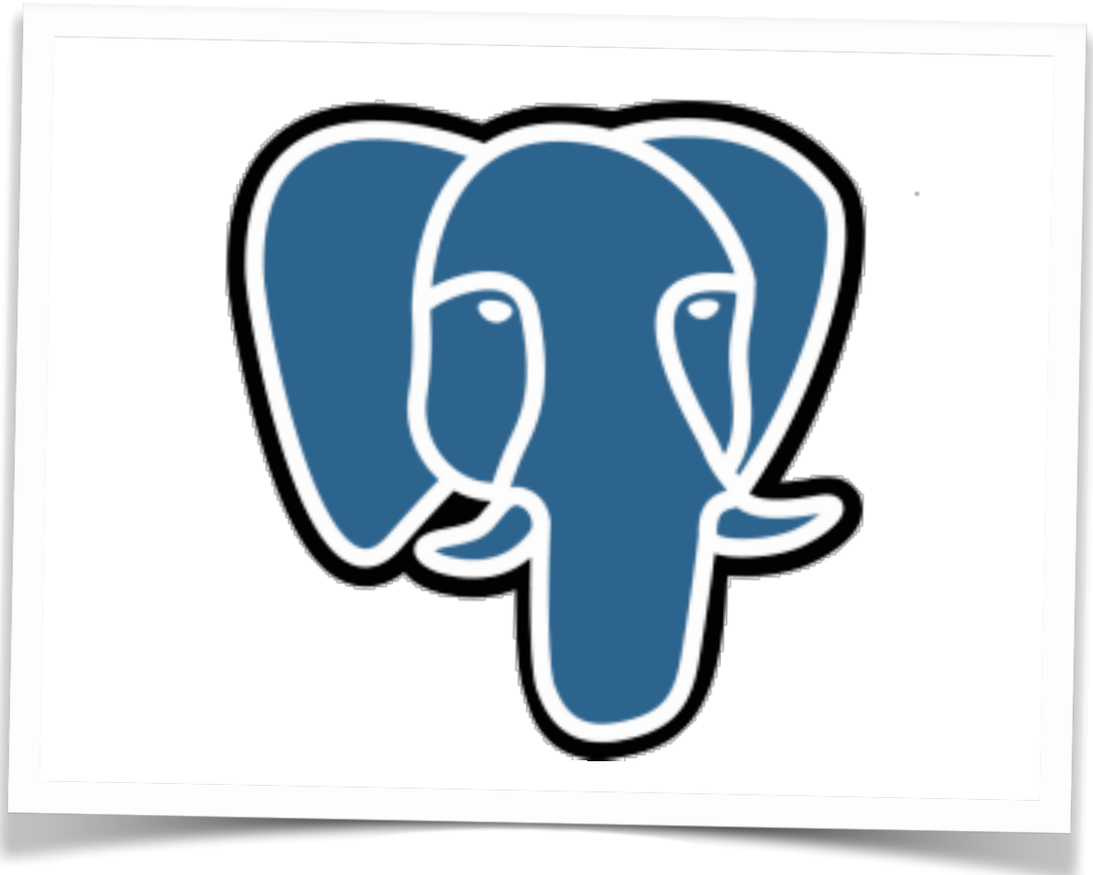

D·A·T·A·B·A·S·E·S

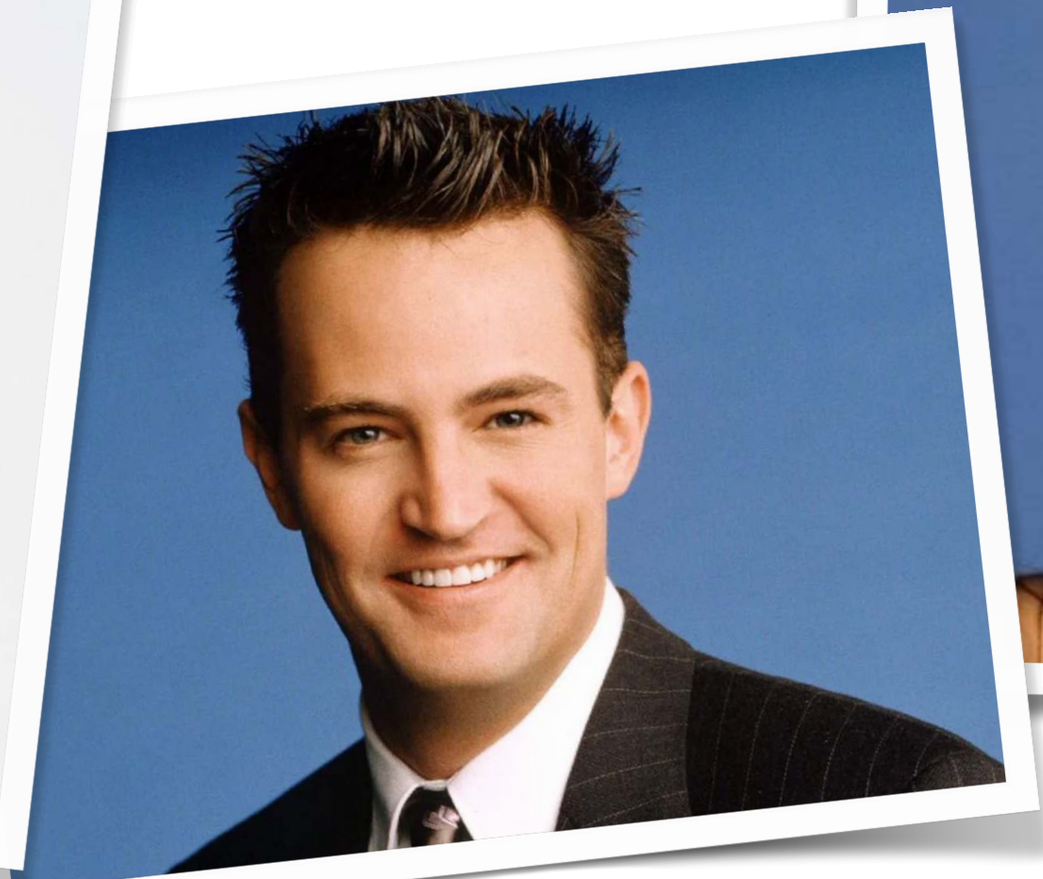
D·A·T·A·B·A·S·E·S



guest appearance PostgreSQL

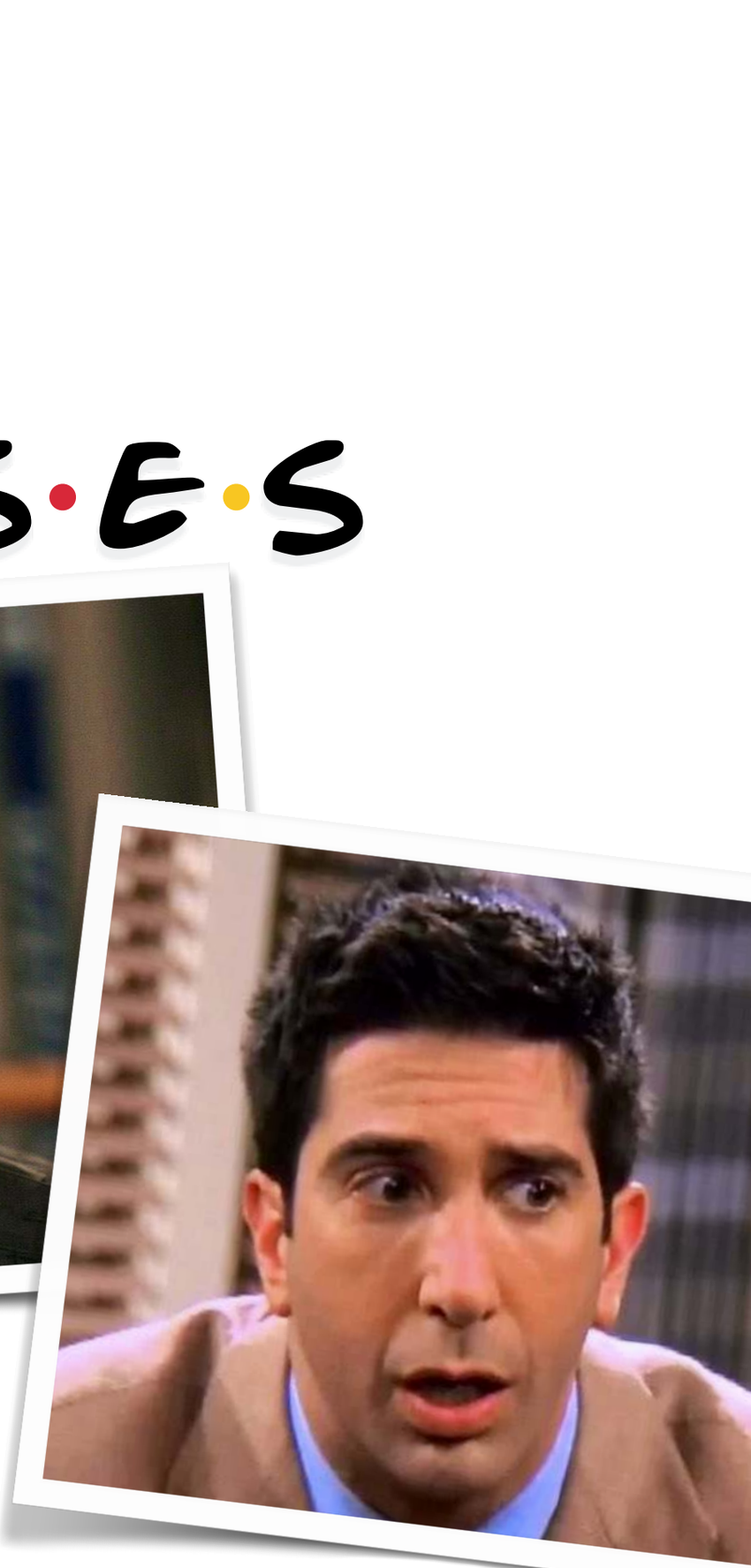
with helpful tips from

D·A·T·A·B·A·S·E·S



with helpful tips from

D·A·T·A·B·A·S·E·S



D·A·T·A·B·A·S·E·S

(insert CouchDB joke here)

What are document databases

with Ross

Relational schema

id	name	year
1	Alex	1993
2	Bob	1984
3	Ceasar	1975
4	Dorothee	1977
5	Engelbert	1997
6	Friedrich	1996
7	Gustav	1997
8	Hodor	1992
9	Ingrid	1994

"relational schema" – circa 1970, colorized, Edgar Codd

Relational schema

Actors

id	name	year	movie_id
1	Alex	1971	1
2	Bob	1979	2
3	Ceasar	1999	1
4	Dorothee	1997	5
5	Engelbert	1972	4
6	Friedrich	1997	2
7	Gustav	1980	3
8	Hodor	1984	3
9	Ingrid	1993	3

Movies

id	name
1	The Great Gatsby
2	Last Christmas
3	Aristocats
4	Brazil

"2NF - 3NF" – circa 1970, colorized, Edgar Codd

Flow-based programming paradigm

```
CREATE TABLE table_output
SELECT * FROM table_input
```

table_input

id	foo	bar
1	123	1980
2	456	1986
3	789	1999

table_output

id	zig
1	123-abc
2	456-def
3	789-hij

"2NF - 3NF" – circa 1970, colorized, Edgar Codd

