Leverage the Power of NoSQL for Suitable Workloads

Best Practices for Migrating from RDBMS to Amazon DynamoDB

Best Practices for Migrating from RDBMS to Amazon DynamoDB: Leverage the Power of NoSQL for Suitable Workloads

Table of Contents

Abstract and introduction **1**
Abstract 1
Introduction 1
Overview of Amazon DynamoDB **4**
Suitable workloads **6**
Unsuitable workloads **9**
Key concepts **10**
Migrating to DynamoDB from RDBMS **16**
Planning phase **17**
Data analysis phase **20**
Data modeling phase **22**
Data modeling example 23
Cost estimate calculation 31
Testing Phase **34**
Data migration phase **36**
Conclusion **37**
Contributors **38**
Further reading **39**
Document history **40**
Appendix: Concepts **41**
Notices **42**
AWS Glossary **43**

Abstract and introduction

Publication date: **February 28, 2022** (*Document history*)

Abstract

Software architects and developers have an array of choices for data storage and persistence. These include not only traditional relational database management systems (RDBMS), but also NoSQL databases, such as Amazon DynamoDB. Certain workloads will scale better and be more cost-effective to run using a NoSQL solution. This whitepaper highlights the best practices for migrating these workloads from an RDBMS to DynamoDB. It also discusses how NoSQL databases like DynamoDB differ from a traditional RDBMS, and proposes a framework for analysis, data modeling, and migration of data from an RDBMS into DynamoDB.

Introduction

For decades, the RDBMS was the de facto choice for data storage and persistence. Any data driven application, be it an e-commerce website or an expense reporting system, was almost certain to use a relational database to retrieve and store the data required by the application. The reasons for this are numerous and include the following:

- RDBMS is a mature and stable technology.
- The query language, SQL, is feature-rich and versatile.
- The servers that run an RDBMS engine are typically some of the most stable and powerful in the IT infrastructure.
- All major programming languages contain support for the drivers used to communicate with an RDBMS, as well as a rich set of tools for simplifying the development of database-driven applications.

These factors, and many others, have supported the wide adoption of the RDBMS. For architects and software developers, there simply wasn't a reasonable alternative for data storage and persistence – until now.

The growth of *internet scale* web applications, such as e-commerce and social media, the explosion of connected devices like smart phones and tablets, and the rise of big data have resulted in

new workloads that traditional relational databases are not well suited to handle. As systems designed for transaction processing, all RDBMS must support certain fundamental properties. These properties are defined by the acronym ACID: Atomicity, Consistency, Isolation, and Durability. Atomicity refers to *all or nothing* operations – a transaction processes completely or not at all. Consistency means that the process of a transaction causes a valid state transition or the transaction is cancelled. Once the transaction is committed, the state of the resulting data must conform to the constraints imposed by the database schema. Isolation requires that concurrent transactions run separately from one another. The isolation property guarantees that if concurrent transactions are run in serial, the end state of the data will be the same. Durability requires that the state of the data, once a transaction processes, be preserved. In the event of power or system failure, the database must be able to recover to the last known state.

These ACID properties are all desirable, but support for all four requires an architecture that poses some challenges for today's data intensive workloads. For example, consistency requires a well-defined schema and that all data stored in a database conform to that schema. This is great for ad-hoc queries and read-heavy workloads. For a workload consisting almost entirely of writes, such as the saving of a player's state in a gaming application, this enforcement of schema is expensive from a storage and compute standpoint. The game developer benefits little by forcing this data into rows and tables that relate to one another through a well-defined set of keys.

Consistency also requires locking some portion of the data until the transaction modifying it completes and then making the change immediately visible. For a bank transaction, which debits one account and credits another, this is required. This type of transaction is called *strongly consistent*. For a social media application, on the other hand, there really is no requirement that all users see an update to a data feed at precisely the same time. In this latter case, the transaction is *eventually consistent*. It is far more important that the social media application scale to handle potentially millions of simultaneous users even if those users see changes to the data at different times. Scaling an RDBMS to handle this level of concurrency, while maintaining strong consistency, requires upgrading to more powerful (and often proprietary) hardware. This is called *scaling up* or *vertical scaling*, it usually carries an extremely high cost and has an upper scalability limit. The more cost-effective way to scale a database to support this level of concurrency is to add server instances running on commodity hardware. This is called *scaling out* or *horizontal scaling* and it is typically far more cost-effective than vertical scaling.

NoSQL databases, such as Amazon DynamoDB, address the scaling and performance challenges found with RDBMS. The term *NoSQL* simply means that the database doesn't follow the relational model espoused by E.F Codd in his 1970 paper *A Relational Model of Data for Large Shared Data Banks*, which would become the basis for all modern RDBMS. As a result, NoSQL databases

vary much more widely in features and functionality than a traditional RDBMS. There is no common query language analogous to SQL, and query flexibility is generally replaced by high I/O performance and horizontal scalability. NoSQL databases don't enforce the notion of schema in the same way as an RDBMS. NoSQL databases may store semi-structured data, like JSON, they may store related values as column sets, or they may simply store key/value pairs.

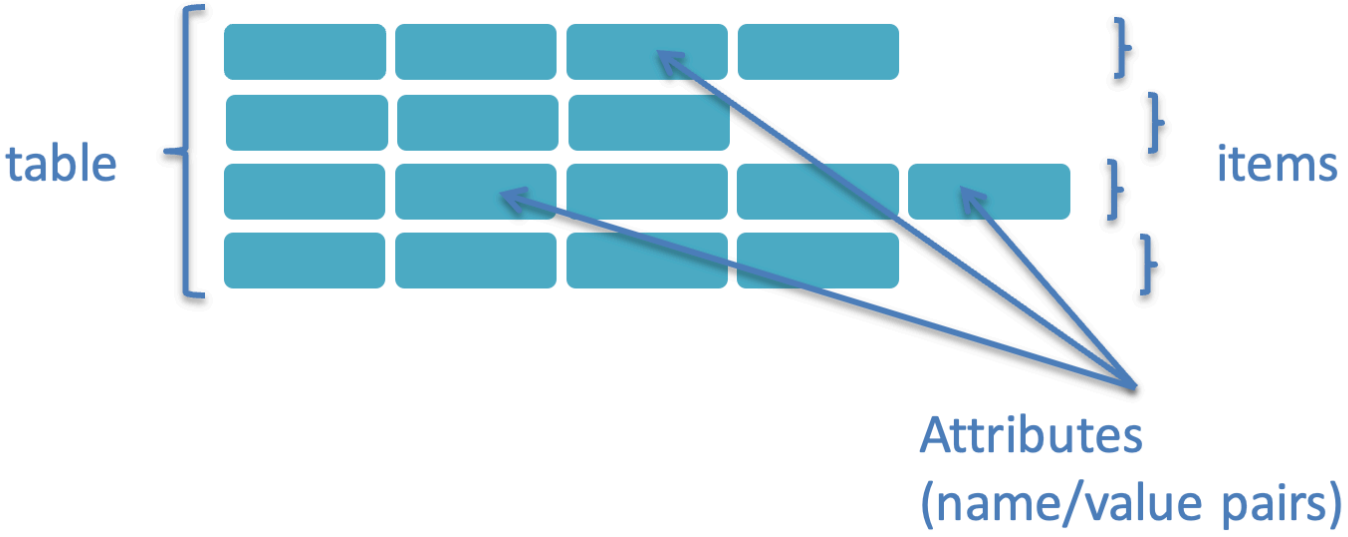
The net result is that NoSQL databases usually trade some of the query capabilities and ACID properties of an RDBMS for a much more flexible data model that scales horizontally. These characteristics make NoSQL databases an excellent choice in situations where use of an RDBMS (like the aforementioned game state example) is resulting in some combination of performance bottlenecks, operational complexity, and rising costs. DynamoDB offers solutions to all these problems, and is an excellent platform for migrating these workloads off of an RDBMS. In addition, DynamoDB supports strong consistency and ACID transactions, so even workloads that require such capabilities, which traditionally were not considered suitable for NoSQL databases, can take advantage of DynamoDB's scalability, flexible data model, and operational simplicity.

