SQL to NoSQL and Beyond: Architecture Differences and the Benefits of BuildDB™

CONTENTS

SQL and NoSQL: Understanding the Tradeoffs between Flexibility, Scale and Cost

When and how to migrate data from SQL to NoSQL are matters of much debate. It can certainly be a daunting task, but when your SQL systems hit architectural limits or your cloud provider expenses skyrocket, it's probably time to consider a move.

SQL VERSUS NOSQL BASICS

Since their invention in 1970 by Edgar Codd, relational databases have served as the default data store for almost every IT organization, large or small. Today, the most iconic and familiar relational databases include IBM DB2, Oracle Database, Microsoft SQL Server, PostgreSQL, and MySQL.

Structured Query Language, or SQL, was invented at IBM soon after the introduction of the relational database. Since its introduction, SQL has become the most widely used database language, used for querying data, data manipulation (insert, update and delete), data definition (schema creation and modification), and data access control. Though the terms refer to different technologies, 'SQL' and 'RDBMS' have become virtually interchangeable. Though some non-relational databases support SQL, the term "SQL database" generally means a relational database.

During the decades in which relational databases proliferated, data entry was largely a manual process. Times have changed. The advent of smartphones, the 'app economy,' and cloud computing in the late 2000s caused a change in the workloads, query types, and traffic patterns needed to support a global user base.

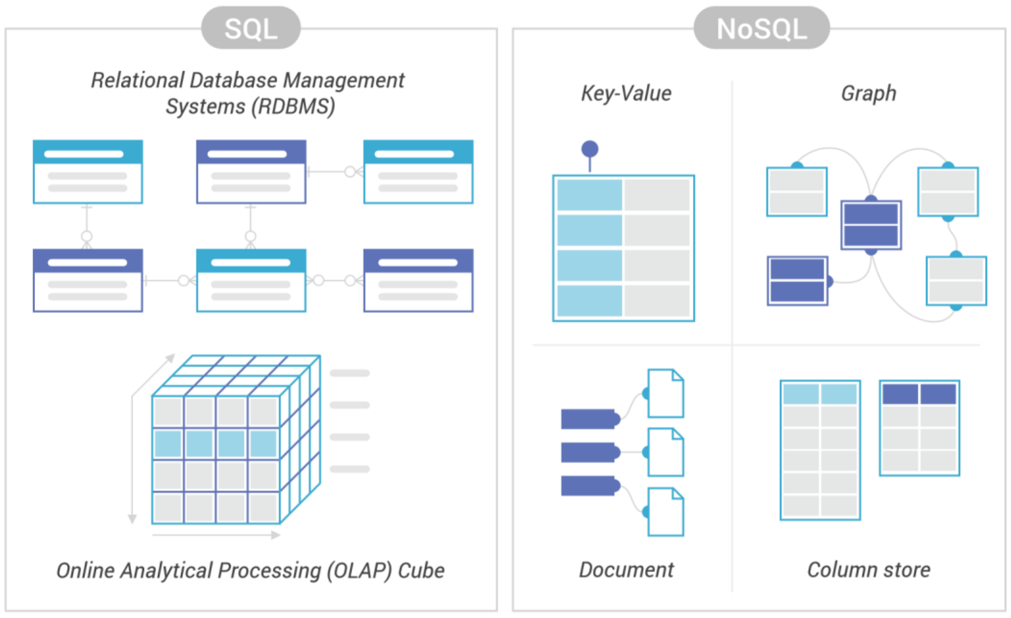
Fast forward to 2020, when people, smart devices, sensors, and machines emit continuous streams of data, such as user activity, IoT and machine-generated data, and metadata that encompasses geolocation and telemetry. As early as 2013, one researcher noted that 90% of the world's data had been generated over the previous two years. This trend has only accelerated. In response to this torrent of data, many organizations have been reevaluating their use of traditional relational databases.

Cloud computing exposed many limitations of relational databases. RDBMSs proliferated in an age when databases were isolated islands with relatively stable user bases, running in a traditional client-server configuration. RDBMSs support arbitrary reshaping and joining of data, but performance can be variable and unpredictable. In restricted environments, such variable performance characteristics could be managed. The shift to a mobile, globally dispersed user base caught many organizations off guard.

During the same time period, consumer demands shifted radically. Today, users expect low-latency applications that deliver an extremely responsive experience, regardless of the user's location. Apps that are slow and unresponsive contribute significantly to customer churn. Predictable performance became more important than the semantic flexibility afforded by RDBMSs.

