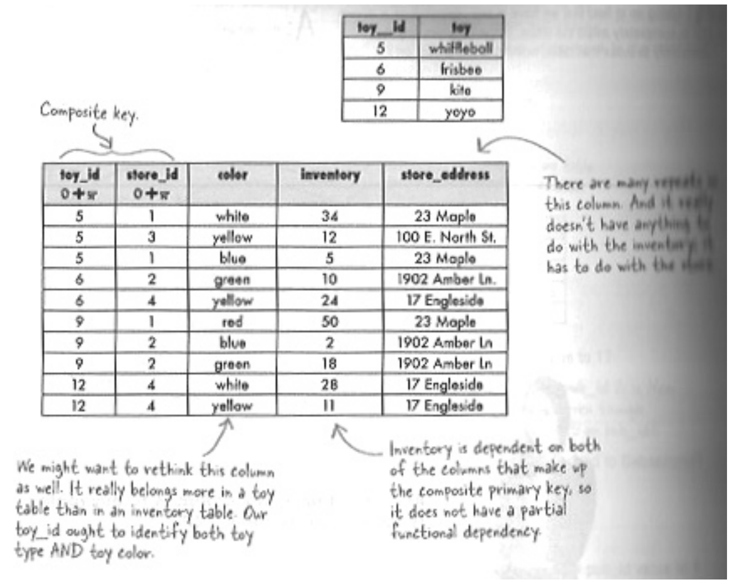
Your column can also be independent. For instance, if `arch_enemy_city` is not dependent on `arch_enemy_id` or any other column.

- If you changed a row in your column, would anything else change? No?
- Your column is independent!

Avoiding dependencies

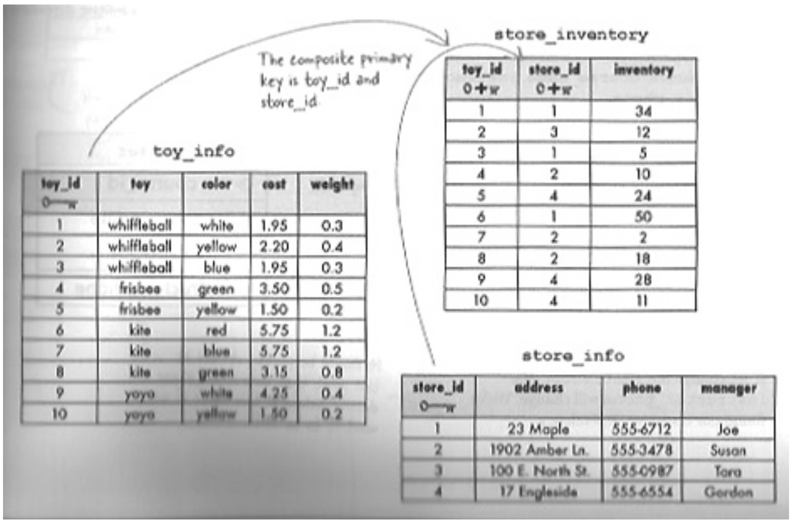
You'll generally avoid dependencies by making sure all tables have a unique primary key. There are some good arguments for both a *synthetic key* and a *natural key*. Google it!

2NF



toy_id	toy
5	whiffleball
6	frisbee
9	kite
12	yoyo

toy_id	store_id	color	inventory	store_address
5	1	white	34	23 Maple
5	3	yellow	12	100 E. North St.
5	1	blue	5	23 Maple
6	2	green	10	1902 Amber Ln.
6	4	yellow	24	17 Engleside
9	1	red	50	23 Maple
9	2	blue	2	1902 Amber Ln
9	2	green	18	1902 Amber Ln
12	4	white	28	17 Engleside
12	4	yellow	11	17 Engleside



toy_info

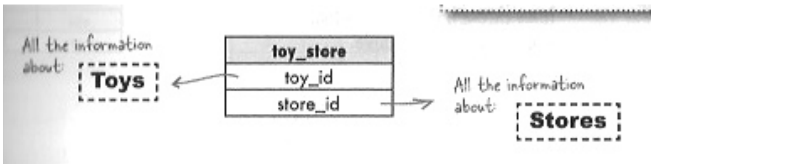
toy_id	toy	color	cost	weight
1	whiffleball	white	1.95	0.3
2	whiffleball	yellow	2.20	0.4
3	whiffleball	blue	1.95	0.3
4	frisbee	green	3.50	0.5
5	frisbee	yellow	1.50	0.2
6	kite	red	5.75	1.2
7	kite	blue	5.75	1.2
8	kite	green	3.15	0.8
9	yoyo	white	4.25	0.4
10	yoyo	yellow	1.50	0.2

store_inventory

toy_id	store_id	inventory
1	1	34
2	3	12
3	1	5
4	2	10
5	4	24
6	1	50
7	2	2
8	2	18
9	4	28
10	4	11

store_info

store_id	address	phone	manager
1	23 Maple	555-6712	Joe
2	1902 Amber Ln.	555-3478	Susan
3	100 E. North St.	555-0987	Tara
4	17 Engleside	555-6554	Gordon



How the *primary key* in a table *relates* to the data in it. Any table with an **artificial primary key** and no composite primary key is always 2NF.

1. Must be in 1NF
2. Has no partial functional dependencies

3NF

If changing any of the *non-key* columns could cause any other columns to change, it's a transitive dependency. We're looking to avoid these!

1. Must be in 2NF
2. Has no transitive dependencies

```
--[ courses ]--
course_id
course_name
instructor
instructor_phone
```

- We can ignore *primary key* when considering 3NF
- No columns will change if we change a `course_name`

- `instructor` will not change if `instructor_phone` changes …
- But `instructor_phone` will change if `instructor` does!

