- S02.04 MongoDB deeper dive
  - SQL vs NoSQL
  - The workouts example - data is flexible
  - OLTP vs OLAP
  - Wrap up statement
  - Conclusion to SQL vs NoSQL - PostgreSQL vs MongoDB
  - Performance
  - Performance Comparison PostgreSQL vs MongoDB
  - Query planning explain in MongoDB
  - Query planning comparison
  - Schema design
  - Referencing vs Embedding
    - Embedding Advantages
    - Embedding Limitations
  - Referencing
    - Referencing Advantages
    - Referencing Limitations
  - Schema design and types of relationships
    - One-to-One
    - One-to-Few
    - One-to-Many
    - One-to-Squillions
    - Many-to-Many
  - Recap:
  - Schema design patterns
  - Extended reference pattern
  - The outlier pattern
  - Schema enforcement and validation
  - Enforcing a data type
  - practice
  - Conclusion
  - Links

S02.04 MongoDB deeper dive

Querying data is important.

But, if you already know SQL, it's easy and efficient and smart to give the database schema as background context to an LLM (chatGPT, Claude, ...) and convert the query from SQL to MongoDB.

Or simply prompt the LLM to write the query for you. At this point what matters is your ability to read, understand and challenge the queries written by the LLM. Not write them from scratch.

More important is learning what really characterizes MongoDB

- Choosing MongoDB over SQL
- Schema design patterns
- Specific features of MongoDB
- Query optimization
- Index creation

SQL vs NoSQL

This is the first question at the *early stage* of your application design.

We've talked about it last time. But it's worth looking into it in more details.

In short Document based database make sense

- web scale
- flexible data schema

btw: it's well known that **MongoDB is webscale**: https://www.youtube.com/watch?v=b2F-DItXtZs

You've outgrown a simple sqlite or JSON based data store. What should you aim for next ?

An excellent article on when to use NoSQL: https://medium.com/@sqlinsix/when-to-use-sql-or-nosql-b50d4a52c157 (available as pdf in the github repo)

Quoting:

> Some questions that I consider when I compare these different back-ends:

- How often will I be reading and writing the data?
- How often will the structure of the data change?
- How much of the data is unknown?
- What does the final product look like?

The workouts example - data is flexible

Imagine a workout app where you track your workouts.

We have standard SQL friendly entities:

a `user` has many `sessions`, a `gym` has many `users`, a `user` has a `subscription`, `gyms` have many `subscriptions` and vice versa.

But there's also have a **workout** entity that by nature can vary a lot.

- weights and reps : dead lifts, clean and jerks, ...
- no weights: burpees, sit-ups, squats, planking, ...
- running, rowing, swimming, ...
- bikes, walks, sky jumps, ...
- power yoga, pilates, Zumba, ...

Forcing that workout data into a fixed data structure is difficult.

from that same article: "*The join complexity would be enormous relative to a person's workout, as there may be dozens of combinations. In addition, workouts such as drop sets where the weight is constantly changing and a maximum (or minimum) repetition range must be met would be extremely challenging to track with a set schema using SQL.*"

Workouts is a good example where a **flexible schema** data structure makes sense.

NoSQL from the start: think about how the app would consume the data — for example recording or retrieving the entire workout for a certain day as a whole.

We don't need joins to return all the related data from a normalized data process because the entire entity is stored as a single piece of information.

No Need for joins for reads or for writes.

Note: PostgreSQL now has JSON data types that can be used for such a case of polymorphic data.

OLTP vs OLAP

That's one of the main database design use case. Is your database OLTP or OLAP ?

2 main database usage types: transactions or dashboards

OLTP (Online Transaction Processing) systems focus on handling a large number of short, atomic transactions in real-time. They are optimized for write operations and transactional integrity, commonly used for applications like e-commerce, banking, and inventory management.

In OLTP systems, we optimize for writes as much as possible. Data integrity and normalization is key

OLAP (Online Analytical Processing) systems are designed for querying and analyzing large datasets, often involving complex joins, aggregations, and data transformations. These systems are *read-intensive* and require optimized data retrieval. Denormalization simplifies queries increase

performance. Think OLAP ~= dashboards.

