12 Programming for the NoSQL database service: DynamoDB

This chapter covers

- Advantages and disadvantages of the NoSQL service, DynamoDB
- Creating tables and storing data
- Adding secondary indexes to optimize data retrieval
- Designing a data model optimized for a key-value database
- Optimizing costs by choosing the best fitting capacity mode and storage type

Most applications depend on a database to store data. Imagine an application that keeps track of a warehouse's inventory. The more inventory moves through the warehouse, the more requests the application serves and the more queries the database has to process. Sooner or later, the database becomes too busy and latency increases to a level that limits the warehouse's productivity. At this point, you have to scale the database to help the business. You can do this in two ways:

- *Vertically*—You can use faster hardware for your database machine; for example, you can add memory or replace the CPU with a more powerful model.
- *Horizontally*—You can add a second database machine. Both machines then form a *database cluster*.

Scaling a database vertically is the easier option, but it gets expensive. High-end hardware is more expensive than commodity hardware. Besides that, at some point, you will not find more powerful machines on the market anymore.

Scaling a traditional relational database horizontally is difficult because transactional guarantees such as atomicity, consistency, isolation, and durability—also known as *ACID*—require communication among all nodes of the database during a two-phase commit. To see what we're talking about, here's how a simplified two-phase commit with two nodes works, as illustrated in figure 12.1:

1. A query is sent to the database cluster that wants to change data (`INSERT`, `UPDATE`, `DELETE`).
2. The database transaction coordinator sends a commit request to the two nodes.
3. Node 1 checks whether the query could be executed. The decision is sent back to the coordinator. If the nodes decide yes, the query can be executed, it must fulfill this promise. There is no way back.
4. Node 2 checks whether the query could be executed. The decision is sent back to the coordinator.
5. The coordinator receives all decisions. If all nodes decide that the query could be executed, the coordinator instructs the nodes to finally commit.
6. Nodes 1 and 2 finally change the data. At this point, the nodes must fulfill the request. This step must not fail.

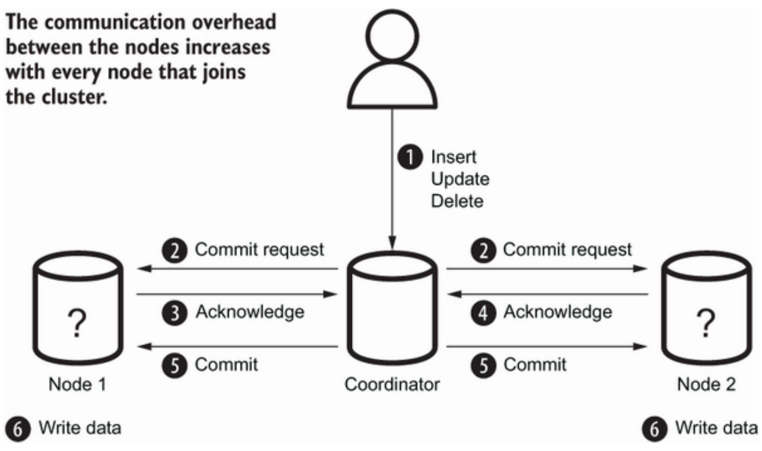


Figure 12.1 The communication overhead between the nodes increases with every node that joins the cluster.

The problem is, the more nodes you add, the slower your database becomes, because more nodes must coordinate transactions between each other. The way to tackle this has been to use databases that don't adhere to these guarantees. They're called *NoSQL databases*.

There are four types of NoSQL databases—document, graph, columnar, and key-value store—each with its own uses and applications. Amazon provides a NoSQL database service called *DynamoDB*, a key-value store. DynamoDB is a fully managed, proprietary, closed source key-value store with document support. In other words, DynamoDB persists objects identified by a unique key, which you might know from the concept of a hash table. *Fully managed* means that you only use the service and AWS operates it for you. DynamoDB is highly available and highly durable. You can scale from one item to billions and from one request per second to tens of thousands of requests per second. AWS also offers other types of NoSQL database systems like Keyspaces, Neptune, DocumentDB, and MemoryDB for Redis—more about these options later.

To use DynamoDB, your application needs to be built for this particular NoSQL database. You cannot point your legacy application to DynamoDB instead of a MySQL database, for example. Therefore, this chapter focuses on how to write an application for storing and retrieving data from DynamoDB. At the end of the chapter, we will discuss what is needed to administer DynamoDB.

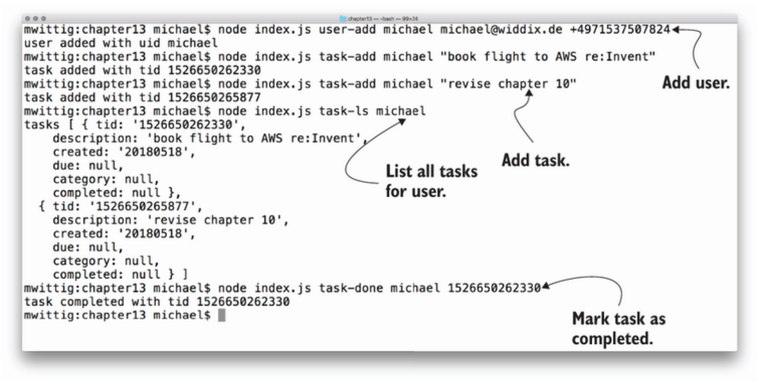
Some typical use cases for DynamoDB follow:

- When building systems that need to deal with a massive amount of requests or spiky workloads, the ability to scale horizontally is a game changer. We have used DynamoDB to track client-side errors from a web application, for example.
- When building small applications with a simple data structure, the pay-per-request pricing model and the simplicity of a fully managed service are good reasons to go with DynamoDB. For example, we used DynamoDB to track the progress of batch jobs.

While following this chapter, you will implement a simple to-do application called nodetodo, the equivalent of a "Hello World" example for databases. You will learn how to write, fetch, and query data. Figure 12.2 shows nodetodo in action.

Examples are 100% covered by the Free Tier

The examples in this chapter are totally covered by the Free Tier. Keep in mind that this applies only if there is nothing else going on in your AWS account. You’ll clean up your account at the end of the chapter.



```
mwittig:chapter13 michael$ node index.js user-add michael michael@widdix.de +4971537507824
user added with uid michael
mwittig:chapter13 michael$ node index.js task-add michael "book flight to AWS re:Invent"
task added with tid 1526650262330
mwittig:chapter13 michael$ node index.js task-add michael "revise chapter 10"
task added with tid 1526650265877
mwittig:chapter13 michael$ node index.js task-ls michael
tasks [ { tid: '1526650262330',
    description: 'book flight to AWS re:Invent',
    created: '20180518',
    due: null,
    category: null,
    completed: null },
  { tid: '1526650265877',
    description: 'revise chapter 10',
    created: '20180518',
    due: null,
    category: null,
    completed: null } ]
mwittig:chapter13 michael$ node index.js task-done michael 1526650262330
task completed with tid 1526650262330
mwittig:chapter13 michael$
```

Figure 12.2 Manage your tasks with the command-line to-do application, nodetodo.

12.1 Programming a to-do application

DynamoDB is a key-value store that organizes your data into tables. For example, you can have a table to store your users and another table to store tasks. The items contained in the table are identified by a unique key. An item could be a user or a task; think of an item as a row in a relational database.

To minimize the overhead of a programming language, you’ll use Node.js/JavaScript to create a small to-do application, nodetodo, which you can use via the terminal on your local machine. nodetodo uses DynamoDB as a database and comes with the following features:

- Creates and deletes users
- Creates and deletes tasks
- Marks tasks as done
- Gets a list of all tasks with various filters

To implement an intuitive CLI, nodetodo uses *docopt*, a command-line interface description language, to describe the CLI interface. The supported commands follow:

- `user-add`—Adds a new user to nodetodo
- `user-rm`—Removes a user
- `user-ls`—Lists users
- `user`—Shows the details of a single user
- `task-add`—Adds a new task to nodetodo
- `task-rm`—Removes a task
- `task-ls`—Lists user tasks with various filters
- `task-la`—Lists tasks by category with various filters
- `task-done`—Marks a task as finished

In the following sections, you’ll implement these commands to learn about DynamoDB hands-on. This listing shows the full CLI description of all the commands, including parameters.

