**CS595 –Research Paper**

* **Paper Topic:** Research Security for NoSQL Database and cloud NoSQL databases (Security between Cassandra, HBASE, DynamoDB and Cosmos DB)
* **Paper subject area:** Compare and evaluate security between Cassandra, HBASE, DynamoDB and Cosmos DB

**Abstract**

Data security is a shared responsibility between you, the customer, and your database provider. Depending on the database provider you choose, the amount of responsibility you carry can vary. If you choose an on-premises solution, you need to provide everything from end-point protection to physical security of your hardware - which is no easy task. If you choose a PaaS cloud database provider such as Azure Cosmos DB, your area of concern shrinks considerably.

NoSQL systems store and manage data in ways that allow for high operational speed and great flexibility on the part of the developers. It can be scaled horizontally across hundreds or thousands of servers. NoSQL encloses a wide variety of different database technologies that were developed in response to the demands presented in building modern applications.

Currently, NoSQL databases are in the evolutionary stage of their lifecycle and the attack vectors for NoSQL databases aren’t well mapped out. It is likely that new attack vectors may emerge in future that will target NoSQL data stores in new ways. NoSQL has not been designed with security as a priority, so developers or security teams need to add a security layer to their organization’s [NoSQL applications](https://www.computerweekly.com/feature/Big-data-applications-bring-new-database-choices-challenges).

***NoSQL database:***

Few of the application that is being widely used as NoSQL database are Cassandra and HBase.

**Cassandra**

It offers capabilities that NoSQL databases simply cannot match such as continuous availability, linear scale performance, operational simplicity and easy data distribution across multiple data centers and cloud availability zones.

Cassandra is a distributed storage system for managing very large amounts of structured data spread out across many commodity servers, while providing highly available service with no single point of failure. Cassandra aims to run on top of an infrastructure of hundreds of nodes. At this scale, small and large components fail continuously. The way Cassandra manages the persistent state in the face of these failures drives the reliability and scalability of the software systems relying on this service. While in many ways Cassandra resembles a database and shares many design and implementation strategies therewith, Cassandra does not support a full relational data model; instead, it provides clients with a simple data model that supports dynamic control over data layout and format. Cassandra system was designed to run on cheap commodity hardware and handle high write throughput while not sacrificing read efficiency.

Facebook runs the largest social networking platform that serves hundreds of millions of users at peak times using tens of thousands of servers located in many data centers around the world. There are strict operational requirements on Facebook's platform in terms of performance, reliability and efficiency, and to support continuous growth the platform needs to be highly scalable. Dealing with failures in an infrastructure comprised of thousands of components is our standard mode of operation; there are always a small but significant number of server and network components that are failing at any given time. As such, the software systems need to be constructed in a manner that treats failures as the norm rather than the exception. To meet the reliability and scalability needs described above Facebook has developed Cassandra.

Cassandra uses a synthesis of well-known techniques to achieve scalability and availability. Cassandra was designed to fulfill the storage needs of the Inbox Search problem. In- box Search is a feature that enables users to search through their Facebook Inbox. At Facebook this meant the system was required to handle a very high write throughput, billions of writes per day, and scale with the number of users.

Cassandra is shaped by two systems: Google’s BigTable and Amazon’s Dynamo. Both systems face the challenge of scaling, but they do it in different ways: BigTable uses the distributed file system Google already had, while Dynamo is based on a distributed hash table. Cassandra combines the data structure of BigTable, and the high availability of Dynamo.

The main features of Cassandra are summarized as follows:

* 1. **Symmetric:** Cassandra is meant to run on a cluster of nodes, although it can run on a single machine as well. All the nodes in the cluster are functionally identical (meaning that the software resides in each node is identical but not the data), so there is no coordinator node, no manager node etc. The symmetry feature ensures no single point of failure, linear scalability and easy administration.
  2. **Consistent Hashing (Distributed Hash Tables):** Consistent hashing is a scheme that provides hash table functionality in a way that the addition or removal of one slot does not significantly change the mapping of keys to slots. (In contrast, in most traditional hash tables, a change in the number of array slots causes nearly all keys to be remapped.) Consistent hashing is utilized to address three requirements. The first is to find a primary, or a secondary server for a resource given the resource key. The second requirement is to assign work to servers in accordance to their capacity, and finally, to add capacity to the system smoothly, without downtime.
  3. **Flexible Partitioning and High Availability:** Both placement of the data and placement of replica is highly flexible. By placement, we mean the canonical location of where data goes. Growing a Cassandra cluster is easy since Cassandra’s replica policy makes the data stored highly available.

Cassandra presents two standard strategies for data replica placement:

* in Rack Un- aware Strategy: replicas are always placed on the next (in increasing Token order) nodes along the ring.
* in Rack Aware Strategy: a replica is placed in the first node along the ring that belongs in another data center than the first; the remaining replicas, if any, are placed on the first nodes along the ring in the same rack as the first.
  1. **The Client interface:** Cassandra utilizes the Apache Thrift framework to provide a cross language RPC based client interface. Thrift is not intended to be used directly by developers and applications, since the RPC interface it provides is not easy to use. Instead, developers are encouraged to utilize higher-level clients based on the Thrift interface that simplify the syntax required to access the database, and also add enterprise level functionality. For example, Hector, a high- level client library written in Java, encapsulates the entire raw Thrift API, and adds (among other features) connection pooling, automatic fail over, JMX (Java Management Extensions) support, type-safety and load- balancing requests across cluster members.
  2. **Data Model**: The Cassandra data model consists of the following main concepts:
     + Keyspace: A keyspace is the first dimension of the Cassandra hash, and is the container for col- umn families. Keyspaces are of roughly the same granularity as a schema or database (i.e. a logical collection of tables) in the RDBMS world.
     + ColumnFamilies: A column family (CF) is a container for columns, analogous to the table in a relational system. A column family holds an ordered list of columns, that can be referenced by the column name.
     + Row: A row is a collection of columns or super- columns identified by a key. Each column family is stored in a separate file, and the file is sorted in row (i.e. key) major order. The row key determines which node the data is stored on. Related columns, that are accessed together, should be kept within the same column family.
     + Column: The column is the smallest increment of data. It’s a triplet that contains a name, a value and a timestamp. In the reminder of this paper timestamp will be elided for readability.
     + SuperColumns: Super Column is a column whose values are columns, that is, a (sorted) associative array of columns.