Latency issues can be addressed by shifting data closer to the customer. To meet this need, data must be replicated across different geographic locations. Such geographical replication turned out to be a struggle for RDBMSs. While RDBMSs are not fit for distributed deployments, non-relational databases are designed specifically to support such topologies.

A number of alternative non-relational database systems have been proposed, including Google's Bigtable (2006) and Amazon's Dynamo (2007). The papers for these projects paved the way for Cassandra (2008) and MongoDB (2009).


SQL
Relational Database Management Systems (RDBMS)
Online Analytical Processing (OLAP) Cube
NoSQL
Key-Value
Graph
Document
Column store

SQL vs NoSQL

Today, a range of mature NoSQL databases are available to help organizations scale big data applications.

Yet, despite their origins in a long-forgotten technology cycle, relational SQL databases are by no means ‘legacy’ technology. Some SQL databases, notably PostgreSQL and MySQL, have experienced a recent resurgence in popularity. A new generation of NewSQL databases, notably Google Spanner and CockroachDB, leverage SQL as a query language and offer a distributed architecture similar to that of NoSQL databases yet provide full transactional support.

ARCHITECTURAL DIFFERENCES BETWEEN SQL AND NOSQL

ECONOMIES OF SCALE

Database administrators add capacity to RDBMS and NoSQL databases in very different ways. Typically, the only way to add capacity in a relational system is to add expensive hardware, faster CPUs, more RAM, and more advanced networking components. This is often referred to as ‘vertical’ scale, or ‘scaling up.’

In contrast, NoSQL databases are designed for low latency and high resilience, being built from the ground up to run across clusters of distributed nodes. This architecture is often referred to as ‘horizontal scale,’ or ‘scaling out.’ To add capacity to a NoSQL database, administrators simply add more nodes, a very simple process in modern cloud environments.

In a NoSQL cluster, nodes are easy to add and remove according to demand, providing ‘elastic’ capacity. This feature enables organizations to align their data footprint with the needs of the business while maintaining availability even in the face of seasonal demand spikes, node failures, and network outages.

The horizontal scale of NoSQL brings tradeoffs of its own. Adding commodity hardware to a cluster can be cheap in terms of software licenses and subscriptions. However, as more and more nodes are added in the pursuit of higher throughput and lower latency, operational overhead and administrative costs spike. Big clusters of small instances demand more attention and generate more alerts than small clusters of large instances.

(Notably, some next-generation NoSQL databases like BuildDB™ are able to overcome this tradeoff, scaling out in a way that can take advantage of the powerful hardware in modern servers and ultimately running in smaller, though still distributed, clusters of fewer nodes.)

REPLICAS OF DATA

Replicating data across multiple nodes allows databases to achieve higher levels of resilience. In the RDBMS world, it’s not trivial to replicate data across multiple instances. Relational databases do not support replication. Instead, they rely on external tools to extract and update copies of datasets. These tools run batch processes that often take hours to complete. As a result, there is no way to ensure real-time synchronization of data among the copies of data.

While non-relational databases provide native support for data replication, they follow three basic models: multi-master databases, such as DynamoDB, master-slave architectures, such as MongoDB, and masterless, such as BuildDB™. Given their reliance on master nodes, both multi-master and master-slave architecture introduce a point of failure. When a master goes down, the process of electing a new master introduces a brief downtime. Even though the delay may be minimal, measured in milliseconds, that delay can still cause SLA violations.

A masterless architecture addresses this limitation. In these databases, data is replicated across multiple nodes, all of which are equal.

Blockchain Nodes

In a masterless architecture, no single node can bring down an entire cluster. A typical masterless topology involves three or more replicas for each dataset. Adopting a NoSQL database that implements a masterless architecture provides yet another layer of resilience for high-volume, low-latency applications.

The core premise behind blockchain is that it is decentralized. Being decentralized means that no single node can corrupt the data in the ledgers of the blockchain cluster.   BuildDB™ leverages the decentralized nature of blockchain to create the masterless NoSQL architecture that is highly available and scalable without the limitations of a traditional relational database.

While traditional blockchain has been associated with performance and latency limitations, BuildDB™ has a patent pending form of consensus that allows storage and retrieval of data in single digit milliseconds while remaining secure and immutable.

