UNIVERSITÉ LIBRE DE BRUXELLES

Distributed Database Systems and DynamoDB

Submitted by

Jose Carlos Badillo
Elena Ouro Paz

for partial fulfillment of the course

Advanced Databases (INFO-H-415)

professor

Esteban Zimányi

Brussels, December 2018

Table of contents

1. Introduction

The concept of database systems has been reinvented throughout history from the initial data records kept in paper to current state-of-the-art databases capable of storing and providing easy access to petabytes of data.

This evolution, like most technological advances, is motivated by need. The need to store increasing amounts of data, provide access to the data to users located all over the world or decrease the necessary time to obtain certain data, for example. Many technologies have been researched and developed in order to tackle these challenges and many others. These technologies co-exist nowadays to satisfy the many different needs a system can have, leaving it up to the designer of the system to identify the most suitable technology.

This paper focuses on one of these technologies, the distributed databases. We define a distributed database as a collection of multiple, logically interrelated databases distributed over a computer network. Therefore, a Distributed database system is based on the union of a database system and computer network technologies.[1]

Early versions of distributed database management systems appear as early as the mid-1970s. The early versions often consisted of a centralized database and replicas of the data in several other locations. Changes to the data happened locally and then they were synchronized periodically. An example of such a type of Distributed DBMS is the one used by the Australian Department of Defence at the time.[2]

The motivation to mix databases and network has come from the need of large-scale applications that tend to have millions of requests from all around the world in short times and need certain modularization in order to serve such charges in an efficient way. Moreover, nowadays the monolithic approach of building software is being abandoned due to the strict consequences that doing even small changes on code or scale require.

In contrast, we have the quick adoption of the microservices architecture which conceives the idea of an application as a collection of loosely coupled services handling different business capabilities. Therefore, if we mapped this architecture to the database domain we may end up with a distributed database composed of different simpler and smaller databases handled for each microservice or business domain. The challenge now would be to guarantee the choreography of the different services and databases in our system.

Within this document, we are going to show a simple case of study implemented with Java and Amazon **DynamoDB** which nowadays is one of the commercial databases that handles the challenges of distributing data in a transparent way for the developers.

2. Promises of Distributed Database Systems

Some of the most common reasons for partitioning and distributing data come from the natural divisions that we can abstract from the real world. For instance, a customer's database of a multinational firm. It could be for security, performance or analytical reasons that this database needs to be divided into geographical regions like continents, countries or maybe states depending on the granularity the business is looking for.

It also happens that certain organizations within the company still require to write or read such distributed database doing operations that involve the full content of the data. In such cases, it is necessary to satisfy the partitioning and distribution needs over different places but also in order to favorize the global operations we would have to replicate the data needed at other sites for performance and reliability reasons. In the worst case, we might end up with a clone of the full database in every site so we still give the performance expected.

Partitioning and Replication are tasks that every Distributed Database should manage in a transparent way, meaning that user can still pose a query and trust that the transaction requested will retrieve a correct result without caring about replicas or location of the data and always keep the consistency without penalizing the performance.

Moreover in the case that one of the sites managing the data falls down unexpectedly, it should be transparent to the user and should not bring the entire system down. In the best of the cases, the full system keeps working with an alternative site that substitutes the node that went down.

In the worst case, only a small fraction of our system will be down losing availability, but all the other modules will be able to operate normally and notify their updates to the fallen node once it comes online again. In that way, the system will automatically be updated and recover its consistency status.

3. Design of a Distributed Database System

This section deals with the fundamentals of designing a distributed DBMS. Knowing the definition of a DDBMS we can intuitively identify that the main question to be answered during the design process is how the data will be distributed, identifying which data is relevant to each of the locations and how should it be stored there.

There are two basic design patterns that can be followed to place the data: "*partitioned*", where the data is divided in several fragments each of which is placed at a different site, and "*replicated*" where all or some of the partitions of the data are stored in each or some of the sites.

For partitioned data, there are two fundamental issues: *fragmentation* and *allocation*. These two issues are explained in depth in sections 3.1 and 3.2.

3.1. Fragmentation

Fragmentation refers to the division of data into partitions or fragments. There are two types of fragmentation of data: "vertical" and "horizontal". The former is intended to have full rows in different blocks and the later will divide the columns. Whatever the case is, the fragmentation must accomplish the following 3 rules:

- Completeness: Each data item must be included in one of the partitions.
- Reconstruction: There must be an operator that will put all the data together again without adding or losing items.
- Disjointness: Each data items is in one and only one of the partitions created.

The type of fragmenting the database is an important decision as it affects the performance of query execution.

3.1.1. Horizontal Fragmentation

We can horizontally partition each Relation in a database following two distinct behaviors.

- **Primary horizontal fragmentation** of a relation is performed using predicates that are defined on that relation.
- **Derived horizontal fragmentation**, is the partitioning of a relation that results from predicates being defined on another relation.

However, In order to take a decision, it is important to note how the database relations are connected to one another, especially with joins. We must use the predicates used in user queries. If it is not possible to analyze all of the user applications to determine these predicates, one should at least investigate the most “important” ones.

3.1.2. Vertical Fragmentation

This type of partitioning is also known as column fragmentation. each partition will have a subset of the attributes of R and the primary key of R. Vertical Fragmentation allows the user queries to deal with smaller relations which can be translated to a lower number of page accesses. This type of partitioning also places in one fragment those attributes usually accessed together.

3.1.3. Fragmentation on DynamoDB

DynamoDB was designed in a way that stores data in horizontal partitions. Partition management is handled entirely by DynamoDB, therefore you never have to manage partitions yourself.

To write an item to the table, DynamoDB uses the value of the partition key as input to an internal hash function. The output value from the hash function determines the partition in which the item will be stored.

To read an item from the table, you must specify the partition key value for the item. DynamoDB uses this value as input to its hash function, yielding the partition in which the item can be found.[3]

3.2. Allocation

Once we have the correct set of fragments of the data, we need to decide where will be located each fragment. The allocation problem involves finding the “optimal” distribution of fragments in a set of Sites. Then we must analyze the cost of query the Fragments from each site.

It is also important to define whether replicas of them are needed in a certain site or the fragment will be maintained as a single copy. In the case of replication, we might have a fully replicated database or only certain fragments could be distributed to certain sites which means it is a partially replicated database. Then we need to think about the cost of updating the fragment at all sites where it is stored.

In order to determine the allocation, we need to know the quantitative data about the database, the applications that run on it, the communication network, the processing capabilities, and storage limitations of each site on the network.

3.2.1. Allocation on DynamoDB

DynamoDB is automatically replicated across multiple Availability Zones within an AWS Region. Therefore every fragment is allocated in each of the Amazon sites available so we consider this a fully replicated database. Again all this replication behavior is managed by dynamoDB and you don't have to handle it by yourself. As a developer, the most probable is that you will never notice from which site your requests have been served.

4. Data Integration

Data Integration is the combination of technical and business processes used to combine data from disparate sources into meaningful and valuable information.[5] This kind of integration process is necessary on the scenario when there are already some existing databases and we need to integrate them in order to form only one single database or a distributed database. In order to integrate them, we first need to define a mediation schema or better know as global concept schema, which is going to be the one governing the common information about the different sources.

The data integration can be divided into two types: "physical" or "logical" depending on the implementation and the needs that we must satisfy with such implementation.

- **Physical integration** refers to the materialization of integrated data into data warehouses and it is usually used for OLAP analysis.

- **Logical integration** is more suitable for OLTP databases where the GCS is only the mediator to make operations between the different databases.

In both cases, the objective is to find the correct GCS and to execute the correct transformation of the data in order to be manipulated from one side to another.

4.1. Data Integration on DynamoDB

DynamoDB is a flexible NoSQL database supporting document and key-value data models. Within DynamoDB apart of the Partition key Type and Sorted Key type, you don't have to define the database schema upfront and even better each of the tuples inside the database could have a totally different structure which facilitates the data integration but also obligate the applications consuming data to be more careful when retrieving it.

