Comparing the Use of Amazon DynamoDB and Apache HBase for NoSQL

January 2020

Notices

Contents

Abstract

One challenge that architects and developers face today is how to process large volumes of data in a timely, cost effective, and reliable manner. There are several NoSQL solutions in the market and choosing the most appropriate one for your particular use case can be difficult. This paper compares two popular NoSQL data stores—Amazon DynamoDB, a fully managed NoSQL cloud database service, and Apache HBase, an open-source, column-oriented, distributed big data store. Both Amazon DynamoDB and Apache HBase are available in the Amazon Web Services (AWS) Cloud.

Introduction

The AWS Cloud accelerates big data analytics. With access to instant scalability and elasticity on AWS, you can focus on analytics instead of infrastructure. Whether you are indexing large data sets, analyzing massive amounts of scientific data, or processing clickstream logs, AWS provides a range of big data products and services that you can leverage for virtually any data-intensive project.

There is a wide adoption of NoSQL databases in the growing industry of big data and real-time web applications. Amazon DynamoDB and Apache HBase are examples of NoSQL databases, which are highly optimized to yield significant performance benefits over a traditional relational database management system (RDBMS). Both Amazon DynamoDB and Apache HBase can process large volumes of data with high performance and throughput.

Amazon DynamoDB provides a fast, fully managed NoSQL database service. It lets you offload operating and scaling a highly available, distributed database cluster. Apache HBase is an open-source, column-oriented, distributed big data store that runs on the Apache Hadoop framework and is typically deployed on top of the Hadoop Distributed File System (HDFS), which provides a scalable, persistent, storage layer.

In the AWS Cloud, you can choose to deploy Apache HBase on Amazon Elastic Compute Cloud (Amazon EC2) and manage it yourself. Alternatively, you can leverage Apache HBase as a managed service on Amazon EMR, a fully managed, hosted Hadoop framework on top of Amazon EC2.

> With Apache HBase on Amazon EMR, you can use Amazon Simple Storage Service (Amazon S3) as a data store using the EMR File System (EMRFS), an implementation of HDFS that all Amazon EMR clusters use for reading and writing regular files from Amazon EMR directly to Amazon S3.

The following figure shows the relationship between Amazon DynamoDB, Amazon EC2, Amazon EMR, Amazon S3, and Apache HBase in the AWS Cloud. Both Amazon DynamoDB and Apache HBase have tight integration with popular open source processing frameworks like Apache Hive and Apache Spark to enhance querying capabilities as illustrated in the diagram.

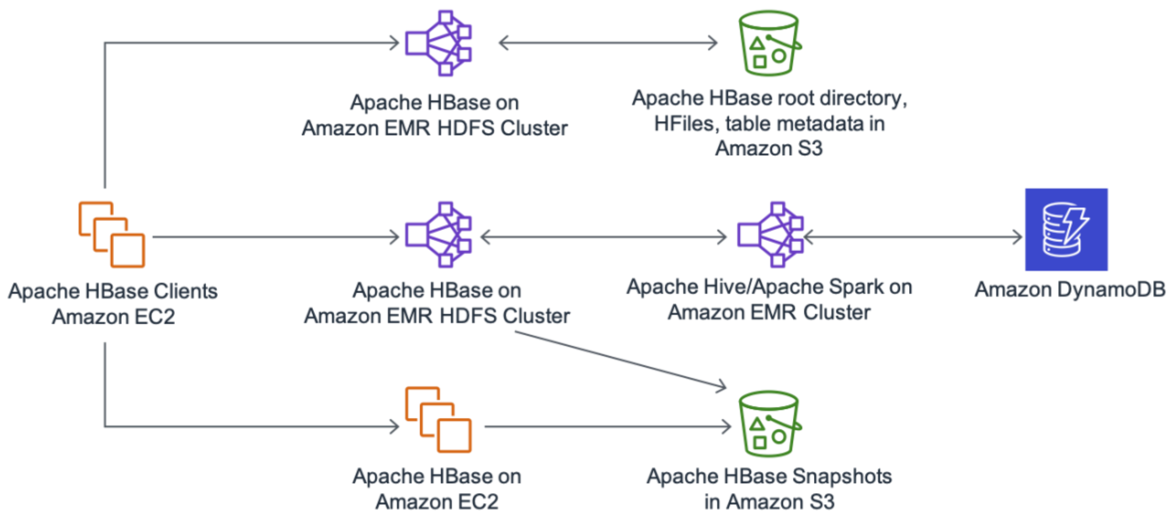

Figure 1: Relation between Amazon DynamoDB, Amazon EC2, Amazon EMR, and Apache HBase in the AWS Cloud

Amazon DynamoDB Overview

Amazon DynamoDB is a fully managed NoSQL database service that provides fast and predictable performance with seamless scalability. Amazon DynamoDB offers the following benefits:

- **Zero administrative overhead—**Amazon DynamoDB manages the burdens of hardware provisioning, setup and configuration, replication, cluster scaling, hardware and software updates, and monitoring and handling of hardware failures.
- **Virtually unlimited throughput and scale—**The provisioned throughput model of Amazon DynamoDB allows you to specify throughput capacity to serve nearly any level of request traffic. With Amazon DynamoDB, there is virtually no limit to the amount of data that can be stored and retrieved.
- **Elasticity and flexibility—**Amazon DynamoDB can handle unpredictable workloads with predictable performance and still maintain a stable latency profile that shows no latency increase or throughput decrease as the data volume rises with increased usage. Amazon DynamoDB lets you increase or decrease capacity as needed to handle variable workloads.

- **Automatic scaling—** Amazon DynamoDB can scale automatically within user-defined lower and upper bounds for read and write capacity in response to changes in application traffic. These qualities render Amazon DynamoDB a suitable choice for online applications with spiky traffic patterns or the potential to go viral anytime.
- **Integration with other AWS services**—Amazon DynamoDB integrates seamlessly with other AWS services for logging and monitoring, security, analytics, and more. For more information, see the Amazon DynamoDB Developer Guide.

Apache HBase Overview

Apache HBase, a Hadoop NoSQL database, offers the following benefits:

- **Efficient storage of sparse data**—Apache HBase provides fault-tolerant storage for large quantities of sparse data using column-based compression. Apache HBase is capable of storing and processing billions of rows and millions of columns per row.
- **Store for high frequency counters**—Apache HBase is suitable for tasks such as high-speed counter aggregation because of its consistent reads and writes.
- **High write throughput and update rates**—Apache HBase supports low latency lookups and range scans, efficient updates and deletions of individual records, and high write throughput.
- **Support for multiple Hadoop jobs**—The Apache HBase data store allows data to be used by one or more Hadoop jobs on a single cluster or across multiple Hadoop clusters.

Apache HBase Deployment Options

The following section provides a description of Apache HBase deployment options in the AWS Cloud.

Managed Apache HBase on Amazon EMR (Amazon S3 Storage Mode)

Amazon EMR enables you to use Amazon S3 as a data store for Apache HBase using the EMR File System and offers the following benefits:

- **Separation of compute from storage—** You can size your Amazon EMR cluster for compute instead of data requirements, allowing you to avoid the need for the customary 3x replication in HDFS.
- **Transient clusters—**You can scale compute nodes without impacting your underlying storage and terminate your cluster to save costs and quickly restore it.
- **Built-in availability and durability—**You get the availability and durability of Amazon S3 storage by default.
- **Easy to provision read replicas—**You can create and configure a read-replica cluster in another Amazon EC2 Availability Zone that provides read-only access to the same data as the primary cluster, ensuring uninterrupted access to your data even if the primary cluster becomes unavailable.

