BRAVIANT

Connecting Galaxies: Information Exchange Techniques for Heterogeneous Environments

Hettie Dombrovskaya &
Alyssa Ritchie, Braviant Holdings
Boris Novikov, HSE

MARCH 2019

WHY GALAXIES

WHY CARE ABOUT DISTANT GALAXIES?

✓

We are very good at optimizing individual queries

✗

We aren’t very good at optimizing communications with this galaxy

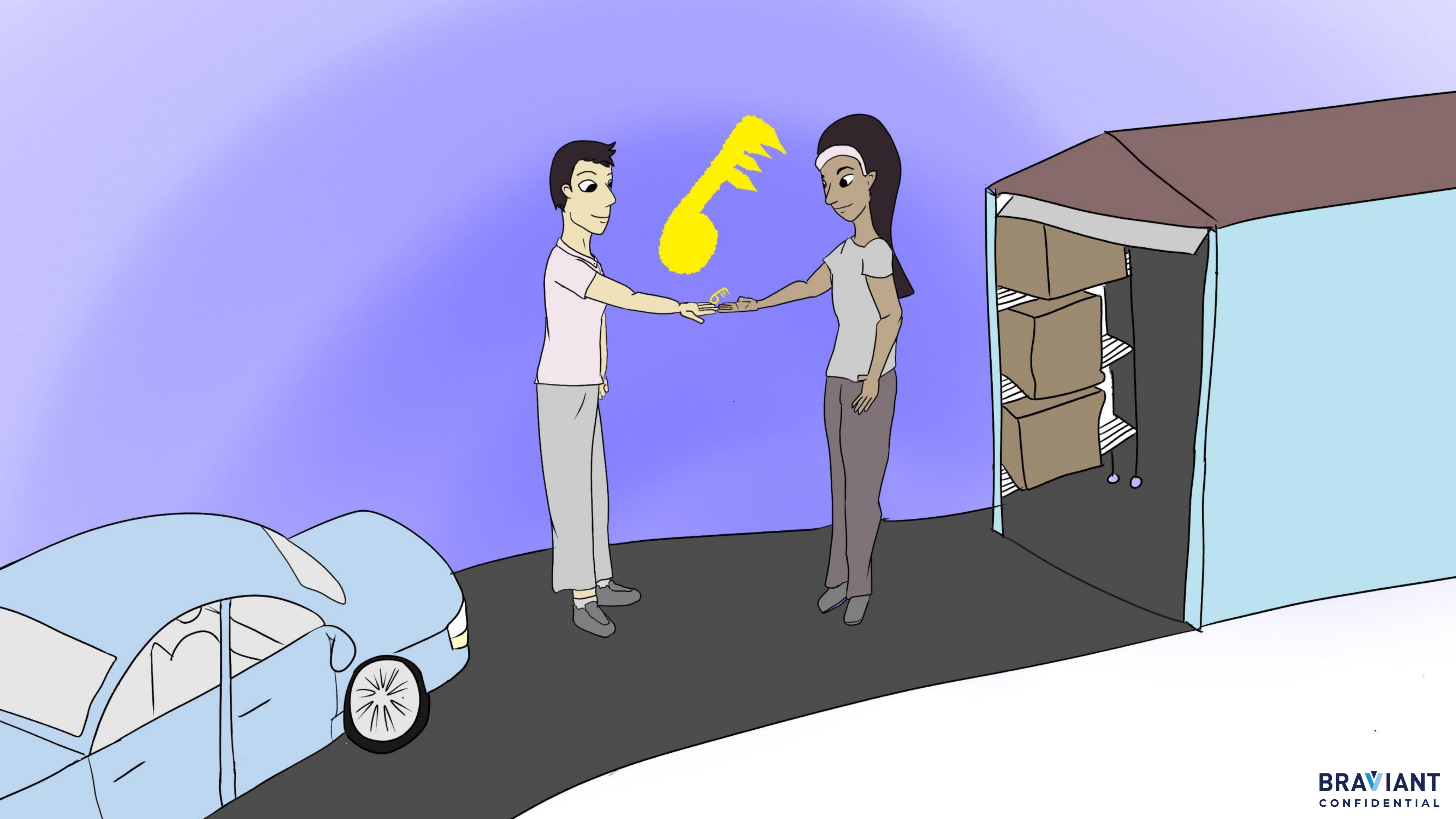
?

Most of the real-world queries come from this galaxy, which knows **nothing** about a database!

THE OBJECT-ORIENTED DREAM

Once upon a time (long ago, in a previous millennium), object-oriented application design and development was born…

However, some parts of the world remain unknown…



WHAT HAPPENED NEXT?