Overview of Amazon DynamoDB

Amazon DynamoDB is a fully managed, serverless, NoSQL database service running in the AWS cloud. The complexity of running a massively scalable, distributed NoSQL database is managed by the service itself, enabling software developers to focus on building applications rather than managing infrastructure. NoSQL databases are designed for scale, but their architectures are sophisticated, and there can be significant operational overhead in running a large NoSQL cluster. With DynamoDB, instead of having to become experts in advanced distributed computing concepts, the developer need only learn DynamoDB's straightforward API using the software development kit (SDK) for the programming language of choice.

DynamoDB is also cost-effective. With DynamoDB, you pay for the storage you are consuming and the read and write operations you are performing. It is designed to scale elastically. When the storage and throughput requirements of an application are low, only a small amount of capacity needs to be provisioned in the DynamoDB service. As the number of users of an application grows and the required I/O throughput increases, additional capacity can be provisioned by the service on the fly. This enables an application to seamlessly grow to support millions of users who are making millions of concurrent requests to the database every second.

Tables are the fundamental construct for organizing and storing data in DynamoDB. These tables consist of items and each item is composed of a primary key that uniquely identifies it and key/value pairs called attributes. While an item is similar to a row in an RDBMS table, all the items in the same DynamoDB table need not share the same set of attributes in the way that all rows in a relational table share the same columns. The figure *DynamoDB Table Structure* shows the structure of a DynamoDB table and the items it contains.



DynamoDB table structure

There is no concept of a column in a DynamoDB table. Each item in the table can be expressed as a tuple containing an arbitrary number of elements, up to a maximum size of 400KB. This schema flexibility of a DynamoDB table simplifies the process of modifying the data structure as requirements change over time and it allows a flexible data model where you can store multiple versions and types of objects in the same table. This data model is well suited for storing data in the formats commonly used for object serialization and messaging in distributed systems. As explained in the next section, workloads that involve this type of data are good candidates to migrate to DynamoDB.

Tables and items are created, updated, queried, and deleted through the DynamoDB API. In addition, DynamoDB supports PartiQL, a SQL-compatible query language that allows access to structured, semi-structured, and nested data. Whether using the native DynamoDB API, or PartiQL, to access the data, it's important to understand your application's data access patterns well in order to make the most effective use of DynamoDB.

Suitable workloads

With the introduction of DynamoDB features like ACID transactions, Amazon DynamoDB Accelerator (DAX), global tables, Amazon DynamoDB Time to Live (TTL), and Amazon DynamoDB Streams, the variety of workloads suitable for DynamoDB has expanded during the recent years. The following table outlines some of the more common use-cases for DynamoDB workloads.

Table 1 – Common use-cases for DynamoDB workloads

Application	Use case
Adtech	Capturing browser cookie state User events, clickstream, impressions datastore
Mobile applications	Storing application data and session state
Gaming applications	Storing user preferences and application state Storing players' game states
Consumer voting applications	Reality TV contests, Superbowl commercials
Large scale websites	Session state User data used for personalization Access control
Application monitoring	Storing application log and event data JSON data
Internet of Things	Sensor data and log ingestion
Retail	Shopping cart Customer profiles Inventory tracking and fulfillment
Finance	User transactions Event-driven transaction processing, fraud detection

All of the use-cases in this table benefit from some combination of the features that make DynamoDB so powerful. Adtech applications typically require extremely low and consistent latency, which is well suited for DynamoDB's low single digit millisecond read and write performance. DAX,

an in-memory cache for DynamoDB, provides microseconds latency for reads in cases where even lower read latency is a requirement. Because DAX is API-compatible with DynamoDB, you don't have to make functional application code changes and don’t need to make separate calls to your in-memory cache.

Mobile applications and consumer voting applications often have millions of users and need to handle many thousands of requests per second. DynamoDB can scale horizontally to meet this load. Gaming applications can rely on DynamoDB global tables for multi-region, active-active replication of data for business continuity and to allow globally distributed users to access the game in a region closest to them to reduce latency. Application monitoring solutions typically ingest hundreds of thousands or millions of data points per minute and DynamoDB’s schema-less data model, high performance, and support for a native JSON data type is a great fit for these types of applications. Finally, financial applications make use of DynamoDB ACID transactions in order to assure data integrity and correctness. They also utilize DynamoDB Streams and TTL to perform event driven processing, while taking advantage of DynamoDB scalability to support their workloads.

Another important characteristic to consider when determining if a workload is suitable for a NoSQL database like DynamoDB is whether it requires horizontal scaling. A mobile application may have millions of users, but each installation of the application will only read and write session data for a single user. This means the user session data in the DynamoDB table can be distributed across multiple storage partitions. A read or write of data for a given user will be confined to a single partition. This allows the DynamoDB table to scale horizontally—as more users are added, more partitions are created. As long as requests to read and write this data are uniformly distributed across partitions, DynamoDB will be able to handle a very large amount of concurrent data access. This type of horizontal scaling is difficult to achieve with an RDBMS without the use of *sharding*, which can add significant complexity to an application’s data access layer. When data in an RDBMS is *sharded*, it is split across different database instances. This requires maintaining an index of the instances and the range of data they contain. In order to read and write data, a client application needs to know which shard contains the range of data to be read or written. Sharding also adds administrative overhead and cost – instead of a single database instance, you are now responsible for keeping several up and running.

It’s also important to evaluate the data consistency requirement of an application when determining if a workload would be suitable for DynamoDB. Writes are always strongly consistent in DynamoDB. For reads there are two consistency models supported in DynamoDB: *strong consistency* and *eventual consistency*, with the former requiring more capacity than the latter. This flexibility enables the developer to get the best possible performance from the database

while being able to support the consistency requirements of the application. If an application does not require *strongly consistent* reads, meaning that updates made by one client do not need to be immediately visible to others, then use of an RDBMS that will force strong consistency can result in a tax on performance with no net benefit to the application. The reason is that strong consistency usually involves having to lock some portion of the data, which can cause performance bottlenecks.

