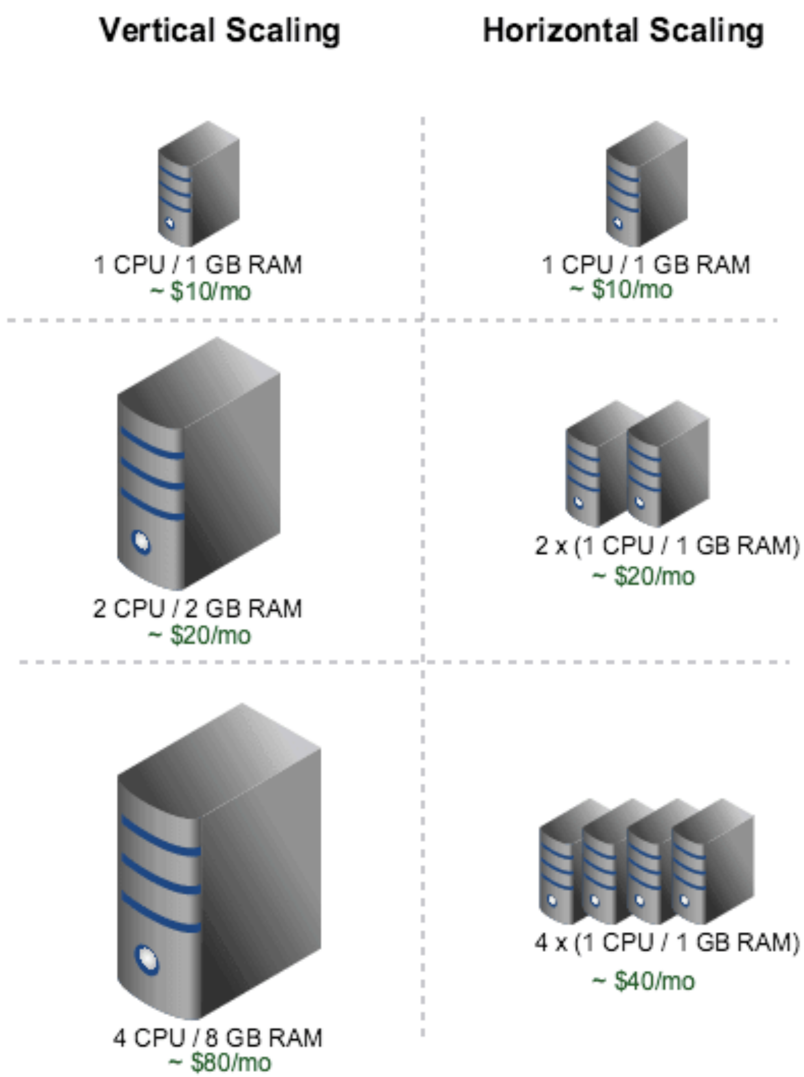
NoSQL Databases and DynamoDB

Based on material from;

http://boto3.readthedocs.io/en/latest/reference/services/dynamodb.html

http://docs.aws.amazon.com/amazondynamodb/latest/gettingstartedguide/GettingStarted.Python.html

Scalability



Relational Databases

- **Strengths**
  - ACID properties
  - Strong consistency, concurrency, recovery
  - Normalization
  - Standard Query language (SQL)
  - Vertical scaling (up scaling)

- **Weaknesses**
  - Not designed to run over wide area network (georeplication)
  - Joins are expensive
  - Transactions are slow
  - Hard to scale horizontally

NoSQL

- **No strict definition for NoSQL databases.**
  - Initially, these systems did not support SQL, but provided a simpler GET/PUT interface
  - Invariably, as system matures, it tends to provide an SQL-like query language

