Supported Operations

Table Operations:

- Allowed operations: Create, Update, List, Delete Table.
- UpdateTable: Increase/decrease table's provisioned throughput
- DescribeTable: Retrieve table info like current status, PKey, creation time ..etc
- ListTable: Get a list of tables in your account.

Item Operations:

- Allowed operations: Add, Update, Delete items from a table.
- UpdateItem: Update existing attributes, add new attribute,delete existing attribute from an Item.
- Conditional Update: Set a condition, so that update will only happen if that condition is met.
- PutItem: Write item.
- GetItem: Retrieve single item.
- BatchGetItem: Retrieve multiple items or items from multiple table.

Query and Scan Operations:

- **Query**: Enables you to query a table using the hash attribute and an optional range filter. If the table has a secondary index, you can also Query the index using its key.You can query only tables whose primary key is of hash-and-range type; you can also query any secondary index on such tables.
- **Scan:** Can be used on a table or a secondary index. The Scan operation reads every item in the table or secondary index. For large tables and secondary indexes, a Scan can consume a large amount of resources; for this reason, we recommend that you design your applications so that you can use the Query operation mostly, and use Scan only where appropriate.

Update - Conditional Updates:
Suppose two clients read the same item. Both clients get a copy of that item from DynamoDB. Client 1 then sends a request to update the item. Client 2 is not aware of any update. Later, Client 2 sends its own request to update the item, overwriting the update made by Client 1. Thus, the update made by Client 1 is lost. DynamoDB supports a "conditional write" feature that lets you specify a condition when updating an item. DynamoDB writes the item only if the specified condition is met; otherwise it returns an error. In the "lost update" example, client 2 can add a condition to verify item values on the server-side are same as the item copy on the client-side. If the item on the server is updated, client 2 can choose to get an updated copy before applying its own updates.

Update - Atomic Counter:
DynamoDB also supports an "atomic counter" feature where you can send a request to add or subtract from an existing attribute value without interfering with another simultaneous write request. For example, a web application might want to maintain a counter per visitor to its site. DynamoDB write operations support incrementing or decrementing existing attribute values.

Create Table

Notes:

- Go through the Best practises page:
http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/BestPractices.html

- Give table name a Prefix for better management. for example: Fin_Accounts, Fin_BankList, Ops_Student, Fwk_Metadata ..etc

- When storing data, DynamoDB divides a table's items into multiple partitions, and distributes the data primarily based on the hash key element.The provisioned throughput associated with a table is also divided evenly among the partitions, with no sharing of provisioned throughput across partitions. To get the most out of DynamoDB throughput (and avoiding HotSpots), **build tables where the hash key element has a large number of distinct values, and values are requested fairly uniformly,** as randomly as possible. (Page# 62: Dev guide)

- The *CreateTable* operation adds a new table to your account. In an AWS account, table names must be unique within each region. That is, you can have two tables with same name if you create the tables in different regions.

- Optimization techniques: Randomizing across Multiple hash Key values, Using a Calculated key Value * *, Partition behavior, Use of Burst capacity, Distribute Write activity during data upload, cache popular item, Consider WorkLoad uniformity when adjusting Provisioned Throughput,
http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/GuidelinesForTables.html
Details on ‘Working with Tables’: :
http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/WorkingWithTables.html

Naming convention:

The table name can include characters a-z, A-Z, 0-9, '_' (underscore), '-' (dash), and '.' (dot). Names can be between 3 and 255 characters long. For having a useful/better naming convention, here are the things to remember:

- There has to be Uniformity i.e. follow it for all Databases.

- Schema/Database name should not be included in Table names. For example: If the schema is: Hir, then table names should not have a prefix like: Hir_tableName

- Table name may have (if required) a prefix that depicts it’s functional behavior. For example:
log_table_name = hold logging data
temp_table_name = hold temp data which is volatile
metadata_table_name = hold metadata info

- It’s better to use either “_” (underscore) OR “-”(dash) as a separator. Mixing both is not usually preferred.