Documents


```
{
 "id": 1,
 "kind": "actor",
 "name": "Alex",
 "year": 1973,
 "movie_id": 1
}
```

```
{
 "id": 2,
 "kind": "actor",
 "name": "Bob",
 "year": 1992,
 "movie_id": 2
}
```

```
{
 "id": 3,
 "kind": "actor",
 "name": "Ceasar",
 "year": 1997,
 "movie_id": 1
}
```

Documents


```
{
 "id": 1,
 "kind": "actor",
 "name": "Alex",
 "year": 1973,
 "movie_id": 1
}
```

```
{
 "id": 2,
 "kind": "actor",
 "name": "Bob",
 "year": 1992,
 "movie_id": 2
}
```

```
{
 "id": 3,
 "kind": "actor",
 "name": "Ceasar",
 "year": 1997,
 "movie_id": 1
}
```

```
{
 "id": 1,
 "kind": "movie",
 "name": "The Great
 Gatsby"
}
```

```
{
 "id": 2,
 "kind": "movie",
 "name": "Last
 Christmas"
}
```

"If you are storing
objects as a whole,
you are *using* a
Document Database."

Captain Obvious Ross

Advantages of document databases

```
{
  "id": 1,
  "kind": "actor",
  "data": {
    "name": "Chandler",
    "year": 1973,
    "movie_id": 1
  }
}
```

with Monica

- Dynamic schema extension
  even at runtime
- Data closely modelled by application code
  not for the database
- Minimize one-to-one relationships
  through nested entities

- Allows for k/v access patterns
  fast reads, fast atomic writes

- Portable between systems
  the schema travels with the data

- Faster development cycles
  less schema, less decisions to make upfront

Practical design considerations

with Chandler

Practical design considerations

```
{
  "id": 1,
  "kind": "actor",
  "data": {
    "name": "Chandler",
    "year": 1973,
    "movie_id": 1
  }
}
```

Separate top level from data attributes

have one schema for all documents

Practical design considerations

```
{
  "id": 1,
  "kind": "actor",
  "data": {
    "name": "Chandler",
    "year": 1973,
    "movie_id": 1
  }
}
```

Define the type at the top level

the type tells the schema and structure of the data

Practical design considerations

```
{
  "id": 1,
  "kind": "actor",
  "version": 4,
  "data": {
    "name": "Chandler",
    "year": 1973
  }
}
```

Track the data schema
with a version field

Practical design considerations

```
{
  "id": 1,
  "kind": "actor",
  "version": 4,
  "data": {
    "name": "Chandler",
    "year": 1973
  }
}
```

Remember to have fun
during development

Schema-*less*?

