Are Stored Procedures a Good Thing?

Bryn Llewellyn

Friday, 15-July-2022

YFTT YugabyteDB Friday Tech Talks

yugabyteDB

History Lesson

When and why did RDMSs first support stored procedures?

- Stored procedures were first supported by RDBMSs in the late 1980s

  Back then, there were only commercial RDBMSs

- Motivation:
  - So-called “run authority”
  - Ownership of responsibility for correct SQL
  - *Esp.* guaranteed atomicity for multi-statement transactions
  - Round-trip reduction for multi-statement transactions

Some Ancient Wisdom

Modular software construction—decades-old wisdom

- Large software systems must be built from modules
- The RDBMS is a module—no less when it's a Distributed SQL system
- "Hard shell" paradigm
- "Result happiness" versus "Result misery" *

* *"Annual income twenty pounds, annual expenditure nineteen nineteen and six, **result happiness**. Annual income twenty pounds, annual expenditure twenty pounds ought and six, **result misery**."* — David Copperfield, 1850

Large software systems must be built from modules

- A module encapsulates specified, coherent functionality
- An API exposes the functionality
- All implementation details are scrupulously hidden behind this API
- Nobody would dream of challenging these notions

The RDBMS is a module—no less when it's a Distributed SQL system

- When an application uses an RDBMS, this is surely a module at the highest level of top-down decomposition
- The structure of the tables, the rules that constrain their rows, and the SQL statements that read and change these rows, are the implementation details
- The API defines and implements the set of atomic business transactions and queries that the database must support
- PostgreSQL, and therefore YSQL, provide subprograms in SQL and in PL/pgSQL to express the API

The “hard shell” paradigm

Use stored procedures*
to encapsulate
the RDBMS’s functionality
behind an impenetrable hard shell API

** I prefer to say “**subrograms whose definitions are stored in the database and that execute in the same process that top-level SQL executes in**”. But “stored procedures” will do as a shorthand.*

The “bag of tables” paradigm

“We don’t use stored procedures.”

All our table have a single owner
and live in a single schema.
Client code can read change
all table content
and even drop and create tables.

“Result happiness” versus “Result misery”

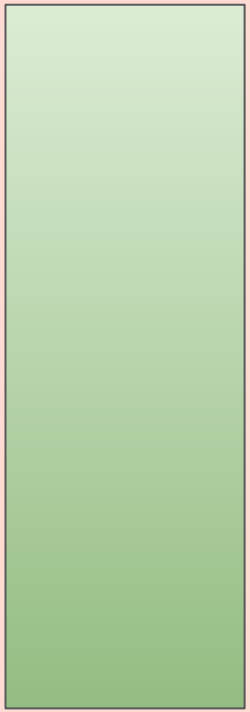
- I’ve spoken to a huge number of developers of database application over the years

- Those who follow the ***hard shell*** paradigm:
  - Are mainly *happy* with their apps
  - Express themselves coherently
  - Explain well how their apps are architected
  - Ask clear questions
  - Make sensible requests for enhancements

- Those who follow the ***bag of tables*** paradigm:
  - Are mainly *miserable*
  - Are hard to understand