6) **Partitioning and Placement:** Partitioning is the policy of rows location, meaning which key resides on which node. The two major policies are random partitioning (RP) and order-preserving partitioning (OPP). OPP has one obvious advantage over RP: it provides the ability to perform range queries. However, using OPP may cause a load-balancing problem.

With both RP and OPP, by default Cassandra will tend to evenly distribute individual keys and their corresponding rows over the nodes in the cluster. The default algorithm is that every time one adds a new node, it will assign a range of keys to that node such that it takes responsibility for half the keys stored on the node that currently stores most of the keys.

7) **Consistency:** Cassandra is an eventually consistent data store, meaning that at any point in time each node might not be entirely up-to-date, but eventually all nodes are updated to the latest data values. The level of consistency can be chosen and indicates the number of replicas blocked to execute an action. The most usable level is quorum. Quorum level guarantees that half of the records are updated before the action returns. Another option is the Write all Read one model in which case every read will be consistent.

8) **Actions (**Write and Read): The client sends a write request to a single, random Cassandra node. This node acts as a proxy and writes the data to the cluster. The cluster of nodes is stored as a ring of nodes and writes are replicated to N nodes using a replication placement strategy. Each of those N nodes gets that write request and performs two actions for this message:

(1) Append the data change to the commit log.

(2) Update an in- memory Memtable structure with the change.

These two actions are the only actions that are performed synchronously. This is the reason why Cassandra is so fast for write actions: the slowest part is appending to a file. Unlike a database, Cassandra does not update data in-place on disk, nor update indices, so there are no intensive synchronous disk operations to block the write.

Reads are similar to writes in that the client makes a read request to a single random node in the Cassandra cluster (aka the Storage Proxy). The proxy determines the nodes in the ring (based on the replica placement strategy) that hold the copies of the data to be read and makes a read request to the appropriate nodes.

**Cassandra API:**

The Cassandra API consists of the following three simple methods.

* insert (table; key; rowMutation)
* get (table; key; columnName)
* delete (table; key; columnName)

columnName can refer to a specific column within a column family, a column family, a super column family, or a column within a super column.

**Cassandra Security Features**:

* Authentication based on internally controlled rolename/passwords: Cassandra authentication is roles-based and stored internally in Cassandra system tables. Administrators can create, alter, drop, or list roles using CQL commands, with an associated password. Roles can be created with superuser, non-superuser, and login privileges. The internal authentication is used to access Cassandra keyspaces and tables, and by cqlsh and DevCenter to authenticate connections to Cassandra clusters and sstableloader to load SSTables.
* Authorization based on object permission management: Authorization grants access privileges to Cassandra cluster operations based on role authentication. Authorization can grant permission to access the entire database or restrict a role to individual table access. Roles can grant authorization to authorize other roles. Roles can be granted to roles. CQL commands GRANT and REVOKE are used to manage authorization.
* Authentication and authorization based on JMX username/passwords: JMX (Java Management Extensions) technology provides a simple and standard way of managing and monitoring resources related to an instance of a Java Virtual Machine (JVM). This is achieved by instrumenting resources with Java objects known as Managed Beans (MBeans) that are registered with an MBean server. JMX authentication stores username and associated passwords in two files, one for passwords and one for access. JMX authentication is used by nodetool and external monitoring tools such as jconsole. In Cassandra 3.6 and later, JMX authentication and authorization can be accomplished using Cassandra's internal authentication and authorization capabilities.
* SSL encryption: Cassandra provides secure communication between a client and a database cluster, and between nodes in a cluster. Enabling SSL encryption ensures that data in flight is not compromised and is transferred securely. Client-to-node and node-to-node encryption are independently configured. Cassandra tools (cqlsh, nodetool, DevCenter) can be configured to use SSL encryption. The DataStax drivers can be configured to secure traffic between the driver and Cassandra.
* General security measures: Typically, production Cassandra clusters will have all non-essential firewall ports closed. Some ports must be open for nodes to communicate in the cluster. These ports are detailed.

Users also get to reap the benefits of several other features including:

* Complete service and ready to use: It gives you a complete product that is powered by Azure and can be automatically replicated in data centers worldwide.
* Multi-API: Because data is indexed automatically, users can access it using any API of their choice. They can see their data using SQL, Gremlin, JavaScript, Azure Table Storage, and MongoDB.
* A number of consistency levels: It uses five different consistency levels: bounded staleness, strong, session, eventual, and consistent-prefix.
* Latency: Very low latency is practically guaranteed at less than 10 milliseconds when reading data and less than 15 milliseconds when writing data.