- Avoid ambiguous abbreviations. For example:
md_table_name = should use: metadata_table_name
log_std_access = Should use: log_student_access

- It's better to use all small Caps letter.

- For Column names, apply same methods.   Use 'underscore' as separator, use small caps

- For Primary Key column which has a name like id/no/, use the table name or meaningful name with it. For example: for a table: student_info {id, name, age ..}   use "student_id" instead of just "id".
Use this same name as Foreign Keys. If there is a situation where other description needs to be added then add it as a postfix. For example:
student_info { student_id, name, age …}   <- Here it is Primary Key
course_taken { course_id, student_id, …}   <- Here it is Foreign Key
groups {group_id, group_name, student_id_1, student_id_2, student_id_3 …} <- Here it is used as foreign key with a postfix.

-

(1) PRIMARY KEY TYPE:

- A **Hash Primary Key** is made of one hash attribute. Amazon DynamoDB builds an unordered hash index on this primary key attribute.
- A **Hash and Range Primary Key** is made of two attributes. The first attributes is the hash attribute and the second one is the range attribute. Amazon DynamoDB builds an unordered hash index on the hash primary key attribute and a sorted range index on the range primary key attribute.

* DynamoDB doesn't support cross-table joins.

Example: A simple example would be a table with a hash key "Customer Id" and range key "Order Date".

(2) INDEX TYPE:

http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/SecondaryIndexes.html

Every index must have a name. When you issue Query requests using the index, you will need to specify the index name. The name must be unique per table. There 2 types of:

- A **Global Secondary Index** (GSI) has an alternate hash key and range key separate from the table and requires it's own throughput capacity. Will be billed accordingly. Besides, if you read a table by a GSI, it must be an Eventual read (not Consistent read).

- A **Local Secondary Index** (LSI) has an alternate range key and shares a hash key and throughput capacity with the table (the one declared primarily). Local Secondary Indexes still rely on the original Hash Key and there is only one provisioned throughput.

Benefit: The Query API lets you retrieve an item from a table by specifying the item's hash and range key. If you add an index, you will have more flexibility with the Query API as you can query an item by specifying that index hash key and range key.

Example:
If you have a user profile table: unique-id, name, email. Here if you need to make the table queryable on name, email - then the only way is to make them GSI (LSI won’t help).

In the previous table with a hash key "Customer Id" and range key "Order Date".
Using a LSI with an index range key "Delivery Date" would enable developers to write queries to answer "Display all orders made by a customer sorted by Delivery Date" or "Display all orders made by a customer with a Delivery Date in the last month".
Using a GSI with an index hash key "Destination City" and index range key "Delivery Date" would enable developers to write queries to answer "Display all orders shipped to Destination City sorted by Delivery Date" or "Display all orders shipped to Destination City with a Delivery Date this week".

Limitation:

- Can declare 5 LSI and 5 GSI per table.
- Number of range keys per has value limit: For a table with LSI, there is a limit on Item Collection size (Dev guide page # 258). For every distinct Hash key value the total size of all table and index items can not exceed 10 GB !

(3) INDEX HASH KEYS:
An index hash key allows you to query the table by specifying an index hash key value and optionally an index range key value. A hash key requires an equal to comparison operator. Choose a hash attribute that ensures that your workload is evenly distributed across hash keys.

(4) INDEX RANGE KEYS:
An index range key allows you to query the table by specifying an index hash key value and index range key value. A range key supports the following comparison operators: equal to, less than, less than or equal to, greater than, greater than or equal, between, and begins with

(5) PROJECTED ATTRIBUTES:
Projected attributes are attributes stored in the index and can be returned by index queries. LSI queries can also return attributes that are not projected by fetching them from the table. GSI queries can only return projected attributes. Note that projected attributes incur write and storage costs.