5. Access Control

One of the most common alternatives to protect data is the use of Views. THe main purpose of this tool is to hide information and to provide an indirect way to manipulate information without going straight to the base relations.

5.1. Virtual Views

A view is a virtual relation, defined as the result of a query on base relations. It is considered a dynamic window in the sense that it reflects all the updates to the database. It can be used as a media of access control because views hide some data. Therefore if we assume that users can only see information of the database through views then all the hidden data would be safe from access and manipulation.

In context of a Distributed Database a view could be derived from distributed relations allocated in different sites. The access to a view requires the execution of a distributed query corresponding to the view definition. The view definitions can be centralized at one site, partially duplicated, or fully duplicated.

5.2. Materialized Views

A materialized view stores the tuples of a view in a database relation, like the other database tuples, possibly with indices. This means that we are physically writing the tuples that answer the query involved in the view, therefore, it is faster to retrieve the data of the query but this improvements has a price to pay which is the maintenance of the materialized views.

The maintenance of the view is the process to update or refresh the materialized view in order to reflect the changes done in the based relations. A view can be refreshed immediately or deferred the former will update the view as part of the transaction that updates the base relations in such way the materialized view will be always consistent. However, the cost of the transaction is higher and this could become very difficult if the materialized view is in a different site than the base data leaving us with the need of a distributed transaction.

In practice the most common approach is the deferred mode, meaning that we are going to use a different transaction in order to refresh the view. This refresh can be triggered lazily (before a query is evaluated on the view) or periodically (at predefined times).

5.3. Data Security

Every database management systems must protect the data against unauthorized access. It means that only authorized users should be able to perform operations they are allowed to perform on the database. The detail of the authorizations must be refined so that different users have different rights on the same database objects.

The majority of database systems will handle this granularity of access granting or revoking selecting, inserting, deleting or updating permissions to a user over certain objects. In the context of a distributed database we must deal with Remote user authentication since any site may accept programs initiated, and authorized, at remote sites. The most common way to deal with the remote authentication information is to keep it at a central site for *global users* which can then be authenticated only once and then provide access to multiple sites based on the global authentication.

DynamoDB handles all the access control using AWS IAM. You can create users with specific permissions to access and create DynamoDB tables. You can create a special IAM Condition to restrict user access to only their own records.[4]

6. Query Processing

The majority of the DBMS use a non-procedural language like SQL that hides the low level detail about the physical organization of the data. This means that the user doesn't have to know the procedures needed to access the data in order to construct the answer of a query. Instead it is the query processor the one in charge of translating the query into an algebraic query and then finding the best approach to retrieve the data.

When we involve the distributed behaviour into the game, then the data accessed from the query needs to be localized and the operations would be translated into fragment based operations. In other words, the objective of query processing is to transform a high-level query into an efficient execution strategy expressed in a low-level language on local databases.

In order to do so, we follow a similar approach than for a centralized databases. The only difference comes when retrieving the best physical plan as we must consider the database statistics of the distribution. This means that we are going to rely now on fragments and its cardinality and size. We might consider as well the communication costs and therefore if there are existing replicas we should be able to favorize the consumption of those that will minimize the communication costs when reading data from one site to another.

In order to provide the best plan to query for a distributed database we start with a process called Query decomposition which basically maps a distributed query into an algebraic query on global relations. This is the same analysis that we make for a centralized database. The next phase is the one that changes as we need to rebuild the global relation by applying reconstruction rules that produces a relational algebra program whose operands are the fragments. This is the localization of the information among the different fragments.

6.1. Query Processing in DynamoDB

In general, the design and the architecture expected for a database implemented on **DynamoDB** is to have only one single Table where we partition our data based on a Partition Key and then we could optionally add a Sorted Key that helps to sorts the tuples within the different partitions.

This kind of design will facilitate the query processing as we don't need to execute any other operation than retrieving the correct fragment where the data has stored. In order to do that dynamoDB offers a query based on the partition key which will run a hash function and simply retrieve all the information on that partition. In case we want to have further selectivity of the data we are expected to scan the data but having filtered the data with the correct Partition key, then this shouldn't be a costly task to compute.

7. Transaction Management

A transaction is a basic unit of consistent and reliable computing. We can talk about two different scenarios of consistency: "database consistency" and "transaction consistency".

We say a database is consistent if it accomplish all the constraints defined over it before and after a set of insertions or deletion that could occur at certain time. It is expected that a database never enters in a inconsistent state. The only time when a database could be temporarily inconsistent is during the execution of transactions, however it should recover its consistent state once the transactions has finished.

Transaction consistency refers to the actions of concurrent transactions. In the context of distributed databases the complexity grows if we consider replicas that are expected to be in mutually consistent state, meaning that all the copies should have the same value in all their items.

Each transaction has two possible outcomes, if it finished successfully then the transactions commits. And this is considered a point of no return where the changes are permanently written in disk.

In the other hand, the transaction can be aborted and all the already executed actions are undone, going to the last state of the database. This is a rollback.

8. NoSQL Databases

Since its appearance in 1989 the Standard Query Language (SQL) has dominated the market and it is to this day used in many database systems. However, with the years that have passed since the birth of SQL much has changed, SQL was designed in a time when computation was relatively cheap but storage was hard to obtain, the first hard drive to be able to store 1GB of data in the 1980s weighted 550 pounds and costed 40,000$.[6] Nowadays the amount of data we generate and want to store has grown exponentially, storage space is now relatively cheap compared to the computational costs over such large quantities of data. This fact, together with the need for better scalability compels research labs and companies to explore alternatives to the traditional relational databases and SQL. NoSQL is one of these alternatives. NoSQL, opposite to what the name may suggest, is not a query language like SQL, it usually refers to many types of databases all of them not based on SQL.
The main NoSQL database types are:

- Document databases (MongoDB)
- Graph databases (Node4j)
- Wide-column databases (Cassandra)
- Key-value databases (DynamoDB)

DynamoDB is a key-store database, this means that every record in the database is a pair containing a key and a value. The value may be anything from a simple number or string to a complex object. Key-value databases are usually implemented as hash tables which makes accessing a pair by its key very efficient. This is the case for DynamoDB.[7]

9. Case of Study

We have developed a small application to exemplify and better explain how to build a simple database system on dynamoDB. In this section of the document, we will go through some of the peculiarities of database system design on dynamoDB as well as how to perform some of the basic actions such as creating a table, inserting, updating and deleting items from the table and how to perform a few queries. All the code found in this section is in Java and uses the official AWS SDK but dynamoDB can be used for other programming languages as well.

Our case of study adapts the design of a relational database containing information from the HR department of a company. A schema of the initial database is illustrated in figure 1. For simplicity, some of the attributes in the original database have removed since they did not illustrate any of the special features of DynamoDB. Our case study is based on an example used by amazon to explain DynamoDB design.[8]

