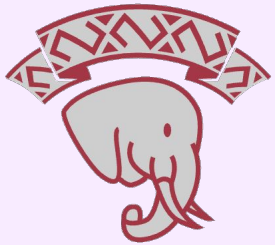
WELCOME

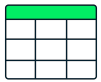
# Normalize or Denormalize?

## Relational SQL Columns or JSON Document Attributes?

Franck Pachot
Developer Advocate, MongoDB



Documents,



Normalized Tables, or

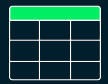
Documents in Tables?

# Document

```
CREATE TABLE blog_posts (
  id   UUID primary key,
  data JSONB -- title, content,
             -- tags, categories…
);
```

# Docs. in table

```
CREATE TABLE blog_posts (
  id       UUID primary key bigserial,
  title    TEXT,
  content  TEXT,
  metadata JSONB -- tags, categories…
);
```

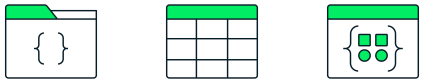


# Tables

```
CREATE TABLE blog_posts (
  id       UUID primary key bigserial,
  title    TEXT,
  content  TEXT
);

CREATE TABLE blog_tags (
  blog_id  UUID references blog_posts,
  num      INT,
  tag      TEXT,
  primary key (id, num)
);

CREATE TABLE blog_categ… -- Many-to-Many
CREATE TABLE categories… -- Lookup table
```



# How do you build your applications?

**Do you know the applications that will access the database?**

🔵 No (database schema first):
- normalize (and add indexes later)

🔴 Yes (application objects first):
- design for the business domain objects
- known: read/write patterns, cardinalities

**Is your team more comfortable with**
- SQL (native SQL, ORM)?
- JSON, Document API?

```
select i as item -- SQL vs SQL/JSON
,( '{1,2,3}'::int[] ) [i] as "SQL array"
,( '[1,2,3]'::json  )->i  as "JSON array"
from generate_series(0,3) i                ;

 item | in SQL array | in JSON array
------+--------------+--------------
    0 |       (null) | 1
    1 |            1 | 2
    2 |            2 | 3
    3 |            3 | (null)
```

# How new is JSON in SQL databases?

**Postgres original concept [ston86]:**

- complex datatypes
- custom objects
- and later TOAST, GIN, JSON, JSONB, SQL/JSON

**Past 25 years of RDBMS:**

- clustered tables
- BLOB
- Arrays
- Nested Tables
- Object-Relational
- XML in the database
- JSON in the database
- embeddings (vectors)

**None were used alone, but as an addition to the relational tables**

# “ JSON is Schemaless

**There is nothing like unstructured data (or it doesn't deserve storing it)**

- application must know the schema
- index keys are on schema's fields

**Polymorphism**

- each document declare its own schema
- some part can be enforced for a collection/table with schema validation or integrity constraints, unique indexes

**Flexible: don't need to declare the schema upfront**

- no need to switch from the application IDE to the database and run DDL

# Tables, SQL, Relations, and Normal forms

# Normal Forms

**Relational algebra, normal forms** are important to understand data modeling concepts

**What matters is the problem it solves:**
- avoid update anomalies
- understand your data
- evolve your data structures
- get good predictable performance

**And the developer experience**

## Normalize?

**Data used together stored in multiple tables:**

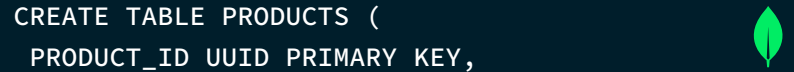
- cannot have one index for both
- must lock at multiple places

**Some data must be duplicated:**

- by business requirement:
  - price used by the order without ref. to catalog
  - total amount for performance (summary)

**Different lifecycle of data**

- shopping cart often deleted may fragment the permanent orders table and indexes



```
CREATE TABLE PRODUCTS (
 PRODUCT_ID UUID PRIMARY KEY,
 PRODUCT_NAME TEXT NOT NULL,
 PRICE NUMERIC
);

CREATE TABLE CUSTOMERS (
 CUSTOMER_ID BIGSERIAL PRIMARY KEY,
 CUSTOMER_NAME TEXT NOT NULL
);

CREATE TABLE ORDERS (
 ORDER_ID UUID PRIMARY KEY,
 CUSTOMER_ID BIGINT NOT NULL REFERENCES CUSTOMERS,
 ORDER_DATE TIMESTAMPTZ, -- null if shopping cart
 TOTAL_AMOUNT NUMERIC
);

CREATE TABLE ORDER_LINES (
 PRIMARY KEY (ORDER_ID, LINE_NO),
 ORDER_ID UUID,
 LINE_NO INT,
 QUANTITY INT,
 PRICE NUMERIC,
 PRODUCT_ID UUID REFERENCES PRODUCTS
);
```