Listing 12.1 CLI description language docopt: Using nodetodo (cli.txt)

```
nodetodo

Usage:
  nodetodo user-add <uid> <email> <phone>
  nodetodo user-rm <uid>
  nodetodo user-ls [--limit=<limit>] [--next=<id>]   ①
  nodetodo user <uid>
  nodetodo task-add <uid> <description> \
➥ [<category>] [--dueat=<yyyymmdd>]                ②
  nodetodo task-rm <uid> <tid>
  nodetodo task-ls <uid> [<category>] \
➥ [--overdue|--due|--withoutdue|--futuredue]        ③
  nodetodo task-la <category> \
➥ [--overdue|--due|--withoutdue|--futuredue]
  nodetodo task-done <uid> <tid>
  nodetodo -h | --help                               ④
  nodetodo --version                                 ⑤

Options:
  -h --help        Show this screen.
  --version        Show version.
```

① The named parameters limit and next are optional.

② The category parameter is optional.

③ Pipe indicates either/or.

④ help prints information about how to use nodetodo.

⑤ Version information

DynamoDB isn't comparable to a traditional relational database in which you create, read, update, or delete data with SQL. You'll use the AWS SDK to send requests to the REST API. You must integrate DynamoDB into your application; you can't take an existing application that uses an SQL database and run it on DynamoDB. To use DynamoDB, you need to write code!

12.2 Creating tables

Each DynamoDB table has a name and organizes a collection of items. An *item* is a collection of attributes, and an *attribute* is a name-value pair. The attribute value can be scalar (number, string, binary, Boolean), multivalued (number set, string set, binary set), or a JSON document (object, array). Items in a table aren't required to have the same attributes; there is no enforced schema. Figure 12.3 demonstrates these terms.

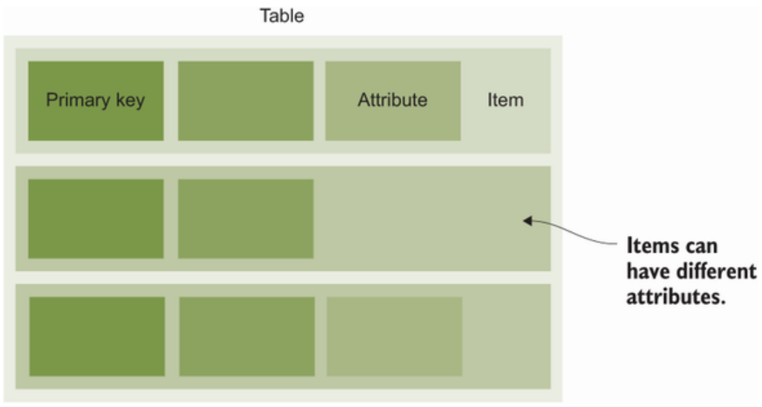


Figure 12.3 DynamoDB tables store items consisting of attributes identified by a primary key.

DynamoDB doesn't need a static schema like a relational database does, but you must define the attributes that are used as the primary key in your table. In other words, you must define the table's primary key schema. Next, you will create a table for the users of the nodetodo application as well as a table that will store all the tasks.

12.2.1 Users are identified by a partition key

To be able to assign a task to a user, the nodetodo application needs to store some information about users. Therefore, we came up with the following data structure for storing information about users:

```
{
  "uid": "emma",                 ①
  "email": "emma@widdix.de",     ②
  "phone": "0123456789"          ③
}
```

① A unique user ID

② The user’s email address

③ The phone number belonging to the user

How to create a DynamoDB table based on this information? First, you need to think about the table’s name. We suggest that you prefix all your tables with the name of your application. In this case, the table name would be `todo-user`.

Next, a DynamoDB table requires the definition of a *primary key*. A primary key consists of one or two attributes. A primary key is unique within a table and identifies an item. You need the primary key to retrieve, update, or delete an item. Note that DynamoDB does not care about any other attributes of an item because it does not require a fixed schema, as relational databases do.

For the `todo-user` table, we recommend you use the `uid` as the primary key because the attribute is unique. Also, the only command that queries data is `user`, which fetches a user by its `uid`. When using a single attribute as primary key, DynamoDB calls this the *partition key* of the table.

You can create a table by using the Management Console, CloudFormation, SDKs, or the CLI. Within this chapter, we use the AWS CLI. The `aws` `dynamodb` `create-table` command has the following four mandatory options:

- `table-name` —Name of the table (can’t be changed).
- `attribute-definitions` —Name and type of attributes used as the primary key. Multiple definitions can be given using the syntax `AttributeName=attr1`, `AttributeType=S`, separated by a space character. Valid types are `S` (string), `N` (number), and `B` (binary).
- `key-schema` —Name of attributes that are part of the primary key (can’t be changed). Contains a single entry using the syntax `AttributeName=attr1, KeyType=HASH` for a partition key, or two entries separated by spaces for a partition key and sort key. Valid types are `HASH` and `RANGE`.
- `provisioned-throughput` —Performance settings for this table defined as `ReadCapacityUnits=5,WriteCapacityUnits=5` (you’ll learn about this in section 12.11).

Execute the following command to create the `todo-user` table with the `uid` attribute as the partition key:

```
$ aws dynamodb create-table --table-name todo-user \                ①
➥ --attribute-definitions AttributeName=uid,AttributeType=S \      ②
```

```
  --key-schema AttributeName=uid,KeyType=HASH \                          ③
  --provisioned-throughput ReadCapacityUnits=5,WriteCapacityUnits=5      ④
```

① Prefixing tables with the name of your application will prevent name clashes in the future.

② Items must at least have one attribute uid of type string.

③ The partition key (type HASH) uses the uid attribute.

④ You'll learn about this in section 12.11.

Creating a table takes some time. Wait until the status changes to `ACTIVE`. You can check the status of a table as follows.

Listing 12.2 Checking the status of the DynamoDB table

```
$ aws dynamodb describe-table --table-name todo-user    ①
{
  "Table": {
    "AttributeDefinitions": [                            ②
      {
        "AttributeName": "uid",
        "AttributeType": "S"
      }
    ],
    "TableName": "todo-user",
    "KeySchema": [                                       ③
      {
        "AttributeName": "uid",
        "KeyType": "HASH"
      }
    ],
    "TableStatus": "ACTIVE",                             ④
    "CreationDateTime": "2022-01-24T16:00:29.105000+01:00",
    "ProvisionedThroughput": {
      "NumberOfDecreasesToday": 0,
      "ReadCapacityUnits": 5,
      "WriteCapacityUnits": 5
    },
    "TableSizeBytes": 0,
    "ItemCount": 0,
    "TableArn": "arn:aws:dynamodb:us-east-1:111111111111:table/todo-user",
    "TableId": "0697ea25-5901-421c-af29-8288a024392a"
  }
}
```

① The CLI command to check the table status

② Attributes defined for that table

③ Attributes used as the primary key

④ Status of the table

12.2.2 Tasks are identified by a partition key and sort key

So far, we created the table `todo-user` to store information about the users of nodetodo. Next, we need a table to store the tasks. A task belongs to a user and contains a description of the task. Therefore, we came up with the following data structure:

```
{
  "uid": "emma",                        ①
```

```
    "tid": 1645609847712,          ②
    "description": "prepare lunch"    ③
  }
```

① The task is assigned to the user with this ID.

② The creation time (number of milliseconds elapsed since January 1, 1970 00:00:00 UTC) is used as ID for the task.

③ The description of the task

How does nodetodo query the task table? By using the `task-ls` command, which we need to implement. This command lists all the tasks belonging to a user. Therefore, choosing the `tid` as the primary key is not sufficient. We recommend a combination of `uid` and `tid` instead. DynamoDB calls the components of a primary key with two attributes partition key and *sort key*. Neither the partition key nor the sort key need to be unique, but the combination of both parts must be unique.

NOTE This solution has one limitation: users can add only one task per timestamp. Because tasks are uniquely identified by `uid` and `tid` (the primary key) there can't be two tasks for the same user at the same time. Our timestamp comes with millisecond resolution, so it should be fine.