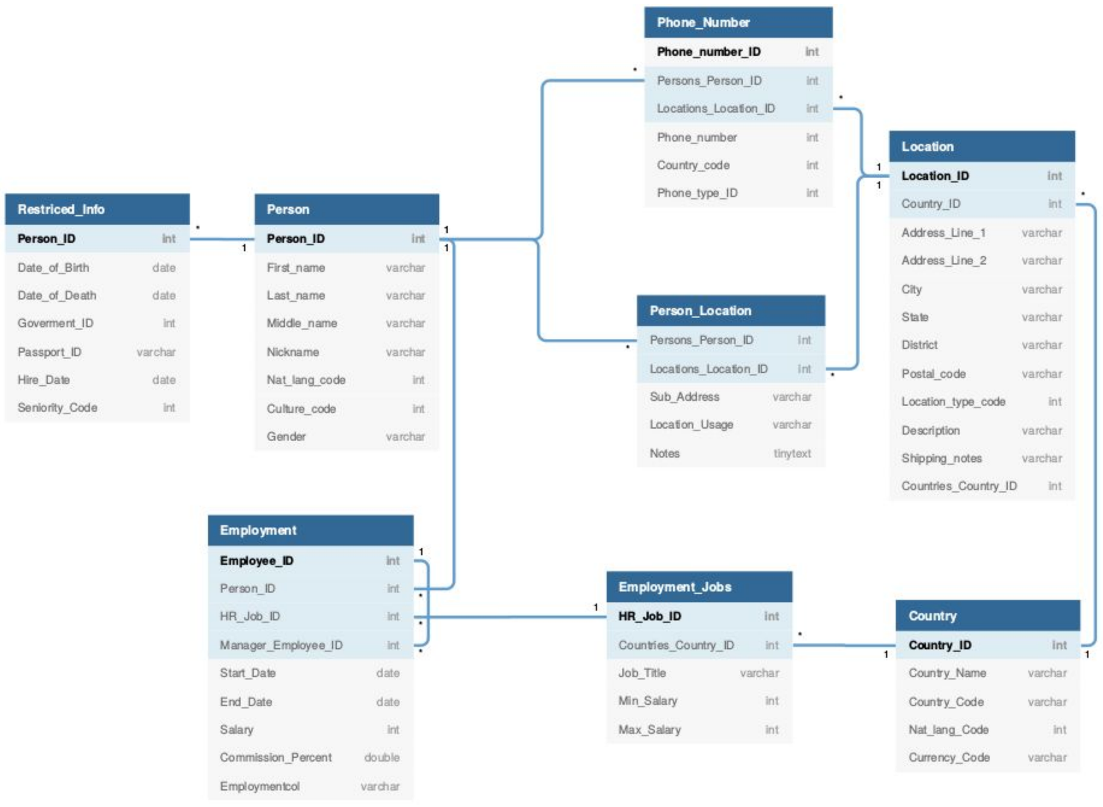


Figure 1: schema of the relational database in which the study case is based

9.1. Database design on dynamoDB

NoSQL databases promise to improve scalability and increase efficiency when executing queries. However, the way in which they are implemented means that to actually accomplish this we have to carefully design our database system.
As mentioned in the previous section, DynamoDB is implemented as a hash table which makes querying information by its key very efficient but querying the database by a different attribute may result in very high computational costs since we might have to scan all items of a table. NoSQL databases can only be queried efficiently in a limited number of ways.

The key to a good NoSQL database design is to first identify which information our database needs to provide. This means defining the queries that will be run in the database. It is essential to understand the business problems of our application and the user cases before designing the database. [9]
Having a list of all queries will allow for a design of the database that optimizes those queries as much as possible. It is also recommended to estimate how often each of the queries will be triggered.

For our case study 5 queries have been identified. These queries can be found on table 1. Of course, in a real life application the list of queries that we would be interested in would be much longer.

1. Get employee details by employee ID
2. Get employee details by employee name
3. Get the current job details of an employee
4. Get details of all employees hired recently
5. Get all employees with a specific job

Table 1: List of queries for the case of study

Traditional relational databases abstract the data to find patterns and then group data together by type. Each table contains a list of records of the same type but records in the same table might not be connected to each other. NoSQL databases like dynamoDB prefer to store data in a way that tries to resemble how data is represented in the real world by storing all related data together.
From this we can infer that the best way to design a database in dynamoDB is by having a limited number of tables. The most efficient designs often consist of a unique table allowing us to access all necessary information without the need of joining tables.

Taking this best practices into account our case of study implements one unique table that contains all the information of the HR department.

Once we are set on designing our database with only one table we need to decide how the data will be organized in this table. DynamoDB uses a primary key to identify related data. All information that in a relational database would be distributed across tables and connected through foreign keys will have the same primary key in dynamoDB.
Although, the idea of the primary key in DynamoDB is similar to the one of it's relational database counterpart it has some peculiarities.

DynamoDB supports two types of primary keys: the first is the simple primary key, composed of only one attribute named *"partition key"*. This is the most similar to the primary keys in other database systems. The second is the composite primary key, composed of two attributes, the *"partition key"* and the *"sort key"*.[10]

DynamoDB stores data items in 10GB storage units called partitions. In order to allocate items in a certain partition DynamoDB uses an internal hash function that takes the partition key as input. Therefore, all items with the same partition key can be found in the same partition as long as they do not exceed the 10GB available.
Items can be efficiently retrieved using their partition key.

The sort key defines the order in which items are stored in a partition. Additionally, if the collection of items with a given partition key is larger than the available space of 10GB DynamoDB splits partitions by sort key.

Using a well designed sort key has two main benefits: the first one is that by having ordered data we can efficiently retrieve ranges of items using operators such as starts-with, between, < and > among others. The second benefit is that sort keys allow definition of hierarchical one-to-many relationships in the data.

Sort keys are often used also for version control, this is achieved by automatically generating a copy of each item that is inserted into the database, one of the copies should be updated and always contain the most up-to-date state for the item while the other one will be used as a historical record. To distinguish them we add prefixes to their sort keys, typically *"v0_"* for the copy containing the latest version of the item and a version code for the others.[11]

In the study case we have not used the sort key for version control although doing it could make the system more efficient to show more varied queries to the database.

To define our primary key we have identified a series of entities from the original relational database schema. Each entity is a type of object that will be stored in our table. For each of the entities we have defined a partition and a sort key that can be found in table 2.

Table 3 illustrates what our table will look like once some items have been added.

Entity name	**Partition key**	**Sort key**
HR-Employee	“HR-”+Employee ID	Employee ID
HR-Job	“HR-JOB”+Job number	Job ID
HR-Country	“HR-COUNTRY”+country number	Country name
HR-Region	“HR-REGION”+region number	Region name
HR-Location	“HR-LOCATION”+Location number	Country name
HR-Department	“HR-DEPARTMENT”+Department	Department ID

Table 2: partition and sort key per type of item

Primary key		Attributes			
PK	**SK**				
HR-EMPLOYEE1	EMPLOYEE1	**Data (full name)**	**Phone no.**	**...**	**City**
		John Smith	665438592	...	Seattle
	HR-CONFIDENTIAL	**Data (Hire date)**	**Date of birth**	**...**	**Passport**
		2015-11-08	1977-02-20	...	ABC100000
	J-AM3	**Data (Job title)**	**Department ID**	**...**	**End date**
		Account Manager	COMMERCIAL	...	9999-12-31
HR-JOB1	J-AM3	**Data(Job title)**			
		Account Manage			
HR-DEPARTMENT1	COMMERCIAL	**Data (Dpt. name)**	**Address**	**...**	**Location**
		Commercial sales	12 Spring rd.	...	WA|SEATTLE

Table 3: schema of the table after loading a few items

Having chosen this partition and sort keys we can now efficiently retrieve the employee details by ID as asked in query number one as well as retrieving the employee details by name, finding all items with an ID starting with “HR-EMPLOYEE” filtered by the employee name. In case of using the sort key for version control we would also be able to find very efficiently the current job details of an employee.
To improve the performance of the other two queries in dynamoDB we can use a secondary index, a data structure that contains a subset or all attributes from a table. The table to which a secondary index is known as a base table.
Secondary indexes work in a similar manner to tables: when the user creates one a primary key is selected as well as a list of attributes from the base table that we would like to project into the index. This list of attributes can be comprised of just the primary key, all attributes from the base table or a subset of them.

DynamoDB supports the use of two different secondary indexes: *Global Secondary Index* (GSI) and *Local Secondary Index* (LSI).[12] There are many aspects in which GSIs and LSIs differ, the main ones are:

- Key Schema: GSIs primary key can be either simple or composite and it does not need to be the same as the one of its base table. LSIs always have a composite key, the partition key has to be the same as the one from the base table but the sort key can be different.

