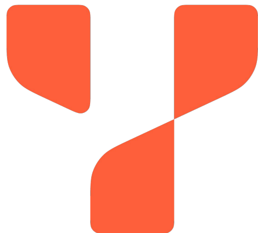
DocumentDB on Postgres

Native BSON, Vectors, and Mongo Compatibility

Hari Krishna Sunder
Architect
YugabyteDB

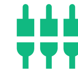


Nitin Ahuja
Senior Software Engineer
AWS

What is DocumentDB?

Document Database Platform

NoSQL database system designed to store, retrieve, and manage data as documents.

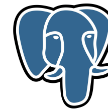


High MongoDB Compatibility

The goal is to provide full compatibility with the MongoDB API, the most popular document database API.

Open Source & Portable

Run anywhere: AWS, Azure, other clouds, on-premises, k8, or local desktops with the same functionality and compatibility.



Built on PostgreSQL

Leverages the reliability, maturity, and power of the PostgreSQL engine as its foundation.

Founding Members

Microsoft

Amazon

Rippling

YugabyteDB

AB InBev

Benefits of the Document Data Model

"Document" is a self-contained unit holding key-value pairs with support for nested structures like arrays and sub-objects (BSON).

Schema Flexibility

- Two documents in the same collection don't have to have the same fields.
- Add or change fields. per-document without migrations.
- Ideal for evolving data models and varying record shapes.

Index and Query by Content

Index and query fields directly inside the document, including nested arrays and sub-objects.

Hierarchical Data

Naturally represent nested relationships without complex relational joins.
E.g.

- Orders with line items
- A user with embedded preferences

Developer Ergonomics

Documents map cleanly onto application objects that have nested fields, lists, and optional properties.
No ORM, no object-relational impedance mismatch.
Preferred choice for modern web and mobile apps.

Aggregation Pipelines

A pipeline is an ordered list of **stages** where each stage transforms a stream of documents and hands the result to the next - like Unix pipes for data.

```
collection → [ $match ] → [ $group ] → [ $sort ] → [ $limit ] → result
```

```
db.orders.aggregate([
{ $match: { status: "shipped" } }, // filter
{ $group: { _id: "$region", total: { $sum: "$amount" } } }, // aggregate
{ $sort: { total: -1 } }, // order
{ $limit: 10 } // top N
])
```

Common Stages

Stage	Description
`$match`	Filter documents (SQL **WHERE**)
`$project / $set / $unset`	Shape fields and projection (SQL **SELECT**)
`$group`	Aggregate data (SQL **GROUP BY**)
`$sort, $limit, $skip`	Ordering and pagination
`$lookup`	Left outer join to another collection
`$unwind`	Deconstruct arrays into individual documents

GenAI Use Cases with DocumentDB

Retrieval Augmented Generation (RAG)

- Personalize LLM on your data
- Cheaper than fine tuning
- Faster iteration on new data

Operational + Semantic Database

- No ETL; Consistent data
- Reduce complexity & costs

Conversational History

- Conversational context & UX
- LLM optimizations
- Auditing capabilities

Semantic Caching

- Drastically reduces latency
- Saves on Token consumption
- Reduces costs for LLM

AI Agents

- Session history and Memories for context
- Agent-to-agent transactions and Statefulness

Capabilities

Native BSON

Built for document flexibility on PostgreSQL.

Document + SQL

Native BSON support with full PostgreSQL compatibility when you need advanced SQL.

Indexing

Rich indexing: Single-field, multi-key, compound, text, and geospatial indexes.

AI Vector Search

Embeddings and similarity search powered by pgvector for GenAI apps.

From CRUD to vector search, all on PostgreSQL!

Why build on PostgreSQL?

Proven Stability & Community

PostgreSQL gives us proven stability, an active community, and extensibility.

Standing on Shoulders

Instead of building from scratch, we use PostgreSQL's storage engine, replication, and ecosystem, adding a document layer on top.

The Power of Extensions

Every PostgreSQL release makes DocumentDB better for free through the power of an extension model.

Relational + Document

You don't have to choose. With DocumentDB on PostgreSQL, you get both relational and document capabilities.

Why OSS and Linux Foundation?

MIT License

Most permissive license with no restrictions for builders.

Community-Driven Development

Collaborative growth fueled by a global ecosystem of contributors.

Vendor-Neutral Governance

Managed under the Linux Foundation to ensure fair and open evolution.