Managed Apache HBase on Amazon EMR (HDFS Storage Mode)

Apache HBase on Amazon EMR is optimized to run on AWS and offers the following benefits:

- **Minimal administrative overhead—**Amazon EMR handles provisioning of Amazon EC2 instances, security settings, Apache HBase configuration, log collection, health monitoring, and replacement of faulty instances. You still have the flexibility to access the underlying infrastructure and customize Apache HBase further, if desired.
- **Easy and flexible deployment options—**You can deploy Apache HBase on Amazon EMR using the AWS Management Console or by using the AWS Command Line Interface (AWS CLI). Once launched, resizing an Apache HBase cluster is easily accomplished with a single API call. Activities such as modifying the Apache HBase configuration at launch time or installing third-party tools such as Ganglia for monitoring performance metrics are feasible with custom or predefined scripts.

- **Unlimited scale—**With Apache HBase running on Amazon EMR, you can gain significant cloud benefits such as easy scaling, low cost, pay only for what you use, and ease of use as opposed to the self-managed deployment model on Amazon EC2.
- **Integration with other AWS services**—Amazon EMR is designed to seamlessly integrate with other AWS services, such as Amazon S3, Amazon DynamoDB, Amazon EC2, and Amazon CloudWatch.
- **Built-in backup feature—**A key benefit of Apache HBase running on Amazon EMR is the built-in mechanism available for backing up Apache HBase data durably in Amazon S3. Using this feature, you can schedule full or incremental backups, and roll back or even restore backups to existing or newly launched clusters anytime.

Self-Managed Apache HBase Deployment Model on Amazon EC2

The Apache HBase self-managed model offers the most flexibility in terms of cluster management, but also presents the following challenges:

- **Administrative overhead**—You must deal with the administrative burden of provisioning and managing your Apache HBase clusters.
- **Capacity planning—**As with any traditional infrastructure, capacity planning is difficult and often prone to significant costly error. For example, you could over-invest and end up paying for unused capacity or under-invest and risk performance or availability issues.
- **Memory management—**Apache HBase is mainly memory-driven. Memory can become a limiting factor as the cluster grows. It is important to determine how much memory is needed to run diverse applications on your Apache HBase cluster to prevent nodes from swapping data too often to the disk. The number of Apache HBase nodes and memory requirements should be planned well in advance.
- **Compute, storage, and network planning—**Other key considerations for effectively operating an Apache HBase cluster include compute, storage, and network. These infrastructure components often require dedicated Apache Hadoop/Apache HBase administrators with specialized skills.

Feature Summary

Amazon DynamoDB and Apache HBase both possess characteristics that are critical for successfully processing massive amounts of data. The following table provides a summary of key features of Amazon DynamoDB and Apache HBase that can help you understand key similarities and differences between the two databases. These features are discussed in later sections.

Table 1: Amazon DynamoDB and Apache HBase Feature Summary

Feature	Amazon DynamoDB	Apache HBase
Description	Hosted, scalable database service by Amazon	Column store based on Apache Hadoop and on concepts of BigTable
Implementation Language	-	Java
Server Operating Systems	Hosted	Linux, Unix, Windows
Database Model	Key-value & Document store	Wide column store
Data Scheme	Schema free	Schema free
Typing	Yes	No
APIs and Other Access Methods	Flexible	Flexible
Supported Programming Languages	Multiple	Multiple
Server-side Scripts	No	Yes
Triggers	Yes	Yes
Partitioning Methods	Sharding	Sharding
Throughput Model	User provisions throughput	Limited to hardware configuration
Automatic Scaling	Yes	No
Partitioning	Automatic partitioning	Automatic sharding
Replication	Yes	Yes

Feature	Amazon DynamoDB	Apache HBase
Durability	Yes	Yes
Administration	No administration overhead	High administration overhead in self-managed and minimal on Amazon EMR
User Concepts	Yes	Yes
Data Model		
Row	Item – 1 or more attributes	Columns/column families
Row Size	Item size restriction	No row size restrictions
Primary Key	Simple/Composite	Row key
Foreign Key	No	No
Indexes	Optional	No built-in index model implemented as secondary tables or coprocessors
Transactions		
Row Transactions	Item-level transactions	Single-row transactions
Multi-row Transactions	Yes	Yes
Cross-table Transactions	Yes	Yes
Consistency Model	Eventually consistent and strongly consistent reads	Strongly consistent reads and writes
Concurrency	Yes	Yes
Updates	Conditional updates	Atomic read-modify-write
Integrated Cache	Yes	Yes
Time-To-Live (TTL)	Yes	Yes
Encryption at Rest	Yes	Yes
Backup and Restore	Yes	Yes
Point-in-time Recovery	Yes	Yes

Feature	Amazon DynamoDB	Apache HBase
Multiregion, Multi-master	Yes	No

Use Cases

Amazon DynamoDB and Apache HBase are optimized to process massive amounts of data. Popular use cases for Amazon DynamoDB and Apache HBase include the following:

- **Serverless applications**—Amazon DynamoDB provides a durable backend for storing data at any scale and has become the de facto database for powering Web and mobile backends for e-commerce/retail, education, and media verticals.
- **High volume special events**—Special events and seasonal events, such as national electoral campaigns, are of relatively short duration and have variable workloads with the potential to consume large amounts of resources. Amazon DynamoDB lets you increase capacity when you need it and decrease as needed to handle variable workloads. This quality renders Amazon DynamoDB a suitable choice for such high volume special events.
- **Social media applications**—Community-based applications, such as online gaming, photo sharing, location-aware applications, and so on, have unpredictable usage patterns with the potential to go viral anytime. The elasticity and flexibility of Amazon DynamoDB make it suitable for such high volume, variable workloads.
- **Regulatory and compliance requirements**—Both Amazon DynamoDB and Amazon EMR are in scope of the AWS compliance efforts and therefore suitable for healthcare and financial services workloads as described in AWS Services in Scope by Compliance Program.
- **Batch-oriented processing**—For large datasets, such as log data, weather data, product catalogs, and so on, you may already have large amounts of historical data that you want to maintain for historical trend analysis but need to ingest and batch process current data for predictive purposes. For these types of workloads, Apache HBase is a good choice because of its high read and write throughput and efficient storage of sparse data.

- **Reporting**—To process and report on high volume transactional data, such as daily stock market trades, Apache HBase is a good choice because it supports high throughput writes and update rates, which make it suitable for storage of high frequency counters and complex aggregations.
- **Real-time analytics**—The payload or message size in event data, such as tweets, E-commerce, and so on, is relatively small when compared with application logs. If you want to ingest streaming event data in real-time for sentiment analysis, ad serving, trending analysis, and so on, Amazon DynamoDB lets you increase throughout capacity when you need it, and decrease it when you are done, with no downtime. Apache HBase can handle real-time ingestion of data, such as application logs, with ease due to its high write throughput and efficient storage of sparse data. Combining this capability with Hadoop's ability to handle sequential reads and scans in a highly optimized way renders Apache HBase a powerful tool for real-time data analytics.

Data Models

Amazon DynamoDB is a key/value as well as a document store and Apache HBase is a key/value store. For a meaningful comparison of Amazon DynamoDB with Apache HBase as a NoSQL data store, this document focuses on the key/value data model for Amazon DynamoDB.