**HBase**

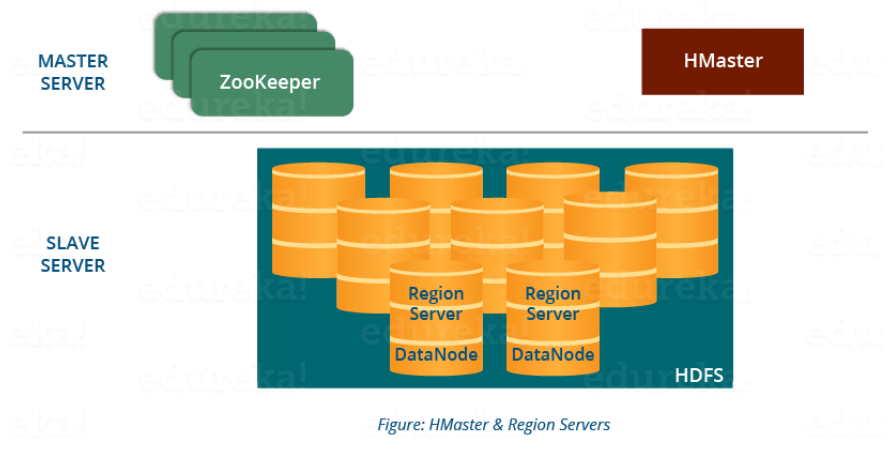
It is an open source NoSQL database that handle huge data sets with billions of rows and millions of columns, and it easily combines data sources that use a wide variety of different structures and schemas. HBase provides the much-needed means for secure communication with other technologies it relies upon.

**HBase Architecture:**

HBase is composed of three types of servers in a master slave type of architecture. Region servers serve data for reads and writes. When accessing data, clients communicate with HBase RegionServers directly. Region assignment, DDL (create, delete tables) operations are handled by the HBase Master process. Zookeeper, which is part of HDFS, maintains a live cluster state.

The Hadoop DataNode stores the data that the Region Server is managing. All HBase data is stored in HDFS files. Region Servers are collocated with the HDFS DataNodes, which enable data locality (putting the data close to where it is needed) for the data served by the RegionServers. HBase data is local when it is written, but when a region is moved, it is not local until compaction.

The NameNode maintains metadata information for all the physical data blocks that comprise the files.



* **HBase Regions:**

HBase Tables are divided horizontally by row key range into “Regions.” A region contains all rows in the table between the region’s start key and end key. Regions are assigned to the nodes in the cluster, called “Region Servers,” and these serve data for reads and writes. A region server can serve about 1,000 regions.

* **HBase HMaster:**

Region assignment, DDL (create, delete tables) operations are handled by the HBase Master.

A master is responsible for:

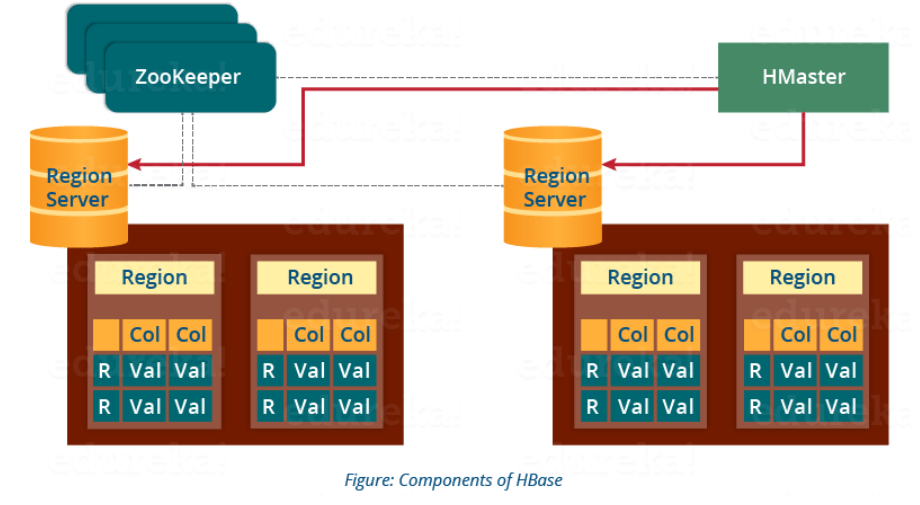
* Coordinating the region servers

- Assigning regions on startup, re-assigning regions for recovery or load balancing

- Monitoring all RegionServer instances in the cluster (listens for notifications from zookeeper)

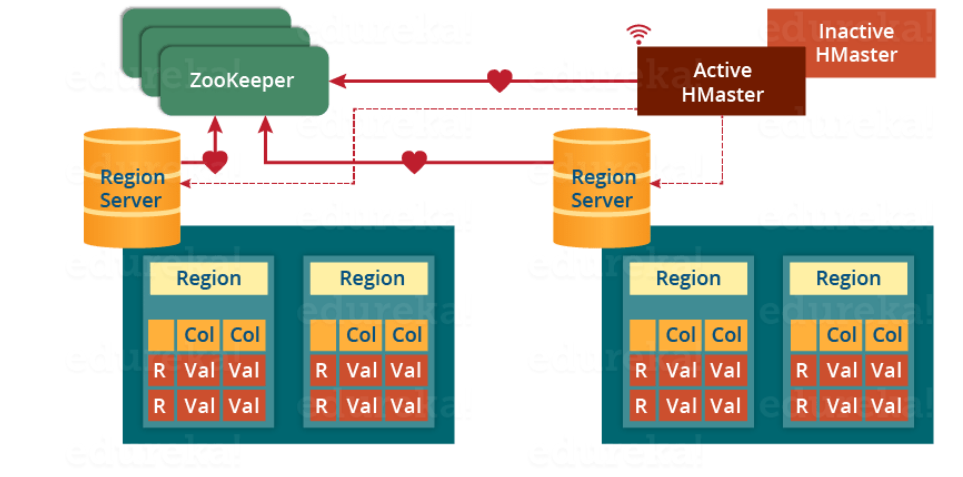
* Admin functions

- Interface for creating, deleting, updating tables



* **HBase ZooKeeper (Coordinator):**

HBase uses ZooKeeper as a distributed coordination service to maintain server state in the cluster. Zookeeper maintains which servers are alive and available and provides server failure notification. Zookeeper uses consensus to guarantee common shared state. Note that there should be three or five machines for consensus.