So a legit question is, *should we use NoSQL or SQL for OLAP ? for OLTP ?*

- For OLTP workloads: MongoDB's document model excels at handling many small, quick transactions because each document contains related data in a single unit.
- For OLAP workloads: MongoDB has some limitations that make it less ideal for complex analytical queries:
    - The aggregation pipeline, while powerful, can become quite complex for sophisticated analytical queries that would be simpler in SQL. Joining large amounts of data across collections can be memory-intensive and slower compared to traditional RDDBMs.
    - MongoDB's storage engine is optimized for random access patterns (typical in OLTP) rather than the sequential scans common in analytical workloads.

[TODO] question the LLM: Question: if you transfer the workload of joins and aggregation to the collection level (through regular background jobs for instance), then a document store such as MongoDB becomes competitive for OLAP applications ? What about views in MongoDB ?

Wrap up statement

This statement sheds a light on when to use NoSQL

from : https://softwareengineering.stackexchange.com/a/355964/440337

> *NOSQL is very much designed as a persistence layer for an application.*
> *So access is optimized for a single object with children.*

Many things to unfold here :

- **persistence layer** refers to the storing and retrieving data so that it survives beyond the application's runtime. It's where / how data gets saved permanently, rather than just existing temporarily in memory.

NoSQL (Document) databases are built specifically to serve applications' **data storage** needs. This differs from traditional databases that are designed for data organization and relationships, with application needs being secondary.

A **single object** refers to a primary data entity (like a user profile, a product, or a workout)

Children refers to the related data that belongs to or is closely associated with that primary object (like a user's addresses, a product's reviews, or a workout's specifics)

Why is it optimized for a *single object* with *children*?

Single is an important word here:

Because:

NoSQL databases physically store the primary object and its children together in a **single location** on disk. Instead of splitting related data across multiple tables as in relational databases.

Retrieving all related data requires fewer disk operations and is faster (less resources intensive).

This statement also implies a key architectural principle: NoSQL databases are designed around the specific ways applications need to access and manipulate data, rather than around maintaining formal data relationships.

This makes them particularly effective when an application needs to quickly retrieve or update a complete object and all its associated data in a single operation.

What about multiple objects ?

[TODO] recap

Several technical factors can affect performance:

- **Indexing**: First, NoSQL databases typically don't maintain the same type of sophisticated indexes that relational databases do for joining and relating data across different collections. While MongoDB does support indexes, they are primarily designed to help locate individual documents rather than optimize complex queries across multiple documents.
- **Disk access**: Second, when you need to access multiple objects that aren't physically stored near each other, the database needs to perform multiple random disk seeks. These random seeks are much slower than sequential reads because the disk head needs to physically move to different locations. This is why operations that need to scan many documents (like analytical queries) can be slower in MongoDB compared to databases designed for that purpose.
- **memory management**: Third, MongoDB's memory management is optimized around the concept of working sets - the documents that are most frequently accessed. When operating on large volumes of data that exceed the size of RAM, MongoDB needs to constantly swap documents in and out of memory, which can significantly impact performance.

On point 2, Disk access: it seemed to me that the problem of data being spread out all over the disk at random locations is true for all databases.

But PostgreSQL is much more optimized for that kind of tasks than MongoDB is.

There are key architectural differences in how both databases handle multi-record operations:

- Storage Layout: Better and more constrained storage organization in PostgreSQL
    - PostgreSQL stores data in tables with fixed schemas, using a concept called "heap files" where records are stored in blocks/pages. These pages are designed for efficient sequential scanning.
    - MongoDB stores each document independently, potentially with varying sizes and schemas. This can lead to more fragmentation and less efficient sequential access.
- Data Access Methods: better query planning
    - PostgreSQL has sophisticated query planning that can choose between different access methods:
    - Sequential scans that read pages efficiently in order

    - Index scans that can use multiple indexes simultaneously
    - Bitmap scans that can combine results from multiple indexes
    - MongoDB primarily relies on single-index scans or collection scans, with fewer options for combining access methods

- Join Operations:
    - PostgreSQL has specialized algorithms (hash joins, merge joins, nested loops) that are optimized for combining data from multiple tables efficiently
    - MongoDB needs to perform lookups one document at a time when combining data across collections, which often means more random I/O

- Buffer Management:
    - PostgreSQL's buffer manager is designed to optimize for both sequential and random access patterns
    - MongoDB's WiredTiger storage engine is primarily optimized for random access patterns

So while both databases do need to perform disk operations, PostgreSQL includes specific optimizations for handling multi-record operations efficiently, particularly when dealing with joins and large table scans. These optimizations are less present in MongoDB because its architecture prioritizes single-document operations.

