Best Practices for Migrating from RDBMS to Amazon DynamoDB

Leverage the Power of NoSQL for Suitable Workloads

Nathaniel Slater

March 2015

Contents

Abstract

Today, 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 paper will highlight the best practices for migrating these workloads from an RDBMS to DynamoDB. We will discuss how NoSQL databases like DynamoDB differ from a traditional RDBMS, and propose 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 this incumbency 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 a system designed for transaction processing, the fundamental properties that all RDBMS must support are defined by the acronym ACID: Atomicity, Consistency, Isolation, and Durability. Atomicity means “all or nothing” – a transaction executes completely or not at all. Consistency means that the execution of a transaction causes a valid state transition. Once the transaction has been committed, the state of the resulting data must conform to the constraints imposed by the database schema. Isolation requires that concurrent transactions execute separately from one another. The isolation property guarantees that if concurrent transactions were executed in serial, the end state of the data would be the same. Durability requires that the state of the data once a transaction executes be preserved. In the event of power or system failure, the database should 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" and it usually carries an extremely high cost. 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, like 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*,[1] 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. Some may store semi-structured data, like JSON. Others may store related values as column sets. Still others may simply store key/value pairs.

The net result is that NoSQL databases 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 for non-relational workloads (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.

Overview of Amazon DynamoDB

Amazon DynamoDB is a fully managed NoSQL database service running in the AWS cloud. The complexity of running a massively scalable, distributed NoSQL database is managed by the service itself, allowing 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

[1] http://www.seas.upenn.edu/~zives/03f/cis550/codd.pdf

overhead in running a large NoSQL cluster. Instead of having to become experts in advanced distributed computing concepts, the developer need only to learn DynamoDB's straightforward API using the SDK for the programming language of choice.

In addition to being easy to use, DynamoDB is also cost-effective. With DynamoDB, you pay for the storage you are consuming and the IO throughput you have provisioned. 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 IO throughput increases, additional capacity can be provisioned on the fly. This enables an application to seamlessly grow to support millions of users making thousands of concurrent requests to the database every second.

Tables are the fundamental construct for organizing and storing data in DynamoDB. A table consists of items. An 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. Figure 1 shows the structure of a DynamoDB table and the items it contains. 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 400K. This data model is well suited for storing data in the formats commonly used for object serialization and messaging in distributed systems. As we will see in the next section, workloads that involve this type of data are good candidates to migrate to DynamoDB.

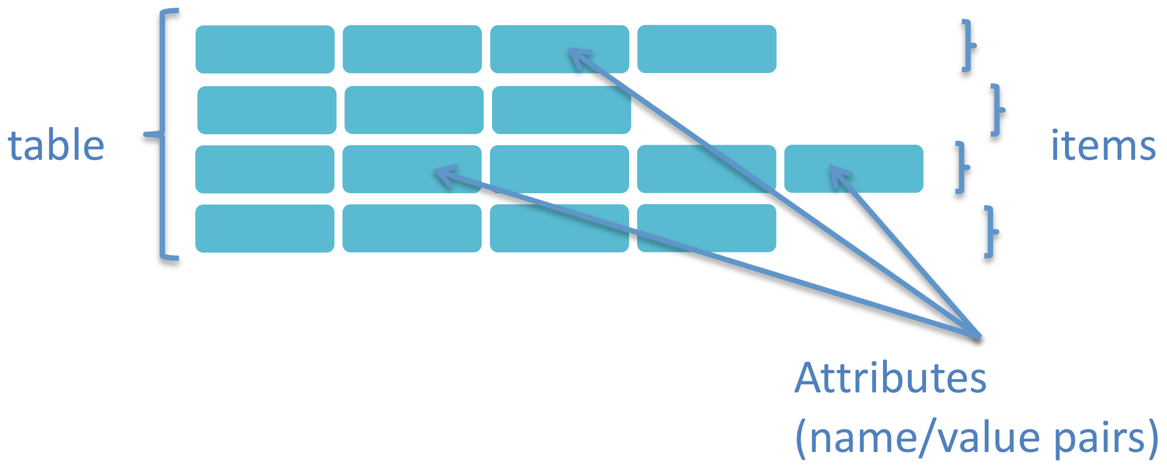

Figure 1: DynamoDB Table Structure

Tables and items are created, updated, and deleted through the DynamoDB API. There is no concept of a standard DML language like there is in the relational database world. Manipulation of data in DynamoDB is done programmatically through object-oriented code. It is possible to query data in a DynamoDB table, but this too is done programmatically through the API. Because there is no generic query language like SQL, it's important to understand your application's data access patterns well in order to make the most effective use of DynamoDB.