# “ No Foreign Keys in JSONB

**Relational model needs foreign key to guarantee referential integrity**

- on delete error, cascade, set null
- on insert check and lock parent
- on update cascade

**But is that so simple?**

- delete cascade often not efficient (row-by-row)
- DB locks just in case (delete parent)
- the business logic may be more complex than simply cascading updates

**Depends on the types of relationships**

# Type of Associations

◆— **“part-of” Composition**

- Exclusive ownership, shared lifecycle

*Example: A blog post owns its comments*

SQL: FK on delete cascade / JSON: embedded

◇— **“has-a” Aggregation**

- Shared ownership, independent life cycles

*Example: A blog post has categories*

SQL: FK on delete restrict / JSON: reference or embed

— **“use-a” Association**

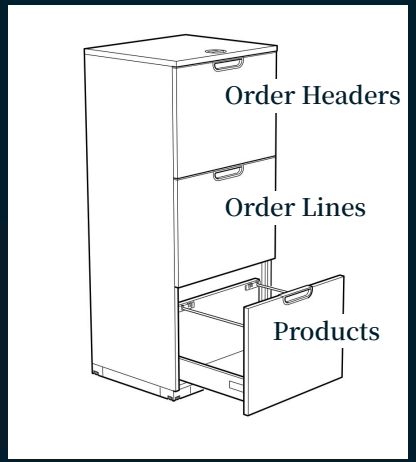
- No ownership, independent life cycles

*Example: Authors use blog posts to publish their content*

SQL: FK on delete set null / JSON: reference

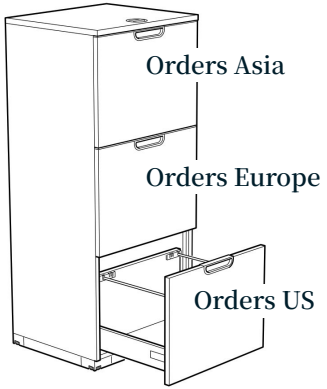
Relational

# Designed for data entities




Document

# Designed for business objects




# SQL Interactive Transactions

```
BEGIN;
 INSERT INTO orders (order_id, customer_id, country)
   VALUES (42, 100010, 'CH');
 INSERT INTO orderlines (order_id, line_id, product_id, quantity)
   VALUES (42, 1, 'X3', 3);
 INSERT INTO orderlines (order_id, line_id, product_id, quantity)
   VALUES (42, 2, 'Y4', 8);
COMMIT;
```

- Many round trip if not a stored procedure (or a PL/SQL block)
- The database doesn't know what happens until commit (cache, two-phase locking)
- N+1 or eager queries to fetch one entity

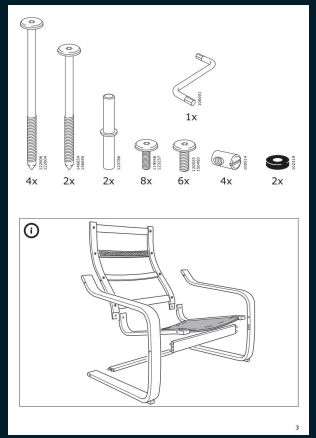
# Single Document API call to the DB

```
db.orders.insertOne({
  order_id: 42,
  customer_id: 100010,
  country: 'CH',
  orderlines: [
    { line_id: 1, product_id: 'X3', quantity: 3 },
    { line_id: 2, product_id: 'Y4', quantity: 8 },
  ],
})
```

- One atomic call with all business transaction data
- DB can optimize and isolate the transaction (with no lock)

Relational

# Normalized by Data Entities




Document

# Designed by Business Objects

# Documents for Domain Driven Design

Modern development

**One database per bounded context:**
- no need to normalize
- DB schema fits application objects
- a document fits business transactions

**More databases but simple schemas**
- no need for ORM / complex queries
- decouple development devops teams
- all logic in application, CI/CD for validation
- Secondary indexes for more use-cases
- change data capture to secondary datastores

https://scabl.blogspot.com/p/advancing-enterprise-ddd.html

# Document model per domain

## Online order entries

```
// customer (1-*) orders (*-*) products
{
  customer_email: ...,
  customer_name: ...,
  shopping_cart: [
   { product: ..., quantity: ...},
   ...
  ],
  recent_orders: [
   { date: ..., product: ..., ...},
   ...
  ]
}
```