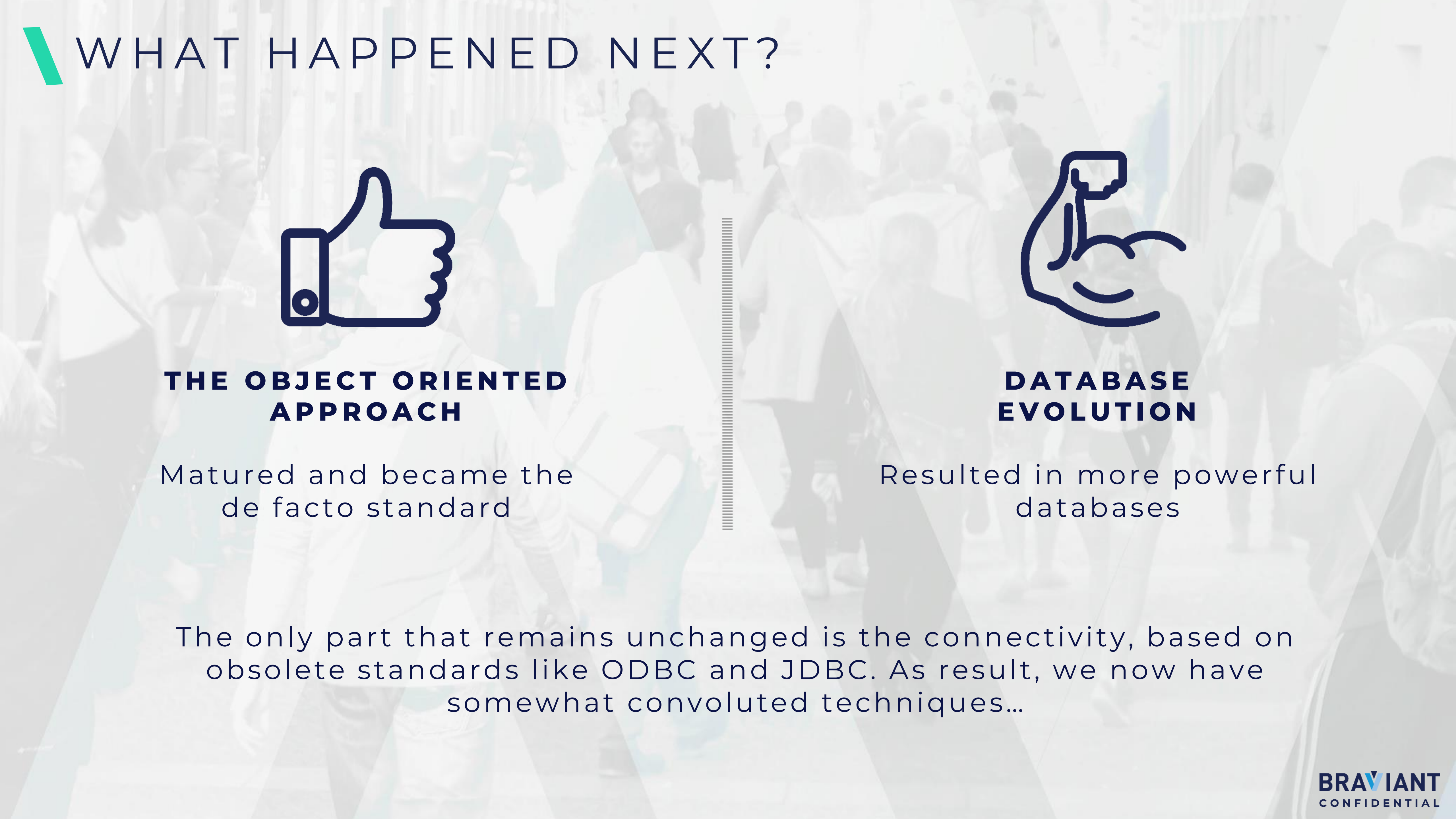
BRAVE PEOPLE tried to introduce object-oriented databases…

Progress
ObjectStore

Technology

Objectivity

WHAT HAPPENED NEXT?



THE OBJECT ORIENTED APPROACH

Matured and became the de facto standard

DATABASE EVOLUTION

Resulted in more powerful databases

The only part that remains unchanged is the connectivity, based on obsolete standards like ODBC and JDBC. As result, we now have somewhat convoluted techniques…

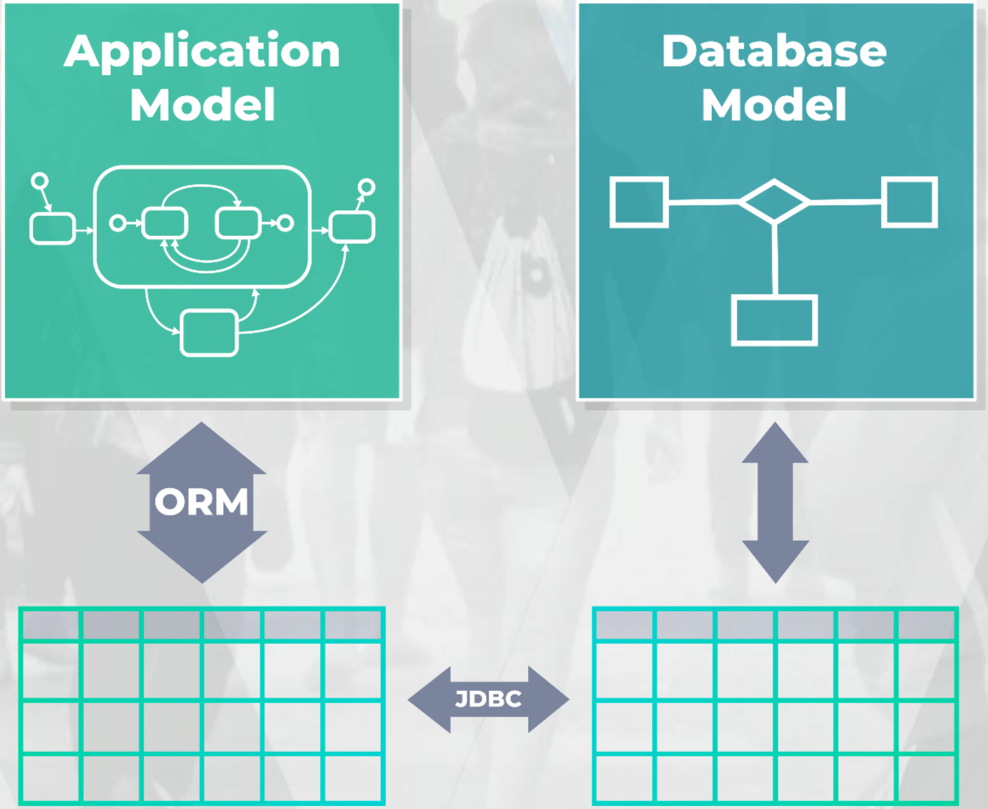
HOW DOES ORM WORK?

