DATABASES AND DEVELOPERS

```
SELECT * FROM exciting_knowledge;
```

First off

ASK QUESTIONS

AT ANY TIME!

WHO AM I?

KYLE TOLLE

kyletolle.com | @kyletolle

Software Engineer on Sabbatical

NOT A DATABASE EXPERT

but I want to

SHARE

some of what

I'VE LEARNED

SLIDES FOR THIS TALK

PRESENTATION:

https://kyletolle.github.io/dbs_and_devs_talk

CODE:

https://github.com/kyletolle/dbs_and_devs_talk

PAST JOBS

- tools supporting satellite simulation
- ICBM command and control system
- geospatial data collection
- mobile wallet applications

OVERVIEW

- Background
- ORMs
- DB Theory
- Important Concepts

Let’s start at

THE VERY BEGINNING

HUMANS ARE

KNOWLEDGE-HUNTERS

&

DATA-GATHERERS

WE LIKE TO

- Collect data
- Store it
- Process it

SOFTWARE CRUNCHES

DATA

DATA NEEDS

A HOME

A.K.A.

A DATA STORE

DATA'S EARLIEST HOME

THE FLAT FILE

cat students.csv

```
Id, Name, Grade
1,  Ted,  B
2,  Stan, A-
3,  Fred, C++
4,  Ned,  5%
```