- Size restrictions: GSIs have no size restrictions while for LSIs the total size of indexed items of each partition key cannot be more than 10GB.

- Online Index partitions: LSIs have to created at the same time as the base table while GSIs can be created at any time.

- Queries and Partitions: GSIs allow queries over the entire table while LSIs only allow for queries inside the partition indicated by the partition key.

- Projected attributes: When querying a GSIs only attributes that are projected into the index can be requested. When querying a LSI any attribute can be requested, if it has not been projected into the index it will be automatically fetched from the base table.

If we look at the queries we want to optimize we can see that we would like to query the database by using an attribute that its not the partition key. Therefore, we should use a GSI over the table previously designed, for every type of item we have selected a partition key and a sort key, the partition key of the GSI will be the sort key of the table base while the sort key will be the data attribute, along with the primary key all attributes from the base table will be projected into the GSI.[13] Table 4 illustrates the GSI design.

This design optimizes queries four and five. Query number 4 requires that we fetch all employees that have have been hired recently, in this case que can query the GSI to obtain all items with "HR-CONFIDENTIAL" as their partition key, sorted by hire date.
For query number 5 we can retrieve all job items with the given job title.

Primary key		**Attributes**		
PK	**SK**			
EMPLOYEE1	**Full name**	**ID**	**...**	**City**
	John Smith	HR-EMPLOYEE1	...	Seattle
HR-CONFIDENTIAL	**Hire date**	**ID**	**...**	**Passport**
	2015-11-08	HR-EMPLOYEE1	...	ABC100000
J-AM3	**Job title**	**ID**	**...**	**End date**
	Account Manager	HR-EMPLOYEE1	...	9999-12-31
COMMERCIAL	**Dpt. name**	**ID**	**...**	**Location**
	Commercial sales	HR-DEPARTMENT1	...	WA|SEATTLE

Table 4: Schema of the table after loading a few items in the GSI

9.2. Creating a table

Having a finalised design for our database we can start with the implementation of the system. The next few sections of this document will show pieces of the code that we have used to implement the database. The first thing that we will focus on is how to create the table designed in the previous section as well as the GSI.

The first thing we need to build our table is to connect to dynamoDB, the code to do so can be found in code 1. For our example we are using dynamoDB locally but the client can be built in a similar manner but changing the endpoint.

```
    AmazonDynamoDB client = AmazonDynamoDBClientBuilder
            .standard()
            .withEndpointConfiguration(new AwsClientBuilder
                    .EndpointConfiguration("http://localhost:8000",
 "local"))
            .build();

    DynamoDB dynamoDB = new DynamoDB(client);
```

Code 1: Connection to dynamoDB.

The code 2 contains a snippet of the code that we used to create the table. The first thing that we see in the code is a *CreateTableRequest,* this instance stores all the information necessary to build the table. first we add the table name to the request, in our case the name is “HR”, followed by a provisioned throughput. The provisioned throughput capacity specifies the number of reads and writes per second that the application needs for that table. Each read unit allows for a strongly consistent read per second or two eventually consistent reads for a 4KB item while each write unit allows for one write per second for an item up to 1KB in size. For the purpose of our study case we have not dwelled too much into this since the database will be very smaller compared to a real life one.
The following attribute to be added to the create table request is a list of attributes, we have only defined three since this are the three attributes that will be present in every item of the table, “ID”, “Name” and “Data”, all of them of type String. After that we indicate that “ID” will be out partition key and “Name” the sort key. Finally, we add the global secondary index to the table, the code defining the GSI can be find in code 3. After that we only need to invoke the createTable method with the request and wait for the table to become active before we can perform any other operations on the table.[14]

To generate the GSI we create a new instance of the class *GlobalSecondaryIndex,* the attributes required for this are the name of the GSI, the provisioned throughput, the key schema indicating that “Name” is the partition key (KeyType.HASH) and “Data” is the sort key (KeyType.RANGE) and the projection of attributes. Projection can take as values *KEY_ONLY*, *ALL* or *INCLUDE*, indicating if the GSI should project only the key values, all values from the base table or a subset. In case of using INCLUDE it needs to be followed by a list of the attributes.[15]

```
public static final String PK = "ID";
public static final String SK = "Name";
public static final String GSI_1_SK = "Data";
[...]
CreateTableRequest createTableRequest = new CreateTableRequest()
         .withTableName(tableName)
         .withProvisionedThroughput( new ProvisionedThroughput()

                                          .withReadCapacityUnits(10L)
                                          .withWriteCapacityUnits(10L))
         .withAttributeDefinitions( Arrays.asList(
               new AttributeDefinition( PK, ScalarAttributeType.S),
                new AttributeDefinition( SK, ScalarAttributeType.S),
                new AttributeDefinition( GSI_1_SK, ScalarAttributeType.S)))
         .withKeySchema( Arrays.asList(
                new KeySchemaElement( PK, KeyType.HASH),
                new KeySchemaElement( SK, KeyType.RANGE)))
         .withGlobalSecondaryIndexes(GSI);
Table table = dynamoDB.createTable( createTableRequest );
table.waitForActive();
```

Code 2: Snippet of the table creation step.

```
ProvisionedThroughput ptIndex = new ProvisionedThroughput()
                                  .withReadCapacityUnits(1L)
                                  .withWriteCapacityUnits(1L);

GlobalSecondaryIndex GSI1 = new GlobalSecondaryIndex()
                              .withIndexName(GSI_1_NAME)
                              .withProvisionedThroughput(ptIndex)
                              .withKeySchema(
                                  new KeySchemaElement()
                                      .withAttributeName(SK)
                                      .withKeyType(KeyType.HASH),
                                 new KeySchemaElement()
                                      .withAttributeName(GSI_1_SK)
                                      .withKeyType(KeyType.RANGE)
                               )
                              .withProjection( new Projection()
                             .withProjectionType("ALL"));
```

Code 3: Generation code for a GSI.