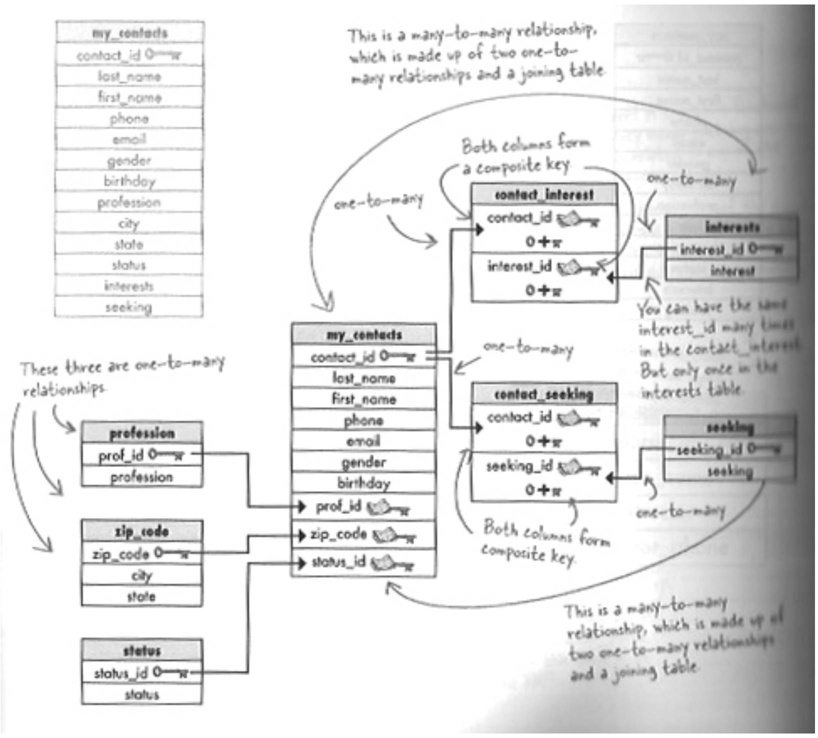
So it's obvious that `instructor_phone` belongs in the `instructor` table.

Let's re-format our single table

From this …

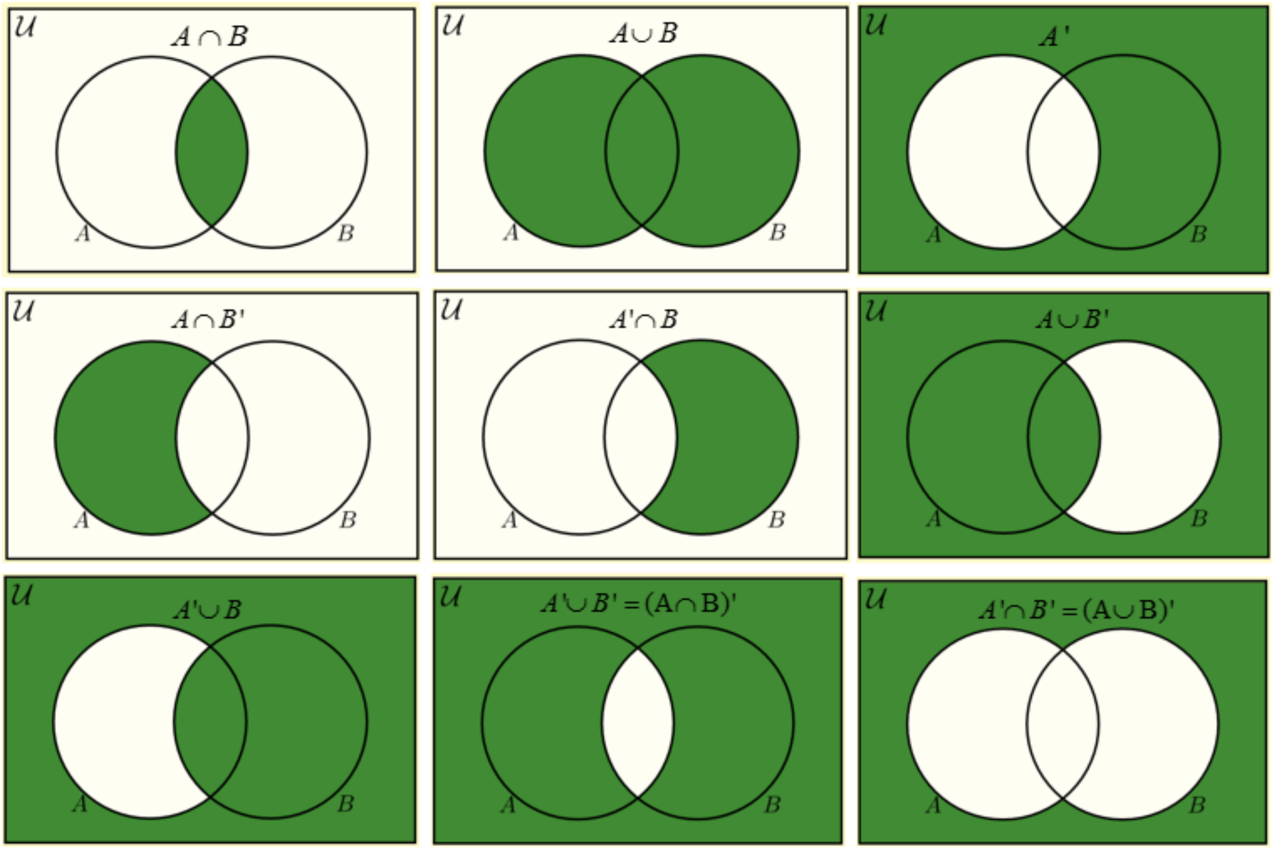
```
-[ One of greg's contacts ]-------------------
contact_id | 1
last_name  | Anderson
first_name | Jillian
email      | jill_anderson@breakneckpizza.com
gender     | F
birthday   | 1980-09-05
profession | Technical Writer
location   | Palo Alto, CA
status     | single
interests  | kayaking, reptiles
seeking    | relationship, friends
```

To this …



08: Joins

Linking tables



Now we've created our *database schema*, we'll need a way to view our data together. **Joins** help us link multiple tables together …

Let's organise our contacts first

You learned how to organise *greg's list* in the previous chapter, to a better 3NF schema. How would we go about doing this in code? Let's take the `interests` column as an example:

```
interests                                  id | interests
-----------------------------     ==>      ---+----------
kayaking, reptiles, cooking                1  | kayaking
                                           2  | reptiles
                                           3  | cooking
```

1. Find all *unique* values in the original `interests` table rows
2. Create a new `interests` table
3. Copy values to new table (no duplicates!)
4. Delete the original `interests` column

For every non-atomic column we need to extract into it's own table, we follow these steps.

How to do it?