EASY PEASY

Some things don’t need to be complicated

FLAT FILES

can be used for

DATA INTERCHANGE

COMMON FLAT FILE TYPES

- CSV
- HTML
- XML
- JSON
- ini
- conf

Then, things grow

COMPLICATED

WANT TO

- know more
- ask questions
- give access

USE THE DATA FOR

- Web application
- Report generation
- Analysis performation

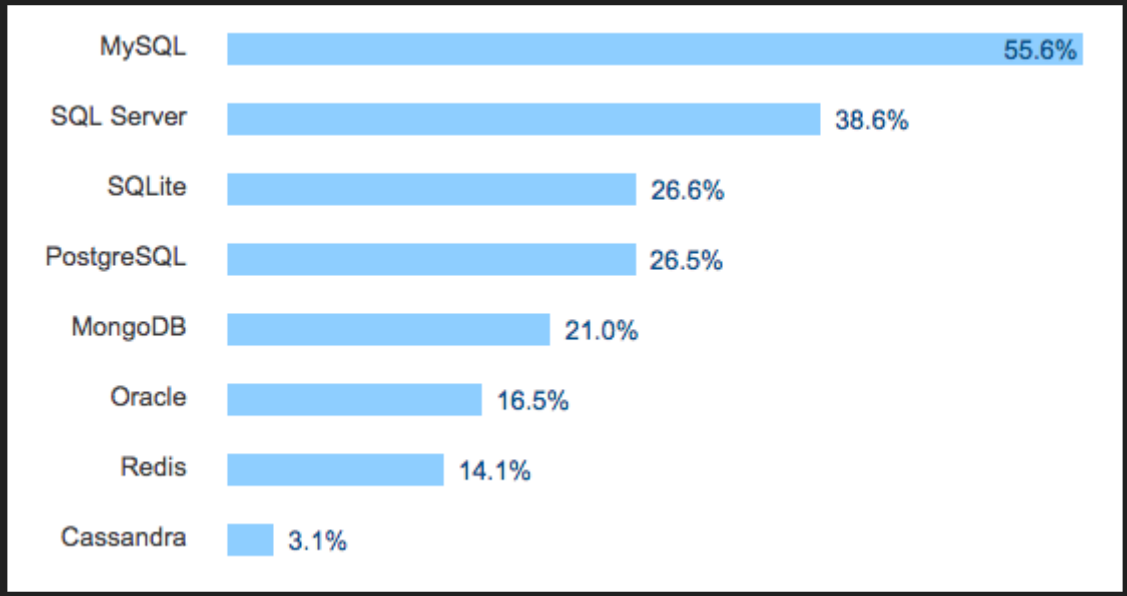
Fortunately for us,

some smart folks

INVENTED THE

DATABASE

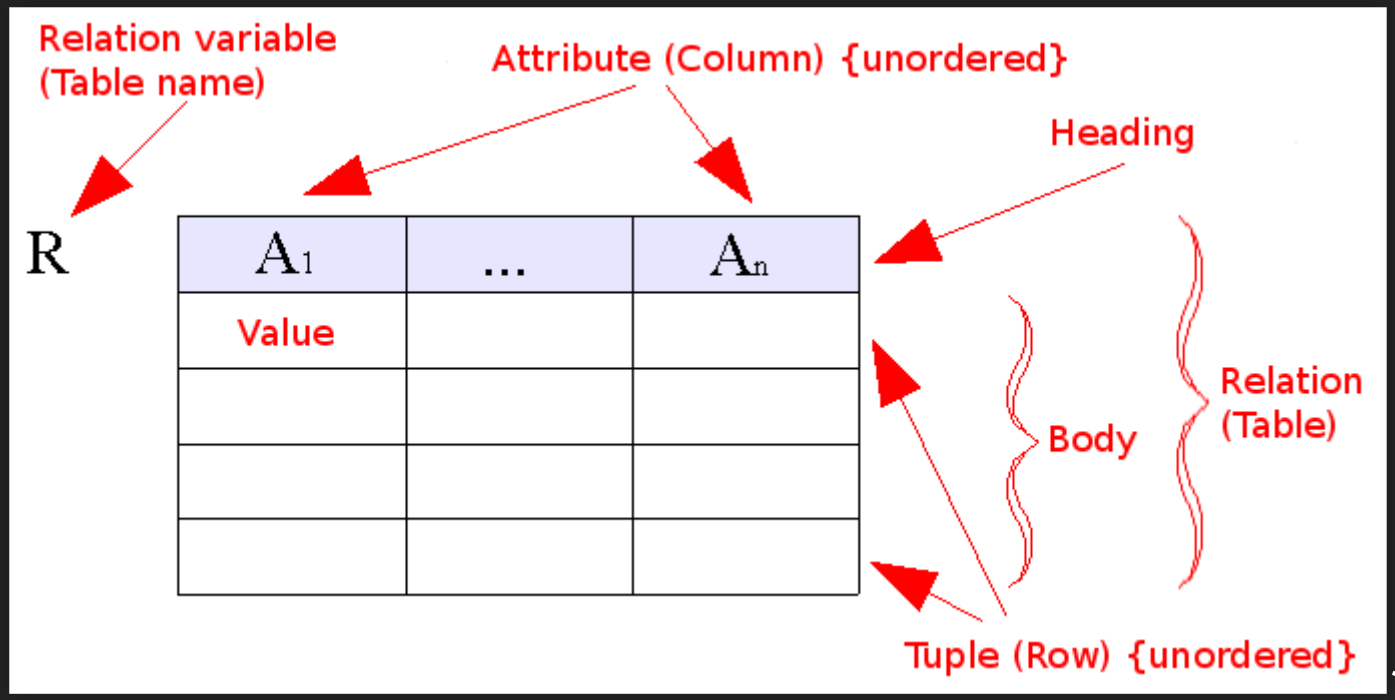
Popular DBs according to Devs



https://insights.stackoverflow.com/survey/2017#technology-databases

Many common DBs use the

RELATIONAL MODEL



*

*: By User:AutumnSnow - Own work, CC BY-SA 3.0, Link

In other words, we have a

TABLE

with

ROWS & COLUMNS

SQL IS A DSL

for managing data in a

RELATIONAL DB

Remember our student data?

```
Id, Name, Grade
1,  Ted,  B
2,  Stan, A-
3,  Fred, C++
4,  Ned,  5%
```

PUT A DATABASE ON IT!

`students`

id	name	grade
1	Ted	B
2	Stan	A-
3	Fred	C++
4	Ned	5%

The `table` has a `schema`.

Each `row` is a `record`.

Each `column` is an `attribute`.

