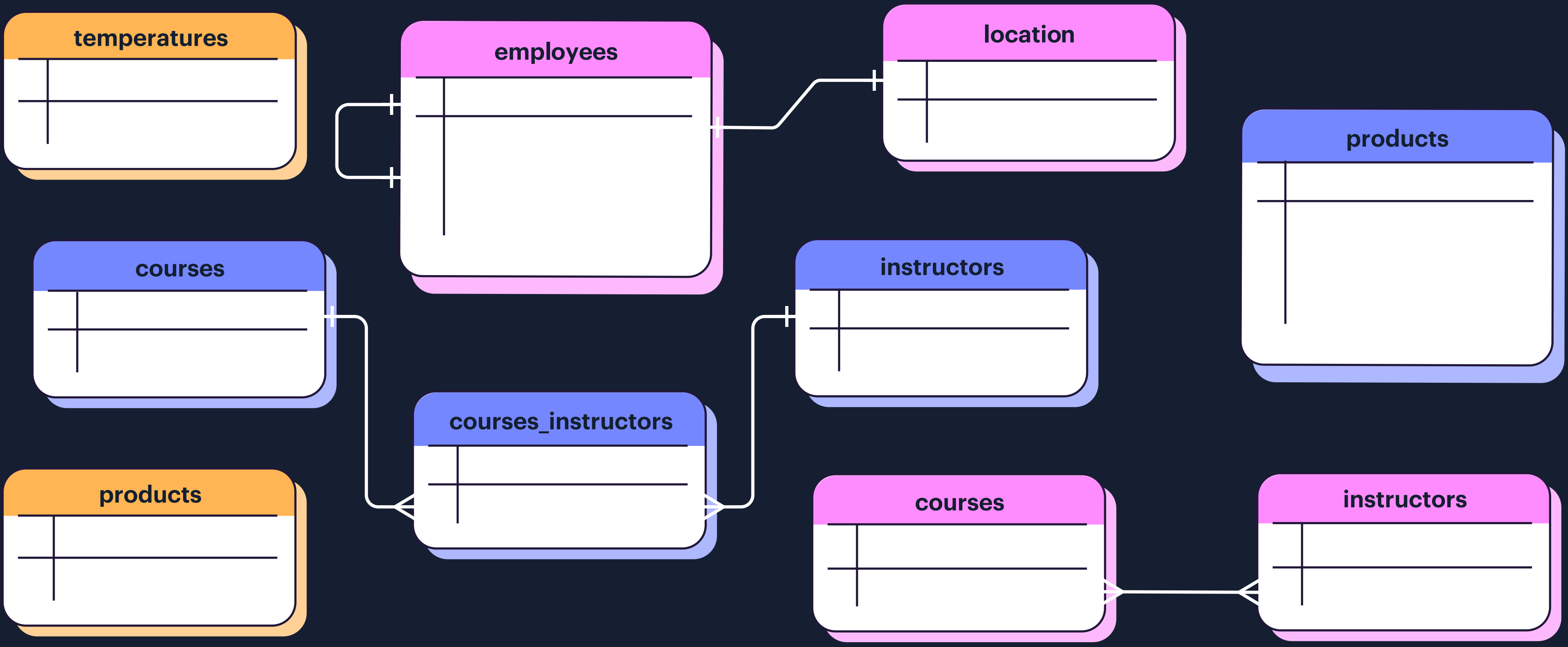
E-Book



8 Data Modeling Patterns in Redis

redis

Table of Contents

Before you Get Started

If you want to follow along with some of the code examples used in this e-book, you can clone the GitHub repository.

Introduction

When someone is looking to use NoSQL for an application, the question that most often comes up is, "How do I structure my data?" The short answer to this question is, as you might guess, it depends. There are several questions that can inform how to structure your data in a NoSQL database.

Is your application read heavy, or write heavy? What does the user experience of your application look like? How does your data need to be presented to the user? How much data will you be storing? What performance considerations do you need to account for? How do you anticipate scaling your application?

These questions are only a small subset of what you need to ask yourself when you start working with NoSQL. A common misconception with NoSQL databases is that since they are "schemaless" you don't need to worry about your schema. In reality, your schema is incredibly important regardless of what database you choose. You also need to ensure that the schema you choose will scale well with the database you plan to use.

In this e-book you will learn how to approach data modeling in NoSQL, specifically within the context of Redis. Redis is a great database for demonstrating several NoSQL patterns and practices. Not only is Redis commonly used and loved by developers, it also is a multi-model database. This means that while many of the patterns covered in this e-book apply to different types of databases (e.g. document, graph, time series, etc.), with Redis you can apply all of the patterns in a single database.

By the end of this, you should have

- A firm understanding of how to approach modeling data in Redis as well as in NoSQL generally.
- An understanding of several NoSQL data modeling patterns, their pros and cons, as well as use cases for them in practice.
- A working knowledge of how to actually write code (with examples) to take advantage of NoSQL patterns within Redis.

I'm sure you're eager to get started, so let's dive in!

SQL versus NoSQL

I'm sure at a certain level you understand the difference between SQL and NoSQL. SQL is a structured query language whereas NoSQL can mean several different things depending on the context. However, generally speaking, the approach to modeling data is fundamentally different in NoSQL than in SQL. There are also differences in terms of scalability, with NoSQL being easier to scale horizontally.

When building applications you are probably using an object-oriented language like JavaScript, Java, C#, or others. Your data is represented as strings, lists, sets, hashes, JSON, and so on. However, if you store data in a SQL database or a document database, you need to squeeze and transform the data into several tables or collections. You also need complex queries (such as SQL queries) to get the data out. This is called **impedance mismatch** and is the fundamental reason why NoSQL exists.

A large application might use other systems for data storage such as Neo4J for graph data, MongoDB for document data, InfluxDB for time series, etc. Using separate databases turns an impedance mismatch problem into a database orchestration problem. You have to juggle multiple connections to different databases, as well as learn the different client libraries used.

With Redis, in addition to the basic data structures such as strings, lists, sets, and hashes, you can also store advanced data structures such as JSON for documents, Search and Query for secondary indexing, RedisGraph for graph data, RedisTimeSeries for time-series data, and RedisBloom for probabilistic data (think leaderboards).

This reduces impedance mismatch because your data is stored in one of 15 structures with little or no transformations. You can also use a single connection (or connection pool) and client library to access your data. What you end up with is a simplified architecture with purpose-built models that are blazing fast and simple to manage. For this reason, this e-book will use Redis to explain several of the NoSQL data modeling patterns.

Most developers have at least a little understanding of SQL and how to model data in it. This is because SQL is widely used and there are several incredible books and even full courses devoted to it. NoSQL is quickly growing and becoming more popular. But given that when you're talking about NoSQL you're talking about more than just a document store, there is a lot of ground to cover. That's why when covering certain NoSQL data modeling patterns in this e-book, you will be presented with what it might look like to model the data in SQL as well.

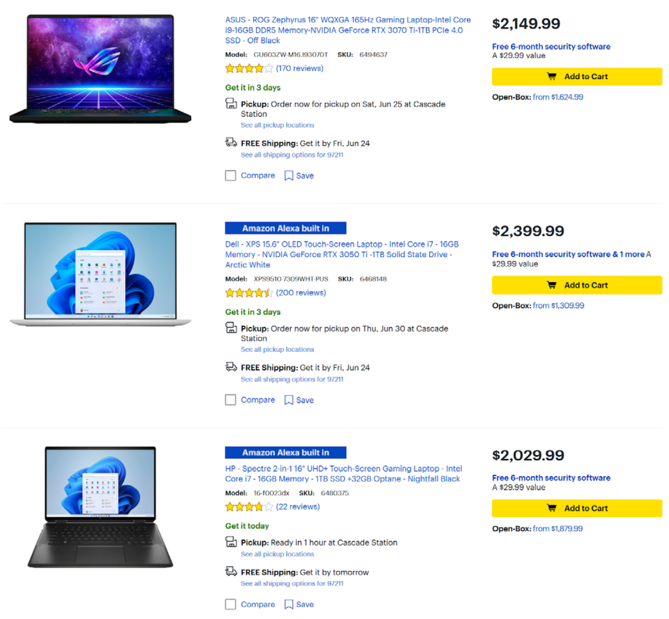
When you approach data modeling in SQL you are typically focused on relationships, as SQL is meant for set-based operations on relational data. NoSQL doesn't have this constraint and is more flexible in the way you model data. However, this can lead to schemas that are overly complex. When considering NoSQL schema design, always think about performance and try to keep things simple.

So to kick things off, let's start by looking at something that is very near and dear to a SQL developer's heart: **relationships**.

Modeling 1-to1 Relationships

Imagine that you are creating a retail app that sells electronics. Let's use **Picture 1** and **Picture 2** as an example of the UI for a standard retail e-commerce app. First, you'll create a list view of all the electronics and then a detailed view that shows all the details of each item. There is a 1-to-1 relationship between each item in the list view and the detailed view (shown in **Picture 2**) of the item. The detailed view shows all the details such as multiple photos, description, manufacturer, dimensions, weight, and so on.

Picture 1



ASUS - ROG Zephyrus 16" WQXGA 165Hz Gaming Laptop-Intel Core i9-16GB DDR5 Memory-NVIDIA GeForce RTX 3070 Ti-1TB PCIe 4.0 SSD - Off Black

Model: GU603ZW-M16.I9307OT **SKU:** 6494637

(170 reviews)

Get it in 3 days

Pickup: Order now for pickup on Sat, Jun 25 at Cascade Station
See all pickup locations

FREE Shipping: Get it by Fri, Jun 24
See all shipping options for 97211

Compare Save

$2,149.99

Free 6-month security software
A $29.99 value

Add to Cart

Open-Box: from $1,624.99

Amazon Alexa built in

Dell - XPS 15.6" OLED Touch-Screen Laptop - Intel Core i7 - 16GB Memory - NVIDIA GeForce RTX 3050 Ti -1TB Solid State Drive - Arctic White

Model: XPS9510-7309WHT-PUS **SKU:** 6468148

(200 reviews)

Get it in 3 days

Pickup: Order now for pickup on Thu, Jun 30 at Cascade Station
See all pickup locations

FREE Shipping: Get it by Fri, Jun 24
See all shipping options for 97211

Compare Save

$2,399.99

Free 6-month security software & 1 more A $29.99 value

Add to Cart

Open-Box: from $1,309.99

Amazon Alexa built in

HP - Spectre 2-in-1 16" UHD+ Touch-Screen Gaming Laptop - Intel Core i7 - 16GB Memory - 1TB SSD +32GB Optane - Nightfall Black

Model: 16-f0023dx **SKU:** 6480375

(22 reviews)