APPLICATION-DRIVEN USE CASES

The rise in popularity of NoSQL databases paralleled the adoption of agile development and DevOps practices. Unlike RDBMSs, NoSQL databases encourage 'application-first' or API-first development patterns. Following these models, developers first consider queries that support the functionality specific to an application, rather than considering the data models and entities. This developer-friendly architecture paved the path to the success of the first generation of NoSQL databases.

In contrast, relational databases impose fairly rigid, schema-based structures to data models; tables consisting of columns and rows, which can be joined to enable 'relations' among entities. Each table typically defines an entity. Each row in a table holds one entry, and each column contains a specific piece of information for that record. The relationships among tables are clearly defined and usually enforced by schemas and database rules.

Relational data models enforce uniformity, whereas non-relational models do not. NoSQL databases permit multiple 'shapes' of data objects to coexist, which is more flexible but can also be more error prone. In the world of relational databases, the schemas that support uniformity are usually managed by database administrators. This can sometimes introduce friction between administrators and development teams, resulting in long, non-agile application development lifecycles. Such highly structured data requires normalization to reduce redundancy. Since the data model is based on the entity being represented; query patterns are a secondary consideration.

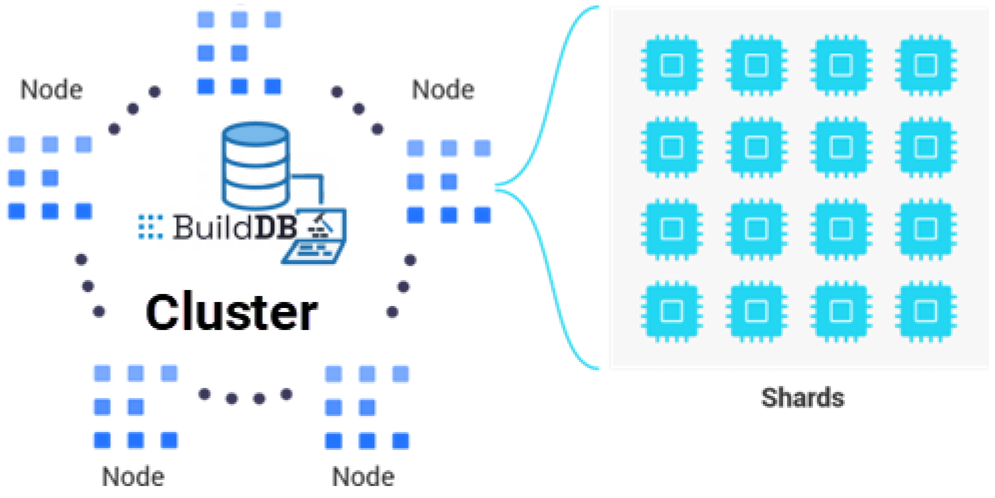
NoSQL inverts this approach, placing more power in the hands of the developer and often decentralizing control over data structures. Nonrelational data models are

flexible, and schema management is often delegated to application developers, who are relatively free to adapt data models independently. Such a decentralized approach can accelerate development cycles and provide a more agile approach to addressing user requirements.

CONSISTENCY VERSUS AVAILABILITY

A consideration of the architectural differences between relational and non-relational databases would not be complete without the CAP theorem. The CAP theorem was formulated by Eric Brewer in 2000, as a way of expressing the key tradeoffs in distributed systems. The CAP theorem states that it is impossible for a distributed data store to provide more than two of the following three guarantees:

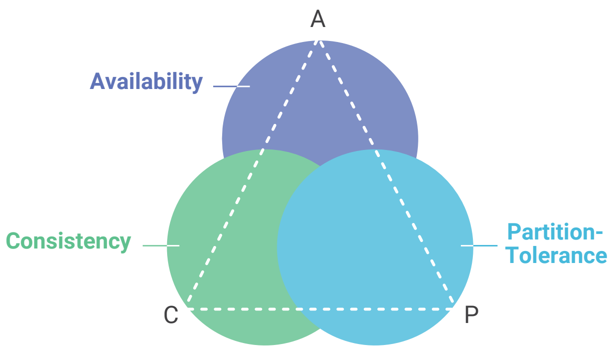
- **Consistency:** Every read receives either the most recent write or an error.
- **Availability:** Every request receives a response that is not an error, but with no guarantee that it contains the most recent



Masterless Architecture in BuildDB™

write.

- **Partition Tolerance:** The system continues to operate even when an arbitrary number of messages are delayed, dropped or reordered among nodes.



