Data modeling with Amazon DocumentDB

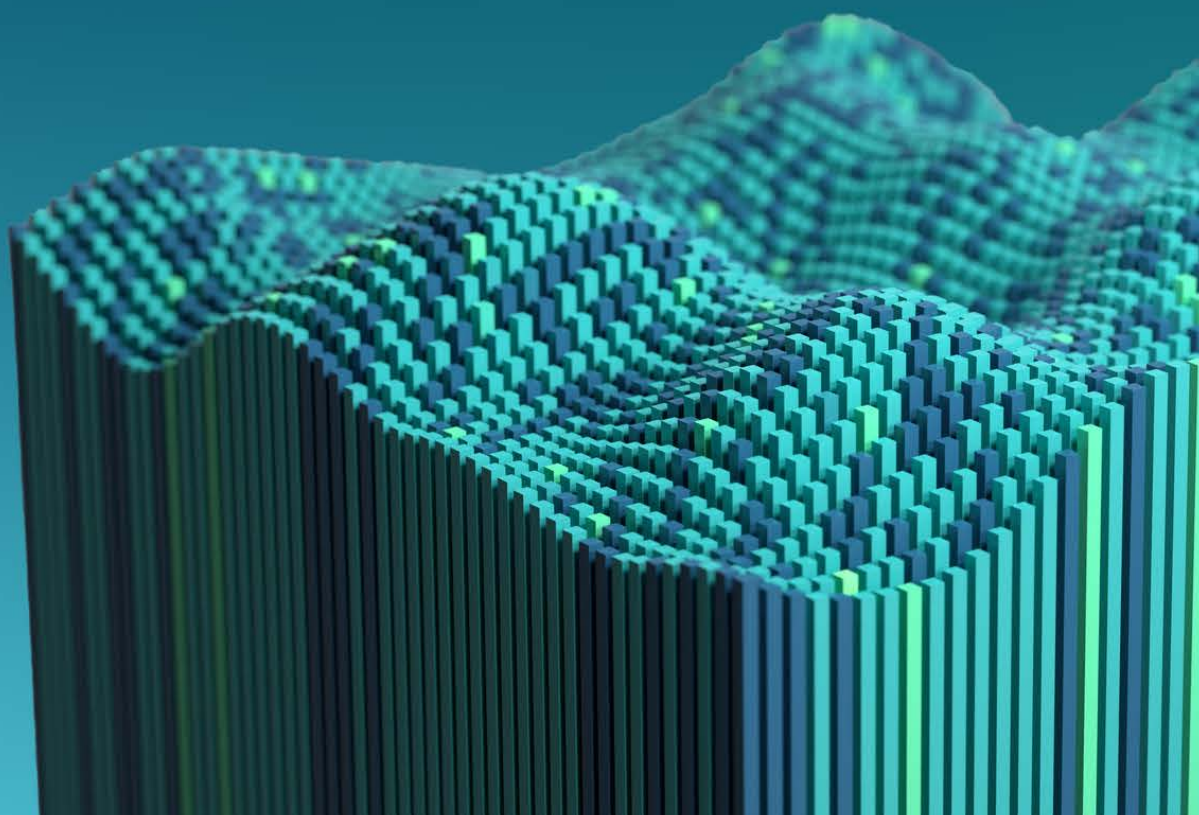
Table of Contents

Introduction

For decades, the relational database was the primary choice for storing data. New developers learned the SQL language and the virtues of database normalization. To reduce the learning curve of SQL or the tedium of mapping between paradigms, developers use object-relational mappers (ORMs) to translate between their live application and their durable storage. The relational database provided enough flexibility and ubiquity to be the default choice.

In the early 2000s, the ground shifted. New databases arose, often grouped under the term 'NoSQL'. While there were many flavors of NoSQL databases, the document database was the most popular. Document databases provided a flexible data model that more closely matched the objects used in application code. This allowed developers to avoid the famed impedance mismatch between the relational model and application objects. Further, document databases provided a flexible schema that allowed developers to evolve their data model over time without the pain of database migrations.

In this book, we'll look at how to model your data in a document database. We'll focus on Amazon DocumentDB, a fully managed document database service that is compatible with MongoDB. We'll start by looking at the differences between the relational model and the document model as well as some key principles for adopting the document model. Then, we'll look at the core API operations in DocumentDB. Finally, we'll look at some common data modeling patterns for DocumentDB.



aws

The relational model

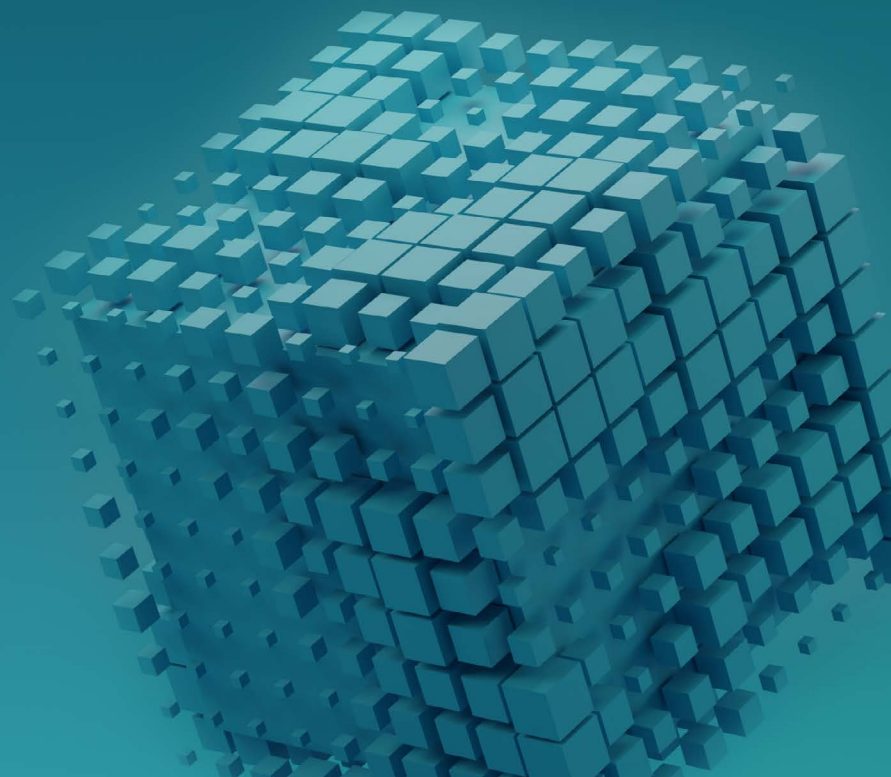
Before we dive into the details of data modeling in DocumentDB, let's take a step back and look at the relational and document models. How are these two models similar and where do they differ?

In a traditional relational model, each entity in your application is contained in a separate *table*. If your application has teams, users, and support tickets, you would have a `teams` table, a `users` table, and a `tickets` table. Each table would contain only the data for that entity.

Within each table, you would define a schema for the records contained in that table. This schema is made up of columns, and each column has a name and data type. In our `users` table, we might have a `username` column of type `string`, a `name` column of type `string`, an `email` column of type `string`, and a `created_at` column of type `datetime`. Each row in the table would contain the data for a single user, and each column would contain a single piece of data for that user.

Traditionally, column types would be simple scalar values like strings and numbers. If you had more complex data, like an array of values, you would split that data into a separate table and use a reference to that table in your original table. For example, a team will have multiple users. Rather than storing the users within the team record, each user record will point to its corresponding team record via a foreign key.

Thus, the three key characteristics of the relational model are (1) a fixed schema for each table, (2) a flat data model using scalar values, and (3) references across entities via foreign keys.



aws

Adapting to the document model

As we'll see, the document model does not have these three main characteristics of a relational database. But while there are significant differences between the document and relational models, you don't need to throw away all of your existing database experience.

You're still working with individual records made of attributes, and these records are grouped together within the database. The specifics of the terminology differ. An individual record is called a *document* in DocumentDB as compared to a *row* in a relational database. Within a document, individual attributes are called *fields* in DocumentDB instead of *columns* like in a relational database. Documents are kept together in *collections* rather than in tables.

The biggest difference between the relational and document models is in the flexibility of designing your data objects. This flexibility can be a benefit, as we'll see throughout this book, but you should be careful in how you use it. In general, there are two areas where the document model is more flexible than the relational model.

First, the document model provides more schema *flexibility* than the relational model. In a relational database, you must define a schema for your table before you can insert any data. This schema defines the columns that are available for each row, and the data type of each column. Once you've defined this schema, you can only insert rows that match the schema. If you try to insert a row with a column that doesn't exist in the schema, or a value that doesn't match the data type of the column, the database will reject the insert.

In contrast, you don't need to define a schema for a document database like DocumentDB. Documents in DocumentDB are self-describing, like JSON objects. This allows you to customize the shape of each document to match the needs of your application. This can prove particularly useful in situations with highly flexible data, such as a content management system that includes a variety of different types of content, each with their own set of fields.

Additionally, DocumentDB's schema flexibility reduces the operational pain of evolving your data model over time. In a relational database, you would need to perform table-level DDL operations to add or remove a column. Traditionally, this was a painful process that could require downtime for your application. Even with modern relational databases with online DDL, performing these operations can require significant resources on your database instance.

While this schema flexibility is useful, it does come with a cost. A relational database would do the work to ensure your data was in the format you expected. With a schemaless database like DocumentDB, these guardrails are gone. You'll need to do the work to enforce the schema of your application data. In the data modeling section below, we'll look at different approaches to managing your schema in a schemaless database like DocumentDB.

A second way that DocumentDB is more flexible than a relational database is in the way it selectively encourages *denormalization* of your application data.. In a relational database, you would typically normalize your data model following the principles of Edgar Codd. This often requires splitting your data into multiple tables and tracking references across tables using foreign keys. It also avoids using nested data structures, such as objects or arrays, in your data model. Rather, you would split these nested data structures into their own tables. To combine data from different tables, you would use joins at query time.