## Product sales analysis

```
// products (*-*) customer
{
  day: ...,
  products: [
   { date: ..., product: ...,
     types_of_customers: [
       { type: ..., quantity: ... },
       ...
      ],
   },
   ...
  ]
}
```

# Indexing on sub-documents (JSON array)

**GIN indexes:**
- multi-key (one entry per array item)
- but bitmap (no index only scan, no range / sort, only JSON operations)

**Expression indexes:**
- can index a path but not through arrays (or it needs one index per item)

**Multiple indexes:**
- can be combined with bitmap scan but does not preserve order for sort

**btree-gin extension:**
- composite GIN + expression but same limitations

# Order with details (JSONB)

```
create table orders(
 primary key(id)
 , id bigserial
 , country_id int
 , created_at timestamptz default clock_timestamp()
 , details jsonb
);
```

# Equality + Sort query (pagination)

```
create table orders(
 primary key(id)
 , id bigserial
 , country_id int
 , created_at timestamptz default clock_timestamp()
 , details jsonb
);

SELECT    country_id, details
 FROM     orders
 WHERE    country_id = 1
 AND      details @> '[{"product_id": 15}]'
 ORDER BY created_at DESC
 LIMIT    10
;
```

# B-Tree index

```
CREATE INDEX orders1 ON orders ("country_id", "created_at" desc)
;
```

```
                                        QUERY PLAN
-----------------------------------------------------------------------------------
 Limit (actual time=0.036..0.117 rows=10 loops=1)
  Buffers: shared hit=32
   ->  Index Scan using orders1 on orders (actual rows=10 loops=1)
         Index Cond: (country_id = 1)
         Filter: (details @> '[{"product_id": 15}]'::jsonb)
         Rows Removed by Filter: 38
         Buffers: shared hit=32
```

# B-Tree index + GIN indexes

```
CREATE INDEX orders1 ON orders ("country_id", "created_at" desc)
;

-- Only for scalar:

CREATE INDEX orders0 ON orders ( (("details"->>'product_id')::int) )
;

-- For paths with array

CREATE INDEX orders2 ON orders using GIN ( details )
;
```

# Used for equality, but requires additional Sort

```
->  Sort (actual time=222.683..226.142 rows=14955 loops=3)
 Sort Key: created_at DESC
 Buffers: shared hit=158 read=61738, temp read=4584 written=4592
 ->  Parallel Bitmap Heap Scan on orders (actual rows=14955 loops=3)
   Recheck Cond:
   ((country_id = 1) AND (details @> '[{"product_id": 15}]'::jsonb))
   Rows Removed by Index Recheck: 96641
   Heap Blocks: exact=9701 lossy=11292
   Buffers: shared hit=145 read=61733
   ->  BitmapAnd (actual time=78.189..78.190 rows=0 loops=1)
   Buffers: shared hit=145 read=613
   ->  Bitmap Index Scan on orders1 (actual rows=100362 loops=1)
       Index Cond: (country_id = 1)
       Buffers: shared read=387
    ->  Bitmap Index Scan on orders2 (actual rows=499460 loops=1)
       Index Cond: (details @> '[{"product_id": 15}]'::jsonb)
       Buffers: shared hit=145 read=226
```

# B-Tree + GIN index (with extension)

```
CREATE EXTENSION BTREE_GIN
;


CREATE INDEX orders3 ON orders
           using GIN (country_id , details, created_at)
;

SELECT    country_id, details
 FROM     orders
 WHERE    country_id = 1
 AND      details @> '[{"product_id": 15}]'
 ORDER BY created_at DESC
 LIMIT    10
;
```

# Equality but requires additional sort

```
->  Sort (actual time=109.979..113.574 rows=14955 loops=3)
  Sort Key: created_at DESC
  Sort Method: external merge  Disk: 12456kB
  Buffers: shared hit=237 read=40117, temp read=4585 written=4594
  Worker 0:  Sort Method: external merge  Disk: 11720kB
  Worker 1:  Sort Method: external merge  Disk: 12504kB
  ->  Parallel Bitmap Heap Scan on orders (actual rows=14955 loops=3)
   Recheck Cond:
   ((country_id = 1) AND (details @> '[{"product_id": 15}]'::jsonb))
   Rows Removed by Index Recheck: 1760
   Heap Blocks: exact=13486
   Buffers: shared hit=226 read=40110
   ->  Bitmap Index Scan on orders3 (actual rows=50144 loops=1)
    Index Cond:
    ((country_id = 1) AND (details @> '[{"product_id": 15}]'::jsonb))
    Buffers: shared hit=226 read=251
```