Get it today

Pickup: Ready in 1 hour at Cascade Station
See all pickup locations

FREE Shipping: Get it by tomorrow
See all shipping options for 97211

Compare Save

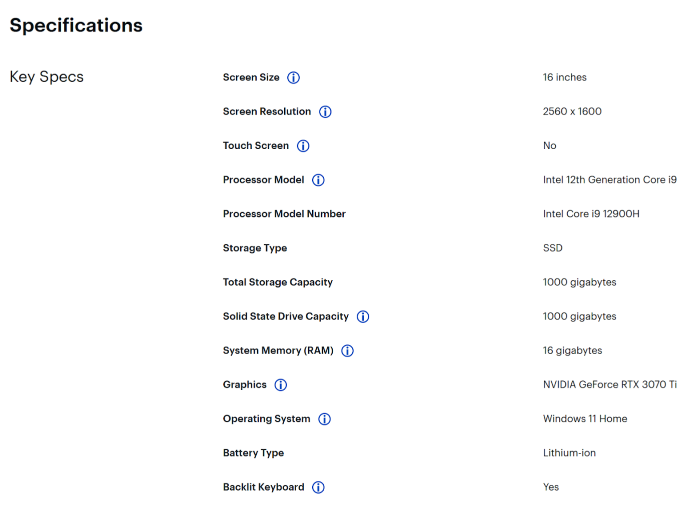
$2,029.99

Free 6-month security software
A $29.99 value

Add to Cart

Open-Box: from $1,879.99

Picture 2



Specifications

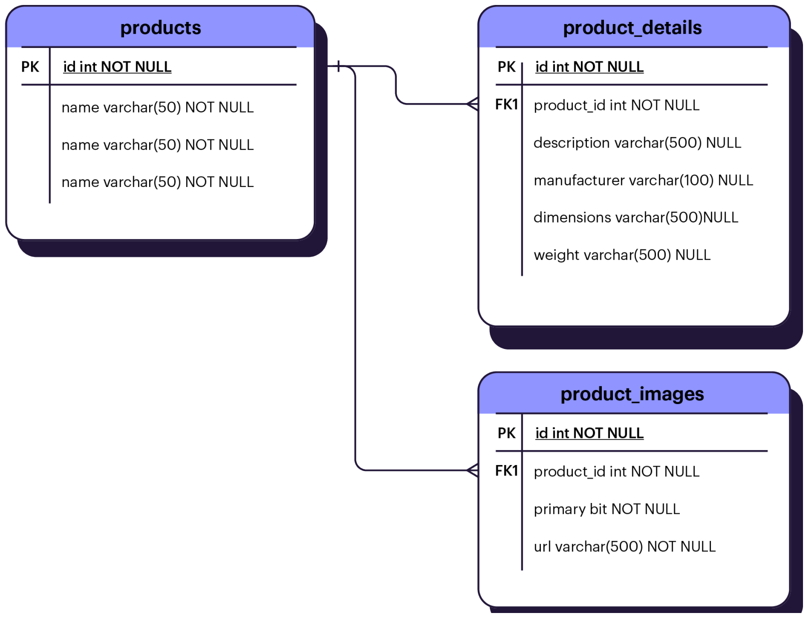
Key Specs	**Screen Size**	16 inches
	Screen Resolution	2560 x 1600
	Touch Screen	No
	Processor Model	Intel 12th Generation Core i9
	Processor Model Number	Intel Core i9 12900H
	Storage Type	SSD
	Total Storage Capacity	1000 gigabytes
	Solid State Drive Capacity	1000 gigabytes
	System Memory (RAM)	16 gigabytes
	Graphics	NVIDIA GeForce RTX 3070 Ti
	Operating System	Windows 11 Home
	Battery Type	Lithium-ion
	Backlit Keyboard	Yes

1-to-1 Relationships using SQL

In a relational database, you may create a table called **products** where each row holds just enough data to display the information in the list view. Then, you may create another table called **product_details** where each row holds the rest of the details. You would also need a **product_images** table, where you store all of the images for a product. You can see the entity relationship diagram in **Picture 3.**

Picture 3 depicts the entity relationships between **products, product_details, and product_images** and represents a normalized data model with a single denormalized field **image** in the **products** table. The reason for this is to avoid having to use a SQL JOIN when selecting the **products** for the list view. Using this model, the SQL query used to get the data needed for the listview might resemble **Code Example 1.**

Picture 3


products
PK id int NOT NULL
name varchar(50) NOT NULL
name varchar(50) NOT NULL
name varchar(50) NOT NULL
product_details
PK id int NOT NULL
FK1 product_id int NOT NULL
description varchar(500) NULL
manufacturer varchar(100) NULL
dimensions varchar(500)NULL
weight varchar(500) NULL
product_images
PK id int NOT NULL
FK1 product_id int NOT NULL
primary bit NOT NULL
url varchar(500) NOT NULL

Code Example 1

```
SELECT
   p.id, p.name, p.image, p.price, pi.url

FROM
   products p
```

1-to-1 Relationships using Redis

In Redis, similar to a relational database, you can create a collection called **products** and another called **product_details**. But with JSON you can improve this by simply embedding **product_images** and **product_details** directly into the **Products** collection. Then, when you query the **Products** collection, specify which fields you need based on which view you are trying to create.

This will allow you to easily keep all the data in one place. This is called the **Embedded Pattern** and is one of the most common patterns you will see in NoSQL document databases like JSON. **Code Example 2** uses Python and a client library called Redis OM (an ORM for Redis) to model **Products** and **ProductDetails**. Note that **ProductDetails** is embedded into **Products** directly, so all of the data for a product will be stored within the same document.

Code Example 2

```
class ProductDetail(EmbeddedJsonModel):
   description: str
   manufacturer: str
   dimensions: str
   weight: str
   images: List[str]

class Product(JsonModel):
   name: str = Field(index=True)
   image: str = Field(index=True)
   price: int = Field(index=True)
   details: Optional[ProductDetail]
```

Code Example 2 also shows how you can index fields using Redis OM and Search and Query. Doing this turns Redis into not only a document store but also a search engine since Search and Query enables secondary indexing and searching. When you create models using Redis OM, it will automatically manage secondary indexes with Search and Query on your behalf.

Using Redis OM we can write a function to retrieve our **products** list for the list view, as shown in **Code Example 3**.

Code Example 3

```
async def get_product_list():
    results = await connections \
        .get_redis_connection() \
        .execute_command(
            f'FT.SEARCH {Product.Meta.index_name} * LIMIT 0 10 RETURN 3 name image 
price'
        )

    return Product.from_redis(results)
```

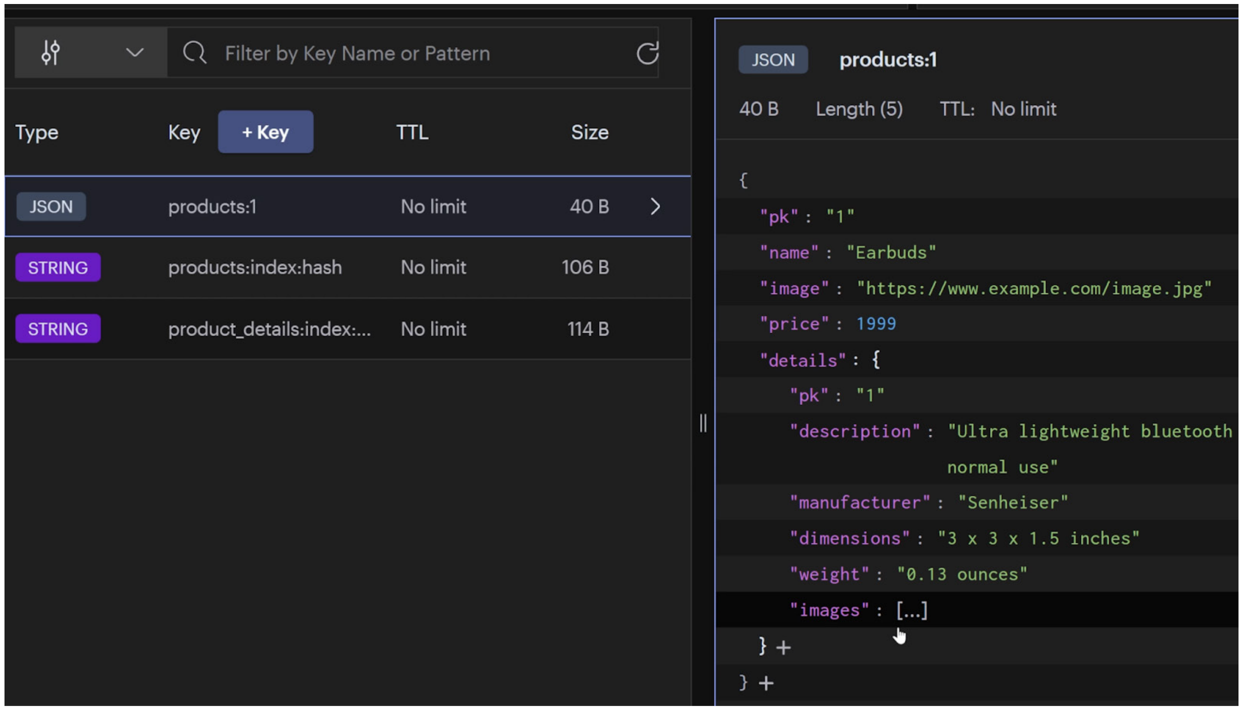
Notice that in **Code Example 3** we are using the FT.SEARCH (Search and Query) command, which specifies the index managed on our behalf by Redis OM and returns three fields: name, image, and price. While the documents all have details and images embedded, we don't want to display them in the list view so we don't need to query them. When we want the detailed view, we can query an entire Product document. See **Code Example 4** for how to query an entire document.

When using Redis, you can use RedisInsight as a GUI tool to visualize and interact with the data in your database. **Picture 4** shows you what a **Products** document looks like.

Code Example 4

```
async def get_product_details(product_id: str):
    return await Product.get(product_id)
```

Picture 4



Type	Key	TTL	Size
JSON	products:1	No limit	40 B
STRING	products:index:hash	No limit	106 B
STRING	product_details:index:...	No limit	114 B

JSON products:1

40 B Length (5) TTL: No limit

```
{
  "pk" : "1"
  "name" : "Earbuds"
  "image" : "https://www.example.com/image.jpg"
  "price" : 1999
  "details" : {
    "pk" : "1"
    "description" : "Ultra lightweight bluetooth
                     normal use"
    "manufacturer" : "Senheiser"
    "dimensions" : "3 x 3 x 1.5 inches"
    "weight" : "0.13 ounces"
    "images" : [...]
  } +
} +
```

Modeling 1-to-Many Relationships

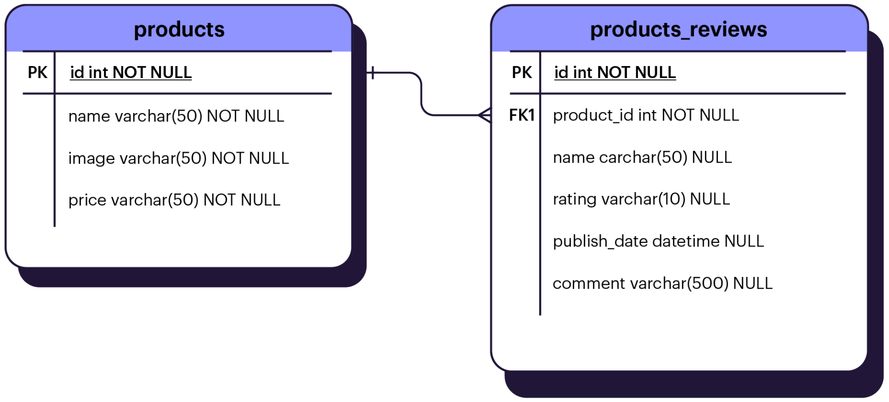
Revisiting our electronics e-commerce store example, let's talk about 1-to-many relationships. Let's imagine in the detailed view of a product you want to display a list of reviews for the product that show the reviewer name, rating, publish_date, and comment. This is a 1-to-many relationship because one product can have multiple reviews and a review can only relate to a single product.

