
BRAVIANT
A world without credit barriers

Connecting Galaxies: Bridging the Gap Between Applications and Databases

Hettie Dombrovskaya — Braviant Holdings
Jeff Czaplewski
Boris Novikov — HSE University

COMPUTER SCIENCE AND INDUSTRY

- **Most of CS students never saw a real database**
- **Most of their professors never saw them either**
- **Thus, research papers like this one never make it to academic conferences**

I AM ALL ABOUT MONEY

RESPONSE TIME MATTERS

TIME = MONEY

MONEY BY THE NUMBERS

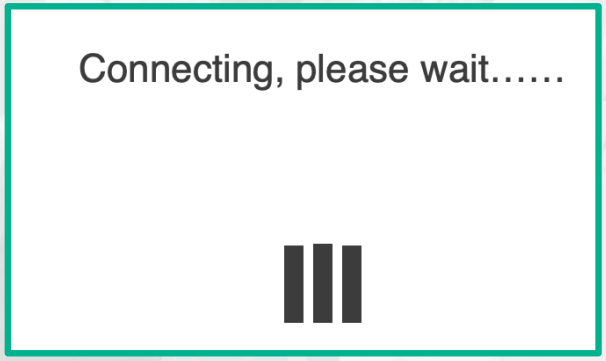
1 sec page load slowdown =>
$1.6 billion lost sales/year

Slowing search results by 0.4 sec –
loss of 8 million searches per day

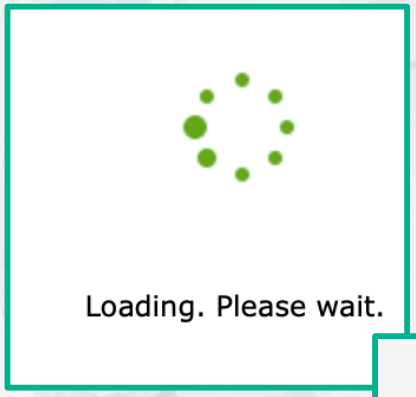
50 % visitors abandon the site, which is not loaded within 3 sec

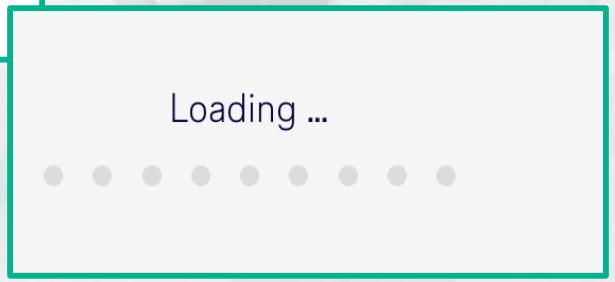
79% visitors will never return again

PLEASE, WAIT…


Connecting, please wait……


Please wait

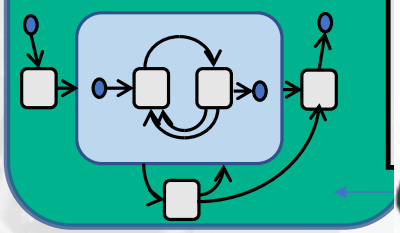

Loading. Please wait.


Loading …

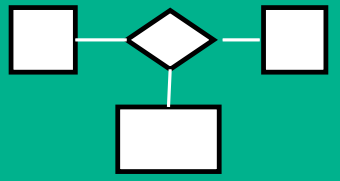
WHY DOES THIS HAPPEN?

WHY GALAXIES

Application Model
(Object-Oriented)



Database Model
(Object-
Relational)



WHY IT IS IMPORTANT?

HOW BAD IT CAN BECOME

SELECT * FROM loans WHERE id=?

executions: 11,500,000
avg time: <10ms
total execution time about 2.5 hours!

Some application controllers would have **over 1,000 database calls for each screen refresh!**

WORST PAGE RESPONSE TIME >=20 sec

MORE HARDWARE?

