Un Éléphant de Première Classe

A bit of History

1970

A Relational Model of Data for Large Shared Data Banks

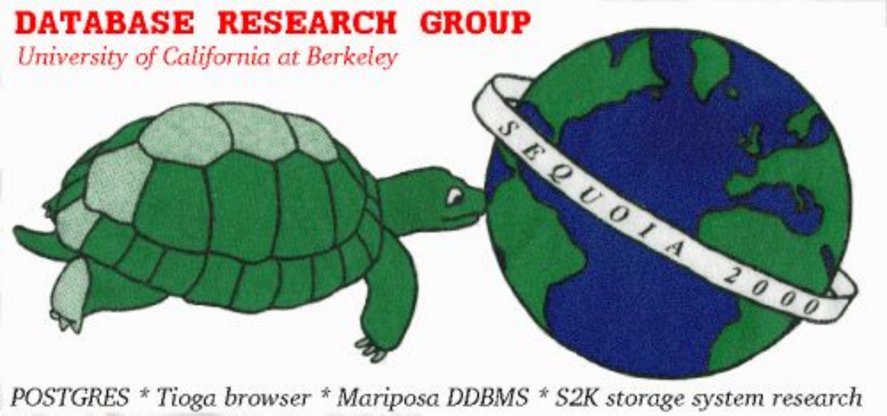
E. F. Codd
IBM Research Laboratory, San Jose, California

1970 - 1979 System R

1973 - 1985 Ingres & Quel

1989

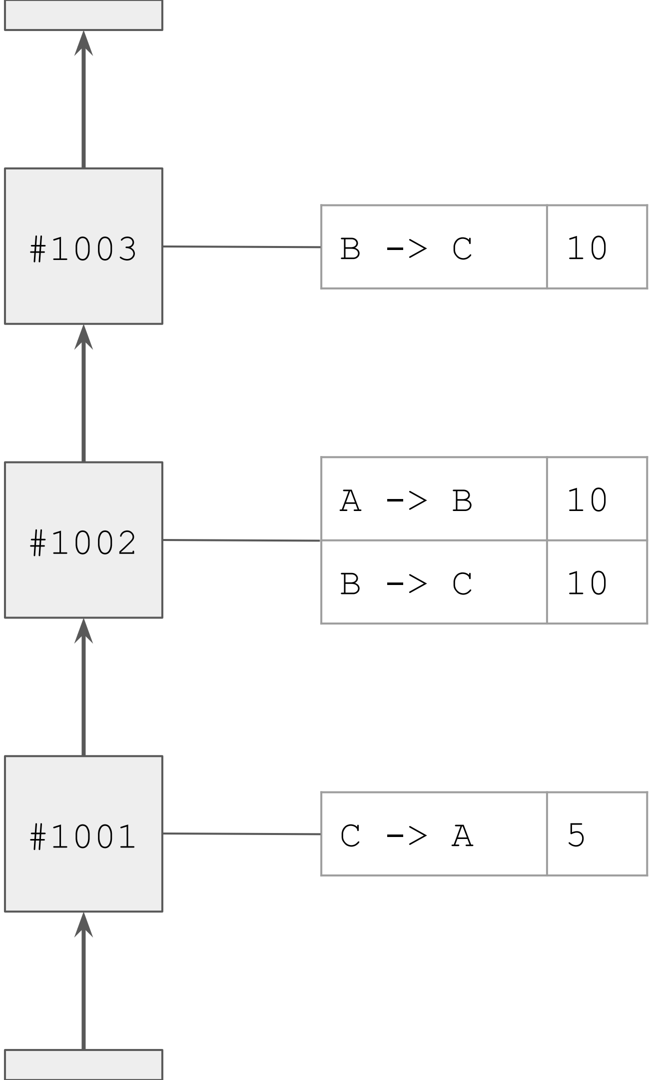
Postgres & PostQuel



1996 PostgreSQL

Why this talk ?