1. The application disassembles an object into undividable (scalar) parts

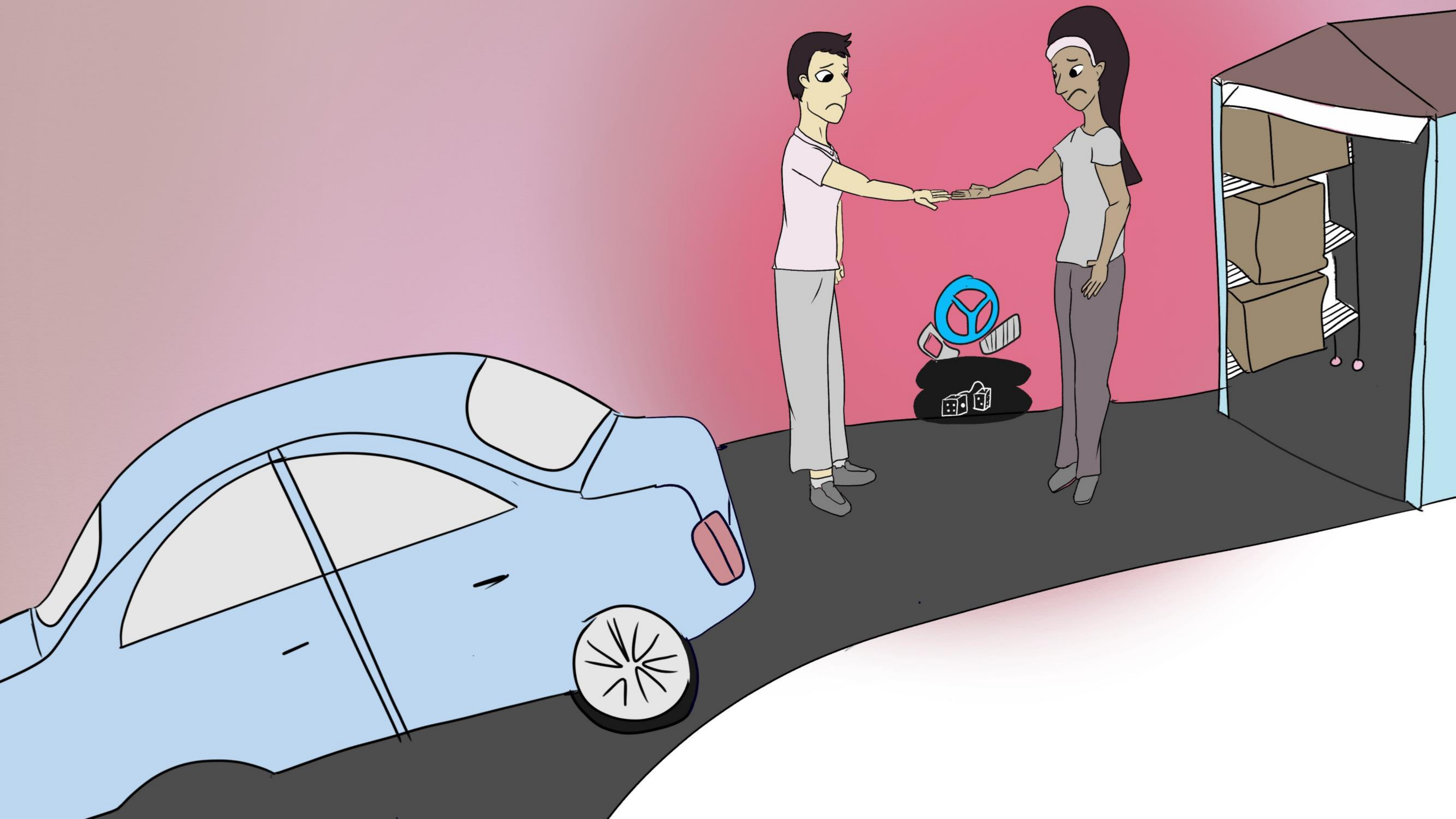
2. The parts are sent to/from the database separately

3. At the database site, the complex data structure is re-assembled

A lot of transfers are needed to send a complex object



BRAVIANT
CONFIDENTIAL



AND IS ANYONE HAPPY?

NO

WHY IS NOBODY HAPPY?