Using a partition key and a sort key uses two of your table's attributes. For the partition key, an unordered hash index is maintained; the sort key is kept in a sorted index for each partition key. The combination of the partition key and the sort key uniquely identifies an item if they are used as the primary key. The following data set shows the combination of unsorted partition keys and sorted sort keys:

```
  ["john", 1] => {             ①
    "uid": "john",
    "tid": 1,
    "description": "prepare customer presentation"
  }
  ["john", 2] => {             ②
    "uid": "john",
    "tid": 2,
    "description": "plan holidays"
  }
  ["emma", 1] => {             ③
    "uid": "emma",
    "tid": 1,
    "description": "prepare lunch"
  }
  ["emma", 2] => {
    "uid": "emma",
    "tid": 2,
    "description": "buy nice flowers for mum"
  }
  ["emma", 3] => {
    "uid": "emma",
    "tid": 3,
    "description": "prepare talk for conference"
  }
```

① The primary key consists of the uid (john) used as the partition key and tid (1) used as the sort key.

② The sort keys are sorted within a partition key.

③ There is no order in the partition keys.

Execute the following command to create the `todo-task` table with a primary key consisting of the partition key `uid` and sort key `tid` like this:

```
$ aws dynamodb create-table --table-name todo-task \
- --attribute-definitions AttributeName=uid,AttributeType=S \    ①
- AttributeName=tid,AttributeType=N \
- --key-schema AttributeName=uid,KeyType=HASH \
- AttributeName=tid,KeyType=RANGE \                               ②
- --provisioned-throughput ReadCapacityUnits=5,WriteCapacityUnits=5
```

① At least two attributes are needed for a partition key and sort key.

② The tid attribute is the sort key.

Wait until the table status changes to `ACTIVE` when you run `aws dynamodb describe-table --table-name todo-task`.

The `todo-user` and `todo-task` tables are ready. Time to add some data.

12.3 Adding data

Now you have two tables up and running to store users and their tasks. To use them, you need to add data. You'll access DynamoDB via the Node.js SDK, so it's time to set up the SDK and some boilerplate code before you implement adding users and tasks.

Installing and getting started with Node.js

Node.js is a platform for executing JavaScript in an event-driven environment so you can easily build network applications. To install Node.js, visit **https://nodejs.org** and download the package that fits your OS. All examples in this book are tested with Node.js 14.

After Node.js is installed, you can verify that everything works by typing `node --version` into your terminal. Your terminal should respond with something similar to `v14.*`. Now you're ready to run JavaScript examples like nodetodo for AWS.

Do you want to learn more about Node.js? We recommend *Node.js in Action* (second edition) by Alex Young, et al. (Manning, 2017), or the video course *Node.js in Motion* by PJ Evans (Manning, 2018).

To get started with Node.js and docopt, you need some magic lines to load all the dependencies and do some configuration work. Listing 12.3 shows how this can be done.

Where is the code located?

As usual, you'll find the code in the book's code repository on GitHub: **https://github.com/AWSinAction/code3**. nodetodo is located in /chapter12/. Switch to that directory, and run `npm install` in your terminal to install all needed dependencies.

Docopt is responsible for reading all the arguments passed to the process. It returns a JavaScript object, where the arguments are mapped to the parameters in the CLI description.

Listing 12.3 nodetodo: Using docopt in Node.js (index.js)

```
const fs = require('fs');                ①
const docopt = require('docopt');        ②
const moment = require('moment');        ③
const AWS = require('aws-sdk');          ④
const db = new AWS.DynamoDB({
  region: 'us-east-1'
});

const cli = fs.readFileSync('./cli.txt',
➥ {encoding: 'utf8'});                 ⑤
const input = docopt.docopt(cli, {       ⑥
  version: '1.0',
  argv: process.argv.splice(2)
});
```

① Loads the fs module to access the filesystem

② Loads the docopt module to read input arguments

③ Loads the moment module to simplify temporal types in JavaScript

④ Loads the AWS SDK module

⑤ Reads the CLI description from the file cli.txt

⑥ Parses the arguments, and saves them to an input variable

Next, you will add an item to a table. The next listing explains the `putItem` method of the AWS SDK for Node.js.

Listing 12.4 DynamoDB: Creating an item

```
const params = {
  Item: {                            ①
    attr1: {S: 'val1'},              ②
    attr2: {N: '2'}                  ③
  },
  TableName: 'app-entity'            ④
};
db.putItem(params, (err) => {        ⑤
  if (err) {                         ⑥
    console.error('error', err);
  } else {
    console.log('success');
  }
});
```

① All item attribute name-value pairs

② Strings are indicated by an S.

③ Numbers (floats and integers) are indicated by an N.

④ Adds item to the app-entity table

⑤ Invokes the putItem operation on DynamoDB

⑥ Handles errors

The first step is to add an item to the `todo-user` table.

12.3.1 Adding a user

The following listing shows the code executed by the `user-add` command.

Listing 12.5 nodetodo: Adding a user (index.js)

```
if (input['user-add'] === true) {
  const params = {
    Item: {                                                    ①
      uid: {S: input['<uid>']},                                ②
      email: {S: input['<email>']},                            ③
      phone: {S: input['<phone>']}                             ④
    },
    TableName: 'todo-user',                                    ⑤
    ConditionExpression: 'attribute_not_exists(uid)'          ⑥
  };
  db.putItem(params, (err) => {                                ⑦
    if (err) {
      console.error('error', err);
    } else {
      console.log('user added');
    }
  });
}
```

① Item contains all attributes. Keys are also attributes, and that’s why you do not need to tell DynamoDB which attributes are keys if you add data.

② The uid attribute is of type string and contains the uid parameter value.

③ The email attribute is of type string and contains the email parameter value.

④ The phone attribute is of type string and contains the phone parameter value.

⑤ Specifies the user table

⑥ If putItem is called twice on the same key, data is replaced. ConditionExpression allows the putItem only if the key isn’t yet present.

⑦ Invokes the putItem operation on DynamoDB

When you make a call to the AWS API, you always do the following:

1. Create a JavaScript object (map) filled with the needed parameters (the params variable).
2. Invoke the function via the AWS SDK.
3. Check whether the response contains an error, and if not, process the returned data.

Therefore, you only need to change the content of `params` if you want to add a task instead of a user. Execute the following commands to add two users:

```
node index.js user-add john john@widdix.de +11111111
node index.js user-add emma emma@widdix.de +22222222
```

Time to add tasks.

12.3.2 Adding a task

John and Emma want to add tasks to their to-do lists to better organize their everyday life. Adding a task is similar to adding a user. The next listing shows the code used to create a task.

Listing 12.6 nodetodo: Adding a task (index.js)

```
if (input['task-add'] === true) {
  const tid = Date.now();                                          ①
  const params = {
    Item: {
      uid: {S: input['<uid>']},
      tid: {N: tid.toString()},                                    ②
      description: {S: input['<description>']},
      created: {N: moment(tid).format('YYYYMMDD')}                 ③
    },
    TableName: 'todo-task',                                        ④
    ConditionExpression: 'attribute_not_exists(uid)
➥ and attribute_not_exists(tid)'                                  ⑤
  };
  if (input['--dueat'] !== null) {                                 ⑥
    params.Item.due = {N: input['--dueat']};
  }
  if (input['<category>'] !== null) {                              ⑦
    params.Item.category = {S: input['<category>']};
  }
  db.putItem(params, (err) => {                                    ⑧
    if (err) {
      console.error('error', err);
    } else {
      console.log('task added with tid ' + tid);
    }
  });
}
```

① Creates the task ID (tid) based on the current timestamp

② The tid attribute is of type number and contains the tid value.

③ The created attribute is of type number (format 20150525).

④ Specifies the task table

⑤ Ensures that an existing item is not overridden

⑥ If the optional named parameter dueat is set, adds this value to the item

⑦ If the optional named parameter category is set, adds this value to the item

⑧ Invokes the putItem operation on DynamoDB

Now we'll add some tasks. Use the following command to remind Emma about buying some milk and a task asking John to put out the garbage:

```
node index.js task-add emma "buy milk" "shopping"
node index.js task-add emma "put out the garbage" "housekeeping" --dueat "20220224"
```

Now that you are able to add users and tasks to nodetodo, wouldn't it be nice if you could retrieve all this data?