Unsuitable workloads

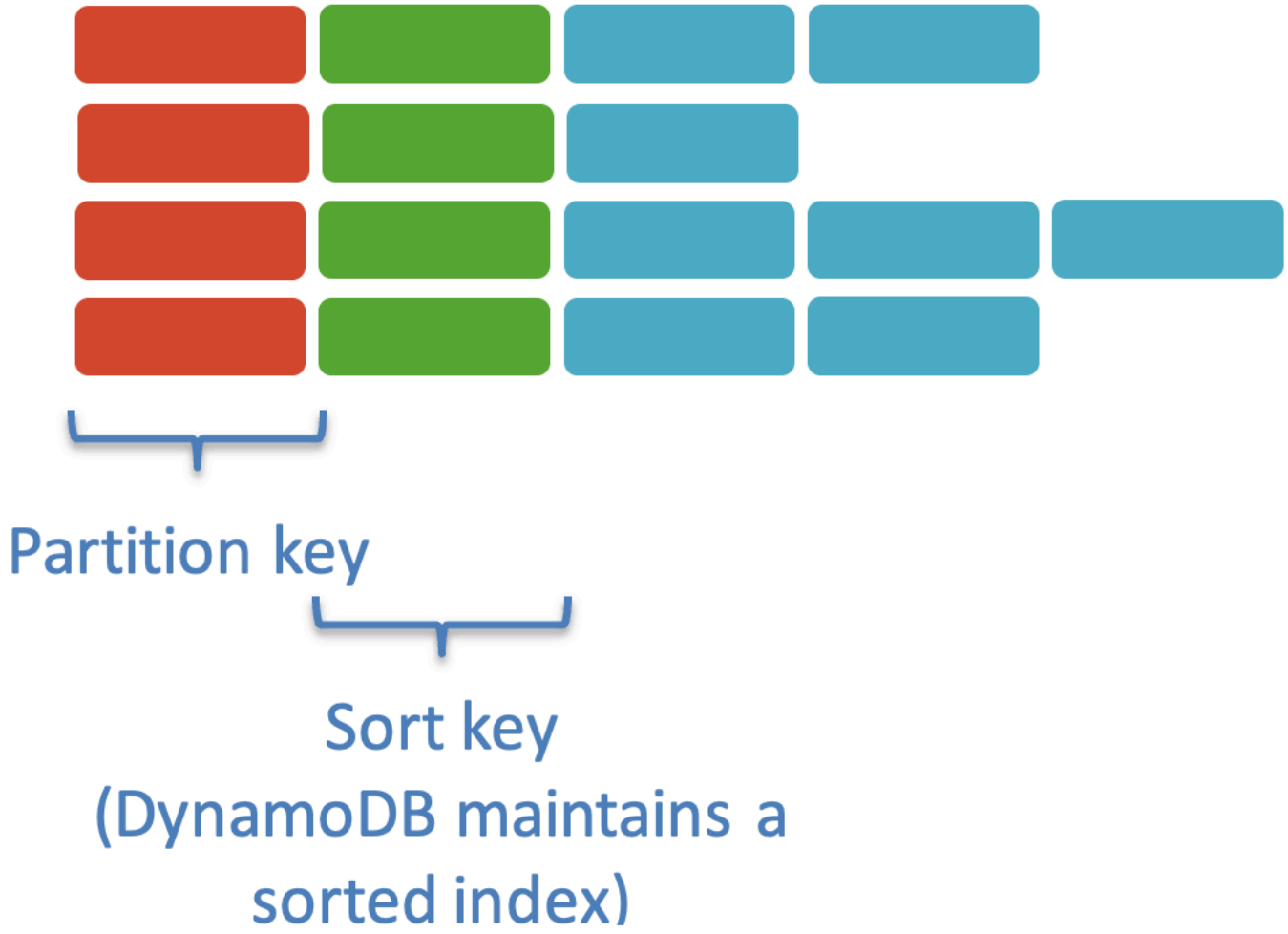
Some workloads are unsuitable for DynamoDB, including:

- Ad-hoc queries
- Online analytical processing (OLAP)
- Binary Large Object (Blob) storage

Because there is no concept of a table join in DynamoDB, constructing ad-hoc queries is not as efficient as it is with RDBMS. Running such queries with DynamoDB is possible, but requires the use of Amazon EMR, or AWS Glue with Hive, or Apache Spark. Likewise, OLAP applications are difficult to deliver as well, because the dimensional data model used for analytical processing requires joining fact tables to dimension tables. Finally, due to the size limitation of a DynamoDB item, storing Blobs is often not practical. DynamoDB does support a binary data type, but this is not suited for storing large binary objects, like images or documents. However, storing a pointer in the DynamoDB table to a large Blob stored in Amazon Simple Storage Service (Amazon S3) easily supports this last use-case.

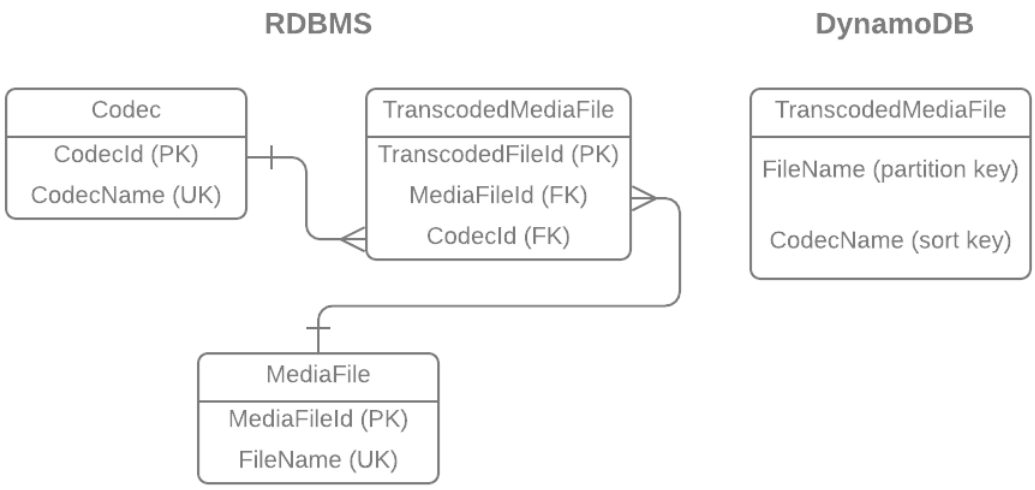
Key concepts

As described in the previous section, DynamoDB organizes data into tables consisting of items. Each item in a DynamoDB table can define an arbitrary set of attributes, but all items in the table must define a primary key that uniquely identifies the item. This key must contain an attribute known as the *partition key* and optionally an attribute called the *sort key*. The following figure shows the structure of a DynamoDB table that defines both a partition and sort key.



DynamoDB Table with partition and sort keys

If an item can be uniquely identified by a single attribute value, then this attribute can function as the partition key. In other cases, an item may be uniquely identified by two values. In this case, the primary key will be defined as a composite of the partition key and the sort key. The following figure demonstrates this concept.



Example of partition and sort keys

RDBMS tables relating media files with the codec used to transcode them can be modeled as a single table in DynamoDB using a primary key consisting of a partition and sort key. Note how the data is de-normalized in the DynamoDB table. This is a common practice when migrating data from an RDBMS to a NoSQL database, and will be discussed in more detail later in this whitepaper.

The ideal partition key will contain a large number of distinct values uniformly distributed across the items in the table. A user ID is a good example of an attribute that tends to be uniformly distributed across items in a table. Attributes that would be modeled as lookup values or enumerations in an RDBMS tend to make poor partition keys. The reason is that certain values may occur much more frequently than others. These concepts are shown in Table 2. Notice how the counts of user_id are uniform whereas the counts of status_code are not. If the status_code is used as a partition key in a DynamoDB table, the value that occurs most frequently will end up being stored on a single partition, and this means that most reads and writes will be hitting that single partition. This is called a *hot partition* and this will negatively impact performance.

Table 2 – Uniform and non-uniform distribution of potential key values

```
select user_id, count(*) as total from user_preferences group by user_id
```