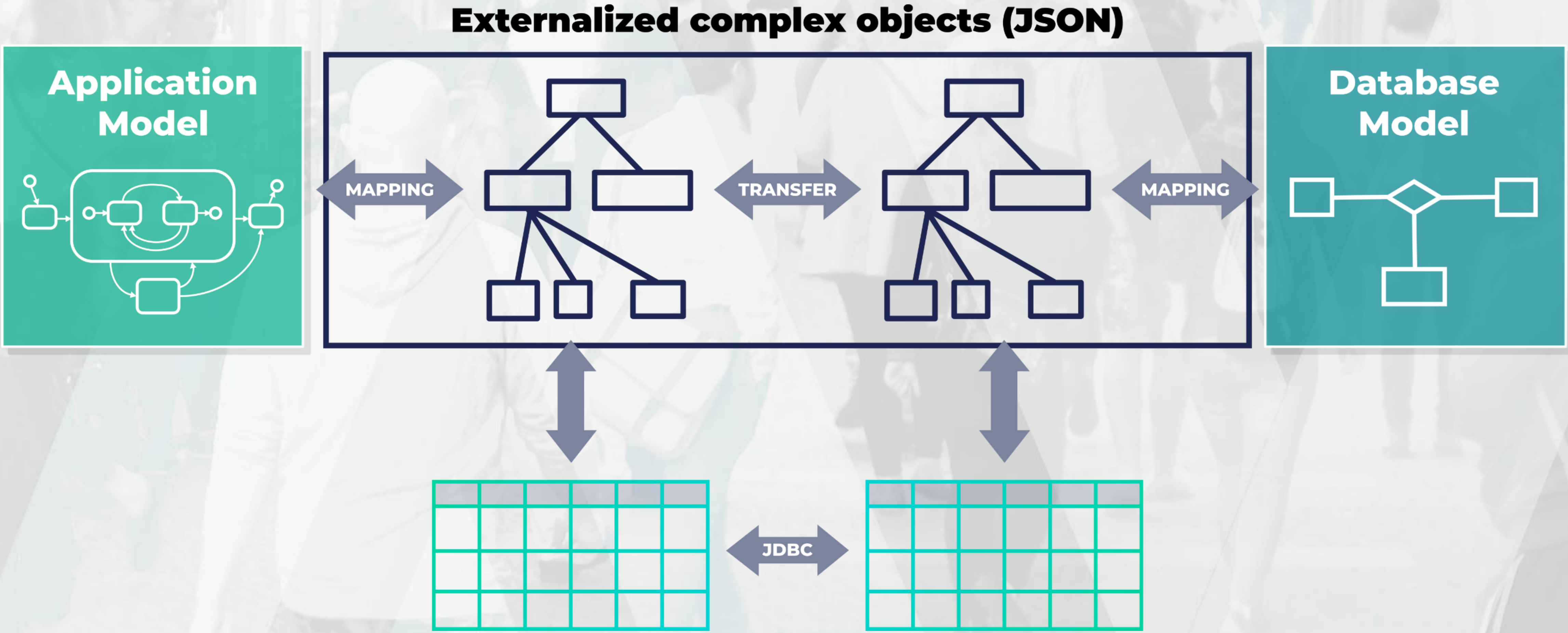
APPLICATION

- Complexity of building compound objects
- Embedding database specifics into the application
- Multiple database calls, which slow performance

DATABASE

- The power of query languages isn’t used
- Too many small database calls bottleneck communication

CAN WE FIX IT? ABSOLUTELY!



BRAVIANT
CONFIDENTIAL

OUR SOLUTION:

Mapping both DB objects (D-objects) and application objects (A-objects) to the transfer objects (T-objects)

ON THE APP SIDE, this object is handled with standard serialization/deserialization tools.

ON THE DATABASE SIDE, the transferred JSON object is mapped to the database schema with declarative SQL statements, exploiting the power of query processing features.

JDBC is still needed, but it is only used as a transport layer for the JSON objects.

IMPLEMENTATION
(HOW DID WE GET THERE)

THE BEST OPTION FOR A SINGLE QUERY? A FUNCTION!

First, we tried a well-known approach: implementing functions that return record sets:

```
create type user_account_record as (
user_account_id  bigint,
username text ,
brand  text,
lms_customer_id  int);

select * from select_user_account(1) returns a record
```

AND MANY MORE…

```
select * from select_user_account_by_username(‘username1’)
```

Returns the same record type

```
select * from select_user_account_by_phone(‘888888888’)
```

```
select * from select_user_account_by_last_name(‘last_name’)
```

All of them return the same record type, and we hide the query details!

COMPLEX OBJECTS ARE NESTED…

…and we started creating record types with embedded records:

```
create type user_account_record as (
user_account_id  bigint,
username text ,
brand  text,
lms_customer_id  int,
email_address email_address_record[] ,
addresses address_record[],
phones phone_record[],
bank_information_id bank_information_record[] );
```

THE PROBLEM IN THIS APPROACH

PostgreSQL does not preserve the type of the embedded record, so the output of

```
select * from user_account_get (1)
```

Will look like this:

```
1,
’username@email.com’,
’chorus’,
{‘city’,’street’,’IL’, ‘60606’},
{1, ‘primary’, ‘4445556666’}
```

CAN WE USE JSON FOR NESTED OBJECTS?

We sure can!

```
select user_account_id,
username ,
brand,
json_build_object (’address_1’, addr_line_1,
       ‘city’,  city,
       ‘state’, state_code,
       ‘zip’, zip) as address
from user_account u
join address a
on a.user_account_id=u.user_account_id
```