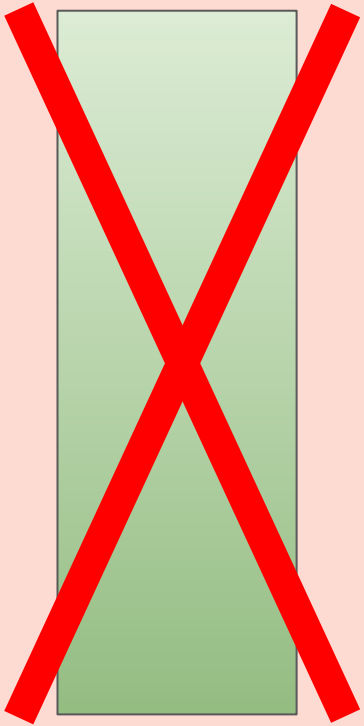
Hard Shell in Easy Pictures

public



“app” database

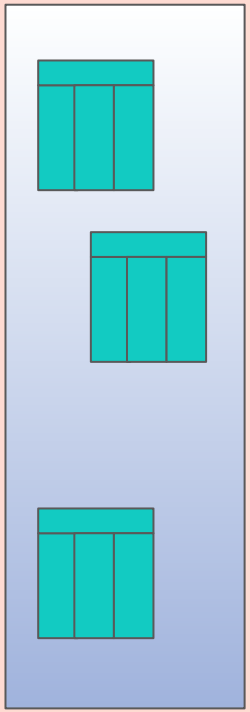
public



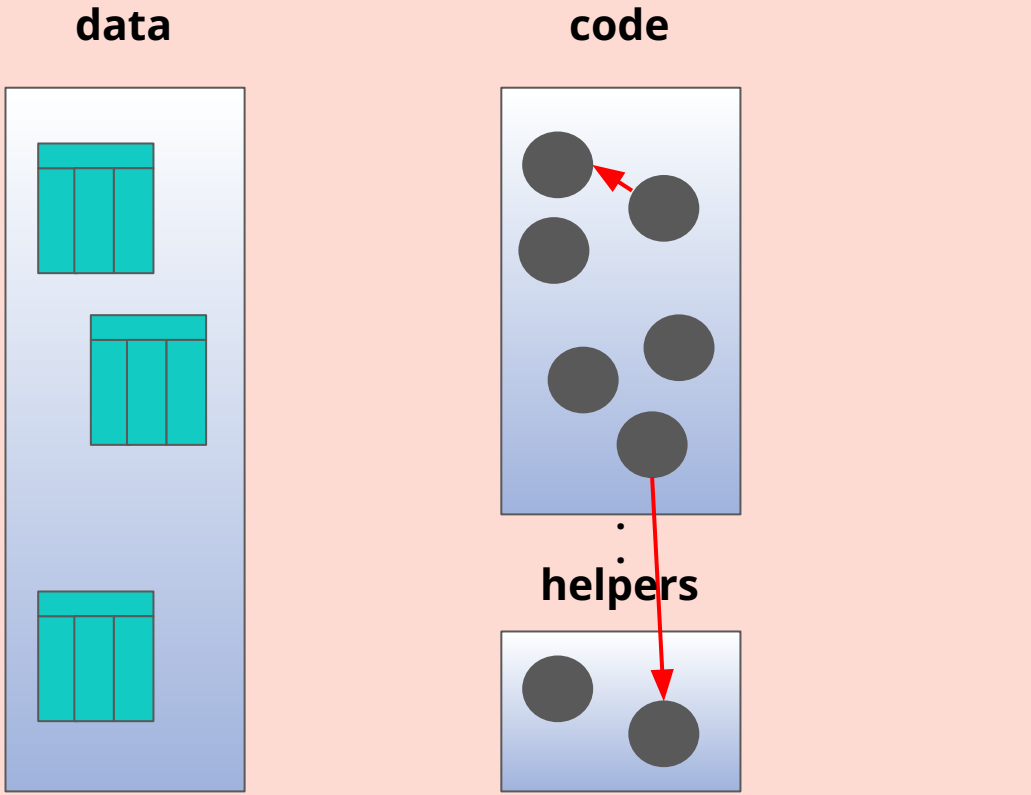
“app” database

“app” database

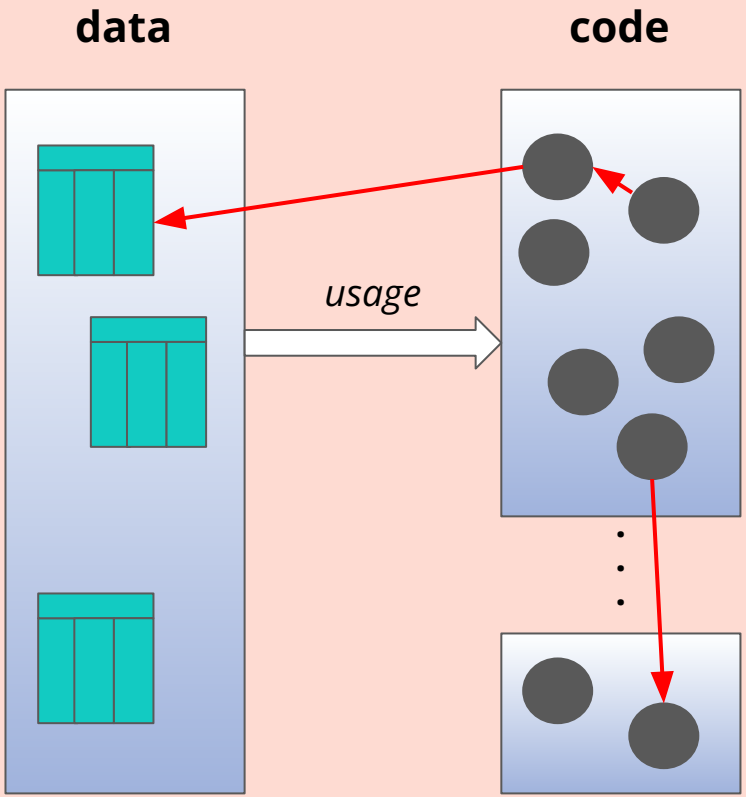
data



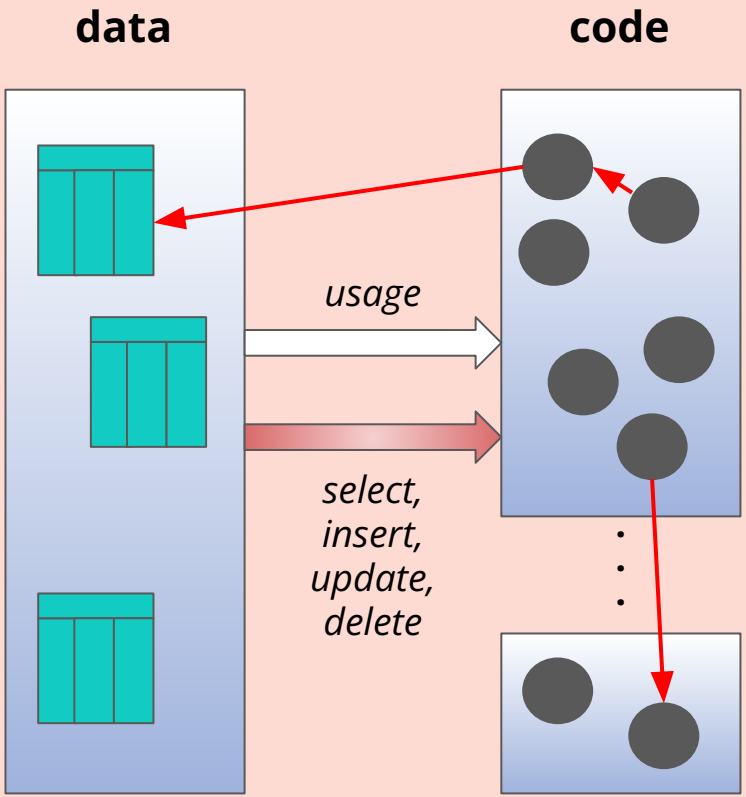
“app” database



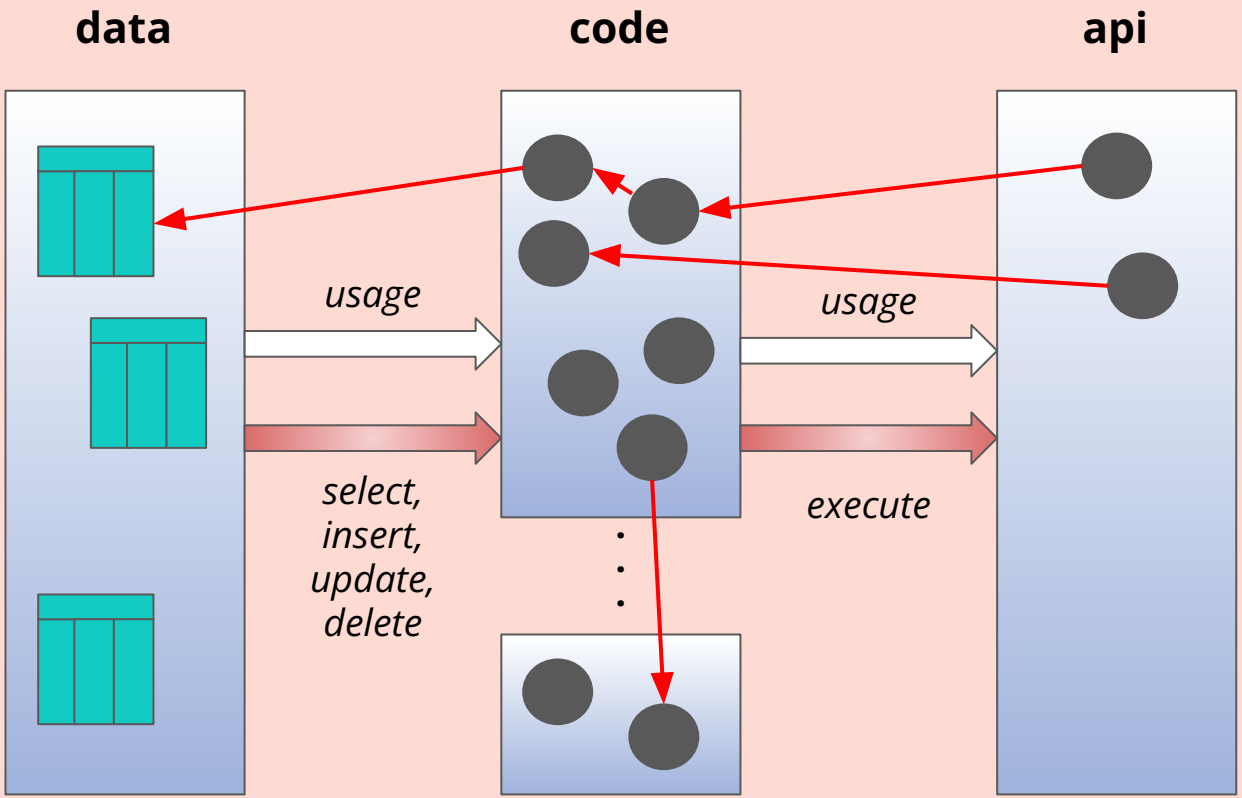
“app” database



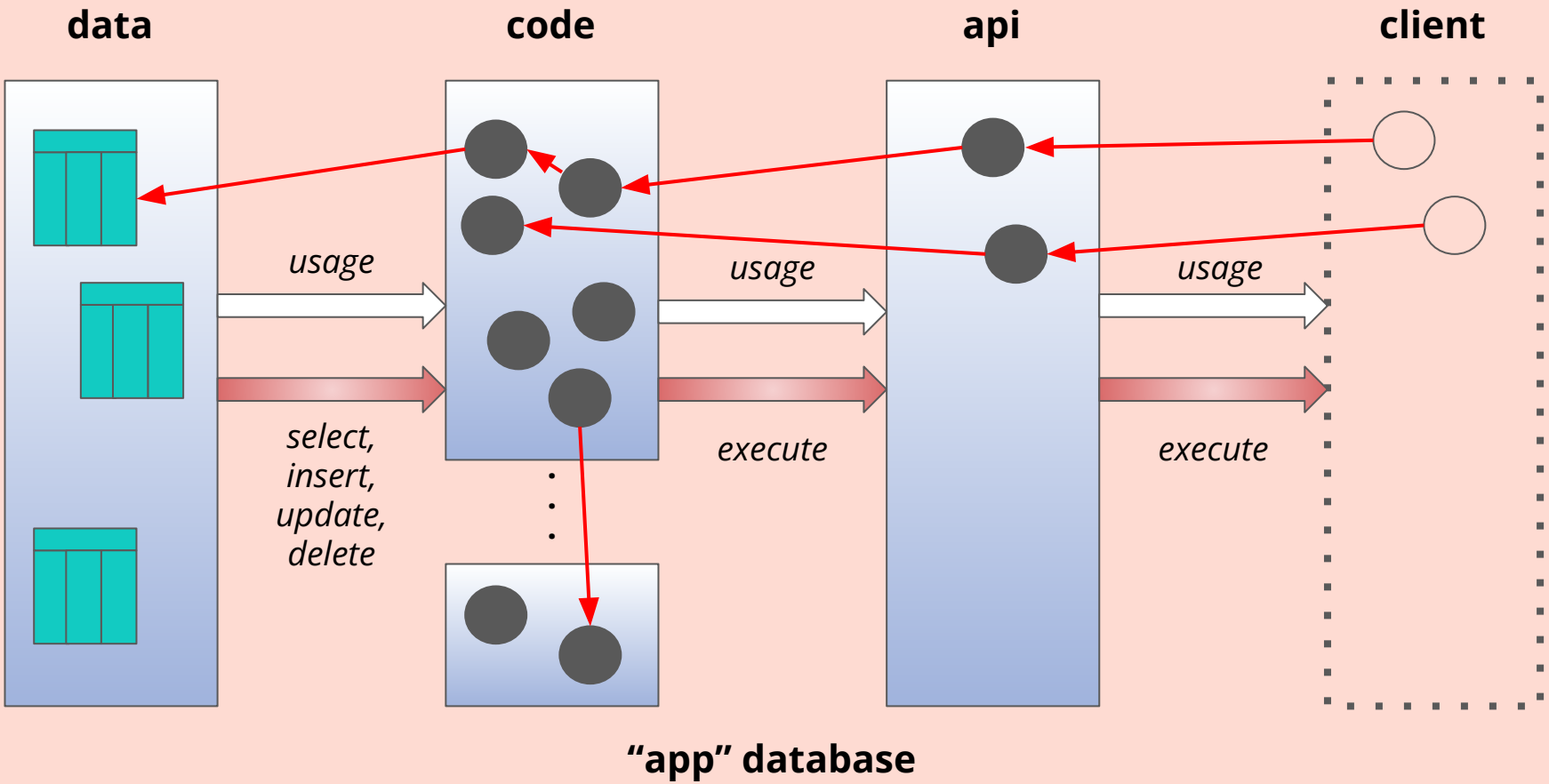
“app” database



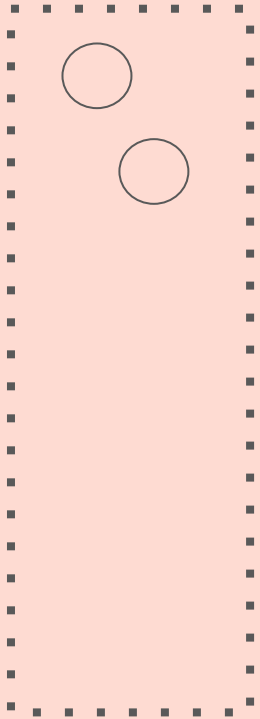
“app” database



“app” database

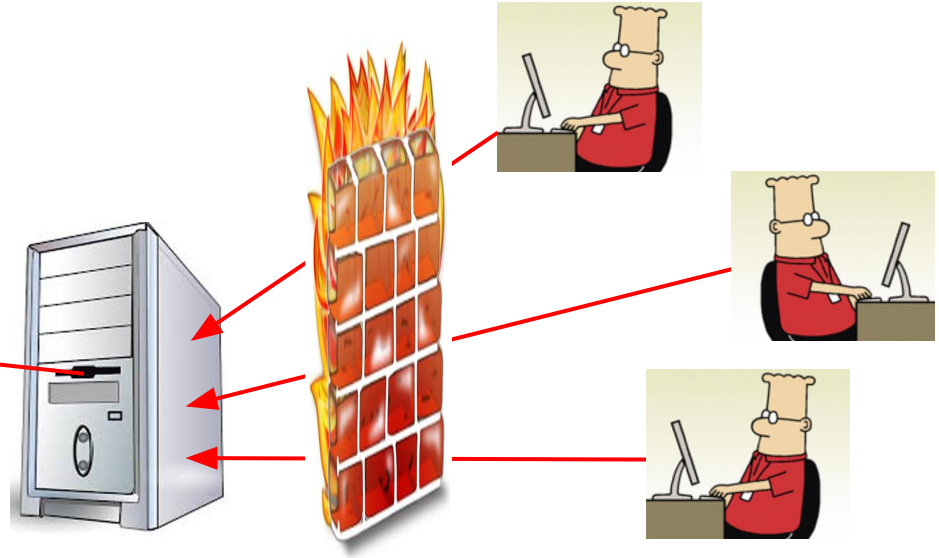


client

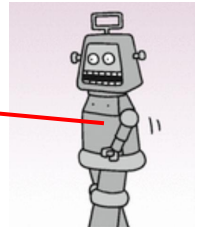


“app” database

ui_client



robot_client



“app” database

Don’t Let This Happen…

Everybody has seen something like this…

Express the API as a Set of JSON-In / JSON-Out Procedures

Client-side environments have different type systems than YSQL

- JSON was invented as a generic data interchange format between systems with different type systems

- All modern client-side programming environments have built-in functionality to transform, in each direction, between an arbitrarily complex compound value and its JSON representation

- YSQL inherits PostgreSQL's corresponding built-in functionality

- The natural, easy, and best design choice is to parameterize the hard shell API as JSON-in / JSON-out procedures
  - *"REST, JSON, And All That: A Memorable History of Client/Server communication"*

Why express the API as procedures and not functions?

- Procedures and functions are different

- A procedure *does* something
  - It's invoked with the *call* statement—meaning "do this"
  - It's named with an imperative verb (phrase)
  - But it can also have "out" arguments

- A function *names* a computed value
  - It's invoked as a term in an expression (in SQL or in PL/pgSQL)
  - It's named with an noun (phrase)—just as you name a column or a variable
  - Functions should not have side-effects
  - So a function with "out" arguments is a nasty anti-pattern

Every single API subprogram might need to *do* something

- Something can always go wrong—like with the Oracle Partner Store's *ORA-06550*
  - Even a query can go wrong if it expects exactly one row for a business unique key
  - It might get no rows — like you mentioned a non-existent order number
  - It might get *many* rows — meaning and earlier constraint-enforcement error

- Such application errors must *never* escape the database
  - the ORA-06550 error says that the app constructed a subprogram that had a syntax error

- So these "unexpected" errors (i.e. developer bugs) must be recorded in an *incidents* table
  - Inserting a row is *doing* something!

The Use Case for the Demo App

Classic (agnostic) master-details Create and Read

```
create table data.masters(
  mk uuid default gen_random_uuid()
    constraint masters_pk primary key,
  v text not null
    constraint masters_v_unq unique
    constraint masters_v_chk check(length(v) <= 10));

create table data.details(
  mk uuid,
  dk uuid default gen_random_uuid(),
  v text not null,

  constraint details_pk primary key(mk, dk),

  constraint details_fk foreign key(mk)
    references data.masters(mk)
    match full
    on delete cascade
    on update restrict,

  constraint details_mk_v_unq unique(mk, v));
```

Create new master and details or new details for existing master (bad)

```
create type m_and_ds_ as(m text, ds text[]);

create procedure do_insert(this in m_and_ds_)
  language plpgsql
as $body$
declare
  ...
begin
  begin
    insert into masters(v) values(this.m) returning mk into new_mk;
  exception when unique_violation then
    select mk into new_mk from masters where v = this.m;
    new_master := false;
  end;

  if cardinality(this.ds) > 0 then
    -- Notorious anti-pattern: many single row SQLs in a loop.
    foreach d in array this.ds loop
      insert into details(mk, v) values(new_mk, d);
    end loop;
  end if;
end;
$body$;
```

Aside — Brief tutorial on "cross join lateral" with "unnest(arr)"

```
create table t(
  m text not null,
  d text not null,
  constraint t_pk primary key(m, d));

create type facts as(m text, ds text[]);


with c(v) as (
  select ('Joe', array['fork', 'spoon', 'knife'])::facts)
insert into t(m, d)
select (c.v).m, arr.d
from
  c
  cross join lateral
