Data Modeling, Normalization and Denormalization

Nordic PgDay 2018, Oslo

Dimitri Fontaine

CitusData

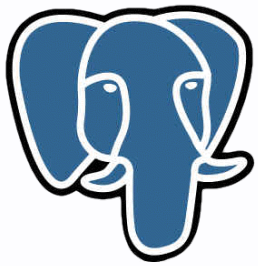
March 13, 2018

Dimitri Fontaine

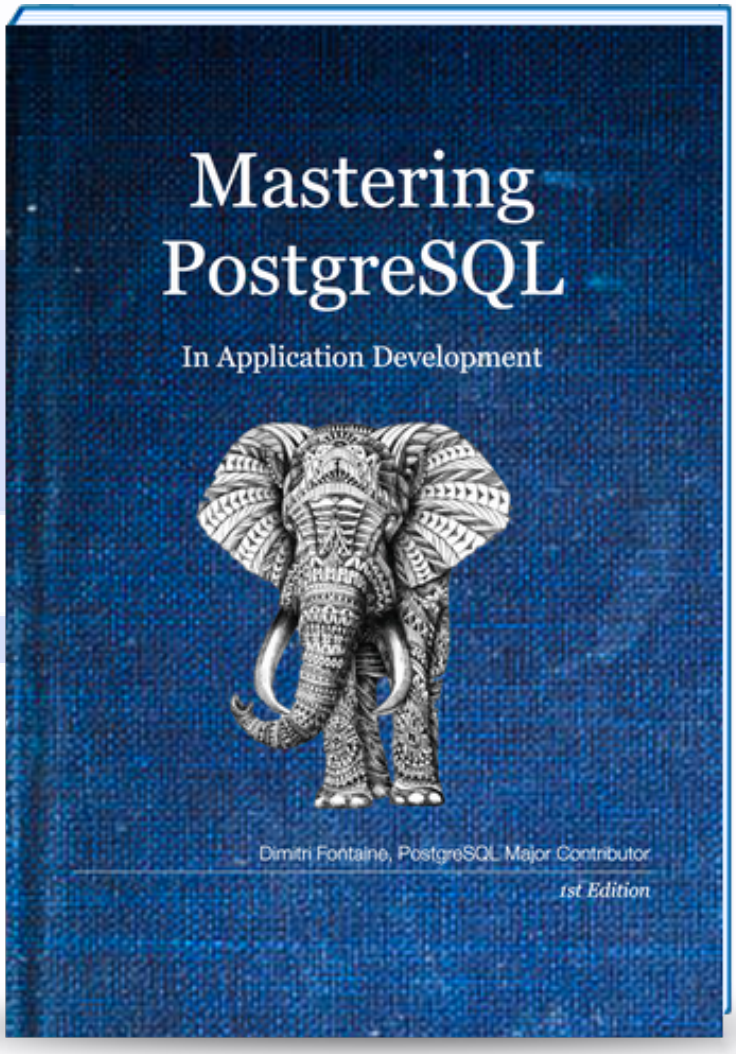
PostgreSQL Major Contributor

- **pgloader**
- `CREATE EXTENSION`
- `CREATE EVENT TRIGGER`
- *Bi-Directional Réplication*
- *apt.postgresql.org*

PostgreSQL



Mastering PostgreSQL in Application Development



I wrote a book!