Suitable Workloads

DynamoDB is a NoSQL database, which means that it will perform best for workloads involving non-relational data. Some of the more common use-cases for non-relational workloads are:

- Ad-Tech
  - Capturing browser cookie state
- Mobile applications
  - Storing application data and session state
- Gaming applications
  - Storing user preferences and application state
  - Storing players' game state
- 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

All of these use-cases benefit from some combination of the features that make NoSQL databases so powerful. Ad-Tech applications typically require extremely low latency, which is well suited for DynamoDB's low single digit millisecond read and write performance. Mobile applications and consumer voting applications often have millions of users and need to handle thousands of requests per second. DynamoDB can scale horizontally to meet this load. Finally, application monitoring solutions typically ingest hundreds of thousands of data points per minute, and DynamoDB's schema-less data model, high IO performance and support for a native JSON data type is a great fit for these types of applications.

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. There are actually two consistency models supported in DynamoDB: strong and eventual consistency, with the former requiring more provisioned IO than the latter. This flexibility allows 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

Not all workloads are suitable for a NoSQL database like DynamoDB. While in theory one could implement a classic entity-relationship model using DynamoDB tables and items, in practice, this would be extremely cumbersome to work with. Transactional systems that require well-defined relationships between entities are still best implemented using a traditional RDBMS. Some other unsuitable workloads include:

- Ad-hoc queries
- OLAP
- BLOB storage

Because DynamoDB does not support a standard query language like SQL, and because there is no concept of a table join, 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 and Hive. 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 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 “hash key” and optionally an attribute called the “range key.” Figure 2 shows the structure of a DynamoDB table defining both a hash and range key.

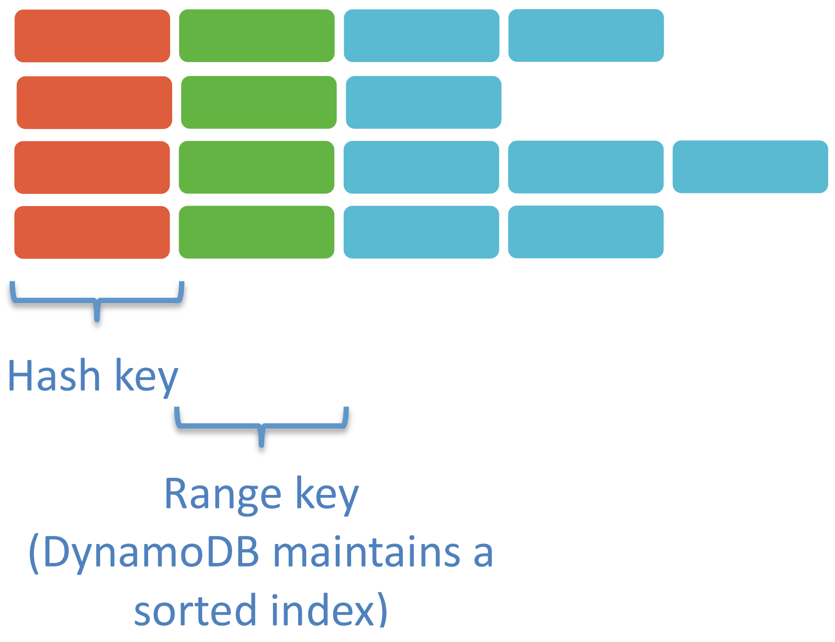

Figure 2: DynamoDB Table with Hash and Range Keys

If an item can be uniquely identified by a single attribute value, then this attribute can function as the hash 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 hash key and the range key. Figure 3 demonstrates this concept. An RDBMS table 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 hash and range 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 paper.