LEDGER



A	1001	+5
A	1002	-10
B	1001	+10
B	1002	+10
C	1001	-10
C	1002	-10

+

cassandra

❌ Optimized for write-based workflow

cassandra

- ❌ Optimized for write-based workflow
- ❌ Heavy in term of maintenance

cassandra

- ❌ Optimized for write-based workflow
- ❌ Heavy in term of maintenance
- ❌ Joins must be performed on the application side

Applicative joins ?

```scala
class OrderService:

  val orderRepository: OrderRepository       = ???
  val customerRepository: CustomerRepository = ???

  def findLatestOrdersWithCustomer(): Seq[OrderWithCustomer] =
    val latests = orderRepository.findLatestOrders()

    val result = ListBuffer.empty[OrderWithCustomer]

    latests.foreach: order =>
      val customer = customerRepository.findById(order.curstomerID)
      result.addOne(OrderWithCustomer(order, customer))

    result.toList
```

✅ Great for our read-heavy workflow

✅ Lighter maintenance and cheaper (50%)

✅ Joins can just be joins !

What is first class Postgres ?

- Database is at the center of the system
- The schema is the source of truth
- Treat SQL as a *real programming language*
- Commit to Postgres features

Setup your development loop

Tools for composition

Step up your indexing game

Data types shenanigans

Setup your development loop

Tools for composition

Step up your indexing game

Data types shenanigans

Hardware Setup

- Modern hardware is worth it
- Scaling vertically is a viable strategy
- On-prem kube (rancher) + bare metal PG

Dual Intel Xeon Gold 5515+ - 16c/32t - 3.2GHz/3.6GHz

256GB DDR5 ECC 4800MHz

2x SSD NVMe 960GB Datacenter Class Soft RAID

6× 3.84TB SSD NVMe Soft RAID

- Modern hardware is worth it
- Scaling vertically is a viable strategy
- On-prem kube (rancher) + bare metal PG

Dual Intel Xeon Gold 5515+ - 16c/32t - 3.2GHz/3.6GHz

256GB DDR5 ECC 4800MHz

2x SSD NVMe 960GB Datacenter Class Soft RAID

6× 3.84TB SSD NVMe Soft RAID

~ 400 €

Software Setup

Use a migration tool (we use flyway)

Prefer idempotent migrations

Think about scheduling baselines

Abuse comments on everything !

Abuse comments on everything !

```sql
-- Transaction => block join table, a.k.a. confirmation (inclusion of a transaction in a block)
create table v2.chain_tx_block
(
   block_hash text not null,
   tx_id      text not null,

   unique (tx_id),
   primary key (block_hash, tx_id),
   foreign key (tx_id)      references v2.chain_tx (tx_id),
   foreign key (block_hash) references v2.chain_block (block_hash) on delete cascade
);
comment on table v2.chain_tx_block is 'represents the inclusion of a transaction in a block (a.k.a., a
confirmation) - note that it can be a minority fork block';
comment on column v2.chain_tx_block.tx_id is 'transaction identifier (immutable, globally unique identifier,
typically transaction hash)';
comment on column v2.chain_tx_block.block_hash is 'block identifier (immutable, globally unique)';
```

No ORM : prefer data access libraries

```
skunk.exception.PostgresErrorException:
🔥
🔥  Postgres ERROR 42804 raised in transformAssignedExpr (parse_target.c:595)
🔥
🔥    Problem: Column "success" is of type boolean but expression is of type bigint.
🔥       Hint: You will need to rewrite or cast the expression.
🔥
🔥  The statement under consideration was defined
🔥    at /Users/raphael.lemaitre/Developer/projects/ledger/atlas/modules/polkadot/src/main/scala/polkadot/queries/ExtrinsicTable.scala:18
🔥
🔥    INSERT INTO extrinsics
🔥                SELECT height                                      AS height,
🔥                       index                                       AS index,
🔥                       data ->> 'hash'                             AS hash,
🔥                       data -> 'method' ->> 'pallet'               AS pallet,
🔥                       data -> 'method' ->> 'method'               AS method,
🔥                       (data ->> 'nonce') :: bigint                AS nonce,
🔥                       data -> 'signature' -> 'signer' ->> 'id'    AS signer,
🔥                       data -> 'args'                              AS args,
🔥                       (data ->> 'tip') ::bigint                   AS tip,
🔥                       (data -> 'info' ->> 'weight') :: bigint     AS weight,
🔥                       data -> 'info' ->> 'class'                  AS class,
🔥                       (data -> 'info' ->> 'partialFee') :: bigint AS partial_fee,
🔥                       data -> 'era'                               AS era,
🔥                       (data -> 'success') :: bigint              AS success,
🔥                        └─── Column "success" is of type boolean but expression is of type bigint.
🔥                       (data -> 'paysFee') :: boolean              AS pays_fee
🔥                FROM raw_extrinsics
🔥                WHERE height >= $1 AND height <= $2
🔥
🔥  If this is an error you wish to trap and handle in your application, you can do
🔥  so with a SqlState extractor. For example:
🔥
🔥    doSomething.recoverWith { case SqlState.DatatypeMismatch(ex) =>  ...}
🔥
```