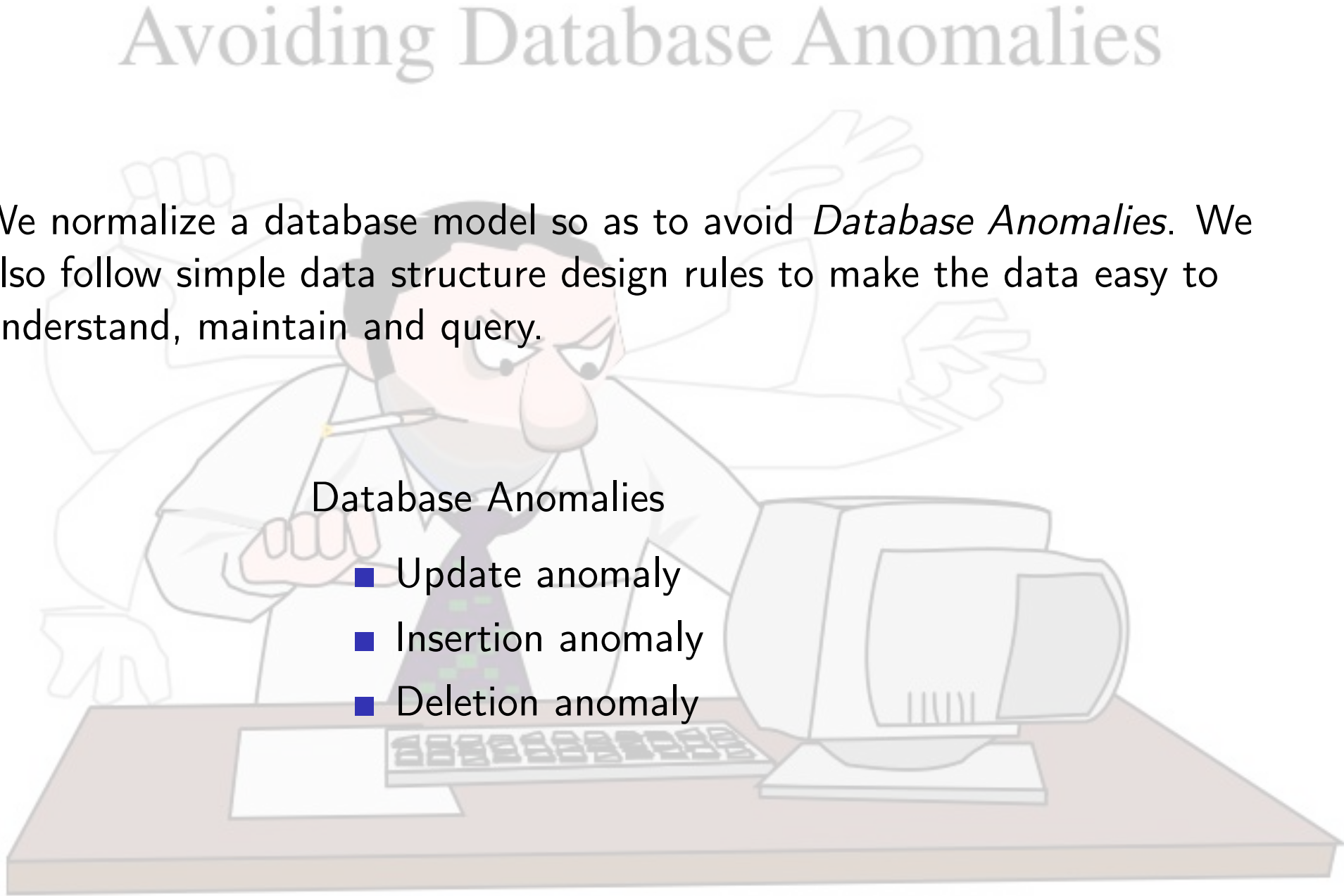
Mastering PostgreSQL in Application Development teaches SQL to developers: learn to replace thousands of lines of code with simple queries.

`http://MasteringPostgreSQL.com`

Rule 5. ***Data dominates.***

If you've chosen the right data structures and organized things well, the algorithms will almost always be self-evident. Data structures, not algorithms, are central to programming. (Brooks p. 102.)

Database Anomalies



We normalize a database model so as to avoid *Database Anomalies*. We also follow simple data structure design rules to make the data easy to understand, maintain and query.

Database Anomalies

- Update anomaly
- Insertion anomaly
- Deletion anomaly

Update Anomaly

Employees' Skills

Employee ID	Employee Address	Skill
426	87 Sycamore Grove	Typing
426	87 Sycamore Grove	Shorthand
519	94 Chestnut Street	Public Speaking
519	96 Walnut Avenue	Carpentry

Insertion Anomaly

Faculty and Their Courses

Faculty ID	Faculty Name	Faculty Hire Date	Course Code
389	Dr. Giddens	10-Feb-1985	ENG-206
407	Dr. Saperstein	19-Apr-1999	CMP-101
407	Dr. Saperstein	19-Apr-1999	CMP-201
424	Dr. Newsome	29-Mar-2007	?