- Choose your own poison
  no outside restrictions
  more freedom of choice

- Less overhead during development
  fewer decisions to make upfront

- Less normalization
  carefully selected composition

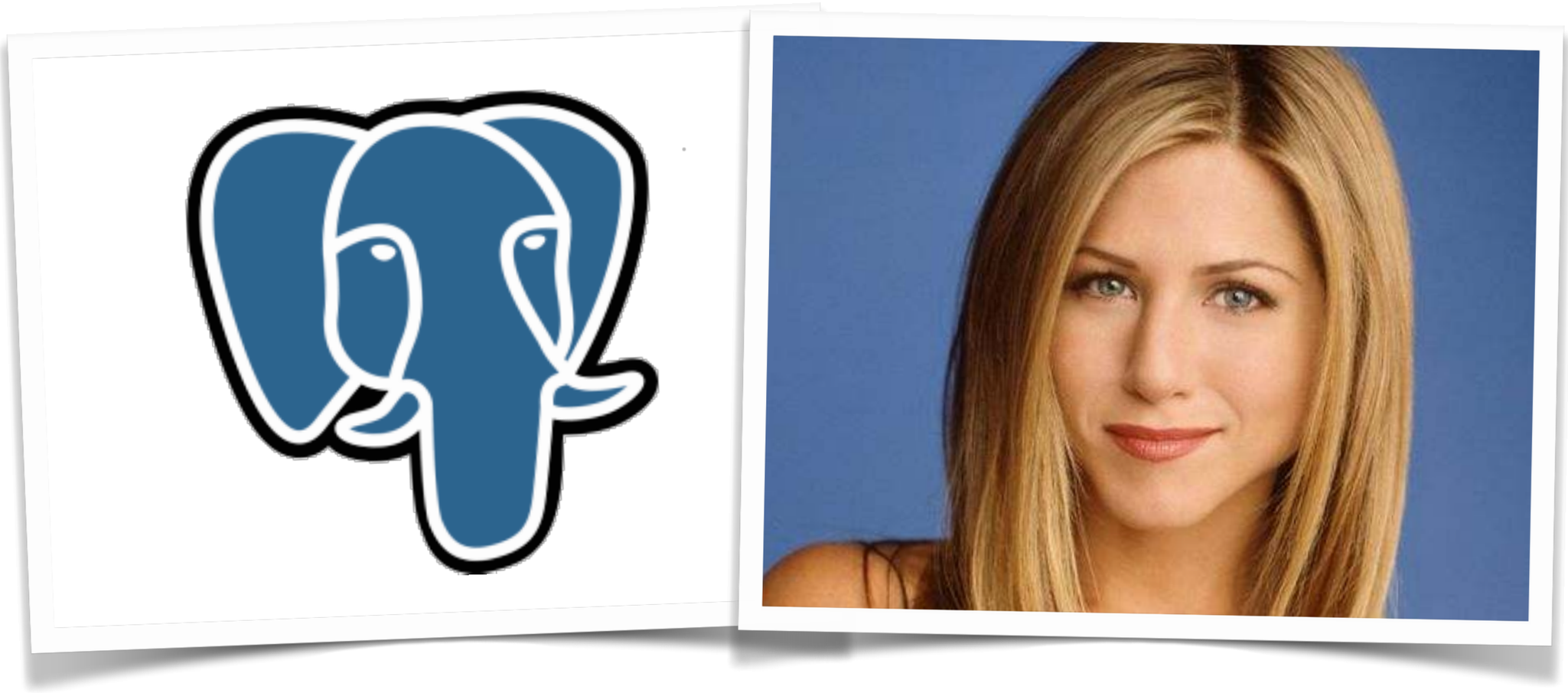
- Less schema
  != no schema

by Phoebe

"Remember, you want
to end up with *less*,
not with a *mess*"