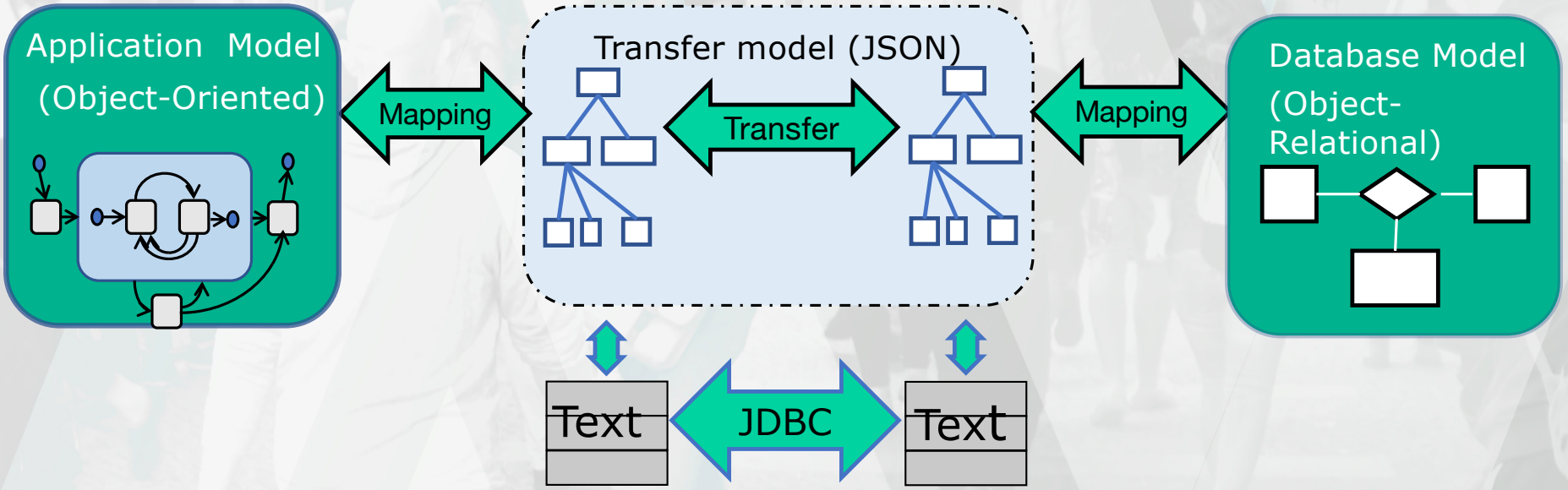
80 thread processors
2.4GHz
512 Gb RAM – almost completely used by disk cache
1066MHz (responses from RAM are 0.9 ns)
I/O 4Gb/sec with avg response time 3ms
I/O utilization: 40%

Even with the best hardware available we can make it only **twice** faster
Current cost**:** **20K** (commodity)
Next – **100K** – somewhat faster (non-commodity)
Next - **1,000K** - twice faster (mainframe)

CURRENT MAIN MEMORY SIZE: 4 TB
AND RESPONSE IS STILL SLOW!