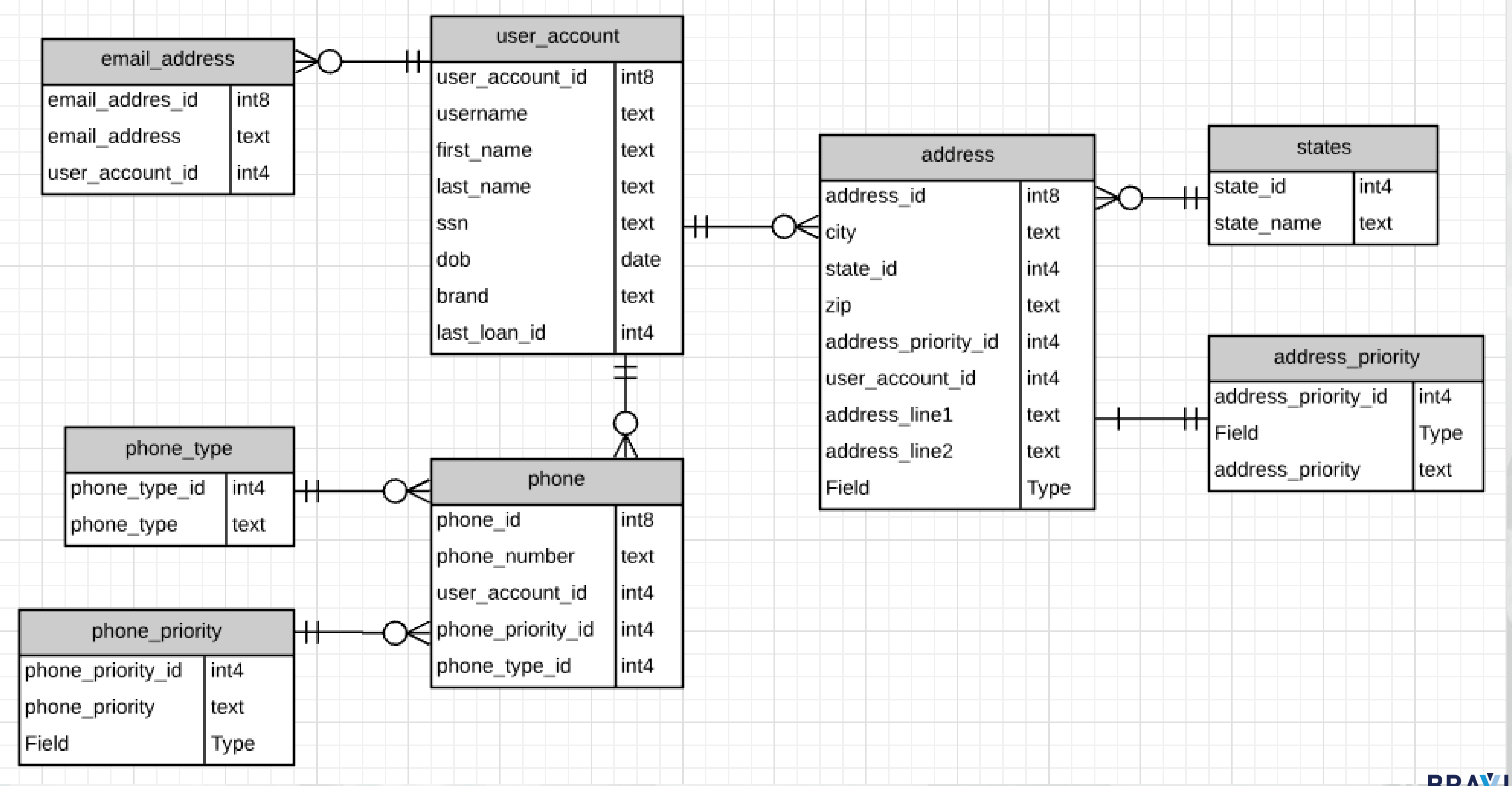
HOW FAR CAN WE GO?

NOW OUR FUNCTIONS RETURN JSON OBJECTS

```
{
  "dob": "1971-01-10",
  "ssn": "111223333",
  "username": "john.smith@email.com",
  "last_name": "John",
  "first_name": "Smith",
  "phones": [
    { "phone_number": "1112223333",  "phone_priority_id": 1,
"phone_priority": "primary", "phone_type_id": 1 },
    { "phone_number": "4445556666", "phone_priority_id": 2,
"phone_priority": "secondary", "phone_type_id": 1 }
  ]
}
```

In other words, we mapped D-objects to T-objects

DB SCHEMA (D-OBJECTS)



email_address	
email_addres_id	int8
email_address	text
user_account_id	int4

user_account	
user_account_id	int8
username	text
first_name	text
last_name	text
ssn	text
dob	date
brand	text
last_loan_id	int4

address	
address_id	int8
city	text
state_id	int4
zip	text
address_priority_id	int4
user_account_id	int4
address_line1	text
address_line2	text
Field	Type

states	
state_id	int4
state_name	text

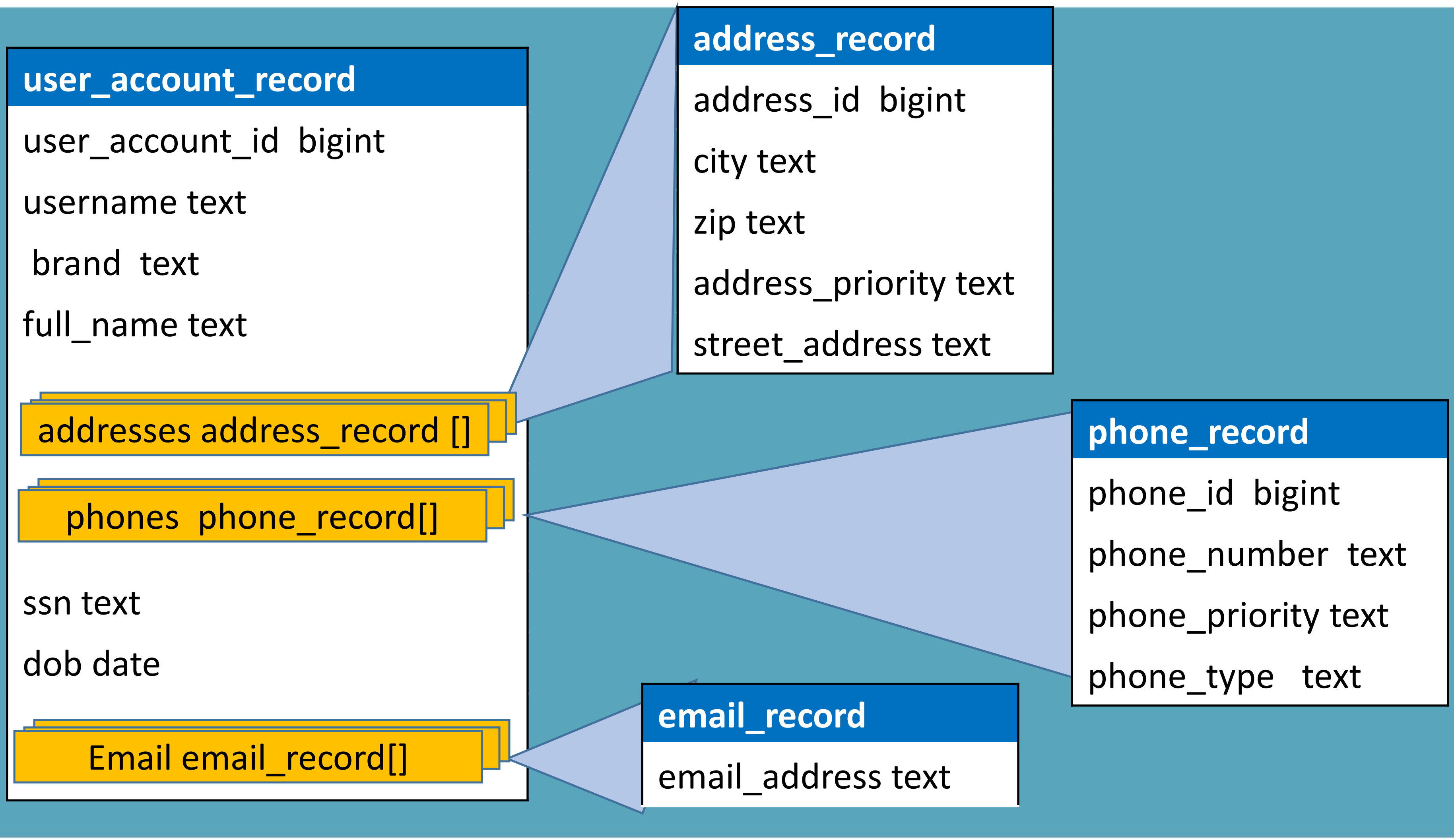
address_priority	
address_priority_id	int4
Field	Type
address_priority	text