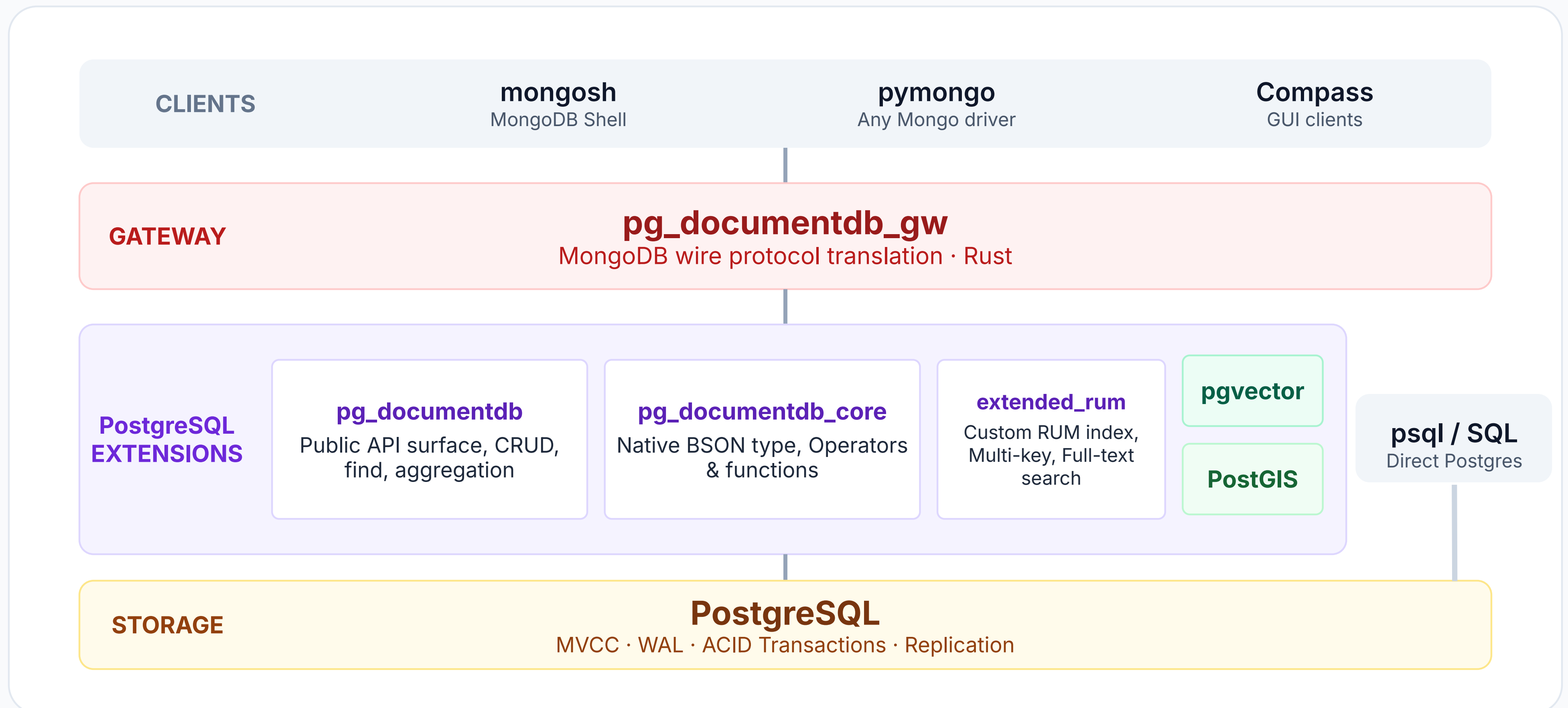
Deployment Choice

Total flexibility to deploy on-premise or in the cloud.

Full Source Access

Easy local development, testing, and the freedom to modify everything.