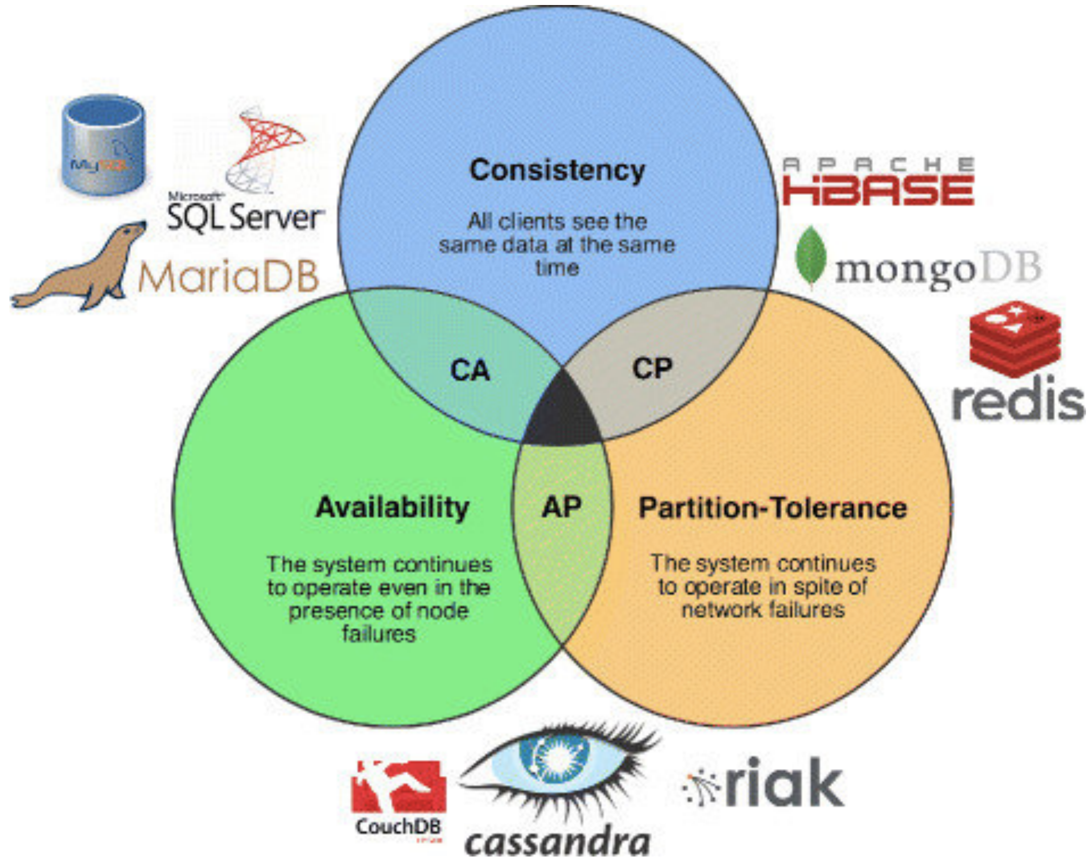
- **It is a nonrelational database.**

- **Designed to use for Big Data and Real time web applications.**

- **Key idea:**
  - Relax ACID and consistency
  - Avoid complexity of full SQL
  - Increase horizontal scalability

CAP Theorem

- **impossible for a distributed data store to simultaneously provide more than two out of the following three guarantees**
  - Consistency: Every read receives the most recent write or an error
  - Availability: Every request receives a (non-error) response, without the guarantee that it contains the most recent write
  - Partition tolerance: The system continues to operate despite an arbitrary number of messages being dropped (or delayed) by the network between nodes



NoSQL Types

Type	Example	
Key-Value Store	redis	riak
Wide Column Store	H·BASE	cassandra
Document Store	mongoDB	CouchDB relax
Graph Store	Neo4j	InfiniteGraph The Distributed Graph Database

Source: https://www.algoworks.com/blog/nosql-database/

Key Value Pair Based

- Data model: (key, value) pairs
- Dictionary
- Collection of records having fields containing data.
- Stored and retrieved using a key that uniquely identifies the record
- Example: Oracle NoSQL Database, Riak.

Column Based

- It store data as Column families containing rows that have many columns associated with a row key.
- Each row can have different columns.
- Column families are groups of related data that is accessed together.
- Example:Cassandra, HBase, Hypertable, Amazon DynamoDB.

DynamoDB

- **Fully managed NoSQL database service**
  - More similar to a key-value store than a relational database