12.4 Retrieving data

So far, you have learned how to insert users and tasks into two different DynamoDB tables. Next, you will learn how to query the data, for example, to get a list of all tasks assigned to Emma.

DynamoDB is a key-value store. The key is usually the only way to retrieve data from such a store. When designing a data model for DynamoDB, you must be aware of that limitation when you create tables (as you did in section 12.2). If you can use only the key to look up data, you'll sooner or later experience difficulties. Luckily, DynamoDB provides two other ways to look up items: a secondary index lookup and the scan operation. You'll start by retrieving data by the items' primary key and continue with more sophisticated methods of data retrieval.

DynamoDB Streams

DynamoDB lets you retrieve changes to a table as soon as they're made. A Stream provides all write (create, update, delete) operations to your table items. The order is consistent within a partition key. DynamoDB Streams also help with the following:

- If your application polls the database for changes, DynamoDB Streams solves the problem in a more elegant way.
- If you want to populate a cache with the changes made to a table, DynamoDB Streams can help.

12.4.1 Getting an item by key

Let's start with a simple example. You want to find out the contact details about Emma, which are stored in the `todo-user` table.

The simplest form of data retrieval is looking up a single item by its primary key, for example, a user by its ID. The `getItem` SDK operation to get a single item from DynamoDB can be used like this.

Listing 12.7 DynamoDB: Query a single item (index.js)

```
const params = {
  Key: {
    attr1: {S: 'val1'}                  ①
  },
  TableName: 'app-entity'
};
db.getItem(params, (err, data) => {     ②
  if (err) {
    console.error('error', err);
  } else {
    if (data.Item) {                    ③
      console.log('item', data.Item);
    } else {
      console.error('no item found');
    }
  }
});
```

① Specifies the attributes of the primary key

② Invokes the getItem operation on DynamoDB

③ Checks whether an item was found

The `user` `<uid>` command retrieves a user by the user's ID. The code to fetch a user is shown in listing 12.8.

Listing 12.8 nodetodo: Retrieving a user (index.js)

```
const mapUserItem = (item) => {                            ①
  return {
    uid: item.uid.S,
    email: item.email.S,
    phone: item.phone.S
  };
};

if (input['user'] === true) {
  const params = {
    Key: {
      uid: {S: input['<uid>']}                             ②
    },
    TableName: 'todo-user'                                 ③
  };
  db.getItem(params, (err, data) => {                      ④
    if (err) {
      console.error('error', err);
    } else {
      if (data.Item) {                                     ⑤
        console.log('user', mapUserItem(data.Item));
      } else {
        console.error('user not found');
      }
    }
  });
}
```

① Helper function to transform DynamoDB result

② Looks up a user by primary key

③ Specifies the user table

④ Invokes the getItem operation on DynamoDB

⑤ Checks whether data was found for the primary key

For example, use the following command to fetch information about Emma:

```
node index.js user emma
```

You can also use the `getItem` operation to retrieve data by partition key and sort key, for example, to look up a specific task. The only change is that the `Key` has two entries instead of one. `getItem` returns one item or no items; if you want to get multiple items, you need to query DynamoDB.

12.4.2 Querying items by key and filter

Emma wants to get check her to-do list for things she needs to do. Therefore, you will query the `todo-task` table to retrieve all tasks assigned to Emma next.

If you want to retrieve a collection of items rather than a single item, such as all tasks for a user, you must query DynamoDB. Retrieving multiple items by primary key works only if your table has a partition key and sort key. Otherwise, the partition key will identify only a single item. Listing 12.9 shows how the `query` SDK operation can be used to get a collection of items from DynamoDB.

Listing 12.9 DynamoDB: Querying a table

```
const params = {
  KeyConditionExpression: 'attr1 = :attr1val
➥ AND attr2 = :attr2val',                         ①
  ExpressionAttributeValues: {
    ':attr1val': {S: 'val1'},                       ②
    ':attr2val': {N: '2'}                           ③
  },
  TableName: 'app-entity'
};
db.query(params, (err, data) => {                   ④
  if (err) {
    console.error('error', err);
  } else {
    console.log('items', data.Items);
  }
});
```

① The condition the key must match. Use AND if you're querying both a partition and sort key. Only the = operator is allowed for partition keys. Allowed operators for sort keys are =, >, <, >=, ⇐, BETWEEN x AND y, and begins_with. Sort key operators are blazing fast because the data is already sorted.

② Dynamic values are referenced in the expression.

③ Always specify the correct type (S, N, B).

④ Invokes the query operation on DynamoDB

The `query` operation also lets you specify an optional `FilterExpression`, to include only items that match the filter and key condition. This is helpful to reduce the result set, for example, to show only tasks of a specific category. The syntax of `FilterExpression` works like `KeyConditionExpression`, but no index is used for filters. Filters are applied to all matches that `KeyConditionExpression` returns.

To list all tasks for a certain user, you must query DynamoDB. The primary key of a task is the combination of the `uid` and the `tid`. To get all tasks for a user, `KeyConditionExpression` requires only the partition key.

The next listing shows two helper functions used to implement the `task-ls` command.

Listing 12.10 nodetodo: Retrieving tasks (index.js)

```
const getValue = (attribute, type) => {     ①
  if (attribute === undefined) {
    return null;
  }
  return attribute[type];
};

const mapTaskItem = (item) => {              ②
  return {
    tid: item.tid.N,
    description: item.description.S,
    created: item.created.N,
    due: getValue(item.due, 'N'),
    category: getValue(item.category, 'S'),
    completed: getValue(item.completed, 'N')
  };
};
```

① Helper function to access optional attributes

② Helper function to transform the DynamoDB result

The real magic happens in listing 12.11, which shows the implementation of the `task-ls` command.

Listing 12.11 nodetodo: Retrieving tasks (index.js)

```
if (input['task-ls'] === true) {
  const yyyymmdd = moment().format('YYYYMMDD');
  const params = {
    KeyConditionExpression: 'uid = :uid',                    ①
    ExpressionAttributeValues: {
      ':uid': {S: input['<uid>']}                            ②
    },
    TableName: 'todo-task',
    Limit: input['--limit']
  };
  if (input['--next'] !== null) {
    params.KeyConditionExpression += ' AND tid > :next';
    params.ExpressionAttributeValues[':next'] = {N: input['--next']};
  }
  if (input['--overdue'] === true) {
    params.FilterExpression = 'due < :yyyymmdd';             ③
    params.ExpressionAttributeValues
➥ [':yyyymmdd'] = {N: yyyymmdd};                            ④
  } else if (input['--due'] === true) {
    params.FilterExpression = 'due = :yyyymmdd';
    params.ExpressionAttributeValues[':yyyymmdd'] = {N: yyyymmdd};
  } else if (input['--withoutdue'] === true) {
    params.FilterExpression =
➥ 'attribute_not_exists(due)';                              ⑤
  } else if (input['--futuredue'] === true) {
    params.FilterExpression = 'due > :yyyymmdd';
    params.ExpressionAttributeValues[':yyyymmdd'] = {N: yyyymmdd};
  } else if (input['--dueafter'] !== null) {
    params.FilterExpression = 'due > :yyyymmdd';
    params.ExpressionAttributeValues[':yyyymmdd'] =
      {N: input['--dueafter']};
  } else if (input['--duebefore'] !== null) {
    params.FilterExpression = 'due < :yyyymmdd';
    params.ExpressionAttributeValues[':yyyymmdd'] =
      {N: input['--duebefore']};
  }
  if (input['<category>'] !== null) {
    if (params.FilterExpression === undefined) {
      params.FilterExpression = '';
    } else {
      params.FilterExpression += ' AND ';                    ⑥
    }
    params.FilterExpression += 'category = :category';
    params.ExpressionAttributeValues[':category'] =
      S: input['<category>']};
  }
  db.query(params, (err, data) => {                          ⑦
    if (err) {
      console.error('error', err);
    } else {
      console.log('tasks', data.Items.map(mapTaskItem));
      if (data.LastEvaluatedKey !== undefined) {
        console.log('more tasks available with --next=' +
          data.LastEvaluatedKey.tid.N);
      }
    }
  });
}
```

① Primary key query. The task table uses a partition and sort key. Only the partition key is defined in the query, so all tasks belonging to a user are returned.

