DATABASES

MongoDB Intermediate/Advanced

Aggregation Framework & Deep Dive · Schema Design at Scale

Transactions & Replication · Sharding

Performance & Security · Capstone Project

Philip Osztromok

Generated with Claude

Table of Contents

MongoDB Intermediate/Advanced — 8 Chapters

🔄 The Aggregation Framework

MongoDB Fundamentals covered `find()` — filtering and shaping single documents. But "total spent per customer" or "top 5 products by revenue" needs grouping, computing, and combining data across documents, the same territory `mysql_advanced_07`'s window functions and `mysql_foundations_07`'s `GROUP BY` chapter covered for SQL. MongoDB's answer is the **aggregation pipeline** — this chapter builds it up stage by stage, translating a familiar SQL query into it along the way.

Why find() Isn't Enough

`find()` is built to locate and shape **individual** documents — filter by a condition, pick which fields to return, maybe sort the results. What it can't do is combine **many** documents into a computed summary: it has no built-in way to sum a field across every matching document, count how many documents fall into each category, or pull in related data from another collection. Those are exactly the jobs SQL's `GROUP BY`, aggregate functions, and `JOIN` handle. The aggregation framework is MongoDB's dedicated tool for that same class of problem.

The Pipeline Model: Stages in Sequence

An aggregation is an **array of stages** passed to `db.collection.aggregate([ ... ])`. Each stage takes the documents coming out of the previous stage, transforms them somehow, and passes the result to the next stage — a pipeline in the literal sense, the same mental model as piping commands together in a shell. Nothing is implicit the way a SQL engine's internal query planner is: you're explicitly building the sequence of transformations yourself, one stage at a time.

A SQL Query, Then the Same Thing as a Pipeline

SQL

```
SELECT customer_id, SUM(total) AS total_spent
FROM orders
WHERE status = 'completed'
GROUP BY customer_id
ORDER BY total_spent DESC;
```

MongoDB Aggregation Pipeline

```
db.orders.aggregate([
 { $match: { status: "completed" } },
 { $group: {
    _id: "$customer_id",
    total_spent: { $sum: "$total" }
 } },
 { $sort: { total_spent: -1 } }
])
```

Three SQL clauses became three pipeline stages, in the same order they'd logically run: filter first, then group, then sort. That ordering correspondence holds throughout this chapter — it's the fastest way to reason about a new pipeline stage-by-stage.

$match — Filtering Documents (like WHERE)

`$match` keeps only documents matching a condition, using the same query syntax as `find()`. It's almost always the first stage when there's a filter to apply, because it shrinks the document set before any heavier work happens downstream.

```
{ $match: { status: "completed" } }
```

$group — Grouping & Computing (like GROUP BY)

`$group` collapses documents that share a value into one output document per group, exactly like SQL's `GROUP BY`. The group's key goes in `_id` — even though it's not a document's real `_id` field here, just a naming convention this stage reuses — and every other field is built from an **accumulator expression** such as `$sum`, `$avg`, `$max`, `$min`, or `$count`.

```
{ $group: {
   _id: "$customer_id",
   total_spent: { $sum: "$total" },
   order_count: { $sum: 1 },
   largest_order: { $max: "$total" }
} }
```

That single stage does the SQL equivalent of `SUM(total)`, `COUNT(*)`, and `MAX(total)` in one pass. Grouping "everything into one bucket" (a single grand total across the whole collection) uses `_id: null`.

$project — Reshaping Output (like SELECT's column list)

`$project` chooses which fields survive into the next stage and can compute new ones, the same job SQL's column list and computed expressions do in a `SELECT`.

```
{ $project: {
   _id: 0,
   customer_id: "$_id",
   total_spent: 1,
   average_order: { $divide: [ "$total_spent", "$order_count" ] }
} }
```

$sort — Ordering Results (like ORDER BY)

`$sort` orders documents by one or more fields — `1` for ascending, `-1` for descending, mirroring SQL's `ASC`/`DESC`.

```
{ $sort: { total_spent: -1 } }
```

$lookup — Joining Collections (like JOIN)

MongoDB Fundamentals' reference table (`mongodb1-1`) flagged `$lookup` as this course's answer to relational `JOIN`s. It pulls in matching documents from another collection, attaching them as an array field — a **left outer join** by default, so a customer with zero orders still comes through with an empty array rather than being dropped.

```
{ $lookup: {
   from: "customers",
   localField: "customer_id",
   foreignField: "_id",
   as: "customer_info"
} }
```

Putting It Together: The Full Translated Pipeline

Extending the SQL query above to also pull in each customer's name — a second table in SQL, a second collection joined via `$lookup` here:

```
db.orders.aggregate([
 { $match: { status: "completed" } },
 { $group: {
    _id: "$customer_id",
    total_spent: { $sum: "$total" }
 } },
 { $lookup: {
    from: "customers",
    localField: "_id",
    foreignField: "_id",
    as: "customer_info"
 } },
 { $sort: { total_spent: -1 } }
])
```

Five lines of SQL became four explicit stages — each one doing exactly one job, in the exact order it logically needs to run.

Pipeline Stage	SQL Equivalent	Purpose
`$match`	`WHERE`	Filter documents by a condition
`$group`	`GROUP BY` + aggregate functions	Collapse documents into computed groups
`$project`	`SELECT` column list	Reshape output, add computed fields
`$sort`	`ORDER BY`	Order the result set
`$lookup`	`JOIN`	Attach matching documents from another collection

When to Reach for the Aggregation Framework

find() Is Still Right For

Fetching individual documents, simple field filtering, basic sorting/pagination of one collection — no computation across documents needed.

Reach for aggregate() When

You need totals/averages/counts per group, data reshaped into a different structure, or documents combined across collections — the same signal that would make you write `GROUP BY` or a `JOIN` in SQL.

💻 Coding Challenges

Challenge 1: Translate a GROUP BY Query

Translate this SQL into an aggregation pipeline: `SELECT category, COUNT(*) AS product_count, AVG(price) AS avg_price FROM products GROUP BY category ORDER BY product_count DESC;`

Goal: Practice mapping a full `WHERE`-free `GROUP BY`/`ORDER BY` query onto `$group` and `$sort` stages, including two accumulators in the same stage.

→ Solution

Challenge 2: Add a $match Filter

Extend Challenge 1's pipeline so it only counts products where `in_stock` is `true`, without changing the grouping or sorting logic.