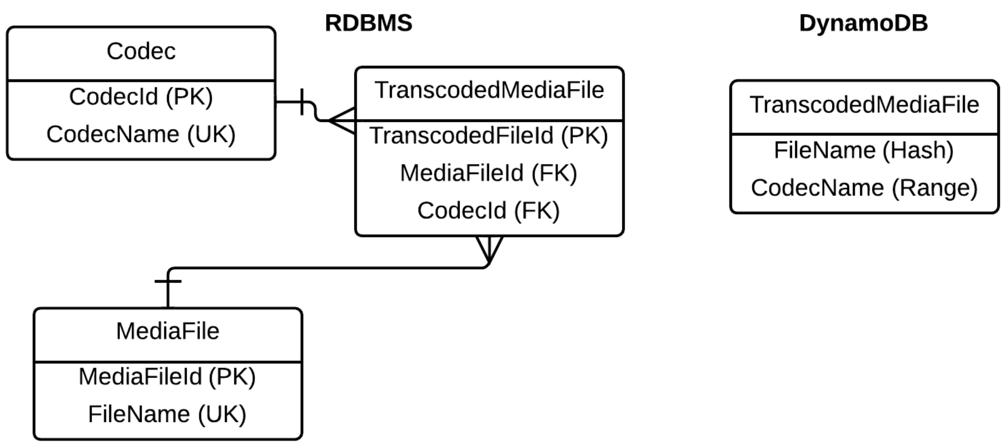

Figure 3: Example of Hash and Range Keys

The ideal hash 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 hash keys. The reason is that certain values may occur much more frequently than others. These concepts are shown in Figure 4. 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 hash key in a DynamoDB table, the value that occurs most frequently will end up being stored on the same 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.

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

user_id	total
8a9642f7-5155-4138-bb63-870cd45d7e19	1
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	10000
505	2

Figure 4: Uniform and Non-Uniform Distribution of Potential Key Values

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 hash and the range keys. DynamoDB supports these operations through local and global secondary indexes. A local secondary index uses the same hash key as defined on the table, but a different attribute as the range key. Figure 5 shows how a local secondary index is defined on a table. A global secondary index can use any scalar attribute as the hash or range key. Fetching items using secondary indexes is done using the query interface defined in the DynamoDB API.

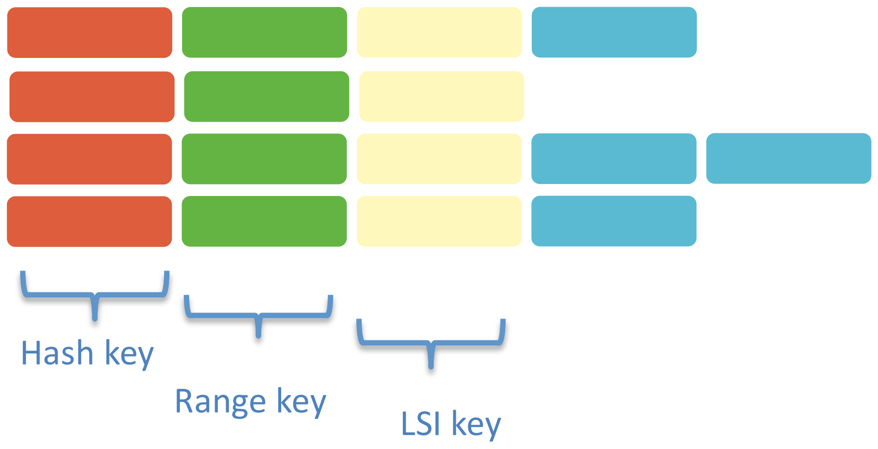

Figure 5: A Local Secondary Index

Because there are limits to the number of local and global secondary indexes that can exist per table, it is important to fully understand 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 need 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. Figure 6 demonstrates this concept. Note how the original hash and range key attributes are automatically promoted in 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 use provisioned IO (discussed in detail below) for reads and writes to the index. This means that each time an item is inserted or updated in the main table, any secondary indexes will consume IO to update the index.