phone_type	
phone_type_id	int4
phone_type	text

phone	
phone_id	int8
phone_number	text
user_account_id	int4
phone_priority_id	int4
phone_type_id	int4

phone_priority	
phone_priority_id	int4
phone_priority	text
Field	Type

T-OBJECTS



user_account_record

user_account_id bigint

username text

brand text

full_name text

addresses address_record []

phones phone_record[]

ssn text

dob date

Email email_record[]

address_record

address_id bigint

city text

zip text

address_priority text

street_address text

phone_record

phone_id bigint

phone_number text

phone_priority text

phone_type text

email_record

email_address text

NEW PROBLEMS

When we return JSON from a function, we are loosing strong types

Building JSON with embedded SELECTs can be slow

INITIAL SOLUTION

SPECIAL “STRUCTURE-DEFINING” FUNCTIONS:

```
create or replace function user_account_json() returns text
language sql immutable as $body$select $$'user_id,'dob’,'ssn’,
username,last_name,first_name,phones$$::text; $body$;
```

```
create or replace function phone_json() returns text language sql
immutable as $body$select $$phone_number,phone_priority_id,
phone_priority,phone_type_id$$::text; $body$;
```

We started to **use these functions in json_build_object**

EMBEDDING SQL: NO GOOD SOLUTION

```
SELECT
v_dminfo_json[1],match_status_1,
v_dminfo_json[2],match_status_2,
v_dminfo_json[3],
      array_to_json(array(select preapproval_id
       FROM origination.application_preapproval
             WHERE application_id =a.application_id  and match_type=1)),
v_dminfo_json[4],
       array_to_json(array(select preapproval_id
        FROM origination.application_preapproval
            WHERE application_id =a.application_id  and match_type=2 )))
  ))::text
       FROM origination.application
             WHERE …
```

NEW SOLUTION: JSON AGG FUNCTIONS

MAKING JSON BUILD AN AGGREGATE!

```
create or replace function common.json_agg_next  (agg_sta text, val  json)    returns text  as
$$ begin
 if  val is not null then
 if  agg_sta = '' then     agg_sta :=  val::text ;
        else  agg_sta :=  agg_sta ||  ',' || (val::text) ;
        end if;
        end if;
 return agg_sta;
END;$$ LANGUAGE plpgsql;

create or replace function common.json_agg_final (agg_sta  text)    returns json   as
$$ begin
 if  agg_sta = '' then return null;
  else  return ('[' || agg_sta || ']')::json;
 end if;
END;
$$ LANGUAGE plpgsql;

drop AGGREGATE if exists common.json_agg (json);
create  AGGREGATE common.json_agg (json) (    sfunc=common.json_agg_next,
 STYPE = text,    FINALFUNC = common.json_agg_final,    INITCOND = ''    );
```

HOW SELECT LOOKS NOW

```
SELECT array_agg(single_item)
  from (select row(
match_status_1,
match_status_2,
SELECT array_agg(row(
                 preapproval_id ::preapproval_record)
         FROM application_preapproval
                 WHERE application_id =a.application_id
                  AND match_type=1)) AS preapproval_1,
SELECT array_agg(row(
                 preapproval_id ::preapproval_record)
         FROM application_preapproval
                 WHERE application_id =a.application_id
                 AND match_type=2)) AS preapproval_2,
v_dminfo_json[4],
         array_to_json(array(select preapproval_id
          from origination.application_preapproval
            where application_id =a.application_id  and match_type=2 )))
  ))::app_preapproved_record
         FROM origination.application
                 WHERE …
```