Goal: Practice inserting a `$match` stage in the correct position (before `$group`) and explain why that position matters.

→ Solution

Challenge 3: Design a $lookup Join

Given an `orders` collection with a `product_id` field and a separate `products` collection, write a `$lookup` stage that attaches each order's full product details as a field named `product_details`.

Goal: Practice identifying `localField`/`foreignField`/`as` for a one-collection join, the same skill this chapter's combined example used.

→ Solution

⚠️ Gotcha: Stage Order Isn't Just Style — It Changes Both Correctness and Speed

Putting `$match` as early as possible isn't a stylistic preference — a pipeline that groups first and filters after has to process every document in the collection before throwing most of them away, while filtering first (which can use an index, the same way a SQL `WHERE` clause does) shrinks the working set immediately. There's a correctness trap here too: once `$group` or `$project` renames or reshapes a field, every stage after it has to refer to the **new** name — reaching for a field's original name in a later stage is a common source of a silently-empty result rather than an error.

🎯 What's Next

The next chapter is **Aggregation Deep Dive** — new stages (`$unwind`, `$facet`, `$bucket`) for building genuinely multi-stage analytical pipelines, the spiritual counterpart to `mysql_advanced_05`'s CTEs and `mysql_advanced_06`'s window functions.

🧩 Aggregation Deep Dive

Chapter 1 built pipelines from `$match`, `$group`, `$project`, `$sort`, and `$lookup` — enough to answer most "one grouped total" questions. This chapter adds three stages built for genuinely analytical work: `$unwind`, `$facet`, and `$bucket` — MongoDB's spiritual counterpart to `mysql_advanced_05`'s CTEs and `mysql_advanced_06`'s window functions, for the same reason: single queries that used to need several round trips or awkward application-side code.

$unwind — Deconstructing Arrays Into Documents

Chapter 1's `$lookup` attaches matches as an **array** field, even when there's exactly one match (Challenge 3 flagged this directly). `$unwind` takes an array field and outputs one separate document per array element — the reverse of grouping. If an order's `product_details` array has one element, `$unwind` turns it into a plain object; if a document had an array of three tags, `$unwind` would turn one document into three, each with a different single tag.

```
db.orders.aggregate([
 { $lookup: {
    from: "products",
    localField: "product_id",
    foreignField: "_id",
    as: "product_details"
 } },
 { $unwind: "$product_details" }
])
```

After this pipeline, each order document has a plain `product_details` *object* instead of a one-element array — much easier for application code (or a later `$group`) to work with directly via dot notation like `product_details.name`.

⚠️ Gotcha: $unwind Drops Documents With an Empty or Missing Array by Default

If an order's `product_id` didn't match anything, `$lookup` attaches an **empty** array — and `$unwind`'s default behavior is to drop a document entirely when the array it's unwinding is empty or missing, since there's no element to "become" the new document. That silently removes otherwise-valid orders from the pipeline's output. To keep them (with `product_details` set to `null` instead), pass the object form: `{ $unwind: { path: "$product_details", preserveNullAndEmptyArrays: true } }`.

$facet — Multiple Result Sets, One Round Trip

A common real-world need: return a page of results **and** a total count of all matching documents, for pagination — normally two separate queries. `$facet` runs several independent sub-pipelines against the *same* input documents in a single aggregation call, each sub-pipeline's output landing in its own named array in one result document.

```
db.products.aggregate([
 { $match: { category: "electronics" } },
 { $facet: {
    paginatedResults: [
     { $sort: { price: -1 } },
     { $skip: 0 },
     { $limit: 10 }
    ],
    totalCount: [
     { $count: "count" }
    ]
 } }
])
```

The output is *one* document with two fields: `paginatedResults` (an array of up to 10 products) and `totalCount` (an array holding one document like `{ count: 143 }`). Both sub-pipelines started from the same `$match`-filtered set, computed independently, without either one affecting the other.

$bucket — Grouping Into Ranges

`$group` groups by an *exact* value shared across documents. `$bucket` groups by which **range** a value falls into — a built-in histogram, the kind of grouping SQL usually has to fake with a `CASE WHEN price < 50 THEN '0-50' ...` expression inside a subquery before it can `GROUP BY` that computed label.

```
db.products.aggregate([
 { $bucket: {
    groupBy: "$price",
    boundaries: [0, 50, 100, 200, 500],
    default: "500+",
    output: {
     count: { $sum: 1 },
     products: { $push: "$name" }
    }
 } }
])
```

`boundaries` defines the range edges (each bucket is `[inclusive, exclusive)`), and `default` catches anything falling outside all of them — here, anything `500` or above. `output` works exactly like `$group`'s accumulators, just applied per bucket instead of per exact value.

$group vs. $bucket

> **$group**
>
> Groups documents that share the exact same value — every order with `customer_id: "C104"` in one group.

> **$bucket**
>
> Groups documents whose value falls into the same range — every product priced `$50–$99.99` in one group, regardless of the exact price.

Combining All Three: A Real Analytical Pipeline

A dashboard query: for completed orders, get a price-range histogram AND a grand total, in one call.

```
db.orders.aggregate([
 { $match: { status: "completed" } },
 { $facet: {
    priceHistogram: [
     { $bucket: {
        groupBy: "$total",
        boundaries: [0, 25, 50, 100, 250],
        default: "250+",
        output: { count: { $sum: 1 } }
     } }
    ],
    grandTotal: [
     { $group: { _id: null, sum: { $sum: "$total" } } }
    ]
 } }
])
```

One aggregation call, two independent analytical results — the kind of thing that would otherwise mean two separate SQL queries against the same table.

Stage	Purpose	Closest SQL Equivalent
`$unwind`	Turn one array field into many documents	A join against a comma-split/normalized child table
`$facet`	Run parallel sub-pipelines, one input, several outputs	Several separate queries, or a CTE reused twice
`$bucket`	Group by value range, not exact value	`CASE WHEN` + `GROUP BY` on the computed label

💻 Coding Challenges

Challenge 1: Unwind a Tags Array

Given articles with a `tags: ["mongodb", "database", "nosql"]` array field, write a pipeline that unwinds `tags`, then groups by tag to count how many articles use each one.

Goal: Practice the unwind-then-group pattern for counting values inside an array field, not just a top-level field.

→ Solution

Challenge 2: Build a $facet Dashboard Query

Write a single aggregation on a `reviews` collection that returns both the average rating (`$avg`) and the 3 most recent reviews (sorted by `createdAt` descending, limited to 3) in one call.