Zookeeper is used to coordinate shared state information for members of distributed systems. Region servers and the active HMaster connect with a session to ZooKeeper. The ZooKeeper maintains ephemeral nodes for active sessions via heartbeats.

Each Region Server creates an ephemeral node. The HMaster monitors these nodes to discover available region servers, and it also monitors these nodes for server failures. Zookeeper determines the first one and uses it to make sure that only one master is active. The active HMaster sends heartbeats to Zookeeper, and the inactive HMaster listens for notifications of the active HMaster failure.

If a region server or the active HMaster fails to send a heartbeat, the session is expired, and the corresponding ephemeral node is deleted. Listeners for updates will be notified of the deleted nodes. The active HMaster listens for region servers and will recover region servers on failure. The Inactive HMaster listens for active HMaster failure, and if an active HMaster fails, the inactive HMaster becomes active.

**HBase Security:**

For the most part, securing an HBase cluster is a one-way operation, and moving from a secure to an unsecure configuration should not be attempted without contacting Cloudera support for guidance.

To configure HBase security, complete the following tasks:

* **Configure HBase Authentication:** You must establish a mechanism for HBase servers and clients to securely identify themselves with HDFS, ZooKeeper, and each other (called authentication). This ensures that, for example, a host claiming to be an HBase RegionServer or a particular HBase client are in fact who they claim to be.
* **Configure HBase Authorization:** You must establish rules for the resources that clients are allowed to access (called authorization). After you have configured HBase authentication as described in the previous section, you must establish authorization rules for the resources that a client is allowed to access. HBase currently allows you to establish authorization rules at the table, column and cell-level. Cell-level authorization is fully supported since CDH 5.2.

Like all NoSQL databases, HBase and Cassandra have their security issues (the main one being that securing data spoils performance making the system heavy and inflexible). But it is safe to say that both databases have some features to ensure data security: authentication and authorization in both and inter-node + client-to-node encryption in Cassandra. HBase, in its turn, provides the much-needed means for secure communication with other technologies it relies upon.

Both Cassandra and HBase provide not just database-wide access control but allow a certain level of granularity. Cassandra enables row-level access and HBase goes as deep as cell-level. Cassandra defines user roles and sets conditions for these roles which later determine whether a user can see particular data or not. While HBase has an inverse ‘move.’ Its administrators assign a visibility label to data sets and then ‘tell’ users and user groups what labels they can see.

A Chart Comparing HBase and Cassandra:

|  |  |  |
| --- | --- | --- |
| **Capabilities** | **HBase** | **Cassandra** |
| **CAP Theorem** | **Consistency & Availability** | **Availability and Partition Tolerance** |
| **Coprocessor** | **Yes** | **No** |
| **Rebalancing** | **HBase provides Automatic rebalancing within a cluster.** | **Cassandra also provides rebalancing but not for overall cluster** |
| **Architecture Model** | **It is based on Master-Slave Architecture Model** | **Cassandra is based on Active-Active Node Modal** |
| **Base of Database** | **It is based on Google BigTable** | **Cassandra is based on Amazon DynamoDB** |
| **SPoF (Single Point of Failure)** | **If Master Node is not available entire cluster will not be accessible** | **All nodes having the same role within cluster so no SPoF** |
| **DR (Disaster Recovery)** | **DR is possible if Two Master Nodes are configured.** | **Yes, as all nodes having the same role** |
| **HDFS Compatibility** | **Yes, As HBase stores all meta-data in HDFS** | **No** |
| **Consistency** | **Strong** | **Not Strong as HBase** |

Facebook & another social networking side would prefer HBase (earlier both were using Cassandra, refer Facebook post) because of its availability other side banking domain sector looks for security for its every financial transaction so they would select Cassandra over HBase.

Cassandra Key characteristics involve High Availability, Minimal administration and No SPoF (Single Point of Failure) other side HBase is good for faster reading and writing the data with linear scalability.

Companies like Verizon, Bloomberg, Bank of America and much more are using HBase and Cassandra is being used by major social networking sites such as Twitter, Facebook etc.

We can’t conclude which one is best, HBase and Cassandra both are having their own advantage and disadvantages. Actual performance of both HBase and Cassandra Databases can be seen in the production environment.

***Cloud NoSQL database:***

Few of the application that is being widely used as cloud NoSQL database are Amazon DynamoDB and Azure Cosmos DB.

**Amazon DynamoDB**

It is a nonrelational database that delivers reliable performance at any scale. It's a fully managed, multi-region, multi-master database that provides consistent single-digit millisecond latency, and offers built-in security, backup and restore, and in-memory caching.

DynamoDB is a key-value store. It works really well if you are retrieving individual records based on key lookups. Complex queries or scans require careful indexing and are tricky or straight-up inadvisable to write — even if you don’t have a terribly large amount of data, and even if you have some familiarity with NoSQL design principles.

Amazon DynamoDB is designed for scale and performance. In most cases, the DynamoDB response times can be measured in single-digit milliseconds. However, there are certain use cases that require response times in microseconds. For these use cases, DynamoDB Accelerator (DAX) delivers fast response times for accessing eventually consistent data.