Database normalization is a useful technique that helps to avoid data anomalies and provide query flexibility. However, normalization has its downsides as well. It can reduce query-time performance as your join is hopping around multiple tables to assemble your required data. It can also increase the complexity of your application code if you need to perform multiple queries to assemble your data. Finally, it can be difficult to translate the normalized data model into the objects used by your application, which often use nested data structures.

In a document model, you use a more denormalized model when you don't need the benefits of normalization. You may duplicate data across multiple documents, particularly when that data is not changing frequently. This can change an expensive join operation to a simple lookup. Further, you may choose to use nested data structures. This selective denormalization can improve the performance of your queries and simplify your application code.

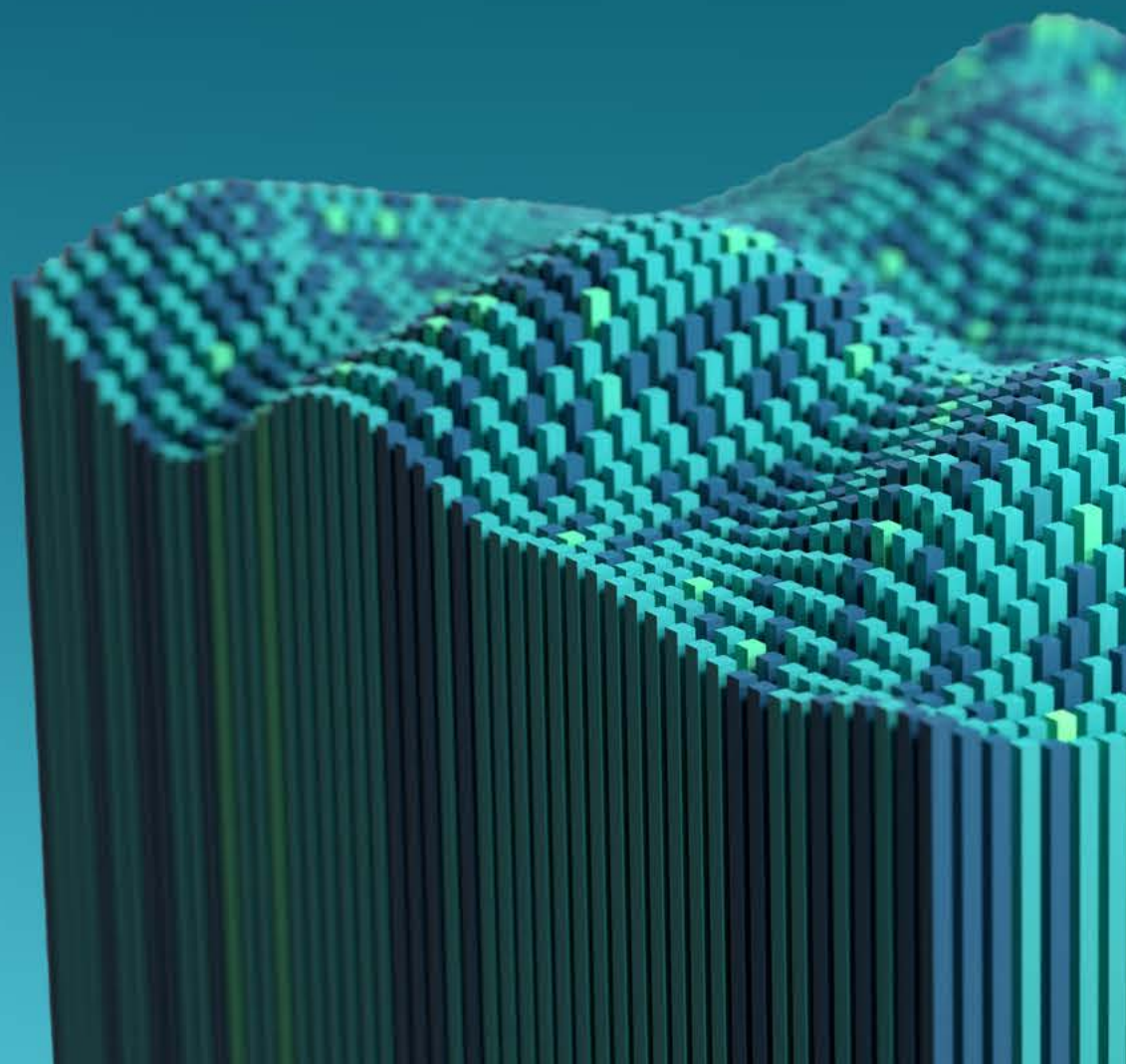
These two changes - schema flexibility and careful denormalization - make up the fundamental differences between the relational model and the document model.

In the rest of this ebook, we'll look how to adopt our data modeling techniques to take advantage of these differences. First, we'll start with an overview of the DocumentDB API. Then, we'll cover the three core aspects of proper data modeling in DocumentDB. Finally, we'll close with some advanced tips for using DocumentDB well.

The DocumentDB API

Just as there are differences in terminology and style between DocumentDB and a traditional relational database, there are also differences in the API. In this section, we'll look at core API operations in DocumentDB to get a feel for how to interact with the database. These operations will form the building blocks for the data modeling patterns we'll look at later.

DocumentDB provides compatibility with the MongoDB API. Thus, you can use the MongoDB client libraries as well as third-party ORMs and other tools. In the examples below, we'll use the Node.js SDK for MongoDB. However, the concepts are the same regardless of the SDK you use.



aws

Documents and the _id field

As noted, a single record in DocumentDB is called a *document*. This is comparable to a row in a relational database. As we've seen, a document is more similar to a JSON object than a record you're used to in a relational database. A document has no set schema by default and can contain nested data structures.

There is one caveat to DocumentDB's schemaless model: every document must have an '_id' field. This field is required for every document, and it must uniquely identify a single document in a collection. This _id field is similar to a primary key in a relational database.

You may provide a value for the _id field when inserting a document. This is useful when you have a natural key for your document, such as a username or email address. The _id field is indexed by default, so using a natural key for the _id field can take full advantage of this index.

If you don't provide an _id field for a document, DocumentDB will provide one for you. This will be in the form of an ObjectId, which is a twelve-byte unique identifier. The first four bytes of an ObjectId are a timestamp indicating when the object was created. In addition to providing uniqueness, this can be useful for sorting documents by creation time.

Finally, remember that each document must belong to a collection in DocumentDB. When interacting with documents, you must specify the collection that contains the document. Thus, you often have some initialization code to get a reference to the collection you want to interact with.

For example, the following code gets a reference to the `users` collection in the `main` database.

```
const db = client.db('main');
const collection = db.collection('users');
```

Then, you can use the `collection` variable to interact with documents in the `users` collection.

With this core understanding of documents, let's look at how you can interact with them in DocumentDB.

Inserting documents

The first operation we'll look at is inserting documents. This is the equivalent of and `INSERT` operation in a relational database. You can insert a single document or multiple documents at a time.

To insert a document into your database, you will use the `insertOne` method.

```
const result = await collection.insertOne({
  username: 'johndoe',
  name: 'John Doe',
  email: 'john@example.com',
  created_at: new Date(),
});

console.log(result.insertedId); // 65836d8ad4211546b47af455
```

You can provide a single object to the `insertOne` method. This will be serialized into a document and inserted into the collection. The `insertOne` method returns a result object that contains the `_id` of the inserted document as an `insertedId` property.

As noted above, you can specify the `_id` property yourself if you want. However, the write will fail if you specify an `_id` that already exists in the collection.

```
const result1 = await collection.insertOne({
  _id: 'johndoe',
  name: 'John Doe',
  email: 'john@example.com',
  created_at: new Date(),
});

console.log(result.insertedId); // johndoe

const result2 = await collection.insertOne({
  _id: 'johndoe',
  name: 'Jonathan Doe',
  email: 'john.doe12@example.com',
  created_at: new Date(),
}); // Throws error because _id already exists
```

You can also insert multiple documents at once using the `insertMany` method.

```
const result = await collection.insertMany([
  {
    username: 'johndoe',
    name: 'John Doe',
    email: 'john@example.com',
    created_at: new Date(),
  },
  {
    username: 'janedoe',
    name: 'Jane Doe',
    email: 'jane@example.com',
    created_at: new Date(),
  }
]);

console.log(result.insertedIds); // [ 65836d8ad4211546b47af455,
65836d8ad4211546b47af456 ]
```

When inserting multiple documents at once, the `insertedIds` property of the result object will be an array of the `_id` values for each inserted document, in the order they were inserted.

Reading documents

Inserting documents is all well and good, but our data is only really useful if we can read it back out. In this section, we'll look at how to read documents from DocumentDB.

To read documents from DocumentDB, you will use the `find` method.

```
const result = collection.find();

for await (const doc of result) {
  console.log(doc);
}
```

The result of the `find` method is a cursor object, similar to a cursor in a relational database. This cursor object is an async iterator, which means you can use it in a `for await ... of` loop to iterate over the documents in the result set. Under the hood, the `find` method is retrieving batches of records. As you iterate over the cursor, it will fetch the next batch of records as needed.

The example above returns all documents within a collection. This would be like a `SELECT * FROM users` query in a relational database. However, you often want to filter the documents during a particular query to only return the documents you need. You can do this by passing a filter object to the `find` method.

```
const result = collection.find({
  username: 'johndoe'
});

for await (const doc of result) {
  console.log(doc);
}
```

With this filter object, DocumentDB will only return documents where the `username` field is equal to `johndoe`. This would be like a `SELECT * FROM users WHERE username = 'johndoe'` query in a relational database.