Deletion Anomaly

Faculty and Their Courses

Faculty ID	Faculty Name	Faculty Hire Date	Course Code
389	Dr. Giddens	10-Feb-1985	ENG-206
407	Dr. Saperstein	19-Apr-1999	CMP-101
407	Dr. Saperstein	19-Apr-1999	CMP-201

DELETE

Database Design and User Workflow

Show me your flowcharts and conceal your tables, and I shall continue to be mystified. Show me your tables, and I won't usually need your flowcharts; they'll be obvious.
(Fred Brooks)

Database Modeling & Tooling

Tooling for Database Modeling

We can use `psql` and SQL scripts to edit database schemas:

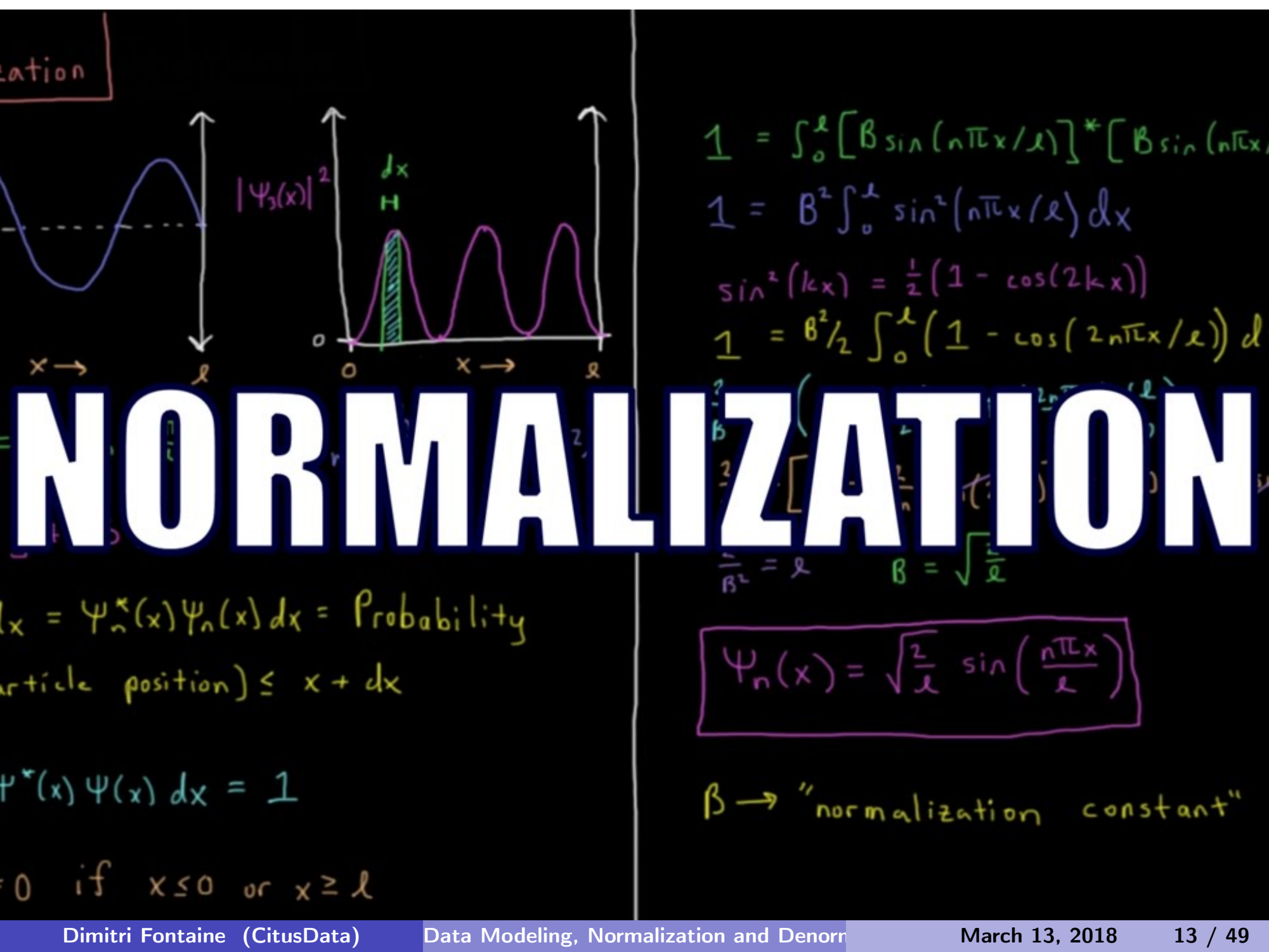
```
BEGIN;

create schema if not exists sandbox;

create table sandbox.category
 (
   id    serial primary key,
   name  text not null
 );

insert into sandbox.category(name)
     values ('sport'),('news'),('box office'),('music');

ROLLBACK;
```

Object Relational Mapping

The R in **ORM** stands for *"Relation"*. The result of a SQL query is a relation. That's what you should be mapping, not your base tables!