CAP Theorem

Another way of putting this is that the CAP theorem dictates that any data store brings with it a fundamental trade-off. As such, many databases are referred to as CP (consistent and partition tolerance, but not available) or AP (available and partition-tolerant, but not consistent). In CAP terms, the critical trade-off that distinguishes relational and nonrelational data stores is between availability and consistency. SQL data stores sacrifice availability in favor of data consistency. NoSQL data stores sacrifice consistency in favor of availability.

It is important to note that the CAP theorem has come under significant criticism. Martin Kleppmann, in particular, has written a comprehensive Critique of the CAP Theorem. So, it is important to keep in mind that the theorem is merely a simplified model for understanding a very complex topic.

ACID VERSUS BASE CONSISTENCY

One of the defining tradeoffs between relational and non-relational datastores is in the type of consistency that they provide. In simple terms, RDBMS provides strong consistency, while NoSQL databases provide a weaker form. Consistency in general refers to a database's ability to process concurrent transactions while preserving the integrity of the data. Somewhat confusingly, 'consistency' as defined in the CAP theorem has a different, though related, meaning than the consistency discussed in this section. The definition used by Brewer in the CAP theorem derives from distributed systems theory, while the definition used in this section derives from database theory.

In simple terms, consistency is a guarantee that a read should return the result of the latest successful write. This seems simple, but such a guarantee is incredibly difficult to deliver without impacting the performance of the system as a whole. In a relational database, a single data item is actually split across independent registers that must agree with one another. Thus, a single database write is actually decomposed into several small writes to these registers, which must be completed and visible when the read is executed. With concurrent operations running against the database, the semblance of order between the group of sub-operations needs to be maintained; the concurrent operations must be atomic. ACID consistency means the rules of relations must be satisfied. In a globally distributed database topology, which involves multiple clusters each containing many nodes the problem of consistency becomes exponentially more complex.

In general, relational databases that support
'strong consistency' provide 'ACID guarantees.' ACID is an acronym designed to capture the essential elements of a strongly consistent database. The components of the ACID are as follows:

- **Atomicity:** Guarantees that each transaction is treated as a single "unit", which either succeeds completely or fails completely.
- **Consistency:** Guarantees that each transaction only changes affected data in permitted ways.
- **Isolation:** Guarantees that the concurrent execution of transactions leaves the database in the same state that would have been obtained if the transactions were executed sequentially.
- **Durability:** The transactions results are permanent, even in the event of system failure.

ACID compliance is a complex and often contested topic. In fact, one popular system of analysis, the Jepsen test, is dedicated to verifying vendor consistency claims.

By their nature, ACID-compliant databases are generally slow, difficult to scale, and expensive to run and maintain. It should be noted some RDBMS systems enable performance to be improved by relaxing ACID guarantees. Still, all SQL databases are ACID compliant to varying degrees, and as such, they all share this downside. The practical effect of ACID compliance is to make it extraordinarily difficult and expensive to achieve resilient, distributed SQL database deployments.

In contrast to RDBMS' ACID guarantees, NoSQL databases provide so-called 'BASE guarantees.' BASE enables availability and relaxes the stringent consistency. The acronym BASE designates:

- **Basic Availability:** Data is available most of the time, even during a partial system failure.
- **Soft state:** Individual data items are independent and do not have to be consistent with each other.
- **Eventual consistency:** Data will become consistent at some unspecified point in the future.

As such, NoSQL databases sacrifice a degree of consistency in order to increase availability. Rather than providing strong consistency, NoSQL databases generally provide eventual consistency. A data store that provides BASE guarantees can occasionally fail to return the result of the latest write, providing different answers to applications making requests. Developers building applications against eventually consistent data stores often implement consistency checks in their application code.

Lightweight transactions

In a traditional SQL RDBMS, a "transaction" is a logical unit of work — a group of tasks that provides the ACID guarantees discussed above. To compensate for relaxed consistency, some NoSQL databases offer 'lightweight transactions' (LWTs).

Lightweight transactions are limited to a single conditional statement, which enables an atomic "compare and set" operation. Such an operation checks whether a condition is true before it conducts the transaction. If the condition is not met, the transaction is not executed. (For this reason, LWTs are sometimes called 'conditional statements'). LWTs do not truly lock the database for the duration of the transaction; they only 'lock' a single cell or row. LWTs leverage a consensus protocol such as Paxos to ensure that all nodes in the cluster agree the change is committed. In this way, LWTs can provide sufficient consistency for applications that require the availability and resilience of a distributed database.

