Database Systems

Shan-Hung Wu

CS, NTHU

Outline

- Why DBMS?
- Data modeling
- SQL queries
- WeatherMood + DBMS
- Managing “big” data
  - Text indexing
  - Pagination
- Deployment

Outline

- Why DBMS?
- Data modeling
- SQL queries
- WeatherMood + DBMS
- Managing “big” data
  - Text indexing
  - Pagination
- Deployment

Why not file systems?

Advantages of a Database System

- It answers ***queries*** fast
  - E.g., among all posts, find those written by Bob and contain word “db”
- Groups modifications into ***transactions*** such that either all or nothing happens
  - E.g., money transfer
- Recovers from crash
  - Modifications are logged
  - No corrupt data after recovery

Advantages of a Database System

- It answers ***queries*** fast
  - E.g., among all posts, find those written by Bob and contain word “db”
- Groups modifications into ***transactions*** such that either all or nothing happens
  - E.g., money transfer
- Recovers from crash
  - Modifications are logged
  - No corrupt data after recovery

Queries

Q: find ID and text of all pages written by Bob and containing word “db”

Step1: structure data using ***tables***

users

id	name	karma
729	Bob	35
730	John	0

column

posts

id	text	ts	authorId
33981	‘Hello DB!’	1493897351	729
33982	‘Show me code’	1493854323	812

row

Queries

Q: find ID and text of all pages written by Bob and containing word “db”

Step2:

```
SELECT p.id, p.text
FROM posts AS p, users AS u
WHERE u.id = p.authorId
      AND u.name='Bob'
      AND p.text ILIKE '%db%';
```

users

id	name	karma
729	Bob	35
730	John	0

posts

id	text	ts	authorId
33981	‘Hello DB!’	1493897351	729
33982	‘Show me code’	1493904323	812

How Is a Query Answered?

```
SELECT p.id, p.text
FROM posts AS p, users AS u
WHERE u.id = p.authorId
      AND u.name='Bob'
      AND p.text ILIKE '%db%';
```

(p, u)

p.id	p.text	p.ts	p.authorId	u.id	u.name	u.karma
33981	'Hello DB!'	...	729	729	Bob	35
33981	'Hello DB!'	...	729	730	John	0
33982	'Show me code'	...	812	729	Bob	35
33982	'Show me code'	...	812	730	John	0

p

id	text	ts	authorId
33981	'Hello DB!'	...	729
33982	'Show me code'	...	812

u

id	name	karma
729	Bob	35
730	John	0

How Is a Query Answered?

```
SELECT p.id, p.text
FROM posts AS p, users AS u
WHERE u.id = p.authorId
      AND u.name='Bob'
      AND p.text ILIKE '%db%';
```

where(p, u)

p.id	p.text	p.ts	p.authorId	u.id	u.name	u.karma
33981	'Hello DB!'	...	729	729	Bob	35

(p, u)

p.id	p.text	p.ts	p.authorId	u.id	u.name	u.karma
33981	'Hello DB!'	...	729	729	Bob	35
33981	'Hello DB!'	...	729	730	John	0
33982	'Show me code'	...	812	729	Bob	35
33982	'Show me code'	...	812	730	John	0

How Is a Query Answered?

```
SELECT p.id, p.text
FROM posts AS p, users AS u
WHERE u.id = p.authorId
      AND u.name='Bob'
      AND p.text ILIKE '%db%';
```

select(where(p, u))

p.id	p.text
33981	'Hello DB!'

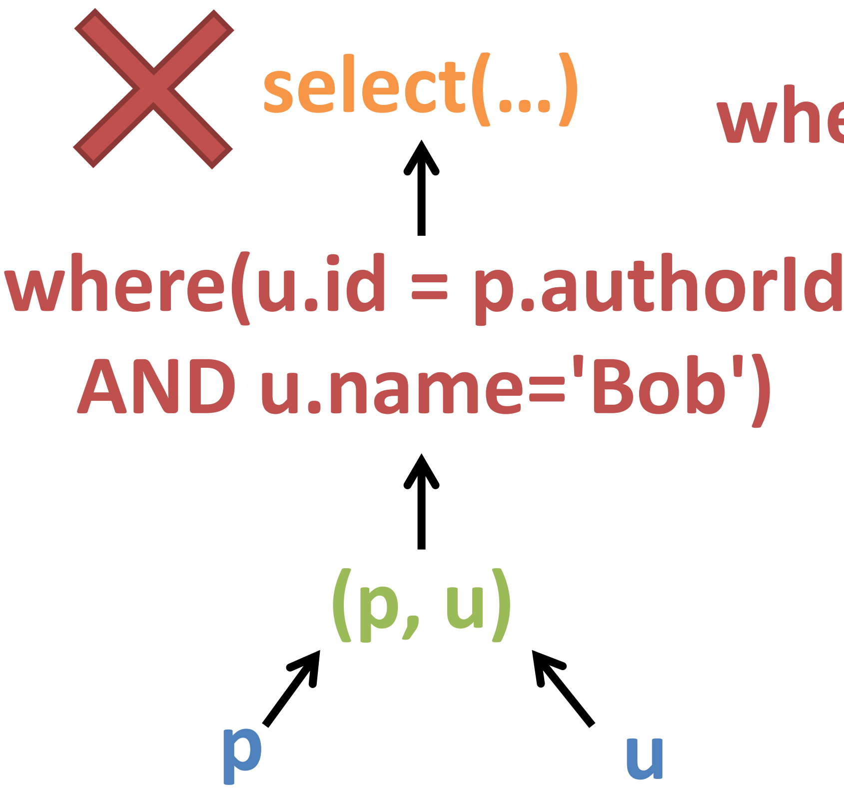
where(p, u)