When mapping base tables, you end up trying to solve different complex issues at the same time:

- User Workflow
- Consistent view of the whole world at all time



NORMALIZATION

Basics of the Unix Philosophy: principles

Some design rules that apply to Unix and to database design too:

- Rule of Clarity
  Clarity is better than cleverness.
- Rule of Simplicity
  Design for simplicity; add complexity only where you must.
- Rule of Transparency
  Design for visibility to make inspection and debugging easier.
- Rule of Robustness
  Robustness is the child of transparency and simplicity.

Normal Forms

The Normal Forms are designed to avoid database anomalies, and they help in following the listed rules seen before.

1st Normal Form, Codd, 1970

1. There are no duplicated rows in the table.
2. Each cell is single-valued (no repeating groups or arrays).
3. Entries in a column (field) are of the same kind.

Second Normal Form

2nd Normal Form, Codd, 1971

A table is in *2NF* if it is in *1NF* and if all non-key attributes are dependent on all of the key. Since a partial dependency occurs when a non-key attribute is dependent on only a part of the composite key, the definition of *2NF* is sometimes phrased as:

"A table is in *2NF* if it is in *1NF* and if it has no partial dependencies."

Third Normal Form and Boyce-Codd Normal Form

3rd Normal Form (Codd, 1971)
and BCNF (Boyce and Codd, 1974)

3NF A table is in 3NF if it is in 2NF and if it has no transitive dependencies.

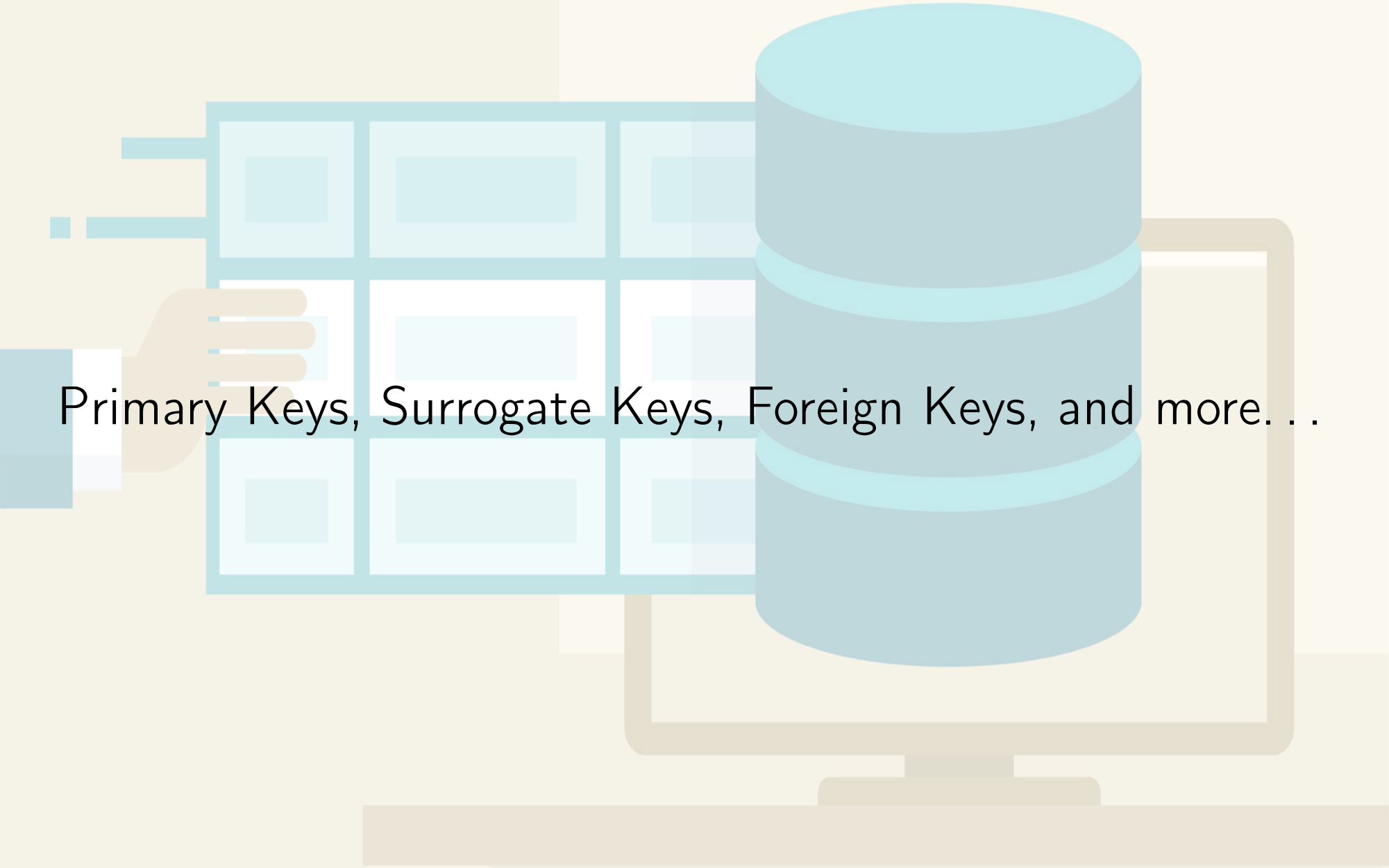
BCNF A table is in BCNF if it is in 3NF and if every determinant is a candidate key.