QUERY LANGUAGES: SQL and CQL VERSUS QUERYCHAIN™

As we've noted, relational databases are defined in part by their use of the Structured Query Language (SQL). In contrast, NoSQL databases employ a host of alternative query languages that have been designed to support diverse application use cases. A partial list includes MongoDB Query Language (MQL), Couchbase's
N1QL, Elasticsearch's Query DSL, Microsoft Azure's Cosmos DB query language, and Cassandra Query Language (CQL).

In this paper, we will focus on the most widely used NoSQL query language, CQL. While CQL is the primary language for communicating with Apache Cassandra, it is also supported by a range of familiar NoSQL databases. Common CQL-compliant databases include DataStax Enterprise, Microsoft's cloud-native Azure Cosmos DB, and Amazon Keyspaces.
CQL's similarity to SQL enables developers to move between the languages with relative ease. A few distinctions between SQL and CQL include:

Joins

SQL and CQL share similar statements to store and modify data, such as Create, Alter, Drop, and Truncate commands, but unlike SQL, CQL is not designed to support joins between tables. In CQL, relations are implemented within the application, rather than within the database query.

Values versus objects

Query results are also returned differently. SQL natively returns data-typed values, usually to be read into an object one field at a time. In contrast, CQL natively returns complete objects, often serialized in extensible markup language (XML) or Javascript object notation (JSON). This makes applications responsible for parsing these objects to obtain the desired result of a query.

Scaling characteristics

In NoSQL, data is stored across nodes in a cluster based on a token range, which is a hashed value of the primary key. By using token ranges, NoSQL databases enable objects to be stored on different nodes. CQL queries are inherently more scalable than SQL queries, having been specifically designed to query across a horizontally distributed cluster of servers, rather than a single database at a time.

QueryChain™ characteristics

In BuildDB™ NoSQL, data is stored across nodes in a cluster based on a token range like other NoSQL databases. Instead of SQL or CQL, BuildDB™ utilizes a proprietary query language called QueryChain™ and is based on an API. Accessing each block of data is based on a unique query pattern based on the Chain of Events Consensus™. This consensus is hierarchical in nature and used to create zero trust between each block of data stored onto the nodes. Although QueryChain™ does not support joins (as it is not designed for relational data storage) the filtering and retrieval of data from the blockchain is event based and nearly instantaneous in comparison to relational query languages like SQL or CQL.

CONSIDERATIONS FOR SQL TO NOSQL MIGRATIONS

Data models

SQL data models follow a normalized design; different but related pieces of information are stored in 'relations,' which are separate logical tables connected by joins. NoSQL databases use denormalized data models, in which redundant copies of data are added as needed by the consuming applications. The point of denormalization is to increase performance and lower latency since the joins involved in normalized data models can introduce significant performance overhead, especially in distributed topologies.

When migrating from SQL to NoSQL, the primary key in the relational table becomes the partition key in the NoSQL table. If the RDBMS table must be joined to additional tables to retrieve the business object, those closely related tables should combine into a single NoSQL table. The NoSQL cluster ordering key determines the physical order of records, so it should be a unique value (often a composite value) that would be useful for searching.

Of course, partition keys and cluster ordering keys a not the only way data is queried. Additional indexes on the relational table provide the basis for secondary indexes or materialized views, in order to support an application's search and filtering requirements.