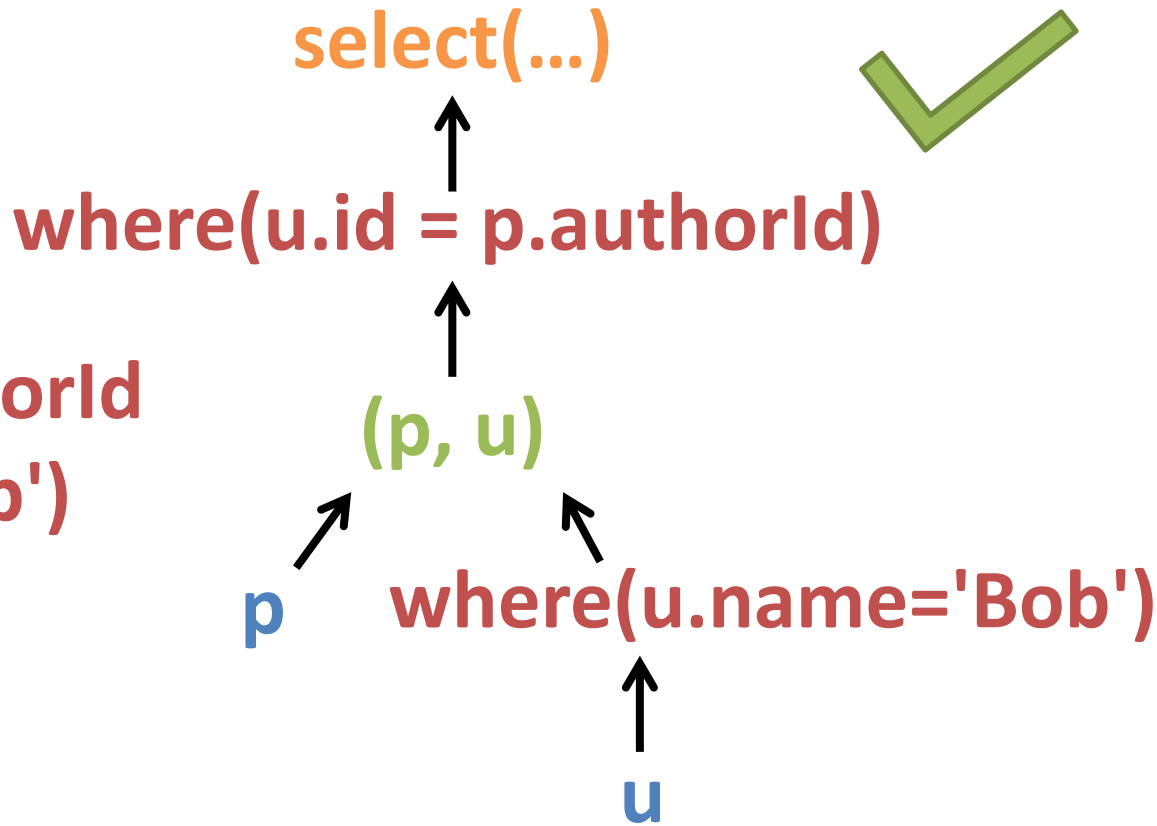
p.id	p.text	p.ts	p.authorId	u.id	u.name	u.karma
33981	'Hello DB!'	...	729	729	Bob	35

Why fast?

Query Optimization

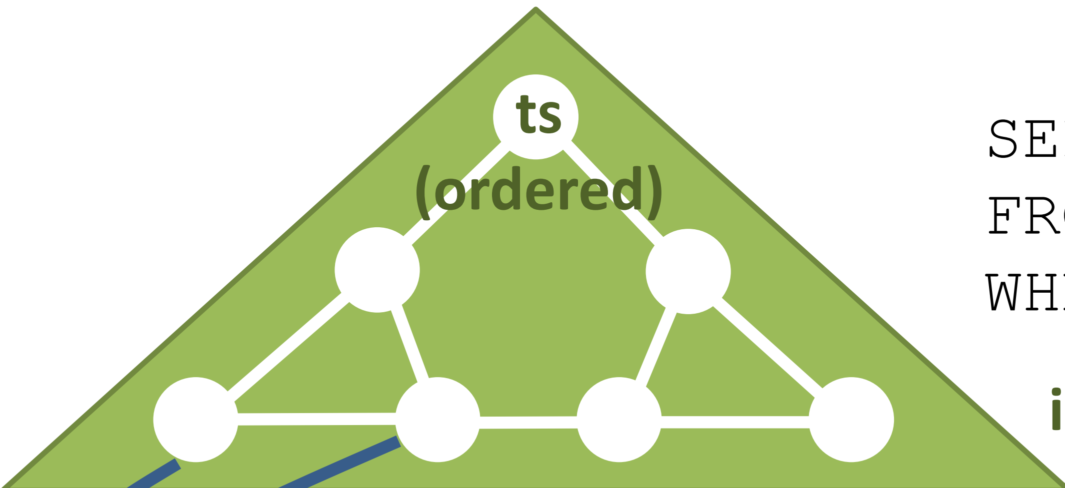
- ***Planning***: DBMS finds the best ***plan tree*** for each query