Goal: Practice structuring two genuinely different sub-pipelines inside one `$facet` stage.

→ Solution

Challenge 3: Design a $bucket Age Histogram

Given a `users` collection with an `age` field, write a `$bucket` stage grouping users into `18-25`, `26-35`, `36-50`, and a catch-all for anything older, counting users in each bucket.

Goal: Practice choosing `boundaries` and a `default` bucket correctly, including which edge (upper or lower) each boundary belongs to.

→ Solution

⚠️ Gotcha: $facet Runs Every Sub-Pipeline in Memory, With No Index Help Past the First Stage

Because `$facet`'s whole point is running several sub-pipelines against the *same* input simultaneously, MongoDB can't use an index for the sub-pipelines themselves — indexes only help the stage that produces `$facet`'s input in the first place. Always put a `$match` (and `$sort`, if applicable to an existing index) **before** `$facet`, exactly as in this chapter's combined example, so every sub-pipeline starts from an already-small, already-filtered set rather than the full collection.

🎯 What's Next

The next chapter is **Schema Design Patterns at Scale** — the embedding-vs-referencing question from `mongodb1-5` revisited for large, real applications: the polymorphic pattern, the bucket pattern, the computed pattern, and when embedding breaks down enough that it's time to refactor toward references.

Schema Design Patterns at Scale

`mongodb1-5` framed schema design as a single choice: embed, or reference. At real scale that choice needs more nuance — three named patterns for problems that show up repeatedly, and a clear signal for when a schema that started out embedded has outgrown embedding and needs refactoring toward references.

Recap: Embedding vs. Referencing

Embedding nests related data directly inside a parent document — fast to read as one unit, no join needed. **Referencing** stores an ID and looks the related data up separately (via `$lookup`, Chapter 1) — more flexible, but costs a join. Every pattern below is really a more specific answer to that same underlying trade-off.

The Polymorphic Pattern

Sometimes a collection needs to hold documents that represent genuinely **different kinds** of the same broad concept — a `user_activity` collection logging logins, purchases, and profile edits, all sharing a `user_id` and a timestamp, but each carrying completely different extra fields. Cramming every possible field from every activity type onto one rigid shape wastes space and makes queries confusing. The **polymorphic pattern** keeps them in one collection anyway, distinguished by a **discriminator field** (commonly named `type`) that tells you which shape to expect.

```
// Three different "shapes," one collection, one discriminator field
{ user_id: "U1", type: "login", timestamp: ISODate("2026-07-01"), ip: "203.0.113.4" }
{ user_id: "U1", type: "purchase", timestamp: ISODate("2026-07-02"), order_id: "O55", amount: 42.50 }
{ user_id: "U1", type: "profile_edit", timestamp: ISODate("2026-07-03"), field_changed: "email" }
```

Querying "everything this user did" is one simple `find({ user_id: "U1" })` across all types; querying "just their purchases" adds `type: "purchase"` to the filter. This is the same discriminator idea `mongodb1-6`'s BSON type chapter used for `$jsonSchema` validation, just applied to distinguishing whole documents rather than validating one.

The Bucket Pattern

Naming note first: this is a **schema design** pattern, unrelated to the `$bucket` aggregation stage from Chapter 2 — they share a name because they solve conceptually similar range-grouping problems, but one shapes how you store data and the other shapes how you query it.

Time-series-like data (sensor readings, log events) often arrives one small document at a time — a document per reading, every few seconds, forever. That creates two problems: an enormous number of tiny documents, and an index entry for every single one of them. The **bucket pattern** groups many readings that fall in the same time window into **one** larger document instead.

One Document Per Reading vs. One Bucket Per Hour

Without Bucketing

```
{ sensor_id: "S1", timestamp: "10:00:01", temp: 21.4 }
{ sensor_id: "S1", timestamp: "10:00:02", temp: 21.5 }
// ...3,600 documents for one sensor, one hour
```

With the Bucket Pattern

```
{
 sensor_id: "S1",
 hour: "2026-07-06T10:00",
 readings: [
  { t: "10:00:01", temp: 21.4 },
  { t: "10:00:02", temp: 21.5 }
  // ...
 ],
 count: 3600
}
```

One document per sensor per hour, holding an array of that hour's readings, drastically cuts both document count and index size — at the cost of updating (appending to) an existing bucket document instead of always inserting a brand-new one.

The Computed Pattern

Chapter 1's aggregation pipelines can always compute a total or a count on demand — but doing that on **every single read** of frequently-accessed data is wasted, repeated work if the underlying data doesn't actually change very often. The **computed pattern** precomputes an expensive value once and stores it directly on the document, updating it incrementally as writes happen — the same `$inc` operator from `mongodb1-4`'s page-view-counter example, just applied to a running total instead of a single counter.

```
// Every new order updates the precomputed total directly — no re-aggregation needed to read it
db.customers.updateOne(
 { _id: "C104" },
 { $inc: { lifetime_spend: 42.50, order_count: 1 } }
)
```

Reading `lifetime_spend` is now a plain field lookup — no `$group` required — at the cost of that value only being as accurate as the last write that updated it, and needing every code path that creates an order to remember to run the increment.

When Embedding Breaks Down

Embedding Is Still Fine When

The embedded data has a bounded, predictable size, is almost always read together with its parent, and isn't updated independently by unrelated parts of the application.

Time to Refactor to Referencing When

An array can grow without any real limit (comments, readings, log entries), the embedded data is frequently read or updated on its own, or the parent document is creeping toward MongoDB's hard **16MB per-document size limit**.

Pattern	Problem It Solves	Trade-off
Polymorphic	One collection needs to hold genuinely different "shapes"	Every query and every reader needs to branch on the discriminator field
Bucket (schema)	Too many tiny time-series-style documents	Writes become updates-to-a-bucket instead of simple inserts
Computed	Re-aggregating the same value on every read is wasteful	Stored value can drift if a write path forgets to update it
Refactor to referencing	An embedded array's growth is unbounded, or nearing 16MB	Reads now need a `$lookup` join instead of being self-contained

Combining Patterns: An IoT Dashboard

A sensor-monitoring collection using both the bucket pattern (grouping readings by hour) and the computed pattern (precomputing that hour's min/max/average so the dashboard never re-aggregates raw readings just to render a summary card):

```
{
  sensor_id: "S1",
  hour: "2026-07-06T10:00",
  readings: [ /* ...3,600 entries... */ ],
  summary: { min: 20.9, max: 22.1, avg: 21.5 }
}
```

The dashboard's live "current hour" card reads `summary` directly — a plain field lookup, no aggregation pipeline needed at render time — while the full `readings` array stays available for anyone who genuinely needs the raw detail.