LET’S SEE WHAT A BASIC FUNCTION LOOK LIKE

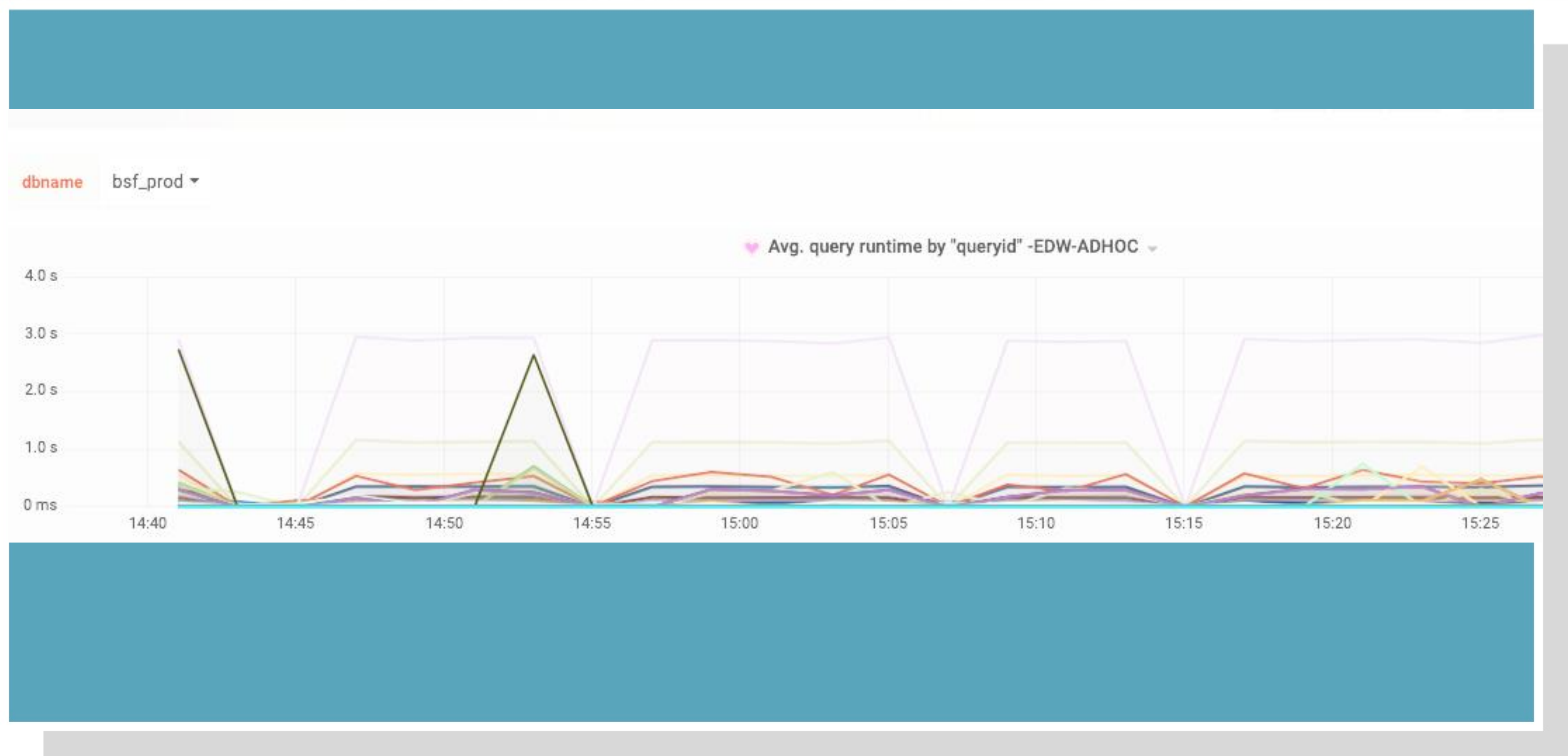
THE CODE

DATA MODIFICATION

```
UPDATE address and DELETE phone:

{
   "user_account_id":1,
   "addresses":[ { "address_id":10, "street_address":"111 MyStreet" } ],
   "phones":[ { "phone_id":22, "command": "delete" } ]
}

UPDATE full name and INSERT email address:

{
   "user account id":1,
   "full_name": "NewFirst NewLast",
   "email_addresses":[ { "email_address": ”username@email.com" } ]
}
```