Remember our girl guides? We used `SELECT DISTINCT` to retrieve the dates, without duplicates. Let's do the same to see if we can pull out unique values:

```
SELECT DISTINCT interests
FROM my_contacts;
```

```
-[ RECORD 1 ]-----------------------------
interests | RPG, kayaking
```

```
-[ RECORD 2 ]----------------------------
interests | RPG, anime
...
-[ RECORD 10 ]---------------------------
interests | women
-[ RECORD 11 ]---------------------------
interests | NULL
```

Oh no! Our records are `'string, list, items'`, so that won't work.

How about a function?

Our records are just strings. How about a string function?

```
SELECT split_part(interests, ',', '1')  -- #1
FROM my_contacts
WHERE interests IS NOT NULL;  -- #2
```

1. Split string at first `,`, return first part
2. We have some `NULL` values, so ignore those!

```
-[ RECORD 1 ]----------------
split_part | RPG
-[ RECORD 2 ]----------------
split_part | RPG
...
-[ RECORD 10 ]---------------
split_part | women
```

Migrate our data

We're using a string function to select our data. Let's migrate it into our new `interests` table. One way is to:

1. Create new columns
2. Split our `string, list` into atomic data
3. Add each list item into it's own column
4. Migrate each column into our new `interests` table

```
ALTER TABLE my_contacts
ADD COLUMN interest1 character varying(50),
ADD COLUMN interest2 character varying(50),
ADD COLUMN interest3 character varying(50),
ADD COLUMN interest4 character varying(50);
```

```
interests  | RPG, kayaking
interest1  | NULL
interest2  | NULL
interest3  | NULL
```

We can now copy and paste `interests` first item into `interest1` …

```
UPDATE my_contacts
SET interest1 = split_part(interests, ',', 1);
```

```
interests | RPG, kayaking
interest1 | RPG
```

Now you can delete that first item from `interests`!

```
UPDATE my_contacts
SET interests = substr(interests, length(interest1) + 2 + 1)  -- #1
```

```sql
UPDATE my_contacts
SET interests = NULL;
```

1. `substr(...)` returns a portion of the string
   - `|R|P|G|,| |K|a|y|a|k|i|n|g|` row has 13 characters
   - We want to return `Kayaking`
   - Our *position start* should be `6`
     - Take the length of `RPG` (now in `interest1`)
     - Plus `2` (the comma and space)
     - Add `1` (to move our imaginary cursor to `K`)
   - Returns *position* 6, to end.
2. `NULL` our old `interests` column (we'll delete it later)

```
interests | kayaking
interest1 | RPG
```

Repeat the process for the remaining `interests` list items, moving into `interests2`, `interests3`. Remember to ignore your new `NULL` values!

We're not quite done yet!

We can view all our interests with a little digging …

```sql
SELECT
  interest1,
  interest2,
  interest3
FROM my_contacts
WHERE
  interest1 IS NOT NULL
  OR interest2 IS NOT NULL
  OR interest3 IS NOT NULL;
```

But we can't easily pull these out into a single results set. We need a way to merge them.

Merging data

There are **3 ways** to merge all our `interest` tables! It can be helpful to know different ways to perform a task, as it can speed up your queries. Let's use our `professions` table as an example.

`CREATE TABLE`, then `INSERT` with `SELECT`

1. The `SELECT` holds our list of values
   - You'd usually use `VALUES` here
2. `GROUP BY` helps us avoid duplicates
3. Add our values alphabetically

```sql
CREATE TABLE profession (
  id SERIAL,
  profession varchar(20)
);

INSERT INTO profession (profession)
  SELECT profession     -- #1
  FROM my_contacts
  GROUP BY profession   -- #2
  ORDER BY profession;  -- #3
```

`CREATE TABLE` with `SELECT`, then `ALTER` to add primary key

1. Grab our values from `my_contacts` `profession` column
   - Which populates our new `profession` table
2. Add in our primary key after

```
CREATE TABLE profession AS
  SELECT profession     -- #1
  FROM my_contacts
  GROUP BY profession
  ORDER BY profession;

ALTER TABLE profession
ADD COLUMN id SERIAL;    -- #2
```

`CREATE`, `SELECT` and `INSERT` — all at the same time!

1. Create the table (as you would normally)
2. Add on the `SELECT` statement holding all our `profession` values

```
CREATE TABLE profession (
  id SERIAL,
  profession varchar(20)
) AS
  SELECT profession
  FROM my_contacts
  GROUP BY profession
  ORDER BY profession;
```

Aliasing

You can think of the `AS` keyword as a variable. It stores the value of `X` as the alias `Y`.

- An alias is *temporary*
  - It displays in your results …
  - But *doesn't* change original values
- It helps to *simplify*, and make values *easy to read*
- You can alias *without* using `AS`
  - Just remove the `AS` keyword!
  - It sometimes makes things clearer to use it, though

Column aliases

You've seen these before. `AS` allows us to store the value of a column:

```
CREATE TABLE profession (
  id SERIAL,
  profession varchar(20)
) AS
  SELECT profession AS mc_prof   -- #1
  FROM my_contacts
  GROUP BY mc_prof
  ORDER BY mc_prof;
```

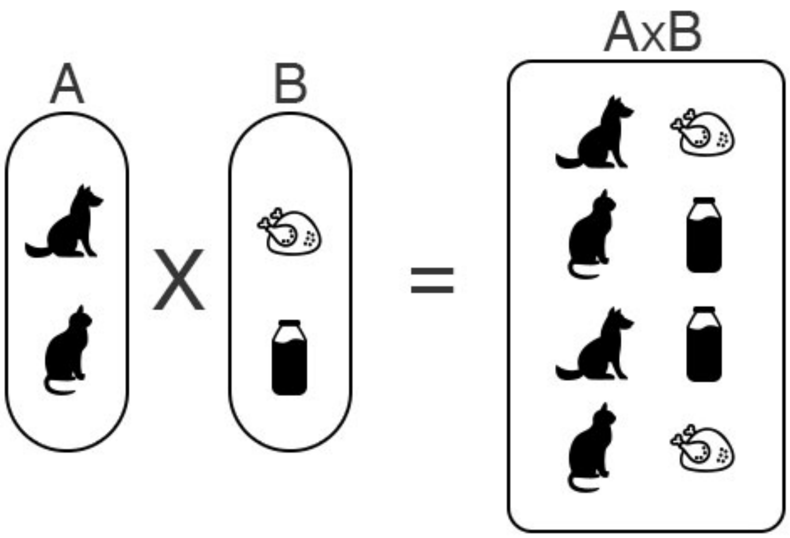
Table aliases

- Also known as *correlation names*.