  unnest((c.v).ds) as arr(d);


select m, d from t order by m, d;

  m  |   d
-----+-------
 Joe | fork
 Joe | knife
 Joe | spoon
```

Create new master and details or new details for existing master (good)

```
create type m_and_ds_   as(m text,  ds text[]);
create type mk_and_ds_  as(mk uuid, ds text[]);

create procedure do_insert(this in m_and_ds_)
  language plpgsql
as $body$
declare
  ...
begin
  begin
    insert into masters(v) values(this.m) returning mk into new_mk;
  exception when unique_violation then
    select mk into new_mk from masters where v = this.m;
    new_master := false;
  end;

  if cardinality(this.ds) > 0 then
    -- Optimal: on single "bulk" SQL.
    with c as (
      select (new_mk, this.ds)::mk_and_ds_ as v)
    insert into details(mk, v)
    select (c.v).mk, arr.d
    from c cross join lateral unnest((c.v).ds) as arr(d);
  end if;
end;
$body$;
```

Recap — Anti-pattern: insert many “details” rows one-by-one in a loop

```
create procedure do_insert(this in m_and_ds_)
  language plpgsql
as $body$
declare
  ...
begin
  begin
    insert into masters(v) values(this.m) returning mk into new_mk;
  exception when unique_violation then
    select mk into new_mk from masters where v = this.m;
    new_master := false;
  end;

  if cardinality(this.ds) > 0 then
    foreach d in array this.ds loop
      insert into details(mk, v) values(new_mk, d);
    end loop;


  end if;
end;
$body$;
```

Recap — Optimal: insert many “details” rows with one single “bulk” SQL

```
create procedure do_insert(this in m_and_ds_)
  language plpgsql
as $body$
declare
  ...
begin
  begin
    insert into masters(v) values(this.m) returning mk into new_mk;
  exception when unique_violation then
    select mk into new_mk from masters where v = this.m;
    new_master := false;
  end;

  if cardinality(this.ds) > 0 then
    with c as (
      select (new_mk, this.ds)::mk_and_ds_ as v)
    insert into details(mk, v)
    select (c.v).mk, arr.d
    from c cross join lateral unnest((c.v).ds) as arr(d);
  end if;
end;
$body$;
```

Read existing master and its details

```
create type m_and_ds_   as(m text,  ds text[]);

create function master_and_details_report(mv_in in text)
  returns m_and_ds_
  language plpgsql
  security definer
as $body$
declare
  m_and_ds m_and_ds_;
begin
  select m.v, array_agg(d.v order by d.v)
  into m_and_ds
  from
    data.masters m
    left outer join
    data.details d
    using (mk)
  where m.v = mv_in
  group by 1
  order by 1;

  return m_and_ds;
end;
$body$;
```

The App-Specific JSON-In / JSON-Out Protocol

procedure insert_master_and_details(j *in text*, j_outcome *inout text*)

No problems

```
call insert_master_and_details(
  '{"m": "Mary", "ds": ["shampoo", "soap", "toothbrush", "towel"]}', '');
```

```
→ {"status": "success"}
```

New-master cannot have dup details

```
{"m": "Arthur", "ds": ["scissors", "saucer", "spatula", "spatula", "scissors"]}
```

```
→ {"reason": "New  master 'Arthur' bad duplicate details: 'scissors','spatula',",
    "status": "user error"}
```

Program bug: forgot to cater for masters_v_chk violation

```
{"m": "Christopher", "ds": []}', '')
```

```
→ {"status": "unexpected error", "ticket": 1}
```

TICKET NO. 1

```
unit:                   procedure code.insert_master_and_details(text, text)
returned_sqlstate:      23514
message_text:           new row for relation "masters" violates check constraint "masters_v_chk"
pg_exception_detail:    Failing row contains (cc93bd34-b68a-4d47-b9e9-0033031cefb7, Christopher).
constraint_name:        masters_v_chk
table_name:             masters
schema_name:            data

pg_exception_context
--------------------
SQL statement "insert into data.masters(v) values(m_and_ds.m) returning mk"
PL/pgSQL function code.insert_master_and_details(text,text) line 17 at SQL statement
SQL statement "call code.insert_master_and_details(j, outcome)"
PL/pgSQL function insert_master_and_details(text,text) line 3 at CALL
```

procedure do_master_and_details_report(j *in text*, j_outcome *inout text*)

No problems

```
call do_master_and_details_report(
  '{"key": "Mary"}', '');
```

```
→ {"status": "m-and-ds report success",
    "m_and_ds": {"m": "Mary", "ds": ["shampoo", "soap", "toothbrush", "towel"]}}
```