When doing an exact match filter on the `username` property, it's likely you will receive only a single document back. In situations where you expect a single document, you can use the `findOne` method instead of the `find` method.

```
const result = await collection.findOne({
  username: 'johndoe'
});

console.log(result); // { _id: 65836d8ad4211546b47af455,
username: 'johndoe', ... }
```

Note that the `findOne` method returns a single document, not a cursor. This means you wouldn't use a `for await ... of` loop to iterate over the results. Rather, you can simply await the result of the `findOne` method.

In the examples above, we've used a simple exact match on the `username` field to locate matching records. However, you can use a wide range of operators to filter your results. Let's quickly look at a few of these operators.

A common requirement is to find matching records with a value greater than or less than a particular value. You can use the `$gt` and `$lt` operators to do this.

For example, you can use the `$gt` operator to find documents where the `created_at` field is greater than a particular date.

```
const result = collection.find({
  created_at: { $gt: new Date('2024-01-01') }
});
```

Notice that the value for the `created_at` field is an object with a `$gt` property. This is how you specify operators in DocumentDB. The value of the `$gt` property is the value you want to compare against. The use of a `$` is an indicator that this is an operator, not a field name.

If you want values that are greater than or equal to (or less than or equal to), you can use the `$gte` and `$lte` operators.

Another common requirement is to find matching records where a field is in a list of values. For example, we may want to find all support tickets that were created by a particular set of users.

You can use the `$in` operator to do this:

```
const result = collection.find({
  username: { $in: ['johndoe', 'janedoe'] }
});
```

This will return all documents where the `username` field is equal to either `johndoe` or `janedoe`.

As mentioned before, DocumentDB is a schemaless database that allows for nested data structures. You can query those data structures directly using dot notation -- referring to the nested field by the path from the root of the document.

For example, let's extend our `users` collection to include an `address` field with nested values:

```
{
  _id: 65836d8ad4211546b47af455,
  username: 'johndoe',
  name: 'John Doe',
  email: 'john@example.com',
  address: {
    street: '123 Broadway',
    city: 'New York',
    state: 'NY',
    zip: '11111'
  }
}
```

If you want to find all the users that live in a specific zip code, you can use dot notation to query the nested `zip` field.

```
const result = collection.find({
  'address.zip': '11111'
});
```

This will return all documents where the `zip` field in the `address` object is equal to `11111`.

Finally, while dot notation works well for nested objects, you may want to query for documents that contain a particular value in an array. For example, let's extend our `users` collection to include a `roles` field with an array of roles.

```
{
  _id: 65836d8ad4211546b47af455,
  username: 'johndoe',
  name: 'John Doe',
  email: 'john@example.com',
  address: {
    street: '123 Broadway',
    city: 'New York',
    state: 'NY',
    zip: '11111'
  },
  roles: ['admin', 'user']
}
```

If you want to find all the users that have a particular role, you can use the `$elemMatch` operator. With the query below, we can find all users with the `admin` role.

```
const result = collection.find({
  roles: { $elemMatch: { $eq: 'admin' } }
});
```

The `$elemMatch` operator is composable, so you construct a subquery that is applied to all elements in an array. DocumentDB will return any document where the subquery matches at least one element in the array.

There are a number of other operators available in DocumentDB, including regex matching, JSON schema matching, geospatial queries, and bitwise operations. For a full listing of compatibility with MongoDB APIs, see the DocumentDB documentation. For details on the operators available in DocumentDB, see the MongoDB documentation on query operators.

Sorting, projecting, and other options

When reading documents from DocumentDB, you may want to alter the returned results in some way. This could mean changing the order of a set of documents by including a preferred sort order. Additionally, it could mean limiting the fields returned in the result set to only those you need.

DocumentDB has a chainable API that allows you to specify these options. Let's look at a few of the most common options.

First, it's common to want to sort the results of a query. You can do this using the `sort` method.

If we wanted to return users in order of the most recently created, we could do this as follows:

```
const result = collection.find().sort({ created_at: -1 });
```

The `sort` method takes an object with the fields you want to sort on. The value of each field is either `1` for ascending order or `-1` for descending order. In this case, we want to sort by the `created_at` field in descending order, so we use `-1`.

You can sort by multiple fields by including multiple fields in the sort object. The documents will be sorted by the first field, then the second field, and so on.

In the example below, we can find users ordered by state, then by created date to find the most recent users in each state.

```
const result = collection.find().sort({ state: 1, created_at: -1 });
```

One of the easiest ways to improve the performance of your queries is to reduce the amount of data you're reading and sending back. If you're retrieving a set of records but have a maximum number of records you want to return, you can use the `limit` method.

```
const result = collection.find().limit(10);
```

This is powerful when combined with the `sort` method. For example, if you want to find the ten most recently created users, you can use the `sort` method to sort by `created_at` in descending order and then use the `limit` method to limit the results to ten. For top performance, index on your sort field to avoid a full collection scan. More on indexing can be found in the data modeling section below.

Finally, in addition to reducing the number of records sent back, you can also reduce the size of each record. If you don't need all the fields in a document, you can use the `project` method to specify which fields to include or exclude.

```
const result = collection.find().project({ username: 1, name: 1 });
```

The `project` method uses a similar syntax as the `sort` method, but it includes some explicit and implicit behavior. Essentially, you can explicitly specify the fields you want by listing them with a value of `1`. This implicitly excludes all other fields.

Alternatively, you can explicitly state the fields you *don't* want by listing them with a value of 0. This implicitly includes all other fields.

If you have large, sprawling documents but only need a subset of them, using a `project` operation can significantly reduce the amount of data you're sending back and forth. For large response bodies, this can significantly improve the performance of your application. Check out the indexing section below for more information on how to optimize this further by using covered queries.

Updating documents

In addition to writing new documents to DocumentDB, you'll often need to update existing documents. Perhaps a user wants to change their email address, upgrade their plan, or change their address. In this section, we'll look at how to update documents in DocumentDB.

Like the read operations, DocumentDB contains two core update methods: `updateOne` and `updateMany`. Like the read operations, the operation you choose depends on whether you want to update a single document or multiple documents.

Let's start with the `updateOne` method. This method takes two arguments: a filter object and an update object. The filter object is used to find the document you want to update. The update object is used to specify the changes you want to make to the document.

In the example below, we will update the `johndoe` user to have a new email address.

```
const result = await collection.updateOne(
  { username: 'johndoe' },
  { $set: { email: 'johndoe@example.com' } }
);
```

Notice our filter object -- `{ username: 'johndoe' }` -- is the same as the filter object we used to find the document in the section on reading documents. All the query syntax works the same for update operations as it does for read operations.

In the update object, we use the `$set` operator to specify the changes we want to make to the document. The `$set` operator takes an object with the fields you want to update. In this case, we want to update the `email` field to `johndoe@example.com.`

Sometimes you want to remove a field from a document altogether. You can do this using the `$unset` operator.

```
const result = await collection.updateOne(
  { username: 'johndoe' },
  { $unset: { email: '' } }
);
```

The `$set` and `$unset` operators will do most of the update work you need. However, there are other operators available as well, including `$inc` for incrementing a counter or `$rename` for renaming an existing field. You can find the full documentation on update operators here.

Note that the `updateOne` method will always modify a single document, regardless of how many documents are matched by your filter object. If multiple documents are matched, only the first one will be modified by the `updateOne` operation.

Sometimes you want to update multiple documents at once. This may be an application-level update like updating all users that belong to a specific team or organization, or it may be a schema migration where you want to update all documents in a collection.

In these cases, you can use the `updateMany` method. This method takes the same arguments as the `updateOne` method, but it will update all documents that match the filter object.

In the example below, we will rename the `email` field to `email_address` for all documents in the `users` collection.

```
const results = await collection.updateMany(
    {},
    { $rename: { email: 'email_address' } }
)
```

Notice that we've used an empty filter object. This will match all documents in the collection. This is similar to a `SELECT * FROM users` query in a relational database.

The `updateMany` operation can be helpful for changing multiple documents at once, but use caution when using it. Performing a write operation on large numbers of documents can be a resource-intensive operation. If you're updating a large number of documents, you may want to perform the update in batches to avoid overloading your database.

A final update operation is the `replaceOne` method. Remember that a document in DocumentDB is uniquely identified by its `_id` property. In the default update operations, the document will only alter the fields specified by the update object. All other properties on the document will remain unchanged.

Sometimes you want to replace the entire document with a new document. This is where the `replaceOne` method is useful. You can use it to completely overwrite an existing document with a new document.

Like the `updateOne` method, the `replaceOne` method takes two arguments. The first is a filter object that is used to identify the object to replace, and the second object is the new document to replace it with.

In the example below, we will replace the `johndoe` user with a new document.

```
const result = await collection.replaceOne(
  { username: 'johndoe' },
  {
    username: 'johndoe',
    name: 'Johnny Doe',
    email: 'johnny12@example.com',
  }
);
```

There are two important things to note with the `replaceOne` method. First, note that the replacement will occur for only the first matching document for the filter object. If you include a wide filter that matches multiple documents, only one will be updated. Thus, ensure you're specific with your filter object when using the `replaceOne` method. Ideally you are using the `_id` field or another unique field to identify the document to replace.

Second, notice that the `replaceOne` document takes in a full document rather than an update object with the update operators. This is more similar to the `insertOne` method than the `updateOne` method.

In each situation where using the update methods, consider what type of updates you want to make.

- Are you updating a single document or multiple documents?
- Do you want to update the document in place or replace it with a new document?

The answers to these questions will help you determine which update method to use.

Deleting documents

The final CRUD operation we'll look at is deleting documents. This is the equivalent of a `DELETE` operation in a relational database. You can delete a single document or multiple documents at a time.

Like the update operations, there are separate operations for deleting a single document and for deleting multiple documents. Let's start with the `deleteOne` method.

The `deleteOne` method takes a filter object as its only argument. This filter object is used to identify the document to delete.

```
const result = await collection.deleteOne({ username: 'johndoe' });
```

This will delete a single document where the `username` field is equal to `johndoe`. Like the `updateOne` method, it will only delete the first matching document if the filter matches multiple documents.