OUR APPROACH

NORM: ORM REPLACEMENT



CONTRACT

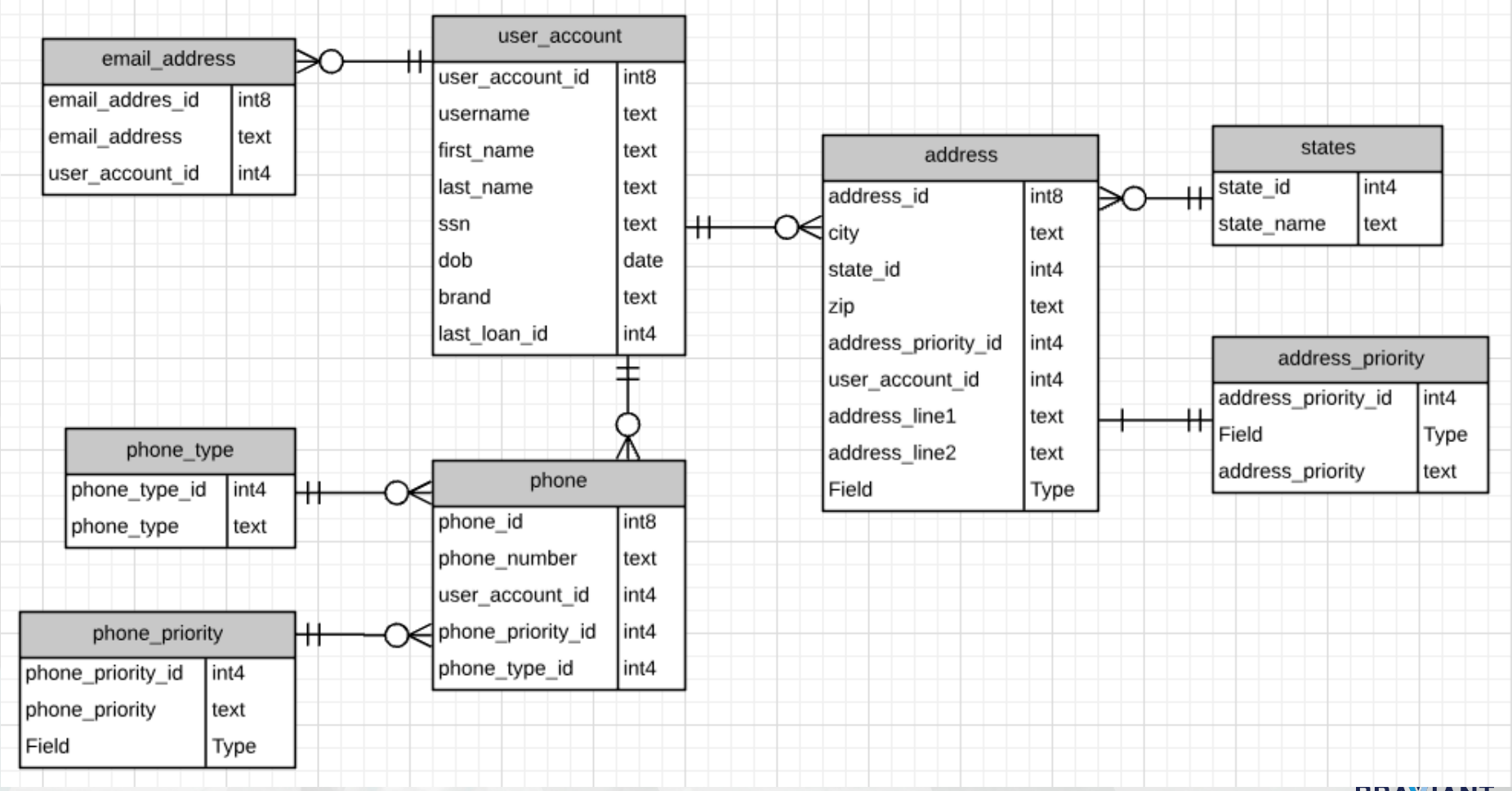
- Both sides (an application and a database) convert internal representations into complex hierarchical object
- Contract establishes object structure implemented on both sides
- Now, for any application endpoint it takes one database call to transfer data to and from database