💻 Coding Challenges

Challenge 1: Model Polymorphic Notifications

Design polymorphic documents for a `notifications` collection holding three kinds: a friend request, a comment reply, and a system announcement. Give each a sensible `type` value and its own type-specific fields.

Goal: Practice choosing which fields are shared across all types (belong outside any per-type structure) versus which are genuinely type-specific.

→ Solution

Challenge 2: Decide — Bucket or Not?

A fitness app logs one heart-rate reading per user every 5 seconds during a workout. Should this use the bucket pattern? Justify your answer, and if yes, propose a reasonable bucket boundary (e.g. per-workout, per-minute).

Goal: Practice recognizing the "many tiny, frequent documents" signal that motivates the bucket pattern, and picking a boundary that fits the data's natural shape.

→ Solution

Challenge 3: Spot the Embedding Refactor

A `blog_posts` collection embeds comments directly (per `mongodb1-1`'s Challenge 2). One post has gone viral and now has 40,000 comments. What breaks, and what should the schema change to?

Goal: Practice applying this chapter's "time to refactor to referencing" signal to a concrete, previously-reasonable embedding decision that scale has broken.

→ Solution

⚠️ Gotcha: "Bucket Pattern" and $bucket Are Two Different Things

It's easy to conflate this chapter's bucket **pattern** (a schema design decision made when you insert data) with Chapter 2's `$bucket` aggregation **stage** (a query-time grouping operation on data that's already stored). A collection built with the bucket pattern doesn't require using `$bucket` to query it, and using `$bucket` in a pipeline says nothing about how the underlying documents were designed. They share a name because both group things by range — that's the only real connection.

🎯 What's Next

The next chapter is **Transactions in MongoDB** — multi-document ACID transactions, when a document-model database actually needs them despite embedding reducing the need in the first place, and the session-based API that makes them work.

🔒 Transactions in MongoDB

Chapter 3 ended with a signal for when embedding breaks down and referencing takes over. Referencing has a cost beyond needing `$lookup` to join data back together: once related data lives in **separate documents**, a single logical operation can span more than one of them — and MongoDB needs a way to make sure they change together, or not at all. That's what transactions are for.

Why Transactions? The Multi-Document Problem

A write to a **single** document in MongoDB is already atomic by default — an `updateOne()` either fully applies or doesn't happen at all, with no built-in transaction needed. That's part of why embedding (Chapter 3, `mongodb1-5`) is so attractive: an embedded post-with-comments update touches exactly one document, so it's automatically all-or-nothing.

The problem shows up once an operation legitimately needs to change **multiple** documents — possibly in different collections — as one unit. Chapter 3's comment-count refactor is a direct example: after splitting comments into their own collection, "add a comment" now means *two* separate writes — inserting the new comment document, and incrementing the post's precomputed `comment_count` field. Without a transaction, a crash between those two writes leaves the comment saved but the count wrong — the two documents have drifted out of sync.

ACID, Recap

The same four guarantees SQL transactions provide (`mysql_foundations_10`) apply here: **Atomicity** (all writes in the transaction succeed, or none do), **Consistency** (the database moves from one valid state to another), **Isolation** (concurrent transactions don't see each other's half-finished work), and **Durability** (once committed, it survives a crash). MongoDB's multi-document transactions genuinely provide all four — this isn't a partial or "best effort" version of ACID.

The Session-Based API

A transaction is tied to a **session** — every operation that should be part of the transaction has to explicitly run inside that session. The three core steps: `startSession()`, `startTransaction()`, then either `commitTransaction()` on success or `abortTransaction()` on failure.

```
const session = client.startSession();

try {
  session.startTransaction();

  await accounts.updateOne(
    { _id: "A1" },
    { $inc: { balance: -100 } },
    { session }
  );

  await accounts.updateOne(
    { _id: "A2" },
    { $inc: { balance: 100 } },
    { session }
  );

  await session.commitTransaction();
} catch (error) {
  await session.abortTransaction();
  throw error;
} finally {
  session.endSession();
}
```

Every operation that should be part of the transaction must pass `{ session }` explicitly — an operation that forgets it runs entirely outside the transaction, unprotected by any of its guarantees. If either update throws (say, account A1 doesn't have enough balance and application logic rejects it), `abortTransaction()` rolls back **both** writes — the debit never happened, either, even though it ran first and technically "succeeded" on its own.

withTransaction(): The Safer Shorthand

The driver also provides `withTransaction()`, which wraps the try/commit/catch/abort boilerplate above **and** automatically retries the whole callback on certain transient errors (see this chapter's closing gotcha) — the recommended approach for production code:

```
const session = client.startSession();

try {
  await session.withTransaction(async () => {
    await accounts.updateOne({ _id: "A1" }, { $inc: { balance: -100 } }, { session });
    await accounts.updateOne({ _id: "A2" }, { $inc: { balance: 100 } }, { session });
  });
} finally {
  session.endSession();
}
```

Chapter 3's Comment Refactor, Without vs. With a Transaction

Without a Transaction

Insert the comment, then increment `comment_count`, as two independent writes. A crash (or thrown error) between them leaves the comment saved but the count under-reporting it — permanently, until something notices and fixes it.

With a Transaction

Both writes run inside one `withTransaction()` block. Either the comment is inserted **and** the count is incremented, or neither happens — the two documents can never be observed out of sync with each other.

When You Actually Need Transactions

Skip Transactions When

The operation only touches one document — embedding (Chapter 3) already made this the common case on purpose, and a single-document write is atomic for free, with no transaction overhead at all.

Use Transactions When

An operation must change multiple documents — possibly across collections — and it would be genuinely wrong for some of those writes to succeed while others fail: a funds transfer, an order that must decrement inventory as it's created, a referenced comment-count update.

Method	Purpose
`startSession()`	Creates a session — the container a transaction is scoped to
`startTransaction()`	Begins a transaction on that session
`commitTransaction()`	Applies every write made in the transaction, atomically
`abortTransaction()`	Rolls back every write made in the transaction so far
`withTransaction(fn)`	Runs `fn` inside a transaction, auto-retrying on transient errors, auto-committing/aborting

💻 Coding Challenges

Challenge 1: Identify the Multi-Document Risk

An e-commerce checkout needs to (a) insert a new `order` document and (b) decrement the ordered product's `stock` field in a separate `products` collection. Explain, in terms of this chapter's ACID recap, what could go wrong without a transaction.