- Help when you're selecting data from more than one table
- Reduces finger fatigue (no typing table names over and over!)

```
SELECT profession mc_prof
FROM my_contacts mc
GROUP BY mc_prof
ORDER BY mc_prof;
```

Cartesian join



Cartesian product of two sets: *tables A and B have* `2 * 2` *or* `6` *possible combinations*

- Also known as the *cartesian product*
- Returns *all* possible combinations of rows, from two (or more) tables
- Can help you fix bugs in your joins (if you accidentally get cartesian results)
- Test the speed of your RDBMS and it's configuration
- **DO NOT** use on large datasets — it could kill your machine

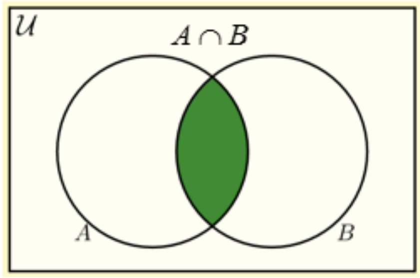
It can be written like this in SQL:

```
SELECT a.animal, b.ingredient
FROM animal AS a
  CROSS JOIN
  breakfast AS b;

-- or

SELECT a.animal, b.ingredient
FROM animal a, breakfast b;
```

Inner join



An *inner join* combines records from two tables, using *comparison operators* in a *condition*:

```
SELECT somecolumns
FROM table1
  INNER JOIN
    table2
ON somecondition;
```

You can use `GROUP BY`, `ORDER BY`, `WHERE`, and functions with a `JOIN` — just like a regular `SELECT`!

Equijoin

Equijoin inner joins test for equality — here, we're matching foreign key to a primary key:

```
SELECT somecolumns
FROM table1 AS t1
  INNER JOIN
    table2 AS t2
ON t1.id = t2.id
```

If we had a table of boys with one toy each, and a one-to-one relationship with a table of toys an equijoin would give us the following results:

```
id | boy    | toy_id       toy_id | toy
---+--------+-------       -------+-----
1  | Davey  | 3            1      | hula hoop
2  | Bobby  | 5            2      | balsa glider
3  | Beaver | 2            3      | toy soldiers
...                        ...
```

```
-[ RECORD 1 ]-------
boy | Richie
toy | hula hoop
-[ RECORD 2 ]-------
boy | Billy
toy | balsa glider
-[ RECORD 3 ]-------
boy | Beaver
toy | balsa glider
-[ RECORD 4 ]-------
boy | Davey
toy | toy soldiers
-[ RECORD 5 ]-------
boy | Johnny
toy | harmonica
-[ RECORD 6 ]-------
boy | Bobby
toy | baseball cards
```

Non-equijoin

You can use the *not equal to* (<> or `!=`) returns the toys each boy *doesn't* have:

```
SELECT boys.boy AS boy, toys.toy
FROM boys
  INNER JOIN
  toys
ON boys.toy_id <> toys.toy_id
ORDER BY boy;
```

```
-[ RECORD 1 ]-------
boy | Beaver
toy | baseball cards
-[ RECORD 2 ]-------
boy | Beaver
toy | etch-a-sketch
-[ RECORD 3 ]-------
boy | Beaver
toy | harmonica
-[ RECORD 4 ]-------
boy | Beaver
toy | toy soldiers
-[ RECORD 5 ]-------
boy | Beaver
toy | slinky
-[ RECORD 6 ]-------
boy | Beaver
toy | tinker toys
-[ RECORD 7 ]-------
boy | Beaver
toy | hula hoop
-[ RECORD 8 ]-------
boy | Billy
...
```

Natural join

Natural joins only work if the column you're joining by has *the same name on both tables*. It'll *naturally* attempt to join the two tables on their *identical* column names.

```
SELECT boys.boy, toys.toy
FROM boys
  NATURAL JOIN
  toys;
```

Will return the *very same* result from our first inner join!

09: Subqueries

Queries within Queries

A *subquery* is simply one query, inside another. It's helpful to:

- Search a growing database
- Pass one query (data set) to another query
- Look up one thing first, then perform manipulations
- Answer more than one question

```
Which x is y
           ↳ what Y = ?
```

You can use subqueries with `INSERT`, `DELETE`, `UPDATE`, and `SELECT`.

Outer and inner query

- A subquery within another query is an *inner query*
- It's parent, is called an *outer query*

Scalar values

A Subquery should generally return a *scalar value*

- One query, one row, returning a *single value*

Subqueries within subqueries

Can you write subqueries *inside* subqueries? Yes! However, you should limit these wherever possible.