App

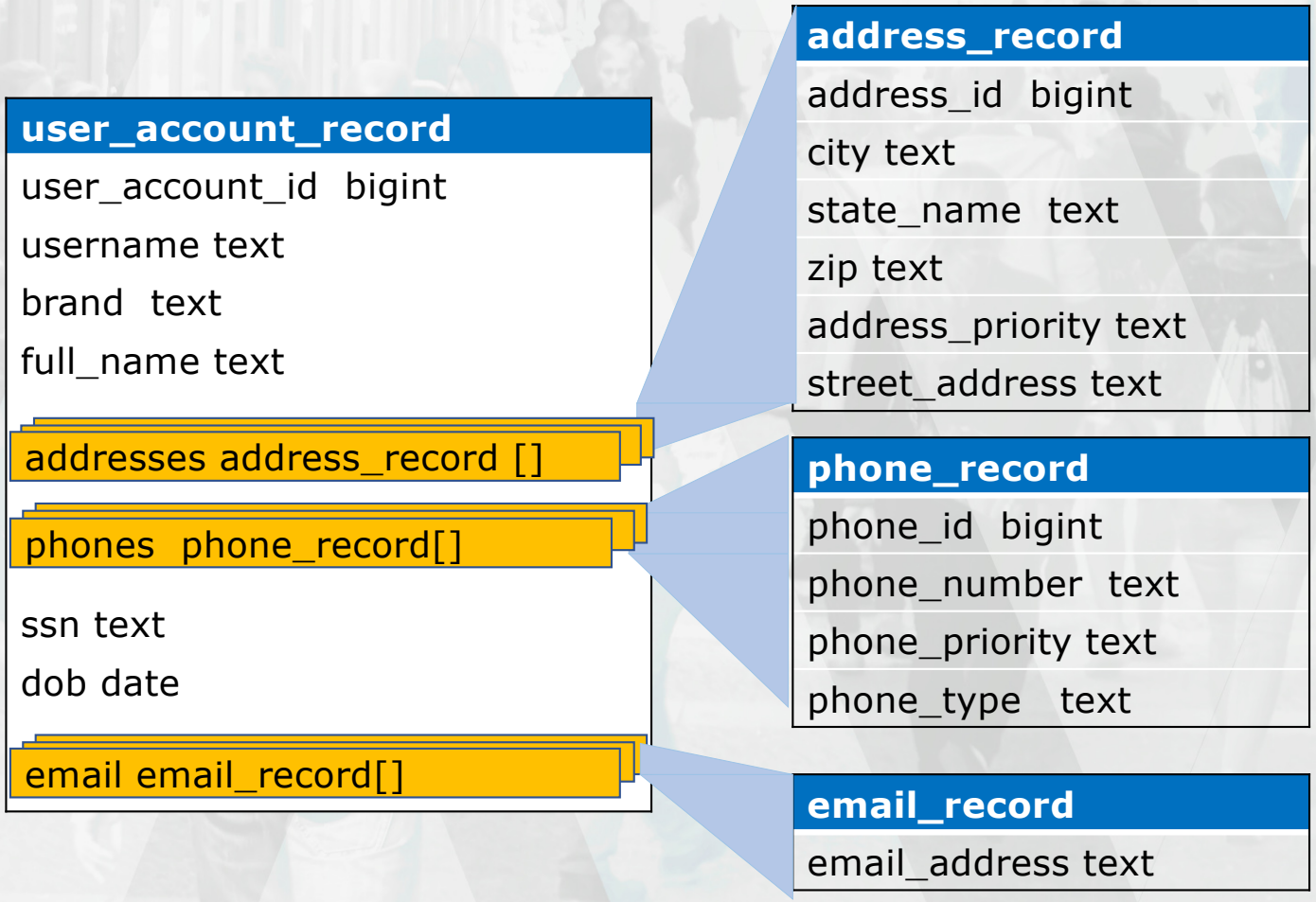
DB

IMPLEMENTATION OVERVIEW

DB SCHEMA (D-OBJECTS)



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
state_name 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

IMPLEMENTATION DETAILS

HOW TO SELECT ALL AT ONCE?

Just SELECT!

```
SELECT user_account_id,
      username,
      ssn,
      dob
      brand
FROM user_account
```

WE CAN CREATE A TYPE

```
CREATE TYPE user_account_record as (
user_account_id  bigint,
username text ,
ssn text,
dob date,
brand  text);

SELECT row(user_account_id,
      username,
      ssn,
      dob,
      brand)::user_account_record
FROM user_account
```

NESTED COMPLEX OBJECTS

```
CREATE TYPE user_account_record AS (
user_account_id  bigint,
username text ,
ssn text,
dob date,
brand text,
addresses address_record[],
phones phone_record[],
email_address email_address_record[]
);
```

AND NOW WE HAVE A PROBLEM!