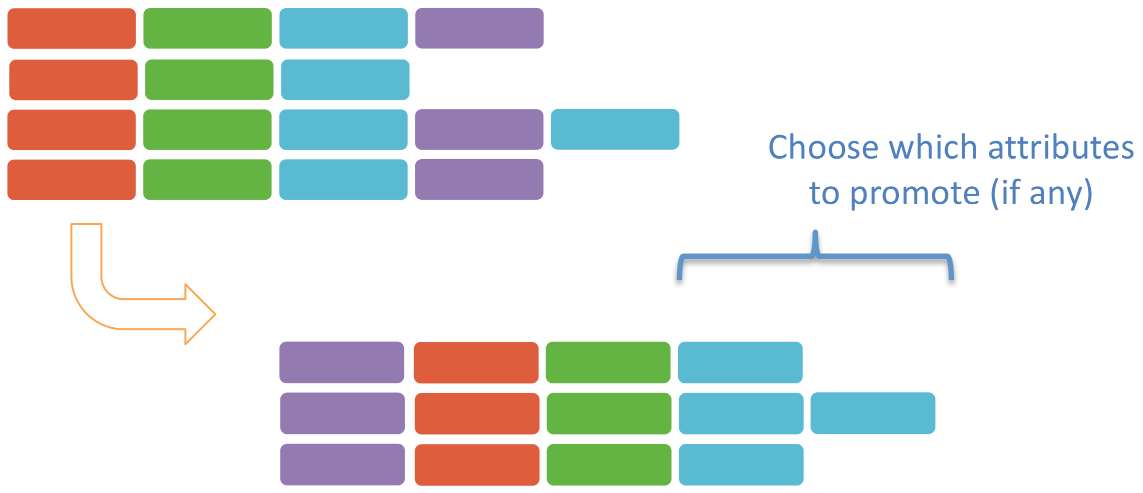

Figure 6: Create a global secondary index on a table

Whenever an item is read from or written to a DynamoDB table or index, the amount of data required to perform the read or write operation is expressed as a "read unit" or "write unit." A read unit consists of 4K of data, and a write unit is 1K. This means that fetching an item of 8K in size will consume 2 read units of data. Inserting the item would consume 8 write units of data. The number of read and write units allowed per second is known as the "provisioned IO" of the table. If your application requires that 1000 4K items be written per second, then the provisioned write capacity of the table would need to be a minimum of 4000 write units per second. 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 IO requirements when designing the tables. However, both read and write capacity can be altered on an existing table, and if an application suddenly experiences a spike in usage that results in throttling, the provisioned IO can be increased to handle the new workload. Similarly, if load decreases for some reason, the provisioned IO can be reduced. This ability to dynamically alter the IO characteristics of a table is a key differentiator between DynamoDB and a traditional RDBMS, in which IO throughput is going to be fixed based on the underlying hardware the database engine is running on.

Migrating to DynamoDB from RDBMS

In the previous section, we discussed some of the key features of DynamoDB, as well as some of the key differences between DynamoDB and a traditional RDBMS. In this section, we will propose a strategy for migrating from an RDBMS to DynamoDB that takes into account these key features and differences. 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 our proposed migration strategy:

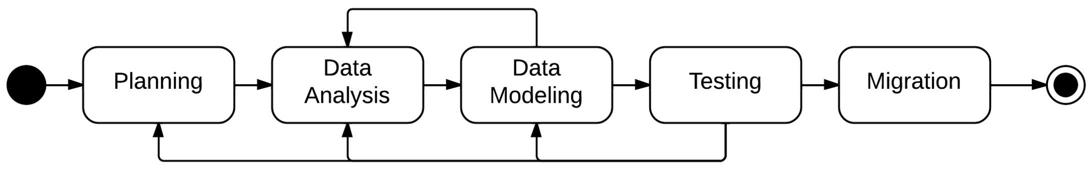

Figure 7: Migration Phases

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 testing. In most cases, it will be necessary to iterate over these phases multiple times before reaching the final data migration state. Each phase will be discussed in detail in the sections that follow.

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

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. As we mentioned previously, tables being used by workloads involving non-relational data make excellent choices for migration to DynamoDB. Migration of such tables to DynamoDB can result in significantly improved application performance, as well as lower costs and lower loads on the RDBMS. 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:

Challenge	Impact on Migration Strategy
Writes to the RDBMS source table cannot be acquiesced before or during the migration.	Synchronization of the data in the target DynamoDB table with the source will be difficult. Consider a migration strategy that involves writing data to both the source and target tables in parallel.
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 service 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	Export the data from the source table and transfer it to S3. Consider using EMR to perform the

Challenge	Impact on Migration Strategy
DynamoDB.	data transformation, and import the transformed data into DynamoDB.
The primary key structure of the source table is not portable to DynamoDB.	Identify column(s) that will make suitable hash and range keys for the imported items. Alternatively, consider adding a surrogate key (such as a UUID) to the source table that will act as a suitable hash key.
The data in the source table is encrypted.	If the encryption is being managed by the RDBMS, then the data will need to be decrypted when exported, and re-encrypted upon import using an encryption scheme enforced by the application, not the underlying database engine. The cryptographic keys will need to be managed outside of DynamoDB.