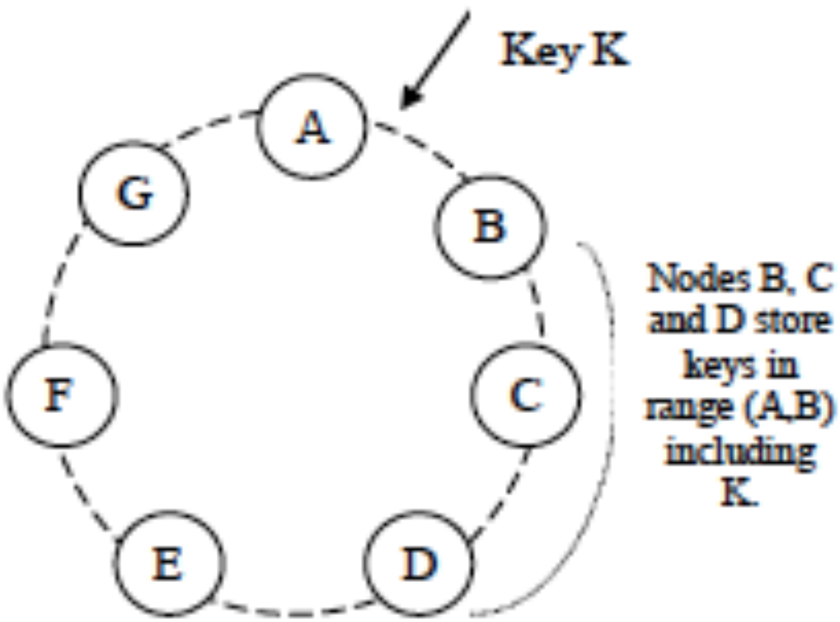
- **Provides:**
  - 1) seamless scalability
    - Store data on a cluster of computers
    - Data is replicated for performance and fault tolerance
      - Eventual consistency (Default)
      - Strong consistency
  - 2) fast and predictable performance
    - Reads can be answered by a single node
    - All data for an object stored together (no joins)

Key Principles

- Incremental scalability
- Decentralization
- O(1) routing
- Simple API:
  - get(key)
  - put(key, context, object)

Data Partitioning

- Consistent hashing
- Each node has (multiple) position on the ring
- Data stored on node based on a key hash



Replication

- Data replicated to N hosts
- Node handling a request = coordinator
- Coordinator replicates keys to N-1 successors

Consistency

- Eventual consistency model
  - If no new updates are made to a given data item, eventually all accesses to that item will return the last updated value.
- Quorum-based consistency protocol
  - Min. no. of replicas needed for read (R)/write (W)
  - Coordinator waits for R/W responses before replying to the client

Concepts

- **Tables**
  - A *table* is a collection of data.
- **Items**
  - An *item* is a group of attributes that is uniquely identifiable among all of the other items.
  - Each table contains multiple items.
  - In many ways, items are similar to rows, records, or tuples in relational database systems.
- **Attributes**
  - Fundamental data element, something that does not need to be broken down any further.
    - Similar fields or columns in other database management systems.
  - Each item is composed of one or more attributes.
- **Primary Key**
  - Uniquely identifies each item in the table

People

```
{
    "PersonID": 101,
    "LastName": "Smith",
    "FirstName": "Fred",
    "Phone": "555-4321"
}
```

```
{
    "PersonID": 102,
    "LastName": "Jones",
    "FirstName": "Mary",
    "Address": {
        "Street": "123 Main",
        "City": "Anytown",
        "State": "OH",
        "ZIPCode": 12345
    }
}
```

```
{
    "PersonID": 103,
    "LastName": "Stephens",
    "FirstName": "Howard",
    "Address": {
        "Street": "123 Main",
        "City": "London",
        "PostalCode": "ER3 5K8"
    },
    "FavoriteColor": "Blue"
}
```

Data Types

- Scalar Types
  - Represent exactly one value.
  - number, string, binary, Boolean, and null
- Document Types
  - Represent a complex structure with nested attributes
  - list and map.
- Set Types
  - Represent multiple scalar values
  - string set, number set, and binary set.

Primary Key Options

- **Partition key**
  - A simple primary key, composed of one attribute known as the *partition key*.
  - Hash of partition key determines the partition where the item is stored.
- **Partition key and sort key**
  - A composite primary key, composed of two attributes.
  - The first attribute is the *partition key*, and the second attribute is the *sort key*.
  - Hash of partition key determines the partition where the item is stored.
  - All items with the same partition key are stored together, in sorted order by sort key value.