Query Optimization

- ***Indexing***: creates a search tree for column(s)




```
SELECT text
FROM posts
WHERE ts > 1493897220;
```

posts

id	text	ts	authorId
1	‘Good day’	1493880220	664
…	…	…	…
33981	‘Hello DB!’	1493897351	729
33982	‘Show me code’	1493904323	812

Advantages of a Database System

- It answers ***queries*** fast
  - E.g., among all posts, find those written by Bob and contain word “db”
- Groups modifications into ***transactions*** such that either all or nothing happens
  - E.g., money transfer
- Recovers from crash
  - Modifications are logged
  - No corrupt data after recovery

Transactions I

- Each query, by default, is placed in a ***transaction*** (***tx*** for short) automatically

```
BEGIN;
   SELECT ...; -- query
COMMIT;
```

Transactions II

- Can group multiple queries in a tx
  - ***All or nothing*** takes effect
- E.g., karma transfer

users

id	name	karma
729	Bob	35
730	John	0

```
BEGIN;
  UPDATE users
  SET karma = karma - 10
  WHERE name='Bob';

  UPDATE users
  SET karma = karma + 10
  WHERE name='John';
COMMIT;
```

ACID Guarantees

- ***Atomicity***
  - Operation are all or none in effect
- ***Consistency***
  - Data are correct after each tx commits
  - E.g., `posts.authorId` must be a valid `users.id`
- ***Isolation***
  - Concurrent txs = serial txs (in some order)
- ***Durability***
  - Changes will not be lost after a tx commits

Outline

- Why DBMS?
- Data modeling
- SQL queries
- WeatherMood + DBMS
- Managing “big” data
  - Text indexing
  - Pagination
- Deployment

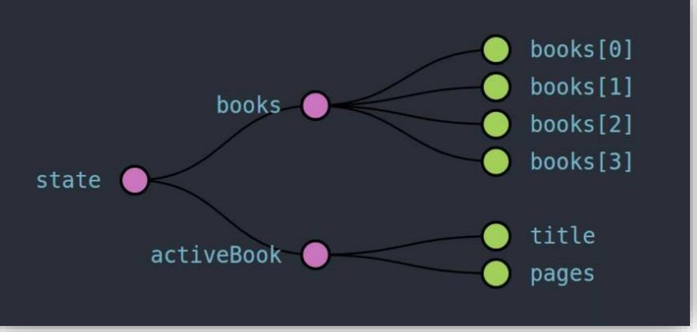
Why model data as ***tables***?

users

id	name	karma
729	Bob	35
730	John	0

posts

id	text	ts	authorId
33981	'Hello DB!'	1493897351	729
33982	'Show me code'	1493904323	812



Tree Model

- At client side, data are usually stored as ***trees***

```
{ // state of client 1
  name: 'Bob',
  karma: 32,
  posts: [...],
  friends: [{
    name: 'Alice',
    karma: 10
  }, {
    name: 'John',
    karma: 17
  }, ...],
  ...
}
```

```
{ // state of client 2
  name: 'Alice',
  karma: 10,
  posts: [...],
  friends: [{
    name: 'Bob',
    karma: 32
  }, {
    name: 'John',
    karma: 17
  }, ...],
  ...
}
```

Problems at Server Side

- Space: large ***redundancy***

```
{ // state of a client 1
  name: 'Bob',
  karma: 35,
  posts: [...],
  friends: [{
    name: 'Alice',
    karma: 10
  }, {
    name: 'John',
    karma: 17
  }, ...],
  ...
}
```

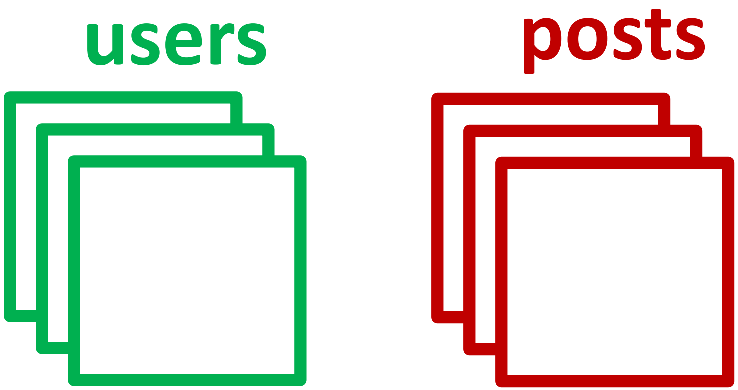
```
{ // state of a client 2
  name: 'Alice',
  karma: 10,
  posts: [...],
  friends: [{
    name: 'Bob',
    karma: 35
  }, {
    name: 'John',
    karma: 17
  }, ...],
  ...
}
```

Speed: slow update

Data Modeling at Server Side

1. Identify ***entity classes***
   – Each class represents an “atomic” part of the data
2. Store entities of the same class in a ***table***