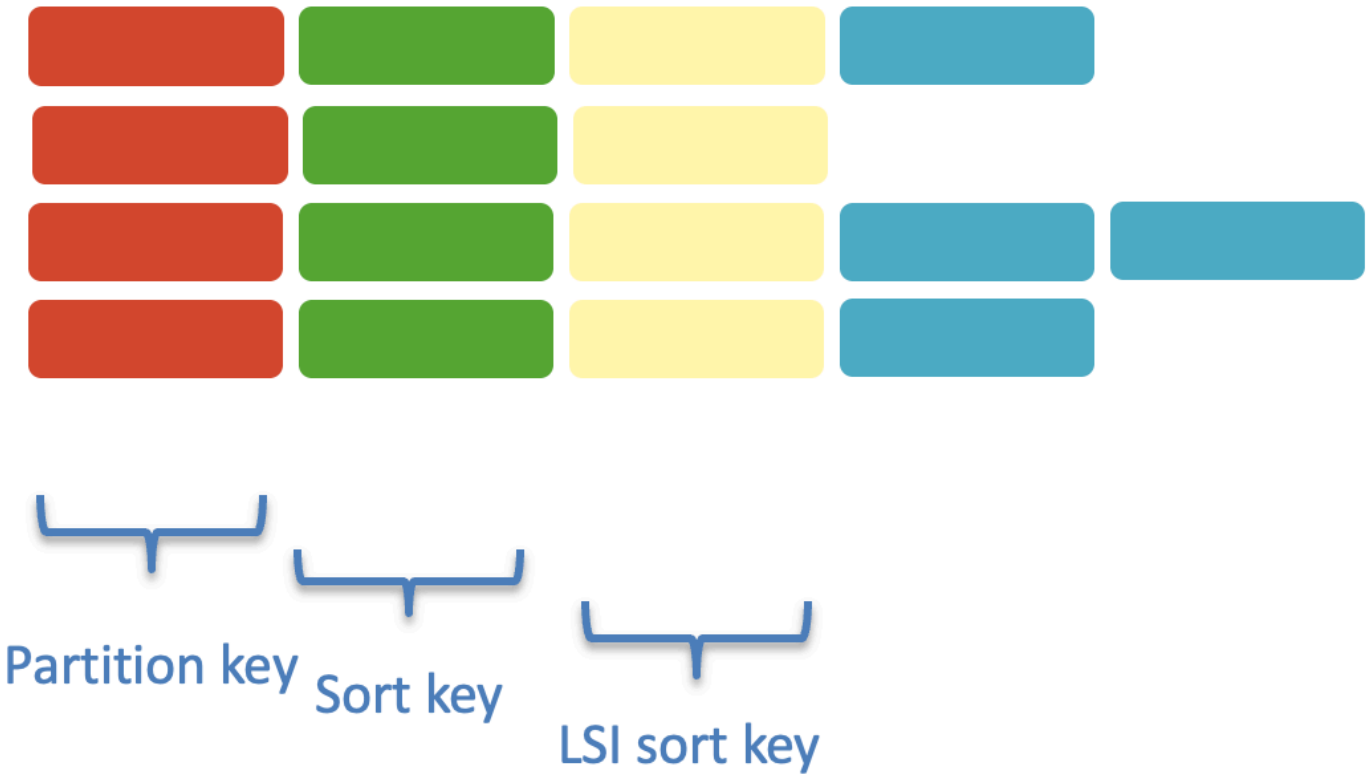
user_id	total	
8a9642f7-5155-4138-bb63-870cd45d7e19	1	

user_id	total
31667c72-86c5-4afb-82a1-a988bfe34d49	1
693f8265-b0d2-40f1-add0-bbe2e8650c08	1

```
select status_code, count(*) as total from status_code sc, log l where
l.status_code_id = sc.status_code_id
```

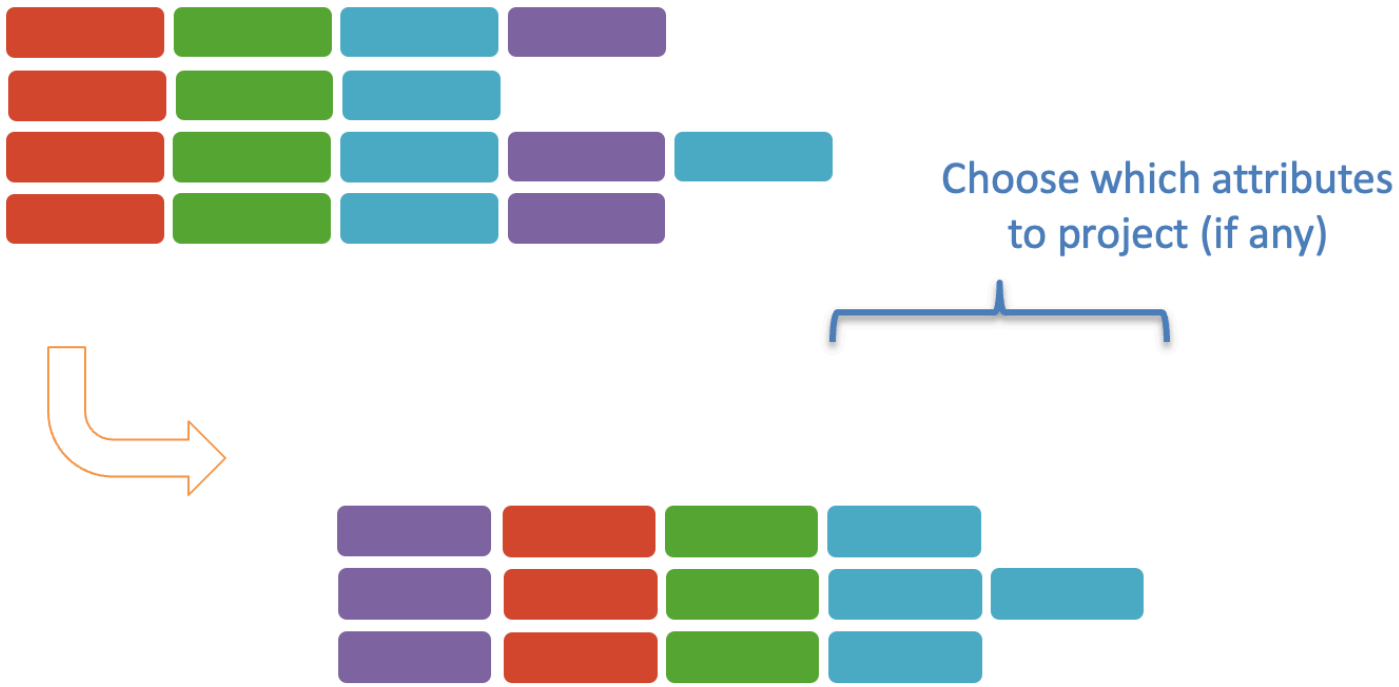
status_code	total
400	125000
403	250
500	1000
505	2

Items can be fetched from a table using the primary key. Often, it is useful to be able to fetch items using a different set of values than the partition and the sort keys. DynamoDB supports these operations through local and global secondary indexes. A local secondary index uses the same partition key as defined on the table, but a different attribute as the sort key. The following figure shows how a local secondary index is defined on a table. A global secondary index can use any scalar attribute as the partition or sort key. An important difference between the two index types is that a local secondary index can only be created at the time of the table's creation and it stays present until the table is deleted, while a global secondary index can be created and deleted at any moment. Fetching items using secondary indexes is done using the query interface defined in the DynamoDB API.



A local secondary index

Adding secondary indexes consumes additional storage and capacity for writes and so, as with any database, it is important to limit the number of indexes you define for a table. This requires understanding the data access requirements of any application that uses DynamoDB for persistent storage. In addition, global secondary indexes require that attribute values be projected into the index. What this means is that when an index is created, a subset of attributes from the parent table needs to be selected for inclusion into the index. When an item is queried using a global secondary index, the only attributes that will be populated in the returned item are those that have been projected into the index. The following figure demonstrates this concept.



Create a global secondary index on a table

The original partition and sort key attributes are automatically projected into the global secondary index. Reads on global secondary indexes are always eventually consistent, whereas local secondary indexes support eventual or strong consistency. Finally, both local and global secondary indexes consume capacity for reads and writes to the index. This means that when an item is inserted or updated in the main table, secondary indexes will consume capacity to update the index. The only exceptions to this are cases where the item isn't written to an index, because the attributes that are part of the index's primary key are not present in the item (refer to Sparse Indexes) or when an item modification isn't reflected in the index because the changed attributes aren't projected to the index.

DynamoDB allows for specifying the capacity mode for each table. With the on-demand capacity mode, which is suitable for workloads that are less predictable, the service takes care of managing capacity for you, and you only pay for what you consume. With provisioned capacity mode you are required to specify the table's read and write capacity and you pay based on the provisioned capacity.

Whenever an item is read from or written to a DynamoDB table or index, the amount of capacity required to perform the read or write operation is expressed as read capacity units (RCUs) or write capacity units (WCUs). One RCU represents one strongly consistent read per second, or two eventually consistent reads per second, for items up to 4KB in combined size.