DAX is a DynamoDB-compatible caching service that enables you to benefit from fast in-memory performance for demanding applications. DAX addresses three core scenarios:

1. As an in-memory cache, DAX reduces the response times of eventually-consistent read workloads by an order of magnitude, from single-digit milliseconds to microseconds.
2. DAX reduces operational and application complexity by providing a managed service that is API-compatible with Amazon DynamoDB, and thus requires only minimal functional changes to use with an existing application.
3. For read-heavy or busty workloads, DAX provides increased throughput and potential operational cost savings by reducing the need to over-provision read capacity units. This is especially beneficial for applications that require repeated reads for individual keys.

DAX supports server-side encryption. With encryption at rest, the data persisted by DAX on disk will be encrypted. DAX writes data to disk as part of propagating changes from the primary node to read replicas.

**Amazon DynamoDB Global Tables:**

Amazon DynamoDB global tables provide a fully managed solution for deploying a multi-region, multi-master database, without having to build and maintain your own replication solution. When you create a global table, you specify the AWS regions where you want the table to be available. DynamoDB performs all of the necessary tasks to create identical tables in these regions and propagate ongoing data changes to all of them.

To illustrate one use case for a global table, suppose that you have a large customer base spread across three geographic areas—the US east coast, the US west coast, and western Europe. Customers would need to update their profile information while using your application. To address these requirements, you could create three identical DynamoDB tables named CustomerProfiles, in three different AWS regions. These three tables would be entirely separate from each other, and changes to the data in one table would not be reflected in the other tables. Without a managed replication solution, you could write code to replicate data changes among these tables; however, this would be a time-consuming and labor-intensive effort.

Instead of writing your own code, you could create a global table consisting of your three region-specific CustomerProfiles tables. DynamoDB would then automatically replicate data changes among those tables, so that changes to CustomerProfiles data in one region would be seamlessly propagated to the other regions. In addition, if one of the AWS regions were to become temporarily unavailable, your customers could still access the same CustomerProfiles data in the other regions.

DynamoDB global tables are ideal for massively scaled applications, with globally dispersed users. In such an environment, users expect very fast application performance. Global tables provide automatic multi-master replication to AWS regions world-wide, so you can deliver low-latency data access to your users no matter where they are located.

**Capturing Table Activity with DynamoDB Streams**

Many applications can benefit from the ability to capture changes to items stored in a DynamoDB table, at the point in time when such changes occur. Here are some example use cases:

* An application in one AWS region modifies the data in a DynamoDB table. A second application in another AWS region reads these data modifications and writes the data to another table, creating a replica that stays in sync with the original table.
* A popular mobile app modifies data in a DynamoDB table, at the rate of thousands of updates per second. Another application captures and stores data about these updates, providing near real time usage metrics for the mobile app.
* A global multi-player game has a multi-master topology, storing data in multiple AWS regions. Each master stays in sync by consuming and replaying the changes that occur in the remote regions.
* An application automatically sends notifications to the mobile devices of all friends in a group as soon as one friend uploads a new picture.
* A new customer adds data to a DynamoDB table. This event invokes another application that sends a welcome email to the new customer.

DynamoDB Streams enables solutions such as these, and many others. DynamoDB Streams captures a time-ordered sequence of item-level modifications in any DynamoDB table and stores this information in a log for up to 24 hours. Applications can access this log and view the data items as they appeared before and after they were modified, in near real time. Encryption at rest encrypts the data in DynamoDB streams.

A DynamoDB stream is an ordered flow of information about changes to items in an Amazon DynamoDB table. When you enable a stream on a table, DynamoDB captures information about every modification to data items in the table.

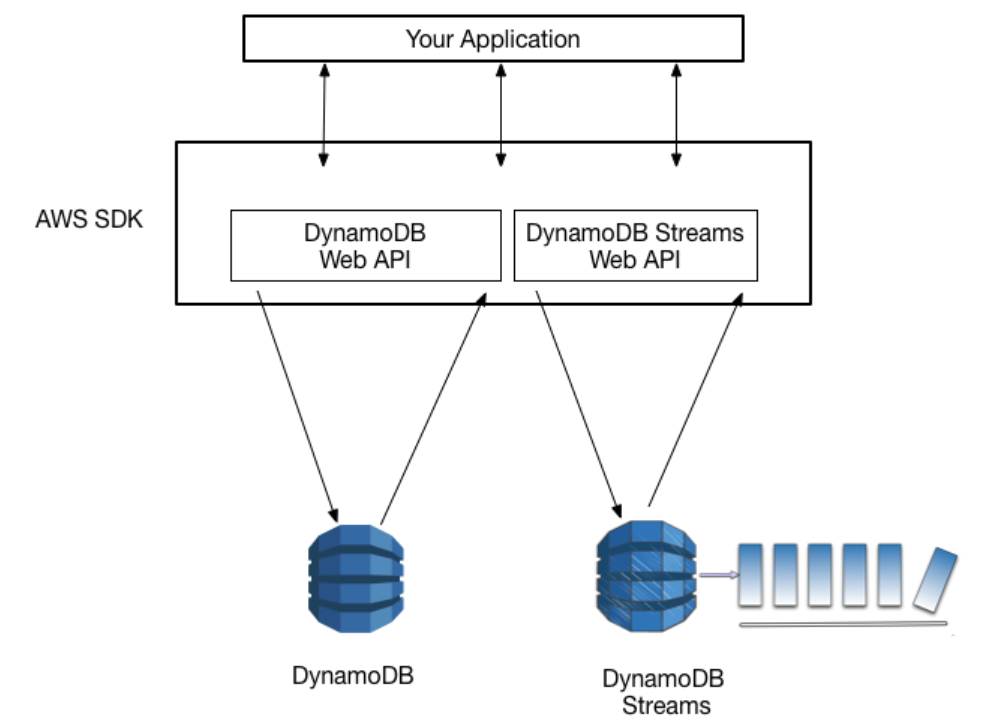
Whenever an application creates, updates, or deletes items in the table, DynamoDB Streams writes a stream record with the primary key attribute(s) of the items that were modified. A stream record contains information about a data modification to a single item in a DynamoDB table. You can configure the stream so that the stream records capture additional information, such as the "before" and "after" images of modified items.