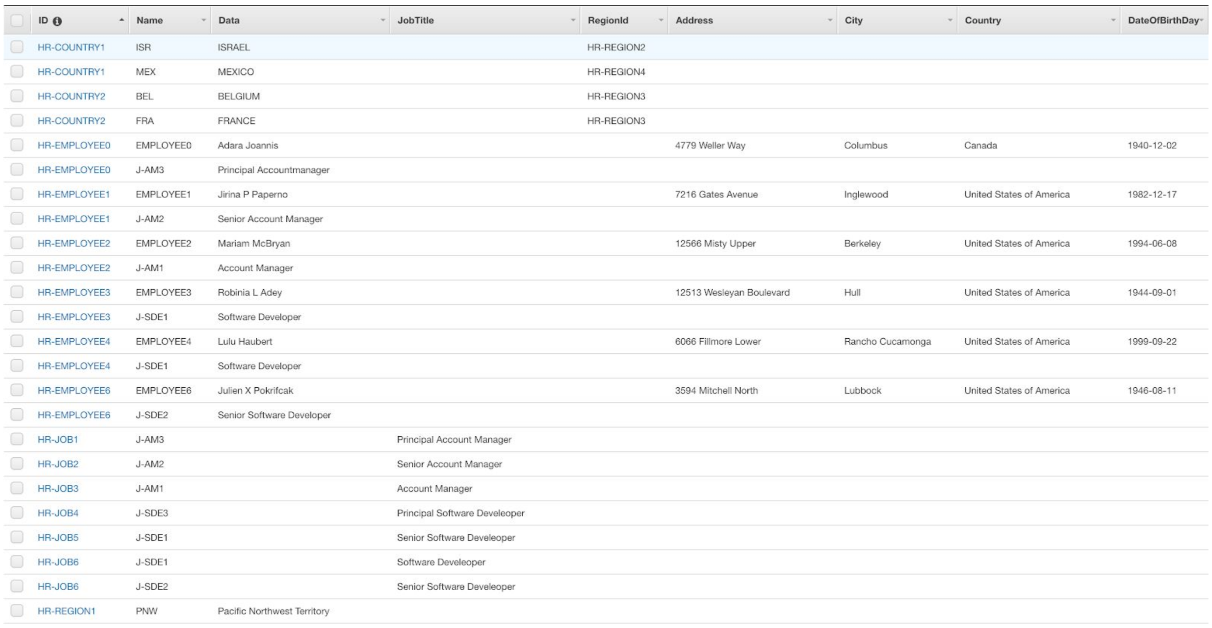
9.3. Loading data into a table

The easiest and fastest method to load data into the database with DynamoDB is to read the data items from a JSON file. Code 4 shows how the items are loaded in our system. We have used the JSON simple library in order to parse the JSON file but many other libraries can be used for this purpose.
We start by retrieving the table to which we wish to load the data from the DynamoDB client, then, we proceed to parsing the JSON file, the file contains an array of JSON objects, each one an item to be loaded into the database. For each of the items in the file we initialize an instance of the *Item* class, to it we every attribute that can be found inside the file by means of the *withInt*, *withLong*, *withString* and *withDouble* methods depending on the type of the attribute being the primary key an exception to this, regardless of its type we add the primary key with the method *withPrimaryKey.* Methods for other types of data exist but they are not used in our database.
Once the item instance has all its attributes we load it into the table by invoking the method *putItem*.[16]

```
    Table table = dynamoDB.getTable( tableName );
    JSONParser parser = new JSONParser();
    JSONArray items = null;
    items = (JSONArray)parser.parse(new FileReader(path));
    if( items != null ) {
         int i = 0;
         for (Object o : items) {
              JSONObject current = (JSONObject) o;
              Iterator it = current.entrySet().iterator();
              Item DBItem = new Item();
              String id = current.get( CreateTable.PK ).toString();
              String name = current.get( CreateTable.SK ).toString();
              DBItem.withPrimaryKey( CreateTable.PK, id, CreateTable.SK, name );
               for(Object key : current.keySet()) {
                     if( !key.toString().equals( CreateTable.PK )
                     && !key.toString().equals( CreateTable.SK )) {
                                 if( current.get(key) instanceof String ) {
                                       DBItem.withString( key.toString(),
                                        current.get(key).toString() );
                                 } else if ( current.get(key) instanceof Long ){
                                       DBItem.withLong( key.toString(),
                                        (Long)current.get(key) );
                                 } else if (current.get(key) instanceof
Integer){
                                       DBItem.withInt( key.toString(),
                                        (Integer)current.get(key) );
                                 } else if (current.get(key) instanceof Double){
                                       DBItem.withDouble( key.toString(),
                                        (Double)current.get(key) );
                                 }
                           }
                     }
                table.putItem(DBItem);
         }
    }
```

Code 4: Code to load data from a JSON into a table.

The data generated can be visualized very easily from the AWS Console within the DynamoDB service. The Figure 2 shows the information contained in DynamoDB after the execution of code 4.



ID	Name	Data	JobTitle	RegionId	Address	City	Country	DateOfBirthDay
HR-COUNTRY1	ISR	ISRAEL		HR-REGION2				
HR-COUNTRY1	MEX	MEXICO		HR-REGION4				
HR-COUNTRY2	BEL	BELGIUM		HR-REGION3				
HR-COUNTRY2	FRA	FRANCE		HR-REGION3				
HR-EMPLOYEE0	EMPLOYEE0	Adara Joannis			4779 Weller Way	Columbus	Canada	1940-12-02
HR-EMPLOYEE0	J-AM3	Principal Accountmanager						
HR-EMPLOYEE1	EMPLOYEE1	Jirina P Paperno			7216 Gates Avenue	Inglewood	United States of America	1982-12-17
HR-EMPLOYEE1	J-AM2	Senior Account Manager						
HR-EMPLOYEE2	EMPLOYEE2	Mariam McBryan			12566 Misty Upper	Berkeley	United States of America	1994-06-08
HR-EMPLOYEE2	J-AM1	Account Manager						
HR-EMPLOYEE3	EMPLOYEE3	Robinia L Adey			12513 Wesleyan Boulevard	Hull	United States of America	1944-09-01
HR-EMPLOYEE3	J-SDE1	Software Developer						
HR-EMPLOYEE4	EMPLOYEE4	Lulu Haubert			6066 Fillmore Lower	Rancho Cucamonga	United States of America	1999-09-22
HR-EMPLOYEE4	J-SDE1	Software Developer						
HR-EMPLOYEE6	EMPLOYEE6	Julien X Pokrifcak			3594 Mitchell North	Lubbock	United States of America	1946-08-11
HR-EMPLOYEE6	J-SDE2	Senior Software Developer						
HR-JOB1	J-AM3		Principal Account Manager					
HR-JOB2	J-AM2		Senior Account Manager					
HR-JOB3	J-AM1		Account Manager					
HR-JOB4	J-SDE3		Principal Software Develeoper					
HR-JOB5	J-SDE1		Senior Software Develeoper					
HR-JOB6	J-SDE1		Software Develeoper					
HR-JOB6	J-SDE2		Senior Software Develeoper					
HR-REGION1	PNW	Pacific Northwest Territory						

Figure 2: data content of the table populated with code 4

9.4. Removing items from a table

Removing an item is a simple task on dynamoDB if we have it's primary key, the code to do this can be found on code 5, we simply need to retrieve the table and invoke *deleteItem* over it. *deleteItem* takes an instance of *DeleteItemSpec* as sole parameter, it contains the primary key to identify the item to be deleted.
In some instances we might need to delete an item in case it satisfies a given condition, to do so a condition expression can be added to the *DeleteItemSpec*, an example of this can be found on code 6.. This method of deletion can only remove one item from the database while in circumstances we might want to remove several items.[17] To solve this, DynamoDB also supports batch deletion of items, through the operation *BatchWriteItem* can comprise as many as 25 operations but it does not support conditions.[18]

```
Table table = dynamoDB.getTable(tableName);
DeleteItemSpec deleteItemSpec = new DeleteItemSpec();
deleteItemSpec.withPrimaryKey(new PrimaryKey("ID", PK, "Name", SK));
table.deleteItem(deleteItemSpec);
```

Code 5: Delete process of an item from a table given its primary key.

```
Table table = dynamoDB.getTable(tableName);
   DeleteItemSpec deleteItemSpec = new DeleteItemSpec();
   deleteItemSpec.withPrimaryKey(new PrimaryKey("ID", PK, "Name", SK))
                  .withConditionExpression("City = :c")
                  .withValueMap(new
ValueMap().withString(":c","Seattle"));
   table.deleteItem(deleteItemSpec);
```

Code 6: Conditional delete of an item.

9.5. Updating items

The update process works very similarly to loading data. For our study case we take the updated information of an item from a JSON file, the process we follow can be found in code 7. We start as in previous occasions by retrieving the table and parsing the JSON file, we need an instance of *updateItemSpec* that will keep all the information necessary for the item update: the primary key is added through the *withPrimaryKey* method, an update expression needs to be built and added through the method *withUpdateExpression,* the expression indicates how each attribute needs to be updated, in this example we are substituting the values in the database for the ones in the JSON.

If we want to update, for example, the attributes “City” and “Phone number” of an employee and give them the values: “NY” and “678362743” respectively, the update expression would be of the form: *“City = :a1, PhoneNumber= :a2”*. We, then, need a map to connect the labels *“:a1”* and *“:a2”* with the new values, for this purpose we initialize an instance of the class Value map, we identify the type of data of every attribute and we use the methods *withNumber* and *withString* to add entries into the map.

Once our update expression is built we use the *updateItem* method of the class Table.