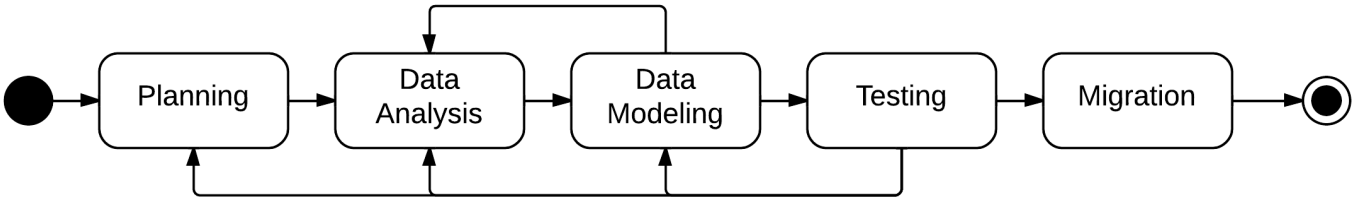
Transactional read requests require two RCUs to perform one read per second for items up to 4KB. One WCU represents one write per second for an item up to 1KB in size. Transactional write requests require two WCUs to perform one write per second for item up to 1KB. This means that fetching one or more items with a total size of 8KB in a single strongly consistent read will consume two RCUs. Making a regular (non-transactional) insert of an item of 8KB in size will consume eight WCUs.

With provisioned capacity mode you choose the number of RCUs and WCUs the table supports. If your application requires that 1000 4KB items be written per second, then the provisioned write capacity of the table would need to be a minimum of 4000 WCUs. When an insufficient amount of read or write capacity is provisioned on a table, the DynamoDB service will *throttle* the read and write operations. This can result in poor performance and in some cases throttling exceptions in the client application. For this reason, it is important to understand an application's I/O requirements when designing the tables. However, both read and write capacity can be dynamically altered on an existing table. If an application suddenly experiences a spike in usage that results in throttling, the provisioned capacity can be increased to handle the new workload. Similarly, if load decreases for some reason, the provisioned capacity can be reduced. This dynamic change in the table's read or write capacity can be achieved through a simple API call, or automatically through DynamoDB Auto Scaling. In addition, you can change the table's capacity mode once per 24 hours. This ability to dynamically alter the I/O characteristics of a table is a key differentiator between DynamoDB and a traditional RDBMS, in which I/O throughput is going to be fixed based on the underlying hardware the database engine is running on. This means that in many cases DynamoDB can be much more cost-effective than a traditional RDBMS, that is usually provisioned for peak consumption and stays underutilized for the majority of time.

Migrating to DynamoDB from RDBMS

In the previous section, some of the key features of DynamoDB were discussed, as well as some of the key differences between DynamoDB and a traditional RDBMS. In this section, a strategy for migrating from an RDBMS to DynamoDB that takes into account these key features and differences is proposed. Because database migrations tend to be complex and risky, we advocate taking a phased, iterative approach. As is the case with the adoption of any new technology, it's also good to focus on the easiest use cases first. It's also important to remember, as we propose in this section, that migration to DynamoDB doesn't need to be an all or nothing process. For certain migrations, it may be feasible to run the workload on both DynamoDB and the RDBMS in parallel, and switch over to DynamoDB only when it's clear that the migration has succeeded and the application is working properly.

The following state diagram expresses the proposed migration strategy:



Migration phases

It is important to note that this process is iterative. The outcome of certain states can result in a return to a previous state. Oversights in the data analysis and data-modeling phase may not become apparent until the testing phase. In most cases, it will be necessary to iterate over these phases multiple times before reaching the final data migration state. Each phase is discussed in detail in the following sections.

Planning phase

The first part of the planning phase is to identify the goals of the data migration. These often include, but are not limited to:

- Increasing application performance
- Lowering costs
- Reducing the load on an RDBMS
- Reducing operational overhead

In many cases, the goals of a migration may be a combination of all of the above. Once these goals have been defined, they can be used to inform the identification of the RDMBS tables to migrate to DynamoDB. Some good candidates for migration are:

- Entity-Attribute-Value tables
- Application session state tables
- User preference tables
- Logging tables

Once the tables have been identified, any characteristics of the source tables that may make migration challenging should be documented. This information will be essential for choosing a sound migration strategy. Let's take a look at some of the more common challenges that tend to impact the migration strategy:

Table 3 – Challenges that impact migration strategy

Challenge	Impact on migration strategy
Writes to the RDBMS source table will continue before and during the migration.	Consider a migration strategy that involves writing data to both the source and target tables in parallel in addition to bulk importing the existing data. As an alternative migration strategy, you can use the AWS Database Migration Service with ongoing replication to DynamoDB.

Challenge	Impact on migration strategy
The amount of data in the source table is in excess of what can reasonably be transferred with the existing network bandwidth.	Consider exporting the data from the source table to removable disks and using the AWS Import/Export or AWS Snowball services to import the data to a bucket in S3. Import this data into DynamoDB directly from S3. Alternatively, reduce the amount of data that needs to be migrated by exporting only those records that were created after a recent point in time. All data older than that point will remain in the legacy table in the RDBMS.
The data in the source table needs to be transformed before it can be imported into DynamoDB.	Consider using one of the following strategies: • Export the data from the source table and transfer it to S3. Consider using a Glue ETL job to perform the data transform ation, and import the transformed data into DynamoDB. • Use the AWS Database Migration Service with object mapping. • Perform the transformation in an applicati on code that will read data from the source database, transform as needed and write to DynamoDB.
The primary key structure of the source table is not portable to DynamoDB.	Identify columns that will make suitable partition keys and sort keys for the imported items. Alternatively, consider adding a surrogate key, such as a universally unique identifier (UUID), to the source table that will act as a suitable partition key.

Finally, and perhaps most importantly, the backup and recovery process should be defined and documented in the planning phase. If the migration strategy requires a full cutover from the

RDBMS to DynamoDB, defining a process for restoring functionality using the RDBMS in the event the migration fails is essential. To mitigate risk, consider running the workload on DynamoDB and the RDBMS in parallel for some length of time. In this scenario, the legacy RDBMS-based application can be disabled only once the workload has been sufficiently tested in production using DynamoDB.

Data analysis phase

The purpose of the data analysis phase is to understand the composition of the source data, and to identify the data access patterns used by the application. This information is required input for the data-modeling phase. It is also essential for understanding the cost and performance of running a workload on DynamoDB. The analysis of the source data should include:

- An estimate of the number of items to be imported into DynamoDB
- A distribution of the item sizes
- The multiplicity of values to be used as partition or sort keys

DynamoDB pricing contains two main components – read and write capacity and storage. By estimating the number of items that will be imported into a DynamoDB table, and the approximate size of each item, the storage and the capacity requirements for the table can be calculated. Common SQL data types will map to one of three scalar types in DynamoDB: string, number, or binary. The length of the number data type is variable, and strings are encoded using binary UTF-8. Focus should be placed on the attributes with the largest values when estimating item size, as capacity units are given in integral 1KB increments—there is no concept of a fractional capacity unit in DynamoDB. If an item is estimated to be 3.3KB in size, it will require four 1KB write capacity units and one 4KB read capacity unit to write and read a single item, respectively. Since the size will be rounded to the nearest kilobyte, the exact size of the numeric types is unimportant. In most cases, even for large numbers with high precision, the data will be stored using a small number of bytes. Because each item in a table may contain a variable number of attributes, it is useful to compute a distribution of item sizes and use a percentile value to estimate item size. For example, one may choose an item size representing the 95th percentile and use this for estimating the storage and capacity costs. In the event that there are too many rows in the source table to inspect individually, take samples of the source data and use these for computing the item size distribution.

Correctly estimating the required capacity is key to both guaranteeing the required application performance as well as understanding cost. Understanding the distribution frequency of RDBMS column values that could be partition or sort keys is essential for obtaining maximum performance as well. Columns containing values that are not uniformly distributed (that is, some values occur in much larger numbers than others) are not good partition or sort keys because accessing items with keys occurring in high frequency will hit the same DynamoDB partitions, and this has negative performance implications.

The second purpose of the data analysis phase is to categorize the data access patterns of the application. Because DynamoDB does not support joins and isn't optimized for ad-hoc queries, it is essential to document the ways in which data will be written to and read from the tables. This information is critical for the data-modeling phase, in which the tables, the key structure, and the indexes will be defined. Some common patterns for data access are:

- Write only – items are written to a table and never read by the application.
- Fetches by distinct value – items are fetched individually by a value that uniquely identifies the item in the table.
- Queries across a range of items – items are fetched as groups using a value that identifies multiple items in the table, for example, items that belong to the same user ID.

