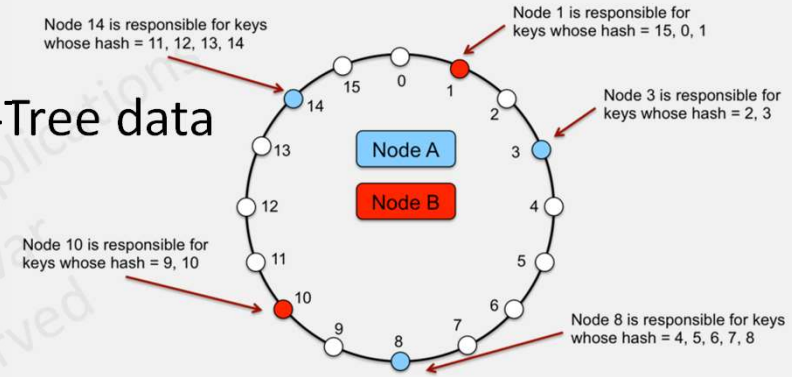
CLOUD COMPUTING APPLICATIONS

Serverless Storage: DynamoDB

Prof. Reza Farivar

DynamoDB

- DynamoDB is a fully managed NoSQL database provided by Amazon AWS
- Think of it as a massive distributed B-Tree data structure in the cloud
  - Accessing specific items is blazingly fast
- Distributed system
  - Using the consistent hashing algorithm in a ring



Usage model

- First create a table
  - Remember that it's a managed service, so just create a table using the console (or CLI, API)
    - We will use the Python Boto3 package in this lesson
- While creating the table, define the primary key
  - This key will be used by DynamoDB to distribute key/values in different partitions
- Optionally, identify a sort key
  - The sort key is used to keep the items in a partition sorted
  - Will be useful for query and scan later on

Using the table: Put

- Having defined the table, we can now put values into it.
  - Note: DynamoDB items are limited to 400KB size

```
import boto3
dynamodb = boto3.resource('dynamodb')
table = dynamodb.Table('users')

table.put_item(
      Item={
            'username': 'janedoe',
            'first_name': 'Jane',
            'last_name': 'Doe',
            'age': 25,
            'account_type': 'standard_user',
      }
)
```

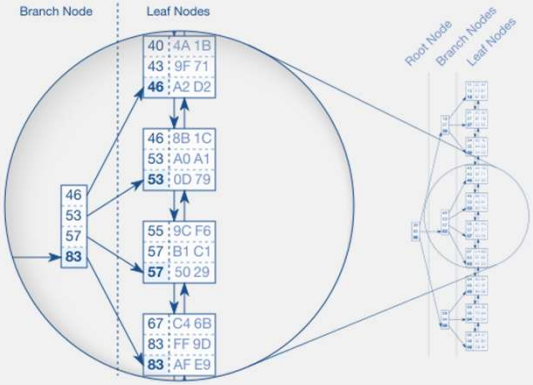
Using the table: Get

- Retrieve an item

```
response = table.get_item(
    Key={
        'username': 'janedoe',
        'last_name': 'Doe'
    }
)
item = response['Item']
print(item)



#Expected output:

{u'username': u'janedoe',
 u'first_name': u'Jane',
 u'last_name': u'Doe',
 u'account_type': u'standard_user',
 u'age': Decimal('25')}
```

Query and Scanning

- In RDBMS world*, query is usually defined as an operation where there is a usable index available, and we can quickly retrieve the item in Log(n) time
- In comparison, scan happens where there is no usable index, and the engine has to read every record and test for a condition
- An RDBMS engine parses a SELECT statement, and performs query optimization, all behind the curtain
- DynamoDB just allows you to be your own database engine!



*See https://use-the-index-luke.com/sql/where-clause/searching-for-ranges/greater-less-between-tuning-sql-access-filter-predicates

Query

- Query only works on the primary key already defined for the table
- Or any other attribute that we have explicitly made a secondary index for it
- If a composite primary key was used (hash key + sort key), we can ask query to return a conditional range of value

```
response = table.query(
    KeyConditionExpression=Key('username').eq('johndoe')
)
items = response['Items']
print(items)


#Expected output:

[{u'username': u'johndoe',
  u'first_name': u'John',
  u'last_name': u'Doe',
  u'account_type': u'standard_user',
  u'age': Decimal('25'),
  u'address': {u'city': u'Los Angeles',
               u'state': u'CA',
               u'zipcode': Decimal('90001'),
               u'road': u'1 Jefferson Street'}}]
```

Scan

- What if we want to perform a query conditioned on attributes that there is no index for them?
- Scan will return everything!
  - It allows to filter based on any arbitrary condition

```
response = table.scan(
    FilterExpression=Attr('age').lt(27)
)
items = response['Items']
print(items)
```

Secondary Index

- Similar to the main index, requires a partition key and a sort key
- Local (LSI)
  - First released by Amazon in 2013
  - Immediately consistent
  - Once created, the table cannot grow any more
    - All the records tha share the same partition key need to fit in 10GB
    - Once the allocation is full, writes fail ☹
- Global (GSI)
  - Came a few months after local indexes
    - Eventual consistency model
    - Do not constrain table size ☺