(6) PROVISIONED THROUGHPUT CAPACITY:
Throughput capacity are allocated for the table and each GSI. Amazon DynamoDB lets you specify how much read and write throughput capacity you wish to provision for your table. Using this information, Amazon will provision the appropriate resources to meet your throughput needs.
Provisioned Throughput = Work/Sec allowed on a table
Capacity Unit = Amount of provisioned throughput consumed by an operation

* *
When **deciding the capacity units** for your table, you must take the following into consideration:
(i) Item size ,
(ii) Expected read and write request rates

(iii) Consistency
(iv) Local secondary indexes

(7) READ CAPACITY UNIT:
A unit of Read Capacity enables you to perform one strongly consistent read per second (or two eventually consistent reads per second) of items of up to 4KB in size. Larger items will require more capacity.

Applicable for GetItem/BatchGetItem. If you run Query/Scan then the calculation will be different.

1 Read capacity unit = 1 Strongly consistent read/second OR 2 eventually consistent read/second - for items up 4 KB in size.

You can compute your required read capacity by the following formula:
```
Units of Capacity required for reads =
Number of item reads per second x ( item size (KB) / 4 )
(Rounded up to the nearest whole number)
```

For example, to calculate the number of read operations for an item of 10 KB, you would round up to the next multiple of 4 KB (12 KB) and then divide by 4 KB, for 3 read operations.

More on Read Capacity Calculation: Developer’s Guide - Page# 19.

(8) WRITE CAPACITY UNIT:
A unit of Write Capacity enables you to perform one write per second for items of up to 1KB in size. Larger items will require more capacity.

1 Write capacity unit = 1 write/second for items up to 1 KB in size.

You can compute your required write capacity by the following formula:
```
Units of Capacity required for writes =
Number of item writes per second x item size
(rounded up to the nearest KB).
```

* For tables with secondary indexes, DynamoDB consumes additional capacity units. For example, if you wanted to add a single 1 KB item to a table, and that item contained an indexed attribute, then you would need two write capacity units—one for writing to the table, and another for writing to the index.

Limitations:
- Provisioned throughput capacity Unit limits (for all region except US East region):

Per table 10,000 read or 10,000 write capacity units. Per account 20,000 read or 20,000 write capacity units.
- ‘*UpdateTable*' - Limits on modifying provisioned throughput on tables and GSI:

When Increasing, can call UpdateTable as often as required to increase Read/WriteCapacityUnits but must stay within the Per Table/Account limit.

When Decreasing, can call UpdateTable to reduce throughput but no more than 4 times in a single UTC calendar day.

(9) READ / WRITE CONSISTENCY:

Amazon DynamoDB stores three geographically distributed replicas of each table to enable high availability and data durability. Read consistency represents the manner and timing in which the successful write or update of a data item is reflected in a subsequent read operation of that same item.

Writing successfully in strongly consistent mode requires that your write succeed on a majority of servers that can contain the record, therefore any future consistent reads will always see the same data, because a consistent read must read a majority of the servers that can contain the desired record. If you do not perform a strongly consistent read, the system will ask a random server for the record, and it is possible that the data will not be up-to-date.

STRONGLY CONSISTENT:

Note:
- Strong consistent reads will cost twice the amount of consumed units than eventually consistent reads, thus consumes more TPS and hence costs more.
- It's also slightly slower.

EVENTUALLY CONSISTENT:

Eventually consistent services are often classified as providing BASE (Basically Available, Soft state, Eventual consistency) semantics, in contrast to traditional ACID (Atomicity, Consistency, Isolation, Durability) guarantees.

Note:
- GSIs support eventual consistency. When items are inserted or updated in a table, the GSIs are not updated synchronously. Under normal operating conditions, a write to a global secondary index will propagate in a fraction of a second. In unlikely failure scenarios, longer delays may occur. Because of this, your application logic should be capable of handling GSI query results that are potentially out-of-date. Note that this is the same behavior exhibited by other DynamoDB APIs that support eventually consistent reads.