Architecture Overview



Internals

Collections

Every Mongo collection becomes one Postgres heap table in the **documentdb_data** schema.

```
Table "documentdb_data.documents_<collection_id>"
 Column          | Type                     | Nullable
-----------------+--------------------------+----------
 shard_key_value | bigint                   | not null
 object_id       | documentdb_core.bson     | not null
 document        | documentdb_core.bson     | not null
 creation_time   | timestamp with time zone |
Indexes:
    "collection_pk_<collection_id>" PRIMARY KEY, btree (shard_key_value,
object_id)
```

Metadata Catalog

The collection name → `collection_id` → Postgres table mapping lives in `documentdb_api_catalog`

Query Processing Flow

1. Client Request
Mongo Client
```
db.find({price: {$gt: 20}})
```

2. Wire Protocol
Gateway
Parses the query & invoke SQL UDF
```
SELECT cursorpage
FROM
documentdb_api.find_cursor_first_page(
'mydb', '{ "find": "users", "filter": {
"price": { "$gt": 20 } } }' );
```

3. Core Logic
pg_documentdb Extension
Parses BSON and expands to use native PostgreSQL operators (@=, @>, etc.)

4. Optimization
Planner
Leverages PostgreSQL planner with hooks to guide index selection

5. Execution
Executor
Leverages standard PostgreSQL executor for result retrieval

Process completes with BSON results returned to the Mongo Client via the gateway.

pg_documentdb_gateway

What it is

A Rust-based protocol-translation layer that lets MongoDB clients talk to a PostgreSQL backend. Shipped as the `pg_documentdb_gw` crate and extension.

How it works

Listen
Terminates TLS connections

Authenticate
Validates via connection pool

Parse
Decodes BSON commands

Translate
Rewrites to SQL UDF

Execute
Dispatches over Postgres

Respond
Serializes results back

Why it matters

Drop-in MongoDB API on top of Postgres/YugabyteDB

Same driver, same queries.

BSON as a Native PostgreSQL Type

BSON (Binary JSON)
MongoDB's binary document format with typed values, designed for fast traversal.

pgbson = varlena

Consists of a 4-byte header plus raw BSON bytes

```
typedef struct {
  int32 vl_len_; /* varlena header */
  char vl_dat[]; /* BSON bytes */
} pgbson;
```

- Lives in Postgres heap pages
- Automatically **TOASTed** when large

PostgreSQL treats it like any other type — it can be indexed, compared, and passed to functions.

BSON Operators

MONGODB	POSTGRES OPERATOR	DESCRIPTION
`$eq`	`@=`	Equality
`$gt`	`@>`	Greater than
`$regex`	`@~`	Regex match
`$exists`	`@?`	Field exists
`$in`	`@*=`	In array

Note: Each operator supports both runtime evaluation (filter) and index pushdown.
This implementation is how MongoDB semantics become truly PostgreSQL-native.

Planner Integration

`planner_hook`
Rewrites queries and expands aggregation pipelines, then delegates to standard planner.

`set_rel_pathlist_hook`
Steers index selection and injects custom scan nodes for specific MongoDB operations.

`get_relation_info_hook`
Sorts indexes by priority: PK > Composite > Regular > Wildcard > SeqScan.

`explain_get_index_name_hook`
Maps internal identifiers to show recognizable MongoDB index names in EXPLAIN output.

"We don't replace the Postgres planner — we guide it"

Custom Scan Nodes

`Streaming cursor scans`

Handles pagination and continuation beyond standard planner state.

`Text & Vector search queries`

Carries specialized state like $meta score and vector evaluation metrics.

`EXPLAIN wrapper`

Captures and reports extended explain output for complex execution paths.

`Extension Initialization`

Registered at init to wrap standard execution paths with custom state context.

"We don't replace PostgreSQL execution — we wrap it with extra context"

The Indexing Challenge

Storage Foundation

Documents stored as a single BSON column in PostgreSQL.

Extraction & Indexing

Need to extract values from inside a binary blob and index specific keys (e.g., `db.createIndex({price: 1})`).

Search Semantics

Ordered scans are required for MongoDB's `.sort()` semantics and complex configurations like wildcards or collation.

Multikey Support

Arrays must produce multiple index entries for a single heap tuple (e.g., tags: [1, 2, 3] → three terms).

How Documents Get Indexed

Specific Key Path Extraction

Index is on a specific path, not the whole document. `db.createIndex({price: 1})` extracts values at document.price to generate index terms.

Compound & Wildcard Indices

Compound indices (`{price: 1, city: 1}`) index paths together. Wildcards (`{"$**": 1}`) decompose all fields into (path, value) pairs.

Multikey Array Support

Arrays like `tags: [1, 2, 3]` produce separate index terms for the same heap tuple, enabling powerful multikey search semantics.

Serialization & Storage

All terms are serialized as `BsonIndexTerm` and stored as bytea in the inverted index for high-performance retrieval.