We wrap SELECT statements into functions which return sets of records

```
CREATE FUNCTION select_user_account (p_id bigint)
RETURNS SETOF user_account_record AS
…

SELECT * FROM select_user_account(12345)
```

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

Nested types definitions are lost!

USING JSON FOR NESTED OBJECTS

```
SELECT user_account_id,
username ,
brand,
json_build_object (’address_id’, address_id,
      ‘street_address’, address_line1
      ‘city’,  city,
      ‘state’, state_name,
      ‘zip’, zip) AS address
FROM user_account u
JOIN address a
  ON a.user_account_id=u.user_account_id
JOIN …
```

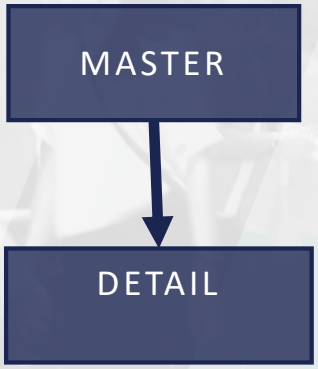
RETURNING RESULT SET AS JSON

```
{
  "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

BUILDING COMPLEX OBJECTS WITH SIMPLE NESTING

1:M



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

```
SELECT
   master.columns,
   array_agg((detail.columns)::T_detail_type)
FROM master JOIN detail ON master.id = detail.master_id
GROUP BY master_columns;
```

HOW SELECT LOOKS NOW

```
SELECT
        array_agg(single_item)
    FROM
   (SELECT ROW( user_account_id ,
                email_address ,
                array_agg(row(phone_id,
                    phone_number,
                    phone_priority,
                    phone_type)::phone_record)AS phones,
                array_agg(row(address_id,
                              street_addr,
                              city,
                              state,
                              zip)::address_record) AS addresses
                              )):: user_account_record AS
                single_item) a FROM …
```

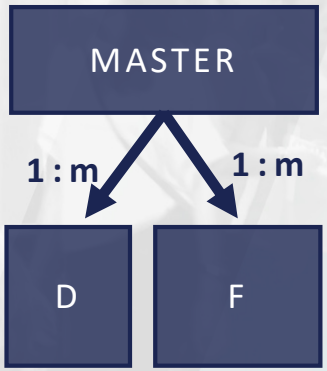
CONVERT TO TEXT AS THE LAST STEP

```
CREATE OR REPLACE FUNCTION array_transport (all_items  anyarray) RETURNS
SETOF text
RETURNS NULL ON NULL INPUT
LANGUAGE plpgsql   AS
$body$
DECLARE
  item  record;
BEGIN
FOREACH   item  IN array all_items
LOOP
   RETURN next( to_json(item))::text;
END LOOP;
END;
$body$;
```

DATA MODIFICATION

```
UPDATE address and DELETE phone:

{ ”user account id”: ”1”, ”addresses”: [
{ ”address id”:”10”,
”street address”:”111 MyStreet”
}
{ ”phone id”: 22, ”command”: ”delete” }
]
  }


UPDATE full name and INSERT email address:

{ ”user account id”: ”1”,
   ”full name”: ”Name LastNameNew”,
      [ {”email address”:”newusername@email.com”} ]