② Query attributes must be passed this way.

③ Filtering uses no index; it's applied over all elements returned from the primary key query.

④ Filter attributes must be passed this way.

⑤ attribute_not_exists(due) is true when the attribute is missing (opposite of attribute_exists).

⑥ Multiple filters can be combined with logical operators.

⑦ Invokes the query operation on DynamoDB

As an example, you could use the following command to get a list of Emma's shopping tasks. The query will fetch all tasks by `uid` first and filter items based on the category attribute next:.

```
node index.js task-ls emma shopping
```

Two problems arise with the query approach:

- Depending on the result size from the primary key query, filtering may be slow. Filters work without an index: every item must be inspected. Imagine you have stock prices in DynamoDB, with a partition key and sort key: the partition key is a ticker like AAPL, and the sort key is a timestamp. You can make a query to retrieve all stock prices of Apple (AAPL) between two timestamps (20100101 and 20150101). But if you want to return prices only on Mondays, you need to filter over all prices to return only 20% (one out of five trading days each week) of them. That's wasting a lot of resources!
- You can only query the primary key. Returning a list of all tasks that belong to a certain category for all users isn't possible, because you can't query the `category` attribute.

You can solve these problems with secondary indexes. Let's look at how they work.

12.4.3 Using global secondary indexes for more flexible queries

In this section, you will create a secondary index that allows you to query the tasks belonging to a certain category.

A *global secondary index* is a projection of your original table that is automatically maintained by DynamoDB. Items in an index don't have a primary key, just a key. This key is not necessarily unique within the index. Imagine a table of users where each user has a country attribute. You then create a global secondary index where the country is the new partition key. As you can see, many users can live in the same country, so that key is not unique in the index.

You can query a global secondary index like you would query the table. You can imagine a global secondary index as a read-only DynamoDB table that is automatically maintained by DynamoDB: whenever you change the parent table, all indexes are asynchronously (eventually consistent!) updated as well.

In our example, we will create a global secondary index for the table `todo-tasks` which uses the `category` as the partition key and the `tid` as the sort key. Doing so allows us to query tasks by category. Figure 12.4 shows how a global secondary index works.

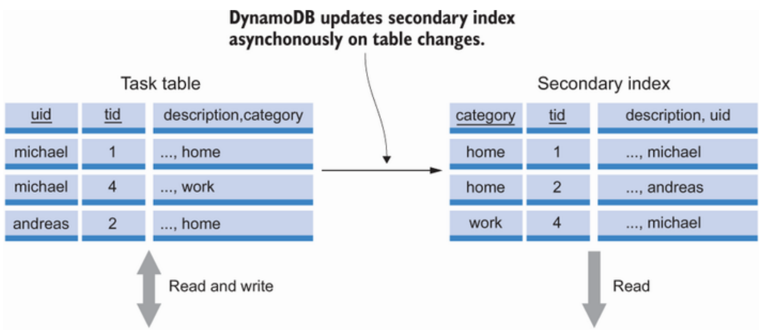


Figure 12.4 A global secondary index contains a copy (projection) of your table's data to provide fast lookup on another key.

A global secondary index comes at a price: the index requires storage (the same cost as for the original table). You must provision additional write-capacity units for the index as well, because a write to your table will cause a write to the global secondary index as well.

Local secondary index

Besides global secondary indexes, DynamoDB also supports local secondary indexes. A local secondary index must use the same partition key as the table. You can only vary on the attribute that is used as the sort key. A local secondary index consumes the read and write capacity of the table.

A huge benefit of DynamoDB is that you can provision capacity based on your workload. If one of your global secondary indexes gets tons of read traffic, you can increase the read capacity of that index. You can fine-tune your database performance by provisioning sufficient capacity for your tables and indexes. You'll learn more about that in section 13.9.

12.4.4 Creating and querying a global secondary index

Back to nodetodo. John needs the shopping list including Emma's and his tasks for his trip to town.

To implement the retrieval of tasks by category, you'll add a secondary index to the `todo-task` table. This will allow you to make queries by category. A partition key and sort key are used: the partition key is the `category` attribute, and the sort key is the `tid` attribute. The index also needs a name: `category-index`. You can find the following CLI command in the README.md file in nodetodo's code folder and simply copy and paste:

```
$ aws dynamodb update-table --table-name todo-task \     ①
➥ --attribute-definitions AttributeName=uid,AttributeType=S \
➥ AttributeName=tid,AttributeType=N \
➥ AttributeName=category,AttributeType=S \                  ②
➥ --global-secondary-index-updates '[{\
➥ "Create": {\                                              ③
➥ "IndexName": "category-index", \
➥ "KeySchema": [{"AttributeName": "category",
➥ "KeyType": "HASH"}, \                                     ④
➥ {"AttributeName": "tid", "KeyType": "RANGE"}], \
➥ "Projection": {"ProjectionType": "ALL"}, \                ⑤
➥ "ProvisionedThroughput": {"ReadCapacityUnits": 5, \
➥ "WriteCapacityUnits": 5}\
➥ }}]'
```

① Adds a global secondary index by updating the table that has already been created previously

② Adds a category attribute, because the attribute will be used in the index

③ Creates a new secondary index

④ The category attribute is the partition key, and the tid attribute is the sort key.

⑤ All attributes are projected into the index.

Creating a global secondary index takes about five minutes. You can use the CLI to find out if the index is ready like this:

```
$ aws dynamodb describe-table --table-name=todo-task \
➥ --query "Table.GlobalSecondaryIndexes"
```

The next listing shows the code to query the global secondary index to fetch tasks by category with the help of the `task-la` command.

Listing 12.12 nodetodo: Retrieving tasks from a global secondary index (index.js)

```
if (input['task-la'] === true) {
  const yyyymmdd = moment().format('YYYYMMDD');
  const params = {
    KeyConditionExpression: 'category = :category',        ①
    ExpressionAttributeValues: {
      ':category': {S: input['<category>']}
    },
    TableName: 'todo-task',
    IndexName: 'category-index',                           ②
    Limit: input['--limit']
  };
  if (input['--next'] !== null) {
    params.KeyConditionExpression += ' AND tid > :next';
    params.ExpressionAttributeValues[':next'] = {N: input['--next']};
  }
  if (input['--overdue'] === true) {
    params.FilterExpression = 'due < :yyyymmdd';           ③
    params.ExpressionAttributeValues[':yyyymmdd'] = {N: yyyymmdd};
  }
  [...]
  db.query(params, (err, data) => {
    if (err) {
      console.error('error', err);
    } else {
      console.log('tasks', data.Items.map(mapTaskItem));
      if (data.LastEvaluatedKey !== undefined) {
        console.log('more tasks available with --next='
          + data.LastEvaluatedKey.tid.N);
      }
    }
  });
}
```

① A query against an index works the same as a query against a table ...

② ... but you must specify the index you want to use.

③ Filtering works the same as with tables.

When following our example, use the following command to fetch John and Emma’s shopping tasks:

```
node index.js task-la shopping
```

But you’ll still have situations where a query doesn’t work. For example, you can’t retrieve all users from `todo-user` with a query. Therefore, let’s look at how to scan through all items of a table.

12.4.5 Scanning and filtering all of your table’s data

John wants to know who else is using nodetodo and asks for a list of all users. You will learn how to list all items of a table without using an index next.

Sometime you can’t work with keys because you don’t know them up front; instead, you need to go through all the items in a table. That’s not very efficient, but in rare situations, like daily batch jobs or rare requests, it’s fine. DynamoDB provides the `scan` operation to scan all items in a table as shown next.