If you want to bulk delete items, you can use the `deleteMany` method. This can be good for cascading deletes or bulk cleanup operations.

In the operation below, we will delete all users that have not logged in for the past 90 days.

```
const result = await collection.deleteMany({
  last_login: { $lt: new Date(Date.now() - 90 * 24 * 60 * 60 * 1000) }
});
```

Just like the `updateMany`, think carefully about how many items will be affected before running a `deleteMany` operation. You don't want to tie up your database with a large delete operation.

Aggregation framework

The core CRUD operations we've looked at so far are the basic building blocks for working with DocumentDB. However, DocumentDB also provides some advanced tools for data querying. One of these tools is the aggregation framework.

The aggregation framework is a pipeline-based operation that allows you to perform a series of operations on a set of documents. Each operation in the pipeline takes the results of the previous operation and performs some transformation on it. The result of the final operation is returned to the caller.

One common example is to emulate JOIN-like functionality in a relational database by combining two documents via a reference.

Imagine you have one collection where every document has a `userId` field that points to the `_id` field of a document in the users collection. You might use the `aggregate` operation as follows:

```
const result = await collection.aggregate([
    {
        $match: {
            "user_id": '65836d8ad4211546b47af456'
        }
    },
    {
        $lookup: {
            from: "users",
            localField: "user_id",
            foreignField: "_id",
            as: "userInfo"
        }
    }
]);
```

In the `aggregate` operation, you provide an array of steps. These steps are performed in sequential order by the database engine. In this case, we have two steps. First, we use the `$match` operation to filter the documents in the collection to only those that match the provided filter. Then, we use the `$lookup` operation to join the `users` collection to our original collection.

Enriching data with a `$lookup` operation is a common use case for the aggregation framework. However, DocumentDB supports a number of other stage operations as well. Many of these operations compare to different operators in SQL.

We won't run through examples for all of the stages, but some of the more useful ones include:

- `$match`: Filters the documents in the collection to only those that match the provided filter (compare to a `WHERE` clause in SQL);
- `$project`: Allows you to specify which fields to include or exclude from the result set (compare to a `SELECT` clause in SQL);
- `$group`: Allows you to group documents together based on a field or fields (compare to a `GROUP BY` clause in SQL);
- `$unwind`: Allows you to explode arrays into individual documents per element in the array to work on those elements individually;
- `$bucket`: Allows you to group documents together based on a range of values (compare to a `GROUP BY` clause with a range in SQL);
- `$sort`: Allows you to sort the documents in the result set (compare to an `ORDER BY` clause in SQL);
- `$limit`: Allows you to limit the number of documents returned (compare to a `LIMIT` clause in SQL);
- `$skip`: Allows you to skip a number of documents in the result set (compare to an `OFFSET` clause in SQL).

See the Advanced Tips section below for more on using the aggregation framework well.

Transactions

Another advanced query feature of DocumentDB is support for transactions. Transactions allow you to perform multiple operations in a single, atomic operation. If one of your steps fails, the entire transaction is rolled back and no changes are made to the database. This is a common feature of relational databases but was missing in early versions of document databases.

There are a few common reasons you may need to use transactions. The simplest is to ensure that multiple operations are performed atomically -- that is, either all of the operations succeed or none of them succeed.

A simple example is in creating a user that will belong to a team. Perhaps we want to keep track of the total members of the team on the team object. To do so, we increment the `memberCount` field on the team object when we create a new user.

If we need this operation to be atomic, we can use a transaction to ensure that the user is created and the `memberCount` field is incremented in a single operation. If either operation fails, the entire transaction is rolled back and no changes are made to the database.

You could handle this using the following code:

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

try {
  session.startTransaction();

  // Create the user
  const usersCollection = client.db('main').
collection('users');
  await usersCollection.insertOne({ ... }, { session });

  // Increment the memberCount field on the team
  const teamsCollection = client.db('main').
collection('teams');
  await teamsCollection.updateOne(
    { _id: '65836d8ad4211546b47af456' },
    { $inc: { memberCount: 1 } },
    { session }
  );

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

This is complex, so let's walk through it step-by-step.

First, you create a session using the `startSession()` method on the client. This session will be used to perform the transaction.

You then call the `startTransaction()` method on the session to start the transaction. Remember that errors could occur during your transaction, so you'll want to wrap it in a try/catch block to handle the error and rollback the transaction.

Then, you would perform the actions you want to take. In this case, we're creating a new user and incrementing the `memberCount` field on the team.

To complete the transaction, you call the `commitTransaction()` method on the session to commit the transaction to the database.

Notice in our `catch` block that we call the `abortTransaction()` method on the session. This will roll back the transaction and undo any changes made during the transaction. Any changes made during the transaction will be discarded.

Finally, we use a `finally` block to close the session using the `closeSession()` method.

In every transaction you do, you must ensure you handle the preparation and clean up tasks. Start your session at the beginning, and close it when your work is through. Start a new transaction from your session, and commit or abort it when you're done. If you don't do this, you'll end up with orphaned sessions and transactions that will eventually time out, all while holding locks on your database.

Further, you should try to design your data model to avoid transactions where possible. Transactions require coordination, which can be expensive. Try to lean into a document model and avoid the need for transactions where possible. You can do this by using the patterns discussed below, including the embedded pattern to keep related data together.

Both the aggregation framework and transaction capabilities are powerful features of DocumentDB that give you some of the same power and flexibility of a traditional relational database. However, be sure to use these features sparingly. See where you can find a simpler, more scalable solution using the patterns discussed above.

Operations conclusion

In this section, we learned the basic CRUD operations in DocumentDB. These will be the bulk of your operations in DocumentDB, so you should understand them well. Specifically, think about how DocumentDB will match the documents you want to read, update, or delete. Work to ensure this is a fast, efficient operation. You can do this using some of the indexing patterns discussed below.

We also reviewed some of the advanced features of the DocumentDB API, such as the aggregation framework and transactions. These are powerful features that can help you solve complex problems. However, they can also be expensive operations that can slow down your database. Use them sparingly and consider whether there are simpler solutions to your problem.

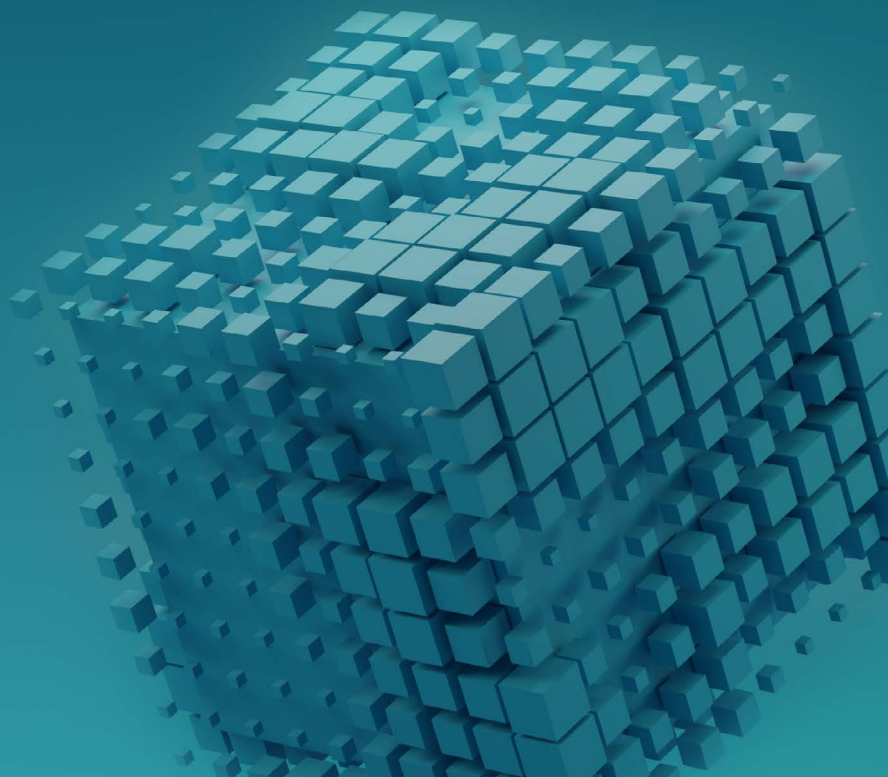
Data modeling patterns

Now that we know the basics of the DocumentDB API, let's look at some common data modeling patterns. The sections below provide tactical advice for real-life data modeling problems. These patterns are not exhaustive, but they should provide a good starting point for your data modeling efforts.

In general, three areas will make the biggest difference for success in DocumentDB:

- Schema management;
- Modeling relationships;
- Proper indexes.

Focus on getting these three areas right first, then focus on the more advanced topics at the end of this ebook.



aws

Schema management in DocumentDB

One of the best things you can do for a scalable and efficient data model is to maintain sanity for the shape of your existing documents. We saw above that DocumentDB is a schemaless database by default -- we won't be defining column names and data types for our documents like we would with a relational database. While that flexibility is useful, particularly as your schema evolves over time, you do want to manage some semblance of a schema for your documents.

There are two common patterns for managing your schema in DocumentDB. The first, more traditional, way is to verify your schema yourself in your application code. The second, more modern, way is to use JSON Schema to define a schema for your documents and use DocumentDB's validation feature to enforce that schema.

Let's review each of these patterns in turn.

Managing your schema in your application code

In the early days of NoSQL databases, there was no server-side schema management available. This meant the only pattern available to enforce your schema was via your application code. You would need to verify that your documents matched your expected schema before writing them to the database. To be safe, you'd likely want to verify the schema again when reading the document back from the database.

To manage this schema, you can use a generic schema validation tool like JSON Schema. Alternatively, you can use a language-specific tool such as Zod to define your schema and validate your documents. Some client libraries for DocumentDB, such as Mongoose for Node.js, include schema validation tools as well.