Use a proper SQL editor (DataGrip, DBeaver)

`psql` can be a very powerful tool

`psql` tips :

- `\e` Open the EDITOR with the latest query you typed. Closing run the buffer.
- `\e myfile.sql` Open the EDITOR with the file in it.
- `\x` For extended display
- `\watch 1` Repeat the query every 1 second

Testing SQL

Testing with a real DB

- Testcontainers + fixtures to add migration
- Cache DB instances !
- Never use sqlite / h2 / derby anymore

Crafting SQL Tests

- Test on the boundaries.
  - Typed language -> Input / Output models then translate into domain
  - Dynamic languages -> check aggressively
  - Defense in depth strategies

- Test functions and procedures

- Make tooling around sql files (syntax highlighting)

Testing on real data

Use transactions to benchmark your migrations before running them with migrations

```
begin isolation level read committed;  -- Queries sees data before it starts
begin isolation level repeatable read; -- Queries sees data before the tx started
begin isolation level serializable;    -- repeatable read + error if write conflict

-- Migrate, migrate, migrate

savepoint before_risky_thing;

-- Migrate, migrate, migrate

-- Oh no ! I deleted the prod table :/

rollback to before_risky_thing; -- Rollback to savepoint, restoring the state

-- Migrate, migrate, migrate

commit; -- or abort if you want your migrations to pick it up
```

Optimizing

```
EXPLAIN (ANALYZE, COSTS, VERBOSE, BUFFERS)
```

```
explain (analyse, verbose, costs, buffers)
select header_id, tx_index, address
from log
     join address on contract_id = address_id
where (header_id, tx_index) = (2278166300, 290);

                                              QUERY PLAN

----------------------------------------------------------------------------------------------------------
 Nested Loop  (cost=[...] rows=10 width=33) (actual time=0[...] rows=5 loops=1)
   Output: log.header_id, log.tx_index, address.address
   Inner Unique: true
   Buffers: shared hit=30
   ->  Index Scan using log_pkey on v2.log  (cost=[...]) (actual time=[...] rows=5 loops=1)
      Output: [columns]
      Index Cond: ((log.header_id = '2278166300'::bigint) AND (log.tx_index = 290))
      Buffers: shared hit=5
   ->  Index Scan using address_pkey on v2.address  (cost=[...]) (actual time=[...] rows=1 loops=5)
      Output: address.address_id, address.address
      Index Cond: (address.address_id = log.contract_id)
      Buffers: shared hit=25
 Query Identifier: -8177880399755634493
 Planning:
   Buffers: shared hit=10
 Planning Time: 0.467 ms
 Execution Time: 0.094 ms
(17 rows)
```

Always `analyse` after large inserts

`Scans` are the name of the game

- `Seq Scan` Traverse in order : maybe an index is missing ?
- `Index Scan` Scans a table heap using an index
- `Index Only Scan` Only touches the index pages
- `Bitmap Index Scan` Build a bitmap from an index
- `Bitmap Heap Scan` Build a bitmap from the heap
- `TID Scan` Directly access physical data

`Sort` is a very costly operation

- `union, distinct` introduce a sort
- Cover the order with the indices (and preserve order through the query)
- Sorting large set can spill on disk : `set work_mem` can help

`Shared Hit`	data is coming from cached pages
`Read`	data is coming from disk

`Rows Removed by Filter` means your indices are not efficient enough explain.dalibo.com