As discussed in the next section, documentation of an application's data access patterns using categories such as those described above will drive much of the data-modeling decisions.

Data modeling phase

In this phase, the tables, partition key, sort key, and secondary indexes are defined. The data model produced in this phase must support the data access patterns described in the data analysis phase. The first step in data modeling is to determine the partition key and sort key for a table. The primary key, whether consisting only of the partition key or a composite of the partition and sort key, must be unique for all items in the table. When migrating data from an RDBMS, it is tempting to use the primary key of the source table as the partition key, but in reality, this key is often meaningless to the application. For example, a user table in an RDBMS may define a numeric primary key, but an application responsible for logging in a user will ask for an email address, not the numeric user ID. In this case, the email address is the *natural key* and would be better suited as the partition key in the DynamoDB table, as items can easily be fetched by their partition key values. Modeling the partition key in this way is appropriate for data access patterns that fetch items by distinct value. For other data access patterns, like *write only*, using a randomly generated numeric ID will work well for the partition key. In this case, the items will never be fetched from the table by the application, and as such, the key will only be used to uniquely identify the items, not as a means of fetching data.

RDBMS tables that contain a unique index on two key values are good candidates for defining a primary key using both a partition key and a sort key. Intersection tables used to define many-to-many relationships in an RDBMS are typically modeled using a unique index on the key values of both sides of the relationship. Because fetching data in a many-to-many relationship requires a series of table joins, migrating such a table to DynamoDB would also involve denormalization of the data (discussed in more detail below).

Date values are also often used as sort key. A table counting the number of times a URL was visited on any given day could define the URL as the partition key and the date as the sort key. As with primary keys consisting solely of a partition key, fetching items with a composite primary key requires the application to specify both the partition and sort key values. This needs to be considered when evaluating whether a surrogate key or a natural key would make the better choice for the partition and or sort key.

Because non-key attributes can be added to an item arbitrarily, for tables with a simple primary key, the only attributes that must be specified in a DynamoDB definition are the partition key. For a table with a composite primary key, the partition key and the sort key must be specified. However, if secondary indexes are going to be defined on any non-key attributes, then these must be included in the table definition. Inclusion of non-key attributes in the table definition does not

impose any sort of schema on all the items in the table. Aside from the primary key, each item in the table can have an arbitrary list of attributes.

The support for SQL in an RDBMS means that records can be fetched using any of the column values in the table. These queries may not always be efficient – if no index exists on the column used to fetch the data, a full table scan may be required to locate the matching rows. With DynamoDB it is possible to do a full table scan by using either the native query interface or PartiQL, but this is inefficient and will consume substantial read units if the table is large. Instead, items can be fetched from a DynamoDB table by the primary key of the table, or the key of a local or global secondary index defined on the table. Because an index in a non-key column of an RDBMS table suggests that the application commonly queries for data on this value, these attributes make good candidates for local or global secondary indexes in a DynamoDB table. There are limits to the number of secondary indexes allowed on a DynamoDB table and, as previously explained, an index consumes additional storage and capacity for writes, so it is important to choose keys for these indexes using attribute values that the application will use most frequently for fetching data.

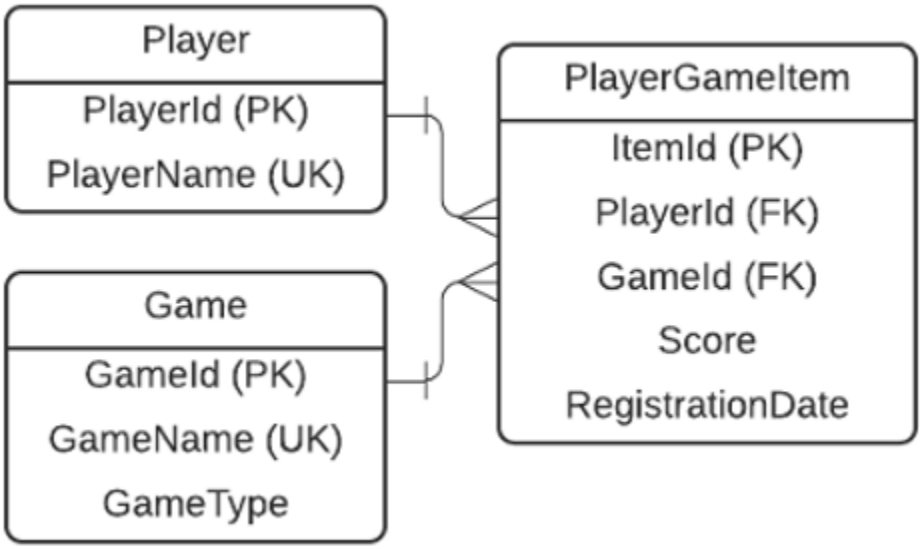
DynamoDB does not support the concept of a table join, so migrating data from an RDBMS table often requires denormalization of the data. To those used to working with an RDBMS, this will be a foreign and perhaps initially uncomfortable concept. For example, if a relational database contains a User and a UserAddress table, related through the UserID, one would combine the User attributes with the Address attributes into a single DynamoDB table. In the relational database, normalizing the UserAddress information allows for multiple addresses to be specified for a given user. This is a requirement for a contact management or CRM system. However, in DynamoDB, a user table would likely serve a different purpose—perhaps keeping track of a mobile application's registered users. In this use-case, the multiplicity of users to addresses is less important than scalability and fast retrieval of user records.

It is recommended to use the NOSQL Workbench for DynamoDB in order to perform data modeling for your DynamoDB tables. This client graphical user interface (GUI) application simplifies data modeling and visualization for DynamoDB.

Data modeling example

Let's walk through an example that combines the concepts described in this section and the previous. This example will demonstrate how to use secondary indexes for efficient data access, and how to estimate both the item size and the required capacity for a DynamoDB table. For this

example there is a gaming application that tracks players and the games they play. The following figure contains an Entity Relationship (ER) Diagram for the RDBMS schema of this application.



RDBMS schema for gaming application

A player can play multiple games and each game can host many players, so there is a many-to-many relationship between players and games, leading to a PlayersGamesItems association table in RDBMS.

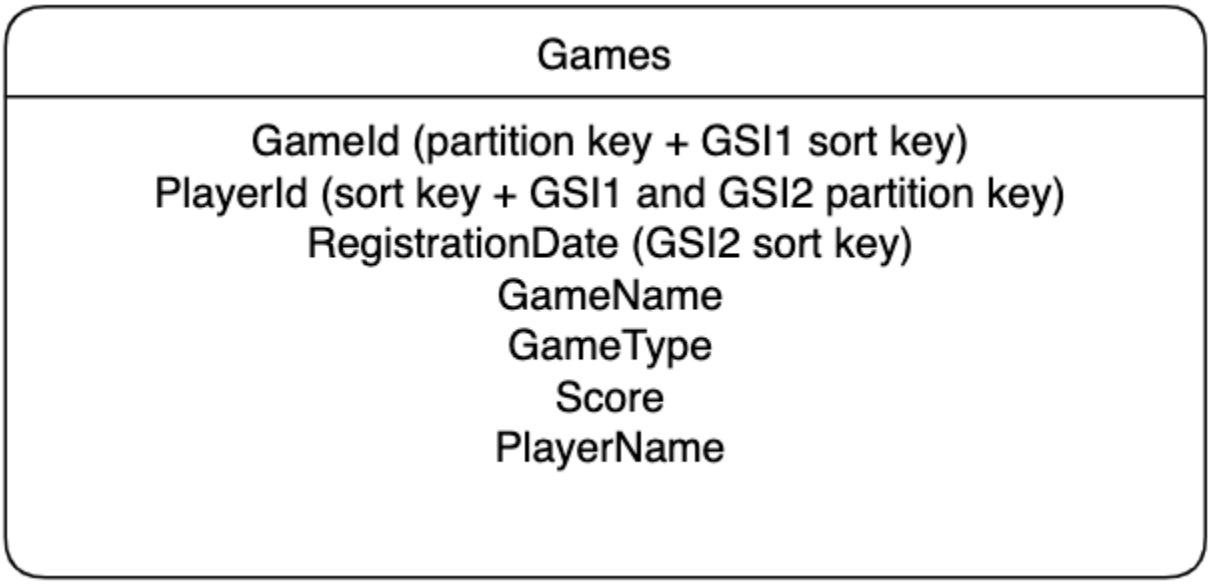
In order to perform data modeling for DynamoDB, we should first list the application's access patterns, from the most to the least frequently used. In this case the access patterns are:

Table 4 – Application access patterns

	Access pattern	Frequency/sec
1	Get details of a certain game	2000
2	Get details of a certain player	1000
3	Get list of games a player registered to within a certain time range	500
4	Get list of all games a player ever played	150
5	Update the player's score in a certain game	100

	Access pattern	Frequency/sec
6	Add an association between a player and a game he/she played	50
7	Get list of players who ever played a specific game	1

The following is the DynamoDB table and global secondary index (GSI) structure that will support the above access patterns:



DynamoDB schema for gaming application

In DynamoDB the user utilizes a single table with `GameId` being its partition key and `PlayerId` its sort key. This creates a one-to-many relationship between games and players and allows retrieval of data about a specific player or all players playing a particular game. In order to create a many-to-many relationship between games and players, the user creates a GSI with reversed partition and sort key. The user creates a second GSI to support additional access patterns: retrieving a list of games a player played sorted on the registration date or a list of games a player registered to between certain dates. The following figure shows DynamoDB table and GSIs with several added items, as can be seen in the NoSQL Workbench.

Games GSI: GSI1 GSI: GSI2

Primary key		Attributes				
Partition key: GameId	Sort key: PlayerId					
Game1	Player1	RegistrationDate	GameName	GameType	Score	PlayerName
		2021-11-22	BestGameEver	Sports	750	FirstPlayer
	Player2	RegistrationDate	GameName	GameType	Score	PlayerName
		2020-11-01	BestGameEver	Sports	2	SecondPlayer
Game2	Player1	RegistrationDate	GameName	GameType	Score	PlayerName
		2021-01-01	PopularGame	Shooters	57	FirstPlayer

Games GSI: GSI1 GSI: GSI2

Primary key		Attributes				
Partition key: PlayerId	Sort key: GameId					
Player1	Game1	RegistrationDate	GameName	GameType	Score	PlayerName
		2021-11-22	BestGameEver	Sports	750	FirstPlayer
	Game2	RegistrationDate	GameName	GameType	Score	PlayerName
		2021-01-01	PopularGame	Shooters	57	FirstPlayer
Player2	Game1	RegistrationDate	GameName	GameType	Score	PlayerName
		2020-11-01	BestGameEver	Sports	2	SecondPlayer

Games GSI: GSI1 GSI: GSI2

Primary key		Attributes				
Partition key: PlayerId	Sort key: RegistrationDate					
Player1	2021-01-01	GameId	GameName	GameType	Score	PlayerName
		Game2	PopularGame	Shooters	57	FirstPlayer
	2021-11-22	GameId	GameName	GameType	Score	PlayerName
		Game1	BestGameEver	Sports	750	FirstPlayer
Player2	2020-11-01	GameId	GameName	GameType	Score	PlayerName
		Game1	BestGameEver	Sports	2	SecondPlayer

DynamoDB table and GSIs in NoSQL Workbench

As the DynamoDB table and GSIs in NoSQL Workbench figure shows, the user denormalized the data. The game name, type, and player name are duplicated in each item in the table. This increases the storage usage, but also enables retrieving these details in a single, fast query, for example when querying for the games Player2 ever played. Depending on the requirements, we could model our data differently. If retrieving the game name and type isn't required for the majority of our queries, we could store these details in a single *root* game item, instead of storing these in all the items. Similarly, we could store any generic player related data in root player item. With such data modeling the data would appear as follows:

Games_alternative GSI: GSI1 GSI: GSI2

Primary key		Attributes		
Partition key: PK	Sort key: SK			
G#Game1	P#Player1	RegistrationDate	Score	PlayerId
		2021-11-22	750	P#Player1
	P#Player2	RegistrationDate	Score	PlayerId
		2020-11-01	2	P#Player2
	root	GameName	GameType	
		BestGameEver	Sports	
G#Game2	P#Player1	RegistrationDate	Score	PlayerId
		2021-01-01	57	P#Player1
	root	GameName	GameType	
		PopularGame	Shooters	
P#Player1	root	PlayerName		
		FirstPlayer		
P#Player2	root	PlayerName		
		SecondPlayer		

Games_alternative GSI: GSI1 GSI: GSI2

Primary key		Attributes		
Partition key: PlayerId	Sort key: PK			
P#Player1	G#Game1	SK	RegistrationDate	Score
		P#Player1	2021-11-22	750
	G#Game2	SK	RegistrationDate	Score
		P#Player1	2021-01-01	57
P#Player2	G#Game1	SK	RegistrationDate	Score
		P#Player2	2020-11-01	2

Games_alternative GSI: GSI1 GSI: GSI2

Primary key		Attributes		
Partition key: PlayerId	Sort key: RegistrationDate			
P#Player1	2021-01-01	PK	SK	Score
		G#Game2	P#Player1	57
	2021-11-22	PK	SK	Score
		G#Game1	P#Player1	750
P#Player2	2020-11-01	PK	SK	Score
		G#Game1	P#Player2	2

DynamoDB table and GSIs with alternative data modeling in NoSQL Workbench

With the alternative data modeling, the *partition key* attribute is called PK and the *sort key* attribute is called SK. This is because the table is now overloaded with multiple types of objects. There are root game items containing generic game details, root player items containing generic player details, and items that associate players to games they played.

Also note that these different types of items contain different attributes. DynamoDB's schema-less data model allows us to perform this kind of table overloading and it is considered a best practice in DynamoDB. You might also notice that the PlayerID is now present in both the SK and the PlayerID attributes. This in order to use the PlayerID attribute as the GSI's sort key, to only include players to games association items in the GSIs, making them sparse indexes. As can be seen, the user prefixes the game IDs with G# and the Player IDs with P# to ensure that game and player IDs are unique.

The next step is to list access patterns and the tables, or GSIs, that serve them. Here is the list with the fully denormalized modeling and the list with alternative modeling:

Table 5 – Access patterns mapping to DynamoDB table and GSIs with fully denormalized modeling

	Access pattern	Read served by	Passed primary key values
1	Get game details for the given `GameId`	Table	GameId=X, limit=1
2	Get player details for the given `PlayerId`	GSI1	PlayerId=X, limit=1

	Access pattern	Read served by	Passed primary key values
3	Get a list of games a player registered to within a time range, for the given PlayerId and time range	GSI2	PlayerId=X, RegistrationDate between Y and Z
4	Get list of all games a player ever played	GSI1	PlayerId=X
5	Update the player's score in a certain game	–	GameId=X, PlayerId=Y
6	Add an association between a player and a game he/she played	–	GameId=X, PlayerId=Y
7	Get list of players who ever played a specific game	Table	GameId=X

Table 6 – Access patterns mapping to DynamoDB table and GSIs with alternative modeling

	Access pattern	Read served by	Passed primary key values
1	Get game details for the given game ID passed in as PK	Table	PK=X, SK=root
2	Get player details for the given player ID passed in as PK	Table	PK=X, SK=root
3	Get a list of games a player registered to within a time range, for the given `PlayerId` and time range	GSI2	PlayerId=X, RegistrationDate between Y and Z
4	Get list of all games a player ever played	GSI1	PlayerId=X
5	Update the player's score in a certain game	–	PK=X, SK=Y
6	Add an association between a player and a game he/she played	–	PK=X, SK=Y

	Access pattern	Read served by	Passed primary key values
7	Get list of players who ever played a specific game	Table	PK=X

Cost estimate calculation

This section will show a cost calculation for the fully denormalized data modeling. The number of rows that will be migrated is around 10^8, so computing the 95^{th} percentile of item size iteratively is not practical. Instead, the user will perform simple random sampling with replacement of 10^6 rows. This will give the user adequate precision for the purposes of estimating item size. To do this, construct a SQL view that contains the fields that will be inserted into the DynamoDB table. The sampling routine then randomly selects 10^6 rows from this view and computes the size representing the 95^{th} percentile.