Example:
Imagine three servers. Server 1, server 2 and server 3. To write a strongly consistent record, you pick two servers at minimum, and write the data. Let's pick 1 and 2. Now you want to read the data consistently. Pick a majority of servers. Let's say we picked 2 and 3. Server 2 has the new data, and this is what the system returns. Eventually consistent reads could come from server 1, 2, or 3. This means if server 3 is chosen by random, your new write will not appear yet, until replication occurs. If a single server fails, your data is still safe, but if two out of three servers fail your new write may be lost until the offline servers are restored.

<additional: Distributed DB system limitation>
In theoretical computer science, the CAP theorem, also known as Brewer's theorem, states that it is impossible for a distributed computer system to simultaneously provide all three of the following guarantees:

Consistency (all nodes see the same data at the same time)
Availability (a guarantee that every request receives a response about whether it succeeded or failed)
Partition tolerance (the system continues to operate despite arbitrary message loss or failure of part of the system)

QUERY/ SEARCH

In short:
<Query> Hash Key must be provided. Directly use Keys map/ Index. Doesn't scan full table.
<Scan> Hash key not needed. Can search on any Attribute. Does a full scan. Costly.
<Get Item> Hash and Range key must be provided.
Generally, a Query operation is more efficient (with limitations) than a Scan operation. A Scan operation SCANS ENTIRE TABLE, then filters out values to provide the desired result, essentially adding the extra step of removing data from the result set.

(1) **Scan**
(flexible but expensive, slow),
http://docs.aws.amazon.com/amazondynamodb/latest/APIReference/API_Scan.html

“A scan operation scans the entire table. You can specify filters to apply to the results to refine the values returned to you, after the complete scan. Amazon DynamoDB puts a 1MB limit on the scan (the limit applies before the results are filtered)”

(2) **Query**
(less flexible: you have to specify an hash, but less expensive, fast)
http://docs.aws.amazon.com/amazondynamodb/latest/APIReference/API_Query.html

“A query operation searches only primary key attribute values and supports a subset of comparison operators on key attribute values to refine the search process. A query returns all of the item data for the matching primary keys (all of each item's attributes) up to 1MB of data per query operation”

The supported subset for the RangeKeyCondition:ComparisonOperator of the Query API excludes CONTAINS and IN, which are both available within the Scan API though; only the comparison operator BETWEEN is available within both APIs. This limitation most likely stems from performance considerations, i.e. supporting CONTAINS would probably defeat the DynamoDB goal of predictable performance/throughput. As usual with NoSQL solutions, you will need to account for these limitations by tailoring your application design accordingly, i.e. addressing both your use case and the specific NoSQL architecture you are targeting at.

(3) **GetItem**
http://docs.aws.amazon.com/amazondynamodb/latest/APIReference/API_GetItem.html

To read an item from a DynamoDB table, use the GetItem operation. You must provide the name of the table, along with the primary key of the item you want. You need to specify the entire primary key. For example, if a table has a hash and range type primary key, you must supply a value for the hash attribute and a value for the range attribute. The following are the default behaviors for GetItem:
(1) performs eventually consistent read
(2) returns all of the item's attributes
(3) doesn't return any info about how many provisioned capacity units it consumes.

You can override these defaults using GetItem parameters. You can optionally request a strongly consistent read instead; this will consume additional read capacity units, but it will return the most up-to-date version of the item.

*** * How to get the best performance, how to reduce provisioned throughput costs, and how to avoid throttling by staying within read and write capacity units.**

i. Use One-to-Many Tables Instead Of Large Set Attributes:

If your table has items that store a large set type attribute, such as number set or string set, consider removing the attribute and breaking the table into two tables. To form one-to-many relationships between these tables, use the primary keys.

ii. Use Multiple Tables to Support Varied Access Patterns

If you frequently access large items in a DynamoDB table and you do not always use all of an item's larger attributes, you can improve your efficiency and make your workload more uniform by storing your smaller, more frequently accessed attributes as separate items in a separate table.