Identifying Entity Classes

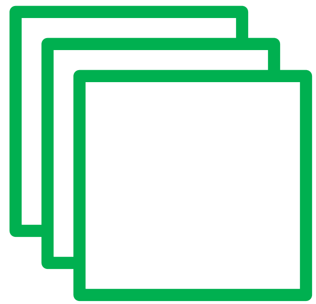



```
{ // state of a client 1
  name: 'Bob',
  karma: 32,
  posts: [...],
  friends: [{
    name: 'Alice',
    karma: 10
  }, {
    name: 'John',
    karma: 17
  }, ...],
  ...
}
```

```
{ // state of a client 2
  name: 'Alice',
  karma: 10,
  posts: [...],
  friends: [{
    name: 'Bob',
    karma: 32
  }, {
    name: 'John',
    karma: 17
  }, ...],
  ...
}
```

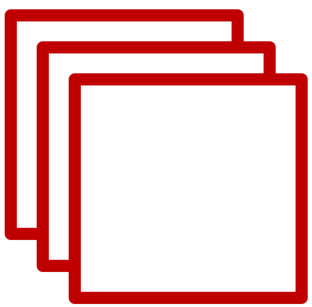
One Table per Entity Class

users



id	name	karma
729	Bob	35
730	John	0

posts



id	text
33981	‘Hello DB!’
33982	‘Show me code’

- No redundancy
- No repeated update

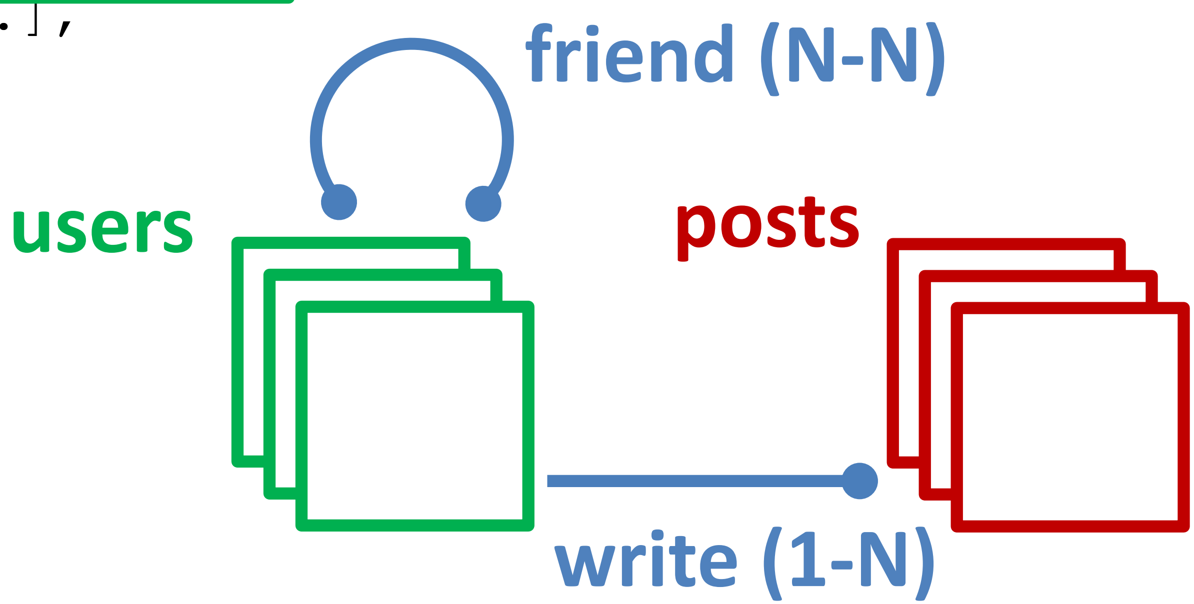
Wait, relationship is missing!

Step1 (ER Model)

```
{ // state of a client 1
  name: 'Bob',
  karma: 32,
  posts: [...],
  friends: [{
    name: 'Alice',
    karma: 10
  }, {
    name: 'John'
    karma: 17
  }, ...],
  ...
}
```

friend

write



Step 2 (Relational Model)

users

id	name	karma
729	Bob	35
730	John	0

friend

uId1	uId2	since
729	730	14928063
729	882	14827432

foreign keys

posts / **write**

id	text	authorId	ts
33981	Hello DB!’	729	1493897351
33982	Show me code’	729	1493854323

Terminology

- Columns: fields
- Rows: records, tuples
- Tables: relations

- Database: a collection of tables
  – ≠ database system
- ***Schema***: column definitions of tables in a database
  – Basically, the “look” of a database

Outline

- Why DBMS?
- Data modeling
- SQL queries
- WeatherMood + DBMS
- Managing “big” data
  - Text indexing
  - Pagination
- Deployment

PostgreSQL

- Download and install
- For Mac users, try PostgreSQL.app instead

SQL

```
$ createdb <db>
$ psql <db> [user]
> \h or \?
> SELECT now(); -- SQL commands
```

- Default schema: `public` (`\dn`)
- Multiple lines until ‘;’
- ‘--’ for comments
- Case insensitive
  - Use “” to distinguish lower and upper cases
  - E.g., `SELECT "authorId" FROM posts;`

Create Tables/Relations

```
CREATE TABLE posts (
  id             serial PRIMARY KEY NOT NULL,
  mood           mood NOT NULL,
  text           text NOT NULL,
  ts             bigint NOT NULL
                 DEFAULT (extract(epoch from now())),
  "clearVotes"   integer NOT NULL DEFAULT 0,
  "cloudsVotes" integer NOT NULL DEFAULT 0,
  ...
);
CREATE TYPE mood AS ENUM ('Clear', 'Clouds', ...);
```

- Primary key: unique, has index
- Column types: varchar(10), integer, bigint, real, double, string, etc.