Conclusion to SQL vs NoSQL - PostgreSQL vs MongoDB

There's no escaping complexity. Speed is not the only factor you should take into account when choosing a database.

Side note: availability and productivity of engineers is independent of any technical consideration but has more impact on costs and revenue than a few milliseconds.

What drives the adoption of MongoDB vs PostgreSQL is

- app data consumption
- single record storing and retrieving
- complexity of data

Performance

Performance Comparison PostgreSQL vs MongoDB

Read this post https://medium.com/@vosarat1995/postgres-vs-mongo-performance-comparison-for-semi-structured-data-9a5ec6486cf6

The author creates a large dataset and tests the performance of MongoDB and

```
| Test                                | Mongo     | PostgreSQL  |
|-------------------------------------|-----------|-------------|
| Index creation on an empty database | 152.9 ms  | 402.0 ms    |
| Batch insert                        | 53.50 ms  | 219.15 ms   |
| One by one insert                   | 319.3 ms  | 641.9 ms    |
| Read multiple records               | 21.83 ms  | 66.63 ms    |
| Index creation on filled database   | 1.563 s   | 1.916 s     |
| Aggregated query                    | 174.51 ms | 60.43 ms    |
```

Bash

Query planning explain in MongoDB

see https://chatgpt.com/c/6750283d-9f4c-800e-99ed-930e0ad24cf3

We won't go into the types of indexes for NoSQL database. but just illustrate the query planning of a few queries with and without indexes

to EXPLAIN a query

- direct querying: `db.movies.find(<query predicate>).explain("executionStats")`
- aggregation pipelines: `db.movies.explain("executionStats").aggregate(<the_pipeline>)`

Look for

- `totalDocsExamined`
- `nReturned`
- `executionTimeMillisEstimate` : estimate the cumulative execution time for the pipeline up to and including that stage. top is highest
- `rejectedPlans`
- `stage` : COLLSCAN, IXSCAN

compare: `db.movies.find({ "title": "The Perils of Pauline" }).explain("executionStats")`

Then create an index `db.movies.createIndex({ title: 1 })`

again `db.movies.find({ "title": "The Perils of Pauline" }).explain("executionStats")`

Drop the index `db.movies.dropIndex({title : 1})`

Explain an aggregation pipeline `db.movies.explain("executionStats").aggregate([ { $group: { _id: "$genres", averageRating: { $avg: "$imdb.rating" } } }] )`

Query planning comparison

Consider a similar query scenario that involves joining/relating data and aggregating results.

Query: *find all movies from the 1990s along with their directors, grouped by director with average*

ratings.

First, MongoDB:

```bash
db.movies.aggregate([
  { $match: {
      year: { $gte: 1990, $lt: 2000 }
  }},
  { $unwind: "$directors" },
  { $group: {
      _id: "$directors",
      avgRating: { $avg: "$imdb.rating" },
      movieCount: { $sum: 1 }
  }}
]).explain("executionStats")
```

This would produce an explain plan like:

```javascript
{
  "stages": [
    {
      "$cursor": {
        "queryPlanner": {
          "plannerVersion": 1,
          "namespace": "sample_mflix.movies",
          "indexFilterSet": false,
          "parsedQuery": {
            "year": { "$gte": 1990, "$lt": 2000 }
          },
          "winningPlan": {
            "stage": "COLLSCAN",  // Full collection scan
            "filter": { "year": { "$gte": 1990, "$lt": 2000 } }
          },
          "rejectedPlans": []
        }
      }
    },
    { "$unwind": "$directors" },  // Memory-intensive operation
    {
      "$group": {
        "_id": "$directors",
        "avgRating": { "$avg": "$imdb.rating" },
        "movieCount": { "$sum": 1 }
      }
    }
  ]
}
```

Now the equivalent in PostgreSQL (assuming normalized schema):

```SQL
EXPLAIN ANALYZE
SELECT d.name as director,
  AVG(m.rating) as avg_rating,
  COUNT(*) as movie_count
FROM movies m
JOIN movie_directors md ON m.id = md.movie_id
JOIN directors d ON md.director_id = d.id
WHERE m.year >= 1990 AND m.year < 2000
GROUP BY d.name;
```