More Normal Forms!

Each level builds on the previous one.

- *4NF* A table is in 4NF if it is in BCNF and if it has no multi-valued dependencies.
- *5NF* A table is in 5NF, also called "Projection-join Normal Form" (PJNF), if it is in 4NF and if every join dependency in the table is a consequence of the candidate keys of the table.
- *DKNF* A table is in DKNF if every constraint on the table is a logical consequence of the definition of keys and domains.

Database Constraints



Primary Keys, Surrogate Keys, Foreign Keys, and more. . .

Primary Keys

First Normal Form requires no duplicated row. I know, let's use a Primary Key!

```
create table sandbox.article
 (
   id          bigserial primary key,
   category    integer references sandbox.category(id),
   pubdate     timestamptz,
   title       text not null,
   content     text
 );
```

Primary Keys, Surrogate Keys

Artificially generated key is named a *surrogate key* because it is a substitute for *natural key*. A *natural key* would allow preventing duplicate entries in our data set.

```
insert into sandbox.article (category, pubdate, title)
     values (2, now(), 'Hot from the Press'),
            (2, now(), 'Hot from the Press')
  returning *;
```

Primary Keys, Surrogate Keys

Oops.

```
-[ RECORD 1 ]--------------------------
id       | 3
category | 2
pubdate  | 2018-03-12 15:15:02.384105+01
title    | Hot from the Press
content  |
-[ RECORD 2 ]--------------------------
id       | 4
category | 2
pubdate  | 2018-03-12 15:15:02.384105+01
title    | Hot from the Press
content  |

INSERT 0 2
```

Primary Keys, Surrogate Keys

Fixing the model is easy enough: implement a *natural primary key*.

```
create table sandboxpk.article
 (
   category    integer references sandbox.category(id),
   pubdate     timestamptz,
   title       text not null,
   content     text,

   primary key(category, pubdate, title)
 );
```

Primary Keys, Foreign Keys

Now we have to reference the whole natural key everywhere:

```
create table sandboxpk.comment
 (
   a_category integer     not null,
   a_pubdate  timestamptz not null,
   a_title    text        not null,
   pubdate    timestamptz,
   content    text,

   primary key(a_category, a_pubdate, a_title, pubdate, content),

   foreign key(a_category, a_pubdate, a_title)
    references sandboxpk.article(category, pubdate, title)
 );
```

Primary Keys, Foreign Keys

One solution is to have both a surrogate and a natural key:

```
create table sandbox.article
 (
   id          integer      generated always as identity,
   category    integer      not null references sandbox.category(id),
   pubdate     timestamptz  not null,
   title       text         not null,
   content     text,

   primary key(category, pubdate, title),
   unique(id)
 );
```

Normalization Helpers: database constraints

To help you implement Normal Forms with strong guarantees even when having to deal with concurrent access to the database, we have *constraints*.

- Primary Keys
- Foreign Keys
- Not Null
- Check Constraints
- Domains
- Exclusion Constraints

```
1  create table rates
2   (
3    currency text,
4    validity daterange,
5    rate     numeric,
6
7    exclude using gist
8    (
9       currency with =,
10      validity with &&
11   )
12  );
```

Denormalization

The first rule
of denormalization is that you
don't
do denormalization.