These are the possible searches by index:

- By Hash
- By Hash + Range
- By Hash + Local Index
- By Global index
- By Global index + Range Index

Retrieving multiple ITEMS:

You can use the Query and Scan operations in DynamoDB to retrieve multiple consecutive items from a table in a single request. With these operations, DynamoDB uses the cumulative size of the processed items to calculate provisioned throughput. For example, if a Query operation retrieves 100 items that are 1 KB each, the read capacity calculation is not { 100 × (4KB / 4) } = 100 read capacity units, as if those items were retrieved individually using GetItem or BatchGetItem. Instead, the total would be only 25 read capacity units:
{ (100 * 1024 bytes) = 100 KB, which is then divided by 4 KB}.

DYNAMODB LIMITATIONS:
While Amazon DynamoDB tackles the core problems of database scalability, management, performance, and reliability, it does not have all the functionality of a relational database. It does not support complex relational queries (e.g. joins) or complex transactions. If your workload requires this functionality, or you are looking for compatibility with an existing relational engine, you may wish to run a relational engine on Amazon RDS or Amazon EC2. While relational database engines provide robust features and functionality, scaling a workload beyond a single relational database instance is highly complex and requires significant time and expertise. As such, if you anticipate scaling requirements for your new application and do not need relational features, Amazon DynamoDB may be the best choice for you.

AWS SDK

Recommend to use the AWS Software Development Kits (SDKs). The easy-to-use libraries in the AWS SDKs make it unnecessary to call the DynamoDB API directly from the application. The libraries take care of request authentication, serialization, and connection management. If you decide not to use the AWS SDKs, then your application will need to construct individual service requests. Each request must contain a valid JSON payload and correct HTTP headers, including a valid AWS signature.

Object Persistence Model API:
AWS SDKs for Java and .Net provides an Object Persistence Model API that developer can use to map Client-Side Classes to DynamoDB Tables. This allows to call Object Methods instead of Low-Level DynamoDB API calls.

==
<check ??>
Document Path
Item Sharding

Region and End points

To reduce data latency in your applications, most Amazon Web Services products allow you to select a regional endpoint to make your requests. An endpoint is a URL that is the entry point for a web service. For example, `https://dynamodb.us-west-2.amazonaws.com` is an entry point for the Amazon DynamoDB service.

Amazon DynamoDB region list:
http://docs.aws.amazon.com/general/latest/gr/rande.html

Relevant useful links

Links:
http://aws.amazon.com/dynamodb/faqs/
http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/GettingStartedBeforeYouBegin.html

Local DynamoDB:
http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Tools.DynamoDBLocal.html

Pricing *:
http://aws.amazon.com/dynamodb/pricing/

Design Issue:

Support for JSON Structure:
http://aws.amazon.com/about-aws/whats-new/2014/10/08/amazon-dynamodb-now-supports-json-document-data-structure-and-large-items/
http://stackoverflow.com/questions/26118067/dynamodb-how-can-i-create-a-table-with-nested-json-structure

Why DynamoDB: Mongo vs Dynamo ?
http://bpossolo.blogspot.com/2013/02/dynamodb-vs-mongodb-vs-gae-datastore.html

DynamoDB API: Data Types:
http://docs.aws.amazon.com/amazondynamodb/latest/APIReference/API_Types.html

DynamoDB Purpose:
The main reason one would use DynamoDB is when they need scalable throughput; in other words, when your needs for write and/or read speeds fluctuate drastically and when you know you will occasionally spike to extremely high throughput requirements. For times when you expect to have huge throughput for writing, you can pay to scale for that small period of time and then you can reduce your costs by throttling down to a more sane limit. You can run MapReduce jobs over DynamoDB tables using Amazon Elastic Map Reduce. And you can also copy a DynamoDB table into an Amazon Redshift "warehouse"; once the data is copied into Redshift you can run efficient SQL queries over it and Redshift can efficiently do that over petabytes worth of data.