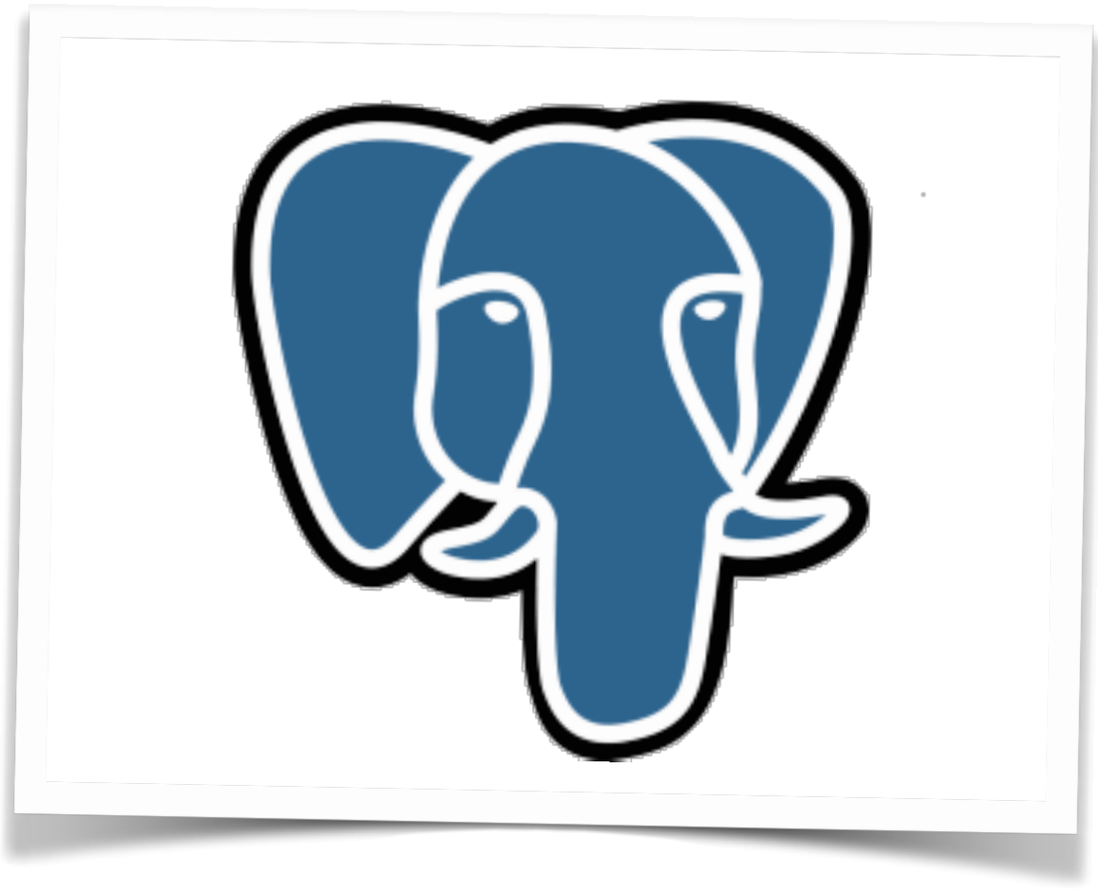
Chandler

Storing documents in PostgreSQL

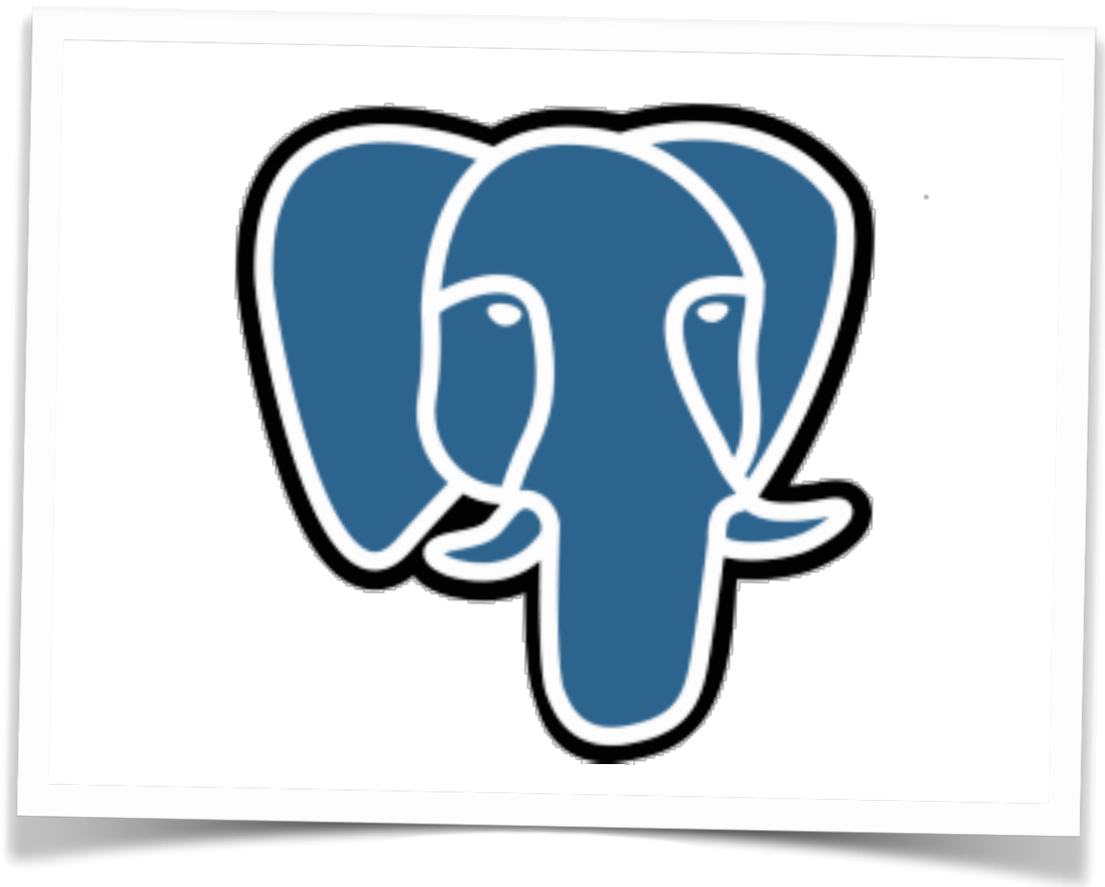


with Rachel
& Postgres

Storing documents in PostgreSQL



Combining the best
of both worlds



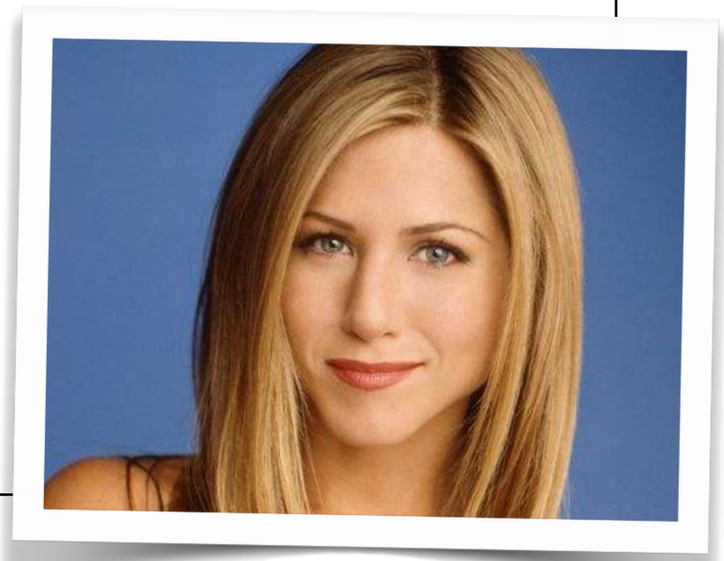
- JSON/JSONB columns
  many native functions
  and operators

- Table inheritance
  Table partitioning
  maintain and develop
  your document storage

- Generated columns
  Materialized views
  normalise and combine

Practical PostgreSQL: Query into JSONB

```
SELECT * FROM documents
WHERE kind = 'actor' AND data->>'age' > 32

SELECT count(*) FROM documents