1-to-Many Relationships using SQL

In a relational database, you would have a table called products and another table called product_reviews. Picture 5 shows the entity relationship diagram for products and product_reviews.

Using the entity relationship in **Picture 5**, you would need two SQL statements to get a product and its reviews. **Code Example 5** demonstrates what the SQL might look like. Your API would need to join the two queries together before sending the data to the client.

Picture 5


products
PK
id int NOT NULL
name varchar(50) NOT NULL
image varchar(50) NOT NULL
price varchar(50) NOT NULL
products_reviews
PK
id int NOT NULL
FK1
product_id int NOT NULL
name carchar(50) NULL
rating varchar(10) NULL
publish_date datetime NULL
comment varchar(500) NULL

Code Example 5

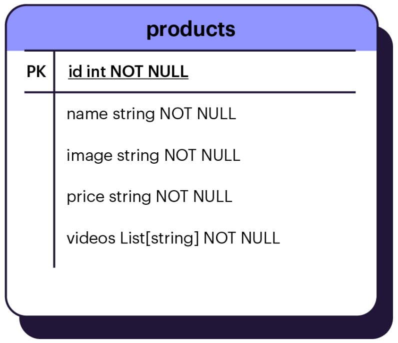
```
SELECT
   id, ‘name‘, ‘image‘, price
FROM
   products
WHERE
   id = 1;
SELECT
   ‘name‘ , rating, publish_date, comment
FROM
   product_reviews
WHERE
   product_id = 1;
```

1-to-Many Relationships using Redis

In Redis, similar to a relational database, you could create two collections called **products**, and **product_reviews** exactly like the entities above. This strategy (having two separate collections) works well for documents that are unbounded and can keep growing.

For example, a product could have hundreds of thousands of reviews, but it might only have a few related videos. Reviews in this case are unbounded, but videos are bounded. If you have a 1-to-many relationship where the "many" is limited to just a few documents, then you can simply embed it directly in the parent document.

Picture 6



Let's say a product can have up to three videos. We still have a 1-to-many relationship between products and videos but since the number of videos is limited, we can model this by embedding a list of video URLs, shown in **Picture 6**, into our **products** collection.

Code Example 6

```
class Product(JsonModel):
   name: str = Field(index=True)
   image: str = Field(index=True)
   price: int = Field(index=True)
   videos: Optional[List[str]]
```

Code Example 6 shows how you would embed a list of videos directly into your **products** collection using JSON and Redis OM for Python. When making a query, if you don't want to show the videos, you can leave them out of your FT.SEARCH query (See **Code Example 7**).

Code Example 7

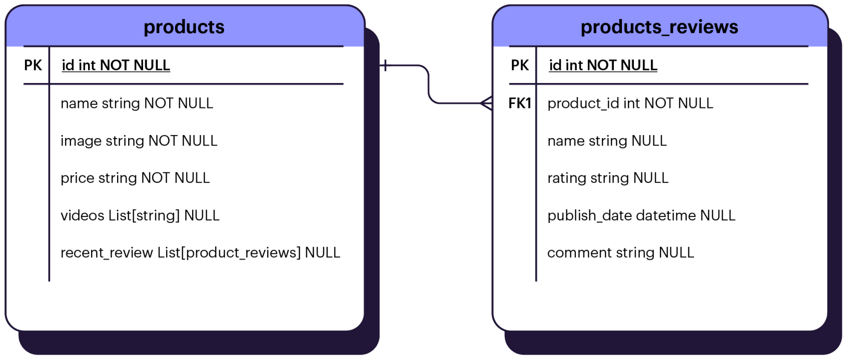
```
async def get_products():
    results = await connections \
        .get_redis_connection() \
        .execute_command(
            f'FT.SEARCH {Product.Meta.index_name} * LIMIT 0 10 RETURN 3 name image
price'
        )

    return Product.from_redis(results)
```

1-to-Many Relationships using Redis with the Partial Embed Pattern

You can also combine these techniques if it makes sense for the application you are building. For example, let's say even though your product reviews are unbounded, you want to quickly show the recent reviews all the time. Instead of doing two different queries, you can simply embed the recent reviews directly into the parent document and still keep the rest of the reviews in a different collection. This is called the **partial embed pattern**. **Picture 7** shows the entity relationship diagram for partially embedding **product_reviews**.

Code Example 8 shows the data model for products with embedded recent reviews.

Picture 7



Code Example 8

```
class ProductDetail(JsonModel):
   product_id: str = Field(index=True)
   reviewer: str
   rating: str
   published_date: datetime.date
   comment: str

class Product(JsonModel):
   name: str = Field(index=True)
   image: str = Field(index=True)
   price: int = Field(index=True)
   videos: Optional[List[str]]
   recent_reviews: Optional[List[ProductReview]]
```

Looking at **Code Example 9** you can see five functions. The first function shows how to get a list of products with the name, image, and price fields. This is useful for the listview because you don't need to show videos or recent reviews in your list of products. For the detailed view, you do want to show product videos and recent reviews. For that, you can simply use the **get_product** function above.

This makes sense and enables your UI to provide a glimpse of the reviews for a product. Then, you might have a "See all reviews" button in your UI which triggers a call to get the rest of the reviews. The **get_reviews** function in **Code Example 9** demonstrates how you can offset before querying for reviews.

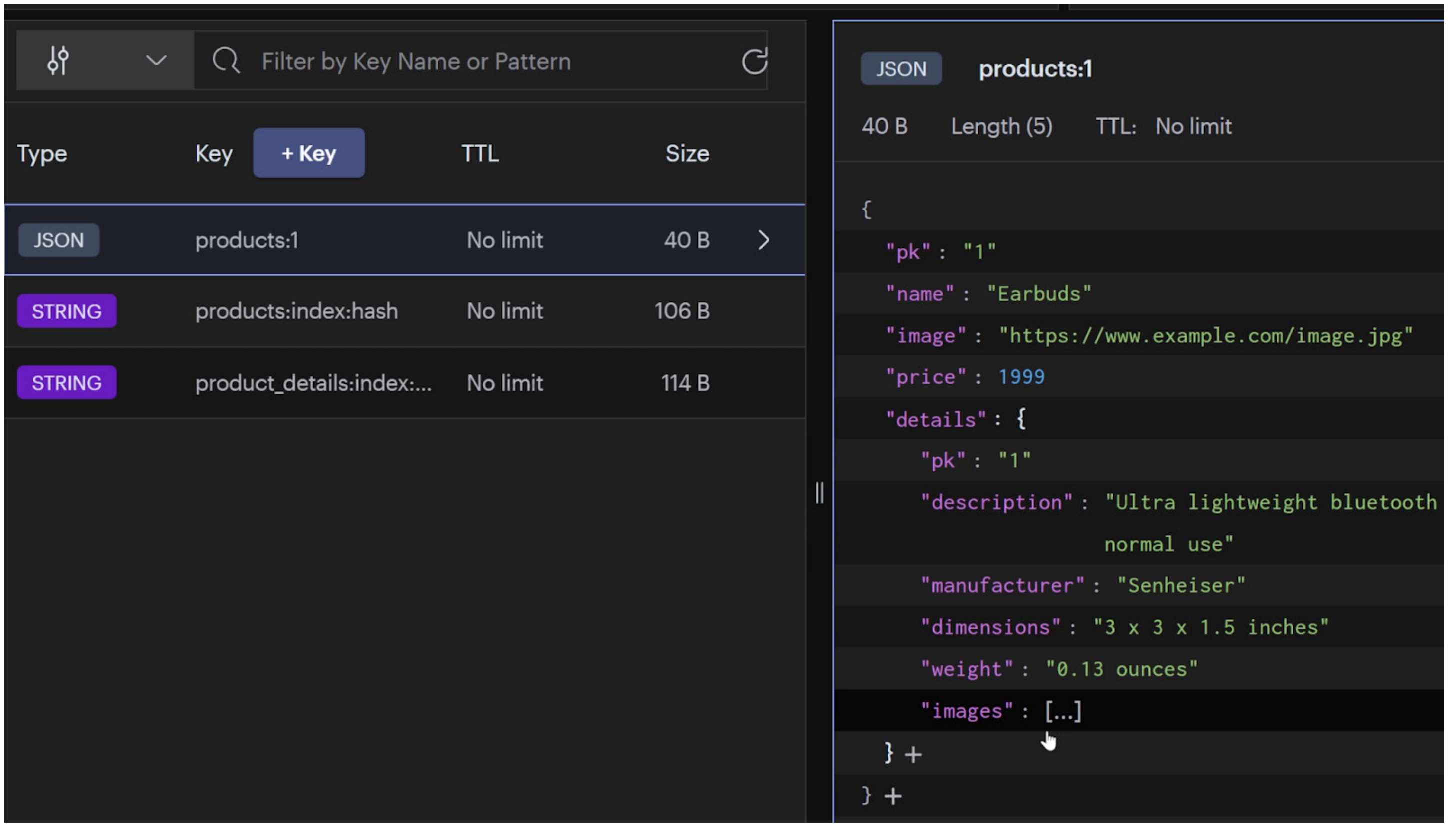
Code Example 9

```
async def get_products():
    results = await connections \
        .get_redis_connection() \
        .execute_command(
            f'FT.SEARCH {Product.Meta.index_name} * LIMIT 0 10 RETURN 3 name image 
price'
        )

    return Product.from_redis(results)

async def get_products(product_id: str):
    return await Product.get (product_id)

async def get_reviews(product_id: str):
    return await = ProductReview.find (ProductReview.product_id == product_id)
    query.offset = 2
    return await query.all ()

async def add_review(review: ProductReview):
    product = await Product.get (review.product_id)
    review.recent_reviews.insert(0, review )

    if (len(product.recent_reviews) > 2):
        product.recent_reviews.pop()

    await review.save()
    await product.save()
```

The assumption in the code is that we only store the two most recent reviews embedded in each product document. Finally, **add_review** shows how you would insert a new review into the product document. You would simultaneously insert it into your **product_reviews** collection and pop off the last review in your embedded **recent_reviews** if necessary.

Picture 8 shows what the data looks like in RedisInsight. You can see one **product** and five **product_reviews** in the database, and also see that there are two **recent_reviews** embedded within the product, and all **videos** embedded.