Setup your development loop

Tools for composition

Step up your indexing game

Data types shenanigans

```sql
select first_name
     , last_name
     , performance_rating
     , project_count
from (select employee_id
           , first_name
           , last_name
           , performance_rating
      from employees
      where performance_rating >= 4.5) as employee_performance
     join (select employee_id
                , count(project_id) as project_count
           from employee_projects
           group by employee_id) as project_allocations
      using (employee_id)
where pa.project_count > 1
order by performance_rating desc
       , project_count desc
;
```

Oh no ! Subselects everywhere

- Each clause that require a *set of record*
  - can be replaced by a subselect

- Each clause that require a *value*
  - can be replaced by a subselect that yields only ONE element

Common Table Expression

```sql
with employee_performance as
       (select employee_id, first_name, last_name, performance_rating
        from employees
        where performance_rating >= 4.5 )

   , project_allocations as
       (select employee_id, count(project_id) as project_count
        from employee_projects
        group by employee_id )

   , high_performers_multiple_projects as
       (select first_name, last_name, performance_rating, project_count
        from employee_performance ep
        join project_allocations pa using (employee_id)
        where project_count > 1)

select *
from high_performers_multiple_projects
order by performance_rating desc, project_count desc;
```

Common Table Expression

- Extract logical subqueries from the main one
- Can even be recursive (but out of the scope of this talk)
- Warning : They can introduce memoization
  - `materialized`/`not materialized` prefixes

Views

```sql
create view employee_performance as
     select employee_id, first_name, last_name, performance_rating
     from employees
     where performance_rating ⩾ 4.5;

create view project_allocations as
     select employee_id, count(project_id) as project_count
     from employee_projects
     group by employee_id;

select first_name
     , last_name
     , performance_rating
     , project_count
from employee_performance
     join project_allocations using (employee_id)
where project_counts > 1
order by (performance_rating, project_count) desc
;
```

Views

- Extract logical subqueries into reusable piece of code
- Allow neat tricks for schema upgrade !
- Warning : Documentation

Remember ?

```
select first_name
     , last_name
     , performance_rating
     , project_count
from (select employee_id
           , first_name
           , last_name
           , performance_rating
      from employees
      where performance_rating ≥ 4.5) as employee_performance
     join (select employee_id
                , count(project_id) as project_count
           from employee_projects
           group by employee_id) as project_allocations
       using (employee_id)
where pa.project_count > 1
order by performance_rating desc
       , project_count desc
;
```

Remember ?

```
select first_name
     , last_name
     , performance_rating
     , (select count(project_id)
        from employee_projects ep
        where employee_id = ep.employee_id) as project_count

from employees

where performance_rating >= 4.5
  and project_count > 1

order by performance_rating desc
       , project_count desc
;
```

Functions

```sql
create function get_project_count( employee_id int )
returns bigint as $$
begin return
     select count(project_id)
     from employee_projects ep
     where employee_id = ep.employee_id;
end;
$$ language plpgsql;
```

Functions

```sql
select first_name
     , last_name
     , performance_rating
     , get_project_count(employee_id) as project_count

from employees

where performance_rating >= 4.5
  and project_count > 1

order by ( performance_rating, project_count ) desc
;
```

Functions

- In a functional language, using functions is a good idea
- As testable as any SQL query
- Composes way better than fragments

Set returning functions

```sql
create function project_stats( employee_id int )
returns table ( rd_project_count      bigint
              , it_project_count      bigint
              , pi_project_count      bigint
              , mean_project_duration interval) as $$
begin return query
     select count(project_id) filter (where type = 'R&D') as rd_project_count
          , count(project_id) filter (where type = 'IT')  as it_project_count
          , count(project_id) filter (where type = 'PI')  as pi_project_count
          , avg(age(start_at, end_at)) as mean_project_duration
     from employee_projects ep
     where employee_id = ep.employee_id;
end;
$$ language plpgsql;
```

SRF + Lateral

```
select first_name
     , last_name
     , performance_rating
     , rd_project_count
     , it_project_count
     , pi_project_count
     , mean_project_duration

from employees
     join lateral project_allocations(employee_id) on true

where performance_rating ⩾ 4.5
  and project_count > 1

order by ( performance_rating, project_count ) desc
;
```