Amazon DynamoDB and Apache HBase are designed with the goal to deliver significant performance benefits with low latency and high throughput. To achieve this goal, key/value stores and document stores have simpler and less constrained data models than traditional relational databases. Although the fundamental data model building-blocks are similar in both Amazon DynamoDB and Apache HBase, each database uses a distinct terminology to describe its specific data model.

At a high level, a database is a collection of tables, and each table is a collection of rows. A row can contain one or more columns. In most cases, NoSQL database tables typically do not require a formal schema except for a mandatory primary key that uniquely identifies each row. The following table illustrates the high-level concept of a NoSQL database.

Table 2: High-Level NoSQL Database Table Representation

Table		
Row	Primary Key	Column 1

Columnar databases are devised to store each column separately so that aggregate operations for one column of the entire table are significantly quicker than the traditional row storage model.

From a comparative standpoint, a row in Amazon DynamoDB is referred to as an *item*, and each *item* can have any number of *attributes.* An attribute comprises a key and a value and commonly referred to as a name-value pair. An Amazon DynamoDB table can have unlimited items indexed by primary key, as shown in the following example.

Table 3: High-Level Representation of Amazon DynamoDB Table

Table					
Item 1	Primary Key	Attribute 1	Attribute 2	Attribute 3	Attribute …n
Item 2	Primary Key	Attribute 1		Attribute 3	
Item n	Primary Key		Attribute 2	Attribute 3	

Amazon DynamoDB defines two types of primary keys: a simple primary key with one attribute called a partition key (*Table 4*) and a composite primary key with two attributes (*Table 5*).

Table 4: Amazon DynamoDB Simple Primary Key (Partition Key)

Table					
Item	Partition Key	Attribute 1	Attribute 2	Attribute 3	Attribute …n

Table 5: Amazon DynamoDB Composite Primary Key (Partition & Sort Key)

Table						
Item	Partition Key	Sort Key	Attribute 1	Attribute 2	Attribute 3	attribute …n

A JSON representation of the item in the Table 5 with additional nested attributes is given below:

```
{
       "Partition Key": "Value",
       "Sort Key": "Value",
       "Attribute 1": "Value",
       "Attribute 2": "Value",
       "Attribute 3": [
        {
             "Attribute 4": "Value",
             "Attribute 5": "Value",
        },
        {
             "Attribute 4": "Value",
             "Attribute 5": "Value",
        }
        ]
}
```

In Amazon DynamoDB, a single attribute primary key or *partition* key is useful for quick reads and writes of data. For example, `PersonID` serves as the partition key in the following `Person` table.

Table 6: Example Person Amazon DynamoDB Table

Person Table					
Item	PersonId (Partition Key)	FirstName	LastName	Zipcode	Gender
Item 1	1001	Fname-1	Lname-1	00000	
Item 2	1002	Fname-2	Lname-2		M
Item 3	2002	Fname-3	Lname-3	10000	F

A composite key in Amazon DynamoDB is indexed as a *partition* key and a *sort* key. This multi-part key maintains a hierarchy between the first and second element values. Holding the partition key element constant facilitates searches across the sort key element to retrieve items quickly for a given partition key. In the following `GameScores` table, the composite partition-sort key is a combination of `PersonId` (partition key) and `GameId` (sort key).

Table 7: Example GameScores Amazon DynamoDB Table

GameScores Table						
	PersonId (Partition Key)	GameId (Sort Key)	TopScore	TopScoreDate	Wins	Losses
item1	1001	Game01	67453	2013-12-09:17:24:31	73	21
item2	1001	Game02	98567	2013-12-11:14:14:37	98	27
Item3	1002	Game01	43876	2013-12-15:19:24:39	12	23
Item4	2002	Game02	65689	2013-10-01:17:14:41	23	54

The partition key of an item is also known as its hash attribute and sort key as its range attribute. The term hash attribute arises from the use of an internal hash function that takes the value of the partition key as input and the output of that hash function determines the partition or physical storage node where the item will be stored. The term range attribute derives from the way DynamoDB stores items with the same partition key together, in sorted order by the sort key value.

Although there is no explicit limit on the number of attributes associated with an individual item in an Amazon DynamoDB table, there are restrictions on the aggregate size of an item or payload, including all attribute names and values. A small payload can potentially improve performance and reduce costs because it requires fewer resources to process. For information on how to handle items that exceed the maximum item size, see Best Practices for Storing Large Items and Attributes.

In Apache HBase, the most basic unit is a column. One or more columns form a row. Each row is addressed uniquely by a primary key referred to as a *row key*. A row in Apache HBase can have millions of columns. Each column can have multiple versions with each distinct value contained in a separate *cell*.

One fundamental modeling concept in Apache HBase is that of a *column family*. A column family is a container for grouping sets of related data together within one table, as shown in the following example.

Table 8: Apache HBase Row Representation

Table							
		Column Family 1		Column Family 2		Column Family 3	
row	row key	Column 1	Column 2	Column 3	Column 4	Column 5	Column 6

Apache HBase groups columns with the same general access patterns and size characteristics into column families to form a basic unit of separation. For example, in the following `Person` table, you can group personal data into one column family called `personal_info` and the statistical data into a `demographic` column family. Any other columns in the table would be grouped accordingly as well, as shown in the following example.

Table 9: Example Person Table in Apache HBase

Person Table					
		personal_info		demographic	
	row key	firstname	lastname	zipcode	gender
row 1	1001	Fname-1	Lname-1	00000	
row 2	1002	Fname-2	Lname-2		M
row 3	2002	Fname-3	Lname-3	10000	F

Columns are addressed as a combination of the column family name and the column qualifier expressed as `family:qualifier`. All members of a column family have the same prefix. In the preceding example, the `firstname` and `lastname` column qualifiers can be referenced as `personal_info:firstname` and `personal_info:lastname`, respectively.

Column families allow you to fetch only those columns that are required by a query. All members of a column family are physically stored together on a disk. This means that optimization features, such as performance tunings, compression encodings, and so on, can be scoped at the column family level.

The row key is a combination of user and game identifiers in the following Apache HBase `GameScores` table. A row key can consist of multiple parts concatenated to provide an immutable way of referring to entities. From an Apache HBase modeling perspective, the resulting table is *tall-narrow.* This is because the table has few columns relative to the number of rows, as shown in the following example.

Table 10: Tall-Narrow GameScores Apache HBase Table

GameScores Table					
		top_scores		metrics	
	row key	score	date	wins	loses
row 1	1001-game01	67453	2013-12-09:17:24:31	73	21
row 2	1001-game02	98567	2013-12-11:14:14:37	98	27
row 3	1002-game01	43876	2013-12-15:19:24:39	12	23
row 4	2002-game02	65689	2013-10-01:17:14:41	23	54

Alternatively, you can model the game identifier as a column qualifier in Apache HBase. This approach facilitates precise column lookups and supports usage of filters to read data. The result is a *flat-wide* table with few rows relative to the number of columns. This concept of a flat-wide Apache HBase table is shown in the following table.