DynamoDB Streams guarantees the following:

* Each stream record appears exactly once in the stream.
* For each item that is modified in a DynamoDB table, the stream records appear in the same sequence as the actual modifications to the item.

DynamoDB Streams writes stream records in near real time, so that you can build applications that consume these streams and take action based on the contents.

**Endpoints for DynamoDB Streams**



AWS maintains separate endpoints for DynamoDB and DynamoDB Streams. To work with database tables and indexes, your application will need to access a DynamoDB endpoint. To read and process DynamoDB Streams records, your application will need to access a DynamoDB Streams endpoint in the same region.

**Amazon DynamoDB Security**

Access to Amazon DynamoDB requires credentials. Those credentials must have permissions to access AWS resources, such as an Amazon DynamoDB table. For using AWS Identity and Access Management (IAM) and DynamoDB to help secure access to your resources, DynamoDB provides security in the form of:

1. Authentication, and

2. Access Control

1.Authentication: You can access AWS as any of the following types of identities:

AWS account root user – When you first create an AWS account, you begin with a single sign-in identity that has complete access to all AWS services and resources in the account. This identity is called the AWS account root user and is accessed by signing in with the email address and password that you used to create the account. It is recommended to not use the root user for your everyday tasks, even the administrative ones. Instead, adhere to the best practice of using the root user only to create your first IAM user. Then securely lock away the root user credentials and use them to perform only a few account and service management tasks.

IAM user – An IAM user is an identity within your AWS account that has specific custom permissions (for example, permissions to create a table in DynamoDB). You can use an IAM user name and password to sign in to secure AWS webpages like the AWS Management Console, AWS Discussion Forums, or the AWS Support Center.

In addition to a user name and password, you can also generate access keys for each user. You can use these keys when you access AWS services programmatically, either through one of the several SDKs or by using the AWS Command Line Interface (CLI). The SDK and CLI tools use the access keys to cryptographically sign your request. If you don’t use AWS tools, you must sign the request yourself. DynamoDB supports Signature Version 4, a protocol for authenticating inbound API requests.

IAM role – An IAM role is an IAM identity that you can create in your account that has specific permissions. It is similar to an IAM user, but it is not associated with a specific person. An IAM role enables you to obtain temporary access keys that can be used to access AWS services and resources. IAM roles with temporary credentials are useful in the following situations:

Federated user access – Instead of creating an IAM user, you can use existing user identities from AWS Directory Service, your enterprise user directory, or a web identity provider. These are known as federated users. AWS assigns a role to a federated user when access is requested through an identity provider.

AWS service access – You can use an IAM role in your account to grant an AWS service permission to access your account’s resources. For example, you can create a role that allows Amazon Redshift to access an Amazon S3 bucket on your behalf and then load data from that bucket into an Amazon Redshift cluster.

Applications running on Amazon EC2 – You can use an IAM role to manage temporary credentials for applications that are running on an EC2 instance and making AWS API requests. This is preferable to storing access keys within the EC2 instance. To assign an AWS role to an EC2 instance and make it available to all of its applications, you create an instance profile that is attached to the instance. An instance profile contains the role and enables programs that are running on the EC2 instance to get temporary credentials.

2.Access Control:

You can have valid credentials to authenticate your requests, but unless you have permissions you cannot create or access Amazon DynamoDB resources. For example, you must have permissions to create an Amazon DynamoDB table.

**Azure Cosmos DB**

It offers multiple NoSQL choices including key-value, graph, column-family, and document data in one service. Azure Cosmos DB is Microsoft's globally distributed, multi-model database. With the click of a button, Azure Cosmos DB enables you to elastically and independently scale throughput and storage across any number of Azure's geographic regions. It offers throughput, latency, availability, and consistency guarantees with comprehensive service level agreements (SLAs), something no other database service can offer.

Azure Cosmos DB provides security in the form of:

* Network security: Using an IP firewall is the first layer of protection to secure your database. Azure Cosmos DB supports policy driven IP-based access controls for inbound firewall support. The IP-based access controls are similar to the firewall rules used by traditional database systems, but they are expanded so that an Azure Cosmos DB database account is only accessible from an approved set of machines or cloud services. All requests originating from machines outside this allowed list are blocked by Azure Cosmos DB. Requests from approved machines and cloud services then must complete the authentication process to be given access control to the resources.
* Authorization: Azure Cosmos DB uses hash-based message authentication code (HMAC) for authorization. Each request is hashed using the secret account key, and the subsequent base-64 encoded hash is sent with each call to Azure Cosmos DB. To validate the request, the Azure Cosmos DB service uses the correct secret key and properties to generate a hash, then it compares the value with the one in the request. If the two values matches, the operation is authorized successfully and the request is processed, otherwise there is an authorization failure and the request is rejected.
* Users and permissions: Using the master key for the account, you can create user resources and permission resources per database. A resource token is associated with a permission in a database and determines whether the user has access (read-write, read-only, or no access) to an application resource in the database. Application resources include container, documents, attachments, stored procedures, triggers, and UDFs. The resource token is then used during authentication to provide or deny access to the resource.
* Active directory integration (RBAC): You can also provide access to the database account using Access control (IAM) in the Azure portal, as shown in the screenshot that follows this table. IAM provides role-based access control and integrates with Active Directory.
* Global replication: Azure Cosmos DB offers turnkey global distribution, which enables you to replicate your data to any one of Azure's world-wide datacenters with the click of a button. Global replication lets you scale globally and provide low-latency access to your data around the world.  
  In the context of security, global replication ensures data protection against regional failures.
* Regional failovers: If you have replicated your data in more than one data center, Azure Cosmos DB automatically rolls over your operations should a regional data center go offline. You can create a prioritized list of failover regions using the regions in which your data is replicated.
* Local replication: Even within a single data center, Azure Cosmos DB automatically replicates data for high availability giving you the choice of consistency levels. This guarantees a 99.99% availability SLA for all single region accounts and all multi-region accounts with relaxed consistency, and 99.999% read availability on all multi-region database accounts.
* Automated online backups: Azure Cosmos DB databases are backed up regularly and stored in a georedundant store.
* Restore deleted data: The automated online backups can be used to recover data you may have accidentally deleted up to ~30 days after the event.
* Protect and isolate sensitive data: Personal data and other confidential data can be isolated to specific container and read-write, or read-only access can be limited to specific users.
* Monitor for attacks: By using audit logging and activity logs, you can monitor your account for normal and abnormal activity. You can view what operations were performed on your resources, who initiated the operation, when the operation occurred, the status of the operation.
* Respond to attacks: Once you have contacted Azure support to report a potential attack, a 5-step incident response process is kicked off. The goal of the 5-step process is to restore normal service security and operations as quickly as possible after an issue is detected, and an investigation is started.
* Protected facilities: Data in Azure Cosmos DB is stored on SSDs in Azure's protected data centers.
* HTTPS/SSL/TLS encryption: All client-to-service Azure Cosmos DB interactions are SSL/TLS 1.2 capable. Also, all intra datacenter and cross datacenter replication is SSL/TLS 1.2 enforced.
* Administrative accounts with strong passwords: It's hard to believe we even need to mention this requirement, but unlike some of our competitors, it's impossible to have an administrative account with no password in Azure Cosmos DB.  
  Security via SSL and HMAC secret based authentication is baked in by default.
* Encryption at rest: All data stored into Azure Cosmos DB is encrypted at rest.

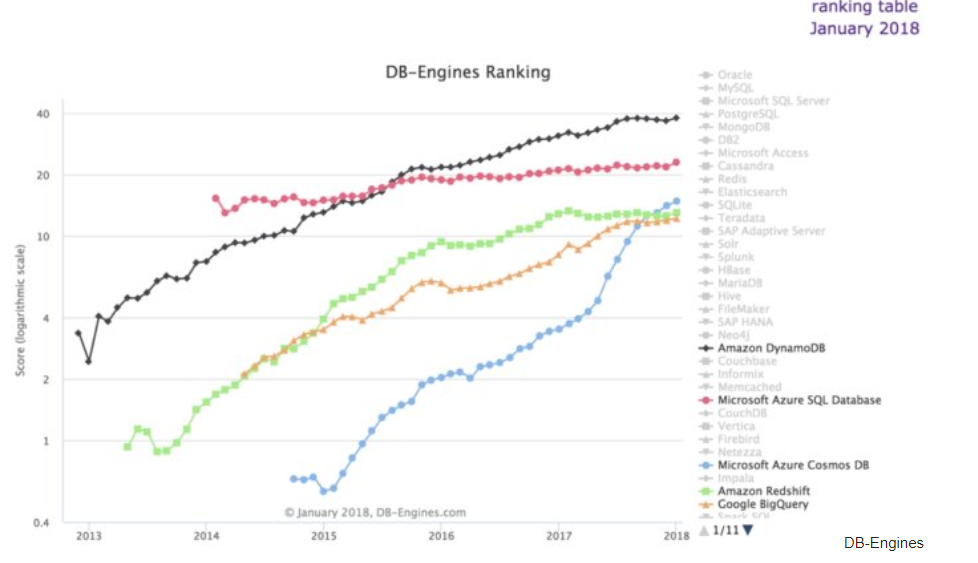
Microsoft’s genius with Cosmos DB is that developers may want to have their polyglot persistence cake and eat it too—all in one place. With Cosmos DB, Microsoft offers multiple consistency models in the same database, so the choice of model can be a function of the workload rather than the product.

Equally huge is the multimodel approach that makes Azure Cosmos DB a bit of a jack-of-all-trades. It's a universal back end for different kinds of databases—likely including future styles of database that haven't been invented yet.”

This lets a developer dig deep into Cosmos DB for a wide array of application requirements, rather than having to learn an equally wide array of point solutions for different application needs. There’s a risk that Cosmos DB’s general-purpose approach could make it a watered-down option for all applications, rather than the best tool for a particular job. Judging from its popularity, however, developers don’t see it this way.

All of this sets us up in 2018 to watch a classic battle between two companies at the top of their cloud games: Amazon and Microsoft. At the AWS conference in December, Amazon announced more new database technologies and feature upgrades than most companies could hope to create in a lifetime, each living in a separate product. Microsoft, by contrast, seems to be doubling down on Cosmos DB, investing it with diverse powers that make it a great default database for any application a developer can think up.