Goal: Practice articulating the specific failure mode (partial, inconsistent writes) a transaction is meant to prevent, not just naming "atomicity" abstractly.

→ Solution

Challenge 2: Write the Checkout Transaction

Using this chapter's `withTransaction()` pattern, write the session-based code for Challenge 1's checkout: insert the order, then decrement the product's stock, both inside one transaction.

Goal: Practice the full session lifecycle — `startSession()`, passing `{ session }` to every operation, and `endSession()` in a `finally` block.

→ Solution

Challenge 3: Transaction or Not?

Decide whether each of these needs a transaction, and justify why: (a) updating a single user's `last_login` timestamp, (b) moving a task between two different users' embedded `tasks` arrays inside their own user documents, (c) transferring loyalty points between two separate `customer_accounts` documents.

Goal: Practice applying the "single document vs. multiple documents" test directly, including the trickier case (b) where two *different top-level documents* are each individually atomic on their own.

→ Solution

⚠️ Gotcha: Transactions Require a Replica Set, and Aren't Free

Multi-document transactions only work against a **replica set** (or sharded cluster) — a standalone single-node MongoDB instance cannot run them at all, which surprises developers testing locally against a bare `mongod`. Transactions also carry real performance overhead compared to independent single-document writes, and can hit **transient errors** (network blips, transaction conflicts) that are expected to be retried, not treated as hard failures — which is exactly why `withTransaction()`, with its built-in retry logic, is the recommended API over hand-rolling `startTransaction`/`commitTransaction` yourself. Keep transactions short and touching as few documents as reasonably possible — they are the exception for cross-document atomicity, not a default way to write MongoDB code.

🎯 What's Next

The next chapter is **Replication** — the replica sets that transactions turn out to require, how MongoDB elects a primary, automatic failover, and how read preference lets an application choose where its reads come from.

🔁 Replication

Chapter 4's closing gotcha mentioned transactions requiring a **replica set** — a standalone `mongod` can't run them. This chapter explains what a replica set actually is, why it's the foundation transactions (and much of MongoDB's real-world resilience) depend on, and how to control where an application's reads actually come from.

What Is a Replica Set?

A **replica set** is a group of `mongod` instances holding copies of the same data. Exactly one member is the **primary** — the only one that accepts writes — while the rest are **secondaries**, continuously replicating the primary's changes so they stay (nearly) in sync. If the primary becomes unavailable, the replica set can elect a new primary from among the secondaries automatically, without a human intervening.

```
// Connecting to a replica set names every member — the driver figures out who's currently primary
mongodb://host1:27017,host2:27017,host3:27017/mydb?replicaSet=rs0
```

The Oplog: How Replication Actually Works

The primary records every write operation into a special capped collection called the **oplog** (operations log). Each secondary continuously reads ("tails") the primary's oplog and re-applies the same operations against its own copy of the data, in the same order they happened on the primary. This is why secondaries are described as "nearly" in sync rather than instantly — there's always some small amount of **replication lag** between a write landing on the primary and that same write showing up on a given secondary.

Elections & Automatic Failover

When the current primary becomes unreachable — a crash, a network partition, planned maintenance — the remaining members hold an **election** to choose a new primary, without any write ever being manually redirected by a person. A candidate needs a **majority** of the replica set's voting members to agree before it becomes primary, which is exactly why replica sets are normally deployed with an **odd** number of voting members (3, 5, 7…) — an even number risks a tie that no majority can break.

```
// Checking replica set health and who's currently primary
rs.status()

// The current primary's own view of the set — includes each member's role and health
rs.isMaster()
```

This is the mechanism Chapter 4's transactions actually rely on for durability: a transaction committed with a **majority** write concern isn't considered safely committed until a majority of replica set members have acknowledged it — meaning even if the primary immediately crashed afterward, an election would promote a secondary that already has that committed data.

Read Preference: Where Reads Actually Go

By default, every read goes to the **primary** — the same place every write goes, guaranteeing a read always sees the absolute latest data. **Read preference** lets an application redirect reads elsewhere instead, trading some staleness risk for benefits like spreading read load across more machines.

Read Preference	Behavior
`primary` (default)	Always reads from the primary; always up to date; no read scaling benefit
`primaryPreferred`	Reads from the primary normally, falls back to a secondary if the primary is unreachable
`secondary`	Always reads from a secondary; never touches the primary; may be slightly stale
`secondaryPreferred`	Prefers a secondary, falls back to the primary if none are available
`nearest`	Reads from whichever member has the lowest network latency, primary or secondary

Primary Reads vs. Secondary Reads

Reading From the Primary

Always current, since it's the same place every write lands. The right choice whenever a stale read would actually be wrong — a bank balance, an inventory count right before checkout.

Reading From a Secondary

Offloads read traffic away from the primary — useful for reporting/analytics queries that can tolerate being a few seconds behind, at the cost of that unavoidable replication lag.

Route to a Secondary When

The read is for analytics, reporting, or a dashboard that's acceptable to be a little behind real-time — offloading it keeps that load off the primary, which is busy serving every write.

Keep Reading From the Primary When

The application just wrote data and immediately needs to read it back (read-your-own-writes), or the read feeds a decision where staleness would cause a real, visible bug.

Setting Read Preference in the Node.js Driver

```
const reports = client.db("shop").collection("orders");

// This one query reads from a secondary; everything else on this client still defaults to primary
const monthlyTotals = await reports
  .aggregate([ /* ...grouping pipeline... */ ], { readPreference: "secondaryPreferred" })
  .toArray();
```

💻 Coding Challenges

Challenge 1: Explain the Odd-Number Rule

A team proposes a 4-member replica set for extra redundancy. Explain, in terms of how elections work, why an odd number of voting members (3 or 5) is the better choice.

Goal: Practice connecting the majority-vote election mechanism to a concrete deployment decision.

→ Solution

Challenge 2: Choose a Read Preference

For each, name the most appropriate read preference and justify it: (a) a nightly sales report query, (b) reading a user's profile immediately after they update it, (c) a customer-facing "check my order status" page.

Goal: Practice applying the primary-vs-secondary trade-off to realistic, differently-sensitive read scenarios.