Listing 12.13 DynamoDB: Scan through all items in a table

```
const params = {
  TableName: 'app-entity',
  Limit: 50                                                    ①
};
db.scan(params, (err, data) => {                               ②
  if (err) {
    console.error('error', err);
  } else {
    console.log('items', data.Items);
    if (data.LastEvaluatedKey !== undefined) {                 ③
      console.log('more items available');
    }
  }
});
```

① Specifies the maximum number of items to return

② Invokes the scan operation on DynamoDB

③ Checks whether there are more items that can be scanned

The following listing shows how to scan through all items of the `todo-user` table. A paging mechanism is used to prevent too many items from being returned at a time.

Listing 12.14 nodetodo: Retrieving all users with paging (index.js)

```
if (input['user-ls'] === true) {
  const params = {
    TableName: 'todo-user',
    Limit: input['--limit']                                    ①
  };
  if (input['--next'] !== null) {
    params.ExclusiveStartKey = {                               ②
      uid: {S: input['--next']}
    };
  }
  db.scan(params, (err, data) => {                             ③
    if (err) {
      console.error('error', err);
    } else {
      console.log('users', data.Items.map(mapUserItem));
      if (data.LastEvaluatedKey !== undefined) {               ④
        console.log('page with --next=' + data.LastEvaluatedKey.uid.S);
      }
    }
  });
}
```

① The maximum number of items returned

② The named parameter next contains the last evaluated key.

③ Invokes the scan operation on DynamoDB

④ Checks whether the last item has been reached

Use the following command to fetch all users:

```
node index.js user-ls
```

The `scan` operation reads all items in the table. This example didn't filter any data, but you can use `FilterExpression` as well. Note that you shouldn't use the `scan` operation too often—it's flexible but not efficient.

12.4.6 Eventually consistent data retrieval

By default, reading data from DynamoDB is *eventually consistent*. That means it's possible that if you create an item (version 1), update that item (version 2), and then read that item, you may get back version 1; if you wait and fetch the item again, you'll see version 2. Figure 12.5 shows this process. This behavior occurs because the item is persisted on multiple machines in the background. Depending on which machine answers your request, the machine may not have the latest version of the item.

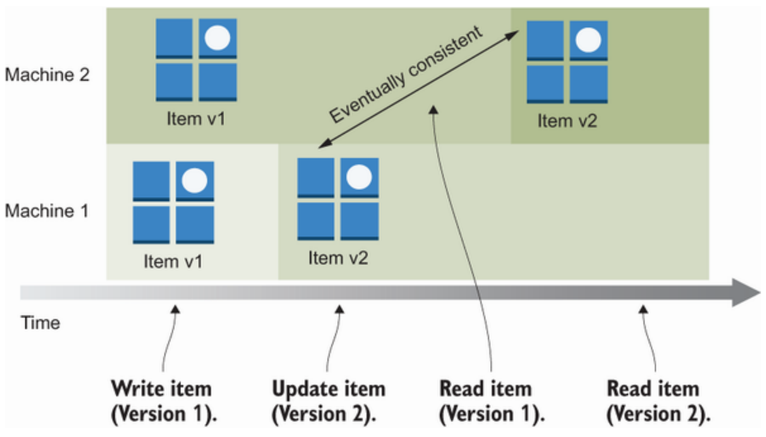


Figure 12.5 Eventually consistent reads can return old values after a write operation until the change is propagated to all machines.

You can prevent eventually consistent reads by adding `"Consistent-Read": true` to the DynamoDB request to get *strongly consistent reads*. Strongly consistent reads are supported by `getItem`, `query`, and `scan` operations. But a strongly consistent read is more expensive—it takes longer and consumes more read capacity—than an eventually consistent read. Reads from a global secondary index are always eventually consistent because the synchronization between table and index happens asynchronously.

Typically, NoSQL databases do not support transactions with atomicity, consistency, isolation, and durability (ACID) guarantees. However, DynamoDB comes with the ability to bundle multiple read or write requests into a transaction. The relevant API methods are called `TransactWriteItems` and `TransactGetItems`—you can either group write or read requests, but not a mix of both of them. Whenever possible, you should avoid using transactions because they are more expensive from a cost and latency perspective. Check out **http://mng.bz/E0ol** if you are interested in the details about DynamoDB transactions.

12.5 Removing data

John stumbled across a fancy to-do application on the web. Therefore, he decides to remove his user from nodetodo.

Like the `getItem` operation, the `deleteItem` operation requires that you specify the primary key you want to delete. Depending on whether your table uses a partition key or a partition key and sort key, you must specify one or two attributes. The next listing shows the implementation of the `user-rm` command.

Listing 12.15 nodetodo: Removing a user (index.js)

```
if (input['user-rm'] === true) {
  const params = {
    Key: {
      uid: {S: input['<uid>']}       ①
    },
    TableName: 'todo-user'           ②
  };
  db.deleteItem(params, (err) => {   ③
    if (err) {
      console.error('error', err);
    } else {
      console.log('user removed');
    }
  });
}
```

① Identifies an item by partition key

② Specifies the user table

③ Invokes the deleteItem operation on DynamoDB

Use the following command to delete John from the DynamoDB table:

```
node index.js user-rm john
```

But John wants to delete not only his user but also his tasks. Luckily, removing a task works similarly. The only change is that the item is identified by a partition key and sort key and the table name has to be changed. The code used for the `task-rm` command is shown here.

Listing 12.16 nodetodo: Removing a task (index.js)

```
if (input['task-rm'] === true) {
  const params = {
    Key: {
      uid: {S: input['<uid>']},
      tid: {N: input['<tid>']}     ①
    },
    TableName: 'todo-task'         ②
  };
  db.deleteItem(params, (err) => {
    if (err) {
      console.error('error', err);
    } else {
      console.log('task removed');
    }
  });
}
```

① Identifies an item by partition key and sort key

② Specifies the task table

You’re now able to create, read, and delete items in DynamoDB. The only operation you’re missing is updating an item.

12.6 Modifying data

Emma is still a big fan of nodetodo and uses the application constantly. She just bought some milk and wants to mark the task as done.

You can update an item with the `updateItem` operation. You must identify the item you want to update by its primary key; you can also provide an `UpdateExpression` to specify the updates you want to perform. Use one or a combination of the following update actions:

- Use `SET` to override or create a new attribute. Examples: `SET attr1 = :attr1val`, `SET attr1 = attr2 + :attr2val`, `SET attr1 = :attr1val, attr2 = :attr2val`.
- Use `REMOVE` to remove an attribute. Examples: `REMOVE attr1`, `REMOVE attr1, attr2`.

To implement this feature, you need to update the task item, as shown next.

Listing 12.17 nodetodo: Updating a task as done (index.js)

```
if (input['task-done'] === true) {
  const yyyymmdd = moment().format('YYYYMMDD');
  const params = {
    Key: {
      uid: {S: input['<uid>']},                        ①
      tid: {N: input['<tid>']}
    },
    UpdateExpression: 'SET completed = :yyyymmdd',    ②
    ExpressionAttributeValues: {
      ':yyyymmdd': {N: yyyymmdd}                       ③
    },
    TableName: 'todo-task'
  };
```

```
    db.updateItem(params, (err) => {                    ④
      if (err) {
        console.error('error', err);
      } else {
        console.log('task completed');
      }
    });
  }
```

① Identifies the item by a partition and sort key

② Defines which attributes should be updated

③ Attribute values must be passed this way.

④ Invokes the updateItem operation on DynamoDB

For example, the following command closes Emma’s task to buy milk. Please note: the `<tid>` will differ when following the example yourself. Use `node` `index.js` `task-ls` `emma` to get the task’s ID:

```
node index.js task-done emma 1643037541999
```

12.7 Recap primary key

We’d like to recap an important aspect of DynamoDB, the primary key. A primary key is unique within a table and identifies an item. You can use a single attribute as the primary key. DynamoDB calls this a partition key. You need an item’s partition key to look up that item. Also, when updating or deleting an item, DynamoDB requires the partition key. You can also use two attributes as the primary key. In this case, one of the attributes is the partition key, and the other is called the sort key.

12.7.1 Partition key

A partition key uses a single attribute of an item to create a hash-based index. If you want to look up an item based on its partition key, you need to know the exact partition key. For example, a user table could use the user’s email address as a partition key. The user could then be retrieved if you know the partition key—the email address, in this case.