Azure Cosmos DB and Amazon DynamoDB ranking table:



A Chart Comparing Azure Cosmos DB and Amazon DynamoDB:

|  |  |  |
| --- | --- | --- |
| Capabilities | Azure Cosmos DB | Amazon DynamoDB |
| Global Distribution | A single entity can span any number of Azure regions (30+) | No |
| Horizontal scalability of both storage and throughput worldwide | Yes, across any number of Azure regions | No |
| Multi-homing APIs | Yes | No |
| Guaranteed low read/write latency worldwide | Yes, at 99th percentile worldwide | No |
| Consistency Model | 5(Strong, Eventual, Bounded staleness, Session, Consistent-Prefix) | 2(Strong, Eventual) |
| Multi-model + multi-API | Yes | Yes |
| Schema management | No management required | Index management required |
| Automatic Indexing | Yes | No |
| Comprehensive SLA’s | Yes (latency, consistency, throughput, high availability) | No |

Properties comparison between Amazon Dynamo DB, Cassandra, HBase and Microsoft Azure Cosmos DB:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Name** | **Amazon DynamoDB** | **Cassandra** | **HBase** | **Microsoft Azure Cosmos DB** |
| **Description** | Hosted, scalable database service by Amazon with the data stored in Amazons cloud | Wide-column store based on ideas of BigTable and DynamoDB | Wide-column store based on Apache Hadoop and on concepts of BigTable | Globally distributed, horizontally scalable, multi-model database service |
| **Primary database model** | [Document store](https://db-engines.com/en/article/Document+Stores) [Key-value store](https://db-engines.com/en/article/Key-value+Stores) | [Wide column store](https://db-engines.com/en/article/Wide+Column+Stores) | [Wide column store](https://db-engines.com/en/article/Wide+Column+Stores) | [Document store](https://db-engines.com/en/article/Document+Stores) [Graph DBMS](https://db-engines.com/en/article/Graph+DBMS) [Key-value store](https://db-engines.com/en/article/Key-value+Stores) [Wide column store](https://db-engines.com/en/article/Wide+Column+Stores) |
| **Website** | [aws.amazon.com/­dynamodb](https://aws.amazon.com/dynamodb/) | [cassandra.apache.org](http://cassandra.apache.org/) | [hbase.apache.org](http://hbase.apache.org/) | [azure.microsoft.com/­services/­cosmos-db](https://azure.microsoft.com/services/cosmos-db) |
| **Technical documentation** | [docs.aws.amazon.com/­dynamodb](https://docs.aws.amazon.com/dynamodb/) | [cassandra.apache.org/­doc/­latest](http://cassandra.apache.org/doc/latest/) | [hbase.apache.org](http://hbase.apache.org/) | [docs.microsoft.com/­en-us/­azure/­cosmos-db](https://docs.microsoft.com/en-us/azure/cosmos-db/) |
| **Developer** | Amazon | Apache Software Foundation | Apache Software Foundation | Microsoft |
| **Initial release** | 2012 | 2008 | 2008 | 2014 |
| **License** | commercial | Open Source | Open Source | commercial |
| **Cloud-based only** | yes | no | No | Yes |
| **Server operating systems** | hosted | BSD Linux OS X Windows | Linux Unix Windows | Hosted |
| **Data scheme** | schema-free | schema-free | schema-free | schema-free |
| **Typing** | yes | yes | No | yes |
| **Secondary indexes** | yes | restricted | No | yes |
| **SQL** | no | SQL-like SELECT, DML and DDL statements (CQL) | No | SQL-like query language |
| **APIs and other access methods** | RESTful HTTP API | Proprietary protocol  Thrift | Java API RESTful HTTP API Thrift | DocumentDB API Graph API MongoDB API RESTful HTTP API Table API |
| **Supported programming languages** | .Net ColdFusion Erlang Groovy Java JavaScript Perl PHP Python Ruby | C# C++ Clojure Erlang Go Haskell Java JavaScript  Perl PHP Python Ruby Scala | C C# C++ Groovy Java PHP Python Scala | .Net C# Java JavaScript JavaScript (Node.js) MongoDB client drivers written for various programming languages Python |
| **Server-side scripts** | no | no | yes | JavaScript |
| **Triggers** | yes | yes | Yes | JavaScript |
| **Replication methods** | yes | selectable replication factor | selectable replication factor | yes |
| **MapReduce** | no | yes | Yes | with Hadoop integration |
| **Consistency concepts** | Eventual Consistency Immediate Consistency | Eventual Consistency Immediate Consistency | Immediate Consistency | Bounded Staleness Consistent Prefix Session Consistency Eventual Consistency Immediate Consistency |
| **Foreign keys** | no | no | No | No |
| **Transaction concepts** | no | no | No | Multi-item ACID transactions with snapshot isolation within a partition |
| **Concurrency** | yes | yes | Yes | Yes |
| **Durability** | yes | yes | Yes | Yes |
| **User concepts** | Access rights for users and roles can be defined via the AWS Identity and Access Management (IAM) | Access rights for users can be defined per object | Access Control Lists (ACL) | Access rights can be defined down to the |

**Conclusion:**

In this research paper, I have reviewed the main functionality and security features of Cassandra, HBASE, DynamoDB and Cosmos DB. The main problems common to all systems include lack of encryption support for the data files, weak authentication both between the client and the servers and between server members, very simple authorization without support for fine-grained authorization, and vulnerability to SQL injection and Denial of Service attacks.

* **Reference resources:**
  + <https://data-flair.training/blogs/hbase-vs-cassandra/>
  + <https://db-engines.com/en/system/Cassandra%3BHBase>slee
  + <https://www.ibm.com/analytics/hadoop/hbase>
  + <https://docs.microsoft.com/en-us/azure/cosmos-db/introduction>
  + <https://mapr.com/blog/in-depth-look-hbase-architecture/>
  + <https://aws.amazon.com/dynamodb/>
  + <https://docs.microsoft.com/en-us/azure/cosmos-db/database-security>
  + https://docs.microsoft.com/en-us/azure/cosmos-db/databases-containers-items