This statistical sampling yields a 95^{th} percentile size of 0.7 KB. The number of write capacity units required to write a single item to the table is:

```
ceiling (0.7KB per item/1KB per write capacity unit) = 1 write capacity
unit per item
```

In the user's fully denormalized data modeling, every item that is written to the table is being propagated to both GSIs, which means that one write capacity unit will in addition be consumed from each GSI.

The number of read capacity units required to read a single item is computed similarly:

```
ceiling (0.7KB per item/4KB per read capacity unit) = 1 read capacity
unit per item for strongly consistent read,or 0.5 read capacity unit with
eventually consistent read
```

The number of read capacity units required to read 100 items for example in a single operation is computed as follows:

```
ceiling (0.7KB per item * 100/4KB per read capacity unit) = 18 read
capacity units for strongly consistent read,or 9 read capacity units with
eventually consistent read
```

All reads in this gaming application are eventually consistent. For the sake of simplicity, the user assumes that all attributes are projected to both GSIs. This doesn't have to be the case in real life. For example, the score attribute doesn't necessarily need to be projected to GSI2, which can reduce costs.

The following table contains the capacity required for each access pattern:

Table 7 – Capacity required per access pattern

	Access pattern	Frequency/ sec	Comments	Served by	Required capacity	
1	Get details of a certain game	2000	–	Table	GameId=X, limit=1	
2	Get details of a certain player	1000	–	GSI1	PlayerId=X, limit=1	
3	Get list of games a player registered to within a certain time range	500	Retrieves 3 items on average	GSI2	PlayerId= X, Registrat ionDate between Y and Z	
4	Get list of all games a player ever played	150	Retrieves 10 items on average	GSI1	PlayerId=X	
5	Update the player's score in a certain game	100	–	Table and Indexes	GameId=X, PlayerId=Y	
6	Add an association between a player and a game he/she played	50	–	Table and Indexes	GameId=X, PlayerId=Y	
7	Get list of players who ever played a specific game	1	Retrieves 1000 items on average	Table	GameId=X	

The following table contains the total capacity required for the workload:

Table 8 – Total capacity required for the workload

	Table/index	WCUs	RCUs	
1	Table	100+50=150	1000+87.5=1087.5	
2	GSI1	100+50=150	500+150=650	
3	GSI2	100+50=150	250	

We now have all the data we need to estimate the DynamoDB cost for this workload:

- Number of items ($10^{8)}$
- Item size (0.7KB)
- Number of tables and indexes (3). This number should be multiplied by the dataset size to derive the total storage size, because in this example all the items and all attributes for each item are projected into each index. Otherwise, when that is not the case, you need to estimate the storage size for each index and then calculate the total storage size by adding the storage sizes for the table and all the indexes.
- Write capacity units (450)
- Read capacity units (1987.5)

These can be run through the AWS Pricing Calculator to derive a cost estimate.

Testing Phase

The testing phase is the most important part of the migration strategy. It is during this phase that the entire migration process will be tested end-to-end. A comprehensive test plan should minimally contain the following:

Table 9 – Data migration test plan

Test category	Purpose
Basic acceptance tests	These tests should be automatically executed upon completion of the data migration routines. Their primary purpose is to verify whether the data migration was successful. Some common outputs from these tests will include: • Total # items processed • Total # items imported • Total # items skipped • Total # warnings • Total # errors If any of these totals reported by the tests deviate from the expected values, then it means the migration was not successful and the issues need to be resolved before moving to the next step in the process or the next round of testing.
Functional tests	These tests exercise the functionality of the application(s) using DynamoDB for data storage. They include a combination of automated and manual tests. The primary purpose of the functional tests is to identify problems in the application caused by the migration of the RDBMS data to DynamoDB. It is during this round of testing that gaps in the data model are often revealed.
Non-functional tests	These tests assess the non-functional characteristics of the applicati on, such as performance under varying levels of load, and resiliency to failure of any portion of the application stack. These tests can also

Test category	Purpose
	include boundary or edge cases that are low-probability but could negatively impact the application (for example, if a large number of clients try to update the same record at the exact same time). The backup and recovery process defined in the planning phase should also be included in non-functional testing.
User acceptance tests	These tests should be executed by the end-users of the applications once the final data migration has completed. These tests help end-users decide if the application is sufficient to meet their requirements.

Because the migration strategy is iterative, these tests are executed multiple times. For maximum efficiency, consider testing the data migration routines using a sampling from the production data if the total amount of data to migrate is large. The outcome of the testing phase often requires revisiting a previous phase in the process. The overall migration strategy becomes more refined through each iteration of the process. After all the tests execute successfully, it is time for the next and final phase: data migration.

Data migration phase

In the data migration phase, the full set of production data from the source RDBMS tables is migrated into DynamoDB. By the time this phase is reached, the end-to-end data migration process has been tested and vetted thoroughly. All the steps of the process are carefully documented, so running it on the production data set should be as simple as following a procedure that has been executed numerous times before.

After the data migration is complete, the user acceptance tests defined in the previous phase should be executed one final time to ensure that the application is in a usable state. In the event that the migration fails for any reason, the backup and recovery procedure—which is also thoroughly tested and vetted at this point—can be initiated. When the system is back to a stable state, a root cause analysis of the failure should be conducted and the data migration rescheduled once the root cause is resolved. If all goes well, the application should be closely monitored over the next several days until there is sufficient data indicating that the application is functioning normally.

Conclusion

Using DynamoDB for suitable workloads can result in lower costs, a reduction in operational overhead, and an increase in performance, availability, and reliability when compared to a traditional RDBMS. In this paper, AWS proposed a strategy for identifying and migrating suitable workloads from an RDBMS to DynamoDB. While implementing such a strategy will require careful planning and engineering effort, the return on investment (ROI) of migrating to a fully managed, serverless, NoSQL solution such as DynamoDB should greatly exceed the upfront cost associated with the effort.

Contributors

Contributors to this document include:

- Nathaniel Slater, Senior Practice Manager, Amazon Web Services
- Leonid Koren, Principal NoSQL Solutions Architect, Amazon Web Services

Further reading

For additional information, refer to:

- DynamoDB Developer Guide
  - Best Practices for Designing and Architecting with DynamoDB
  - NoSQL Design for DynamoDB
  - Best Practices for Modeling Relational Data in DynamoDB
- Amazon DynamoDB Website
- Data modeling with NOSQL Workbench for Amazon DynamoDB blog
- How to determine if Amazon DynamoDB is appropriate for your needs blog

Document history

Cheat sheet

The following is a summary some of the key concepts discussed in this paper, and the sections where those concepts are detailed:

Concept	Section
Determining suitable workloads	Suitable workloads
Choosing the right key structure	Key concepts
Table indexing	Data modeling phase
Provisioning read and write throughput	Data modeling example
Choosing a migration strategy	Planning phase

Notices

Customers are responsible for making their own independent assessment of the information in this document. This document: (a) is for informational purposes only, (b) represents current AWS product offerings and practices, which are subject to change without notice, and (c) does not create any commitments or assurances from AWS and its affiliates, suppliers or licensors. AWS products or services are provided “as is” without warranties, representations, or conditions of any kind, whether express or implied. The responsibilities and liabilities of AWS to its customers are controlled by AWS agreements, and this document is not part of, nor does it modify, any agreement between AWS and its customers.

AWS Glossary

For the latest AWS terminology, see the AWS glossary in the *AWS Glossary Reference*.