As it happened with the delete method, it is possible to add a condition to *updateItem*, that way the update will only happen if the condition is satisfied.

```
 Table table = dynamodb.getTable(tableName);
UpdateItemSpec updateItemSpec = new UpdateItemSpec();
JSONParser parser = new JSONParser();
JSONObject item = null;
item = (JSONObject)parser.parse(new FileReader(path));
String PK, SK;
if(item!=null) {
        PK = item.get("ID").toString();
        SK = item.get("Name").toString();
        int numKeys = 1;
        updateItemSpec.withPrimaryKey("ID",PK,"Name",SK);
        ++numKeys;
        String updateExpression = "set ";
        String attributeKey = ":a";
        Integer attributeIndex = 1;
        ValueMap map = new ValueMap();
        for(Object key : item.keySet()) {
            if(!key.toString().equals("ID")
            &&!key.toString().equals("Name")) {
                    updateExpression += key.toString()+" ="
                  +attributeKey+attributeIndex.toString();
                    if(attributeIndex<(item.keySet().size()-numKeys))
                        updateExpression+=", ";
                          if( item.get(key) instanceof String ) {
                                map.withString(attributeKey
                              +attributeIndex.toString(),
                              item.get(key).toString());
                          } else if ( item.get(key) instanceof Long ) {
                                map.withNumber(attributeKey
                              +attributeIndex.toString(),
                              (Long)item.get(key));
                          } else if ( item.get(key) instanceof Integer ){
                                 map.withNumber(attributeKey
                               +attributeIndex.toString(),
                               (Long)item.get(key));
                          } else if ( item.get(key) instanceof Double ) {
                                 map.withString(attributeKey
                              +attributeIndex.toString(),
                              item.get(key).toString());
                          }
                          ++attributeIndex;
                    }
            }
          updateItemSpec.withUpdateExpression(updateExpression).withValueMap(map)
          .withReturnValues(ReturnValue.UPDATED_NEW);
          UpdateItemOutcome outcome = table.updateItem(updateItemSpec);
    }
```

Code 7: Update of an item from a JSON file.

9.6. Query the database

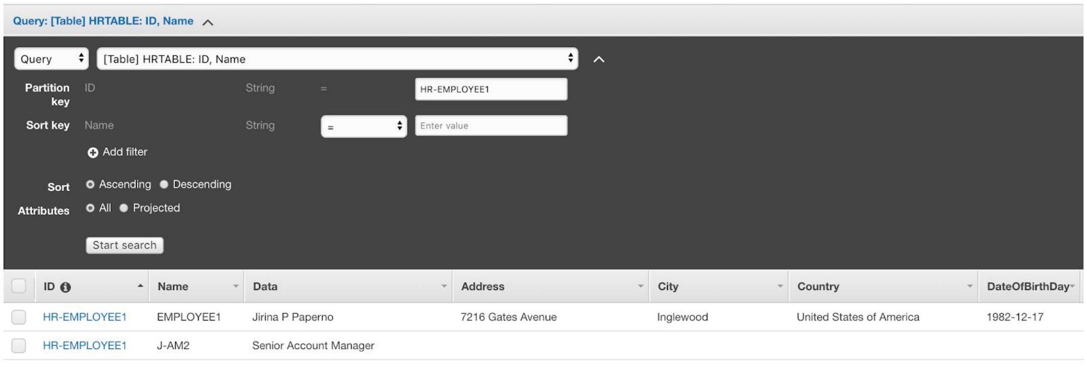
For the last section covering our case of study we will show explain the implementation of the five queries that we identified on section 10.1.

First query: Get employee details by employee ID

Our first query is the most straightforward of all five since we can find the employee by using the primary key of our table. The implementation can be found in code 8. We retrieve the table and then use the *query* method that takes as input the partition key of the item we try to retrieve. The query method returns a collection of items, we iterate through them and display them.[19]

```
Table table = dynamoDB.getTable( tableName );
ItemCollection<QueryOutcome> items = table.query(
        new KeyAttribute(CreateTable.PK, "HR-"+employeeId));
Iterator<Item> iter = items.iterator();
while (iter.hasNext()) {
        Item item = iter.next();
        System.out.println(item.toJSON());
}
```

Code 8: implementation of the first query: get employee by ID



[Table] HRTABLE: ID, Name

Partition key: ID String = HR-EMPLOYEE1

Sort key: Name String = Enter value

Sort: Ascending Descending

Attributes: All Projected

ID	Name	Data	Address	City	Country	DateOfBirthDay
HR-EMPLOYEE1	EMPLOYEE1	Jirina P Paperno	7216 Gates Avenue	Inglewood	United States of America	1982-12-17
HR-EMPLOYEE1	J-AM2	Senior Account Manager				

Figure 3: Result of first query on AWS console

Second query: Get employee details by employee name

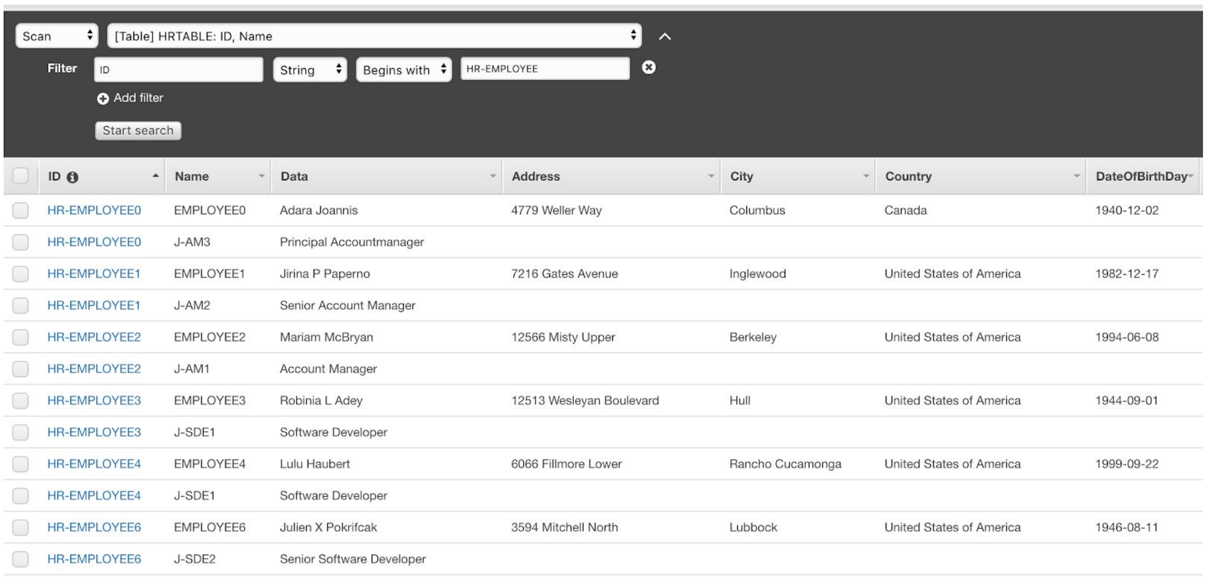
For the second *query* instead of using the query method that we have seen before we use the method *scan*. This is due to the fact that to use query we need to know the partition key of the item to be retrieved, in this case what we know is the Data field, which contains the employee name.