→ Solution

Challenge 3: Trace a Failover

A 3-node replica set's primary crashes. Walk through, step by step, what happens next — up to and including the application's writes succeeding again.

Goal: Practice narrating the full election-and-failover sequence in the right order, including the brief window where writes aren't yet possible.

→ Solution

⚠️ Gotcha: Secondary Reads Are Eventually Consistent, Not Instantly Consistent

Reading from a secondary means accepting **replication lag** — under normal conditions it's milliseconds, but a slow network, heavy write load, or a struggling secondary can widen that gap noticeably. An application that writes data and then immediately reads it back from a secondary can genuinely fail to see its own write yet — a surprising, hard-to-reproduce bug if the read preference wasn't a deliberate choice. Default to `primary` unless there's a specific, understood reason (like this chapter's reporting example) to read from a secondary instead.

🎯 What's Next

The next chapter is **Sharding** — what happens when even a well-replicated dataset outgrows a single primary's capacity: shard keys, choosing a good one, chunks, the balancer, and when sharding is (and isn't) the right call.

🔗 Sharding

Chapter 5's replica sets solve for redundancy and read scaling — but every member still stores a **full copy** of the entire dataset, and every write still passes through **one** primary. Once the data itself, or the write load, outgrows what a single primary can hold or handle, replication alone stops being enough. **Sharding** is MongoDB's answer: splitting the data itself across multiple machines, each holding only a slice of it.

Why Shard? The Ceiling Replication Doesn't Remove

Replication scales *reads* (Chapter 5's read preference) and provides failover, but it doesn't scale *writes* or *storage* — a 10-node replica set still has exactly one primary accepting writes, and every node still needs enough disk to hold 100% of the data. Sharding distributes the data **horizontally**: each shard holds a portion of the total dataset, so total storage and total write throughput both grow as shards are added.

The Building Blocks of a Sharded Cluster

Component	Role
Shard	Holds a portion of the data; in production, each shard is itself a full replica set (Chapter 5) for redundancy
mongos	The query router applications actually connect to; routes each operation to the correct shard(s)
Config servers	Store the cluster's metadata — which shard key ranges live on which shard

An application talks to `mongos` exactly as if it were a single `mongod` — the routing to the correct shard(s) happens transparently underneath.

The Shard Key

The **shard key** is the field (or fields) chosen when a collection is sharded, and it determines which shard each document lives on. It's chosen **once**, when sharding a collection, and changing it afterward is a genuinely heavy operation (resharding) — getting it right up front matters far more here than most schema decisions covered so far in this course.

```
sh.shardCollection("shop.orders", { customer_id: "hashed" })
```

Choosing a Good Shard Key

Three properties make a shard key good:

- **High cardinality** — many distinct possible values, so documents can actually spread across many shards rather than piling onto just a few.
- **Even write distribution** — new writes should land across shards roughly evenly, not concentrate on one.
- **Query isolation** — common queries should ideally be answerable by consulting just one (or a few) shards, not every shard in the cluster.

A Bad Shard Key vs. a Better One

Monotonically Increasing (e.g. `_id` or a timestamp)

Every new document's key value is higher than the last. All new writes land in the same range — and therefore the same shard — creating a **hotspot** while other shards sit idle. Sharding on this field barely helps write throughput at all.

```
{ customer_id: "hashed" }
```

Hashing scrambles the key's natural order before distributing it, so consecutive writes (even from the same fast-growing range of underlying IDs) land essentially randomly across shards — spreading write load evenly, at the cost of range queries no longer targeting a single shard.

Chunks & the Balancer

Within a sharded collection, data is split into **chunks** — contiguous ranges of shard key values. As chunks grow past a size threshold, MongoDB splits them further. The **balancer** is a background process that automatically migrates chunks between shards whenever their distribution becomes uneven — keeping roughly the same number of chunks (and therefore roughly the same amount of data) on every shard, without a human manually moving anything.

Sharding Is the Right Call When

A single replica set's primary genuinely can't hold the working data set in memory, or can't keep up with write throughput, even after the schema and indexing work from earlier chapters.

Sharding Isn't Yet Needed When

The dataset comfortably fits on one well-resourced replica set — sharding adds real operational complexity (choosing a shard key you're stuck with, managing `mongos`/config servers) that isn't worth paying for before it's actually necessary.

A Query That Benefits From Query Isolation

With `customer_id` as the shard key, a query filtering by `customer_id` can be routed directly to the one shard holding that customer's data:

```
db.orders.find({ customer_id: "C104" }) // targeted — hits one shard

db.orders.find({ status: "completed" }) // scatter-gather — hits every shard, since status isn't the shard key
```

💻 Coding Challenges

Challenge 1: Spot the Hotspot

A team proposes sharding an `events` collection on `created_at` (a timestamp) so events can be range-queried by date efficiently. What goes wrong with writes, and why?

Goal: Practice recognizing the monotonically-increasing-key hotspot pattern in a new example, not just the `_id` case from this chapter.

→ Solution

Challenge 2: Explain Query Isolation

A collection is sharded on `tenant_id` (a multi-tenant SaaS app, one value per customer organization). Explain why a query filtering by `tenant_id` is fast, and why a query that omits it is slow.

Goal: Practice explaining scatter-gather versus targeted routing in terms of the shard key actually appearing (or not) in the query filter.

→ Solution

Challenge 3: Shard or Not?

A company's product catalog has 50,000 documents and comfortably runs on a single 3-node replica set with room to spare. Should they shard it now, in anticipation of future growth? Justify your answer using this chapter's decision criteria.

Goal: Practice resisting the urge to shard preemptively, weighing real operational complexity against a need that doesn't yet exist.

→ Solution

⚠️ Gotcha: A Query Without the Shard Key Becomes Scatter-Gather

If a query's filter doesn't include the shard key, `mongos` has no way to know which shard holds the matching documents — it has to send the query to **every** shard and merge the results, a **scatter-gather** query. This isn't a hard error, just a much slower path than a query that targets one shard directly — and it's easy to end up here by accident if the fields an application actually filters by most often weren't the ones considered when the shard key was chosen. Picking a shard key genuinely means picking it around the queries the application runs most, not just around "spreads writes evenly."

🎯 What's Next

The next chapter is **Performance & Security** — revisiting index strategy with everything this course has covered since `mongodb1-7`, the database profiler, authentication/authorization, and connection string security (tying back to the site's Database Security course).