Table 11: Flat-Wide GameScores Apache HBase Table

GameScores Table							
		top_scores			metrics		
	row key	gameId	score	top_score_date	gameId	wins	loses
row 1	1001	game01	98567	2013-12-11:14:14:37	game01	98	27
		game02	43876	2013-12-15:19:24:39	game02	12	23

GameScores Table							
row 2	1002	game01	67453	2013-12-09:17:24:31	game01	73	21
row 3	2002	game02	65689	2013-10-01:17:14:41	game02	23	54

For performance reasons, it is important to keep the number of column families in your Apache HBase schema low. Anything above three-column families can potentially degrade performance. The recommended best practice is to maintain a one-column family in your schemas and introduce a two-column family and three-column family only if data access is limited to a one-column family at a time. Note that Apache HBase does not impose any restrictions on row size.

Data Types

Both Amazon DynamoDB and Apache HBase support unstructured datasets with a wide range of data types.

Amazon DynamoDB supports the data types shown in the following table:

Table 12: Amazon DynamoDB Data Types

Type	Description	Example (JSON Format)
Scalar		
String	Unicode with UTF8 binary encoding	{"S": "Game01"}
Number	Positive or negative exact- value decimals and integers	{"N": "67453"}
Binary	Encoded sequence of bytes	{"B": "dGhpcyB0ZXh0IGlzIGJhc2U2NC1l"}
Boolean	True or false	{"BOOL": true}
Null	Unknown or undefined state	{"NULL": true}
Document		
List	Ordered collection of values	{"L": ["Game01", 67453]}

Type	Description	Example (JSON Format)
Map	Unordered collection of name-value pairs	{"M": {"GameId": {"S": "Game01"}, "TopScore": {"N": "67453"}}}
Multi-valued		
String Set	Unique set of strings	{"SS": ["Black","Green] }
Number Set	Unique set of numbers	{"NS": ["42.2","-19.87"] }
Binary Set	Unique set of binary values	{"BS": ["U3Vubnk=","UmFpbnk=] }

Each Amazon DynamoDB attribute can be a name-value pair with exactly one value (scalar type), a complex data structure with nested attributes (document type), or a unique set of values (multi-valued set type). Individual items in an Amazon DynamoDB table can have any number of attributes.

Primary key attributes can only be scalar types with a single value and the only data types allowed are string, number, or binary. Binary type attributes can store any binary data, for example, compressed data, encrypted data, or even images.

Map is ideal for storing JSON documents in Amazon DynamoDB. For example, in *Table 6*, `Person` could be represented as a map of person id that maps to detailed information about the person: name, gender, and a list of their previous addresses also represented as a map. This is illustrated in the following script:

```
{
      "PersonId": 1001,
      "FirstName": "Fname-1",
      "LastName": "Lname-1",
      "Gender": "M",
      "Addresses": [
       {
            "Street": "Main St",
            "City": "Seattle",
            "Zipcode": 98005,
            "Type": "current",
       },
       {
           "Street": "9th St",
           "City": Seattle,
           "Zipcode": 98005,
           "Type": "past",
            }
        ]
}
```

In summary, Apache HBase defines the following concepts:

- **Row**—An atomic byte array or key/value container.
- **Column—**A key within the key/value container inside a row.
- **Column Family—**Divides columns into related subsets of data that are stored together on disk.
- **Timestamp—**Apache HBase adds the concept of a fourth dimension column that is expressed as an explicit or implicit timestamp. A timestamp is usually represented as a long integer in milliseconds.
- **Value—**A time-versioned value in the key/value container. This means that a cell can contain multiple versions of a value that can change over time. Versions are stored in decreasing timestamp, with the most recent first.

Apache HBase supports a *bytes-in/bytes-out* interface. This means that anything that can be converted into an array of bytes can be stored as a value. Input could be strings, numbers, complex objects, or even images as long as they can be rendered as bytes.

Consequently, key/value pairs in Apache HBase are arbitrary arrays of bytes. Because row keys and column qualifiers are also arbitrary arrays of bytes, almost anything can serve as a row key or column qualifier, from strings to binary representations of longs or even serialized data structures.

Column family names *must* comprise printable characters in human-readable format. This is because column family names are used as part of the directory name in the file system. Furthermore, column families must be declared up front at the time of schema definition. Column qualifiers are not subjected to this restriction and can comprise any arbitrary binary characters and be created at runtime.

Indexing

In general, data is indexed using a primary key for fast retrieval in both Amazon DynamoDB and Apache HBase. Secondary indexes extend the basic indexing functionality and provide an alternate query path in addition to queries against the primary key.

Amazon DynamoDB supports two kinds of secondary indexes on a table that already implements a partition and sort key:

- **Global secondary index**—An index with a partition and optional sort key that can be different from those on the table.
- **Local secondary index**—An index that has the same partition key as the table, but a different sort key.

You can define one or more global secondary indexes and one or more local secondary indexes per table. For documents, you can create a local secondary index or global secondary index on any top-level JSON element.

In the example `GameScores` table introduced in the preceding section, you can define `LeaderBoardIndex` as a global secondary index as follows:

Table 13: Example Global Secondary Index in Amazon DynamoDB

LeaderBoardIndex		
Index Key		Attribute 1
GameId (Partition Key)	TopScore (Sort Key)	PersonId
Game01	98567	1001
Game02	43876	1001
Game01	65689	1002
Game02	67453	2002

The `LeaderBoardIndex` shown in *Table 13* defines `GameId` as its primary key and `TopScore` as its sort key. It is not necessary for the index key to contain any of the key attributes from the source table. However, the table's primary key attributes are always present in the global secondary index. In this example, `PersonId` is automatically projected or copied into the index.

With `LeaderBoardIndex` defined, you can easily obtain a list of top scores for a specific game by simply querying it. The output is ordered by `TopScore`, the sort key. You can choose to project additional attributes from the source table into the index.

A local secondary index, on the other hand, organizes data by the index sort key. It provides an alternate query path for efficiently accessing data using a different sort key.

You can define `PersonTopScoresIndex` as a local secondary index for the example `GameScores` table introduced in the preceding section. The index contains the same partition key, `PersonId`, as the source table and defines `TopScoreDate` as its new sort key. The old sort key value from the source table (in this example, `GameId`) is automatically projected or copied into the index, but it is not a part of the index key, as shown in the following table.

Table 14: Local Secondary Index in Amazon DynamoDB

PersonTopScoresIndex			
Index Key		Attribute1	Attribute2
PersonId (Partition Key)	TopScoreDate (New Sort Key)	GameId (Old Sort Key as attribute)	TopScore (Optional projected attribute)
1001	2013-12-09:17:24:31	Game01	67453
1001	2013-12-11:14:14:37	Game02	98567
1002	2013-12-15:19:24:39	Game01	43876
2002	2013-10-01:17:14:41	Game02	65689

A local secondary index is a sparse index. An index will only have an item if the index sort key attribute has a value.

With local secondary indexes, any group of items that have the same partition key value in a table and all their associated local secondary indexes form an *item collection*. There is a size restriction on item collections in a DynamoDB table. For more information, see Item Collection Size Limit.

The main difference between a global secondary index and a local secondary index is that a global secondary index defines a completely new partition key and optional sort index on a table. You can define any attribute as the partition key for the global secondary index as long as its data type is scalar rather than a multi-value set.

Additional highlights between global and local secondary indexes are captured in the following table.

Table 15: Global and secondary indexes

	Global Secondary Indexes	Local Secondary Indexes
Creation	Can be created for existing tables (Online indexing supported)	Only at table creation time (Online indexing not supported)
Primary Key Values	Need not be unique	Must be unique
Partition Key	Different from primary table	Same as primary table
Sort Key	Optional	Required (different from Primary table)
Provisioned Throughput	Independent from primary table	Dependent on primary table
Writes	Asynchronous	Synchronous

For more information on global and local secondary indexes in Amazon DynamoDB, see Improving Data Access with Secondary Indexes.