WHERE data @> '{"hobbies" :"snowboarding"}';
```



Practical PostgreSQL: Index JSONB

- `CREATE INDEX idx_data_age ON documents USING BTREE ((data->>'age'));`

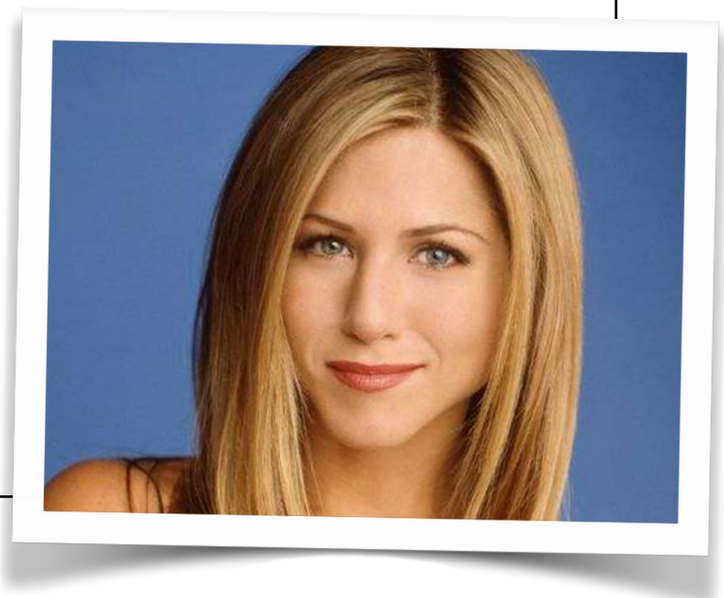
- `CREATE INDEX idx_data_age ON documents USING BTREE ((data->>'age')) WHERE (kind = 'actor');`

- `CREATE INDEX idx_data ON documents USING GIN (data jsonb_ops);`

- `CREATE INDEX idx_data ON documents USING GIN (data jsonb_path_ops);`

Practical PostgreSQL: Table inheritance

```
CREATE TABLE documents (
    kind character varying(255),
    id   character varying(255),
    data jsonb,
    CONSTRAINT documents_pkey PRIMARY KEY (kind, id)
);
```