Denormalization is an optimization technique

> *Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97% of the time:* ***premature optimization is the root of all evil.*** *Yet we should not pass up our opportunities in that critical 3%.*

Donald Knuth, *"Structured Programming with Goto Statements". Computing Surveys 6:4 (December 1974), pp. 261–301, §1.*

Denormalization: cache data

The main trick: repeat data to make it locally available, breaking functional dependency rules. You know have a cache.

Implement Cache Invalidation.

Denormalization example

```
\set season 2017

  select drivers.surname as driver,
         constructors.name as constructor,
         sum(points) as points

    from results
         join races using(raceid)
         join drivers using(driverid)
         join constructors using(constructorid)

   where races.year = :season

group by grouping sets(drivers.surname, constructors.name)
  having sum(points) > 150
order by drivers.surname is not null, points desc;
```

Denormalization example

```
create view v.season_points as
  select year as season, driver, constructor, points
    from seasons left join lateral
         (
            select drivers.surname as driver,
                   constructors.name as constructor,
                   sum(points) as points
              from results
                   join races using(raceid)
                   join drivers using(driverid)
                   join constructors using(constructorid)
             where races.year = seasons.year
          group by grouping sets(drivers.surname, constructors.name
          order by drivers.surname is not null, points desc
          )
          as points on true
order by year, driver is null, points desc;
```

Denormalization example

And now cache the results of the view into a durable relation:

```
create materialized view cache.season_points as
  select * from v.season_points;

create index on cache.season_points(season);
```

When you need to invalidate the cache, just refresh the view:

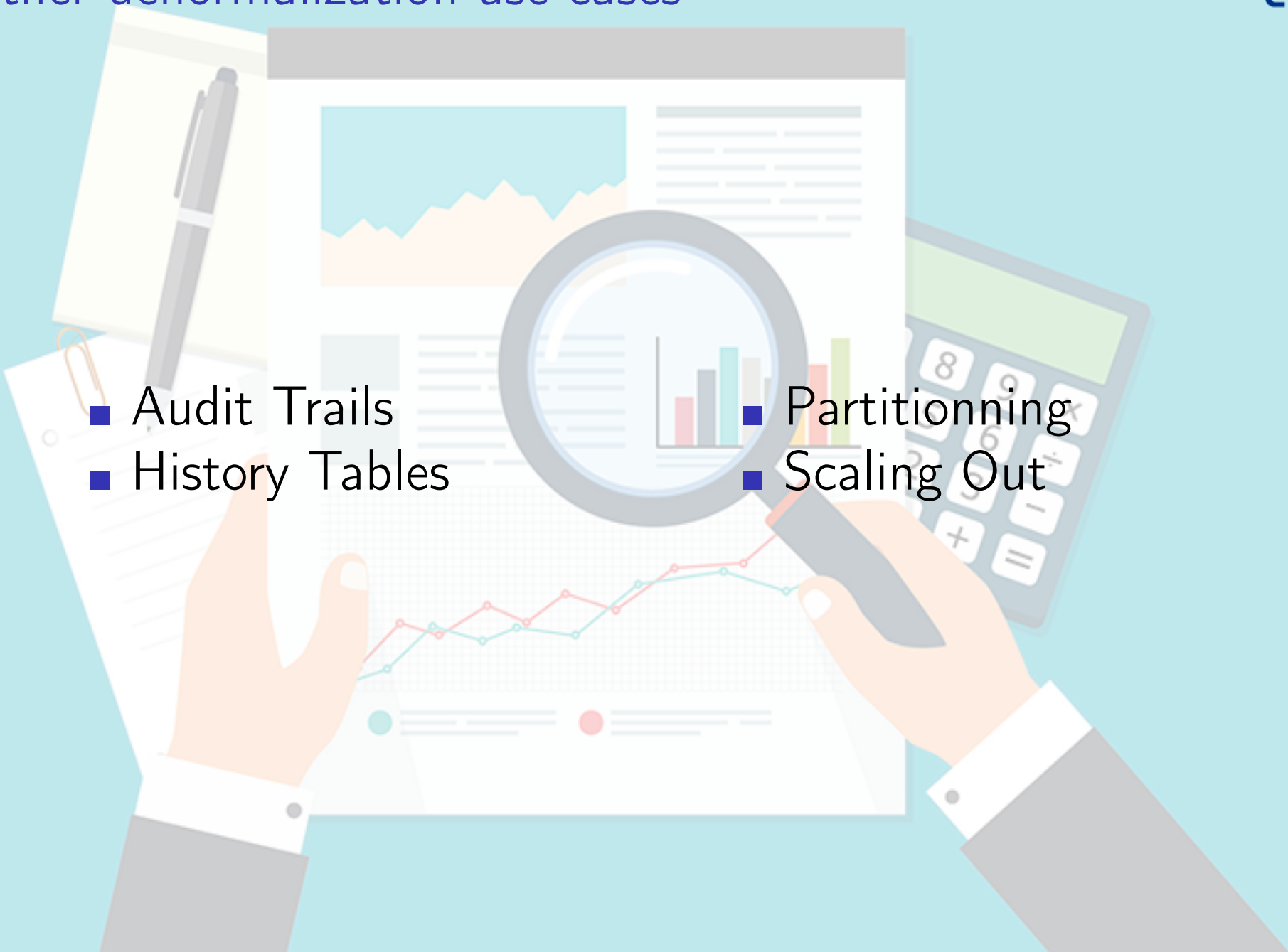
```
  refresh materialized view cache.season_points;
```