🛡️ Performance & Security

This chapter has two halves that both matter before a MongoDB deployment is genuinely production-ready: revisiting `mongodb1-7`'s index strategy with everything this course has since added (aggregation pipelines, sharding), and securing the deployment itself — authentication, authorization, and the connection string that ties an application to it.

Index Strategy, Revisited

`mongodb1-7` introduced the **Equality-Sort-Range** (ESR) rule for ordering compound index fields, and reading `explain()` output to check whether a query used an index at all. Both apply just as much to this course's additions:

- An aggregation's `$match` stage benefits from an index exactly like a `find()` filter does — which is the concrete reason Chapter 1 and Chapter 2 both stressed putting `$match` first.
- A `$sort` stage can also use an index (avoiding an expensive in-memory sort) if it comes right after a `$match` that already narrowed things down using the same compound index, following the same ESR ordering.
- On a **sharded** collection (Chapter 6), the shard key is implicitly part of how data is located — a query including the shard key can be routed to one shard *and* use a local index on that shard, while a query missing it pays the scatter-gather cost regardless of indexing.

The Database Profiler

Where `explain()` checks one query you already suspect is slow, the **profiler** watches everything and records slow operations automatically — the tool for finding problems you didn't already know to look for.

```
// Log any operation slower than 100ms into the system.profile collection
db.setProfilingLevel(1, { slowms: 100 })

// Review what got logged, most recent first
db.system.profile.find().sort({ ts: -1 }).limit(10)
```

Profiling level `1` logs only slow operations (the practical default); level `2` logs *everything*, which is useful briefly while debugging but far too much overhead to leave on in production. Once a slow query is found in `system.profile`, the same `explain()` workflow from `mongodb1-7` diagnoses exactly why.

Authentication & Authorization

MongoDB doesn't require authentication by default on an out-of-the-box install — enabling it is a deliberate, necessary step, not something to assume already happened.

```
// mongod.conf
security:
 authorization: enabled
```

Once enabled, every connection needs a user with specific granted **roles** — MongoDB's authorization model, built around the same least-privilege idea as choosing which SQL grants an application account actually needs.

Built-in Role	Grants
`read`	Read-only access to a specific database
`readWrite`	Read and write access to a specific database
`dbAdmin`	Administrative tasks (indexes, stats) on a specific database, no data read/write
`clusterAdmin`	Cluster-wide administrative actions (replica set/sharding config)
`root`	Superuser — every privilege on every database; reserved for true administrators, never an application

Creating a Scoped Application User

```
db.createUser({
 user: "shop_app",
 pwd: passwordPrompt(),
 roles: [ { role: "readWrite", db: "shop" } ]
})
```

This user can read and write the `shop` database only — it has no ability to touch any other database, drop the database, or perform cluster administration, even if the application's own code somehow tried to.

An Overly Broad Role vs. a Scoped One

root, or readWriteAnyDatabase

Convenient during early development, but means a single compromised application credential (a leaked connection string, an injection bug) grants an attacker access to **every** database on the server, not just this one application's.

`readWrite` Scoped to One Database

The same compromised credential now only exposes the one database that application was ever supposed to touch — the blast radius of a leak is contained by design, not by luck.

Connection String Security

A MongoDB connection string routinely contains a username and password directly in its text: `mongodb+srv://user:password@cluster.example.net/mydb`. Treat it exactly like any other credential — never commit one into source control, matching the CI/CD Pipelines course's core rule that a committed credential is compromised forever the moment it's pushed, even if the commit is later removed. Load it from an environment variable or secrets manager instead, the same pattern `node2-6` covered for configuration generally.

```
// Node.js — read from the environment, never hardcode
const client = new MongoClient(process.env.MONGODB_URI);
```

The connection string is also where TLS gets enabled for encryption in transit (`tls=true`), and where `mongodb+srv://` (versus plain `mongodb://`) resolves the cluster's real hosts via DNS automatically — worth knowing what these options mean rather than copy-pasting a string a hosting provider generated.

Do

Enable authorization, create per-application users scoped to exactly the database(s) they need, load connection strings from environment variables or a secrets manager, use TLS in transit.

Don't

Run with authorization disabled, share one `root` credential across every application, commit a connection string to a repository, or expose a database port directly to the public internet.

💻 Coding Challenges

Challenge 1: Diagnose a Profiler Entry

The profiler logs a slow `find({ email: "..." })` query on a 2-million-document `users` collection with no matching index. Using `mongodb1-7`'s tools, what's the fix, and what would `explain()` have shown before and after?

Goal: Practice connecting a profiler finding back to the indexing chapter's diagnostic workflow.

→ Solution

Challenge 2: Design Least-Privilege Roles

A company has a `shop` app that needs full read/write on a `shop` database, and a separate nightly `reporting` job that only ever reads from it. Design the `createUser()` roles for each.

Goal: Practice applying least privilege to two genuinely different access needs sharing the same database, rather than granting both the same broad role.

→ Solution

Challenge 3: Find the Security Mistake

A developer commits a file containing `const uri = "mongodb+srv://admin:Sup3rSecret@prod-cluster.example.net/shop";` directly into the application's Git repository. Explain everything wrong with this, and how it should be fixed.

Goal: Practice spotting a hardcoded, over-privileged, committed credential as three separate, stacking mistakes — not just one.

→ Solution

⚠️ Gotcha: An Open, Unauthenticated MongoDB Was a Real, Repeated Ransomware Target

This isn't a hypothetical risk — a well-known wave of ransomware attacks specifically targeted MongoDB instances left exposed to the public internet with authentication disabled, scanning for open `27017` ports, wiping the data, and leaving a ransom note in its place. Modern MongoDB versions bind to `localhost` by default specifically to make an accidental public exposure harder, but that default can still be overridden — treat "authorization enabled" and "not reachable from the open internet" as two separate requirements, not one, since either one alone still leaves a real door open.

🎯 What's Next

The final chapter is the **Capstone: Designing a Real Application's Data Model** — combining schema design, indexing, aggregation, and driver code from every chapter in this course into one small, real application built from scratch.

🏁 Capstone: Designing a Real Application's Data Model

Every chapter in this course has reused the same running example — customers, orders, products. This capstone builds a small, real **order-processing system** around them, combining schema design (Ch.3), indexing (Ch.7 + `mongodb1-7`), the aggregation framework (Ch.1–2), transactions (Ch.4), and driver code (`mongodb1-8`) into one coherent application, end to end.