Practical PostgreSQL: Table inheritance

```
CREATE TABLE documents (
    kind character varying(255),
    id   character varying(255),
    data jsonb,
    CONSTRAINT documents_pkey PRIMARY KEY (kind, id)
);

CREATE TABLE actors (
    ...
) INHERITS (documents);
```

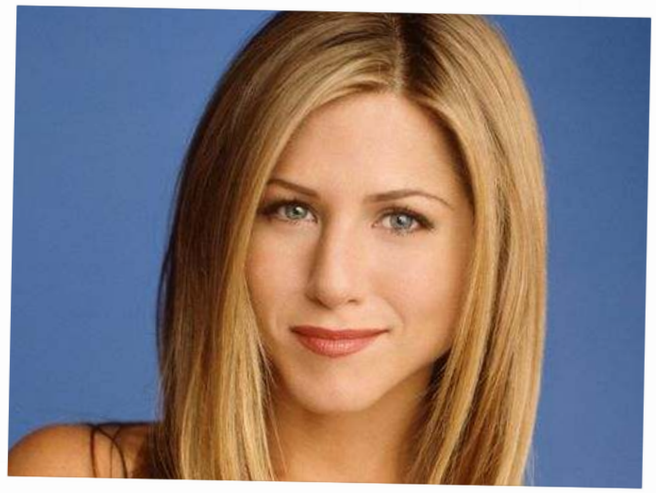
- Evolve your document schema independently
- Re-introduce relational and normalisation features
- Scale document schemas naturally across the table space

Table inheritance

By Joey

Practical PostgreSQL: Table partitioning (LIST)

```
CREATE TABLE documents (
    kind character varying(255),
    id   character varying(255),
    data jsonb,
    CONSTRAINT documents_pkey PRIMARY KEY (kind, id)
) PARTITION BY LIST (kind);

CREATE TABLE actors
PARTITION OF documents FOR VALUES IN ('actor'
```



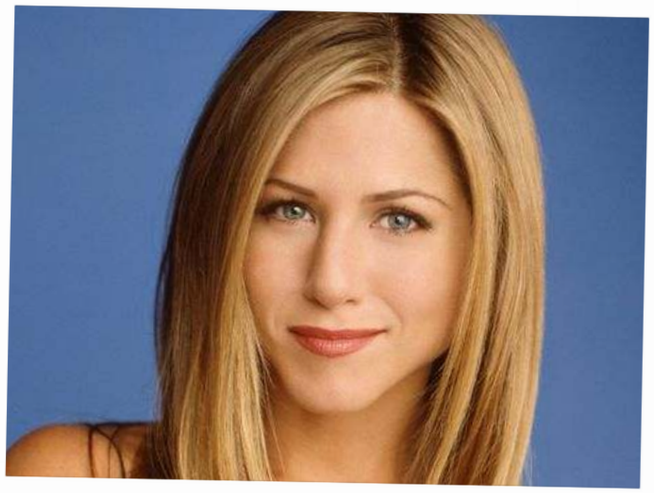
- Separate read/write targets and optimize access patterns
- Add data-type specific indices
- Optimize ANALYZE performance
  by spreading out tables

Table partitioning (LIST)

By Joey

Practical PostgreSQL: Table partitioning (RANGE)

```
CREATE TABLE events (
    kind    character varying(255),
    id      character varying(255),
    logdate timestamptz,
    data    jsonb,
    CONSTRAINT events_pkey PRIMARY KEY (kind, id)
) PARTITION BY RANGE (kind);

CREATE TABLE events_2019_11
PARTITION OF events
FOR VALUES FROM ('2019-11-01')
             TO ('2019-12-01');
```



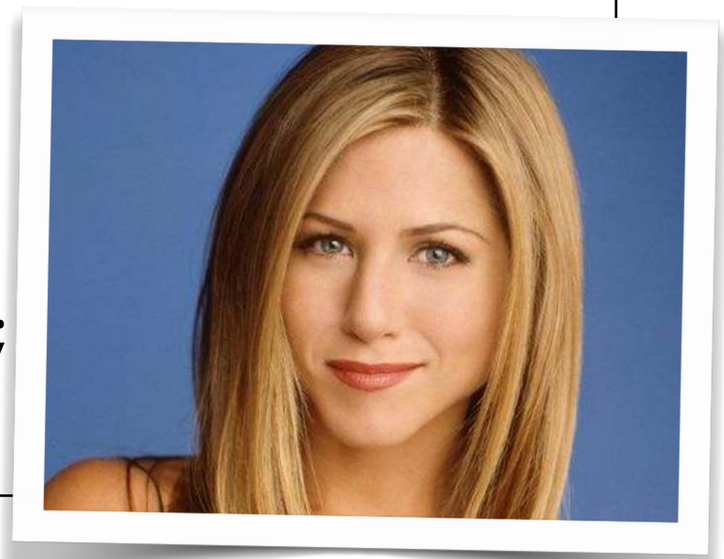
- Quickly drop whole sets of data
- Better performance for queries on recent data
- Scaleability™
- Truly Impress your DBA