}
```

BUILDING COMPLEX OBJECTS WITH MULTIPLE PATH NESTING



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

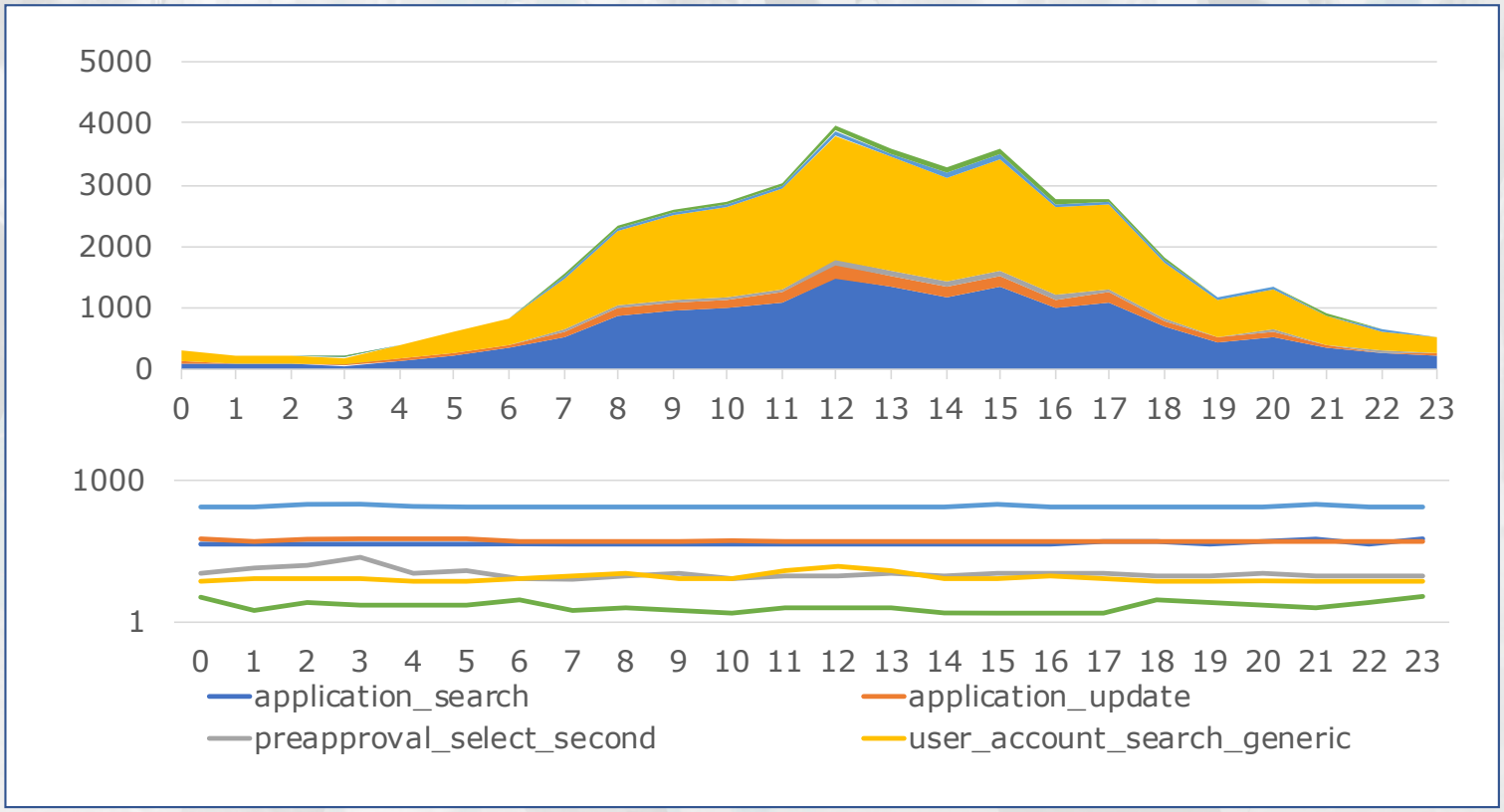
GROUP BY Master

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

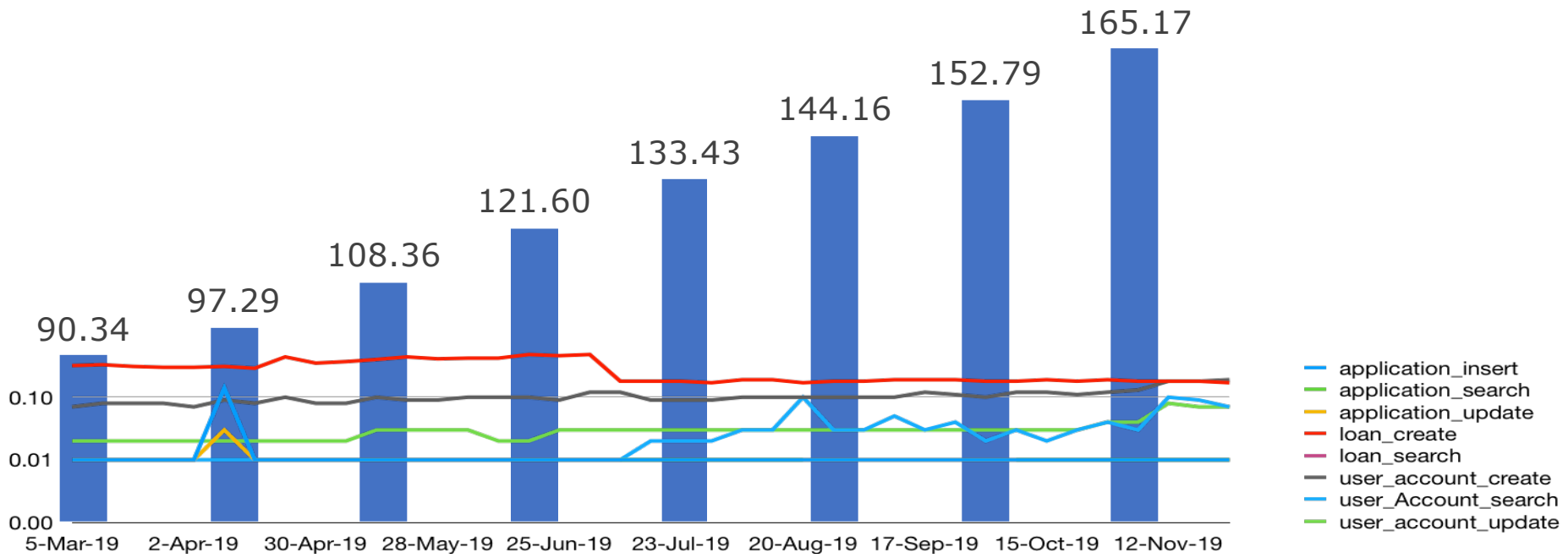
```
SELECT
  m, array_agg(d), array_agg(Ff)
FROM (
SELECT m, d, NULL
FROM master JOIN D . . .
UNION ALL
SELECT m, NULL, f
FROM master JOIN F . . .)
GROUP BY m
```

IT WORKS!
(PERFORMANCE)

AVG EXECUTION TIME AND AVG OPS/MIN PER HOUR



CONSISTENT PERFORMANCE OVER TIME



RELATED AND UNRELATED WORK

WHY NOT MONGO?

- NORM allows flexible hierarchies
- RDBMS search engine inside(PostgreSQL beats Mongo on performance)

WHY NOT STANDARD *json_build* FUNCTIONS?

- Building JSON in SELECT is slow

WHY NOT STORE JSON?

- The search inside JSON is slow

ToroDB: conversion in the opposite direction
PostgREST: conversion only, no complex types

FUTURE WORK

- **Using JSON schema as a source for a contract**
- **Types generation automation**
- **Code generation automation**
- **Creating approved extension**

CONCLUSIONS

- Response time is critically important from the business perspective
- Dominating technologies (Object-Oriented) do not support interactions in terms of respective logical models
- NORM is a technology based on a concept of contract which
  - Allows to use RDBMS search capabilities, at the same time preserving complex objects structures
  - Can be easily adopted by application developers
  - Helps to develop high-performant applications without maxing out hardware resources