Let’s drop into

POSTGRES

SET UP POSTGRES

```
brew install postgres
brew services start postgresql
createdb dbs_and_devs_talk
psql dbs_and_devs_talk
```

CREATE A TABLE

```
CREATE TABLE students(
  id SERIAL PRIMARY KEY,
  name TEXT NOT NULL,
  grade TEXT NOT NULL
);
```

INSERT DATA

```
INSERT INTO students (name, grade) values ('Ted', 'B');
INSERT INTO students (name, grade) values ('Stan', 'A-');
INSERT INTO students (name, grade) values ('Fred', 'C++');
INSERT INTO students (name, grade) values ('Ned', '5%');
```

VIEW DATA

```
SELECT * FROM students;
```

```
 id | name | grade
====+======+=======
  1 | Ted  | B
  2 | Stan | A-
  3 | Fred | C++
  4 | Ned  | 5%
(4 rows)
```

CREATE A RELATIONSHIP

```
CREATE TABLE backpacks(
  id SERIAL PRIMARY KEY,
  color TEXT NOT NULL,
  student_id INT references students(id)
);
```

INSERT DATA

```
INSERT INTO backpacks (color, student_id) values('blue', 1);
INSERT INTO backpacks (color, student_id) values('red', 2);
INSERT INTO backpacks (color, student_id) values('grey', 3);
INSERT INTO backpacks (color, student_id) values('green', 4);
INSERT INTO backpacks (color, student_id) values('clear', 1);
```

VIEW DATA

```
SELECT * FROM backpacks;
```

```
 id | color | student_id
====+=======+============
  1 | blue  |          1
  2 | red   |          2
  3 | grey  |          3
  4 | green |          4
  5 | clear |          1
(5 rows)
```

BACKBACK COLORS FOR A STUDENT

```
SELECT color FROM backpacks WHERE student_id = 1;
```

```
 color
=======
 blue
 clear
(2 rows)
```

BACKPACKS FOR ALL STUDENTS

```
SELECT s.name, b.color FROM students as s
  JOIN backpacks as b ON s.id = b.student_id
  ORDER BY s.name;
```

```
 name | color
======+=======
 Fred | grey
 Ned  | green
 Stan | red
 Ted  | blue
 Ted  | clear
(5 rows)
```

QUIT POSTGRES

```
\q
```

Do stuff

THE HARD WAY

for a while


I DARE YOU
NOT TO USE YOUR TOOLS!
memesnappen.com

ALLOWS YOU TO UNDERSTAND

- the benefits of your tools
- the limits of your tools
- how to work around those limits

Plus, learn to

CHANNEL YOUR ANGER

USEFUL TOOLS

- pgAdmin 4
- Postico
- ORMs

ORM

Object-Relational Mapping

Maps

DATABASE RECORDS

to

OBJECT INSTANCES

RUBY ON RAILS

follows the

ACTIVE RECORD PATTERN

INSTALL RUBY

```
brew install rbenv
rbenv install 2.4.2
rbenv global 2.4.2
```

ENSURE SQLITE3 IS INSTALLED

```
sqlite3 --version
```

INSTALL RAILS

```
gem install rails
rails --version
```

We expect to see `Rails 5.1.4`

CREATE RAILS APP

```
rails new blog
cd blog
```

CREATE POSTS

```
rails generate scaffold Post title:string body:text
```

```ruby
class CreatePosts < ActiveRecord::Migration[5.1]
  def change
    create_table :posts do |t|
      t.string :title
      t.text :body

      t.timestamps
    end
  end
end
```

RUN DB MIGRATION

```
rails db:migrate
```

ENTER RUBY CONSOLE

```
rails console
```

ACTIVERECORD IS A DSL

We can modify SQL data with it

CREATE A NEW POST

```
Post.create title: 'First!', body: 'Viral to the max.'
```

The console shows us

SQL QUERIES

for various

ACTIVERECORD METHODS

OUTPUT FOR CREATE

```
(0.1ms)  begin transaction
SQL (0.4ms)  INSERT INTO "posts"
  ("title", "body", "created_at", "updated_at")
  VALUES (?, ?, ?, ?)
  [
    ["title", "First!"],
    ["body", "Viral to the max."],
    ["created_at", "2017-11-29 01:37:50.231847"],
    ["updated_at", "2017-11-29 01:37:50.231847"]
  ]
(0.7ms)  commit transaction
=> #<Post id: 1, title: "First!",
     body: "Viral to the max.",
     created_at: "2017-11-29 01:37:50",
     updated_at: "2017-11-29 01:37:50">
```