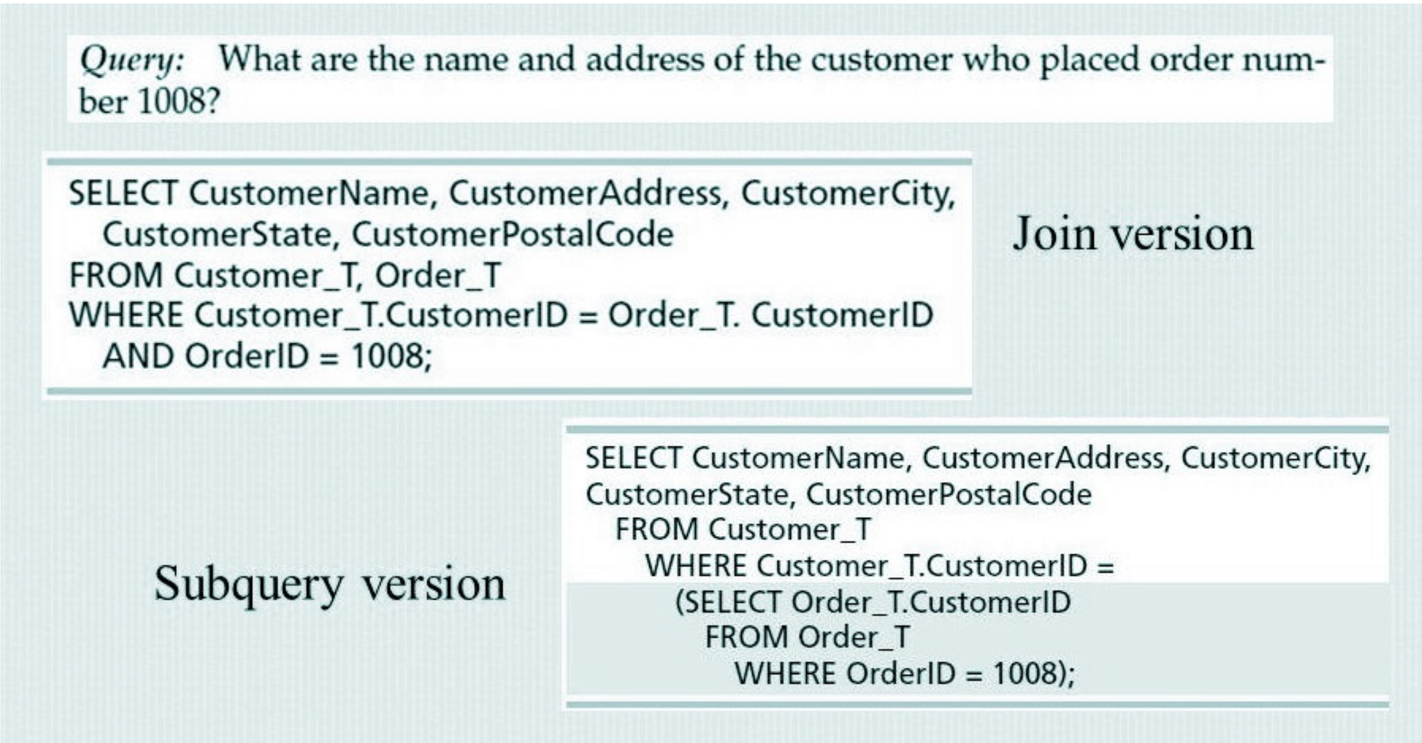
Subquery steps

To solve a subquery:

1. Split into steps
2. What questions are you asking the database?
3. What data is required for each step?
4. Write the query

Finally, you'll combine the queries together with the `WHERE` clause.

Subquery vs join



An example of a subquery and a join — both give the same result!

You often get faster results by using a `JOIN` (or `LIMIT`) *instead* of a subquery. Most subqueries can be replaced and return the same data; you'll need to ask yourself:

- Which method is faster? (experiment)
- Does it need to be fast? (local/live)
- Which is easiest to read?
- Is it simpler to use?

Make the query as *easy as possible* for your database to answer. Write what makes logical sense first, and worry about performance later.

What are the benefits?

- It can sometimes be faster
- You don't explicitly need to know the value
  `x > SELECT`, rather than hardcoding the value to compare

IN keyword

```
-- example of operator with IN
SELECT mc.first_name, mc.last_name, mc.phone, jc.title
FROM job_current AS jc
NATURAL JOIN my_contacts AS mc
WHERE jc.title IN (
  SELECT title FROM job_listings  -- #1
);
```

Using the `IN` keyword with an inner join, we can search within a list of values, without having to add them by hand — great for dynamically changing records.

1. Returns a list of values
    - Is our job title `IN` this list?

Combined with a parent query, it can be very useful!

NOT IN

The inverse of `IN`. It returns records that *aren't* within a list of values:

```
-- example of operator with NOT IN
SELECT mc.first_name, mc.last_name, mc.phone, jc.title
FROM job_current js
NATURAL JOIN my_contacts mc
WHERE jc.title NOT IN (
  SELECT title FROM job_listings
);
```

Correlation

Non-correlated subquery

```
-- examples of non-correlated subquery
SELECT mc.first_name, jc.salary
FROM my_contacts AS mc
NATURAL JOIN job_current AS jc
WHERE jc.salary > (
  SELECT jc.salary
  FROM my_contacts mc
  NATURAL JOIN job_current jc
  WHERE email = 'andy@weatherorama.com'
);
```

An inner query that **is not** referenced in the outer query.

- `Y` doesn't know about outer query `X` (it's a single value)
- But `X` is dependant on the result of `Y`

Here, the inner query is processed first.

Correlated subquery

```
-- example of subquery in a select column
-- returns those who have 3 interests
SELECT mc.first_name, mc.last_name
FROM my_contacts AS mc
WHERE 3 = (
  SELECT COUNT(*) FROM contact_interest
  WHERE contact_id = mc.contact_id
);
```

A correlated subquery **does** reference the outer query.

- `Y` is dependant on the outer query `X`
    - i.e: reference an alias within subquery

EXISTS

```
SELECT mc.first_name firstname, mc.last_name lastname, mc.email email
FROM my_contacts AS mc
WHERE EXISTS (
  SELECT * FROM contact_interest AS ci
  WHERE mc.contact_id = ci.contact_id
);
```

Find data from `my_contacts` where `contact_id` shows up at least once in `contact_interest` table.

NOT EXISTS

```
SELECT mc.first_name firstname, mc.last_name lastname, mc.email email
FROM my_contacts mc
WHERE NOT EXISTS (
  SELECT * FROM job_current jc
  WHERE mc.contact_id = jc.contact_id
);
```

Find all the rows in the outer query, for which no rows exist in a related table.

10: More joins

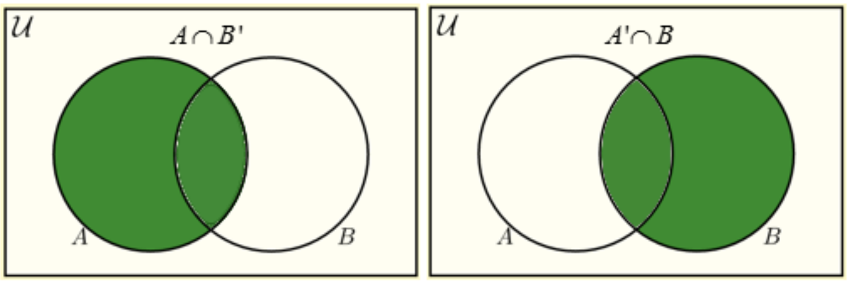
Cleaning up old data

There comes a time when you'll have to clean up your data. You'll need tools to check for *orphan data*:

1. Rows in one table …
2. That are no longer referenced in a parent column

See also subquery vs join

Outer join



You've seen an *inner join* before. Another tool in your belt is the *outer join*.