The Application: A Small Order-Processing System

Requirements: customers place orders for products; each order must record the price paid at that moment (not today's price, which may have changed since); placing an order must safely decrement stock and never double-charge or double-decrement; and the business needs a dashboard showing revenue by category and top-selling products.

Step 1: Schema Design (Chapter 3)

Applying Chapter 3's patterns deliberately, not by default:

```
// An order EMBEDS a snapshot of what was bought — price/name at time of purchase,
// not a live reference, since a later price change must never alter a past order
{
 _id: ObjectId("..."),
 customer_id: "C104",
 status: "completed",
 items: [
  { product_id: "P200", name: "Wireless Mouse", category: "electronics", price_paid: 24.99, qty: 1 }
 ],
 total: 24.99,
 createdAt: ISODate("2026-07-06")
}
```

The **computed pattern** also applies: each customer document carries a precomputed `lifetime_spend`, updated incrementally on every order rather than re-aggregated on every profile view — exactly Chapter 3's page-view-counter idea, applied to a running total.

Step 2: Indexes (Chapter 7 / mongodb1-7)

The queries this app actually runs decide the indexes it needs — not guesswork:

```
db.orders.createIndex({ customer_id: 1, createdAt: -1 }) // "this customer's orders, newest first"
db.orders.createIndex({ status: 1 })                      // the $match in the dashboard pipeline below
```

Step 3: The Transaction — Placing an Order (Chapter 4)

Placing an order touches **three** separate documents — a new order, the product's stock, and the customer's precomputed `lifetime_spend` — which is exactly the multi-document signal Chapter 4 flagged as needing a transaction, not three independent writes.

```
const session = client.startSession();

try {
  await session.withTransaction(async () => {
    await orders.insertOne(newOrder, { session });

    await products.updateOne(
      { _id: "P200" },
      { $inc: { stock: -1 } },
      { session }
    );

    await customers.updateOne(
      { _id: "C104" },
      { $inc: { lifetime_spend: newOrder.total } },
      { session }
    );
  });
} finally {
  session.endSession();
}
```

If the stock decrement fails (say, application logic rejects it because stock is already 0), the whole transaction aborts — the order is never left "placed" against a customer whose spend total was never actually updated.

Step 4: The Aggregation — A Sales Dashboard (Chapters 1 & 2)

The dashboard needs two different views in one call — a job for `$facet` — starting from a `$match` that can use the `status` index created in Step 2, per this course's own indexing-order gotcha (Chapter 1 and revisited in Chapter 7):

```
db.orders.aggregate([
 { $match: { status: "completed" } },
 { $unwind: "$items" },
 { $facet: {
    revenueByCategory: [
     { $group: { _id: "$items.category", revenue: { $sum: "$items.price_paid" } } },
     { $sort: { revenue: -1 } }
    ],
    topProducts: [
     { $group: { _id: "$items.name", unitsSold: { $sum: "$items.qty" } } },
     { $sort: { unitsSold: -1 } },
     { $limit: 5 }
    ]
 } }
])
```

This chains together **every** stage this course covered: `$match` (Ch.1) using an index, `$unwind` (Ch.2) to break each order's embedded items array into individual rows, and `$facet` (Ch.2) running two independent grouped-and-sorted views from that same unwound input in a single round trip.

Step 5: Scaling Considerations (Chapters 5 & 6)

Not needed on day one, but worth designing with in mind: the dashboard aggregation is a read-heavy reporting query, a good candidate for `secondaryPreferred` read preference (Chapter 5) once the replica set is under real load. If `orders` ever outgrows a single replica set, `customer_id` — already indexed, already the natural per-customer query boundary — is the obvious shard key candidate (Chapter 6), hashed to avoid the write hotspot a raw, sequential `_id` or timestamp key would create.

Course Concept	Where It's Used in This App
Embedding vs. referencing (Ch.3)	Order items embed a price/name snapshot; customers are referenced by ID
Computed pattern (Ch.3)	`customers.lifetime_spend`, updated via `$inc` on every order
Compound indexes (Ch.7)	`{ customer_id, createdAt }` and `{ status }`, matching real queries
Transactions (Ch.4)	Placing an order: insert + stock decrement + spend increment, atomically
$match / $unwind / $facet (Ch.1–2)	The dashboard's revenue-by-category and top-products aggregation
Read preference (Ch.5)	Dashboard reads candidate for `secondaryPreferred` at scale
Shard key choice (Ch.6)	Hashed `customer_id`, if/when `orders` needs sharding

💻 Coding Challenges

Challenge 1: Add a Cancellation Path

Design the transaction for cancelling an order: it must set the order's `status` to `"cancelled"`, restore the product's stock, and decrement the customer's `lifetime_spend` — all atomically.

Goal: Practice recognizing that a cancellation is the same multi-document-atomicity problem as placing an order, just reversed.

→ Solution

Challenge 2: Extend the Dashboard Pipeline

Add a third facet to Step 4's aggregation: `ordersPerDay`, counting completed orders grouped by calendar day.

Goal: Practice adding a third independent sub-pipeline to an existing `$facet` stage without disturbing the other two.

→ Solution

Challenge 3: Justify the Embedding Choice

A colleague suggests referencing the product instead of embedding a `name`/`price_paid` snapshot in each order item, to "keep it DRY." Explain, using this course's material, why that would actually introduce a bug.

Goal: Practice defending a specific schema design decision against a plausible-sounding but incorrect simplification.

→ Solution

⚠️ Gotcha: Pieces That Work Alone Don't Automatically Work Together

Each step above was verified against this course's own earlier examples — but combining them is still worth testing end to end, not assumed to compose correctly. A concrete example: if `orders` is later sharded on `customer_id` (Step 5), the Step 3 transaction now spans documents that may live on *different* shards (an order and a product don't share the shard key) — MongoDB's distributed transactions support this, but with added latency and failure modes single-shard transactions don't have. Design decisions made independently, chapter by chapter, still need to be re-verified together once the whole system is assembled — the same discipline this course's earlier capstone-adjacent chapters (Ch.4's transaction retries, Ch.6's scatter-gather gotcha) each flagged on their own.

🎉 Course Complete

That's the full **MongoDB Intermediate/Advanced** course — from the aggregation framework through schema design at scale, transactions, replication, sharding, performance and security, and finally a real application combining all of it. Together with **MongoDB Fundamentals**, this completes both MongoDB courses on the site.