Table 1: Challenges that Impact Migration Strategy

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 into 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 hash or range keys

DynamoDB pricing contains two main components – storage and provisioned IO. 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 provisioned IO requirements for the table can be calculated. Common SQL data types will map to one of 3 scalar types in DynamoDB: string, number, and 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 provisioned IOPS are given in integral 1K increments—there is no concept of a fractional IO in DynamoDB. If an item is estimated to be 3.3K in size, it will require 4 1K write IO units and 1 4K read IO 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 provisioned IO 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.

At a minimum, a table should have enough provisioned read and write units to read and write a single item per second. For example, if 4 write units are required to write an item with a size equal to or less than the 95th percentile, than the table should have a minimum provisioned IO of 4 write units per second. Anything less than this and the write of a single item will cause throttling and degrade performance. In practice, the number of provisioned read and write units will be much larger than the required minimum. An application using DynamoDB for data storage will typically need to issue read and writes concurrently.

Correctly estimating the provisioned IO 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 hash or range keys is essential for obtaining maximum performance as well. Columns containing values that are not uniformly distributed (i.e. some values occur in much larger numbers than others) are not good hash or range 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 a generic query language like SQL, it is essential to document that 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 values – This is seen frequently with temporal data.

As we will see 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, hash and range keys, and secondary indexes will be 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 hash and range keys for a table. The primary key, whether consisting only of the hash key or a composite of the hash and range 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 hash key. But in reality, this key is often semantically 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 hash key in the DynamoDB table, as items can easily be fetched by their hash key values. Modeling the hash 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 hash 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 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 hash key and a range 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 denormalizing the data (discussed in more detail below). Date values are also often used as range keys. A table counting the number of times a URL was visited on any given day could define the URL as the hash key and the date as the range key. As with primary keys consisting solely of a hash key, fetching items with a composite primary key requires the application to specify both the hash and range key values. This needs to be considered when evaluating whether a surrogate key or a natural key would make the better choice for the hash and or range key.

Because non-key attributes can be added to an item arbitrarily, the only attributes that must be specified in a DynamoDB table definition are the hash key and (optionally) the range key. 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. The query interface exposed by the DynamoDB API does not support fetching items from a table in this way. It is possible to do a full table scan, 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 on 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[2], so it is important to choose define 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 will often require denormalizing the data. To those used to working with an RDBMS, this will be a foreign and perhaps uncomfortable concept. Since the workloads most suitable for migrating to DynamoDB from an RDMBS tend to involve non-relational data, denormalization rarely poses the same issues as it would in a relational data model. 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. But 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.

[2] http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Limits.html

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 item size and the required amount of provisioned IO for a DynamoDB table. Figure 8 contains an ER diagram for a schema used to track events when processing orders placed online through an e-commerce portal. Both the RDBMS and DynamoDB table structures are shown.