Table partitioning (RANGE)

By Joey

Practical PostgreSQL: Incremental update order

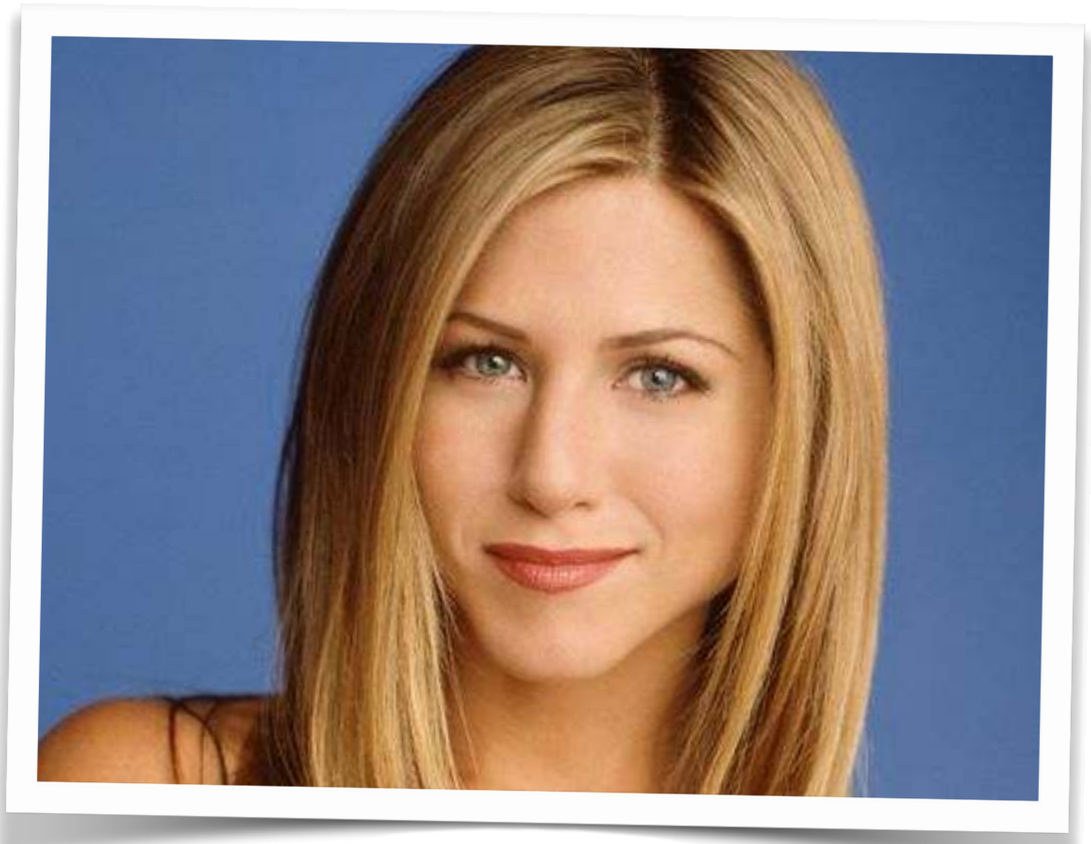
```
CREATE SEQUENCE documents_order;

CREATE FUNCTION set_last_updated_at_order()
RETURNS trigger AS $$
BEGIN
  NEW.last_updated_at := current_timestamp;
  NEW.order := nextval('documents_order');
  RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER set_last_updated_at_order
BEFORE INSERT OR UPDATE ON documents
FOR EACH ROW
EXECUTE PROCEDURE set_last_updated_at_order();
```



Other interesting systems

- Object storage
  PB-scale data store and data lake
- CouchDB
  distributed document database
- BigQuery
  serverless PB-scale data lakehouse/warehouse

by Phoebe



"The versatility of Postgres is unbeatable."

Rachel

Cheers!

L·U·K·A·S

Producer

L·U·K·A·S

Contact me

lukas@parlant.co

Proficient in DevOps, Backend and Data projects with 16+ years experience.

Currently looking for Cloud and Data engineers.

Freelance.

jsonb_eql

https://stackoverflow.com/a/58802265/128351

with Joey

entity component database modeling
https://gist.github.com/Overbryd/24e59e004aa4f66e106217c2227d7ea9

with Joey

Fail safe views in PostgreSQL
https://gist.github.com/Overbryd/d99cc48eb2b230da8f4af87375160469

with Joey

JSON tables

with Joey

Relational, rollup into embeds_many

with Joey

Free managed PostgreSQL (1CPU / 1GB)
https://gist.github.com/Overbryd/b7bf824a2d53fda0225635386fa6acf1

with Joey