Bill doesn't exist

```
{"key": "Bill"}
```

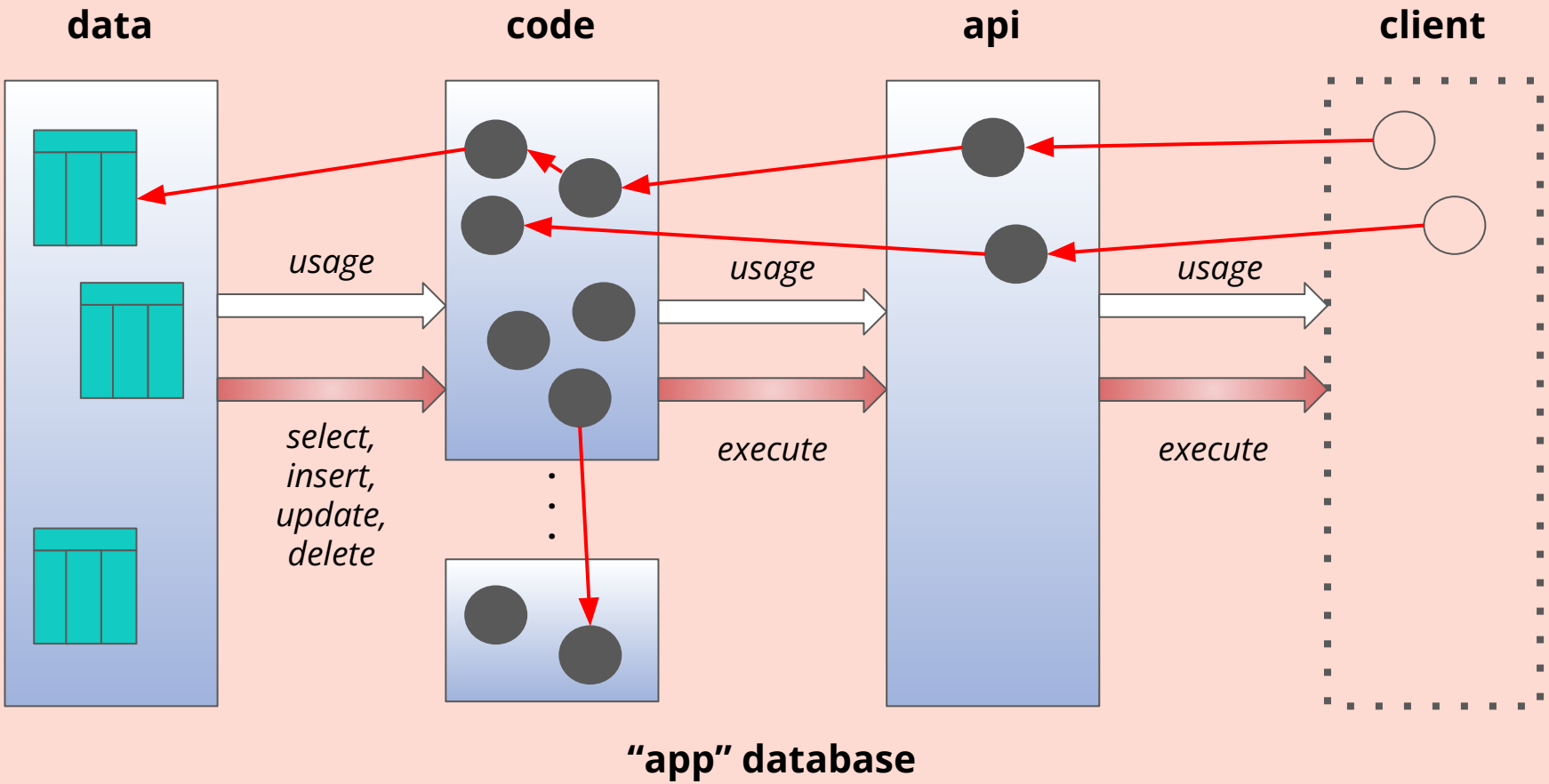
```
→ {"reason": "The master business key 'Bill' doesn't exist", "status": "user error"}
```

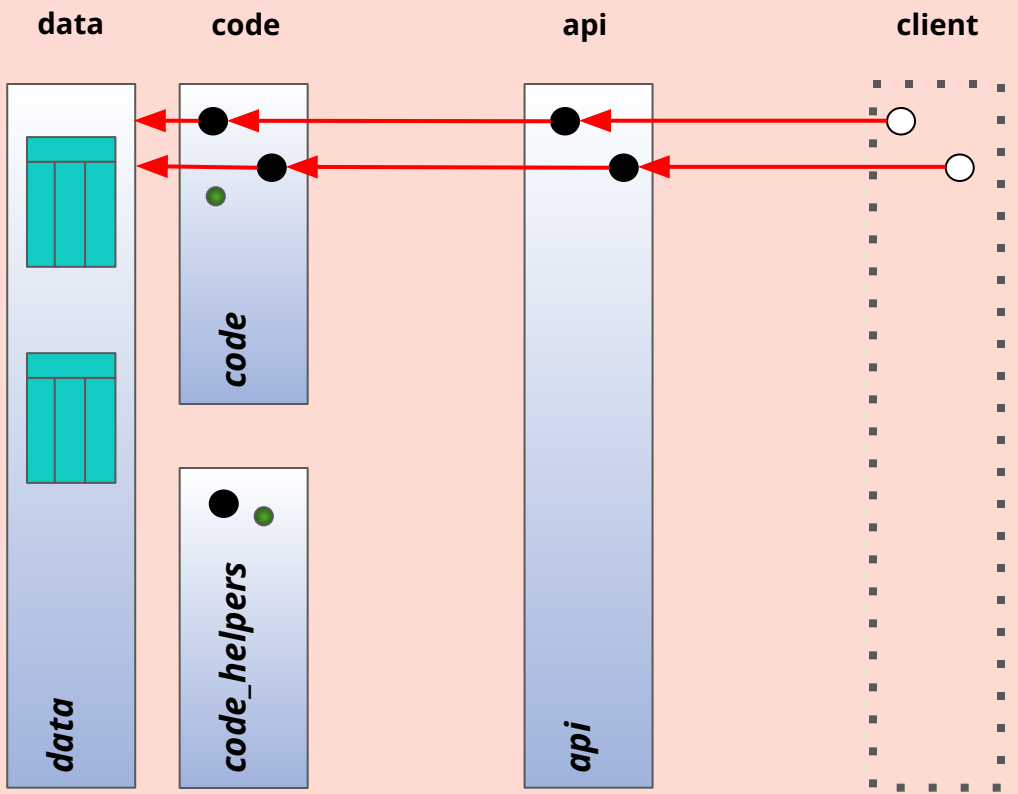
Application program bug: typo "ket" for "key"

```
{"ket": "Fred"}
```

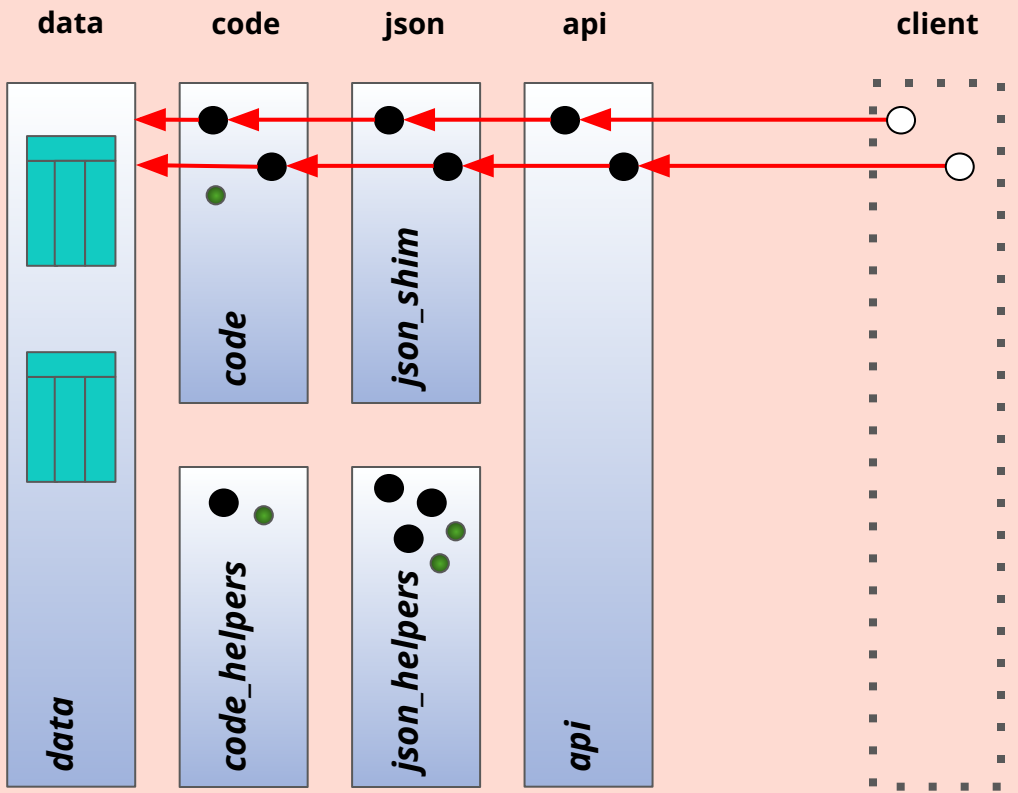
```
→ {"reason": "Bad JSON in: {\"ket\": \"Fred\"}", "status": "client code error"}
```