LET’S BREAK IT DOWN

- Transactions
- Queries
- Duration
- Conventions

TRANSACTIONS

```
(0.1ms)  begin transaction
...
(0.7ms)  commit transaction
```

Multiple queries either

ALL HAPPEN

or

NONE HAPPEN

An error in a query causes a

ROLLBACK

Provides protection from

ERRORS

causing

INCONSISTENT DATA

Important when

MODIFYING MANY RECORDS

in

ONE LOGICAL CHANGE

QUERIES

```
SQL (0.4ms)  INSERT INTO "posts"
  ("title", "body", "created_at", "updated_at")
  VALUES (?, ?, ?, ?)
  [
    ["title", "First!"],
    ["body", "Viral to the max."],
    ["created_at", "2017-11-29 01:37:50.231847"],
    ["updated_at", "2017-11-29 01:37:50.231847"]
  ]
```

DYNAMICALLY GENERATED

ActiveRecord analyzes the table and

generates the correct query

DURATION

```
SQL (0.4ms)  INSERT INTO "posts" ...
```

ActiveRecord measures

HOW LONG

each query takes to run

Can help

PROFILE QUERIES

and find slow ones

CONVENTIONS

NAMING

Table name (`posts`) is pluralized

Class name (`Post`) is singular

Handles

COMPLEX CASES

like

`people` & `Person`

TIMESTAMPS

Notice the

`updated_at` & `created_at`

columns?

ActiveRecord automatically tracks

CREATION & MODIFICATION

times of each record

OTHER EXAMPLES

VIEWING

```
post = Post.last
```

```
SELECT  "posts".* FROM "posts"
  ORDER BY "posts"."id" DESC LIMIT ?  [["LIMIT", 1]]
=> #<Post id: 1,
         title: "First!",
         body: "Viral to the max.",
         created_at: "2017-11-29 01:37:50",
         updated_at: "2017-11-29 01:37:50">
```

USING

`post.title`

```
=> "First!"
```

`post.updated_at`

```
=> Wed, 29 Nov 2017 02:31:07 UTC +00:00
```

COUNTING

`Post.count`

```
SELECT COUNT(*) FROM "posts"
=> 1
```

DESTROYING

`Post.destroy_all`

```
SELECT "posts".* FROM "posts"
DELETE FROM "posts" WHERE "posts"."id" = ?  [["id", 1]]
```

JUST A SAMPLE

This is not an exhaustive list

EXIT RAILS CONSOLE

```
exit
```

ADD A COLUMN TO POST

```
rails g migration AddCurrentSongToPosts current_song:string
```

```
class AddCurrentSongToPosts < ActiveRecord::Migration[5.1]
  def change
    add_column :posts, :current_song, :string
  end
end
```

Migrations support

ROLLING BACK SCHEMA

```
rails db:rollback
```

CREATE A RELATIONSHIP

```
rails g model like post:references
```

```
class CreateLikes < ActiveRecord::Migration[5.1]
  def change
    create_table :likes do |t|
      t.references :post, foreign_key: true

      t.timestamps
    end
  end
end
```

- Adds indexes for `post_id`

COMPLEX RELATIONSHIP

An author can publish many posts, and

a post can have many authors

COMPLEX RELATIONSHIP, CONT.

```
rails g model author name:string email:string
```

```
class CreateAuthors < ActiveRecord::Migration[5.1]
  def change
    create_table :authors do |t|
      t.string :name
      t.string :email

      t.timestamps
    end
  end
end
```

COMPLEX RELATIONSHIP, CONT.

```
rails g migration CreateJoinTableAuthorsPosts author post
```

```
class CreateJoinTableAuthorsPosts < ActiveRecord::Migration[5.
  def change
    create_join_table :authors, :posts do |t|
      # t.index [:author_id, :post_id]
      # t.index [:post_id, :author_id]
    end
  end
end
```

EVEN COMPLEXERER

Sometimes the join tables are more complex

- Give the concept a name, like `Publications`
- Treat it like a model itself
- Consider needing additional data, like a
  - `primary author` boolean attribute
  - `royalty_earned` money attribute
- Use Timestamps to know when changes happen

Quick detour into

DATABASE THEORY

ACID

A single DB can provide

- Atomicity
- Consistency
- Isolation
- Durability

CAP THEOREM

In the event of a network failure,

a distributed DB must choose between

- consistency
- availability

BASE

Some distributed DBs prioritize availability.

Eventually, all the data will be consistent.