This would produce an explain plan like:

```Bash
QUERY PLAN
-----------------------------------------------------------
HashAggregate  (cost=245.97..247.97 rows=200)
  Group Key: d.name
  ->  Hash Join  (cost=121.67..237.42 rows=1710)
        Hash Cond: (md.director_id = d.id)
        ->  Hash Join  (cost=66.50..164.42 rows=1710)
              Hash Cond: (md.movie_id = m.id)
              ->  Seq Scan on movie_directors md
              ->  Hash  (cost=58.00..58.00 rows=680)
                    ->  Index Scan using movies_year_idx on movies m
                          Index Cond: (year >= 1990 AND year < 2000)
        ->  Hash  (cost=40.50..40.50 rows=1173)
              ->  Seq Scan on directors d
```

Key differences in query planning may include:

1. Join Handling:
    - PostgreSQL uses **hash joins** to efficiently combine data from multiple tables
    - MongoDB must `$unwind` the embedded directors array, which creates multiple documents in memory
2. Index Usage:
    - PostgreSQL can use **multiple indexes** simultaneously and choose different join strategies
    - MongoDB typically relies on a single index per stage of the aggregation
3. Memory Management:
    - PostgreSQL's hash joins build hash tables in memory with fallback to disk
    - MongoDB's `$unwind` and `$group` stages must hold their working sets in memory
4. Operation Order:

    - PostgreSQL's query planner can **reorder operations** for efficiency
    - MongoDB must process the aggregation pipeline stages in sequence
5. Statistics Usage:
    - PostgreSQL's explain shows detailed cost estimates based on table statistics
    - MongoDB's explain focuses more on operation type and index usage

The PostgreSQL plan shows it can:

- Use an index scan for the year filter
- Build hash tables for efficient joining
- Perform grouping with hash aggregation
- Estimate costs and row counts at each step

👽👽👽 This structured approach to complex queries is why PostgreSQL often performs better for analytical workloads involving multiple tables/collections and aggregations.

Schema design

MongoDB schema design is the most critical part of deploying a scalable, fast, and affordable database

If you design the MongoDB schema like a relational schema you lose the benefits and power of NoSQL

MongoDB Schema Design: Data Modeling Best Practices

https://www.mongodb.com/developer/products/mongodb/mongodb-schema-design-best-practices/

[TODO] what ??

> It depends. This is because document databases have a rich vocabulary that is capable of expressing data relationships in more nuanced ways than SQL. There are many things to consider when picking a schema. Is your app read or write heavy? What data is frequently accessed together? What are your performance considerations? How will your data set grow and scale?

In SQL databases, the tl;dr of **normalization** is to split up your data into tables, so you don't duplicate data.

- 1 user
- has many cars
- has one jobs

In sql : 3 tables, apply Normalization forms etc

With MongoDB schema design, there is:

- No formal process
- No algorithms
- No rules

Not Scary at all !!! 😳😳😳

What matters is that you design a schema that will work well for your application. Two different apps that use the same exact data might have very different schemas if the applications are used differently.

👹👹👹 **The app drives the schema!**

Referencing vs Embedding

[TODO]: add example to illustrate

Embedding Advantages

- You can retrieve all relevant information in a single query.
- Avoid implementing joins in application code or using $lookup.
- Update related information as a single atomic operation.

[TODO] what ??

Transactions :

- By default, all CRUD operations on a single document are ACID compliant.
- However, if you need a transaction across multiple operations, you can use the transaction operator.
- Though transactions are available starting 4.0, however, I should add that it's an anti-pattern to be overly reliant on using them in your application.

Embedding Limitations

- Large documents mean more overhead if most fields are not relevant. You can increase query performance by limiting the size of the documents for each query.
- There is a **16-MB document size limit in MongoDB**. If you are embedding too much data inside a single document, you could potentially hit this limit.

Referencing

Okay, so the other option for designing our schema is referencing another document using a document's unique object ID and connecting them together using the $lookup operator. Referencing works similarly as the JOIN operator in an SQL query. It allows us to split up data to make more efficient and scalable queries, yet maintain relationships between data.

Referencing Advantages

- By splitting up data, you have smaller documents which are less likely to reach 16-MB-per-document limit.

- Some data is not accessed as frequently as other data. So it makes sense to separate it from the main data.
- It reduces the amount of duplication of data.

Referencing Limitations

- In order to retrieve all the data in the referenced documents, a minimum of two queries or `$lookup` required to retrieve all the information. MongoDB is less efficient for joins than a SQL db can be.