SRF + Lateral

```
select first_name
     , last_name
     , performance_rating
     , rd_project_count
     , it_project_count
     , pi_project_count
     , mean_project_duration

from employees
   , lateral project_allocations(employee_id)

where performance_rating ⩾ 4.5
  and project_count > 1

order by ( performance_rating, project_count ) desc
;
```

Setup your development loop

Tools for composition

Step up your indexing game

Data types shenanigans

Indices 101

Persistent data structure to speed up data access:

- `btree` default index type.
  - Allows range lookup and sorting
- `gin` inverted index.
  - Allows multi-key lookup but no sorting.
- `gist` framework to create complex datatype indices.
  - Allows distance based lookup and sorting

Indices 101

Stored in pages like tables : hence `tablespace`

```
create index ...
tablespace nvme_drive;
```

Only one index can be used to scan in order

```sql
create table action (
    id      bigserial   primary key,
    user_id uuid        not null,
    ts      timestamptz not null,
    asset   text        not null,
    status  text        not null
);
```

Multicolumns indices

```
create index on action (user_id, ts desc)
```

Order of the columns is important !

- ✅ `user_id = ???`
- ✅ `user_id = ??? and ts <= ???`
- ❌ `ts between ??? and ???`

Multicolumns indices

```
create index on action (user_id, ts desc)
```

Order of the columns is important !

- ✅ `order by user_id`
- ✅ `order by user_id, ts desc`
- ❌ `order by ts`
- ❌ `order by ts desc, user_id`

Included fields in index

```
create index on action (user_id, ts desc)
include (asset)
```

- Can make a query skip heap storage
- `IndexOnlyScan` vs `IndexScan`
- Useful when joining partial data

Partial Index

```
create index on action (user_id, ts desc)
include (asset)
where status = 'canceled'
```

- Reduce index size (and potentially increase performance)
- Can be combined with `unique` to enforce uniqueness on a subsets of rows

Clustered Tables

```
cluster table_name using idx_action_user_id_ts;
```

- Physically **reorders table rows** based on an **index**.
- Faster index-based scans (esp. range queries).

Clustered Tables

```
cluster table_name using idx_action_user_id_ts;
```

- **Table locked** during clustering (exclusive access).
- Needs **manual re-cluster**
- Can increase **write overhead** if frequently updated.

Setup your development loop

Tools for composition

Step up your indexing game

Data types shenanigans

Using domains

```
create domain invoice_total_amount as numeric(20,2)
check (
    value > 0 and value <= 1000000
)
constraint invoice_total_amount_range_check;
```

Using domains

```
create table invoice (
    id          bigserial            primary key,
    customer_id bigint               not null,
    issued_at   date                 not null,
    total       invoice_total_amount not null
);
```

Using complex data types : ranges & multiranges

- ranges are continuous intervals (e.g. dates, numbers)

- multiranges are sets of non-overlapping ranges

- Lots of operator / functions that handle all the tricky use cases
  - overlap
  - open / closed bounds

- Indexing support using gist and `btree_gist`

Using complex data types : ranges & multiranges

```
create table subscription (
    id            bigserial primary key,
    user_id       bigint    not null,
    active_period daterange not null
);
```

Using complex data types : ranges & multiranges

```
create index on subscription using gist (active_period);
```

Using complex data types : ranges & multiranges

```
select * from subscription
where active_period @> date '2025-06-01';

select * from subscription
where active_period -|- daterange('2025-06-01', '2025-07-01');
```

Using complex data types : exclude constraints

```
create table room_booking (
    id            bigserial primary key,
    room_id       bigint    not null,
    booked_period tsrange   not null,

    exclude using gist (
        room_id       with =,
        booked_period with &&
    )
);
```

Using complex data types : arrays

- collection of elements
- overhead for small collection is huge (24 bytes)
- Indexable using `GIN` (warning: no sorting!)
    - `@>` (inclusion)
    - `&&` (overlap)
- For key-value, use `hstore`