In Apache HBase, all rows are always sorted lexicographically by row key. The sort is byte-ordered. This means that each row key is compared on a binary level, byte by byte, from left to right. Row keys are always unique and act as the primary index in Apache HBase.

Although Apache HBase does not have native support for built-in indexing models such as Amazon DynamoDB, you can implement custom secondary indexes to serve as alternate query paths by using these techniques:

- **Create an index in another table**—You can maintain a secondary table that is periodically updated. However, depending on the load strategy, the risk with this method is that the secondary index can potentially become out of sync with the main table. You can mitigate this risk if you build the secondary index while publishing data to the cluster and perform concurrent writes into the index table.
- **Use the coprocessor framework**—You can leverage the coprocessor framework to implement custom secondary indexes. Coprocessors act like triggers that are similar to stored procedures in RDBMS.

- **Use Apache Phoenix**—Acts as a front-end to Apache HBase to convert standard SQL into native HBase scans and queries and for secondary indexing.

In summary, both Amazon DynamoDB and Apache HBase define data models that allow efficient storage of data to optimize query performance. Amazon DynamoDB imposes a restriction on its item size to allow efficient processing and reduce costs. Apache HBase uses the concept of column families to provide data locality for more efficient read operations.

Amazon DynamoDB supports both scalar and multi-valued sets to accommodate a wide range of unstructured datasets. Similarly, Apache HBase stores its key/value pairs as arbitrary arrays of bytes, giving it the flexibility to store any data type.

Amazon DynamoDB supports built-in secondary indexes and automatically updates and synchronizes all indexes with their parent tables. With Apache HBase, you can implement and manage custom secondary indexes yourself.

From a data model perspective, you can choose Amazon DynamoDB if your item size is relatively small. Although Amazon DynamoDB provides a number of options to overcome row size restrictions, Apache HBase is better equipped to handle large complex payloads with minimal restrictions.

Data Processing

This section highlights foundational elements for processing and querying data within Amazon DynamoDB and Apache HBase.

Throughput Model

Amazon DynamoDB uses a provisioned throughput model to process data. With this model, you can specify your read and write capacity needs in terms of number of input operations per second that a table is expected to achieve. During table creation time, Amazon DynamoDB automatically partitions and reserves the appropriate amount of resources to meet your specified throughput requirements.

Automatic scaling for Amazon DynamoDB automates capacity management and eliminates the guesswork involved in provisioning adequate capacity when creating new tables and global secondary indexes. With automatic scaling enabled, you can specify percent target utilization and DynamoDB will scale the provisioned capacity for reads and writes within the bounds to meet the target utilization percent. For more information, see Managing Throughput Capacity Automatically with DynamoDB Auto Scaling.

To decide on the required read and write throughput values for a table without auto scaling feature enabled, consider the following factors:

- **Item size**—The read and write capacity units that you specify are based on a predefined data item size, per read or per write operation. For more information about provisioned throughput data item size restrictions, see Provisioned Throughput in Amazon DynamoDB.
- **Expected read and write request rates**—You must also determine the expected number of read and write operations your application will perform against the table, per second.
- **Consistency**—Whether your application requires strongly consistent or eventually consistent reads is a factor in determining how many read capacity units you need to provision for your table. For more information about consistency and Amazon DynamoDB, see the Consistency Model section in this document.
- **Global secondary indexes**—The provisioned throughput settings of a global secondary index are separate from those of its parent table. Therefore, you must also consider the expected workload on the global secondary index when specifying the read and write capacity at index creation time.
- **Local secondary indexes—**Queries against indexes consume provisioned read throughput. For more information, see Provisioned Throughput Considerations for Local Secondary Indexes.

Although read and write requirements are specified at table creation time, Amazon DynamoDB lets you increase or decrease the provisioned throughput to accommodate load with no downtime.

With Apache HBase, the number of nodes in a cluster can be driven by the required throughput for reads and/or writes. The available throughput on a given node can vary depending on the data, specifically:

- Key/value sizes
- Data access patterns
- Cache hit rates
- Node and system configuration

You should plan for peak load if load will likely be the primary factor that increases node count within an Apache HBase cluster.

Consistency Model

A database consistency model determines the manner and timing in which a successful write or update is reflected in a subsequent read operation of that same value.

Amazon DynamoDB lets you specify the desired consistency characteristics for each read request within an application. You can specify whether a read is eventually consistent or strongly consistent.

The eventual consistency option is the default in Amazon DynamoDB and maximizes the read throughput. However, an eventually consistent read might not always reflect the results of a recently completed write. Consistency across all copies of data is usually reached within a second.

A strongly consistent read in Amazon DynamoDB returns a result that reflects all writes that received a successful response prior to the read. To get a strongly consistent read result, you can specify optional parameters in a request. It takes more resources to process a strongly consistent read than an eventually consistent read. For more information about read consistency, see Data Read and Consistency Considerations.

Apache HBase reads and writes are strongly consistent. This means that all reads and writes to a single row in Apache HBase are atomic. Each concurrent reader and writer can make safe assumptions about the state of a row. Multi-versioning and time stamping in Apache HBase contribute to its strongly consistent model.

Transaction Model

Unlike RDBMS, NoSQL databases typically have no domain-specific language, such as SQL, to query data. Amazon DynamoDB and Apache HBase provide simple application programming interfaces (APIs) to perform the standard create, read, update, and delete (CRUD) operations.

Amazon DynamoDB Transactions support coordinated, all-or-nothing changes to multiple items both within and across tables. Transactions provide atomicity, consistency, isolation, and durability (ACID) in DynamoDB, helping you to maintain data correctness in your applications. Apache HBase integrates with Apache Phoenix to add cross row and cross table transaction support with full ACID semantics.

Amazon DynamoDB provides atomic item and attribute operations for adding, updating, or deleting data. Further, item-level transactions can specify a condition that must be satisfied before that transaction is fulfilled. For example, you can choose to update an item only if it already has a certain value.

Conditional operations allow you to implement optimistic concurrency control systems on Amazon DynamoDB. For conditional updates, Amazon DynamoDB allows atomic increment and decrement operations on existing scalar values without interfering with other write requests. For more information about conditional operations, see Conditional Writes.

Apache HBase also supports atomic high update rates (the classic read-modify-write) within a single row key, enabling storage for high frequency counters. Unlike Amazon DynamoDB, Apache HBase uses multi-version concurrency control to implement updates. This means that an existing piece of data is not overwritten with a new one; instead, it becomes obsolete when a newer version is added.

Row data access in Apache HBase is atomic and includes any number of columns, but there are no further guarantees or transactional features spanning multiple rows. Similar to Amazon DynamoDB, Apache HBase supports only single-row transactions.

Amazon DynamoDB has an optional feature, DynamoDB Streams, to capture table activity. The data modification events such as add, update, or delete can be captured in near real time, in a time-ordered sequence. If stream is enabled on a DynamoDB table, each event gets recorded as a stream record along with name of the table, event timestamp and other metadata. For more information, see the section on Capturing Table Activity with DynamoDB Streams.