Let's look at an example of using Zod to validate a document before writing it to DocumentDB.

```
const userSchema = z.object({
  username: z.string(),
  name: z.string(),
  email: z.string().email(),
  created_at: z.date(),
});

async function createUser(userData) {
  const user = userSchema.parse(userData);
  return collection.insertOne(user);
}
```

In this example, we've defined a schema for our user documents using Zod. Then, in our `createUser` function, we parse the user data against the schema. If the user data does not match the schema, Zod will throw an error. Otherwise, we can safely insert the user into DocumentDB.

This pattern works well for managing your schema, but it does have some downsides. First, you must manage the schema validation yourself. This means carefully ensuring that, in every place you write to the database, you are validating the schema. This can be difficult to manage as your application grows.

Additionally, you may be doing update operations on your documents that alter only a portion of your document. For example, you may be updating a user's email address. In this case, you would want to validate that the email address is in the correct format, but you wouldn't want to validate the entire document. This requires careful management of your schema validation and a deep understanding of your schema requirements.

For this reason, you may prefer to use server-side validation of your schema instead. Let's explore that next.

Managing your schema with DocumentDB's JSON Schema validation

A second approach to schema validation is to have DocumentDB validate your documents via its JSON schema validation feature. This allows you to define a schema for your documents and have DocumentDB validate that schema before writing the document to the database.

A schema is defined at the collection level and can be added when creating the collection or added to an existing collection. Let's look at an example of creating a collection with a schema.

```
const result = await db.createCollection('users', {
  validator: {
    $jsonSchema: {
      bsonType: 'object',
      required: ['username', 'name', 'email', 'created_at'],
      properties: {
        username: {
          bsonType: 'string',
        },
        name: {
          bsonType: 'string',
        },
        email: {
          bsonType: 'string',
          pattern: '^.+@.+$',
        },
        created_at: {
          bsonType: 'date',
        },
      },
    },
  },
  "validationLevel": "strict", "validationAction": "error"
});
```

Now, when you insert a document, DocumentDB will throw an error if the document does not match the schema.

This schema validation will also apply during updates. In general, this greatly simplifies the management of your schema. As long as you can describe it correctly via JSON schema, you'll be able to validate it with DocumentDB.

DocumentDB also allows for updates to your schema as your application evolves. You can update your schema at any time by using the `collMod` command. Note that this will not retroactively change or validate any existing documents -- you will be responsible for making corresponding changes yourself.

If you have existing documents that will not pass your new schema validation rules and you don't want to update them, you can reduce the validation level of your schema. By setting `validationLevel` to `moderate`, DocumentDB will only apply schema validation to new documents and to existing documents that are valid before the update. If you have prior versions of documents that are invalid, they will not be validated during updates.

The `moderate` validation level can ease the pain of updating your schema, but it does come with a cost. If you have invalid documents in your collection, you may not be able to trust the data in your collection. You'll need to perform additional, application-side validation and transformations to handle invalid documents. For this reason, it's best to keep your schema validation level at `strict` if possible.

Using server-side validation may seem like we've come full circle. Part of the reason document databases gained in popularity was their flexible schema, but now we're back to enforcing it in the database again! Note, however, that DocumentDB still has a more flexible schema system than a traditional relational database. It natively allows for nested data types like objects and arrays. Further, updating your schema is a much easier operation in DocumentDB. It doesn't require locking your table or costly background resources.

In general, the server-side, JSON-schema-based approach to schema validation will be easier to manage than the application-side approach. Whichever pattern you choose, enforcing some schema in your application will make it easier to trust the data in your DocumentDB database.

Managing relationships in DocumentDB

One of the biggest changes in moving from a relational database to a document database is how you manage relationships between data. Many people incorrectly think that 'relational' in a relational database refers to relationships between records. This is not true. Rather, the relational model refers to the fact that data is stored in tables with rows and columns and has more in common with set theory.

Taking this misconception further, some state that a "non-relational" database like DocumentDB cannot handle relationships between data. This is also not true. DocumentDB can handle relationships between data. In fact, it's almost meaningless to consider data without relationships. However, you do need to handle relationships differently in DocumentDB than you would in a relational database.

Let's start by reviewing what relationships (not relations!) are and how they're handled in a relational database. Then, we'll look at how to handle relationships in DocumentDB.

A relationship describes some connection between two pieces of data. For example, a user may belong to a team or organization, a support ticket may be assigned to a user, or a blog post may have comments. These are all examples of relationships between data.

In a relational database, you commonly model different data entities as separate tables. In our example relationships above, we would have a `users` table, a `teams` table, a `support_tickets` table, a `blog_posts` table, and a `comments` table. Each table would have its own set of columns, and each row in the table would represent a single record.

To indicate relationships between different entities, you could use foreign keys. A foreign key uses a *reference* in one table to identify a record in another table. For example, the `support_tickets` table may have a `user_id` column that references the id column in the `users` table. This would indicate that the support ticket belongs to a particular user.

When reading these items back, you would use the `JOIN` operation to combine data from multiple tables. For example, imagine that you want to find all the support tickets for a particular user. You don't know the ID for each support ticket for that user, but you do have the user's username. You could use a `JOIN` operation to find all the support tickets that have a `user_id` that matches the user's ID.

```
SELECT * FROM support_tickets
JOIN users ON support_tickets.user_id = users.id
WHERE users.username = 'johndoe';
```

In this case, we're linking the two tables together using the foreign key relationship. This allows me to filter the `support_tickets` table on a field that's only present in the `users` table.

DocumentDB does include a `JOIN`-like operation -- `$lookup`, via the aggregation framework discussed above -- but you should avoid it where possible.
Rather, lean in to a document model by using the relationship patterns common in document databases.

There are three common patterns for modeling relationships in DocumentDB: embedding, duplicating, and referencing. Let's look at each of these in turn.

Managing relationships with embedding

The first pattern for managing relationships in DocumentDB is to embed the related data directly in the document. As we've seen throughout this book, DocumentDB allows you to store nested data structures in your documents. Using the embedding pattern takes advantage of this feature.

One example where you could use embedding is in selling a book in an online store. A single book may have a few different formats -- hardcover, softcover, ebook, and audiobook. Rather than storing these as four separate documents, you could embed the different formats directly in the book document.

```
{
  _id: 65836d8ad4211546b47af455,
  title: 'The Hobbit (75th Anniversary Edition)',
  formats: [
    {
        ISBN: '978-0547928227',
        type: 'hardcover',
        price: 19.99,
        pages: 304
    },
    {
        ISBN: '978-0547928210',
        type: 'softcover',
        price: 12.99,
        pages: 410
    },
    {
        ISBN: '978-0547928241',
        type: 'ebook',
        price: 9.99,
    },
    {
        ISBN: '978-0547928234',
        type: 'audiobook',
        price: 19.99,
        lengthInMinutes: 912
    }
  ]
}
```

Now, in fetching a book, you can retrieve all the formats for the book in a single query by using an indexed field like the book's title:

```
const result = await collection.findOne({ title: 'The Hobbit (75th Anniversary Edition)' });
```

The embedded pattern works best in the following situations:

- **You often require the related data when retrieving the parent record.** In our example above, users will be searching for a particular book and reviewing its details. In doing so, we want to display information on all available formats so they can choose the one that best fits their needs. By embedding the formats directly in the book document, we can retrieve all the data we need with a single lookup on the book's title.

- **The related data is not retrieved directly.** This factor pairs with the previous one, but if you are only retrieving the embedded data through the parent item, the embedded pattern makes sense. You won't need to index the embedded data separately, and you can get the benefits of indexing the parent item to retrieve the embedded data.

- **The number of related items is limited**. With databases in general, we want to prevent reading data that we don't actually need. If you embed data that is unbounded, your documents will grow in size and result in slower performance. In our example above, we have a limited number of formats for a book. However, if we were to embed all the reviews for a book in the book document, we could end up with a large number of reviews. This would result in a large document that would be slow to read from disk. In this case, we would want to use one of the other patterns.

These factors will be true in a number of situations with related data. In those situations, using the embedded pattern is a great way to simplify your application code by saving your application objects directly to the database. Further, you can enhance performance by avoiding the need for database joins or multiple queries to the database.

Handling relationships with duplication

While the embedded model is popular, there are a number of situations where it doesn't work well. Most commonly, it doesn't work well when the number of related items is unbounded, as your document size will grow as the number of related items grows. This larger document size will result in slower performance.

Additionally, the embedded model doesn't work as well with many-to-many relationships. In that situation, a piece of data will be related to multiple other pieces of data. For example, a user may belong to multiple teams, or a support ticket may be assigned to multiple users. In these situations, you can't embed the related data in a single document.

In situations like these, you can use the duplication pattern to duplicate data across multiple documents. Like the embedded pattern, this is a violation of the normalization principles of the relational model. To achieve third normal form, you should not duplicate data across multiple records. However, when used correctly, duplication can be a powerful tool for improving performance in DocumentDB.

Think back to the initial example in the introduction to this managing relationships section. We had a SQL query that retrieved all the support tickets for a particular user as identified by the username. We used a `JOIN` operation to combine the `support_tickets` and `users` tables to find the matching support tickets.

```
SELECT * FROM support_tickets
JOIN users ON support_tickets.user_id = users.id
WHERE users.username = 'johndoe';
```

We had to perform this `JOIN` operation because the information we had, the username, was only available in the `users` table.

With the duplication pattern in DocumentDB, we could avoid this `JOIN` operation by duplicating the username in our support ticket documents. This would allow us to query the `support_tickets` collection directly to find all the support tickets for a particular user.

A support ticket document could look as follows:

```
{
  _id: 65836d8ad4211546b47af455,
  title: 'Support ticket title',
  description: 'Support ticket description',
  text: '....',
  user: {
    _id: 65836d8ad4211546b47af456,
    username: 'johndoe',
    name: 'John Doe',
    email: 'john@example.com'
  }
}
```

This would allow us to make the following query to find all the support tickets for a particular user:

```
const results = collection.find({ 'user.username': 'johndoe' });
```

We've avoided the `JOIN` operation and gone directly to the support documents to find this information. This can be a powerful performance optimization, especially because DocumentDB allows you to index nested documents.