The *scan* method filters through all the items in the table to return only the ones that follow the conditions specified in the *ScanFilter*. As you can see in code 9, the filter condition in this case is for the primary key of the table to begin with "HR-EMPLOYEE" and for the Data attribute to be equal to the given employee name.

```
   Table table = dynamoDB.getTable( tableName );
   ItemCollection<ScanOutcome> items = table.scan(
   new ScanFilter( CreateTable.PK )
      .beginsWith("HR-EMPLOYEE"), new
ScanFilter("Data").eq(employeeName));
   Iterator<Item> iter = items.iterator();
   while (iter.hasNext()) {
      Item item = iter.next();
      System.out.println(item.toJSON());
   }
```

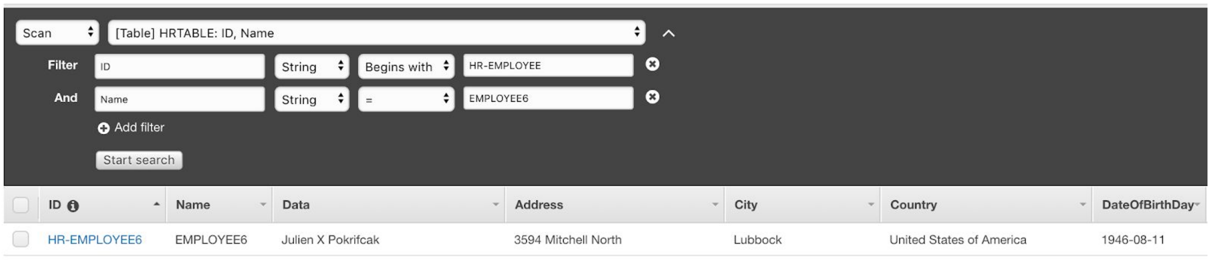
Code 9: implementation of the second query: Get employee details by employee name

Let's forget for a minute about the employee name asked for this scenario and focus only in a case where we would like to see all the employee details for all the existing employees within the database. Then the first scan filter mentioned above would be enough like shown in the Figure 4. The full process of the second query is achieved thanks to the second filter mentioned above an example of the results is shown in Figure 5.



ID	Name	Data	Address	City	Country	DateOfBirthDay
HR-EMPLOYEE0	EMPLOYEE0	Adara Joannis	4779 Weller Way	Columbus	Canada	1940-12-02
HR-EMPLOYEE0	J-AM3	Principal Accountmanager				
HR-EMPLOYEE1	EMPLOYEE1	Jirina P Paperno	7216 Gates Avenue	Inglewood	United States of America	1982-12-17
HR-EMPLOYEE1	J-AM2	Senior Account Manager				
HR-EMPLOYEE2	EMPLOYEE2	Mariam McBryan	12566 Misty Upper	Berkeley	United States of America	1994-06-08
HR-EMPLOYEE2	J-AM1	Account Manager				
HR-EMPLOYEE3	EMPLOYEE3	Robinia L Adey	12513 Wesleyan Boulevard	Hull	United States of America	1944-09-01
HR-EMPLOYEE3	J-SDE1	Software Developer				
HR-EMPLOYEE4	EMPLOYEE4	Lulu Haubert	6066 Fillmore Lower	Rancho Cucamonga	United States of America	1999-09-22
HR-EMPLOYEE4	J-SDE1	Software Developer				
HR-EMPLOYEE6	EMPLOYEE6	Julien X Pokrifcak	3594 Mitchell North	Lubbock	United States of America	1946-08-11
HR-EMPLOYEE6	J-SDE2	Senior Software Developer				

Figure 4: Intermediate step of the second query visualized on AWS console.


Scan | [Table] HRTABLE: ID, Name

Filter | ID | String | Begins with | HR-EMPLOYEE

And | Name | String | = | EMPLOYEE6

Add filter

Start search

ID	Name	Data	Address	City	Country	DateOfBirthDay
HR-EMPLOYEE6	EMPLOYEE6	Julien X Pokrifcak	3594 Mitchell North	Lubbock	United States of America	1946-08-11

Figure 5: Result of second query on AWS console.

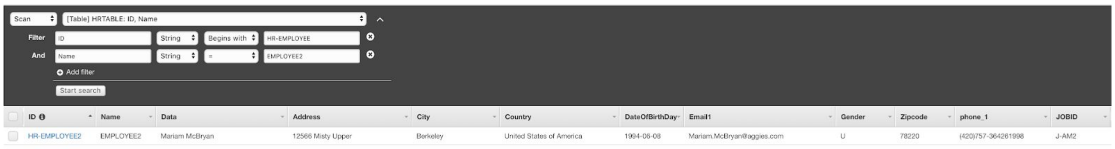
Third query: Get the current job details of an employee

For the third query let's assume we need all information regarding the current job of an employee and that among the employee details we can find the current jobID. We have implemented this query in two parts the first one, which can be found on code 10, retrieves the employee information to retrieve its current jobID and the second one, found in code 11, shows how we retrieve the job details once we have the jobID.

To find the job ID we query the table to find items that have *"HR-"+employeeID* as partition key and *employeeID* as sort key, once we have the result we can take the JobId by means of the method *get("JobID").*

```
    Table table = dynamoDB.getTable( tableName );
    QuerySpec spec = new QuerySpec()
          .withKeyConditionExpression("#id = :v_id and #name = :v_emp_id")
          .withNameMap( new NameMap().with("#id", CreateTable.PK)
          .with("#name", CreateTable.SK ) )
          .withValueMap(new ValueMap()
          .withString(":v_id", "HR-"+employeeId)
          .withString(":v_emp_id", employeeId) )
          .withConsistentRead(true);
    ItemCollection<QueryOutcome> items = table.query( spec );
    Iterator<Item> iter = items.iterator();
    while (iter.hasNext()) {
          Item item = iter.next();
          String jobId = (String) item.get("JobId");
          queryJobId( dynamoDB, tableName, employeeId, jobId );
 }
```

Code 10: Implementation of the first part of the third query, obtaining an employee's JobID



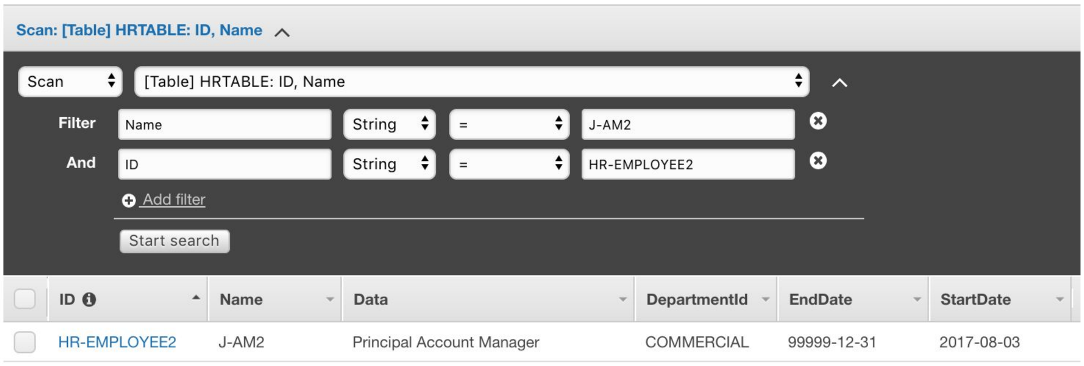
Filter	ID	String	Begins with	HR-EMPLOYEE
And	Name	String	=	EMPLOYEE2

ID	Name	Data	Address	City	Country	DateOfBirthDay	Email1	Gender	Zipcode	phone_1	JOBID
HR-EMPLOYEE2	EMPLOYEE2	Mariam McBryan	12566 Misty Upper	Berkeley	United States of America	1994-06-08	Mariam.McBryan@aggies.com	U	78220	(420)757-364261998	J-AM2

Figure 6: Result of first step for third query on AWS console.

```
    public static void queryJobId( DynamoDB dynamoDB, String tableName,
    String employeeId, String jobId ) {
        Table table = dynamoDB.getTable( tableName );
        QuerySpec spec = new QuerySpec()
          .withKeyConditionExpression("#id = :v_id and #name =
:v_job_id")
          .withNameMap( new NameMap().with("#id", CreateTable.PK)
          .with("#name", CreateTable.SK ) )
          .withValueMap(new ValueMap()
                   .withString(":v_id","HR-"+employeeId)
                  .withString(":v_job_id", jobId) )
          .withFilterExpression("#endDate = 9999-12-31")
          .withValueMap(new ValueMap().with("endDate_", "EndDate"));
        ItemCollection<QueryOutcome> items = table.query(spec);
         Iterator<Item> iter = items.iterator();
         while (iter.hasNext()) {
            Item item = iter.next();
            System.out.println(item.toJSON());
          }
    }
```

Code 11: Implementation of the second part of the third query: getting job details by jobID



Filter	Name	String	=	J-AM2
And	ID	String	=	HR-EMPLOYEE2

ID	Name	Data	DepartmentId	EndDate	StartDate
HR-EMPLOYEE2	J-AM2	Principal Account Manager	COMMERCIAL	99999-12-31	2017-08-03

Figure 7: Result of second step for third query on AWS console.