Amazon DynamoDB Streams can be used with AWS Lambda to create trigger-code that executes automatically whenever an event of interest (add, update, delete) appears in a stream. This pattern enables powerful solutions, such as data replication within and across AWS Regions, materialized views of data in DynamoDB tables, data analysis using Amazon Kinesis, notifications via Amazon Simple Notification Service (Amazon SNS) or Amazon Simple Email Service (Amazon SES) and much more. For more information, see DynamoDB Streams and AWS Lambda Triggers.

Table Operations

Amazon DynamoDB and Apache HBase provide scan operations to support large-scale analytical processing. A scan operation is similar to cursors in RDBMS. By taking advantage of the underlying sequential, sorted storage layout, a scan operation can easily iterate over wide ranges of records or entire tables. Applying filters to a scan operation can effectively narrow the result set and optimize performance.

Amazon DynamoDB uses parallel scanning to improve performance of a scan operation. A parallel scan logically sub-divides an Amazon DynamoDB table into multiple segments, and then processes each segment in parallel. Rather than using the default scan operation in Apache HBase, you can implement a custom parallel scan by means of the API to read rows in parallel.

Both Amazon DynamoDB and Apache HBase provide a Query API for complex query processing in addition to the scan operation. The Query API in Amazon DynamoDB is accessible only in tables that define a composite primary key. In Apache HBase, bloom filters improve Get operations and the potential performance gain increases with the number of parallel reads.

In summary, Amazon DynamoDB and Apache HBase have similar data processing models in that they both support only atomic single-row transactions. Both databases also provide batch operations for bulk data processing across multiple rows and tables.

One key difference between the two databases is the flexible provisioned throughput model of Amazon DynamoDB. The ability to increase capacity when you need it and decrease it when you are done is useful for processing variable workloads with unpredictable peaks.

For workloads that need high update rates to perform data aggregations or maintain counters, Apache HBase is a good choice. This is because Apache HBase supports a multi-version concurrency control mechanism, which contributes to its strongly consistent reads and writes. Amazon DynamoDB gives you the flexibility to specify whether you want your read request to be eventually consistent or strongly consistent depending on your specific workload.

Architecture

This section summarizes key architectural components of Amazon DynamoDB and Apache HBase.

Amazon DynamoDB Architecture Overview

At a high level, Amazon DynamoDB is designed for high availability, durability, and consistently low latency (typically in the single digit milliseconds) performance.

Amazon DynamoDB runs on a fleet of AWS managed servers that leverage solid state drives (SSDs) to create an optimized, high-density storage platform. This platform decouples performance from table size and eliminates the need for the working set of data to fit in memory while still returning consistent, low latency responses to queries. As a managed service, Amazon DynamoDB abstracts its underlying architectural details from the user.

Apache HBase Architecture Overview

Apache HBase is typically deployed on top of HDFS. Apache ZooKeeper is a critical component for maintaining configuration information and managing the entire Apache HBase cluster.

The three major Apache HBase components are the following:

- **Client API** — Provides programmatic access to Data Manipulation Language (DML) for performing CRUD operations on HBase tables.
- **Region servers** — HBase tables are split into regions and are served by region servers.
- **Master server** — Responsible for monitoring all region server instances in the cluster and is the interface for all metadata changes.

Apache HBase stores data in indexed store files called HFiles on HDFS. The store files are sequences of blocks with a block index stored at the end for fast lookups. The store files provide an API to access specific values as well as to scan ranges of values, given a start and end key.

During a write operation, data is first written to a commit log called a write-ahead-log (WAL) and then moved into memory in a structure called Memstore. When the size of the Memstore exceeds a given maximum value, it is flushed as a HFile to disk. Each time data is flushed from Memstores to disk, new HFiles must be created. As the number of HFiles builds up, a compaction process merges the files into fewer, larger files.

A read operation essentially is a merge of data stored in the Memstores and in the HFiles. The WAL is never used in the read operation. It is meant only for recovery purposes if a server crashes before writing the in-memory data to disk.

A region in Apache HBase acts as a store per column family. Each region contains contiguous ranges of rows stored together. Regions can be merged to reduce the number of store files. A large store file that exceeds the configured maximum store file size can trigger a region split.

A region server can serve multiple regions. Each region is mapped to exactly one region server. Region servers handle reads and writes, as well as keeping data in-memory until enough is collected to warrant a flush. Clients communicate directly with region servers to handle all data-related operations.

The master server is responsible for monitoring and assigning regions to region servers and uses Apache ZooKeeper to facilitate this task. Apache ZooKeeper also serves as a registry for region servers and a bootstrap location for region discovery.

The master server is also responsible for handling critical functions such as load balancing of regions across region servers, region server failover, and completing region splits, but it is not part of the actual data storage or retrieval path.

You can run Apache HBase in a multi-master environment. All masters compete to run the cluster in a multi-master mode. However, if the active master shuts down, then the remaining masters contend to take over the master role.

Apache HBase on Amazon EMR Architecture Overview

Amazon EMR defines the concept of instance groups, which are collections of Amazon EC2 instances. The Amazon EC2 virtual servers perform roles analogous to the master and slave nodes of Hadoop. For best performance, Apache HBase clusters should run on at least two Amazon EC2 instances. There are three types of instance groups in an Amazon EMR cluster.

- **Master**—Contains one master node that manages the cluster. You can use the Secure Shell (SSH) protocol to access the master node if you want to view logs or administer the cluster yourself. The master node runs the Apache HBase master server and Apache ZooKeeper.
- **Core**—Contains one or more core nodes that run HDFS and store data. The core nodes run the Apache HBase region servers.
- **Task**—(Optional). Contains any number of task nodes.

Managed Apache HBase on Amazon EMR (Amazon S3 Storage Mode)

When you run Apache HBase on Amazon EMR with Amazon S3 storage mode enabled, the HBase root directory is stored in Amazon S3, including HBase store files and table metadata. For more information, see HBase on Amazon S3 (Amazon S3 Storage Mode).

For production workloads, EMRFS consistent view is recommended when you enable HBase on Amazon S3. Not using consistent view may result in performance impacts for specific operations.

Partitioning

Amazon DynamoDB stores three geographically distributed replicas of each table to enable high availability and data durability within a region. Data is auto-partitioned primarily using the partition key. As throughput and data size increase, Amazon DynamoDB will automatically repartition and reallocate data across more nodes.

Partitions in Amazon DynamoDB are fully independent, resulting in a shared nothing cluster. However, provisioned throughput is divided evenly across the partitions.

A region is the basic unit of scalability and load balancing in Apache HBase. Region splitting and subsequent load-balancing follow this sequence of events:

1. Initially, there is only one region for a table, and as more data is added to it, the system monitors the load to ensure that the configured maximum size is not exceeded.
2. If the region size exceeds the configured limit, the system dynamically splits the region into two at the row key in the middle of the region, creating two roughly equal halves.
3. The master then schedules the new regions to be moved off to other servers for load balancing, if required.

Behind the scenes, Apache ZooKeeper tracks all activities that take place during a region split and maintains the state of the region in case of server failure. Apache HBase regions are equivalent to range partitions that are used in RDBMS sharding. Regions can be spread across many physical servers that consequently distribute the load, resulting in scalability.

In summary, as a managed service, the architectural details of Amazon DynamoDB are abstracted from you to let you focus on your application details.

With the self-managed Apache HBase deployment model, it is crucial to understand the underlying architectural details to maximize scalability and performance. AWS gives you the option to offload Apache HBase administrative overhead if you opt to launch your cluster on Amazon EMR.