An inner join:

- `row a` in `table a` matches to …
    - `row b` in `table b`

Returns exact matches between `table a` and `table b`, using a matching *primary key* or *id*

An outer join:

- rows in `table a` match to …
    - rows in `table b`
        - even if there's `NULL` matching result

Returns exact matches between `table a` and `table b` — also returns `NULL` values if no matching *primary key* or *id* can be found on `table b`.

Left and right joins

R

ColA	ColB
A	1
B	2
D	3
F	4
E	5

S

SColA	SColB
A	1
C	2
D	3
E	4

R.ColA = S.SColA S

Left outer join

A	1	A	1
D	3	D	3
E	5	E	4
B	2	-	-
F	4	-	-

A *left outer join* matches `table a`, against `table b`

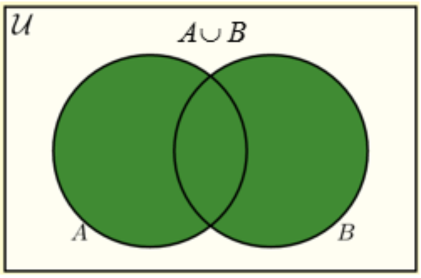
Right outer join

A	1	A	1
D	3	D	3
E	5	E	4
-	-	C	2

A *right outer join* matches `table b` against `table a`

It's generally best to **stick to one** and switch the actual tables instead.

Full outer join



A	1	A	1
D	3	D	3
E	5	E	4
B	2	-	-
F	4	-	-
-	-	C	2

Returns exact matches between `table a` with `table b` — also returns `NULL` values if no matching *primary key* or *id* **on both sides**.

Join on a single table

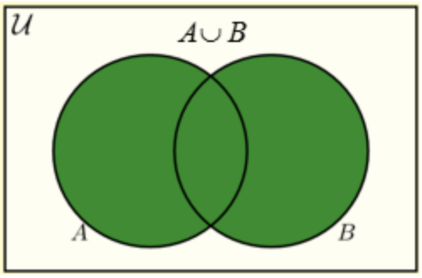
```
employee_id     Int     Primary Key
employee_name   String
manager_id      Int     Foreign key going back to the EmployeeId
```

```
SELECT t1.employee_name AS employee, t2.manager_name AS boss
FROM employees AS t1
INNER JOIN employees AS t2
ON t2.manager_id = t1.employee_id;
```

You can also join on the *same table*. One example is using a self-referencing foreign key:

- Self referencing foreign key:
  - primary key of table used within the same table for a different use case
  - you join the table *to itself*
- Simulates having two tables
- Reference one id on another

UNION



`UNION` allows you to merge two or more sets, returning common values:

```
{'designer', 'dentist'} =>
{'nurse', 'designer'}   =>  {'designer', 'dentist', 'manager', 'nurse'}
{'manager'}             =>
```

```sql
SELECT title FROM job_current
UNION
SELECT title FROM job_desired
UNION
SELECT title FROM job_listings;
```

```
title
--------
designer
dentist
manager
nurse
```

Limitations

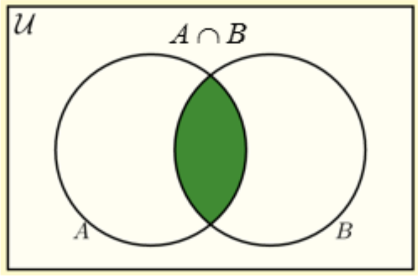
Each `SELECT` query must have:

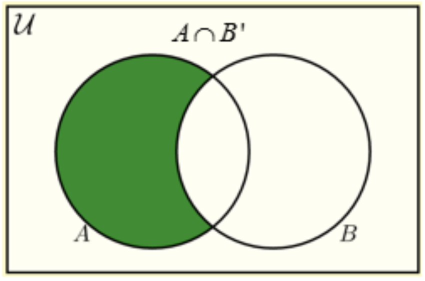
- Same number of columns
- Same datatype

UNION ALL

To show *every single* record from each `SELECT` — *including* duplicates — use `UNION ALL`.

INTERSECT and EXCEPT





INTERSECT

```
SELECT title FROM job_current
INTERSECT
SELECT title FROM job_desired;
```

Returns columns/records that are in **both** the first and second query.

EXCEPT

```
SELECT title FROM job_current
EXCEPT
SELECT title FROM job_desired;
```

Returns columns/records that are in the ***first* query**, but ***not** the second*.

11: Constraints, views and transactions

Maintaining control

As your database grows, it's important to carefully maintain control:

1. Manage access (control)
2. Manage data input (validity)
3. Manage data manipulation (integrity)

There's three methods to do this: `CHECK`, `VIEW` and `TRANSACTION`.

CHECK

Image of scheme

A `CHECK` is a constraint that restricts and validates input. You've already seen a few of these already:

- NOT NULL
- PRIMARY KEY
- FOREIGN KEY
- SERIAL (or UNIQUE)

Imagine our database table `my_contacts`. Our new recruit, Jim, has the task of adding new entries:

1. He's adding in a new member, *Pat*
2. He isn't sure if Pat is a `Male` or `Female`
3. To avoid a `NULL` entry, he uses `'X'` instead

```
-- Jim looks up the profession id
SELECT profession_id WHERE profession = 'teacher';
-- He also looks up status id
SELECT status_id WHERE status = 'single';
-- Finally, he adds Pat to the database
-- including gender, profession_id, status_id
INSERT INTO my_contacts
VALUES ('Pat', 'patmurphy@someemail.com', 'X', 19, 3)
```

What happens if, in the future, we count `Male` or `Female` members? We'd have some members with gender `X`, so our results are messed up:

```
-- All members (1000)
SELECT * FROM my_contacts;
-- Males (450)
SELECT * FROM my_contacts
WHERE gender = 'M';
-- Females (300)
SELECT * FROM my_contacts
WHERE gender = 'F';
```

Check constraint to the rescue!

```
CREATE TABLE piggy_bank (
  id INT SERIAL PRIMARY KEY,
  coin CHAR(1) CHECK (coin IN ('P', 'N', 'D', 'Q'))
);
```

```
[data] ----> check ----> success/error
```

Say if we had a piggy bank, which could only take a certain type of coin. We can check it against a list of values, as above:

1. Similar to `WHERE` clause
2. You can use `AND`, `OR`, `IN`, `NOT`, `BETWEEN`
3. You can also use things like string functions

So, we can now update our original `gregs_list` table to avoid Jim making his `X` errors:

```
UPDATE TABLE my_contacts
ADD CONSTRAINT constraint_name CHECK (gender IN ('M', 'F'));
```

Notes on MySQL

MySQL **does not** enforce data integrity with `CHECK` (there are alternative methods).