Dynamo Example

Name	Course	Grade
Rosa	Comp101	A
Rosa	Eng101	B
Rosa	Math101	A+
Jane	Chem101	B
Jane	Math101	A
Jake	French101	A

Partition Key

Sort Key

Dynamo Example

Partition1: A-M

Name	Course	Grade
Jane	Chem101	B
Jane	Math101	A
Jake	French101	A

Partition2: N-Z

Name	Course	Grade
Rosa	Comp101	A
Rosa	Eng101	B
Rosa	Math101	A+

- **Efficient query:** **All course taken by Jake**
- **Inefficient query:** **All students that took Math101**
  All students that got an A

Secondary Indexes

- **Most reads use primary key attribute values.**
- **Provide efficient access to data with attributes other than the primary key.**
- **Associated with exactly one table, from which it obtains its data.**
- **Define an alternate key for the index (partition key and sort key).**
- **Define the attributes that you want to be projected, or copied, from the base table into the index.**
- **Can query or scan the index just as you would a table.**
- **Secondary index automatically maintained by DynamoDB.**
  - When you add, modify, or delete items in the base table, any indexes on that table are also updated to reflect these changes.

:

Secondary Index Types

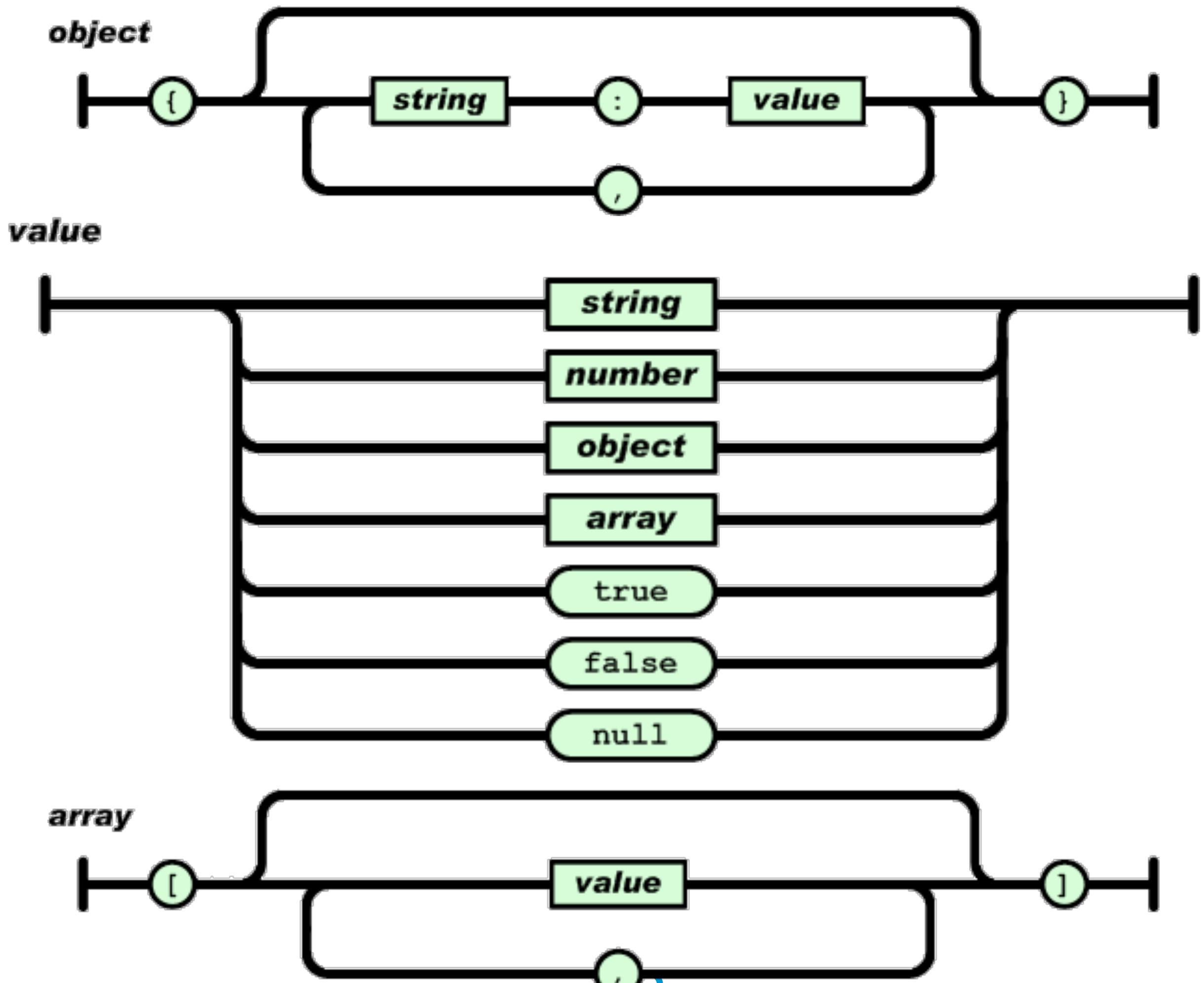
- **Global secondary index**
  - Partition key and a sort key can be different from those on the base table.
  - Queries on the index can span all of the data in the base table, across all partitions.
- **Local secondary index**
  - Same partition key as the base table, but a different sort key.