12.7.2 Partition key and sort key

When you use both a partition key and a sort key, you’re using two attributes of an item to create a more powerful index. To look up an item, you need to know its exact partition key, but you don’t need to know the sort key. You can even have multiple items with the same partition key: they will be sorted according to their sort key.

The partition key can be queried only using exact matches (=). The sort key can be queried using `=`, `>`, `<`, `>=`, `<=`, and `BETWEEN` `x` `AND` `y` operators. For example, you can query the sort key of a partition key from a certain starting point. You cannot query only the sort key—you must always specify the partition key. A message table could use a partition key and sort key as its primary key; the partition key could be the user’s email, and the sort key could be a timestamp. You could then look up all of user’s messages that are newer or older than a specific timestamp, and the items would be sorted according to the timestamp.

12.8 SQL-like queries with PartiQL

As the developer of nodetodo, we want to get some insight into the way our users use the application. Therefore, we are looking for a flexible approach to query the data stored on DynamoDB on the fly.

Because SQL is such a widely used language, hardly any database system can avoid offering an SQL interface as well, even if, in the case of NoSQL databases, often only a fraction of the language is supported. In the following examples, you will learn how to use PartiQL, which is designed to provide unified query access to all kinds of data and data stored. DynamoDB supports PartiQL via the Management Console, NoSQL Workbench, AWS CLI, and DynamoDB APIs. Be aware that DynamoDB supports only a small subset of the PartiQL language. The following query lists all tasks stored in table `todo-task`:

```
$ aws dynamo3db execute-statement \            ①
➥ --statement "SELECT * FROM \""todo-task\"""     ②
```

① The command execute-statement supports PartiQL statements as well.

② A simple SELECT statement to fetch all attributes of all items from table todo-task. The escaped " is required because the table name includes a hyphen.

PartiQL allows you to query an index as well. The following statement fetches all tasks with category equals `shopping` from index `category-index`:

```
$ aws dynamodb execute-statement --statement \
➥ "SELECT * FROM \""todo-task\"".\"category-index\""
➥ WHERE category = 'shopping'"
```

Please note that combining multiple queries (`JOIN`) is not supported by DynamoDB. However, DynamoDB supports modifying data with the help of PartiQL. The following command updates Emma's phone number:

```
aws dynamodb execute-statement --statement \
➥ "Update \"todo-user\" SET phone='+33333333' WHERE uid='emma'"
```

Be warned, an `UPDATE` or `DELETE` statement must include a `WHERE` clause that identifies a single item by its partition key or partition and sort keys. Therefore, it is not possible to update or delete more than one item per query.

Want to learn more about PartiQL for DynamoDB? Check out the official documentation: http://mng.bz/N5g2.

In our opinion, using PartiQL is confusing because it pretends to provide a flexible SQL language but is in fact very limited. We prefer using the DynamoDB APIs and the SDK, which is much more descriptive.

12.9 DynamoDB Local

Imagine a team of developers is working on a new app using DynamoDB. During development, each developer needs an isolated database so as not to corrupt the other team members' data. They also want to write unit tests to make sure their app is working. To address their needs, you could create a unique set of DynamoDB tables with a CloudFormation stack for each developer. Or you could use a local DynamoDB for offline development. AWS provides a local implementation of DynamoDB, which is available for download at **http://mng.bz/71qm**.

Don't run DynamoDB Local in production! It's only made for development purposes and provides the same functionality as DynamoDB, but it uses a different implementation: only the API is the same.

NoSQL Workbench for DynamoDB

Are you looking for a graphical user interface to interact with DynamoDB? Check out NoSQL Workbench for DynamoDB at **http://mng.bz/mJ5P**. The tool allows you to create data models, analyze data, and import and export data.

That's it! You've implemented all of nodetodo's features.

12.10 Operating DynamoDB

DynamoDB doesn't require administration like a traditional relational database, because it's a managed service and AWS takes care of that; instead, you only have a few things to do.

With DynamoDB, you don't need to worry about installation, updates, machines, storage, or backups. Here's why:

- DynamoDB isn't software you can download. Instead, it's a NoSQL database as a service. Therefore, you can't install DynamoDB like you would MySQL or MongoDB. This also means you don't have to update your database; the software is maintained by AWS.
- DynamoDB runs on a fleet of machines operated by AWS. AWS takes care of the OS and all security-related questions. From a security perspective, it's your job to restrict access to your data with the help of IAM.
- DynamoDB replicates your data among multiple machines and across multiple data centers. There is no need for a backup from a durability point of view. However, you should configure snapshots to be able to recover from accidental data deletion.

Now you know some administrative tasks that are no longer necessary if you use DynamoDB. But you still have things to consider when using DynamoDB in production: monitoring capacity usage, provisioning read and write capacity (section 12.11), and creating backups of your tables.

Backups

DynamoDB provides very high durability. But what if the database administrator accidentally deletes all the data or a new version of the application corrupts items? In this case, you would need a backup to restore to a working table state from the past. In December 2017, AWS announced a new feature for DynamoDB: on-demand backup and restore. We strongly recommend using on-demand backups to create snapshots of your DynamoDB tables to be able to restore them later, if needed.

12.11 Scaling capacity and pricing

As discussed at the beginning of the chapter, the difference between a typical relational database and a NoSQL database is that a NoSQL database can be scaled by adding new nodes. Horizontal scaling enables increasing the read and write capacity of a database enormously. That's the case for DynamoDB as well. However, a DynamoDB table has two different read/write capacity modes:

- *On-demand mode* adapts the read and write capacity automatically.
- *Provisioned mode* requires you to configure the read and write capacity upfront. Additionally, you have the option to enable autoscaling to adapt the provisioned capacity based on the current load automatically.

At first glance, the first option sounds much better because it is totally maintenance free. Table 12.1 compares the costs between on-demand and provisioned modes. All of the following examples are based on costs for `us-east-1` and assume you are reading and writing items with less than 1 KB.

Table 12.1 Comparing pricing of DynamoDB on-demand and provisioned mode

Throughput	On-demand mode	Provisioned mode
10 writes per second	$32.85 per month	$4.68 per month
100 reads per second	$32.85 per month	$4.68 per month

That's a significant difference. Accessing an on-demand table costs about seven times as much as using a table in provisioned capacity mode. But the example is not significant, because of the assumption that the provisioned throughput of 10 writes and 100 reads per second is running at 100% capacity 24/7. This is hardly true for any application. Most applications have large variations in read and write throughput. Therefore, in many cases, the opposite is correct: on-demand is more cost-efficient than provisioned. For example, we are operating a chatbot called marbot on AWS. When we switched from provisioned to on-demand capacity in December 2018, we reduced our monthly costs for DynamoDB by 90% (see **http://mng.bz/DD69** for details).

As a rule of thumb, the more spikes in your traffic, the more likely that on-demand is the best option for you. Let's illustrate this with another example. Assume your application is used only during business hours from 9 a.m. to 5 p.m. The baseline throughput is 100 reads and 10 write per second. However, a batch job is running between 4 p.m. and 5 p.m., which requires 1,000 reads and 100 writes per second.

With provisioned capacity mode, you need to provision for the peak load. That's 1,000 reads and 10 writes per second. That's why the monthly costs for a table with provisioned capacity costs is $24.75 higher than when using on-demand capacity in this example, as shown here:

```
// Provisioned capacity mode
$0.00065 for 1 writes/sec * 100 * 24 hours * 30 days = $46.80 per month
$0.00013 for 2 reads/sec * 500 * 24 hours * 30 days = $46.80 per month

Read/Writes with provisioned capacity mode per month: $93.60

// On-demand Capacity Mode
(7 hours * 60 min * 60 sec * 10 write) + (1 hour * 60 min * 60 sec * 100 writes) = 612
(7 hours * 60 min * 60 sec * 100 reads) + (1 hour * 60 min * 60 sec * 1000 reads) = 6,

$0.00000125/write * 612,000 writes/day * 30 days = $22.95
$0.00000025/read * 6,120,000 reads/day * 30 days = $45.90
```

```
Read/Writes with On-demand Mode per month: $68.85
```

By the way, you are not only paying for accessing your data but also for storage. By default, you are paying $0.25 per GB per month. The first 25 GB are free. You only pay per use—there is no need to provision data upfront.