Performance Optimizations

Amazon DynamoDB and Apache HBase are inherently optimized to process large volumes of data with high performance. NoSQL databases typically use an on-disk, column-oriented storage format for fast data access and reduced I/O when fulfilling queries. This performance characteristic is evident in both Amazon DynamoDB and Apache HBase.

Amazon DynamoDB stores items with the same partition key contiguously on disk to optimize fast data retrieval. Similarly, Apache HBase regions contain contiguous ranges of rows stored together to improve read operations. You can enhance performance even further if you apply techniques that maximize throughput at reduced costs, both at the infrastructure and application tiers.

Tip: A recommended best practice is to monitor Amazon DynamoDB and Apache HBase performance metrics to proactively detect and diagnose performance bottlenecks.

The following section focuses on several common performance optimizations that are specific to each database or deployment model.

Amazon DynamoDB Performance Considerations

Performance considerations for Amazon DynamoDB focus on how to define an appropriate read and write throughput and how to design a suitable schema for an application. These performance considerations span both infrastructure level and application tiers.

On-demand Mode – No Capacity Planning

Amazon DynamoDB on-demand is a flexible billing option capable of serving thousands of requests per second without capacity planning. For on-demand mode tables, you don't need to specify how much read and write throughput you expect your application to perform. DynamoDB tables using on-demand capacity mode automatically adapt to your application's traffic volume. On-demand capacity mode instantly accommodates up to double the previous peak traffic on a table. For more information, see On-Demand Mode.

Tip: DynamoDB recommends spacing your traffic growth over at least 30 minutes before driving more than 100,000 reads per second.

Provisioned Throughput Considerations

Factors that must be taken into consideration when determining the appropriate throughput requirements for an application are, item size, expected read and write rates, consistency, and secondary indexes, as discussed in the Throughput Model section of this whitepaper.

If an application performs more reads per second or writes per second than a table's provisioned throughput capacity allows, requests above the provisioned capacity will be throttled. For instance, if a table's write capacity is 1,000 units and an application can perform 1,500 writes per second for the maximum data item size, Amazon DynamoDB will allow only 1,000 writes per second to go through, and the extra requests will be throttled.

Tip: For applications where capacity requirement increases or decreases gradually and the traffic stays at the elevated or depressed level for at least several minutes, manage read and write throughput capacity automatically using auto scaling feature. With any changes in traffic pattern, DynamoDB will scale the provisioned capacity up or down within a specified range to match the desired capacity utilization you enter for a table or a global secondary index.

Read Performance Considerations

With the launch of Amazon DynamoDB Accelerator (DAX), you can now get microsecond access to data that lives in Amazon DynamoDB. DAX is an in-memory cache in front of DynamoDB and has the identical API as DynamoDB.

Because reads can be served from the DAX layer for queries with a cache hit, and the table will only serve the reads when there is a cache miss, the provisioned read capacity units can be lowered for cost savings.

Tip: Based on the size of your tables and data access pattern, consider provisioning a single DAX cluster for multiple smaller tables or multiple DAX clusters for a single bigger table or a hybrid caching strategy that will work best for your application.

Primary Key Design Considerations

Primary key design is critical to the performance of Amazon DynamoDB. When storing data, Amazon DynamoDB divides a table's items into multiple partitions, and distributes the data primarily based on the partition key element. The provisioned throughput associated with a table is also divided evenly among the partitions with no sharing of provisioned throughput across partitions.

Tip: To efficiently use the overall provisioned throughput, spread the workload across partition key values.

For example, if a table has a very small number of heavily accessed partition key elements, possibly even a single very heavily used partition key element, traffic can become concentrated on a single partition and create "hot spots" of read and write activity within a single item collection. In extreme cases, throttling can occur if a single partition exceeds its maximum capacity.

To better accommodate uneven access patterns, Amazon DynamoDB adaptive capacity enables your application to continue reading and writing to hot partitions without being throttled, provided that traffic does not exceed your table’s total provisioned capacity or the partition maximum capacity. Adaptive capacity works by automatically and instantly increasing throughput capacity for partitions that receive more traffic.

To get the most out of Amazon DynamoDB throughput, you can build tables where the partition key element has a large number of distinct values. Ensure that values are requested fairly uniformly and as randomly as possible. The same guidance applies to global secondary indexes. Choose partitions and sort keys that provide uniform workloads to achieve the overall provisioned throughput.

Local Secondary Index Considerations

When querying a local secondary index, the number of read capacity units consumed depends on how the data is accessed. For example, when you create a local secondary index and project non-key attributes into the index from the parent table, Amazon DynamoDB can retrieve these projected attributes efficiently.

In addition, when you query a local secondary index, the query can also retrieve attributes that are not projected into the index. Avoid these types of index queries that read attributes that are not projected into the local secondary index. Fetching attributes from the parent table that are not specified in the local secondary index causes additional latency in query responses and incurs a higher provisioned throughput cost.

Tip: Project frequently accessed non-key attributes into a local secondary index to avoid fetches and improve query performance.

Maintain multiple local secondary indexes in tables that are updated infrequently but are queried using many different criteria to improve query performance. This guidance does not apply to tables that experience heavy write activity.

If very high write activity to the table is expected, one option to consider is to minimize interference from reads by not reading from the table at all. Instead, create a global secondary index with a structure that is identical to that of the table, and then direct all queries to the index rather than to the table.

Global Secondary Index Considerations

If a query exceeds the provisioned read capacity of a global secondary index, that request will be throttled. Similarly, if a request performs heavy write activity on the table, but a global secondary index on that table has insufficient write capacity, then the write activity on the table will be throttled.

Tip: For a table write to succeed, the provisioned throughput settings for the table and global secondary indexes must have enough write capacity to accommodate the write; otherwise, the write will be throttled.

Global secondary indexes support eventually consistent reads, each of which consume one half of a read capacity unit. The number of read capacity units is the sum of all projected attribute sizes across all of the items returned in the index query results. With write activities, the total provisioned throughput cost for a write, consists of the sum of write capacity units consumed by writing to the table and those consumed by updating the global secondary indexes.

Apache HBase Performance Considerations

Apache HBase performance tuning spans hardware, network, Apache HBase configurations, Hadoop configurations, and the Java Virtual Machine Garbage Collection settings. It also includes applying best practices when using the client API. To optimize performance, it is worthwhile to monitor Apache HBase workloads with tools such as Ganglia to identify performance problems early and apply recommended best practices based on observed performance metrics.

Memory Considerations

Memory is the most restrictive element in Apache HBase. Performance-tuning techniques are focused on optimizing memory consumption.

From a schema design perspective, it is important to bear in mind that every cell stores its value as fully qualified with its full row key, column family, column name, and timestamp on disk. If row and column names are long, the cell value coordinates might become very large and take up more of the Apache HBase allotted memory. This can cause severe performance implications, especially if the dataset is large.

Tip: Keep the number of column families small to improve performance and reduce the costs associated with maintaining HFiles on disk.

Apache HBase Configurations

Apache HBase supports built-in mechanisms to handle region splits and compactions. Split/compaction storms can occur when multiple regions grow at roughly the same rate, and eventually split at about the same time. This can cause a large spike in disk I/O because of the compactions needed to rewrite the split regions.

Tip: Rather than relying on Apache HBase to automatically split and compact the growing regions, you can perform these tasks manually.