In addition to using duplicated data to help locate related records, you can also duplicate data that needs to be displayed for a given record. For example, imagine our online bookstore wants to show the purchase method used when a customer reviews an order. Rather than including a pointer to a specific payment method, we can simply duplicate that data onto the order record itself.

```
// Example order document
{
    _id: '658c32831bd77b6c38e72615',
    purchaseDate: '2023-12-27T14:19:47.000Z',
    total: 19.99,
    items: [ ... ],
    purchaseMethod: {
        type: 'credit_card',
        cardType: 'Visa',
        lastFour: '1234'
    }
}
```

In our order document, we have a `purchaseMethod` field that contains the details of the payment method used for the order. This allows us to display the payment method on the order without needing to perform a `JOIN` operation to retrieve the payment method. We aren't retrieving the order by the payment method directly, but we are duplicating it to avoid an additional read.

Like all data modeling patterns, there are tradeoffs to the duplication pattern. The benefits of normalization's preference for a single, canonical source of truth is in the consistency of the data. If a data record is updated, all other records that refer to that record will see the benefits of the update when they fetch the canonical record.

On the other hand, if you duplicate data, you'll have to manage and handle corresponding updates to all duplicates. Failure to do so can lead to data inconsistencies and a confusing user experience.

The key factors to look at here are (1) whether the duplicated data is immutable, and (2) how widely the data is duplicated. Let's explore these factors in the context of our examples.

For our first example, we duplicate the user record into our support ticket to handle searching by username. In many applications, the username is immutable. This makes it much safer to duplicate this data as we won't need to update it in the future.

If you look closely at our example, we also duplicate the user's name and email address. These are attributes that are more likely to be mutable and thus require corresponding updates to the duplicated data. Because support tickets are essentially unbounded, this could be an expensive operation for our application. For this reason, we may decide *not* to duplicate these mutable attributes onto our support ticket documents.

For our second example, we duplicate the payment method onto the order document. This *seems* like data that may be mutable, as a user can change their payment methods over time. However, think about it from the perspective of the order -- once an order is placed, the payment method *for that order* does not change. Thus, we can safely duplicate the payment method onto the order document.

The duplication pattern is a powerful pattern and another example of how selective denormalization can improve performance in DocumentDB. However, be sure to consider the tradeoffs of this pattern before using it in your application.

Managing relationships with referencing

The final pattern for managing relationships in DocumentDB is to use references. This is the most similar to the relational model, as you are using a reference to identify a related record.

In general, the reference pattern works well when the embedded or duplication patterns do not fit. If you have a large or unbounded set of data that is mutable, it may not be a good fit for the embedded or duplication patterns. Further, if you're likely to fetch a related record directly, outside of the context of its parent record, the reference pattern may be a good fit. Finally, if you have a many-to-many relationship, the reference pattern is a good fit.

If you use a reference pattern, there are generally two ways to fetch your data. You can use DocumentDB to perform the join for you using the `$lookup` operation, or you can perform multiple queries to fetch the related data.

Let's look at an example of using the `$lookup` operation using the aggregation framework discussed in the previous chapter.

In the support ticket example from the duplication section, we duplicated the user record onto the support ticket to allow us to search for support tickets by username. However, we noted that the user's name and email address are mutable and thus may not be a good fit for duplication.

We can update our support ticket document to look as follows:

```
{
  _id: 65836d8ad4211546b47af455,
  title: 'Support ticket title',
  description: 'Support ticket description',
  text: '....',
  user: {
    _id: 65836d8ad4211546b47af456,
    username: 'johndoe',
  }
}
```

We still copy the `username` and `_id` reference in the user field, but we don't copy the name and email address, but we no longer include the more mutable name and email address fields.

If we need the user's name and email address when fetching a support ticket, we can use the `$lookup` operation to join the `users` collection to the `support_tickets` collection.

```
const result = await collection.aggregate([
    {
        $match: {
            "user._id": '65836d8ad4211546b47af456'
        }
    },
    {
        $lookup: {
            from: "users",
            localField: "user._id",
            foreignField: "_id",
            as: "userInfo"
        }
    }
]);
```

Note that this operation does two steps which are performed sequentially. First, it matches all the support tickets for a particular user. Then, it uses the `$lookup` operation to join the `users` collection to the support tickets collection. This will return all the `support tickets` for a particular user, along with the user's record.

This simplifies our application logic but pushes the compute to our database.

To reduce the compute load on our DocumentDB cluster, we could perform two queries instead. First, we could query the `support_tickets` collection to find all the support tickets for a particular user. Then, we could query the `users` collection to find the user's record.

```
const tickets = collection.find({ "user._id":
'65836d8ad4211546b47af456' });
const user = users.findOne({ _id: '65836d8ad4211546b47af456' });
```

In this case, we're able to perform both queries in parallel, which will reduce the overall time to fetch the data. In some situations, you may need to do the queries sequentially. For example, imagine we fetch a support ticket by its `_id` and then want to fetch the user's record. In this case, we would need to wait for the first query to complete. Then, once we have the `user._id` value, we can perform the second query.

While the selective denormalization patterns of embedding and duplication are often preferred in DocumentDB, the reference pattern can be a good fit in certain situations. Be sure to consider the tradeoffs of each pattern before deciding which one to use.

Indexes in DocumentDB

Good performance in DocumentDB relies on proper data modeling, and data modeling is heavily influenced by the indexes you create. In this section, we'll see why indexes are important and how to create them in DocumentDB. Then, we'll learn about the different types of indexes and how you can use them to solve your needs.

In previous sections, we retrieved a user by their username. By default, there is not an index on the `username` field. This means that DocumentDB must scan the entire collection to find the matching document. This results in a ton of data being read from disk and eventually discarded.

Let's do some quick math to see this in action. Let's assume that each document in our `users` collection is 1KB in size. If we have 1 million users in our collection, that means we have 1GB of data in our collection. If we want to find a user by their username, DocumentDB must read all 1GB of data from disk to find the matching document. Once the matching document is found, we'll throw away the other 999,999 documents that we read from disk. That's a lot of wasted work!

Further, 1GB is a pretty small collection. For a large application, you might have collections that surpass hundreds of GBs or even terabytes of data. In those cases, you can see how inefficient it would be to scan the entire collection to find a single document.

To avoid this, we can create an index on the `username` field. This will allow DocumentDB to find the matching document without scanning the entire collection. The index will not only be much smaller than the full dataset, as only the `username` field is indexed, but it will also be sorted by the `username` field. This allows DocumentDB to use a binary search to find the matching document, which is much faster than scanning the entire collection.

Let's create an index on the `username` field.

```
await collection.createIndex({ username: 1 });
```

In specifying the index, you must provide an object with the field you want to index as the key. The value of the key is the sort order of the index. For a single field index like this one, the sort order doesn't matter too much as there is only one field to sort, and the index can be used for queries in either direction. However, for compound indexes discussed below, the sort order can be important.

You can use single-field indexes on any field in your document, including nested fields. Like when querying nested fields, you use the dot notation syntax to specify the nested field.

For example, if we wanted to index an embedded zip code field, you could do it as following:

```
await collection.createIndex({ 'address.zip': 1 });
```

We'll look at additional index types below, but first let's think about the process when creating an index. When creating an index in DocumentDB, the default setting is to create the index in the foreground. This means DocumentDB will obtain a lock on the collection and prevent all write operations to the collection while the index is being built.

If you have an existing collection with a large amount of data that's serving live traffic, this will result in downtime for your users. Instead of doing a foreground index build, you can do a background build by passing the `background: true` option to the `createIndex` method.

```
await collection.createIndex({ username: 1 }, { background: true });
```

A background index build will take longer but will avoid downtime for your users. This is the recommended approach for building indexes on existing collections.

In addition to the simple single-field indexes, DocumentDB also supports more advanced indexing methods. Let's look at each of those and see when you might want to use them.

Compound indexes

The second most common type of index is a compound index. With a compound index, you're indexing multiple fields in a single index. This is good for situations where you're querying on multiple fields at the same time.

In the users example we've been using, imagine you want to find all adult users in a particular zip code. To do this, you would do an equality query on the `address.zip` field and a range query on the `birthdate` field, as follows:

```
const results = collection.find({
  'address.zip': '12345',
  'birthdate': { $lte: new Date() - 18 * 365 * 24 * 60 * 60 *
1000 }});
```

If you had a single field index on the `address.zip` field, it would be able to find all the users in a particular zip code. However, it would then need to scan all the users in that zip code to find the ones that are adults. This could be a slow operation.

Instead, you can create a compound index on the `address.zip` and `birthdate` fields. This will allow DocumentDB to find all the users in a particular zip code and then find the ones that are adults. This will be much faster than scanning all the users in the zip code.

To create a compound index, you pass an object with the fields you want to index as the key. Like with a single-field index, the value of the key is the sort order of the index.

```
await collection.createIndex({ 'address.zip': 1, 'birthdate': 1 });
```

In this case, we're creating a compound index on the `address.zip` and `birthdate` fields. The index will be sorted first by the `address.zip` field and then by the `birthdate` field.

Proper configuration of your compound indexes can greatly reduce your query times, but subtle mistakes can avoid the power of the index. There are two tricks to make your compound indexes go from good to great.

First, understand that a properly configured compound index can support multiple query patterns. For example, you could query on the `address.zip` field alone, or you could query on the `address.zip` and `birthdate` fields together. However, you cannot query on the `birthdate` field alone. Thus, think carefully about your actual query patterns when creating your compound index. This is covered further in the Advanced Tips section.