Insert Rows

```
INSERT INTO posts(mood, text)
VALUES ('Clear', 'Today is a good day!');
```

- String values should be ***single*** quoted
- To generate dummy rows:

```
INSERT INTO posts(mood, text, "clearVotes")
SELECT
  'Clear',
  'word' || i || ' word' || (i+1),
  round(random() * 10)
FROM generate_series(1, 20) AS s(i);
```

Queries

```
SELECT *
FROM posts
WHERE ts > 147988213 AND text ILIKE '%good%'
ORDER BY ts DESC, id ASC
LIMIT 2;
```

- To see how a query is processed:

```
EXPLAIN ANALYZE -- show plan tree
SELECT *
FROM posts
WHERE ts > 147988213 AND text ILIKE '%good%'
ORDER BY ts DESC, id ASC
LIMIT 2;
```

Outline

- Why DBMS?
- Data modeling
- SQL queries
- WeatherMood + DBMS
- Managing “big” data
  – Text indexing
  – Pagination
- Deployment

Branch `db`

- In `weathermood-server` project

```
$ npm install --save pg-promise
```

- pg-promise
  - A PostgreSQL client library for Node.js
  - Supports ES6 Promise

pg-promise

```
if (!global.db) {
  const pgp = require('pg-promise')();
  db = pgp(`postgres://${u}:${p}@${h}:5432/${db}`);
}
let param= ...;
db.one( // or many(), none(), any(), etc.
  'SELECT * FROM posts WHERE id = $1',
  param
).then(data => {
  ... // data is a post object (columns as props)
}).catch(...);
```

- One shared global `db` per DB client
- pg-promise maintains a pool of connections

Connection Pooling

- Our DB client (web server) handles many requests
- One conn. per request?
  - Repeated conn. establish time
  - Long delay & excessive memory usage on PostgreSQL
- Shared conn. between all requests?
  - PostgreSQL allows only 1 query at a time per conn.
  - No concurrent queries; low throughput
- `db` delegates a query to a conn. in a pool
  - Short delay + concurrent queries
  - All conns. busy → wait; no excessive memory usage

WeatherMood Schema

```
$ npm run schema // or
$ node src/model/schema.js
```

- It's a good idea to keep sensitive info (e.g, password) out of project
- Use environment variables:

```
$ export <key>=<value> // or
$ source ./env.sh       // mac or unix
$ ./env.bat             // windows
```

git ignored

```
// in js
const <value> = process.env.<key>;
```

DB-based Model I

```
db.one( // or many(), none(), any(), etc.
  'SELECT * FROM posts WHERE id = $1',
  param
).then(data => {
  ... // data is a post object (columns as props)
}).catch(err => {...});
```

- ***Never concatenate user input*** for SQL:

```
'SELECT * FROM posts WHERE id = ' + input
```

- SQL injection:

```
'SELECT * FROM posts WHERE id = ' + '0; SELECT ...'
```

DB-based Model II

```
db.one( // or many(), none(), any(), etc.
  'SELECT * FROM posts WHERE id = $1',
  param
).then(data => {
  ... // data is a post object (columns as props)
}).catch(err => {...});
```

- Formatting:
  - `$1`, `$2`, ... for array/primitive `param`
  - `$<prop>` for object `param`
- `$1:name` (or ~): escaped and double-quoted
- `$1:value` (or #): escaped, no quote

DB-based Model III

```
db.one( // or many(), none(), any(), etc.
   'SELECT * FROM posts WHERE id = $1',
   param
).then(data => {
   ... // data is a post object (columns as props)
}).catch(err => {...});
```

- `data` in callback:
  - `none`: null
  - `one`: an object (props as columns in SELECT)
  - `many`: an array of objects

Outline

- Why DBMS?
- Data modeling
- SQL queries
- WeatherMood + DBMS
- Managing “big” data
  - Text indexing
  - Pagination
- Deployment

Be Prepared for “Big” Data

```
INSERT INTO posts(mood, text, "clearVotes")
SELECT
  'Clear',
  'word' || i || ' word' || (i+1),
  rount(random() * 10)
FROM generate_series(1, 1000000) AS s(i);
```

- Some queries will be long:

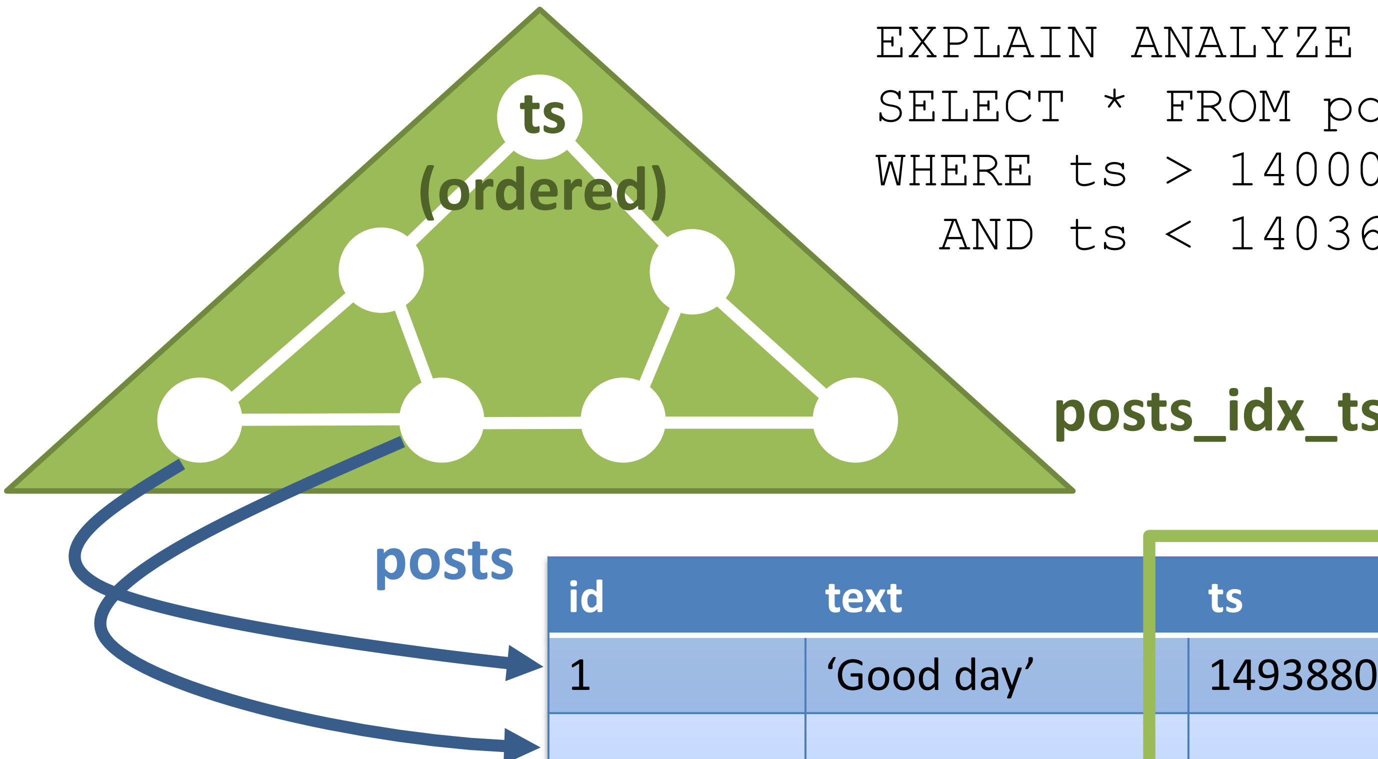
```
EXPLAIN ANALYZE SELECT * FROM posts
WHERE id > 500000 AND id < 501000; -- 1ms

EXPLAIN ANALYZE SELECT * FROM posts
WHERE ts > 1400000000 AND ts < 1403600000; -- 230ms
```

Branch
db-large

```
CREATE INDEX posts_idx_ts
ON posts
USING btree(ts);
\di

EXPLAIN ANALYZE
SELECT * FROM posts
WHERE ts > 1400000000
  AND ts < 1403600000; -- 2ms
```



posts

id	text	ts
1	‘Good day’	1493880220
…	…	…
33981	‘Hello DB!’	1493897351
33982	‘Show me code’	1493904323

Indexing for `ILIKE`?

```
CREATE INDEX posts_idx_text ON posts
USING btree(text);

EXPLAIN ANALYZE SELECT * FROM posts
WHERE text ILIKE '%word500000%'; -- 1.5s
```

- B-tree indices are ***not*** helpful for text searches
- Use GIN (generalized inverted index) instead:

```
CREATE EXTENSION pg_trgm;
\dx
CREATE INDEX posts_idx_text ON posts
USING gin(text gin_trgm_ops);

EXPLAIN ANALYZE SELECT * FROM posts
WHERE text ILIKE '%word500000%'; -- 50ms
```

Outline

- Why DBMS?
- Data modeling
- SQL queries
- WeatherMood + DBMS
- Managing “big” data
  - Text indexing
  - Pagination
- Deployment

Pagination

```
SELECT table_name, pg_relation_size(table_name)
FROM information_schema.tables
WHERE table_schema = 'public';
```

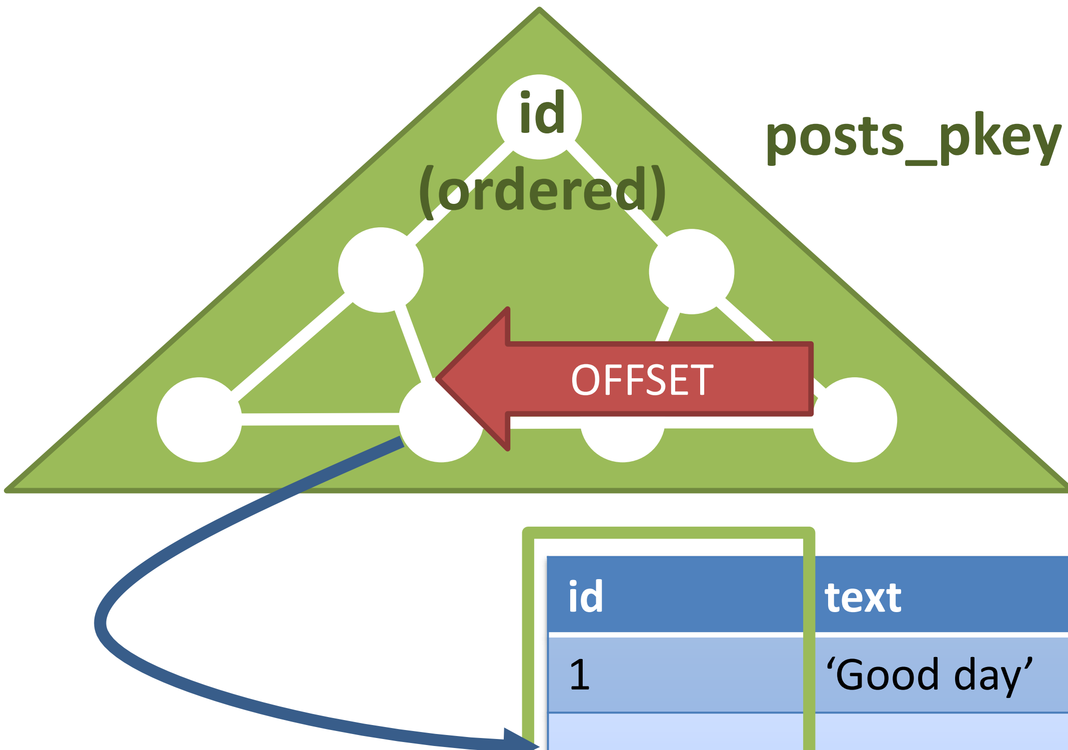
- 1M posts take about 116MB (compressed)