# Think about indexes and TOASTing

**JSON columns:**
good for flexible schema data that is
- inserted as a whole
- queried together
- not too large (fits in a tuple)
- doesn't need range or sort index on fields with an array in the json path

**Doesn't solve the indexing of One-to-Many:**
- avoids a join but needs to combine bitmaps, and recheck conditions
- large documents are stored in TOAST chunks with an additional index

> *One TOAST fits all*
> (Oleg Bartunov)

**PostgresPro talks:**

- Roasted Toasted Json: https://www.pgconf.in/conferences/pgconfin2024/program/proposals/629
- One TOAST fits all: http://www.sai.msu.su/~megera/postgres/talks/toast-pgcon-2022.pdf

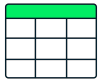
Pluggable Toaster:

- 2023-03 (2023-03-01 – 2023-04-08): Rejected
- 2023-01 (2023-01-01 – 2023-01-31): Moved to different CF
- 2022-11 (2022-11-01 – 2022-11-30): Moved to different CF
- 2022-09 (2022-09-01 – 2022-09-30): Moved to different CF
- 2022-07 (2022-07-01 – 2022-07-31): Moved to different CF
- 2022-03 (2022-03-01 – 2022-03-31): Moved to different CF
- 2022-01 (2022-01-01 – 2022-01-31): Moved to different CF

# Documents, Tables, or Documents in Tables?

### Documents (JSON only)

Application-centric, optimized for the domain access patterns, but different than document databases in terms of index and data locality

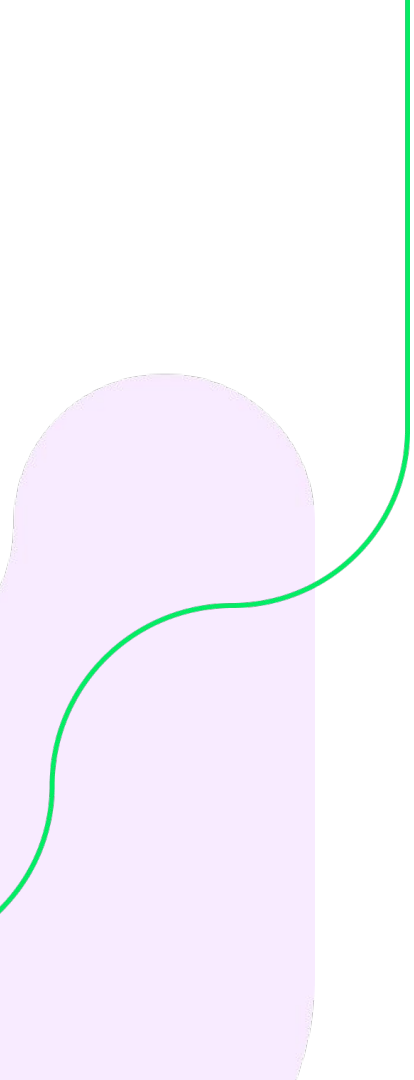


### Relational tables

Data-centric schema, normalized central database designed before the applications

### Tables with JSONB

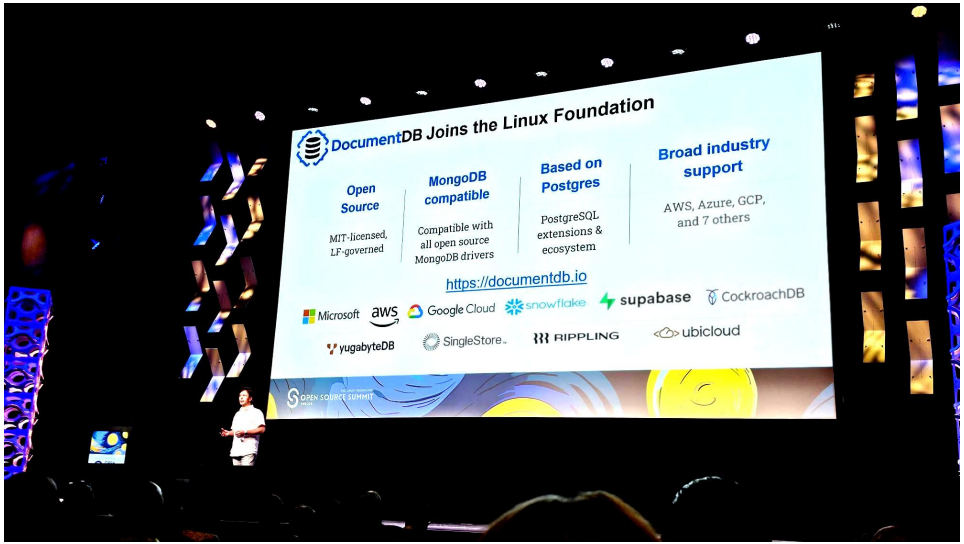
Great to add some small schema-on-read data to a normalized model, or store larger documents as a whole