Using complex data types : arrays and aggregations

```
with exploded(post_id, tag) as
     (select post_id, unnest(tags) from posts)

select post_id
     , array_agg(tag order by tag)
       filter (where tag <> 'draft')
       as cleaned
from exploded
group by post_id;
```

Using complex data types : ltree

- ltree stores hierarchical labels (path in a tree)
- lquery : pattern-matching syntax for querying ltrees
- indexable (GiST) on ltree and ltree[]

Using complex data types : ltree

```sql
create extension if not exists ltree;

create table product_category (
    id   serial primary key,
    name text  not null,
    path ltree not null
);

create index idx_product_path_gist
    on product_category
    using gist (path);
```

Using complex data types : ltree

```sql
-- 1. in Electronics
-- 2. Exactly 2-3 levels are left free
-- 3. Last label must be LED, OLED, or Speakers
-- 4. Exclude any category whose path contains Refurbished anywhere

select id, name, path
from   product_category
where  path ~ 'electronics.*{2,3}.{led|oled|speakers} & !*.refurbished';
```

`JSONB` : If your data has a schema : don't.

Using complex data types : jsonb

- Joins are fast.

- TOAST (The Oversized-Attribute Storage Technique)
    - Compressing and storing large field values (like long text or bytea) out-of-line in a separate pages to preserve row byte length

    - Reads are slower

Using complex data types : jsonb

- base `GIN`
  - `data @> '{"status": "active"}'   -- contains`
  - `data ? 'email'                   -- exsits`
- path `GIN (data jsonb_path_ops)`
  - only for `@>`
- btree on extracted values
  - `create index idx_data_price on my_table ((data ->> 'price'))`

`JSONB` : Great for aggregations

Using complex data types : jsonb aggregations

```sql
create table transactions (
     transaction_id   uuid        primary key default gen_random_uuid(),
     account_id       uuid        not null references accounts(account_id),
     transaction_date timestamptz not null
);

create table transaction_assets (
     detail_id        uuid    primary key default gen_random_uuid(),
     transaction_id   uuid    not null references transactions(transaction_id),
     asset_name       text    not null,
     amount           numeric not null
);
```

Using complex data types : jsonb

```
create function get_transaction_transfers(transaction_id_input uuid)
returns jsonb
language plpgsql as $$
begin
     return select jsonb_agg(to_jsonb(*) - 'transaction_id')
            from transaction_assets
            where transaction_id = transaction_id_input;
end;
$$;
```

Using complex data types : jsonb

```
select transaction_id
     , transaction_date
     , get_transaction_transfers(transaction_id) as transfers
from transactions
where account_id = '<account_id>';
```

Using complex data types : cube

Builtin extension to manipulate *hyper-rectangles* and *points*

- spatial indexing
- Effectively multivariate order-independant index
- range-based searching.

Using complex data types : the cube's trick

```
create table customer_profile (
    customer_id serial primary key,
    logins integer not null,
    session_duration numeric(5,2) not null,
    purchases integer not null
);
```

Using complex data types : the cube's trick

```
create index customer_behavior_idx
on customer_profile
using gist (
    cube(array[
        logins::float8,
        session_duration::float8,
        purchases::float8
    ])
);
```

Using complex data types : the cube's trick

```
select *
from customer_profile
where
     cube(array[ logins::float8, session_duration::float8,
purchases::float8])
  <@ cube(array[10,20,1],array[50,40,10]);
```

Using complex data types : the cube's trick

```
                                              QUERY PLAN
----------------------------------------------------------------------------------------------
 Bitmap Heap Scan on customer_profile  (cost=1.26..3.42 rows=2 width=24)
   Recheck Cond: (cube(ARRAY[...]) <@ '(10, 20, 1),(50, 40, 10)'::cube)
   ->  Bitmap Index Scan on customer_behavior_idx  (cost=0.00..1.26 rows=2 width=0)
         Index Cond: (cube(ARRAY[...]) <@ '(10, 20, 1),(50, 40, 10)'::cube)
```

What is first class Postgres ?

- Database is at the center of the system
- The schema is the source of truth
- Treat SQL as real code
- Commit to Postgres features

This meeting could have been an email.

This meeting could have been an email.

This service could have been a table.