JSON

- JavaScript Object Notation
- Lightweight data-interchange format.
  - Text format
  - Language independent
- Easy for humans to read and write.
- Easy for machines to parse and generate.
- Based on JavaScript

JSON Structures

- A collection of name/value pairs.



JSON Example

```
{
    "id": 1,
    "name": "A green door",
    "price": 12.50,
    "tags": ["home", "green"]
}
```

```
{
    Day: "Monday",
    UnreadEmails: 42,
    ItemsOnMyDesk: [
        "Coffee Cup",
        "Telephone",
        {
            Pens: { Quantity : 3},
            Pencils: { Quantity : 2},
            Erasers: { Quantity : 1}
        }
    ]
}
```

Local Setup

- **Download from:**
  - http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/DynamoDBLocal.html

- **Run:**
  - java -Djava.library.path=./DynamoDBLocal_lib -jar DynamoDBLocal.jar -sharedDb

```
bash-4.0$ java -Djava.library.path=./DynamoDBLocal_lib -jar DynamoDBLocal.jar
Initializing DynamoDB Local with the following configuration:
Port:   8000
InMemory:       false
DbPath: null
SharedDb:       false
shouldDelayTransientStatuses:   false
CorsParams:     *
```

Command Line Interface

```
bash-4.0$ aws dynamodb create-table \
>     --table-name Music \
>     --attribute-definitions \
>         AttributeName=Artist,AttributeType=S \
>         AttributeName=SongTitle,AttributeType=S \
>     --key-schema AttributeName=Artist,KeyType=HASH AttributeName=SongTitle,KeyType=RANGE \
>     --provisioned-throughput ReadCapacityUnits=1,WriteCapacityUnits=1 \
>     --endpoint-url http://localhost:8000
```

```
bash-4.0$ aws dynamodb list-tables --endpoint-url http://localhost:8000
{
    "TableNames": [
        "Music"
    ]
}
```

```
bash-4.0$  aws dynamodb put-item --table-name Music \
> --item '{"Artist":{"S":"ACME Band"}, "SongTitle": {"S":"Happy Day"}, "AlbumTitle": {"S": "About Life"} }' \
> --endpoint-url http://localhost:8000
```

```
bash-4.0$ aws dynamodb scan --table-name Music --endpoint-url http://localhost:8000
{
    "ScannedCount": 1,
    "ConsumedCapacity": null,
    "Items": [
        {
            "SongTitle": {
                "S": "Happy Day"
            },
            "Artist": {
                "S": "ACME Band"
            },
            "AlbumTitle": {
                "S": "About Life"
            }
```

AWS SDK

- **Use boto3**

- **Connect to dynamodb**
  - *dynamodb = boto3.resource('dynamodb', region_name='us-east-1', endpoint_url="http://localhost:8000")*

  - *dynamodb = boto3.resource('dynamodb', region_name='us-east-1’)*

Create Table

```
table = dynamodb.create_table(
        AttributeDefinitions=[
                {
                'AttributeName': 'string',
                 'AttributeType': 'S'|'N'|'B'
                },
        ],
        TableName='string',
        KeySchema=[
                {
                'AttributeName': 'string',
                'KeyType': 'HASH'|'RANGE'
                },
        ],
        ProvisionedThroughput={
                'ReadCapacityUnits': 123,
                'WriteCapacityUnits': 123
        }
)
```