VIEW

> *`VIEW` simplifies your queries, making them easier to remember*

If you find yourself entering the same queries, over and over, it can become tedious. There's a solution for that!

1. Store your query for reuse
2. Simplify and easy to remember
3. Hide sensitive data
4. Avoid making mistakes

```
-- Store view
CREATE VIEW web_designers AS
SELECT mc.first_name, mc.last_name, mc.phone, mc.email
FROM my_contacts mc
NATURAL JOIN job_desired jd  -- or, INNER JOIN
WHERE jd.title = 'Web Designer';
-- Use view
SELECT * FROM web_designers
```

The view actually behaves similar to a *subquery*:

```
SELECT * FROM (
  SELECT ...
  FROM ...
  NATURAL JOIN ...
  WHERE jd.title = 'Web Designer'
) AS web_designers;
```

- You must alias the `SELECT`
- So the `FROM` recognises it as a table

Updating the view

> *There are two types of views, updatable and non-updatable.*

You can change the underlying structure of the `VIEW`, or table, without the user (or app) needing to know about it. `DROP VIEW` to delete it (do this before deleting it's table).

Views and security

- A view can technically use `UPDATE`, `INSERT`, `DELETE` (using `CHECK OPTION`)
- But it's best to use a `VIEW` as **read only**
- **Be careful** when giving others access to views!!

Transactions

When your piggy bank screws up

You're the owner of the bank; three people are accessing their account at the same time, perhaps they're:

1. Checking their balance
2. Withdrawing cash
3. Moving cash between accounts

Let's take one of those examples:

1. Jack checks his account:
    - `£3000` available in account `A`
    - `£0` available in account `B`
2. Jack decides to move `£500` from account `A` to `B`
    - `£2500` in account `A`
    - `£0` in account `B`

That's not right! Unknown to Jack, there was a blackout in the branch — part of his transaction was lost:

```
If account A >= 500
Subtract 300 from account A

==[ BLACKOUT! ]=============
Deposit 300 to account B
============================
```

What happens if multiple accounts have the same problem? Or wires get crossed and mix up transactions? That's a lot of unhappy customers :(

Complete or kill!

> *A transaction is a group of query "steps". If one step fails, all steps fail.*

We use the classic ACID test to help us decide when to use an SQL transaction:

1. Atomicity
2. Consistency
3. Isolation
4. Durability

The following keywords help us perform a transaction:

1. `START TRANSACTION`: Keeps track of sql that follows
2. `COMMIT`: If nothing fails, we can make things permanent
3. `ROLLBACK`: If we have an error, it's as if nothing ever happened

Postgres uses simpler syntax, but it's essentially the same:

```
-- Begin a transaction
BEGIN;
UPDATE accounts SET balance = balance - 100.00
    WHERE name = 'Alice';
    ...
ROLLBACK; -- Oops!

-- Commit a change
BEGIN;
UPDATE accounts SET balance = balance - 100.00
    WHERE name = 'Alice';
    ...
COMMIT;
```

12: Security

Database security

Who needs access and why?

You'll want to limit risk as much as possible, so database security is a **must**!

1. Stop hackers
2. Limit access
3. Avoid data problems

First things first: **never use root**! The default user (root) has unlimited access to your database — imagine if the guy above had access to that?

1. Set a strong password for *root*
   - On `local` and `live`
2. Never commit your password to a repo
3. Keep it secret, keep it safe

Ask someone who knows what they're doing if setting a password on `root` is required in Postgres (it's a bit different and you can use `psql`)

ADD USER

Next, you'll want to create a user. This allows you to:

- Give full, or partial access
- Access without `root`
- Create multiple users

```
CREATE USER username WITH ENCRYPTED PASSWORD 'password';
```

Create a different account for each person who'll access the database. You can then need to decide:

1. Who needs access?
2. How much access?
3. For what reasons?

Also consider their capabilities:

1. What is their level?
2. What is their experience?
3. Will they break anything?

It's better to be safe than sorry; easier to incrementally add access than remove it.

User privileges

Below are some of the commands you'll find yourself using with new users — **read the docs** and tutorials! It's best to learn as you go, only adding what you need.

- `GRANT` access
   - For example, allow them to `SELECT` or `INSERT`
   - But *not* `DELETE` rows
- `REVOKE` access
   - `CASCADE` the changes
   - `RESTRICT` the changes
- `CREATE ROLE`
   - `DROP` and remove it

Appendix

Styleguide

Some useful tips:

1. SQL Style Guide

General guidelines

```
-- UPPERCASE sql statements
-- Always ends in semi-colon
CREATE DATABASE db_name;

-- lowercase, snake_case
CREATE TABLE db_table (
  column_name varchar(10),  -- use correct data type
  ...
);

-- Always use single quotes
-- Numbers do not need quotes
INSERT INTO db_table (column_name, column_name)
VALUES ('value', 1.0),  -- use commas
       ('value', 2.0);  -- except the last one!
```

Gotchas

These notes and this text are true of Postgres 11 — they're subject to change, so do your homework!