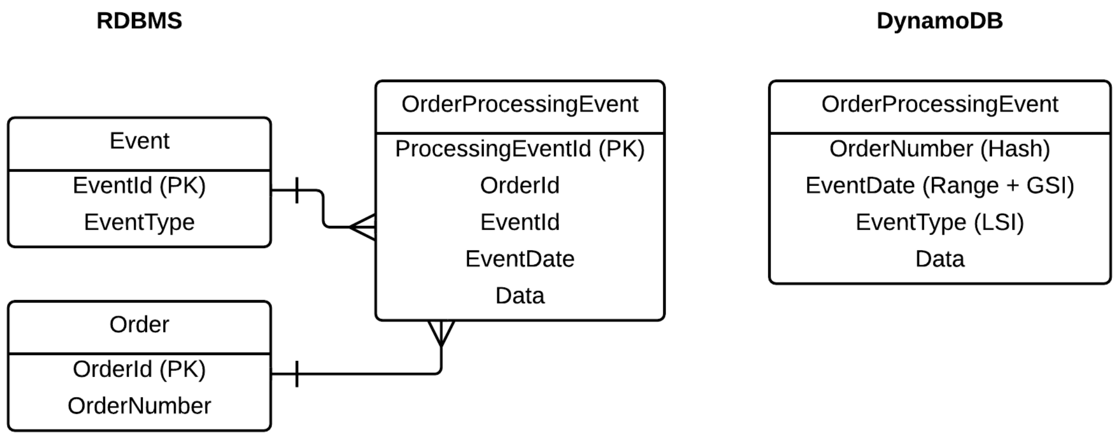

Figure 8: RDBMS and DynamoDB schema for tracking events

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

This statistical sampling yields a 95th percentile size of 6.6 KB, most of which is consumed by the "Data" attribute (which can be as large as 6KB in size). The minimum number of write units required to write a single item is:

$$ceiling(6.6KB\ per\ item/1KB\ per\ write\ unit) = 7\ write\ units\ per\ item$$

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

$$ceiling(6.6KB\ per\ item/4Kb\ per\ read\ unit) = 2\ read\ units\ per\ item$$

This particular workload is write-heavy, and we need enough IO to write 1000 events for 500 orders per day. This is computed as follows:

$$500\ orders\ per\ day \times 1000\ events\ per\ order = \ 5 \times 10^5\ events\ per\ day$$

$$5 \times 10^5 events\ per\ day \times 86400\ seconds\ per\ day\ = 5.78\ events\ per\ second$$

$$ceiling(5.78\ events\ per\ second \times 7\ write\ units\ per\ item) = 41\ write\ units\ per\ second$$

Reads on the table only happen once per hour, when the previous hour's data is imported into an Amazon Elastic Map Reduce cluster for ETL. This operation uses a query that selects items from a given date range (which is why the EventDate attribute is both a range key and a global secondary index). The number of read units (which will be provisioned on the global secondary index) required to retrieve the results of a query is based on the size of the results returned by the query:

$$5.78\ events\ per\ second \times 3600\ seconds\ per\ hour\ = 20808\ events\ per\ hour$$

$$\frac{20808\ events\ per\ hour \times 6.6KB\ per\ item}{1024KB} = 134.11MB\ per\ hour$$

The maximum amount of data returned in a single query operation is 1MB, so pagination will be required. Each hourly read query will require reading 135 pages of data. For strongly consistent reads, 256 read units are required to read a full page at a time (the number is half as much for eventually consistent reads). So to support this particular workload, 256 read units and 41 write units will be required. From a practical standpoint, the write units would likely be expressed in an even number, like 48. We now have all the data we need to estimate the DynamoDB cost for this workload:

1. Number of items ($10^{8)}$
2. Item size (7KB)
3. Write units (48)
4. Read units (256)

These can be run through the Amazon Simple Monthly Calculator[3] to derive a cost estimate.

[3] http://calculator.s3.amazonaws.com/index.html

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:

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 will 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 will assess the non-functional characteristics of the application, such as performance under varying levels of load, and resiliency to failure of any portion of

Test Category	**Purpose**
	the application stack. These tests can also 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 application(s) once the final data migration has completed. The purpose of these tests is for the end-users to decide if the application is sufficiently usable to meet it's primary function in the organization.

Table 2: Data Migration Test Plan

Because the migration strategy is iterative, these tests will be executed numerous 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 will often require revisiting a previous phase in the process. The overall migration strategy will become more refined through each iteration through the process, and once all the tests have executed successfully, it will be a good indication that 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 will be migrated into DynamoDB. By the time this phase is reached, the end-to-end data migration process will have been tested and vetted thoroughly. All the steps of the process will have been 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. In preparation for this final phase, a notification should be sent to the application users alerting them that the application will be undergoing maintenance and (if required) downtime.

Once the data migration has completed, 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 will also have been thoroughly tested and vetted at this point—can be executed. 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 has been 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

Leveraging 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, we 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, we are confident that the ROI of migrating to a fully managed NoSQL solution like DynamoDB will greatly exceed the upfront cost associated with the effort.

Cheat Sheet

The following is a “cheat sheet” summarizing 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*

Further Reading

For additional help, please consult the following sources:

- DynamoDB Developer Guide[4]
- DynamoDB Website[5]

[4] http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/GettingStartedDynamoDB.html

[5] http://aws.amazon.com/dynamodb