Denormalization example

And now rewrite your application's query as:


```
select driver, constructor, points
  from cache.season_points
 where season = 2017
   and points > 150;
```

Other denormalization use cases



- Audit Trails
- History Tables
- Partitionning
- Scaling Out

History tables and audit trails

Another case where you might have to denormalize your database model is when keeping a history of all changes.

- Foreign key references to other tables won't be possible when those reference changes and you want to keep a history that, by definition, doesn't change.
- The schema of your main table evolves and the history table shouldn't rewrite the history for rows already written.

History tables with JSONB

JSONB is very flexible, and can host the archives for all your database model versions in the same table, or for all your source tables at once even.

```
create schema if not exists archive;

create type archive.action_t
    as enum('insert', 'update', 'delete');

create table archive.older_versions
 (
   table_name text,
   date       timestamptz default now(),
   action     archive.action_t,
   data       jsonb
 );
```

Validity periods

A variant of the historic requirement is to keep track of data changes and be able to use the value that were valid at a known time. Currency exchange rates applied to invoices is an example:

```
create table rates
 (
  currency text,
  validity daterange,
  rate     numeric,

  exclude using gist (currency with =,
                      validity with &&)
 );
```

Validity periods

Here's how to use the data from a known time in the past:

```
  select currency, validity, rate
    from rates
   where currency = 'Euro'
   and validity @> date '2017-05-18';

-[ RECORD 1 ]----------------------
currency | Euro
validity | [2017-05-18,2017-05-19)
rate     | 1.240740
```

Denormalization Helpers: advanced datatypes

Composite datatypes help with denormalization. It's possible to keep several values in the same column thanks to them. Spare matrix becomes an *extra* field of `jsonb` type.

- Composite Types
- Arrays
- JSONb
- Enumerated Types

- `hstore`
- `ltree`
- `intarray`
- `pg_trgm`

Partitioning

Partitioning comes with demormalization trade-offs in PostgreSQL 10:

- Index are managed at the partition level
- No Primary Key, No Unique Index, No Exclusion Constraint
- No Foreign Key pointing to a partitionned table
- Lack of `ON CONFLICT` support
- Lack of `UPDATE` support for *re-balancing*

Not Only SQL

Schemaless design

PostgreSQL includes several composite types (multi-value data). JSONb allows the implementation of schemaless design right within PostgreSQL.

```
select jsonb_pretty(data)
  from magic.cards
 where data @> '{"type":"Enchantment",
                 "artist":"Jim Murray",
                 "colors":["White"]
                }';
```