Picture 8



Type	Key	TTL	Size
JSON	products:1	No limit	40 B
STRING	products:index:hash	No limit	106 B
STRING	product_details:index:...	No limit	114 B

JSON products:1

40 B Length (5) TTL: No limit

```
{
  "pk" : "1"
  "name" : "Earbuds"
  "image" : "https://www.example.com/image.jpg"
  "price" : 1999
  "details" : {
    "pk" : "1"
    "description" : "Ultra lightweight bluetooth
                     normal use"
    "manufacturer" : "Senheiser"
    "dimensions" : "3 x 3 x 1.5 inches"
    "weight" : "0.13 ounces"
    "images" : [...]
  } +
} +
```

Modeling Many-to-Many Relationships

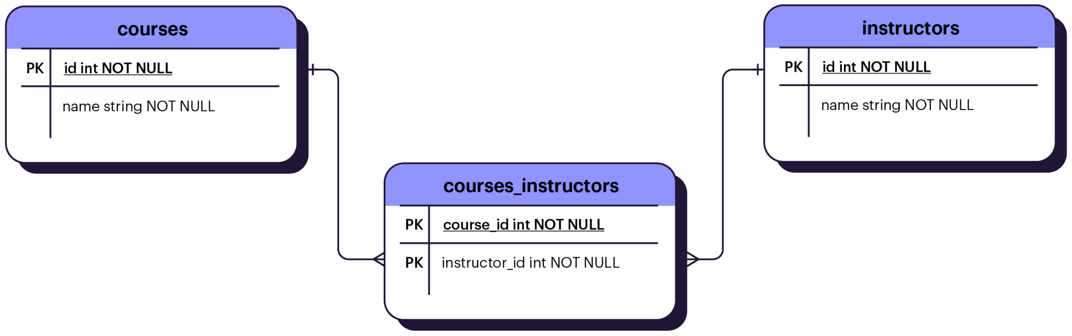
Many-to-Many relationships are very common and can be modeled in several ways with NoSQL databases. Here are the two most common data modeling patterns for many-to-many relationships.

Pattern 1: Many-to-Many Relationship with Bounded Sides

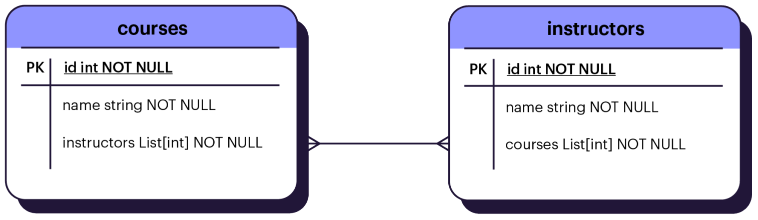
Imagine you are creating an app for an online school that has courses and instructors. There is a many-to-many relationship between courses and instructors, but the list of instructors who teach a course is bounded on both sides, meaning there will be a limited number of instructors teaching a course and a limited number of courses taught by an instructor.

In a relational database, you might have a table called **courses** and another table called **instructors**. Then you would have a junction table called **courses_instructors** that would store the relationship between courses and instructors. This can be seen in **Picture 9**.

Picture 9



Picture 10



In NoSQL, you can simplify this by embedding a list of instructor keys in each course document and a list of course keys in each instructor document. This can be seen in **Picture 10** and is known as two-way embedding. Let's see what this looks like in code.

Code Example 10

```
class Product(JsonModel):
   name: str
   instructors: Optional[List[str]] = Field(index=True)

class Instructor(JsonModel):
   name: str = Field(index=True)
   courses: Optional[List[str]] = Field(index=True)

async def get_courses_with_instructor(instructor_pk: str):
   return await Course.find(Course.instructors << instructor_pk).all()


async def get_instructors_with_course(course_pk: str):
   return await Instructor.find(Instrusctor.courses << course_pk).all()
```

Code Example 10 uses Redis OM for Python to model Courses with a name field and an instructors field that is a list of strings representing the unique keys for instructors. There is also an Instructors collection with a name field and a courses field that is a list of strings representing the unique keys for the courses. Note the code, "Field(index=True)" is used to enable searching for instructors and courses using Search and Query.

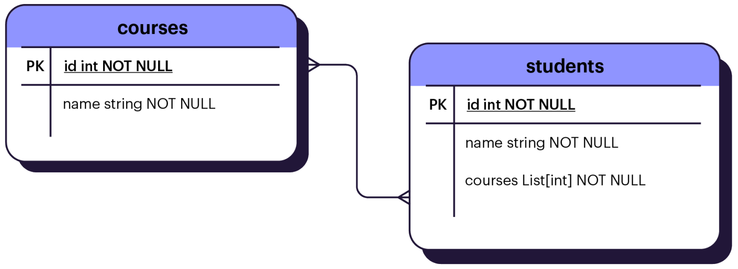
Redis OM will automatically create an index for the specified keys. The "get_courses_with_instructor" function takes in an instructor key and returns all of the courses that contain that instructor. The "get_instructors_with_course" does the opposite, returning instructors for a given course. The two-way embedding pattern works well when both sides of the relationship are bounded. But what about when one side is unbounded?

Pattern 2: Many-to-Many Relationship with One Unbounded Side

Now consider the relationship between courses and students. Let's assume this is an online school, and there could be any number of students enrolled in a course. This is an unbounded many-to-many relationship on the course side. However, the student side is bounded because a student will only enroll in a limited number of courses.

In a relational database, you would still model this with a junction table. However, in NoSQL, it makes sense to model it using an embedded list on the bounded side of the relationship. So you would store a list of course keys in each student document as shown in **Picture 11.**

Picture 11



Code Example 11

```
class Course(JsonModel):
    name: str
    instructors: Optional[List[str]] = Field(index=True)

class Student(JsonModel):
    name: str = Field(index=True)
    courses: Optional[List[str]] = Field(index=True)

async def get_students_in_course(course_pk: str):
    return await Student.find(Student.courses << course_pk).all()


async def get_courses_for_student(student_pk: str):
    student = await Student.get(student_pk)
    return await Course.find(Course.pk << student.courses).all()
```

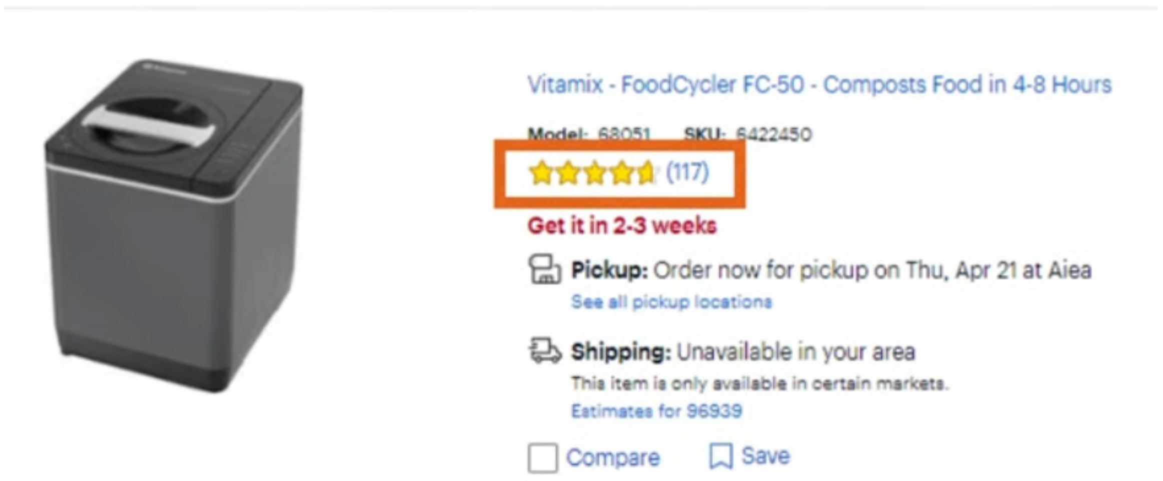
Code Example 11 shows Courses with the name and instructor fields. Since the number of students in a course is unbounded, you don't need to store a list of students in each course document. Instead, Students has a name field and a courses field that is a list of strings representing the unique keys for the courses in which a student is enrolled. You also see two functions, one for finding students in a course and the other for finding courses that have a specific student enrolled. This is how you model many-to-many relationships when one side of the relationship is unbounded and the other is bounded.

To recap, data modeling for many-to-many relationships can be represented by embedding one or both sides of the relationship depending upon whether a side is bounded or unbounded. If both sides are bounded, then you can embed the relationship on both sides. If only one side is bounded, then you should avoid embedding the unbounded side. You should also favor embedding references unless you have information that is primarily static and won't change over time.

The Aggregate Pattern

Let’s revisit the e-commerce site example. Every e-commerce site needs to keep a record of product reviews, and also needs to show average ratings for each product on the page every time a customer looks at the page (shown in **Picture 12**). This means, that for every page visit, the server needs to calculate the average rating for every product. This could cause unnecessary overhead on the server and the database.

Picture 12



Vitamix - FoodCycler FC-50 - Composts Food in 4-8 Hours

Model: 68051 SKU: 6422450

(117)

Get it in 2-3 weeks

Pickup: Order now for pickup on Thu, Apr 21 at Aiea

See all pickup locations

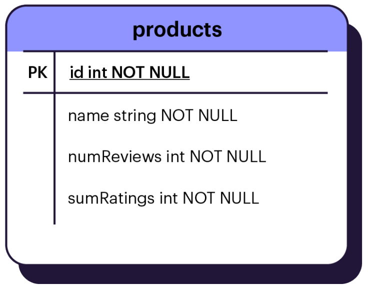
Shipping: Unavailable in your area

This item is only available in certain markets.

Estimates for 96939

This is called the Aggregate Pattern, also known as the Computed Pattern. In our e-commerce example, you can store the number of reviews as well as the sum of ratings on each product's JSON document. This can be seen in **Picture 13**.

Picture 13



When a new review is added you can increment the count and add the new rating to the existing ratings sum. Then, when a customer searches for a product, you read the sum and the count of ratings and calculate the average on the front end by dividing the sum by the count. This way every time a customer visits the page, the server and the database just need to return the pre-calculated values, resulting in improved performance. Let's look at a before and after using the Aggregate Pattern in code.

Without the Aggregate Pattern

Code Example 12 shows how you would model **Products** and **ProductReviews** using Redis OM for Node.js. Redis OM supports not only Python and Node.js but also .NET and Spring. The code shown in **Code Example 12** is not using the Aggregate Pattern, so some things are straightforward, and others are more difficult.

For example, to add a new review you simply create it and save it to Redis. However, getting a list of products is a little bit more complicated but can still be done with Redis OM using the FT.AGGREGATE command. Search and Query provides this command, and it allows you to perform aggregate queries easily. In **Code Example 12** the FT.AGGREGATE command is used to group product reviews by productId and perform a reduce to calculate the average rating and count of reviews.

Code Example 12

```
class Product extends Entity {}
class ProductReview extends Entity {}

const productSchema = new Schema(Product, {
    name: { type: 'string' },
});
const productReviewSchema = new Schema(ProductReview, {
    productId: { type: 'string' },
    author: { type: 'string' },
    rating: { type: 'number' },
});

async function addReview(productId, author, rating) {
    const client = await getClient();
    const productReviewRepo = client.fetchRepository(productReviewSchema);
    await productReviewRepo.createAndSave({
        productId,
        author,
        rating,
    });
}
```

Then, after the command is run and the reviews are returned, the average rating and number of reviews are extracted from the result and mapped onto the list of products before returning to the client.