PERFORMANCE

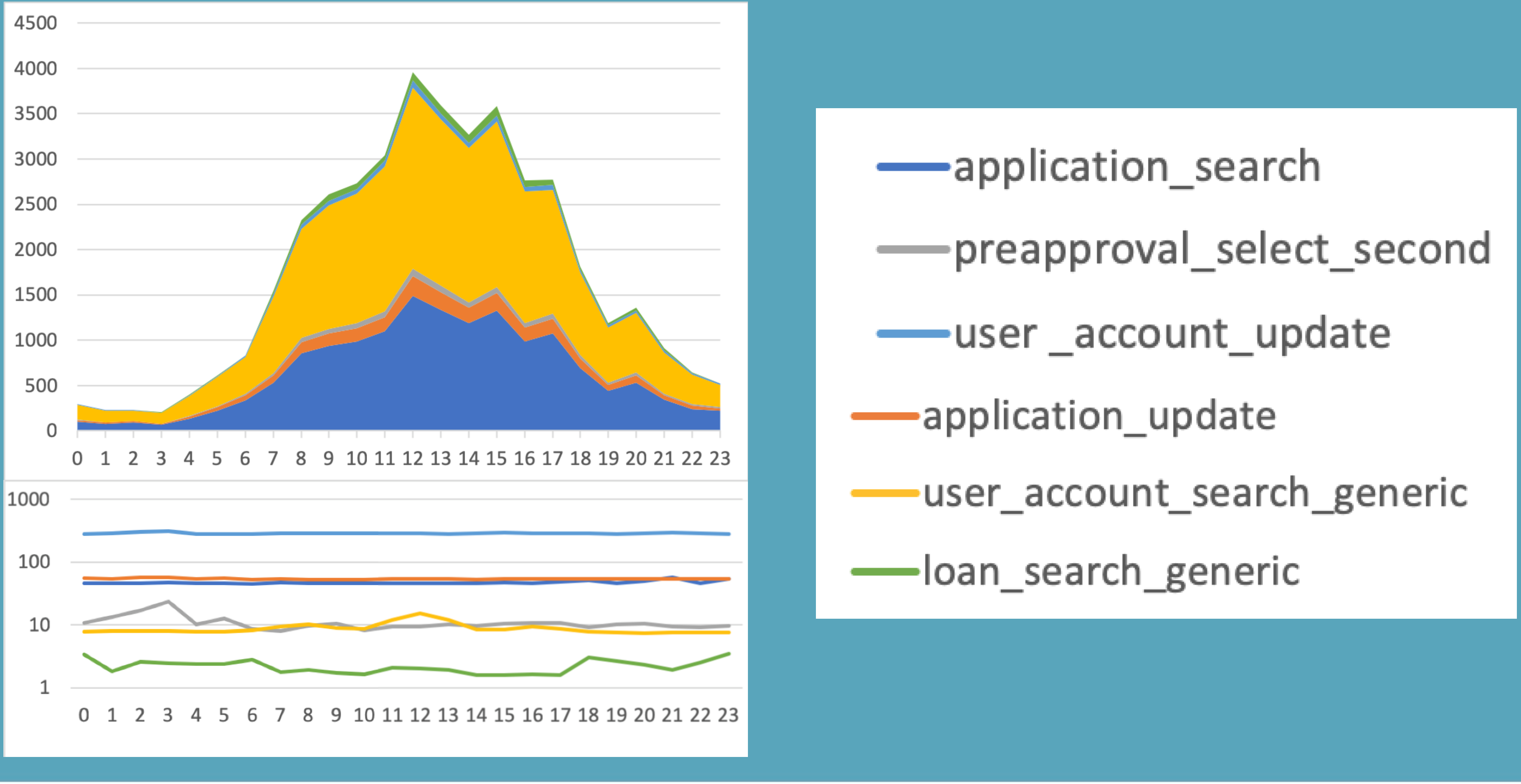
LIVE RUN TIMES BY QUERY



dbname bsf_prod

BRAVIANT CONFIDENTIAL

AVG EXECUTION TIME AND AVG OPS/MIN PER HOUR



WHAT'S LEFT?

ONE PROBLEM REMAINS…

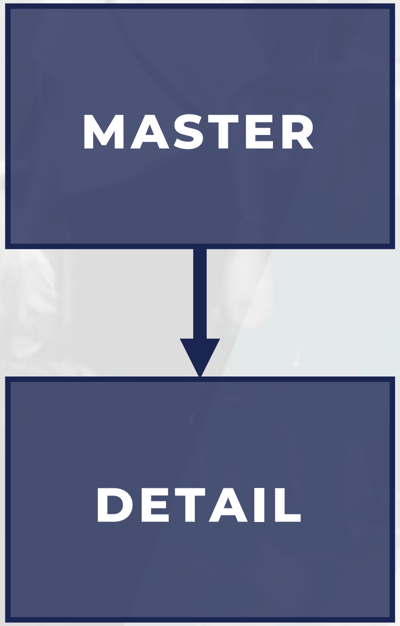
Simple nesting is not always the most efficient.
(That’s why we didn’t use the user account example!)

When the results of the selection are relatively large, the execution of the nested selects may be sub-optimal.

Let’s see how:

BUILDING COMPLEX OBJECTS WITH SIMPLE NESTING

1:M



MASTER

DETAIL

SELECT unnested (denormalized)

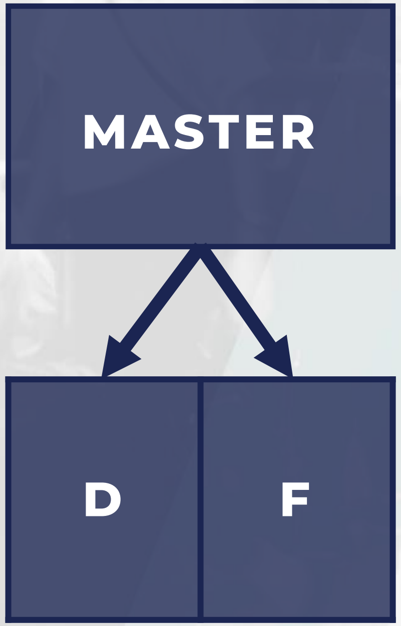
MASTER ATTRIBUTES	DETAIL ATTRIBUTES
M1	D11
M1	D12
M1	D13
M2	D21
M2	D22

GROUP BY Master

MASTER ATTRIBUTES	ARRAY_AGG (DETAIL ATTRIBUTES)
M1	D11
	D12
	D13
M2	D21
	D22

BUILDING COMPLEX OBJECTS WITH MULTIPLE PATH NESTING

1:M



MASTER

D

F

SELECT unnested (denormalized)

MASTER ATTRIBUTES	D ATTRIBUTES	F ATTRIBUTES
M1	D11	NULL
M1	D12	NULL
M1	D13	NULL
M2	D21	NULL
M2	D22	NULL
UNION		
M1	NULL	F11
M1	NULL	F12
M1	NULL	F13
M2	NULL	F21
M2	NULL	F22

GROUP BY Master

MASTER ATTRIBUTES	ARRAY_AGG (D ATTRIBUTES)	ARRAY_AGG (F ATTRIBUTES)
M1	D11 D12 D13	F11 F12
M2	D21 D22	F21 F22 F23

LET'S LOOK AT THE FINAL USER ACCOUNT CODE

THE CODE

QUESTIONS, DISCUSSIONS, AND WHERE WE GO NOW

- **Previous talks:**
  - https://hdombrovskaya.wordpress.com/2018/09/07/our-presentation-on-pg-open-2018/
  - https://hdombrovskaya.wordpress.com/2018/12/30/braviant-holdings-talks-at-2q-pg-conf/
- **EXAMPLE:**
  - https://drive.google.com/file/d/1jnq50-GPA5OaW7Xik-llbt19DScwakkl/view?usp=sharing