Users and Schemas: Refined Picture

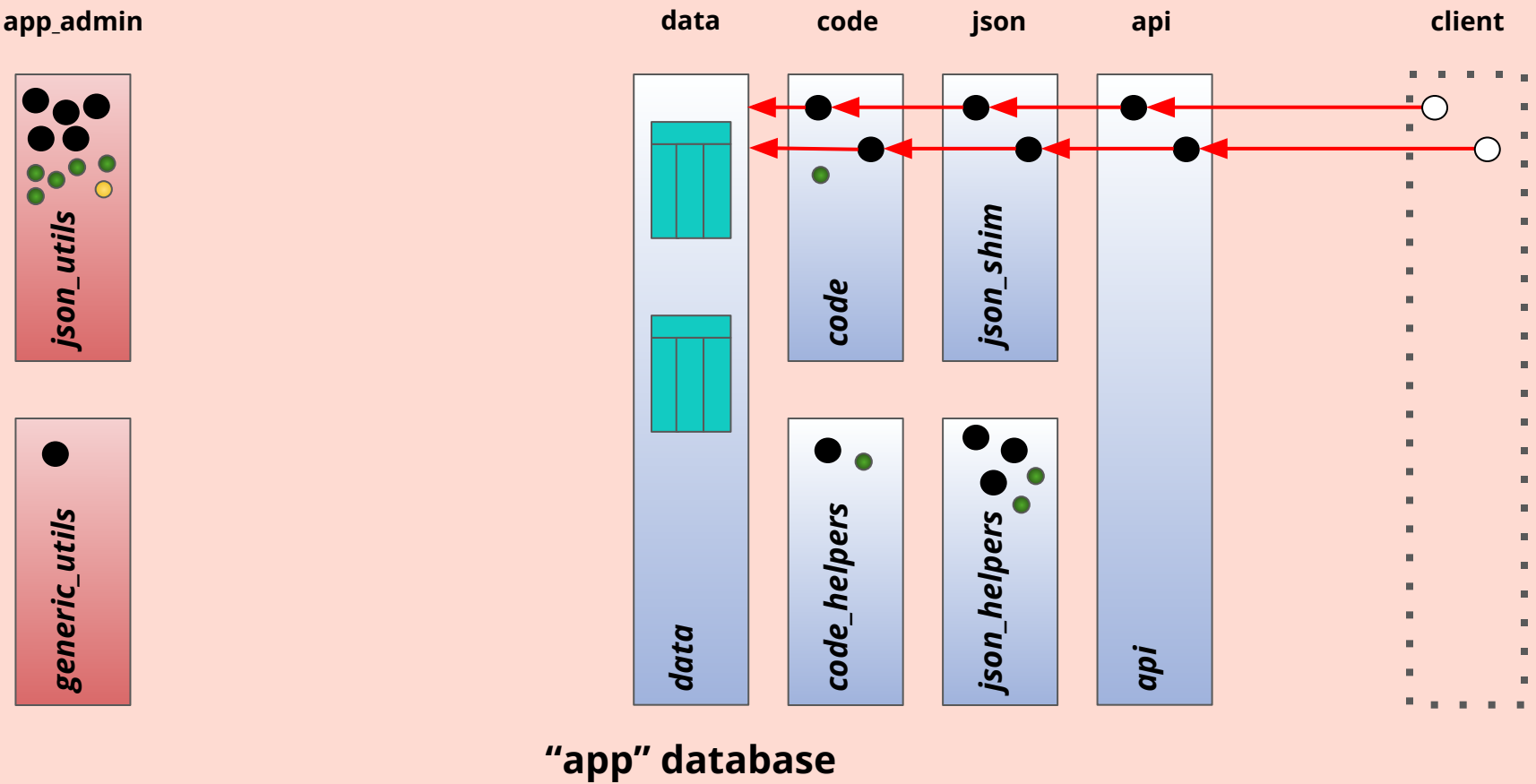




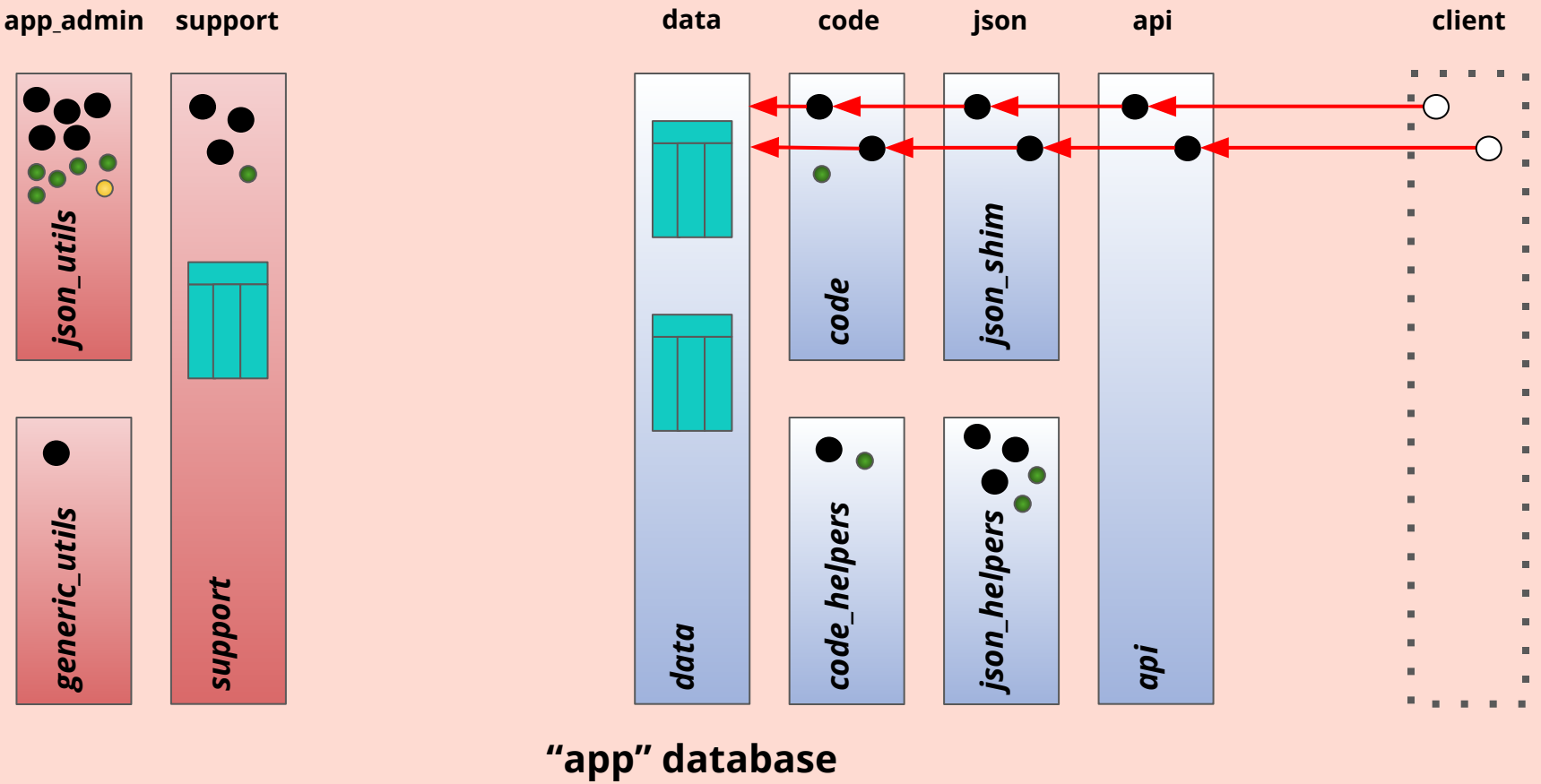
“app” database



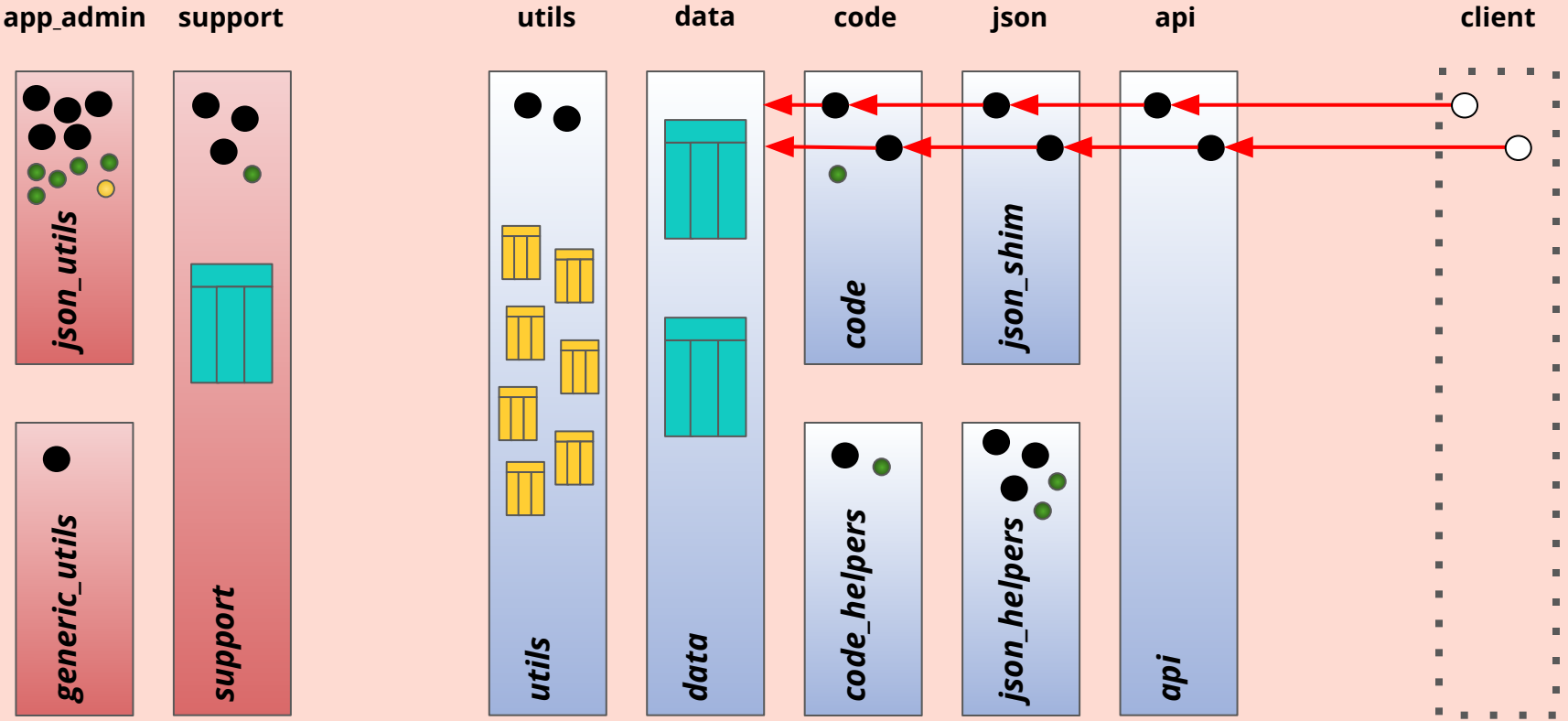
“app” database



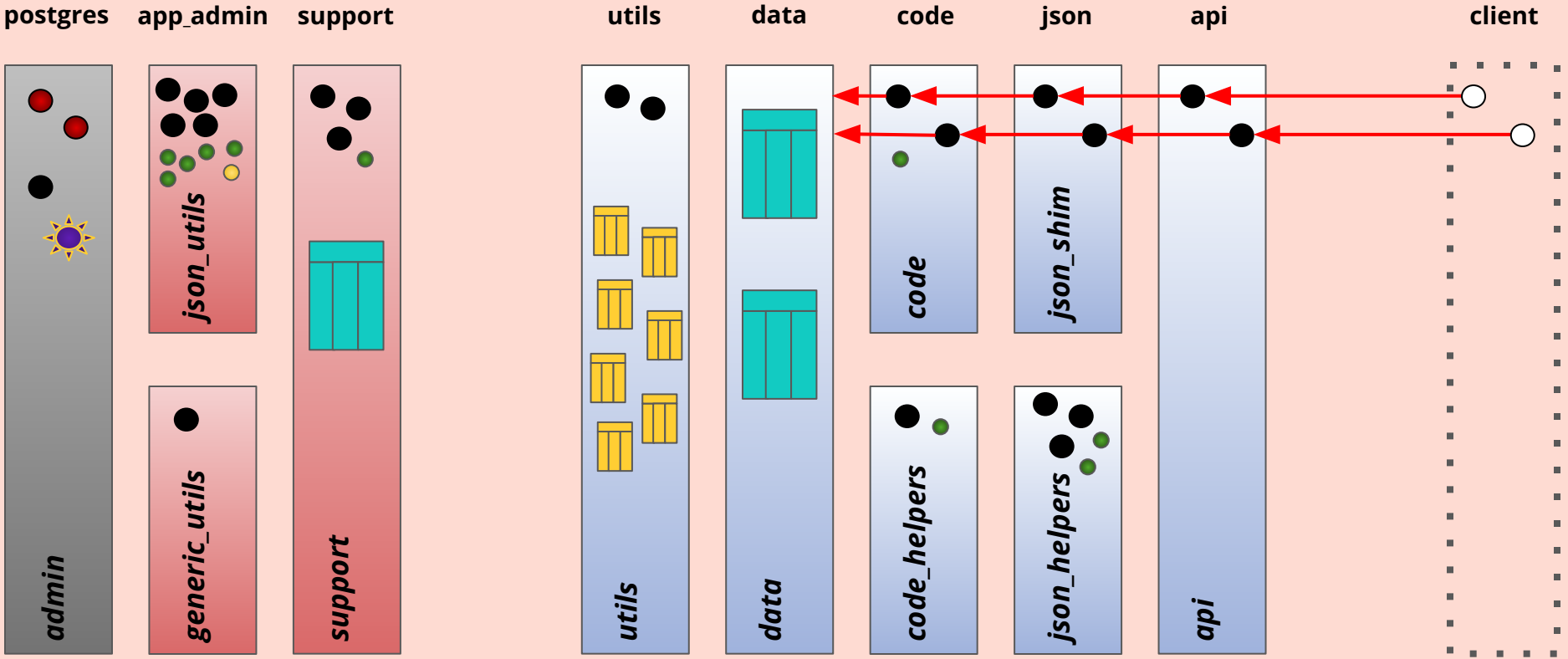
“app” database



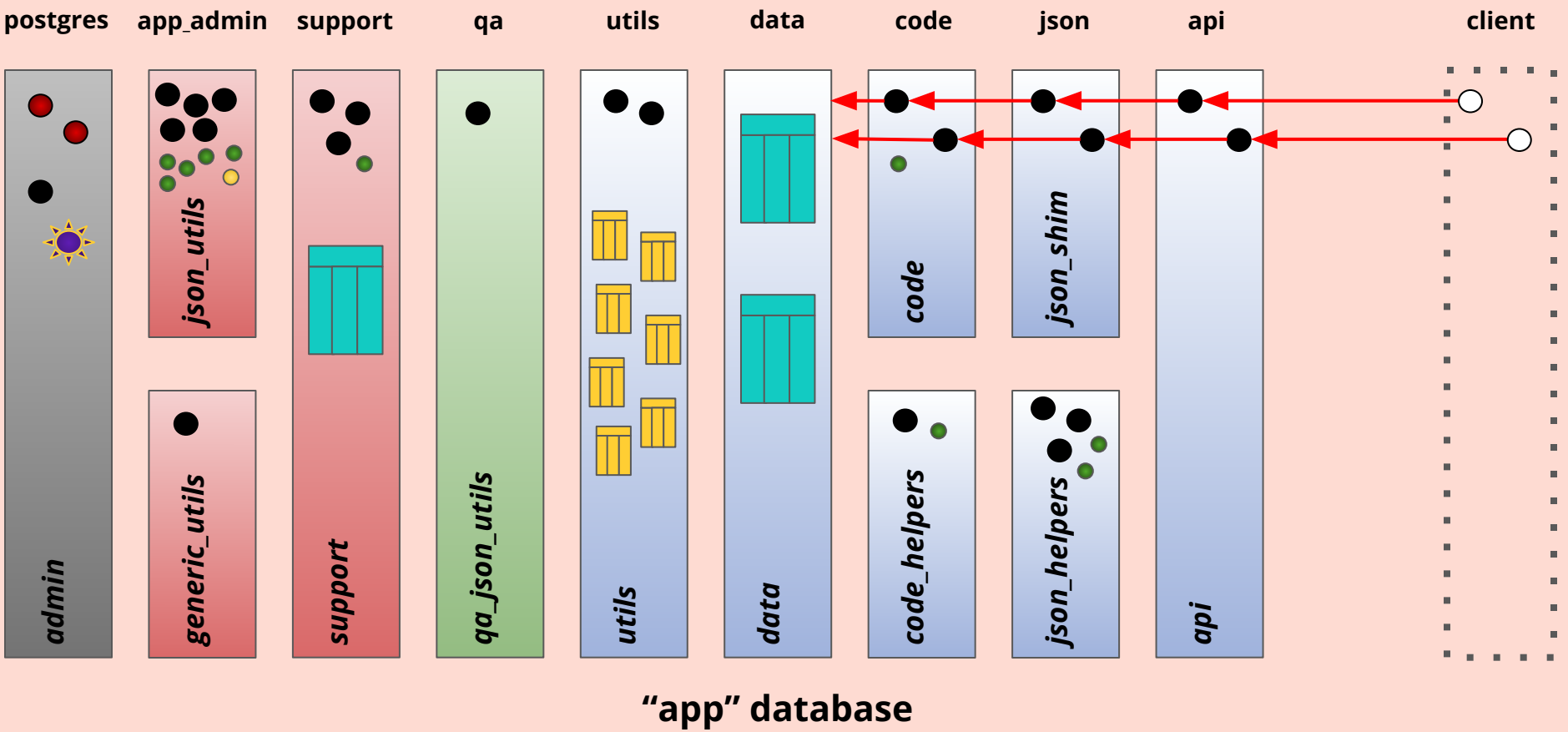
“app” database



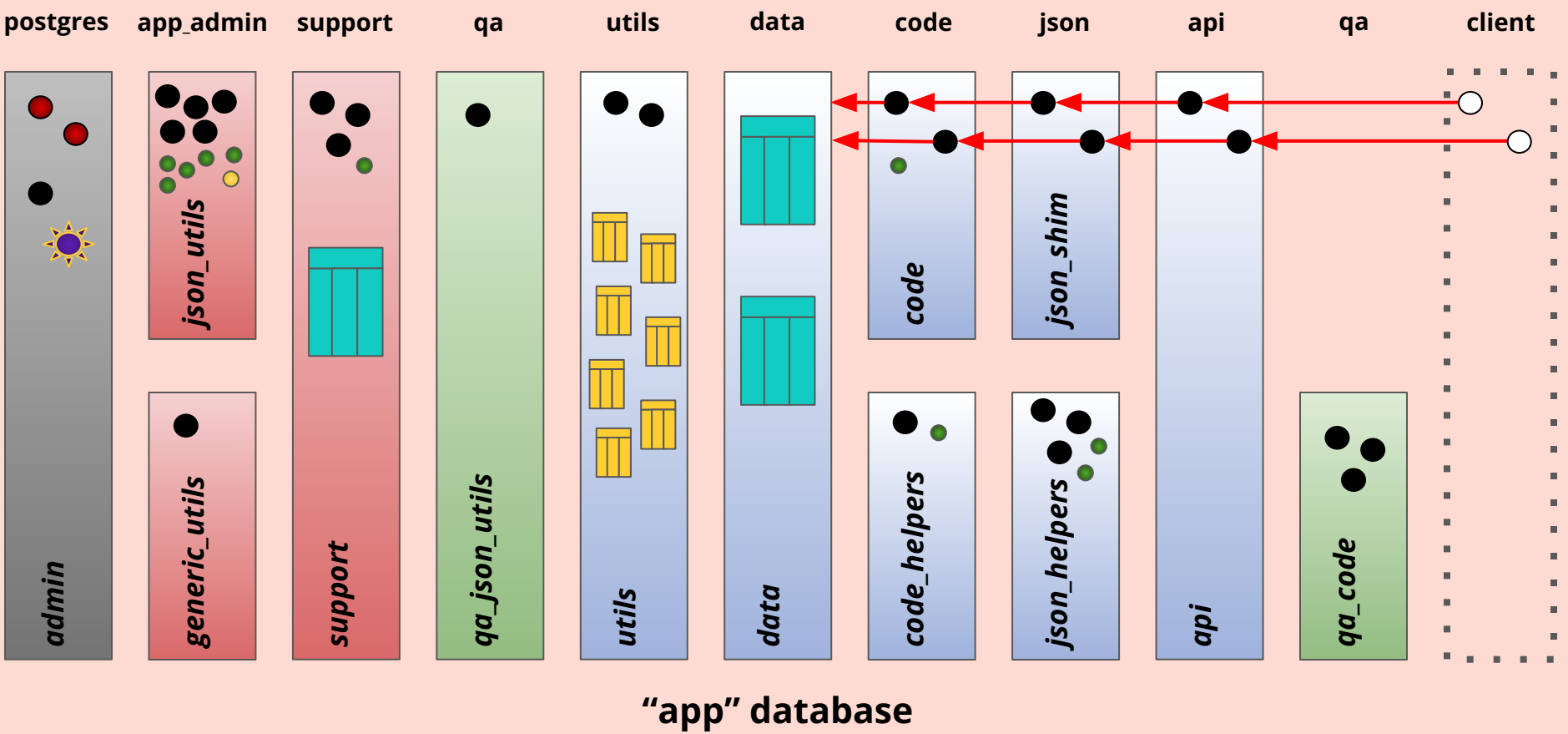
“app” database



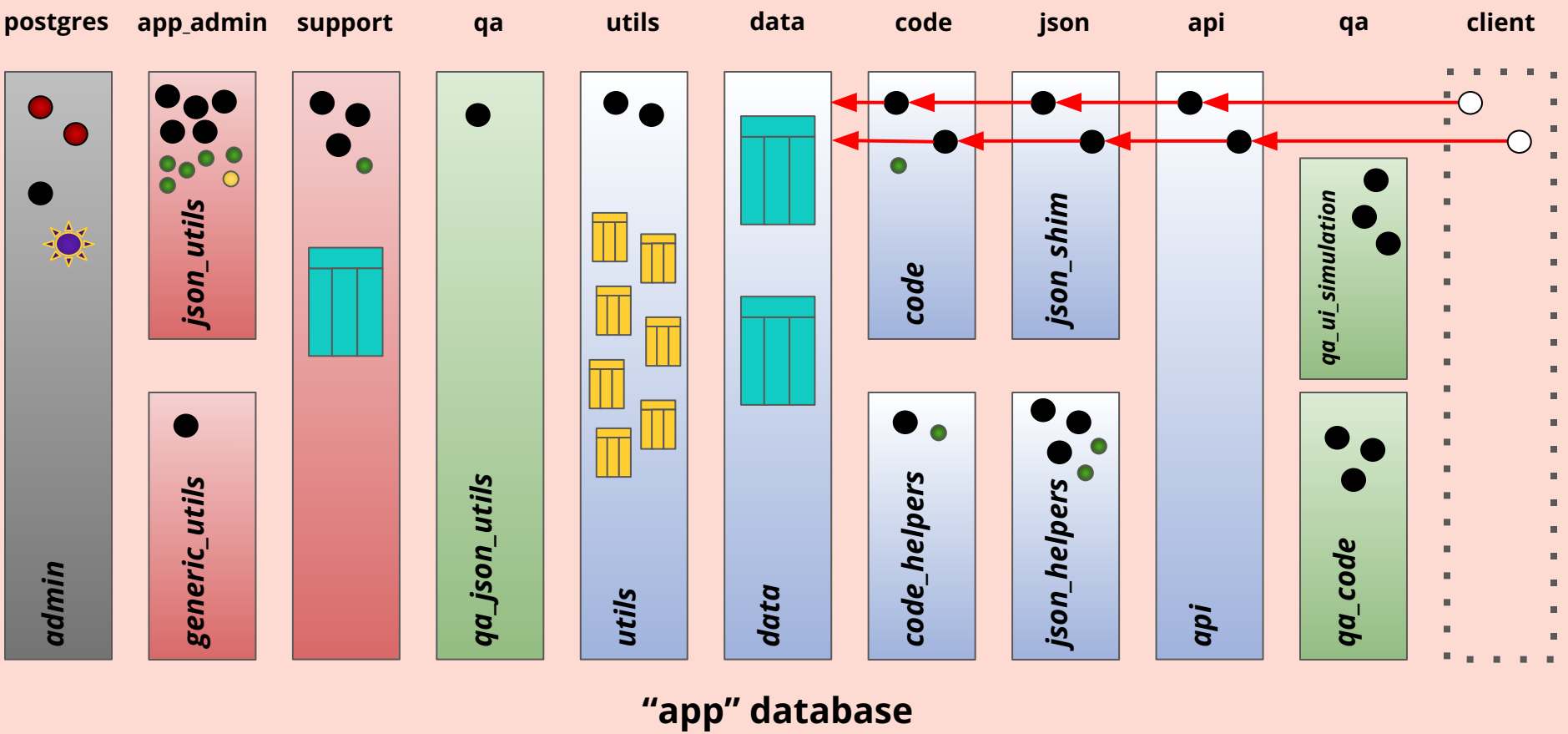
"app" database



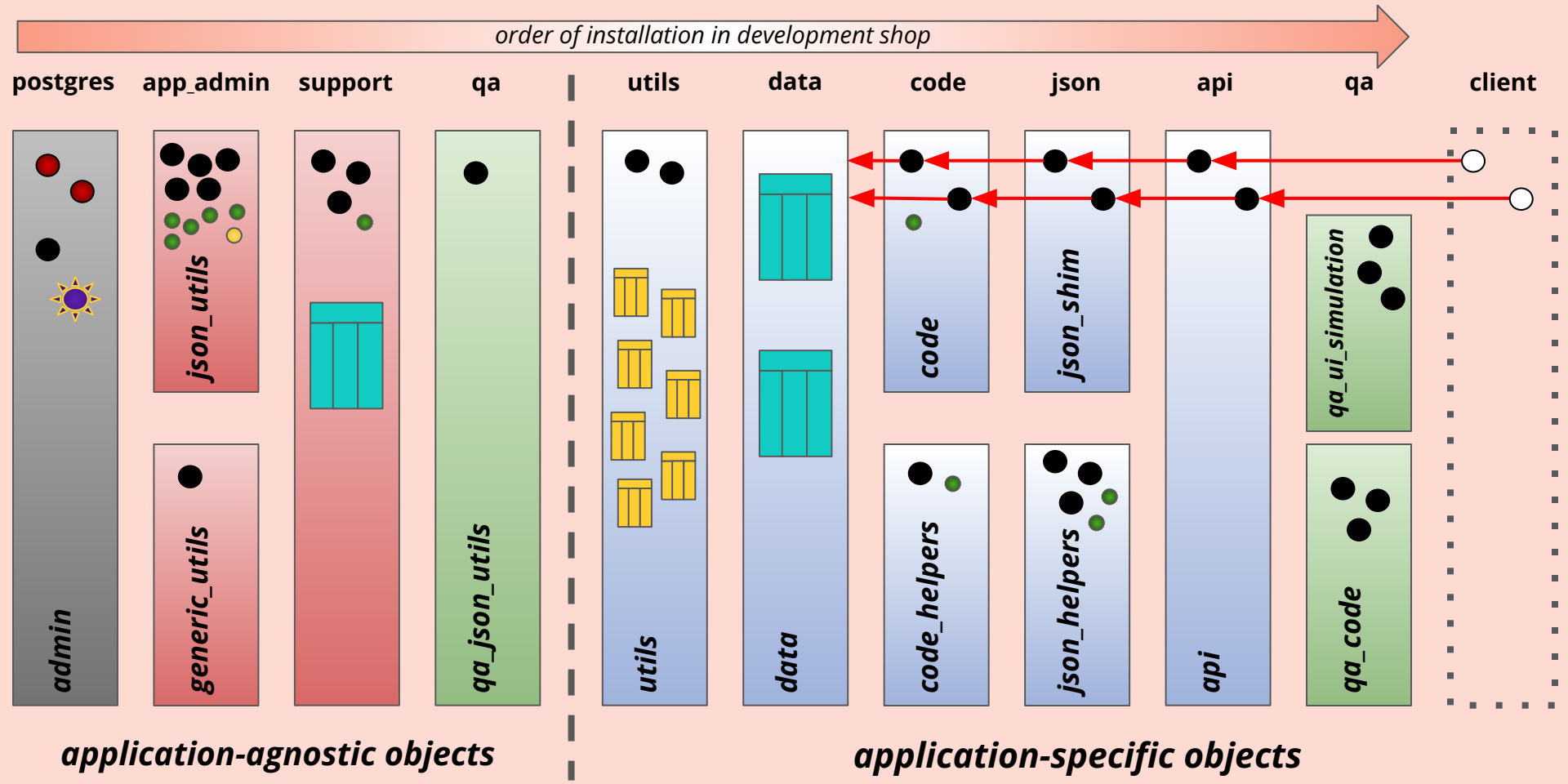
“app” database