Put

```
response = table.put_item(
    Item={
        'string': 'string'|123|Binary(b'bytes')|True|None|set(['string'])|set([123])|set([Binary(b'bytes')])|[]|{}
    },
    Expected={
        'string': {
            'Value': 'string'|123|Binary(b'bytes')|True|None|set(['string'])|set([123])|set([Binary(b'bytes')])|[]|{},
            'Exists': True|False,
            'ComparisonOperator': 'EQ'|'NE'|'IN'|'LE'|'LT'|'GE'|'GT'|'BETWEEN'|'NOT_NULL'|'NULL'|'CONTAINS'|'NOT_CONTAINS'|'BEGINS_W
            'AttributeValueList': [
                'string'|123|Binary(b'bytes')|True|None|set(['string'])|set([123])|set([Binary(b'bytes')])|[]|{},
            ]
        }
    },
    ReturnValues='NONE'|'ALL_OLD'|'UPDATED_OLD'|'ALL_NEW'|'UPDATED_NEW',
    ReturnConsumedCapacity='INDEXES'|'TOTAL'|'NONE',
    ReturnItemCollectionMetrics='SIZE'|'NONE',
    ConditionalOperator='AND'|'OR',
    ConditionExpression=Attr('myattribute').eq('myvalue'),
    ExpressionAttributeNames={
        'string': 'string'
    },
    ExpressionAttributeValues={
        'string': 'string'|123|Binary(b'bytes')|True|None|set(['string'])|set([123])|set([Binary(b'bytes')])|[]|{}
    }
)
```

Get

```
response = table.get_item(
    Key={
        'string': 'string'|123|Binary(b'bytes')|True|None|set(['string'])|set([123])|set([Binary(b'bytes')])|[]|{}
    },
    AttributesToGet=[
        'string',
    ],
    ConsistentRead=True|False,
    ReturnConsumedCapacity='INDEXES'|'TOTAL'|'NONE',
    ProjectionExpression='string',
    ExpressionAttributeNames={
        'string': 'string'
    }
)
```

Get Response Syntax

```
{
    'Item': {
        'string': 'string'|123|Binary(b'bytes')|True|None|set(['string'])|set([123])|set([Binary(b'bytes')
    },
    'ConsumedCapacity': {
        'TableName': 'string',
        'CapacityUnits': 123.0,
        'Table': {
            'CapacityUnits': 123.0
        },
        'LocalSecondaryIndexes': {
            'string': {
                'CapacityUnits': 123.0
            }
        },
        'GlobalSecondaryIndexes': {
            'string': {
                'CapacityUnits': 123.0
            }
        }
    }
}
```

Query

```
response = table.query(
    IndexName='string',
    Select='ALL_ATTRIBUTES'|'ALL_PROJECTED_ATTRIBUTES'|'SPECIFIC_ATTRIBUTES'|'COUNT',
    AttributesToGet=[
        'string',
    ],
    Limit=123,
    ConsistentRead=True|False,
    KeyConditions={
        'string': {
            'AttributeValueList': [
                'string'|123|Binary(b'bytes')|True|None|set(['string'])|set([123])|set([Binary(b'bytes')])|[]|
            ],
            'ComparisonOperator': 'EQ'|'NE'|'IN'|'LE'|'LT'|'GE'|'GT'|'BETWEEN'|'NOT_NULL'|'NULL'|'CONTAINS'|'N
        }
    },
    QueryFilter={
        'string': {
            'AttributeValueList': [
                'string'|123|Binary(b'bytes')|True|None|set(['string'])|set([123])|set([Binary(b'bytes')])|[]|
            ],
            'ComparisonOperator': 'EQ'|'NE'|'IN'|'LE'|'LT'|'GE'|'GT'|'BETWEEN'|'NOT_NULL'|'NULL'|'CONTAINS'|'N
        }
    },
    ConditionalOperator='AND'|'OR',
    ScanIndexForward=True|False,
    ExclusiveStartKey={
        'string': 'string'|123|Binary(b'bytes')|True|None|set(['string'])|set([123])|set([Binary(b'bytes')])|[
    },
    ReturnConsumedCapacity='INDEXES'|'TOTAL'|'NONE',
    ProjectionExpression='string',
    FilterExpression=Attr('myattribute').eq('myvalue'),
    KeyConditionExpression=Key('mykey').eq('myvalue'),
    ExpressionAttributeNames={
        'string': 'string'
    },
    ExpressionAttributeValues={
        'string': 'string'|123|Binary(b'bytes')|True|None|set(['string'])|set([123])|set([Binary(b'bytes')])|[
    }
)
```