Granular path extraction enables efficient querying of BSON-structured data

How RUM works

Relationship to GIN

RUM is a fork of GIN. While GIN maps keys to posting lists of TIDs, RUM adds additional info per posting entry to enable ordered scans.

The RUM Advantage

Ordered scans are critical for supporting MongoDB's `.sort()` semantics natively within the database engine.

Custom Access Method

Registered in PostgreSQL as a custom access method to handle DocumentDB specific indexing requirements.

Extending GIN to provide native MongoDB sorting and search performance

Why Extended-RUM?

The Problem: Standard RUM Limitations

Standard RUM lacked per-index configuration (`amoptsprocnum`). DocumentDB needs to specify path extraction, term size limits, wildcard config, and collation for each index.

The Implementation: Custom Operator Classes

Configuration is delivered through custom operator classes with per-index options. Solution: fork RUM → `pg_documentdb_extended_rum`.

```
CREATE ACCESS METHOD documentdb_extended_rum
TYPE INDEX HANDLER documentdb_extended_rumhandler;
```

Backwards Compatibility

Key invariant: same on-disk storage format. Fully backwards compatible with existing RUM indices.

Architectural Isolation

Changes are isolated to the query path and volatile path only, ensuring data integrity.

Extending RUM for granular per-index control while maintaining storage compatibility

What Extended-RUM Adds

- `Order-by pushdown:` Perform sorting directly within the index scan for efficiency.
- `Index-only scan support:` Fetch data directly from the index without heap access.
- `Scan capabilities:` Adds full support for backwards and parallel index scans.
- `Multikey tracking:` Status tracking integrated directly into the index structure.
- `Enhanced Visibility:` Custom EXPLAIN output for better query plan inspection.
- `Cost Estimation:` Advanced awareness of composite indexes for optimal plan selection.

Optimizing document performance with advanced PostgreSQL index features

Extended-RUM Evolution

`GIN (PostgreSQL core)`
Inverted index with posting lists. Foundation for full-text search, arrays, and JSONB.

`RUM (postgrespro)`
Extends GIN with additional info per posting entry, enabling ordered scans and tsvector distance.

`EX-RUM (DocumentDB)`
Forks RUM to add per-index configuration, order-by pushdown, index-only scan, backwards scan, parallel scan, and multikey tracking.

Postgres ❤ Extensions

Vector Indexing

Approximate-nearest-neighbor (ANN) search over BSON documents

DocumentDB leverages Postgres's **pgvector** via the MongoDB API (`$vectorSearch`).

Index Types & Metrics

Kinds: IVFFlat, HNSW

Distance: Cosine, Euclidean (L2), InnerProduct mapped to pgvector operators

Query Flow

1. Gateway translates `$vectorSearch` to SQL UDF.
2. `bson_extract_vector` pulls the embedding out of the BSON doc/query.
3. Planner rewrites to **`ORDER BY <=> LIMIT k`** against **pgvector** index.
4. Tuning params (`nProbes`, `efSearch`) applied as local GUCs before execution.

One store, one API: Transactional docs + ANN search in a single cluster

pg_cron in DocumentDB

Runs DocumentDB's recurring maintenance inside Postgres.

Why pg_cron?

- Single coordinator semantics: job runs in one backend with SPI access.
- Centralized state: Job status in cron.job; logs in cron.job_run_details.
- Native integration: No external cron required for maintenance scheduling.

Maintenance Jobs

Job Name	Schedule	Action / Purpose
documentdb_ttl_task	Every minute	delete_expired_rows() - sweep TTL-expired docs
documentdb_cursor_cleanup_task	Every minute	cursor_directory_cleanup() - evict stale cursors
documentdb_index_build_task_<N>	Minute / sub-minute	build_index_concurrently(N)

What’s next

Expanding API compatibility

- Change streams
- Full collation support
- More...

Infrastructure & Performance

- Comprehensive functional test suite with CI gating for every PR
- Continuous performance optimizations with dedicated micro-benchmarking workloads

Core Capabilities & Observability

- Exploring B-Tree indexing support for specific workload profiles
- Improved observability

Security & Developer Experience

- Enhanced RBAC with more granular privileges, pluggable auth
- Improved developer experience - easier setup, ecosystem integrations

Q & A

GitHub

Discord

Thank you!

GitHub

Discord