The problem here is that while Redis is incredibly fast, and for many applications will be able to handle this load relatively quickly, it is suboptimal especially for applications with a lot of user traffic. The code in **Code Example 12** in the "getProducts" function would have to be run every time a customer visits the site. This can lead to needless overhead. Instead, let's look at a better approach using the Aggregate Pattern.

Code Example 12 cont'd

```
async function getProducts() {
    const client = await getClient();
    const productRepo = client.fetchRepository(productSchema);
    const productEntities = await productRepo.search().return.all();
    const results = await client.execute(
        'FT.AGGREGATE ProductReview:index * GROUPBY 1 @productId REDUCE AVG 1 @
rating REDUCE COUNT 0'.split(
            /\s+/
        )
    );

    const products = {};
    for (let result of results.slice(1)) {
        const [, productId, , avgRating, , numReviews] = result;
        products[productId] = {
            avgRating: Number(avgRating),
            numReviews: Number(numReviews),
        };
    }

    return productEntities.map((entity) => {
        return {
            ...entity.entityData,
            ...products[entity.entityId],
        };
    });
}
```

With the Aggregate Pattern

Code Example 13 is very similar to **Code Example 12**, only it takes advantage of the Aggregate Pattern and has “numReviews” and “sumRatings” fields on Products. The “addReview” function now requires you to increment “numReviews” and add the rating to the “sumRatings” field.

Code Example 13

```
class Product extends Entity {}
class ProductReview extends Entity {}

const productSchema = new Schema(Product, {
    name: { type: 'string' },
    numReviews: { type: 'number' },
    sumRatings: { type: 'number' },
});
const productReviewSchema = new Schema(ProductReview, {
    productId: { type: 'string' },
    author: { type: 'string' },
    rating: { type: 'number' },
});

async function addReview(productId, author, rating) {
    const client = await getClient();
    const productRepo = client.fetchRepository(productSchema);
    const productReviewRepo = client.fetchRepository(productReviewSchema);
    const productEntity = await productRepo.fetch(productId);

    productEntity.entityData.numReviews += 1;
    productEntity.entityData.sumRatings += rating;

    return Promise.all([
        productRepo.save(productEntity),
        productReviewRepo.createAndSave({
            productId,
            author,
            rating,
```

However, the "getProducts" function is much simpler. Remember that a typical e-commerce application will typically have a much larger number of reads than writes, so you want to optimize your data for reads. When building read-heavy applications, consider using the Aggregate Pattern to reduce the amount of computation necessary at read time for aggregate information.

Code Example 13 cont'd

```
        }),
    ]);
}

async function getProducts() {
    const client = await getClient();
    const productRepo = client.fetchRepository(productSchema);

    return productRepo.search().return.all();
}
```

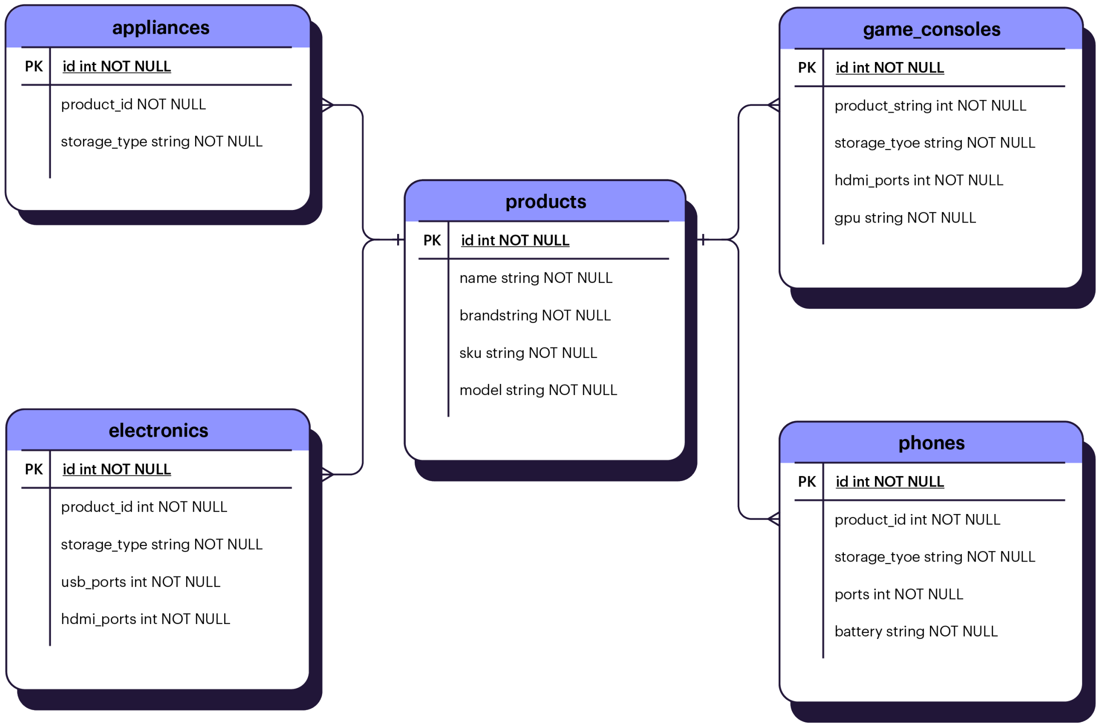
The Polymorphic Pattern

Polymorphism is used when you have things that have some similarities and also some differences. Consider a catalog of products where each product has a name, brand, sku, and model but some products of a certain type might contain size and color while others might not.

For example, a game console and a pair of earbuds might have similar fields such as the product name, brand, model number, sku, and reviews. However the game console has some unique properties such as storage type, number of HDMI ports, etc. The pair of earbuds also has unique fields such as battery life, connection type, fit, etc.

Picture 14 shows the entity relationship diagram you might use in SQL. In SQL you might store some of the shared fields in a "products" table, and then have separate tables to store specifics about the different types of products. Now to get all the products you need to join all these tables.

Picture 14



An example query might look like **Code Example 14**. This can get unwieldy as you add many different types of products to your catalog.

You could also choose to store any possible field in the products table, but this can also get unwieldy, could lead to issues if your database limits the number of columns you can store, and requires you to have a ton of nullable fields in each row.

Code Example 14

```
SELECT
    p.id, p.name, p.brand, p.sku, p.model,
    a.storage_type,
    gc.storage_type, gc.hdmi_ports, gc.gpu,
    e.storage_type, e.usb_ports, e.hdmi_ports,
    ph.storage_type, ph.ports, ph.battery
FROM
    products p
INNER JOIN
    appliances a
ON a.product_id = p.id
INNER JOIN
    game_consoles gc
ON gc.product_id = p.id
INNER JOIN
    electronics e
ON e.product_id = p.id
INNER JOIN
    phones p
ON ph.product_id = p.id
WHERE
    p.id = 1;
```

In Redis, because you get a flexible schema, you can simply store all these in a single collection without having to worry about having a ton of null fields. Further, to distinguish between different product types, you can have a “type” field that groups them together. For example, in our case, we can have type = “game console” and type = “earbuds”.

Let’s look at what this looks like in code.

Code Example 15 shows how using the Polymorphic Pattern in Redis makes it really easy to work with products. The “getProducts” function can get all products by looking at a single collection. If you need to get products of a certain type, like game consoles, you can modify your where clause to search by type.

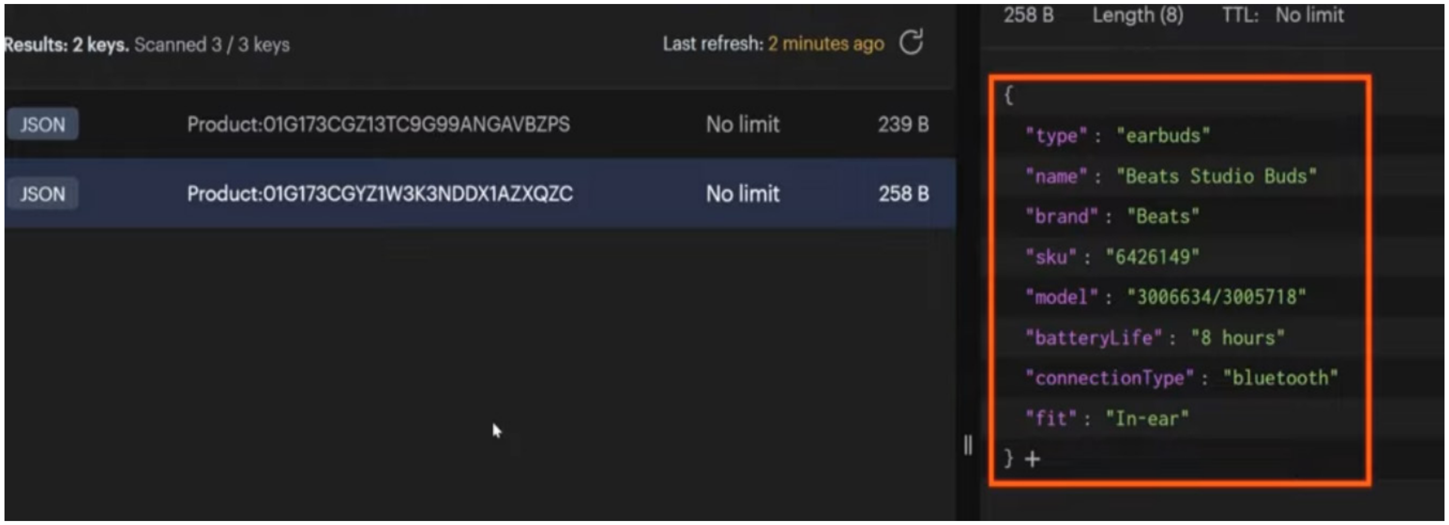
Code Example 15

```
class Product extends Entity {}

const productSchema = new Schema(Product, {
    type: { type: ‘string’ },
    name: { type: ‘string’ },
    brand: { type: ‘string’ },
    sku: { type: ‘string’ },
    model: { type: ‘string’ },
    batteryLife: { type: ‘string’ },
    connectionType: { type: ‘string’ },
    fit: { type: ‘string’ },
    usbPorts: { type: ‘number’ },
    hdmiPorts: { type: ‘number’ },
    storageType: { type: ‘string’ },
});

async function getProducts() {
    const client = await getClient();
    const productRepo = client.fetchRepository(productSchema);

    return productRepo.search().return.all();
}

async function getProductByType(type) {
    const client = await getClient();
    const productRepo = client.fetchRepository(productSchema);

    return productRepo.search().where(‘type’).equals(type).return.all();
}
```

Picture 15 shows what the data might look like in RedisInsight. Something to note is that given Redis allows for a flexible schema you only see the common fields as well as fields specific to a product type. In **Picture 15** you see fields that relate to the earbuds product type, but not fields for any other type.

When building applications, think about how to best design your data schema using fewer collections, and take advantage of the Polymorphic Pattern to combine similar types of data.