Does your workload require storing huge amounts of data that are accessed rarely? If so, you should have a look into Standard-Infrequent Access, which reduces the cost for storing data to $0.10 per GB per month but increases costs for reading and writing data by about 25%. Please visit the https://aws.amazon.com/dynamodb/pricing/ for more details on DynamoDB pricing.

12.11.1 Capacity units

When working with provisioned capacity mode, you need to configure the provisioned read and write capacity separately. To understand capacity units, let's start by experimenting with the CLI, as shown here:

```
$ aws dynamodb get-item --table-name todo-user \
→ --key '{"uid": {"S": "emma"}}' \
→ --return-consumed-capacity TOTAL \                    ①
→ --query "ConsumedCapacity"
{
    "CapacityUnits": 0.5,                                 ②
    "TableName": "todo-user"
}
$ aws dynamodb get-item --table-name todo-user \
→ --key '{"uid": {"S": "emma"}}' \
→ --consistent-read --return-consumed-capacity TOTAL \   ③
→ --query "ConsumedCapacity"
{
    "CapacityUnits": 1.0,                                 ④
    "TableName": "todo-user"
}
```

① Tells DynamoDB to return the used capacity units

② getItem requires 0.5 capacity units.

③ A consistent read ...

④ ... needs twice as many capacity units.

More abstract rules for throughput consumption follow:

- An eventually consistent read takes half the capacity of a strongly consistent read.
- A strongly consistent `getItem` requires one read capacity unit if the item isn't larger than 4 KB. If the item is larger than 4 KB, you need additional read capacity units. You can calculate the required read capacity units using `roundUP(itemSize / 4)`.
- A strongly consistent `query` requires one read capacity unit per 4 KB of item size. This means if your query returns 10 items, and each item is 2 KB, the item size is 20 KB and you need five read units. This is in contrast to 10 `getItem` operations, for which you would need 10 read capacity units.
- A write operation needs one write capacity unit per 1 KB of item size. If your item is larger than 1 KB, you can calculate the required write capacity units using `roundUP(itemSize)`.

If capacity units aren't your favorite unit, you can use the AWS Pricing Calculator at https://calculator.aws to calculate your capacity needs by providing details of your read and write workload.

The provision throughput of a table or a global secondary index is defined in seconds. If you provision five read capacity units per second with `ReadCapacityUnits=5`, you can make five strongly consistent `getItem` requests for that table if the item size isn't larger than 4 KB per second. If you make more requests than are provisioned, DynamoDB will first throttle your request. If you make many more requests than are provisioned, DynamoDB will reject your requests.

Increasing the provisioned throughput is possible whenever you like, but you can only decrease the throughput of a table four to 23 times a day (a day in UTC time). Therefore, you might need to overprovision the throughput of a table during some times of the day.

Limits for decreasing the throughput capacity

Decreasing the throughput capacity of a table is generally allowed only four times a day (a day in UTC time). Additionally, decreasing the throughout capacity is possible even if you have used up all four decreases if the last decrease has happened more than an hour ago.

It is possible but not necessary to update the provisioned capacity manually. By using Application Auto Scaling, you can increase or decrease the capacity of your table or global secondary indices based on a CloudWatch metric automatically. See http://mng.bz/lR7M to learn more.

12.12 Networking

DynamoDB does not run in your VPC. It is accessible via an API. You need internet connectivity to reach the DynamoDB API. This means you can't access DynamoDB from a private subnet by default, because a private subnet has no route to the internet via an internet gateway. Instead, a NAT gateway is used (see section 5.5 for more details). Keep in mind that an application using DynamoDB can create a lot of traffic, and your NAT gateway is limited to 10 Gbps of bandwidth. A better approach is to set up a VPC endpoint for DynamoDB and use that to access DynamoDB from private subnets without needing a NAT gateway at all. You can read more about VPC endpoints in the AWS documentation at http://mng.bz/51qz.

12.13 Comparing DynamoDB to RDS

In chapter 10, you learned about the Relational Database Service (RDS). For a better understanding, let's compare the two different database systems. Table 12.2 compares DynamoDB and the RDS. Keep in mind that this is like comparing apples and oranges: the only thing DynamoDB and RDS have in common is that both are called databases. RDS provides relational databases that are very flexible when it comes to ingesting or querying data. However, scaling RDS is a challenge. Also, RDS is priced per database instance hour. In contrast, DynamoDB scales horizontally with little or no effort. Also, DynamoDB offers a pay-per-request pricing model, which is interesting for low-volume workloads with traffic spikes. Use RDS if your application requires complex SQL queries or you don't want to invest time into mastering a new technology. Otherwise, you can consider migrating your application to DynamoDB.

Table 12.2 Differences between DynamoDB and RDS

Task	DynamoDB	RDS Aurora
Creating a table	Management Console, SDK, or CLI `aws dynamodb create-table`	SQL `CREATE TABLE` statement
Inserting, updating, or deleting data	SDK or PartiQL (limited version of SQL)	SQL `INSERT`, `UPDATE`, or `DELETE` statement
Querying data	SDK `query` or PartiQL	SQL `SELECT` statement
Adding indexes to extend the possibility of query data	No more than 25 global secondary indexes per table	Number of indexes not limited per table
Increasing storage	No action needed; storage grows and shrinks automatically.	Increasing storage is possible via the Management Console, CLI, or API.
Increasing performance	Horizontal, by increasing capacity. DynamoDB will add more machines under the hood.	Vertical, by increasing instance size and disk throughput; or horizontal, by adding up to five read replicas
Distribute data globally	Multiactive replication enables reads and writes in multiple regions.	Read replication enables data synchronization to other regions as well, but writing data is possible only in the source region.
Installing the database on your machine	DynamoDB is available on AWS only. There is a local version for developing on your local machine.	Install MySQL or PostgreSQL on your machine.
Hiring an expert	DynamoDB skills needed	General SQL skills sufficient for most scenarios

12.14 NoSQL alternatives

Besides DynamoDB, a few other NoSQL options are available on AWS as shown in table 12.3. Make sure you understand the requirements of your workload and learn about the details of a NoSQL database before deciding on an option.

Table 12.3 Differences between DynamoDB and some NoSQL databases

Amazon Keyspaces	Columnar store	A fully managed Apache Cassandra–compatible database service	A good fit when dealing with very large data sets. Think of DynamoDB as an open source project. Available as a managed service by other vendors as well.
Amazon Neptune	Graph store	A proprietary graph database provided by AWS	Perfect fit for a social graph, personalization, product catalog, highly connected data sets
Amazon DocumentDB with MongoDB compatibility	Document store	A database service that is purpose-built for JSON data management at scale, fully managed and integrated with AWS. Amazon DocumentDB is compatible with MongoDB 3.6.	A good choice if you are looking for a NoSQL database that also brings flexible query capabilities. A common use case are common CRUD (create, read, update, and delete) applications.
Amazon MemoryDB for Redis	Key-value store	An in-memory database with durability, providing a Redis-compatible interface	A good match for implementing a cache, a session store, or whenever I/O latency is super critical

Cleaning up
Don’t forget to delete your DynamoDB tables after you finish this chapter like so:

```
aws dynamodb delete-table --table-name todo-task
aws dynamodb delete-table --table-name todo-user
```

Summary

- DynamoDB is a NoSQL database service that removes all the operational burdens from you, scales well, and can be used in many ways as the storage backend of your applications.
- Looking up data in DynamoDB is based on keys. To query an item, you need to know the primary key, called the partition key.

- When using a combination of partition key and sort key, many items can use the same partition key as long as their sort key does not overlap. This approach also allows you to query all items with the same partition key, ordered by the sort key.
- Adding a global secondary index allows you to query efficiently on an additional attribute.
- The `query` operation queries a table or secondary indexes.
- The `scan` operation searches through all items of a table, which is flexible but not efficient and shouldn't be used too extensively.
- Enforcing strongly consistent reads avoids running into eventual consistency problems with stale data. But reading from a global secondary index is always eventually consistent.
- DynamDB comes with two capacity modes: on demand and provisioned. The on-demand capacity mode works best for spiky workloads.