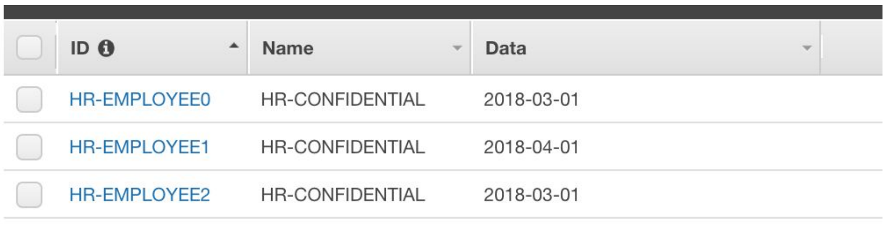
<u>Fourth query: Get details of all employees hired recently</u>

For the fourth query instead of the table, we query the GSI since it has the hire date as sort key for items containing confidential details about an employee. The implementation starts by retrieving the table and, from it, the GSI. Then we create an instance of the class QuerySpec and we add to it the necessary attributes for the query. We know that all confidential information items have as their partition key in the GSI “HR-CONFIDENTIAL”, therefore, we want to retrieve all items with that partition key whose hire date is posterior to the given date. This is indicated in the condition expression for de query. Since we know the partition key of the items we use the method *query* to execute our query.

```
    Table table = dynamoDB.getTable( tableName );
    Index index = table.getIndex(CreateTable.GSI_1_NAME);
    QuerySpec spec = new QuerySpec()
       .withKeyConditionExpression("#name = :v_id and #data >
           :v_hired_date")
       .withNameMap( new NameMap().with("#name", CreateTable.SK)

       .with("#data", CreateTable.GSI_1_SK ) )
       .withValueMap(new ValueMap().withString(":v_id",
"HR-CONFIDENTIAL")
       .withLong(":v_hired_date", date.getTime() ) )
       .withConsistentRead(true);
    ItemCollection<QueryOutcome> items = index.query(spec);
    Iterator<Item> iter = items.iterator();
    while (iter.hasNext()) {
       Item item = iter.next();
       System.out.println(item.toJSON());
    }
```

Code 12: implementation of the fourth query: Get details of all employees hired recently



ID	Name	Data
HR-EMPLOYEE0	HR-CONFIDENTIAL	2018-03-01
HR-EMPLOYEE1	HR-CONFIDENTIAL	2018-04-01
HR-EMPLOYEE2	HR-CONFIDENTIAL	2018-03-01

Figure 8: Result of fourth step for third query on AWS console.

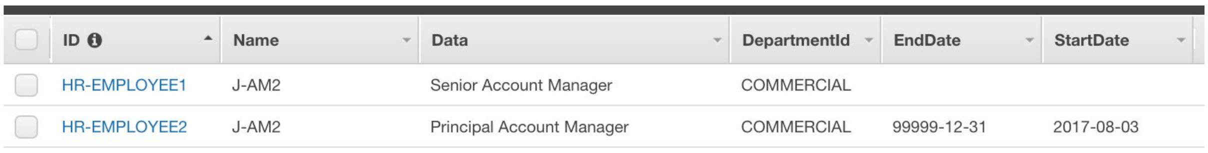
<u>Fifth query: Get all employees who have had a specific job</u>

We run the last query over the GSI instead of the table. We know that the partition key for job items is the job ID so we only need to retrieve all items whose partition key in GSI is the job ID we are given. We do so by once again building a condition expression, in this case *"#name = :v_jobID"* where *"#name"* is *"Name"* in the GSI and *"v_jobID"* is the given ID.

Once more, since we know the partition key for the job we can use the *query* method. The full implementation can be found on code 13.

```
Table table = dynamoDB.getTable( tableName );
Index index = table.getIndex(CreateTable.GSI_1_NAME);
QuerySpec spec = new QuerySpec()
      .withKeyConditionExpression("#name = :v_jobID ")
      .withNameMap( new NameMap().with("#name", CreateTable.SK) )
      .withValueMap(new ValueMap().withString(":v_jobId", jobID ))
      .withConsistentRead(true);
ItemCollection<QueryOutcome> items = index.query(spec);
Iterator<Item> iter = items.iterator();
while (iter.hasNext()) {
      Item item = iter.next();
      System.out.println(item.toJSON());
}
```

Code 13: Implementation of the fifth query: Get all employees who have had a specific job



	ID	Name	Data	DepartmentId	EndDate	StartDate
	HR-EMPLOYEE1	J-AM2	Senior Account Manager	COMMERCIAL		
	HR-EMPLOYEE2	J-AM2	Principal Account Manager	COMMERCIAL	99999-12-31	2017-08-03

Figure 8: Result of fifth step for third query on AWS console.

10. References

[1] Oszu Tamer M, Valduriez Patrick. *Principles of Distributed Database System*. Springer
[2] Lake, P. Crowther, P.(2013). *Concise Guide to Databases*. Springer
[3] Amazon. DynamoDB. Partitions and Data Distribution. [https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/HowItWorks.Partitions.html] Accessed Nov 30, 2018.
[4] Ryan Kroonenburg. AWS Certified Developer - Associate 2018 [https://www.udemy.com/share/1000vABUoceVdaRHw=/] Accessed Nov 30, 2018
[5] IBM. Data integration [https://www.ibm.com/analytics/data-integration] Accessed Dec 1, 2018.
[6] Timeline 50 years of Hard Drives: [https://www.pcworld.com/article/127105/article.html] Accessed Dec 1, 2018.
[7] Stonebraker S. SQL Databases v. NpSQL Databases [https://www.cs.cmu.edu/~pavlo/courses/fall2013/static/papers/stonebraker-sql-2010.pdf] Accessed Dec 1, 2018.
[8] Example of modeling relational data on DynamoDB [https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/bp-modeling-nosql-B.html] Accessed Dec 2, 2018.
[9] NoSQL Design for DynamoDB [https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/bp-general-nosql-design.html] Accessed Dec 2, 2018.
[10] Choosing the right DynamoDB partition key [https://aws.amazon.com/blogs/database/choosing-the-right-dynamodb-partition-key/] Accessed Dec 2, 2018.
[11] Best Practices for Using Sort Keys to Organize Data [https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/bp-sort-keys.html]Accessed Dec 2, 2018.
[12] Improving Data Access with Secondary Indexes [https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/SecondaryIndexes.html] Accessed Dec 3, 2018.
[13] Global Secondary Indexes [https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/GSI.html] Accessed Dec 3, 2018.
[14] Step:1 Create a table [https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/GettingStarted.Java.01.html] Accessed Dec 3, 2018.
[15] Create, Update, and Delete Global Secondary Indexes Using the Amazon DynamoDB Document API [https://aws.amazon.com/blogs/developer/create-update-and-delete-global-secondary-indexes-using-the-amazon-dynamodb-document-api/] Accessed Dec 3, 2018.
[16] Step 2: Load sample data [https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/GettingStarted.Java.02.html] Accessed Dec 4, 2018.

[17] Step 3: Create, Read, Update, and Delete an item
[https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/GettingStarted.Java.03.html] Accessed Dec 4, 2018.
[18] BatchWriteItem
[https://docs.aws.amazon.com/amazondynamodb/latest/APIReference/API_BatchWriteItem.html] Accessed Dec 4, 2018.

[19] Step 4: Query and Scan the data
[https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/GettingStarted.Java.04.html] Accessed Dec 5, 2018.