Picture 15



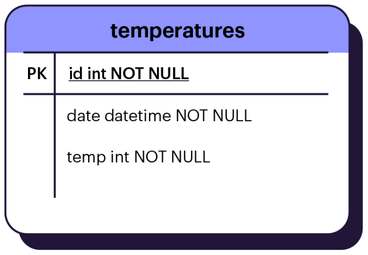
```
Results: 2 keys. Scanned 3 / 3 keys                Last refresh: 2 minutes ago

JSON   Product:01G173CGZ13TC9G99ANGAVBZPS   No limit   239 B
JSON   Product:01G173CGYZ1W3K3NDDX1AZXQZC   No limit   258 B
```

```
258 B   Length (8)   TTL: No limit
{
  "type" : "earbuds"
  "name" : "Beats Studio Buds"
  "brand" : "Beats"
  "sku" : "6426149"
  "model" : "3006634/3005718"
  "batteryLife" : "8 hours"
  "connectionType" : "bluetooth"
  "fit" : "In-ear"
} +
```

The Bucket Pattern

Imagine you are building an application that will take temperature measurements for monitoring purposes. You want to use Redis for this because it is fast, and you will need to access the data frequently. You may think to store each measurement embedded in a JSON document with the timestamp and temperature reading (shown in **Picture 16**). However, while this approach seems reasonable, it can cause issues as your application scales to have tons of these measurements.

Picture 16



Working with Time-series Data in Redis

A better way to store this is to use the time-series capabilities of Redis. With RedisTimeSeries you can store your measurements in a time-series data structure. In this case you might also want to have easy access to the average temperature over a period of time. Let's see what this looks like in code:

Using **Code Example 16** as a guide, you need to first create a time series before you can add measurements to it. However, you want to make sure the time series doesn't already exist before you create it using the EXISTS command in Redis. To create a new time series, you need to use the TS.CREATE command.

Code Example 16

```
async function createTimeSeries() {
    const client = await getClient();
    const exists = await client.execute('EXISTS temperature:raw'.split(' '));

    if (exists === 1) {
        return;
    }

    const commands = [
        'TS.CREATE temperature:raw DUPLICATE_POLICY LAST',
    ];

    for (let command of commands) {
        await client.execute(command.split(' '));
    }
}
```

We are calling our time series "temperature:raw" because it will be storing all of the raw temperature measurements from our data. We are also specifying a DUPLICATE_POLICY of "last", meaning that if we try to add multiple samples with the same timestamp it will always keep the newest reported value.

Code Example 16 cont'd

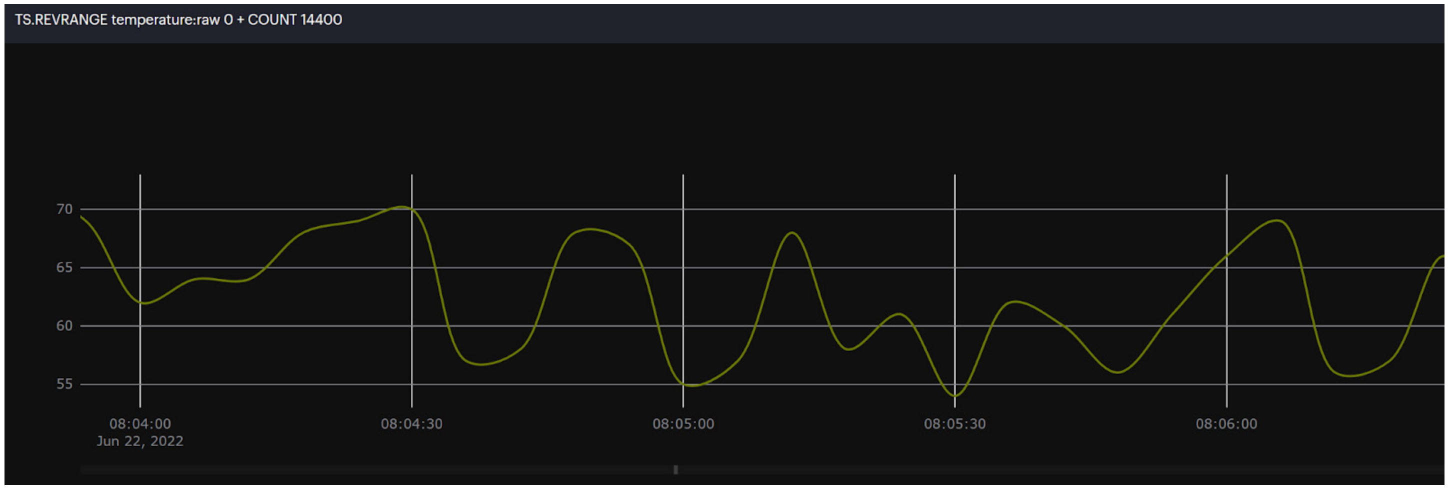
```
async function add(values) {
    const client = await getClient();
    const chunkSize = 10000;

    for (let i = 0; i < values.length; i += chunkSize) {
        const chunk = values.slice(i, i + chunkSize);
        const series = chunk.reduce((arr, value) => {
            return arr.concat([
                'temperature:raw',
                new Date(value.date).getTime(),
                value.temp,
            ]);
        }, []);

        // TS.MADD temperature:raw timestamp temp temperature:raw timestamp temp ...
        await client.execute(['TS.MADD', ...series]);
    }
}
```

```
TS.REVRANGE temperature:raw 0 + COUNT 14400
```

You will also see an add function here that takes in an array of temperature readings with the timestamp and temperature value. The sample data has a year's worth of temperature readings every 6 seconds, totaling about 5.3 million samples. For this reason, we are splitting the data into chunks of 10 thousand samples. We are then using the TS.MADD command to store each batch of 10 thousand samples.

You can use TS.MADD to append new values to one or more time series. In this case, I am appending to the temperature:raw time series. If you want to visualize a time series, RedisInsight is a great tool. You can use the workbench and run a TS.RANGE or TS.REVRANGE command to get a graph of your time-series data. For example, the following command would give us the prior day's temperature readings, and **Picture 17** shows the visualization from RedisInsight.

Picture 17



```
TS.REVRANGE temperature:raw 0 + COUNT 14400
```

Aggregating Time-series Data with Redis

While it is nice to get a view of all the data, what is also nice is to be able to see the average temperature over a period of time. You can do this using the TS.RANGE command and specifying an AGGREGATE command of twa, for time-weighted average, as well as a bucket duration. Let's specify a bucket duration of the number of milliseconds in a month so we can see the average monthly temperature in our time series.

You can use the TS.RANGE command to get the average temperature over a period of time. However, as you store more measurements the time it takes to calculate the average will increase. There is a better way to handle this using the Bucket pattern.

With the Bucket Pattern and Redis, you can automatically aggregate your data as you go along. Say, for example, you want to keep track of the average hourly temperature reading. Redis can do this automatically for you with the TS.CREATERULE command. Let's see what this looks like in code.

Code Example 17 shows two new time series added, temperature:daily and temperature:monthly. It also shows two rules created using TS.CREATERULE to take the time-weighted average temperatures as they are added to temperature:raw and store them in the respective daily and monthly time series.

Code Example 17

```
async function createTimeSeries() {
    const client = await getClient();
    const exists = await client.execute('EXISTS temperature:raw'.split(' '));

    if (exists === 1) {
        return;
    }

    const commands = [
        'TS.CREATE temperature:raw DUPLICATE_POLICY LAST',
        'TS.CREATE temperature:daily DUPLICATE_POLICY LAST',
        'TS.CREATE temperature:monthly DUPLICATE_POLICY LAST',
        'TS.CREATERULE temperature:raw temperature:daily AGGREGATION twa 86400000',
        'TS.CREATERULE temperature:raw temperature:monthly AGGREGATION twa
2629800000',
    ];

    for (let command of commands) {
        await client.execute(command.split(' '));
    }
}
```

The TS.CREATERULE command takes in a sourceKey, destinationKey, aggregator function, and bucketDuration. The sourceKey is the key to the source time series where you are storing your raw data. The destinationKey is where you want to store the new, bucketed time series. The aggregator is the function you want to use for your buckets. In our case, we will use twa to store the time-weighted average. Finally, the bucketDuration is the timespan in milliseconds for your buck-ets.

Note that you should never explicitly add to the bucketed time series as it will be done automatically for you. Also, the rule does not retroactively apply to an existing time series. Only new samples that are added to the source time series will be aggregated. So if you look at the differences between **Code Example 16** and **Code Example 17** you will see that we only needed to add the two new time-series keys and the two rules. Redis takes care of the rest!

Now if we want to get the average monthly temperature we can simply query the monthly time series with the following TS.RANGE command.

```
TS.RANGE temperature:monthly 0
```

Note that you don't have to specify any aggregator function because it's already done for you using TS.CREATERULE. That not only makes the command more readable than the aggregate command we had to run previously, but it also runs much faster.

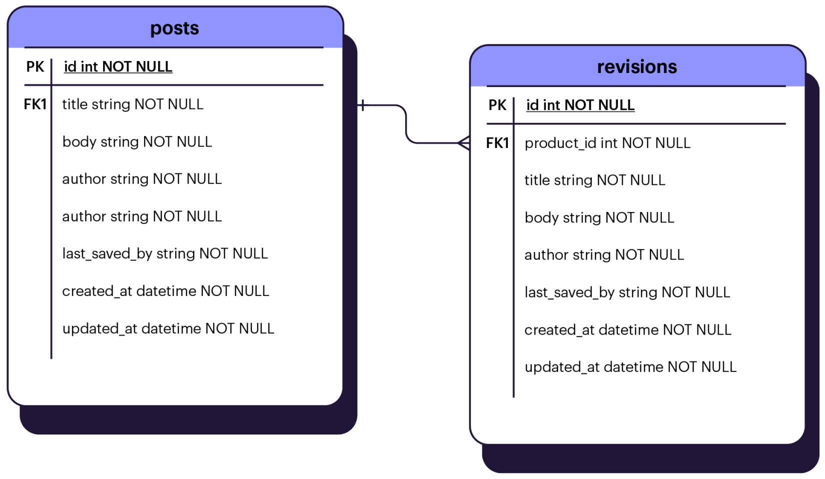
While Redis is incredibly fast when performing aggregate queries, using the bucket pattern to keep track of aggregate values as you go is much faster.

The Revision Pattern

Imagine you're an editor for a digital publication. You work with several team members to write and edit each post before it gets published. You need to keep track of content revisions as well as who made those revisions.

As seen in **Picture 18,** in SQL you might store all posts in a table and have the revisions in a separate table. Then, when you want to view the revisions for a specific post you need to query the latest version from the posts table and join all the revisions from the revisions table that match that post.

Picture 18


posts
PK id int NOT NULL
FK1 title string NOT NULL
body string NOT NULL
author string NOT NULL
author string NOT NULL
last_saved_by string NOT NULL
created_at datetime NOT NULL
updated_at datetime NOT NULL
revisions
PK id int NOT NULL
FK1 product_id int NOT NULL
title string NOT NULL
body string NOT NULL
author string NOT NULL
last_saved_by string NOT NULL
created_at datetime NOT NULL
updated_at datetime NOT NULL

With Redis, you can store a post and its revisions in a single document. This simplifies your queries and lets you organize your content more logically. This is called the Revision Pattern. Let's see what this looks like in code.