# Which API?

- PostgreSQL JSON operators
- SQL/JSON in PostgreSQL
- DocumentDB extension

# DocumentDB extension




- RUM indexes, BSON type
- MongoDB emulation on top
- Open Source, now part of Linux Foundation
- Used in CosmosDB vCore
- May be used in the future for Amazon DocumentDB

Blog:
mdb.link/documentdb-xplan

# db.demo.createIndex({ "a": 1 , ts: -1 }) ;

```
postgres=# \d+ documentdb_data.documents_15*

                    Table "documentdb_data.documents_15"
     Column      |           Type           || Storage  |
-----------------+--------------------------++----------+
 shard_key_value | bigint                   || plain    |
 object_id       | documentdb_core.bson     || extended |
 document        | documentdb_core.bson     || extended |
 creation_time   | timestamp with time zone || plain    |
Indexes:
    "collection_pk_15" PRIMARY KEY, btree (shard_key_value, object_id)
    "documents_rum_index_35" documentdb_rum (
document documentdb_api_catalog.bson_rum_single_path_ops (path=a, tl='2691'),
document documentdb_api_catalog.bson_rum_single_path_ops (path=ts, tl='2691'))
Check constraints:
    "shard_key_value_check" CHECK (shard_key_value = '15'::bigint)
Access method: heap
```

# db.demo.find( { a: 1 } ).sort({ts:-1}).limit(10)

```
Limit  (actual rows=10 loops=1)
 Buffers: shared hit=185
 ->  Sort  (actual rows=10 loops=1)
   Sort Key: (documentdb_api_catalog.bson_orderby(document,
'BSONHEX0d0000001074730ffffffff00'::documentdb_core.bson)) DESC NULLS LAST
Sort Method: top-N heapsort  Memory: 27kB
   Buffers: shared hit=185
   ->  Bitmap Heap Scan on documents_15 collection  (actual rows=10014 loops=1)
    Recheck Cond: (document OPERATOR(documentdb_api_catalog.@=)
'BSONHEX0c0000001061000100000000'::documentdb_core.bson)
    Heap Blocks: exact=169
    Buffers: shared hit=177
    ->  Bitmap Index Scan on "a_1_ts_-1"  (actual rows=10014 loops=1)
     Index Cond: (document OPERATOR(documentdb_api_catalog.@=)
'BSONHEX0c0000001061000100000000'::documentdb_core.bson)
     Buffers: shared hit=8
```

# Multi-key Indexing

Equality, Sort, Range indexes on JSON paths with arrays (One-to-Many sub-documents)

- ✅ MongoDB
- ❌ Amazon DocumentDB
- ❌ Microsoft CosmosDB
- ❌ PostgreSQL with GIN on JSONB
- ❌ PostgreSQL with RUM and DocumentDB
- ❌ Oracle MongoDB API

Blog series on MongoDB Multi-key Indexes:
https://dev.to/franckpachot/series/31244

```
db.orders.createIndex({
  "country_id": 1,
  "order_details.product_id": 1,
  "created_at": -1
});
```

```
db.orders.find( {
 country_id: 1,
 order_details: {
  $elemMatch: { product_id: 15 }
 }
}).sort(       { created_at: -1 }
).limit(10);
```


# Best advice: understand how it works

**Indexing possibilities**
Multi-key range/sort

**Data locality**
MongoDB keeps documents together
PostgreSQL may split them (TOAST)

**Velocity of development**
Start an application without declaring anything,
vs. running DDL on the DB before DML from the application

**For unknown access patterns**
Use SQL and normalized relational schema

**Transactional needs**
MongoDB is ACID optimistic locking, no wait but retry logic
PostgreSQL has explicit locks (e.g FOR UPDATE SKIP LOCKED)

# Thank You For Your Time

## Let's continue the discussion

https://www.linkedin.com/in/franckpachot
blog: https://dev.to/franckpachot
https://x.com/FranckPachot

Franck Pachot
Developer Advocate