Query Response

```
{
    'Items': [
        {
            'string': 'string'|123|Binary(b'bytes')|True|None|set(['string'])|set([123])|set([Binary(b'bytes
        },
    ],
    'Count': 123,
    'ScannedCount': 123,
    'LastEvaluatedKey': {
        'string': 'string'|123|Binary(b'bytes')|True|None|set(['string'])|set([123])|set([Binary(b'bytes')])
    },
    'ConsumedCapacity': {
        'TableName': 'string',
        'CapacityUnits': 123.0,
        'Table': {
            'CapacityUnits': 123.0
        },
        'LocalSecondaryIndexes': {
            'string': {
                'CapacityUnits': 123.0
            }
        },
        'GlobalSecondaryIndexes': {
            'string': {
                'CapacityUnits': 123.0
            }
        }
    }
}
```

Scan

- Accessing every item in a table or a secondary index.
- Can provide a FilterExpression operation.
- If the total number of scanned items exceeds the maximum data set size limit of 1 MB, the scan stops and results are returned to the user as a LastEvaluatedKey value to continue the scan in a subsequent operation. The results also include the number of items exceeding the limit.

Scan

```
response = table.scan(
    IndexName='string',
    AttributesToGet=[
        'string',
    ],
    Limit=123,
    Select='ALL_ATTRIBUTES'|'ALL_PROJECTED_ATTRIBUTES'|'SPECIFIC_ATTRIBUTES'|'COUNT',
    ScanFilter={
        'string': {
            'AttributeValueList': [
                'string'|123|Binary(b'bytes')|True|None|set(['string'])|set([123])|set([Binary(b'bytes')])|[]|
            ],
            'ComparisonOperator': 'EQ'|'NE'|'IN'|'LE'|'LT'|'GE'|'GT'|'BETWEEN'|'NOT_NULL'|'NULL'|'CONTAINS'|'N
        }
    },
    ConditionalOperator='AND'|'OR',
    ExclusiveStartKey={
        'string': 'string'|123|Binary(b'bytes')|True|None|set(['string'])|set([123])|set([Binary(b'bytes')])|[
    },
    ReturnConsumedCapacity='INDEXES'|'TOTAL'|'NONE',
    TotalSegments=123,
    Segment=123,
    ProjectionExpression='string',
    FilterExpression=Attr('myattribute').eq('myvalue'),
    ExpressionAttributeNames={
        'string': 'string'
    },
    ExpressionAttributeValues={
        'string': 'string'|123|Binary(b'bytes')|True|None|set(['string'])|set([123])|set([Binary(b'bytes')])|[
    },
    ConsistentRead=True|False
)
```

Scan


```
{
    'Items': [
        {
            'string': 'string'|123|Binary(b'bytes')|True|None|set(['string'])|set([123])|set([Binary(b'byt
        },
    ],
    'Count': 123,
    'ScannedCount': 123,
    'LastEvaluatedKey': {
        'string': 'string'|123|Binary(b'bytes')|True|None|set(['string'])|set([123])|set([Binary(b'bytes')
    },
    'ConsumedCapacity': {
        'TableName': 'string',
        'CapacityUnits': 123.0,
        'Table': {
            'CapacityUnits': 123.0
        },
        'LocalSecondaryIndexes': {
            'string': {
                'CapacityUnits': 123.0
            }
        },
        'GlobalSecondaryIndexes': {
            'string': {
                'CapacityUnits': 123.0
            }
        }
    }
}
```

Zappa

- Deploy Python WSGI applications on AWS Lambda + API Gateway + DynamoDB.
- https://github.com/Miserlou/Zappa

- Steps:
  - Install and configure AWS CLI
  - Create flask directory structure
  - Create a python virtual environment:

    > python3.6 -m venv venv

    > source venv/bin/activate
  - Install flask and boto3
    - pip install flask
    - pip install boto3
  - Install zappa

    > pip install zappa

    > zappa init

Zappa

- zappa_settings.json

```
{
    "dev": {
        "project_name": "dynamo_lecture",
        "keep_warm": false,
        "debug": true,
        "log_level": "DEBUG",
        "s3_bucket": "ece1779fall2017",
        "app_function": "app.webapp",
        "http_methods": ["GET","POST"],
        "parameter_depth": 1,
        "timeout_seconds": 300,
        "memory_size": 128,
        "use_precompiled_packages": true
    }
}
```

- run.py (for local testing only)

```
in/python
 import webapp

__ == "__main__":
p.run()
```

- Deploy application to AWS Lambda

  > zappa deploy dev