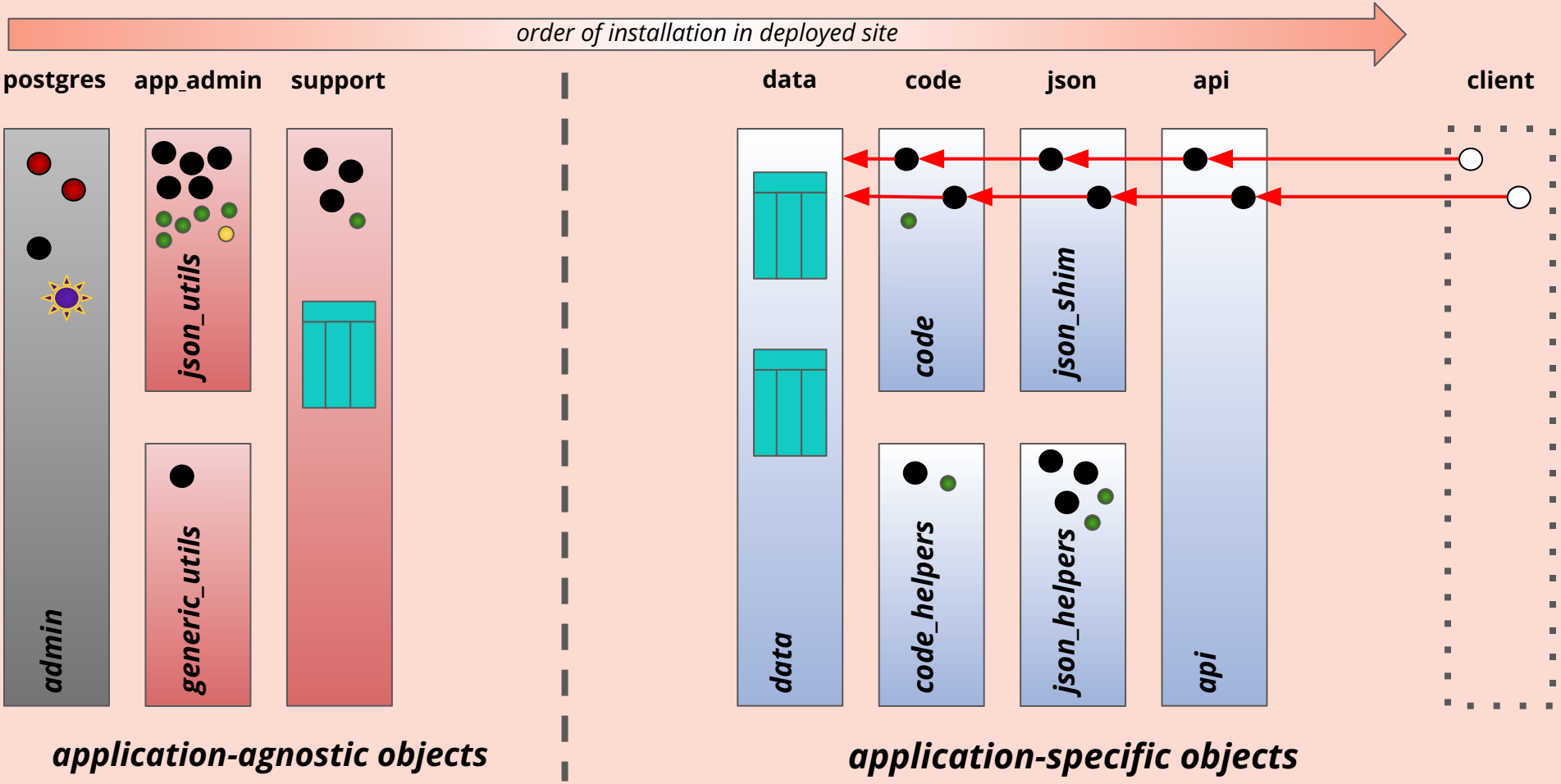


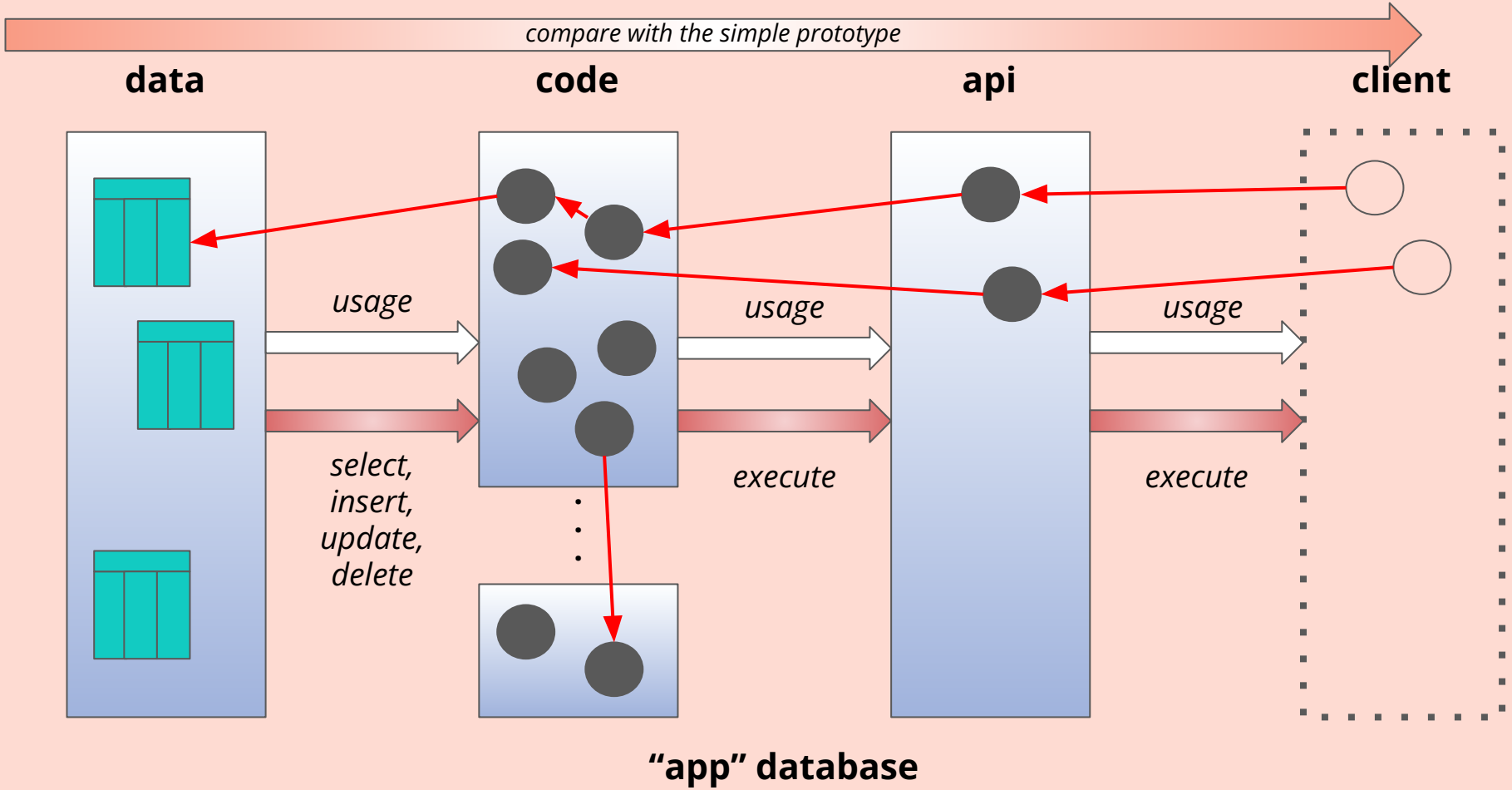
"app" database



"app" database







Comparing the final concept with the prototype

- **Modularity** (separation of skills and concerns using *roles* and/or *schemas*)
  - Ordinary SQL DMLs separated from JSON shim
  - Development-shop-only code separated out
    - QA code added
    - “list my objects” views and table functions added

- **Reusability of the overall design concept**
  - Application-agnostic components separated out
    - JSON schema compliance code parameterized (with own, separate QA)
    - Support subsystem

- **Performance**
  - Model accommodates dedicated development shop performance testing code

And Now the Stage is Set…

Over to the Live Demos

Summary

The Benefits of encapsulating the database behind a hard shell API

- **Correctness**
  - Separation of skills and concerns
  - The right experts own their own tersely coupled modules
  - Data type safety

- **Security**
  - Controlled access to objects — client-code can't change or read tables
  - All that you can do is *call proc1(...)*, *call proc2(...)*, ...
  - Implicit, effortless, SQL-injection proofing

- **Performance**
  - Minimization of client-server round trips
  - Prepare-execute paradigm with no coding effort

Thank You

Join us on Slack: yugabyte.com/slack (#yftt channel)

Star us on Github: github.com/yugabyte/yugabyte-db

YFTT YugabyteDB Friday Tech Talks

yugabyteDB