Code Example 18 shows how you would model Posts and embedded Revisions using Redis OM for Python. Both models share attributes such as title, body, author, last_saved_by, created_at, and updated_at. Posts have an additional attribute which is the list of revisions.

Code Example 18

```
class Revision(EmbeddedJsonModel):
    title: str = Field(index=True)
    body: str = Field(index=True)
    author: str = Field(index=True)
    last_saved_by: str = Field(index=True)
    created_at: datetime.date = Field(index=True)
    updated_at: datetime.date = Field(index=True)

class Post(JsonModel):
    title: str = Field(index=True)
    body: str = Field(index=True)
    author: str = Field(index=True)
    last_saved_by: Optional[str] = Field(index=True)
    created_at: Optional[datetime.date] = Field(index=True)
    updated_at: datetime.date = Field(index=True)
    revisions: Optional[List[Revision]]
```

Code Example 19 shows the standard CRUD operations for a post. To create a new post we simply take in all the post attributes and save them to Redis using Redis OM. To update a post, we first get the post from Redis, create a new revision for it, insert it at the beginning of the post's revisions list, update the post with the new attributes, then save the post.

Code Example 19

```
async def create_post(**args):
    dt = datetime.now().isoformat()
    post = Post(
        title=args[“title”],
        body=args[“body”],
        author=args[“author”],
        last_saved_by=args[“last_saved_by”],
        created_at=dt,
        updated_at=dt,
        revisions=[]
    )

    return await post.save()


async def update_post(id: str, **args):
    post = await Post.get(id)
    revision = Revision(
        title=post.title,
        body=post.body,
        author=post.author,
        last_saved_by=post.last_saved_by,
        created_at=post.created_at,
        updated_at=post.updated_at)
```

When you get a list of posts, you don't always want the revisions for each post. The FT.SEARCH command lets you search Redis using an index and also specify the fields to return. Redis OM automatically creates the Post index for you, and then you can use it to run custom searches if you need to. In "get_posts", we are querying all posts, denoted by the asterisk, and returning 5 fields: title, body, author, created_at, and updated_at. Now let's see what this looks like in RedisInsight.

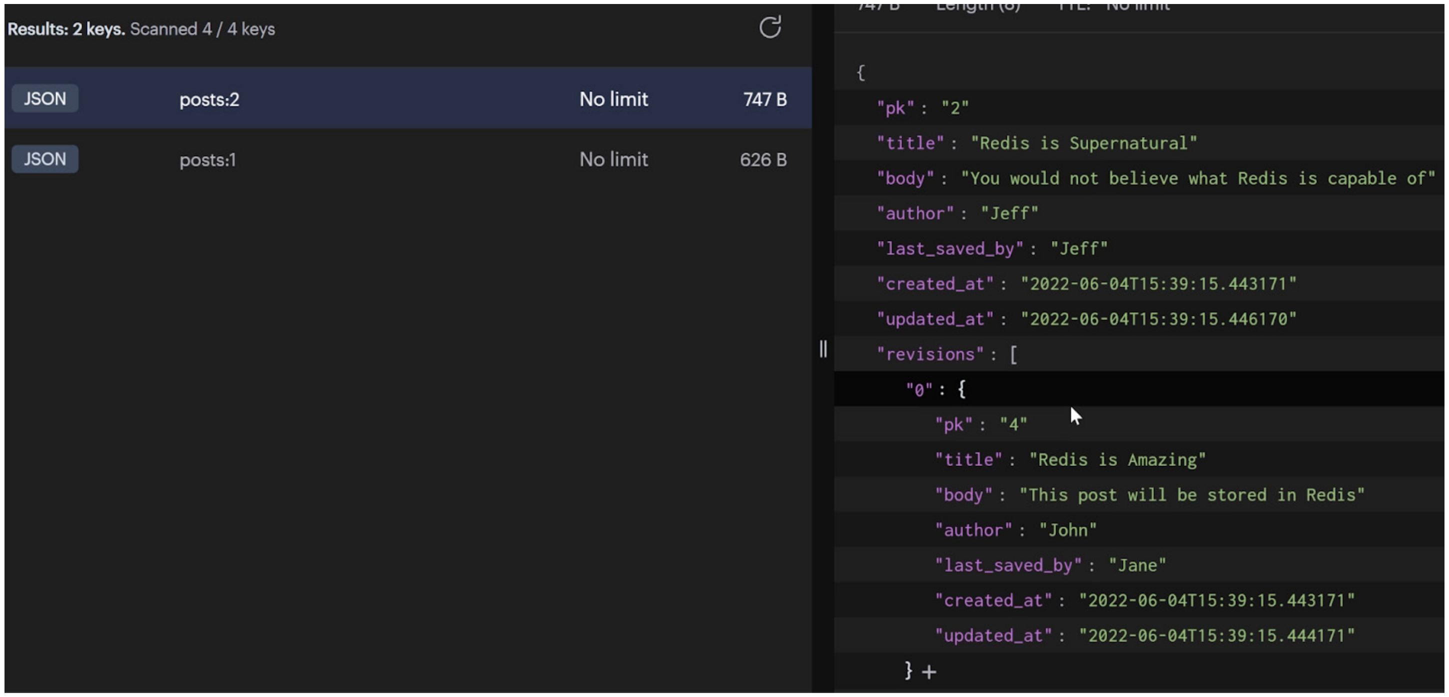
Code Example 19 cont'd

```
    post.revisions.insert(0, revision)
    post.title = args.get("title", post.title)
    post.body = args.get("body", post.body)
    post.author = args.get("author", post.author)
    post.last_saved_by = args.get("last_saved_by", post.last_saved_by)
    post.updated_at = datetime.now().isoformat()

    return await post.save()

async def get_posts():
    results = await connections \
        .get_redis_connection() \
        .execute_command(
            f'FT.SEARCH {Post.Meta.index_name} * LIMIT 0 10 RETURN 5 title body
author created_at updated_at'
        )

    return Post.from_redis(results)

async def get_post(id: str):
    return await Post.get(id)
```

Using **Picture 19**, in RedisInsight you can see there are two posts in the database. For the selected post there are some revisions. Note the title, body, author, and last_saved_ by fields are different in the main post than in the revisions.

While publishing is a very common industry that uses the Revision Pattern it is also applicable to industries where you need an audit trail of all document changes such as law, finance, healthcare, and insurance.

Picture 19



Results: 2 keys. Scanned 4 / 4 keys

JSON	posts:2	No limit	747 B
JSON	posts:1	No limit	626 B

```
{
  "pk" : "2"
  "title" : "Redis is Supernatural"
  "body" : "You would not believe what Redis is capable of"
  "author" : "Jeff"
  "last_saved_by" : "Jeff"
  "created_at" : "2022-06-04T15:39:15.443171"
  "updated_at" : "2022-06-04T15:39:15.446170"
  "revisions" : [
    "0" : {
      "pk" : "4"
      "title" : "Redis is Amazing"
      "body" : "This post will be stored in Redis"
      "author" : "John"
      "last_saved_by" : "Jane"
      "created_at" : "2022-06-04T15:39:15.443171"
      "updated_at" : "2022-06-04T15:39:15.444171"
    } +
```

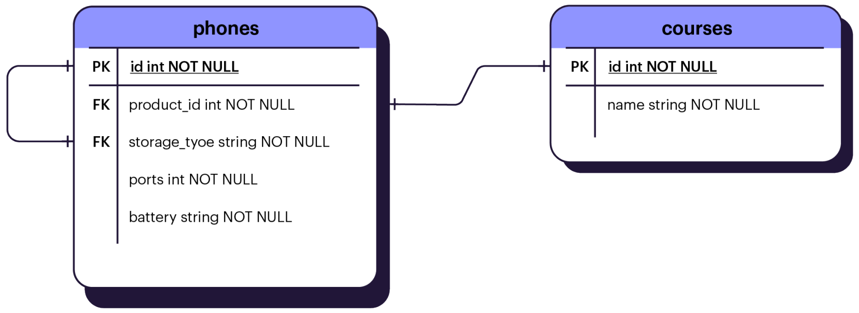
The Tree and Graph Pattern

When working with NoSQL document databases it is generally recommended to minimize the number of JOINs you need to build your data model. Even in SQL, JOINs can cause overhead and slow down data retrieval. However, sometimes your data requirements are such that you cannot avoid JOINs. We've already covered various patterns for modeling relatively simple relationships, including when to embed and when to keep things separate. One thing we didn't talk about is more complex relationships such as graphs or trees.

For example, imagine you are building an enterprise resource planning (ERP) system. One of the most important parts of the system is the org chart. At the very least, you need to be able to show details about each employee, where they are located, and who they report to (or who reports to them). The most logical way to store this data is in a tree. Let's look at how you might do this in traditional SQL as well as NoSQL with Redis using the Tree and Graph Pattern.

Storing trees in SQL is straightforward, as SQL is designed specifically for relationship modeling. To model your org chart, you might have two tables (shown in **Picture 20**): employees and locations.

Picture 20



Using **Picture 20** let's look at two potential SQL queries you might want to make. You need one query for getting employees who work at a specific location and another query for getting employees who work for a specific manager.

Code Example 20 shows two relatively simple queries for getting employees with a specific manager or at a specific location. This works well, but what if you want to go one or more levels deeper with the manager query. For example, if you want to find employees who have two degrees of separation from someone. This becomes more complicated to accomplish with SQL and requires you to add additional JOINs. Depending on the complexity of your query and how many employees you have in your database, it can become prohibitively slow to use SQL to get the information you need.