- Basically Available
- Soft state
- Eventual consistency

OTHER TOPICS

ORMS

support multiple databases through

DB ADAPTERS

ORMS

generate queries on the fly, so

are slower than hard-coded queries

ORMS

inflate objects with

data from the DB, which

ISN’T IDEAL FOR BULK WORK

ALTERNATE SQL LIBRARY

like sequel for Ruby

MIGHT BE MORE PERFORMANT

N+1 QUERY EXPLOSIONS

ORMS may lazily fetch data, resulting in

1,000+1 queries for

1,000 posts by 1 author

PRIMARY KEY

Key (like `id`) that uniquely identifies this record

No two records have the same primary key

FOREIGN KEY

Key (like `post_id`) that uniquely identifies some other row in a different table or in this table

WHY REFERENCE A ROW IN THE SAME TABLE?

To allow parent-child relationships

INDEXES

Can speed up lookups, but require extra space on disk and slow down writes

SANITIZATION

NEVER TRUST USER INPUT

SQL injection can cause your DB to be hacked

Some ORMs may sanitize some data by default

RESPONSIBILITY FALLS ON YOU

TO ENSURE DATA IS SANITIZED

before entering your system

PERMISSIONS

Some users might only need read-only access

Review DB permissions to reduce risk of accidental

DELETION OR MODIFICATION

NORMALIZATION

Removing duplication across tables

NORMAL FORMS

- 1NF
- 2NF
- 3NF
  - “Nothing but the key”

CONSTRAINTS

- primary key
- foreign key
- not null
- unique

MIGRATIONS

Changing the database schema as your needs change

MIGRATIONS, CONT.

- Migrating the database by hand is tricky
- Need to be consistent
- Need to do it across many machines (many developers)
- Need to do it differently in different environments
  - PRD has multiple DBs whereas DEV has one local DB

MIGRATIONS, CONT.

- Take databases offline
  - Can make changes that would break your application
- Keep databases online
  - Have to make sure old version of app works with new DB schema
  - Might have to make multiple deployments

MIGRATIONS, CONT.

To keep the db online and rename a column:

- First deploy
  - Duplicate column `old_name` to one called `new_name`
  - Add code that updates `new_name` as well as `old_name`

MIGRATIONS, CONT.

- Second deploy
  - Change all code references from `old_name` to `new_name`
  - Remove now-unused column `old_name`

DATA INTEGRITY

Application-level integrity vs Database-level integrity

- Database can enforce referential integrity
  - Post exists when creating a like
- Application can enforce higher-level things
  - Only a person with specific permissions can modify this post

LOCKS

Adding a non-null column to a large table

can lock the table

and prevent reads or writes

LOCKS, CONT.

To prevent a long table lock:

- Add a column that allows null values
  - Table is only locked for a short time
- Update each row to have a default value
  - This will lock each row for a short time
- Update the column to not allow null values
  - Since each column already has a value, the lock is released quickly

SCALING

- Vertical
  - Faster CPUs
  - More Memory
  - More Disk Space
  - Caching
  - Partitioning data across tables

SCALING, CONT.

- Horizontal
  - Load balancing
  - Create a pool of nodes
  - Add more nodes to the pool
  - Sharding data across nodes

SCALING, CONT.

- Tools to manage bigness
  - gh-ost for migrations
  - Hadoop

FAULT TOLERANCE

If one node goes down,

another can still serve requests

RECOVERY

Make regular backups of databases

RECOVERY, CONT.

Have a restore process

TRY OUT THE RESTORE PROCESS

to make sure it works!

NOSQL DBS

- MongoDB is a document-based store
- Redis is a key-value store
- Cassandra is a column-based store
- Neo4j is a graph-based store

OTHER TOPICS TO CONSIDER

- Data Types
- Views
- Triggers
- Full text search
- Security

OTHER TOPICS TO CONSIDER, CONT.

- Stored Procedures
- Reports/Analytics
- Spatial extensions
- Self-hosted vs Cloud-hosted

LINKS

- Flat files
- Relational model
- Relational database
- SQL
- Tables
- Fields
- Active Record pattern
- Views
- Transactions
- Commits

LINKS, CONT.

- Rollback
- Distributed data store
- Normalization overview
- Rails ActiveRecord Basics
- Use the Index, Luke
- Python ORMs
- Khan Academy SQL Basics
- 101 things I wish I knew…

DROP POSTGRES TABLE

```
dropdb dbs_and_devs_talk
```

THANKS!