SELECT queries

These don't seem like they should work, but they do. These queries **are not correct**, just a little forgiving:

```
SELECT * FROM easy_drinks WHERE amount2 = 6;       -- Matches 6.0
SELECT * FROM easy_drinks WHERE main > 'soda';     -- Weirdly, yes
SELECT * FROM easy_drinks WHERE amount1 = '1.5';   -- Treats as a number
```

Datatype queries

Type	Single quotes	No quotes
`char`	✔	
`varchar`	✔	
`text`	✔	
`date`	✔	
`datetime`, `timestamp`	✔	
`int`		✔
`decimal`		✔

Escaping characters

```
-- Escape strings
INSERT INTO easy_drinks
VALUES ('Rob''s soda');
-- Escape columns
SELECT "name", price FROM easy_drinks;
```

- Postgres prefers an extra single quote
    - You can also escape with a backslash
- Where possible, let your programming language do it for you!
    - Use a GUI for working with raw SQL
- In standard sql, you can escape with a backslash \ '
- Never use double quotes to escape, as it can cause your software problems
- Use the same method when using a `SELECT` query

You DON'T need a comma for that!

```
SELECT col_name FROM table
WHERE col_name > 't'   -- No need for a comma here
  AND price < 4;       -- but make sure you end with a semi-colon;
```

You DO need a comma for that!

```
ALTER TABLE my_table
ADD COLUMN column_name int,  -- watch out for those commas!
ADD COLUMN column_name int
```

Any value is better than NULL

- It's best to add something, rather than leaving values `NULL`
- It can't be directly selected from a table
- Sometimes it may even be best to delete (or ignore) them completely!

Deleting records

- Always include a `WHERE` statement, or you'll delete *all* your rows!
- Always check your `WHERE` statement for errors
- Always check your `DELETE` order

- Always check other rows for *shared values* (that you don't want to change)
- If in doubt: **use `SELECT` first to check your `WHERE` statement!**

Updating records

- Always include a `WHERE` statement, or you'll update *all* your rows!
- Always check your `WHERE` statement for errors
- Always check other rows for *shared values* (that you don't want to change)
- Order matters: (highest first: `{1.50 -> 2.00, 1.00 -> 1.50}`)
    - Two statements may update the same column
    - Be specific with your `WHERE` clauses :)
- Be sure to check your new values — are they *all* what you expected?
- If in doubt: **use `SELECT` first to check your `WHERE` statement!**

Speed, size, accuracy

- Always make your data as simple as possible
    - Reduce cognitive load (and potential mistakes)
    - Make it fast and easy to: enter, monitor, edit
    - See Atomic Data chapter notes
- Always try to delete unused columns or data
- Always add sensible limits to data types
- Always request what you *need* (`limit` results)
    - But *you must* make sure the order is predictable (`order by`)
    - Row order *is not guaranteed* in SQL queries
- Always keep a diagram of your schema
    - Keep data and schema independent of each other
- Make the question as *easy as possible* for the database to answer
- Smaller is better: `cross join > correlated > non-correlated > join`
    - `join` is generally faster than a `subquery`

Mapping out a database

- See introduction and atomic data
- Split out concerns
    - Tables, atomic, acid, …
- Is it universal?
    - Will this data be used everywhere?
    - Or only with a segment of your entries?
    - Which tables and records link together?
- How will it be used?
    - Do you need to limit access?
    - What parts of the data do they need to see?

Aggregate functions

- Postgres expects columns in `GROUP BY` when using aggregate functions
- It's generally good practice to ignore `0` and `NULL` values:
    - `WHERE column_name > 0` or `WHERE column_name IS NULL`
- `NULL` is **never returned** by any aggregate function
    - `NULL` is not the same as zero!
- Use `GROUP BY` when you want to use aggregate functions
- Use `SELECT DISTINCT` when you want to remove duplicates

- It's better to use an inner select
    - Especially when using `count()` or another aggregate function!
    - Returns the same values as `GROUP BY` (without aggregate functions)

Security

- Restrict user access
    - Who needs to access data, and why?
    - Which tables and actions might they need?
    - Do you restrict your users to *read only* access?
- Is your `VIEW` a security vulnerability?

Postgres: tools

Postgres can be used with an app, or on the command line.

- **Use the CLI** for quick exploration, backups and admin
- **Use a GUI** for data manipulation
- **Postgres guide** (unofficial)

You can download Postgres app for Mac:

1. Enable `psql` on your `.bash_profile`
2. Set up your `psql` config [advanced]

Postgres: PSQL

Documentation	
`help`	intro guide
`\?`	help options for psql commands
`\h`	help for sql commands (`\h [command]`)

Basic commands	
`\q`	quit PSQL (`quit` in #11)
`\l`	list all databases
`\du`	list user roles
`psql -U [username]`	connect with a specific username

Connect to database	
`\c [database]`	connect to database
`\dt`	Lists all tables
`\d` or `\d+`	`\d [table_name]` display table, constraints etc
`\dv`	`\dv [table_name]` display views
`q`	Exit list

The query buffer	
`\p`	show the contents of the query buffer
`\r`	reset (clear) the query buffer
`\s [FILE]`	display history or save it to file

Export database	
pg_dump	pg_dump [database] > [to_filename]

Nice extras	
\x [auto]	enter/exit expanded display mode

Postgres: roles and privileges

Create user

```
CREATE USER [username] WITH ENCRYPTED PASSWORD '[password]';
```

Change password

```
ALTER USER [username] WITH ENCRYPTED PASSWORD '[password]';
```

User privileges (roles)

```
GRANT ALL PRIVILEGES ON DATABASE [dbname] TO [username];
```

On a production server

Be careful! Always keep things secure.

1. Create a super user
2. Login as super user `postgres`
3. Follow the above to create a user and give privileges
4. **Never use root** (`postgres`) to connect to the database on your app

Postgres: datatypes

- Basic overview
- Full documentation

Character

Type	Description
char(n)	fixed-length character (+ padded space if < n)
varchar(n)	variable length character string (≤ n)
text	variable (unlimited) length character string

Numbers

Type	Description
`smallint`	integer with `range(-32 768, 32 767)`
`int`	integer with `range(-2 147 483 648, 2 147 483 647)`
`serial`	integer (auto populate like `AUTO_INCREMENT`)
`real`	double precision
`numeric[(p,s)]`	Real number with (p) digits and (s) precision (alias: `decimal`)

Other types are available, such as `bigint` allowing more flexibilty.

Temporal

Type	Description
`date`	stores data values only
`time`	stores the time of day values
`timestamp`	stores date and time values
`timestampz`	timezone-aware date and time values
`interval`	stores periods of time

Other

See documentation for other datatypes, such as `arrays`, `json`, `uuid` and other special types.