Schema design and types of relationships

Consider the types of relations between entities

One-to-One

Modeled as key value pairs in the database. For instance

- train <-> number of passengers
- user <-> dob
- tree <-> variety
- human <-> weight

One-to-Few

A small sequence of data that's associated with the main entity.

- a person has a few addresses
- a playlist has a few tracks
- a recipe has a few ingredients
- a Parisian has a few favorite cafes

One-to-Many

- a gym has many users
- a product has many parts
- a bus line has many stops
- your instagram has many followers

One-to-Squillions

There could be potentially millions of subdocuments, or more?

- server <-> loads of log events
- university <-> thousands of students
- celebrity social account <-> millions of followers

A server logging application Each server could potentially save a massive amount of data, depending on how verbose you're logging and how long you store server logs for.

instead of tracking the relationship between the host and the log message in the host document, let's let each log message store the host that its message is associated with. By storing the data in the log, we no longer need to worry about an unbounded array messing with our application! Let's take a look at how this might work.

Many-to-Many

Example of a project planning application:

- a user may have many tasks
- a task may have many users assigned to it.

Store the ID of the users as an array in the task document and the Id of the tasks in the user document.

Users:

```JavaScript
{
    "_id": ObjectID("AAF1"),
    "name": "Kate Monster",
    "tasks": [ObjectID("ADF9"), ObjectID("AE02"), ObjectID("AE73")]
}
```

Tasks:

```JavaScript
{
  "_id": ObjectID("ADF9"),
  "description": "Write blog post about MongoDB schema design",
  "due_date": ISODate("2014-04-01"),
  "owners": [ObjectID("AAF1"), ObjectID("BB3G")]
}
```

From this example, you can see that each user has a sub-array of linked tasks, and each task has a sub-array of owners for each item in our to-do app.

Recap:

- **One-to-One** - Prefer key value pairs within the document
- **One-to-Few** - Prefer embedding
- **One-to-Many** - Prefer embedding
- **One-to-Squillions** - Prefer Referencing
- **Many-to-Many** - Prefer Referencing

General Rules for MongoDB Schema Design:

- Rule 1: Favor embedding unless there is a compelling reason not to.
- Rule 2: Needing to access an object on its own is a compelling reason not to embed it.

- Rule 3: Avoid joins and lookups if possible, but don't be afraid if they can provide a better schema design.
- Rule 4: Arrays should not grow without bound. If there are more than a couple of hundred documents on the many side, don't embed them; if there are more than a few thousand documents on the many side, don't use an array of ObjectID references. High-cardinality arrays are a compelling reason not to embed.
- Rule 5: As always, with MongoDB, how you model your data depends entirely on your particular application's data access patterns. You want to structure your data to match the ways that your application queries and updates it.

Rule 1: **Favor embedding unless there is a compelling reason not to.**

Rule 2: **Needing to access an object on its own is a compelling reason not to embed it.**

Rule 3: **Avoid joins/lookups if possible, but don't be afraid if they can provide a better schema design.**

Rule 4: **Arrays should not grow without bound.** If there are more than a couple of hundred documents on the "many" side, don't embed them; if there are more than a few thousand documents on the "many" side, don't use an array of ObjectID references. **High-cardinality arrays are a compelling reason not to embed.**

Rule 5: **how you model your data depends – entirely – on your particular application's data access patterns**. You want to structure your data to match the ways that your application queries and updates it.

Let's look at 2 schema design patterns to illustrate the flexibility and the way the app drives the data schema design.

Schema design patterns

Extended reference pattern

Let's go through this article https://www.mongodb.com/developer/products/mongodb/schema-design-anti-pattern-massive-arrays/

> One of the rules of thumb when modeling data in MongoDB is data that is accessed together should be stored together.

Building has many employees : potentially too many for the 16Mb document limit.

So turn this around with

Employee belongs to a building: embed the bvuildign info in the employee document.

> If we are frequently displaying information about an employee and their building in our application together, this model probably makes sense.

Problem:

- way to much data duplication.
- if we need to update a building information : then we must update all the employees documents

So let's separate employees and building in 2 sepatate colections and use $lookups.

But `$lookups` are expensive

So we use the extended reference pattern where we duplicate some—but not all—of the data in the two collections. We only duplicate the data that is frequently accessed together.