- Impractical to transmit all via `GET /posts`
- Solution?
- `GET /posts?page=$<page>`

SQL OFFSET

```
SELECT * FROM posts
WHERE ...
ORDER BY id DESC
LIMIT 20 OFFSET ($<page> - 1) * 20; -- 20 posts/page
```

- Problem: slower at later pages



posts

id	text	ts
1	‘Good day’	1493880220
…	…	…
33981	‘Hello DB!’	1493897351
33982	‘Show me code’	1493904323

Keyset Pagination

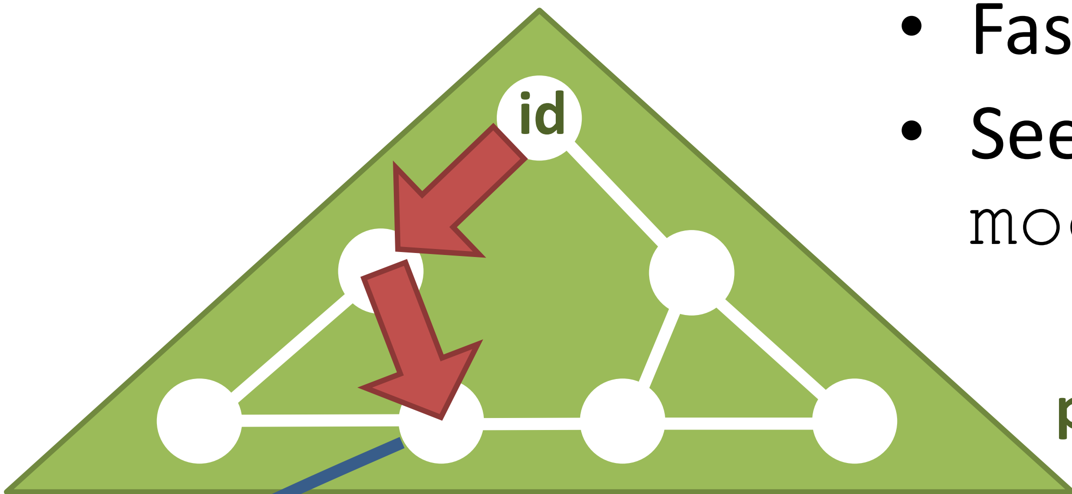
```
SELECT * FROM posts
WHERE ...
ORDER BY id DESC
LIMIT 20; -- 20 posts/page
```

- Assumption: each row has a unique position in `ORDER BY`
- Let the client passes the ***last row in the previous page***
- `GET /posts?start=$<start>`

```
SELECT * FROM posts
WHERE ... AND id < $<start>
ORDER BY id DESC
LIMIT 20; -- 20 posts/page
```

SQL WHERE

- Fast at every page!
- See `list()` in `model/posts.js`



posts

id	text	ts
1	‘Good day’	1493880220
…	…	…
33981	‘Hello DB!’	1493897351
33982	‘Show me code’	1493904323

Client-Side Code

- Checkout branch `server-db-large` of project `weathermood`
- Redux (async) actions:
  - `listPosts()`: GET the first page
  - `listMorePosts()`: GET the next page
  - `createPost()`: POST and then merge
  - `createVote()`: POST and then merge

Infinite Scrolling

```
$ npm install --save react-infinite-scroller
```

```
// in PostList.jsx
<InfiniteScroll loadMore={loadFunc} hasMore={true}>
  {items}
</InfiniteScroll>
```

- Read [more](#) about usage

Access Control

- Common in client-server joint development:

1. Load HTML page (and JS) from site A
2. Send AJAX request to site B
3. Browser ***blocks*** response of B if no `Access-Control-Allow-Origin` header in response

- To allows cross-origin HTTP requests:

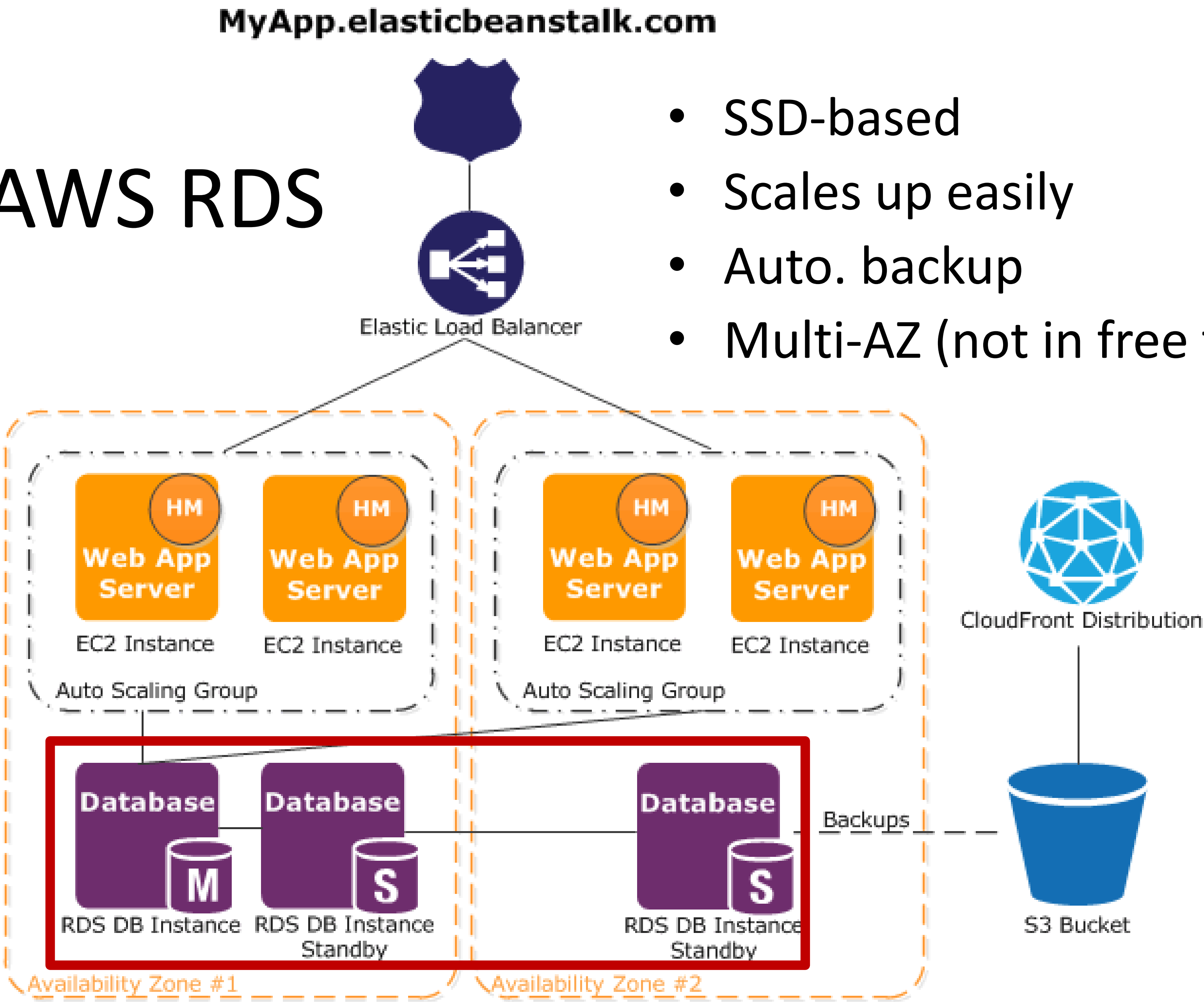
```
// in routers/posts.js (at server side)
router.use(accessController);
```

Outline

- Why DBMS?
- Data modeling
- SQL queries
- WeatherMood + DBMS
- Managing “big” data
  – Text indexing
  – Pagination
- Deployment

AWS RDS

- SSD-based
- Scales up easily
- Auto. backup
- Multi-AZ (not in free tier)



EB + RDS

- Better give decoupled lifecycles
- AWS Console → Services → RDS → Launch DB Instance → PostgreSQL → Dev/Test (free tier)
- Set up RDS security group to allow ingress from machines in the same group
  - In EB console, add ec2 instances to RDS security group

Schema/Data Migration

1. In RDS console, set Public Accessible = true
2. Set up RDS security group to allow ingress from your IP address
3. Migrate schema/data (client-side migration):

```
$ pg_dump -h <dev-server> -U <dev-user> \
  --no-owner [--schema-only] -c \
  weathermood > db.dump

$ psql -h <rds-endpoint> -U <rds-user> \
  weathermood < db.dump
```

4. Set Public Accessible = false

EB Deployment

1. Add environment variables on EB web servers (EC2 instances):

```
$ eb setenv NODE_ENV=production, \
  RDS_HOSTNAME=<rds-endpoint>, RDS_PORT=5432, \
  RDS_USERNAME=<user>, RDS_PASSWORD=<password>, \
  RDS_DB_NAME=weathermood
```

2. Commit and deploy:

```
& git commit
$ eb deploy weathermood-production
```

Assigned Readings

- SQL language tutorial

- Optional:
- Advanced features using PostgreSQL:
  - Full text search
  - Task queue
- Available extensions on AWS RDS PostgreSQL

Assignment: DB-Backed TODOs

- DB schema design
- Model and SQL queries
- Client-server joint development (pagination)
- AWS Deployment (one per group)