A second tip for your compound indexes is to try to provide covered queries if possible. Like most databases, DocumentDB stores full records separately from the index. This means that, when you query on a field that's not in the index, DocumentDB must fetch the full record from disk. When applied to a large number of records, this can greatly increase the latency of your operation.

To avoid this, you can try to provide a covered query. A covered query is one where all the fields you need are in the index. This allows DocumentDB to retrieve the data directly from the index without needing to fetch the full record from disk.

To achieve a covered query, you'll need to provide a projection that includes only the fields you need. In our example above, if we only need the zip code and birthdate of users that match our query, we could do the following:

```
const results = collection.find({
  'address.zip': '12345',
  'birthdate': { $lte: new Date() - 18 * 365 * 24 * 60 * 60 * 1000 }
})
.project({ 'address.zip': 1, 'birthdate': 1 });
```

Because we're only asking for fields that are in the compound index, DocumentDB can handle this query without needing to fetch the full record from disk.

The example here is a simple one, but the harder problem is when you need additional fields from the document that aren't required in your query filter. For example, perhaps we also need the `username` field in this query. We could add this field at the end of our compound index, even though we don't use it for querying, in order to achieve a covered query. However, this results in a larger index and more work for DocumentDB to maintain the index.

Additionally, due to DocumentDB internals, the efficacy of a covered index may drop with a write-heavy workload.

In aiming for a covered query, think carefully about these tradeoffs. An additional field or two in the index may not be a big deal, but if you're adding a large number of fields to the index, you may want to reconsider your approach.

Multi-key indexes

In multiple places above, we've seen that DocumentDB allows you to store arrays in your documents. This is a powerful feature that allows you to store nested data structures in your documents. However, it does present a challenge when indexing your data -- how can you query on an array field?

Fortunately, DocumentDB allows you to query array fields by creating a multi-key index. A multi-key index is an index on an array field that creates an index entry for each element in the array. This allows you to query on the array field and have DocumentDB return all the documents that match the query.

Think back to our bookstore example. We have a book document that uses the embedding pattern to store the related formats for a book.

```
{
  _id: 65836d8ad4211546b47af455,
  title: 'The Hobbit (75th Anniversary Edition)',
  formats: [
    {
        ISBN: '978-0547928227',
        type: 'hardcover',
        price: 19.99,
        pages: 304
    },
    {
        ISBN: '978-0547928210',
        type: 'softcover',
        price: 12.99,
        pages: 410
    },
    {
        ISBN: '978-0547928241',
        type: 'ebook',
        price: 9.99,
    },
    {
        ISBN: '978-0547928234',
        type: 'audiobook',
        price: 19.99,
        lengthInMinutes: 912
    }
  ]
}
```

Note that each format has an ISBN field that is a unique identifier for the book. It's common that we'll want to find a book by the ISBN of one of its formats.

To do this, we can create a multi-key index on the `formats.ISBN` field.

```
await collection.createIndex({ 'formats.ISBN': 1 });
```

Now, we can efficiently query for a book by the ISBN of one of its formats.

```
const results = collection.find({ 'formats.ISBN': '978-0547928227' });
```

These multi-key indexes are very powerful in DocumentDB as they allow us to use the embedded pattern to keep related data together while still allowing us to query on the embedded data directly.

Sparse indexes

When creating your index, you can pass additional properties to configure the index. One of these properties is the `sparse` property. This property allows you to create a sparse index, which is an index that *only* includes documents that have the indexed field.

Using a sparse index can greatly improve your performance and reduce the size of your index in the right circumstances. You may even want to alter the structure of your documents to take advantage of a sparse index.

Let's think of our support tickets example from above. Imagine that we often want to find all the support tickets for a user, sorted by the date they were created. In fetching these, we only want to show open support tickets.

Over many years, our database will gather lots of support tickets. However, most of these support tickets will be irrelevant to our users as they're for old issues that have been resolved.

You may think to create an index like the following:

```
await collection.createIndex({ 'status': 1, 'user._id': 1, 'created_
at': -1 });
```

In this, we're creating a compound index on the `status`, `user._id`, and `created_at` fields. While this would work, it's also bloating the size of our index to include all the closed tickets that we don't care about. This will result in a larger index and slower performance.

A second solution is to delete the closed support tickets, but often this won't fit with your business requirements. You may need to keep historical data for archival or reporting purposes.

Instead, we can alter our document slightly. For *open tickets only*, we can add a `open_created_at` field that is a duplicate of our `created_at` field. Then, we can create a sparse index on the `user._id` and `open_created_at` fields.

```
await collection.createIndex({ 'user._id': 1, 'open_created_at': -1
}, { sparse: true });
```

Notice that we passed `{sparse: true}` as the second argument to the `createIndex()` method. This tells DocumentDB to create a sparse index, which will only include documents that have values for all fields in our index. Because only open tickets will have the `open_created_at` field, only open tickets will be included in the index.

To utilize a sparse index, you must pass the `{$exists: true}` operator on the indexed field to tell the DocumentDB query engine that you only want documents where the field exists.

```
const results = collection.find({ 'open_created_at': { $exists:
true } });
```

Ideally your sparse index will use actual fields on your documents rather than fields that are constructed solely for the index. However, don't be afraid to use this pattern if it fits your use case. It can greatly improve your performance and reduce the size of your index.

Advanced tips

Once you have figured out the core aspects of schema management, relationship modeling, and index optimization in DocumentDB, you can get pretty far. But as you start to push DocumentDB further, you might need more advanced optimizations.

This section includes advanced tips for improving your DocumentDB performance as you scale. We'll look at how to use the aggregation framework well and how to think about scaling your DocumentDB cluster. Additionally, we'll look at tips to reduce your I/O consumption and your overall document size. These tips will take you to the next level with DocumentDB.

aws

Use the aggregation framework wisely

In the section on the DocumentDB API, we saw that the aggregation framework is a tool for complex operations in DocumentDB. This can be both a blessing and a curse. On one hand, you can use the aggregation framework to perform advanced, multi-step operations that would be difficult to manage with a bunch of ad-hoc queries. On the other hand, you can burn through a lot of CPU and I/O with an unoptimized query.

Remember that the aggregation framework allows you to specify multiple steps to be performed sequentially by DocumentDB. Sometimes, DocumentDB may be able to condense multiple steps together in order to reduce processing by the engine. However, you shouldn't count on that and should work to optimize the overall query as much as possible.

In general, you should try to reduce the amount of data being passed between steps. Efficient database usage is all about filtering down to the relevant data as much as possible, and this is particularly critical in the aggregation framework. Think about ways to cull your dataset earlier in the process.

One way to do this is to use the `$match` operator early in your query. The `$match` operator is similar to a SQL `WHERE` clause, and it will filter out records that are unnecessary. The earlier you can do that -- ideally by using an index -- the better.

In addition to `$match`, the `$project` operator is a good way to reduce your data size. If you have documents that are a few KB in size but you only need a few small properties for your aggregation query, use the `$project` operator to select just those properties early on. This will reduce CPU usage for later stages. Likewise, the `$group` and `$bucket` operators help to reduce batches of records into a smaller summary.

Finally, once you have reduced the data set, then apply the `$sort` and `$limit` operators on the remaining records. This will greatly reduce the time to sort and, ideally, avoid spilling to disk to perform the sort operation.

Focus on this core principle -- *reduce your dataset as early as possible* -- in order to keep your aggregation queries performant and efficient.

Scaling with DocumentDB

As your application and DocumentDB usage grows, you may find yourself needing to scale your DocumentDB cluster. At first, this can be as simple as increasing your DocumentDB instance size. This is an operation that the DocumentDB service will manage for you with minimal downtime.

As your needs continue to grow, you may consider horizontal scaling by increasing the number of instances in your DocumentDB cluster. Here, it is beneficial to know more about DocumentDB's underlying architecture.

In many modern databases, a common remedy to scaling is to horizontally scale by sharding your data. In sharding, you split your entire dataset into shards, each of which will hold a subset of the entire dataset. This sharding is typically based on a highly used attribute in your application, such as a userID or tenantID, that allows requests to be routed to a single shard for processing.

With these horizontally scalable databases, you might shard your database for three reasons:

- Increasing *write throughput*, as you can spread writes across more instances in the cluster;
- Increasing *read throughput*, as you can spread reads across more instances; or
- Increasing *storage capacity*, as each instance won't need to hold the entire dataset.

DocumentDB does provide sharding via Elastic Clusters, but you won't need to jump to sharding as quickly as you will with other database systems. In fact, many databases that are sharded on other databases will find they can remove sharding altogether by migrating to DocumentDB as it has a more scalable architecture that separates compute and storage allowing for storage to elastically grow.

Let's run through each of the reasons for sharding above and see how you can handle these with DocumentDB.

First, you will rarely need to shard your DocumentDB cluster to account for additional storage capacity. DocumentDB's storage capacity grows automatically as you use it, up to a maximum capacity of 128 TiB. This is enough for the vast majority of use cases. That said, if you do require additional storage, an elastic cluster can scale its storage to a maximum of 4 PiB.

Second, sharding is usually not the correct approach to increase your read throughput with DocumentDB. DocumentDB allows you to create up to *15 read replica* instances in your DocumentDB cluster. These instances only handle reads and are an easy way to provide additional read capacity to your application. Because DocumentDB uses a shared underlying storage volume, read replicas can be created quickly, regardless of the size of your dataset. Further, there's no impact on the primary instance in your cluster.

You should note that replication to read replicas is asynchronous and thus may be slightly out of date with your primary instance. For most applications, this eventual consistency is fine. If you require stronger consistency guarantees on reads, you can direct certain reads to the primary instance. Further, DocumentDB allows you to monitor replication lag via the `DBInstanceReplicaLag` metric in CloudWatch.

Finally, if you need to increase the write throughput of your DocumentDB cluster and you've decided against increasing the instance size of your primary instance, you can use DocumentDB elastic clusters to shard your data.