If you handle the splits and compactions manually, you can perform them in a time-controlled manner and stagger them across all regions to spread the I/O load as much as possible to avoid potential split/compaction storms. With the manual option, you can further alleviate any problematic split/compaction storms and gain additional performance.

Schema Design

A region can run hot when dealing with a write pattern that does not distribute the load across all servers evenly. This is a common scenario when dealing with streams processing events with time series data. The gradually increasing nature of time series data can cause all incoming data to be written to the same region.

This concentrated write activity on a single server can slow down the overall performance of the cluster. This is because inserting data is now bound to the performance of a single machine. This problem is easily overcome by employing key design strategies such as the following.

- Applying salting prefixes to keys; in other words, prepending a random number to a row.
- Randomizing the key with a hash function.
- Promoting another field to prefix the row key.

These techniques can achieve a more evenly distributed load across all servers.

Client API Considerations

There are a number of optimizations to take into consideration when reading or writing data from a client using the Apache HBase API. For example, when performing a large number of PUT operations, you can disable the auto-flush feature. Otherwise, the PUT operations will be sent one at a time to the region server.

Whenever you use a scan operation to process large numbers of rows, use filters to limit the scan scope. Using filters can potentially improve performance. This is because column over-selection can incur a nontrivial performance penalty, especially over large data sets.

Tip: As a recommended best practice, set the scanner-caching to a value greater than the default of 1, especially if Apache HBase serves as an input source for a MapReduce job.

Setting the scanner-caching value to 500, for example, will transfer 500 rows at a time to the client to be processed, but this might potentially cost more in memory consumption.

Compression Techniques

Data compression is an important consideration in Apache HBase production workloads. Apache HBase natively supports a number of compression algorithms that you can enable at the column family level.

Tip: Enabling compression yields better performance.

In general, compute resources for performing compression and decompression tasks are typically less than the overheard for reading more data from disk.

Apache HBase on Amazon EMR (HDFS Mode)

Apache HBase on Amazon EMR is optimized to run on AWS with minimal administration overhead. You still can access the underlying infrastructure and manually configure Apache HBase settings, if desired.

Cluster Considerations

You can resize an Amazon EMR cluster using core and task nodes. You can add more core nodes, if desired. Task nodes are useful for managing the Amazon EC2 instance capacity of a cluster. You can increase capacity to handle peak loads and decrease it later during demand lulls.

Tip: As a recommended best practice, in production workloads you can launch Apache HBase on one cluster and any analysis tools, such as Apache Hive, on a separate cluster to improve performance. Managing two separate clusters ensures that Apache HBase has ready access to the infrastructure resources it requires.

Amazon EMR provides a feature to backup Apache HBase data to Amazon S3. You can perform either manual or automated backups with options to perform full or incremental backups as needed.

Tip: As a best practice, every production cluster should always take advantage of the backup feature available on Amazon EMR.

Hadoop and Apache HBase Configurations

You can use a bootstrap action to install additional software or change Apache HBase or Apache Hadoop configuration settings on Amazon EMR. Bootstrap actions are scripts that are run on the cluster nodes when Amazon EMR launches the cluster. The scripts run before Hadoop starts and before the node begins processing data.

You can write custom bootstrap actions or use predefined bootstrap actions provided by Amazon EMR. For example, you can install Ganglia to monitor Apache HBase performance metrics using a predefined bootstrap action on Amazon EMR.

Apache HBase on Amazon EMR (Amazon S3 Storage Mode)

When you run Apache HBase on Amazon EMR with Amazon S3 storage mode enabled keep in recommended best practices discussed in this section.

Read Performance Considerations

With Amazon S3 storage mode enabled, Apache HBase region servers use `MemStore` to store data writes in-memory, and use write-ahead logs to store data writes in HDFS before the data is written to HBase `StoreFiles` in Amazon S3. Reading records directly from the StoreFile in Amazon S3 results in significantly higher latency and higher standard deviation than reading from HDFS.

Amazon S3 scales to support very high request rates. If your request rate grows steadily, Amazon S3 automatically partitions your buckets as needed to support higher request rates. However, the maximum request rates for Amazon S3 are lower than what can be achieved from the local cache. For more information about Amazon S3 performance, see Performance Optimization.

For read-heavy workloads caching data in-memory or on-disk caches in Amazon EC2 instance storage is recommended. Because Apache HBase region servers use `BlockCache` to store data reads in memory and `BucketCache` to store data reads on EC2 instance storage, you can choose an EC2 instance type with sufficient instance store.

In addition, you can add Amazon Elastic Block Store (Amazon EBS) storage to accommodate your required cache size. You can increase the `BucketCache` size on attached instance stores and EBS volumes using the `hbase.bucketcache.size` property.

Write Performance Considerations

As discussed in the preceding section, the frequency of `MemStore` flushes and the number of `StoreFiles` present during minor and major compactions can contribute significantly to an increase in region server response times, and consequently impact write performance. Consider increasing the size of the `MemStore` flush and `HRegion` block multiplier, which increases the elapsed time between major compactions for optimal write performance,

Apache HBase compactions and region servers perform optimally when fewer StoreFiles need to be compacted. You may get better performance using larger file block sizes (but less than 5 GB) to trigger Amazon S3 multipart upload functionality in EMRFS.

In summary, whether you are running a managed NoSQL database, such as Amazon DynamoDB or Apache HBase on Amazon EMR, or managing your Apache HBase cluster yourself on Amazon EC2 or on-premises, you should take performance optimizations into consideration if you want to maximize performance at reduced costs.

The key difference between a hosted NoSQL solution and managing it yourself is that a managed solution, such as Amazon DynamoDB or Apache HBase on Amazon EMR, lets you offload the bulk of the administration overhead so that you can focus on optimizing your application.

If you are a developer who is getting started with NoSQL, Amazon DynamoDB or the hosted Apache HBase on the Amazon EMR solution are suitable options, depending on your use case. For developers with in-depth Apache Hadoop/Apache HBase knowledge who need full control of their Apache HBase clusters, the self-managed Apache HBase deployment model offers the most flexibility from a cluster management standpoint.

Conclusion

Amazon DynamoDB lets you offload operating and scaling a highly available distributed database cluster, making it a suitable choice for today's real-time, web-based applications. As a managed service, Apache HBase on Amazon EMR is optimized to run on AWS with minimal administration overhead. For advanced users who want to retain full control of their Apache HBase clusters, the self-managed Apache HBase deployment model is a good fit.

Amazon DynamoDB and Apache HBase exhibit inherent characteristics that are critical for successfully processing massive amounts of data. With use cases ranging from batch-oriented processing to real-time data-serving, Amazon DynamoDB and Apache HBase are both optimized to handle large datasets. However, knowing your dataset and access patterns are key to choosing the right NoSQL database for your workload.

Contributors

Contributors to this document include:

- Wangechi Doble, Principal Solutions Architect, Amazon Web Services
- Ruchika Abbi, Solutions Architect, Amazon Web Services

Further Reading

For additional information, see:

- Amazon DynamoDB Developer Guide
- Amazon EC2 User Guide
- Amazon EMR Management Guide
- Amazon EMR Migration Guide
- Amazon S3 Developer Guide
- HBase: The Definitive Guide, by Lars George
- The Apache HBase™ Reference Guide
- Dynamo: Amazon's Highly Available Key-value Store

Document Revisions

Date	Description
January 2020	Amazon DynamoDB foundational features and transaction model updates
November 2018	Amazon DynamoDB, Apache HBase on EMR, and template updates
September 2014	First Publication