Code Example 20

```
# Get employees with a specific manager
SELECT
    e.id, e.name, e.title
FROM
    employees e
INNER JOIN
    employees e2
ON e2.reportsto_id = e.id
WHERE
    e.id = 1;

# Get employees who work at a specific location
SELECT
    e.id, e.name, e.title
FROM
    employees e
INNER JOIN
    locations l
ON l.id = e.location_id
WHERE
    l.id = 1;
```

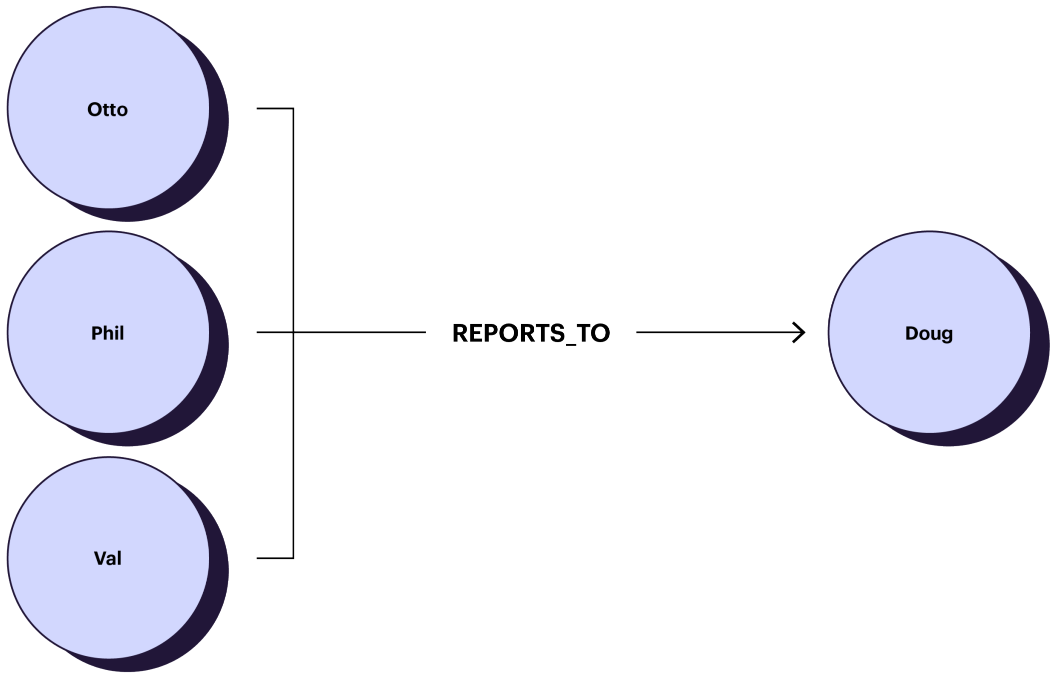
Redis comes with built-in graph capabilities, which makes working with complex relationships more intuitive and faster. Under the hood, Redis uses the Cypher language to allow you to work with trees and graphs. Let's take the same example from Picture 20 and see how you would use Redis to accomplish the same thing.

Code Example 21 is the query you would perform on Redis to get results that match those from **Code Example 20**, and **Picture 21** shows the visualization you get when you run the first query in the RedisInsight Workbench.

Code Example 21

```
# Get employees with a specific manager
GRAPH.QUERY Org "MATCH (e:Employee)-[:REPORTS_TO]->(m:Employee { name: 'Doug'})
RETURN e,m"

# Get employees who work at a specific location
GRAPH.QUERY Org "MATCH (e:Employee)-[:WORKS_AT]->(l:Location { name: 'Seattle'})
RETURN e,l"
```

Picture 21



Code Example 22 shows how, with a small addition of "*2", you can get employees with two degrees of separation from Doug.

The built-in graph capabilities of Redis allow you to use the Tree and Graph Pattern to model complex relationships without having to worry about the complexity of SQL queries. While this pattern is useful for HR systems, it is also seen in Content Management Systems (CMSs), product catalogs, social networks, and more.

Code Example 22

```
# Get employees two degrees separated from a specific manager
GRAPH.QUERY Org "MATCH (e:Employee)-[:REPORTS_TO*2]->(m:Employee { name: 'Doug'})
RETURN e,m"
```

The Schema Version Pattern

In the past, you've probably built an app, designed your data model and everything seemed perfect. Then something comes up that prompts a change to your data model. You need to determine whether you should make a breaking change and rewrite all your application code to use the new data model at the same time.

For example, when your app started maybe you were storing one email address per user but now you need to store multiple email addresses. You could add additional columns such as "email2", "email3", etc. However, a better way is to use a list of email addresses.

While the most future-proof way is to use an embedded list, the problem is all your existing users are stored with email address fields directly in their document rather than in the "email addresses" list. In addition, all of your existing code is using the email address fields on a user, not from within the email address list. So what do you do? This is where you can use the Schema Version pattern to your advantage.

The Schema Version pattern is a way of assigning a version to your data model. This is usually done at the document level, but you may also choose to version all of your data as part of an API version. It is recommended that you always assign a version to your documents so that you can change them in the future without having to worry about immediately migrating all of your data and code. If you're using Redis your schema is flexible, and you can make changes to your existing schema without any downtime.

If you aren't already using the Schema Version Pattern the good news is you can start today without making any significant changes to your application logic. Let's dive into some code to see how you might introduce the Schema Version Pattern into an existing system.

In **Code Example 23** we are using Redis OM for Node.js and defining a User schema with name and email. To create a new user we need to save the user data to Redis, then return the new user. To get a user we simply fetch it from Redis using its unique ID. Finally, to update a user we fetch it by ID and then update the fields. In a full production app, you might do a lot more than this, but this is a simple example of how you might start writing an app to create, read, and update users.

Code Example 23

```
class User extends Entity {}

const userSchema = new Schema(User, {
    name: { type: 'string' },
    email: { type: 'string' },
});

export async function create(data) {
    const client = await getClient();
    const repo = client.fetchRepository(userSchema);
    const user = repo.createEntity(data);

    await repo.save(user);

    return user.toJSON();
}

export async function read(id) {
    const client = await getClient();
    const repo = client.fetchRepository(userSchema);
    const user = await repo.fetch(id);

    return user.toJSON();
}
```

Code Example 23 cont'd

```
export async function update(id, data) {
    const client = await getClient();
    const repo = client.fetchRepository(userSchema);
    const user = await repo.fetch(id);

    user.name = data.name;
    user.email = data.email;

    await repo.save(user);

    return user.toJSON();
```

This is working well for now, but what happens when we need to change users to have a list of email addresses? We need to change the existing schema and write some code to incrementally migrate old users to the new schema.

The best way to do this is to create a translation function that you run whenever a new user is created or an old user is updated. The reason you want to do this is twofold. First, you only want to migrate a user document one time, so you don't want to translate it during read time. Second, you want to allow your existing applications to continue to use older schemas. Let's see how we might incrementally migrate old documents to use the new schema while supporting existing application logic.

Using **Code Example 24**, this time we're defining our User schema with additional fields for schema, contact, and an emails list. We want to eventually rename the email field to contact so it is more straightforward. However, we don't want to remove the email field yet because we still want to support legacy code.

The "translate" function will translate data from the old schema to the new schema. By default, we will assume anyone using the function is still using the old schema. This is safe as you don't want to automatically assume everyone wants to start using the new schema. In the translate function, we do nothing if the old schema is in use. We also do nothing if the schema of the incoming data matches. If neither of those conditions is true, we migrate the old schema to the new format and return it.

Code Example 24

```
class User extends Entity {}

const userSchema = new Schema(User, {
    schema: { type: 'string' },
    name: { type: 'string' },
    email: { type: 'string' },
    contact: { type: 'string' },
    emails: { type: 'string[]' },
});

function translate(data, schema = '1') {
    // Ignore data if using the old schema
    if (schema === '1') {
        return data;
    }

    // Ignore data if already using schema '2'
    if (schema === '2' && data.schema === '2') {
        return data;
    }

    // Migrate old data to new schema
    data.schema = schema;
    data.emails = [data.email];
    data.contact = data.email;
    data.email = null;

    return data;
}
```

We also need to rewrite the create function to support both schemas. Once again we use the old schema by default. Then, we call to translate the data, save it to Redis, and return the JSON. Updating an existing user can be a little bit tricky. First we fetch the user from Redis, then make a call to translate the incoming data. Finally, we update the user fields according to the schema in use.

While this might be a contrived example, the principles still apply in a real-world application. You should always use the Schema Version pattern to aid in incrementally migrating your data as you require changes to your data model.

Code Example 24 cont'd

```
export async function update(id, data, schema = '1') {
   const client = await getClient();
   const repo = client.fetchRepository(userSchema);
   const user = await repo.fetch(id);

   data = translate(data, schema);

   if (schema === '1') {
       user.schema = schema;
       user.name = data.name;
       user.email = data.email;
   } else {
       user.schema = data.schema;
       user.name = data.name;
       user.email = data.email;
       user.contact = data.contact;
       user.emails = data.emails;
   }

   await repo.save(user);

   return user.toJSON();
```

Summarizing The Patterns

That was a lot of information to take in! Let’s briefly look back at all the patterns we learned throughout this e-book, and also consider the use-cases for each one.

The Embedded Pattern

The Embedded Pattern is used in NoSQL document databases, such as Redis, to allow you to keep all information relevant to a specific data type within the same document. This is useful in a wide range of applications. Almost every application you build can and should take advantage of The Embedded Pattern.

The Partial Embed Pattern

The Partial Embed Pattern is where you embed a subset of a larger collection within a document. This is useful in e-commerce applications to store product reviews. It is also useful in publishing and social media applications to show top comments. There are many use-cases for the partial embed pattern, the important thing is for you to recognize when it might be useful based on how you have to present information to your users.

The Aggregate Pattern

The Aggregate Pattern is used to store attributes of a larger embedded (or separate) collection within a document. We used it to store the number of reviews and sum of ratings in our product documents, thus making it easier to calculate the average rating for products in a listview. However, this pattern is useful in IoT, real-time analytics, and other types of catalogs as well.

The Polymorphic Pattern

The Polymorphic Pattern applies when there are several different variations of similar data, with more similarities than differences. It's useful for when you really want data kept in a single collection for viewing purposes. We used the example of a product catalog, where products have some shared properties and other unique properties based on their type.

However, we wanted to be able to query and show all products as easily as possible. The Polymorphic Pattern lets us store all of the permutations of products in a single product collection. This pattern is also very useful in content management systems (CMS), learning management systems (LMS), and customer relationship management systems (CRM).

The Bucket Pattern

You might also call this pattern the time-series pattern. The Bucket Pattern is where you have time-series data and you want to store it in aggregate "buckets" based on time periods. This is useful when managing streaming data such as sensor readings, real-time analytics, and IoT applications. Redis makes this pattern incredibly easy with its built-in time-series bucketing capabilities.

The Revision Pattern

Use the Revision Pattern when you need to maintain previous versions of a document. We used a CMS example, but this pattern is useful in the legal, financial, healthcare, and insurance industries. NoSQL document databases like Redis make it easy to apply this pattern because you can embed revisions with the latest version all within the same document.

The Tree and Graph Pattern

The Tree and Graph Pattern is useful when you need to model complex relationships and you can't take advantage of the Embedded Pattern. This typically means your data is hierarchical and needs to be accessed and changed frequently. The key advantage NoSQL has over SQL here is avoiding multiple JOINs for accessing several levels of a tree. This pattern can be seen in ERPs, CMSs, product catalogs, and social networks.

The Schema Version Pattern

Last, but certainly not least, the Schema Version Pattern applies to every application you ever build using NoSQL. It is incredibly useful for helping you improve your schema over time. Redis and other NoSQL databases are sometimes referred to as "schemaless." While this is true, in reality, a schema is very important in every database.

You also need to be able to change your data model and let applications that use your data upgrade gracefully. The Schema Version Pattern enables you to let applications upgrade when they want, and understand a document based on its schema version.

Conclusion

I hope you enjoyed this e-book. Let it serve as a reference for you as you go out and build amazing applications using NoSQL and Redis! Remember, even though all of the examples in this e-book use Redis, the same patterns and principles apply to other NoSQL databases.

Also, keep in mind that all of the patterns mentioned can be used together in your application. When you are approaching building an application, have these patterns in the back of your mind, and figure out which pattern applies best to the problem you are trying to solve. You have been given the tools and knowledge needed to build applications using NoSQL. Now you just need to get started!