In sharding your database by using elastic clusters, you are adding an additional tier to your database infrastructure. DocumentDB will add a request router layer that will be the primary point to handle your request. After the request router parses the request, it will forward the request to the relevant shard(s) to process the request and return the result to your client. Note that, because of this additional request router tier, there may be a slight increase in overall latency when moving to an elastic cluster.

In creating your elastic cluster, you'll need to choose which collections are sharded. For those that are unsharded, the entire collection will be located on a single shard in your elastic cluster. For collections that are sharded, they will be split across the instances according to a shard key that you specify.

Choosing a shard key for your collection is very important to see the benefits of elastic clusters. You'll want to choose a shard key that is evenly distributed across your dataset so that the data ends up being well-distributed across your shards. If you use an unbalanced shard key, you won't get the full benefit of splitting up your data.

Additionally, you want to use a shard key that correlates with your access patterns. Ideally you are using a shard key that contains an exact match in all or most of your database operations. This way, your operations can be directed to a single shard to handle the request rather than doing a scatter-gather operation across all of the shards. Again, this will allow you to take full advantage of the decision to shard.

Finally, elastic clusters can also be helpful in the rare case where you need to increase the number of connections to your DocumentDB cluster. While a single DocumentDB instance maxes out at 30,000 open connections, an elastic cluster supports up to 300,000 open connections.

DocumentDB provides a number of mechanisms to scale your database to meet your usage. In general, try to avoid sharding your database with elastic clusters where you can, due to the extra latency and planning work that requires. In the event that you do need to scale via sharding, DocumentDB provides a straightforward mechanism via elastic clusters.

Reduce I/O

A second advanced tip is to reduce your I/O consumption in DocumentDB. In DocumentDB, I/O is cost. It is cost not only literally, in the sense that you are charged directly for I/O consumption, but also in the sense that I/O reduces your performance by consuming scarce resources.

To understand how to reduce I/O consumption, let's first review some details about DocumentDB's underlying architecture. Then, we'll look at some tips for optimizing your I/O consumption.

Under the hood, DocumentDB is using a multi-version concurrency control (MVCC) architecture. This is common to many database systems and can assist with handling concurrent operations without the use of locks. A MVCC architecture may have multiple versions of a particular document that existed at different times in the database lifecycle.

To understand DocumentDB's effects on I/O, you need to know both what happens during individual write operations (inserts, updates, and deletes) as well as the garbage collection process.

A caveat up front -- this is an advanced topic that goes deep into DocumentDB architecture. As you learn about these MVCC internals, do not confuse this with whether it affects the *correctness* of your results. You'll read about multiple versions of your document and the impact on indexes, but the query engine understands how to handle these versions to return the proper result.

Every insert of a document will result in not only writing the full document to the heap but also updating every index where applicable for that document. On the other hand, deleting a document will only mark the time when the document was deleted. It won't delete the document from storage (yet!), and it won't update the indexes to remove the entry for the deleted document.

An update operation is a combination of an insert and a delete operation. It will create a new version of the document on the heap and update all indexes accordingly. Additionally, it will mark the time when the old version of the document was deleted.

You might be thinking that this leaves a lot of old data lying around, and you'd be correct. This is where garbage collection comes in. DocumentDB has automated thresholds where it will run a garbage collection process to remove any document versions that are not visible to any current queries and to clean out expired entries from indexes. This garbage collection process consumes I/O in your DocumentDB cluster.

An easy way to synthesize this knowledge is to remember that an insert consumes more I/O during the actual operation, as it has to update the indexes as well. A delete consumes less I/O during the operation but adds more I/O later during the garbage collection process. An update combines the two since it is essentially an insert plus a delete operation.

With that background in place, let's consider how to reduce our I/O consumption. The first way to do this is to reduce the number of indexes you have. An insert updates each index for that document (and a document could even have multiple entries in an index, such as with a multi-key index). Reducing the number of indexes will reduce your I/O consumption accordingly.

The best way to do this is to discover and remove both redundant and unused indexes. Many unoptimized databases have redundant indexes for handling different queries.

Going back to our compound index example, imagine you create an index that looks as follows:

```
await collection.createIndex({ 'address.zip': 1, 'birthdate': 1 });
```

This would allow you to handle multiple types of queries:

1. An exact match query on just the `address.zip` field;
2. A range query on the `address.zip` field;
3. An exact match query on the `address.zip` field plus an exact match on the `birthdate` field;
4. An exact match query on the `address.zip` field plus a range query on the `birthdate` field.

With compound indexes, recall that you don't need to filter on all values to use the index. Compound indexes are evaluated from left-to-right up to and including the first range query.

Thus, an index like the following on just `address.zip` would be redundant:

```
await collection.createIndex({ 'address.zip': 1 });
```

This query could be served by the compound index we created previously.

The DocumentDB documentation provides some advice on locating unused indexes. Some quick analysis here can save you significant I/O in your DocumentDB cluster.

A second, more difficult, tip for reducing I/O is to consider splitting up a single document into multiple documents in certain situations. This is particularly useful when you can break a document into a mutable portion and an immutable portion. When the mutable portion changes, you will have smaller write operations and correspondingly less I/O usage. Further, any immutable portions that are indexed will not be changed when updating the mutable portion.

Reduce document size

Another data modeling optimization that relies on DocumentDB internals is to reduce your document size. In most circumstances, smaller documents will result in better performance. In certain circumstances, this performance impact can be much larger.

Consider why smaller documents can be better. Smaller documents mean less I/O consumption and more documents in memory, skipping the new for I/O altogether. Further, it's less CPU consumption as you're reading over these documents.

There are a couple patterns for reducing your document size. The first, and easiest, is to compress your documents, particularly when they contain highly compressible field names or field values that are not queried or updated directly. DocumentDB does offer compression at the document level.

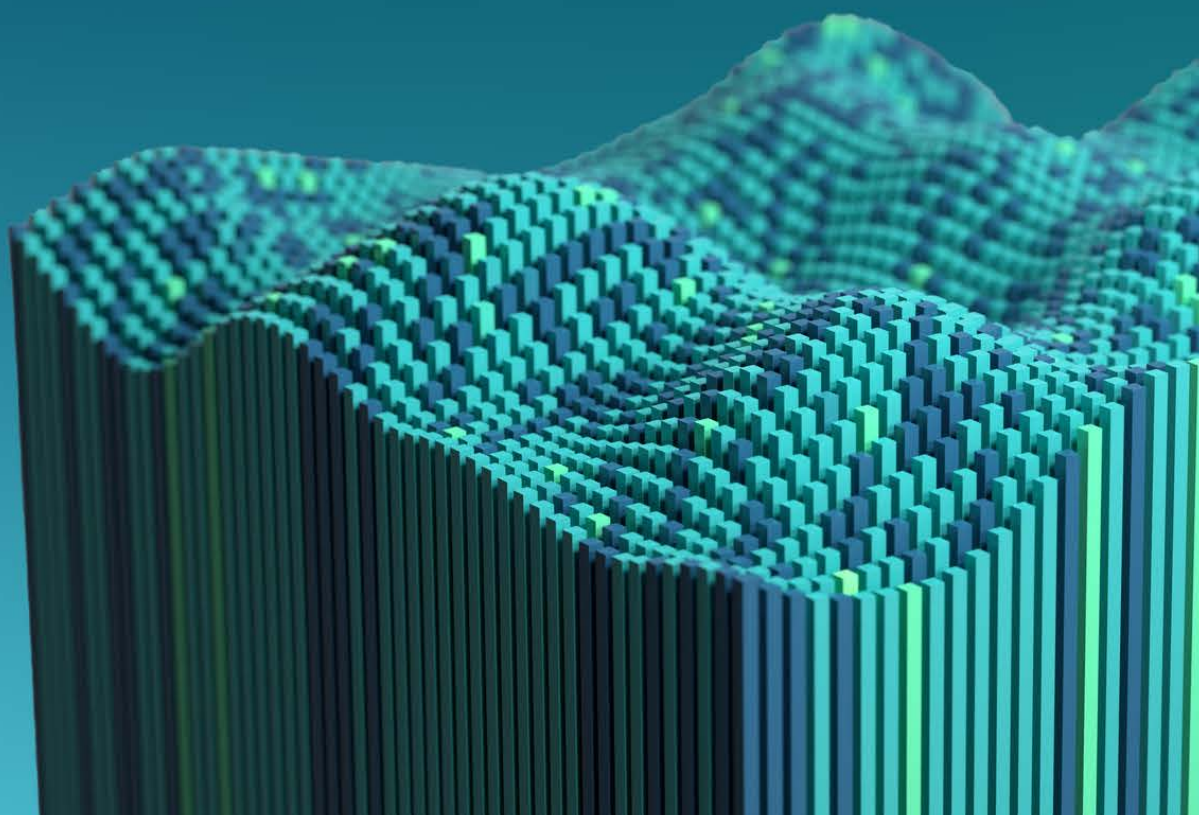
A second option is to reduce the size of your keys in DocumentDB. Remember that DocumentDB objects are self-describing, so they contain their entire schema within it. While descriptive key names like "username", "updatedAt", and "isAPayingMember" may be helpful for reading, they do expand the size of your document. Consider abbreviating these values to shorter values.

As with all of these advanced tips, consider the tips on reducing document size carefully. Make sure that the added complexity is worth the benefits on document size that will result.

Conclusion

In this ebook, we learned how to model data in DocumentDB. We started off by learning the basics of DocumentDB, including how the document model differs from a relational model. Then, we learned about the DocumentDB API and how to use it to interact with our database. Next, we saw some of the key data modeling patterns including how to manage your schema, how to handle relationships, and how to use indexes well. Finally, we looked at some advanced tips that covered proper use of the aggregation framework, scaling your database, and low-level optimizations of indexes and documents.

DocumentDB is a powerful database that can be used for a wide variety of applications. By understanding the basics of DocumentDB and how to model your data, you can build powerful applications that scale to meet your needs.



aws