NoSQL and Durability Trade-Offs

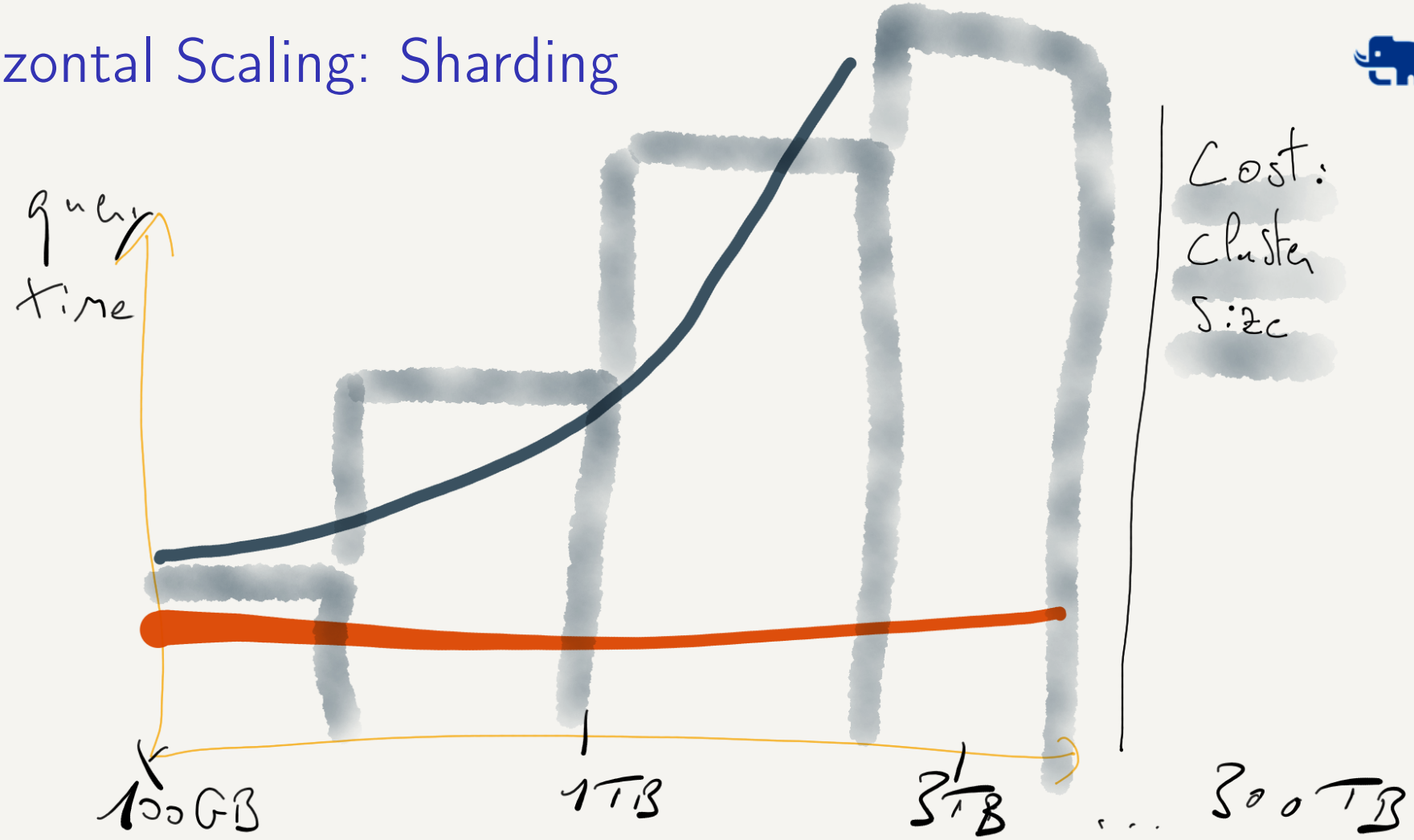
PostgreSQL setup is made with GUC, or *Great Unified Configuration*. You can edit values in the `postgresql.conf` file, or dynamically change it in the session. Or in the transaction with `SET LOCAL`. Or have per-user or per-database settings.

```
create role dbowner with login;
create role app with login;

create role critical  with login in role app inherit;
create role notsomuch with login in role app inherit;
create role dontcare  with login in role app inherit;

alter user critical  set synchronous_commit to remote_apply;
alter user notsomuch set synchronous_commit to local;
alter user dontcare  set synchronous_commit to off;
```

Automatic Per-Transaction Durability Setting

```
SET demo.threshold TO 1000;

CREATE OR REPLACE FUNCTION public.syncrep_important_delta()
  RETURNS TRIGGER
  LANGUAGE PLpgSQL
AS
$$ DECLARE
  threshold integer := current_setting('demo.threshold')::int;
  delta integer := NEW.abalance - OLD.abalance;
BEGIN
  IF delta > threshold
  THEN
    SET LOCAL synchronous_commit TO on;
  END IF;
  RETURN NEW;
END;
$$;
```

Horizontal Scaling: Sharding



Five sharding data models and which is right?

If you were here this morning you've seen Craig's talk, so that's about it.

- Sharding by geography
- Sharding by entity id
- Sharding a graph
- Time partitioning
- Depends. . .

Denormalization and Sharding

Adding the sharding key to every table is another case of *duplicating* information for maintaining a ***cache.*** Beware of *database anomalies*

Questions?

Now is the time to ask!

https://2018.nordicpgday.org/feedback