> For example, if our application has a user profile page that displays information about the user as well as the name of the building and the state where they work, we may want to embed the building name and state fields in the employee document

The outlier pattern

The outlier pattern: when only few documents have a huge amount of children documents.

https://www.mongodb.com/blog/post/building-with-patterns-the-outlier-pattern

Consider a book collection and the list of user who have bought the book.

```JavaScript
{
    "_id": ObjectID("507f1f77bcf86cd799439011")
    "title": "A Genealogical Record of a Line of Alger",
    "author": "Ken W. Alger",
    …,
    "customers_purchased": ["user00", "user01", "user02"]

}
```

Most books sell only a few copies. This is the long tail of books sales.

So it can make sense to have an array of users who have bought the book in each book document.

However some books, very few, sell millions of copies. The purchaser array cannot work.

The outlier pattern consist in adding a field that flags the book document as an outlier. For instance if the book has sold more than a 1000 copies.

```JavaScript
{
    "_id": ObjectID("507f191e810c19729de860ea"),
    "title": "Harry Potter, the Next Chapter",
    "author": "J.K. Rowling",
    …,
   "customers_purchased": ["user00", "user01", "user02", …, "user999"],
   "outlier": "true"
}
```

In the app code we test for the presence of that flag and deal with the data differently if the flag is present.

The outlier pattern is frequently used in situations when popularity is a factor, such as in social network relationships, book sales, movie reviews,

---

Schema enforcement and validation

Enforcing a data type

But it's possible to enforce a data type when declaring the schema

```Bash
db.createCollection("movies", {
    validator: {
        $jsonSchema: {
            bsonType: "object",
            required: ["year", "title"],  // Fields that must be present
            properties: {
                year: {
                    bsonType: "int",  // Enforces that `year` must be an integer
                    description: "Must be an integer and is required"
                },
                title: {
                    bsonType: "string",
                    description: "Title of the movie, must be a string"
                },
                imdb: {
                    bsonType: "object",  // Nested object for IMDb data
                    properties: {
                        rating: {
                            bsonType: "double",  // IMDb rating must be a double
                            description: "Must be a double if present"
                        }
                    }
                }
            }
        }
    }
})
```

Key Points:

- bsonType: Specifies the BSON data type for the field (e.g., int, string, array, object).
- required: Ensures specific fields are mandatory.
- properties: Defines the constraints for each field.
- description: Adds a helpful description for validation errors.

MongoDB supports schema validation starting from version 3.6, which allows you to enforce data types and other constraints on fields within a collection. This is achieved using the $jsonSchema validation feature when creating or updating a collection.

https://www.digitalocean.com/community/tutorials/how-to-use-schema-validation-in-mongodb

When you add validation rules to an existing collection, though, the new rules won't affect existing documents until you try to modify them.

Small example on trees dataset ?

JavaScript

```
{
        "idbase":249403,
        "location_type":"Arbre",
        "domain":"Alignement",
        "arrondissement":"PARIS 20E ARRDT",
        "suppl_address":"54",
        "number":null,
        "address":"AVENUE GAMBETTA",
        "id_location":"1402008",
        "name":"Tilleul",
        "genre":"Tilia",
        "species":"tomentosa",
        "variety":null,
        "circumference":85,
        "height":10,
        "stage":"Adulte",
        "remarkable":"NON",
        "geo_point_2d":"48.86685102642415, 2.400262189227641"
    },
```

write a schema and validator for that data

validator should enforce

- height >= 0 and <100
- stage ['']

---

practice

take a json version of the trees dataaset where cols with null values have been remved

1. freedom rules
    - insert all the data without validation
    - query to check out the absurd values : height, geolocation
2. more controlled approach
    - write schema and validator: using MongoDB's $jsonSchema validator to "dry run" your data

validation.
    - check number of skip documents
    - write a validator that gets as many documents as possible while keeping out absurd values
3. indexes
    - add a unique index on geolocation

in MongoDB, the schema validator is typically created along with the collection, not as a separate step

---

Conclusion

https://galaktika-soft.com/blog/sql-vs-nosql.html

Points to consider:

- Single data point vs large volumes
- aggregation pipelines are slower than complex SQL queries
- reads vs writes
- storage and indexing
- ACID compliance : Mongodb ensures ACID since v4 https://www.mongodb.com/products/capabilities/transactions and see https://www.mongodb.com/resources/products/capabilities/acid-compliance
- MongoDB scales horizontally, SQL vertically

and most important factor

- how is the data consumed by the app

Links

Schema Design Patterns: https://learn.mongodb.com/courses/schema-design-patterns