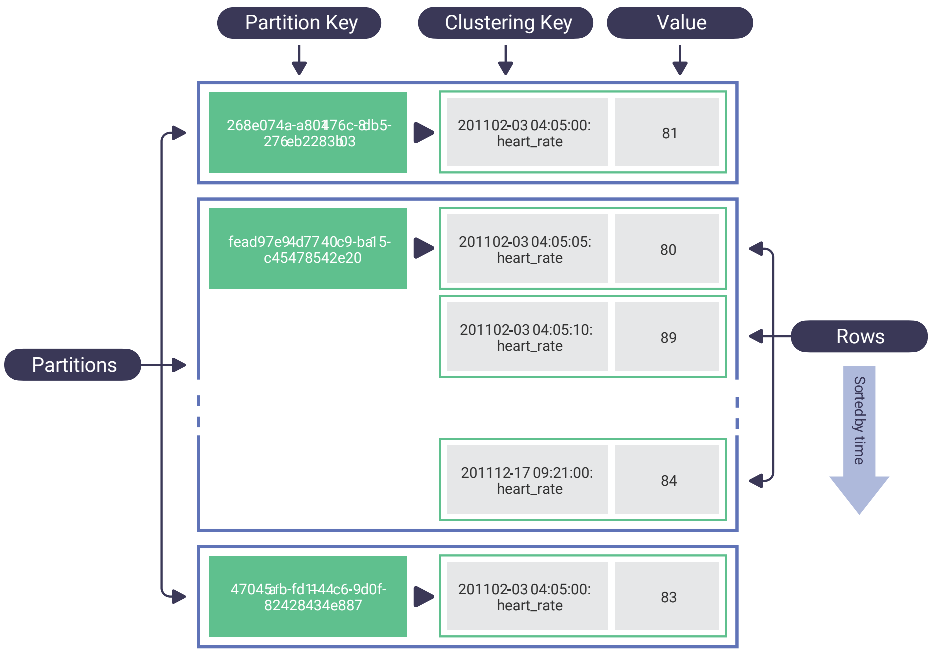

Partition Key
Clustering Key
Value
268e074a-a804476c-8db5-276eb2283b03
201102-03 04:05:00: heart_rate
81
fead97e94d7740c9-ba15-c45478542e20
201102-03 04:05:05: heart_rate
80
201102-03 04:05:10: heart_rate
89
201112-17 09:21:00: heart_rate
84
47045afb-fd1144c6-9d0f-82428434e887
201102-03 04:05:00: heart_rate
83
Partitions
Rows
Sorted by time

Figure 1 Denormalized data and Partition

QUERY PATTERNS

Relational databases are organized around data structures and relationships. In contrast, NoSQL databases are organized around query patterns. As noted above, the NoSQL partition key can be mapped to a primary key in an RDBMS.

Secondary keys and indexes can be added later. A UNIQUE constraint in a SQL database becomes a good candidate for a cluster ordering key in NoSQL.

Materialized views

Common, frequent queries against a database can become expensive. When the same query is run again and again, it makes sense to 'virtualize' the query. Materialized views address this need by enabling common queries to be represented by a database object that is continuously updated as data changes.

Secondary indexes

Secondary indexes enable queries to run against the main table using indexed values, as in an RDBMS, but it is actually implemented as a materialized view. The application is isolated from having to query the secondary index directly.

Referential integrity

Referential integrity ensures that no references between tables are broken, as occurs when a foreign key references a non-existent entry. A lack of referential integrity in a database can result in incomplete query responses, usually failing quietly, with no indication of an error. Relational databases are designed to enforce referential integrity. NoSQL databases shift the responsibility for making sure that objects are complete and correct to the API, which checks entities when loading or saving them.

MIGRATION TO NOSQL and Beyond

Hybrid conversion to NoSQL

Rather than migrating an entire RDBMS to a NoSQL database, some applications benefit from leaving some data on a relational database, while moving a subset of data to a NoSQL database. A hybrid solution that spans two database types can offer the best of both worlds. For example, in some deployments, customer account information, which is infrequently updated, will be stored in an RDBMS, while transactional or streaming data, such as IoT sensor data or telemetry, might be stored in a NoSQL database. Large, growing tables and storage for streaming data, especially in the context of event-driven architecture (EDA), are good candidates to migrate to NoSQL.

Vector Database thru NoSQL

With limited budgets working, smarter and not harder equates to futureproofing your IT investment when possible. With BuildDB™ it is (now) possible to leverage one single solution that avoids the typical database sprawl responsible for inflated IT budgets. Yesterday, it was RDBMS, today it is GraphQL and NoSQL and tomorrow it will be vector databases.

With the advent of AI, databases are forced to either adapt or fall by the side as obsolete. BuildDB™ has NoSQL, RDBMS and vector database features built into its architecture. No matter if an organization is incorporating AI and large language models or not, BuildDB™ has the architecture to be both malleable, scalable and highly performant to handle whatever the future throws at an organization.

Performance Monitoring and Tuning

As development teams become smaller and more agile, they are also increasingly sensitive to database maintenance and administrative overhead. On application-focused teams, database experts are becoming less and less common. Traditional database administration and maintenance responsibilities are often rolled into 'full-stack' developer and DevOps positions. Operational overhead is often in direct competition with product development efforts.

For these reasons, the choice of a database must take into account the expertise of the organization and the need or desire to build up internal expertise around a given technology.

The ongoing maintenance of a database requires close monitoring and frequent performance tuning. As datasets grow and application traffic increases, administrators need to keep a close eye on disk space, CPU consumption, memory allocation, and index fragmentation. Performance adjustments are proprietary to each database and often require significant dedicated expertise.

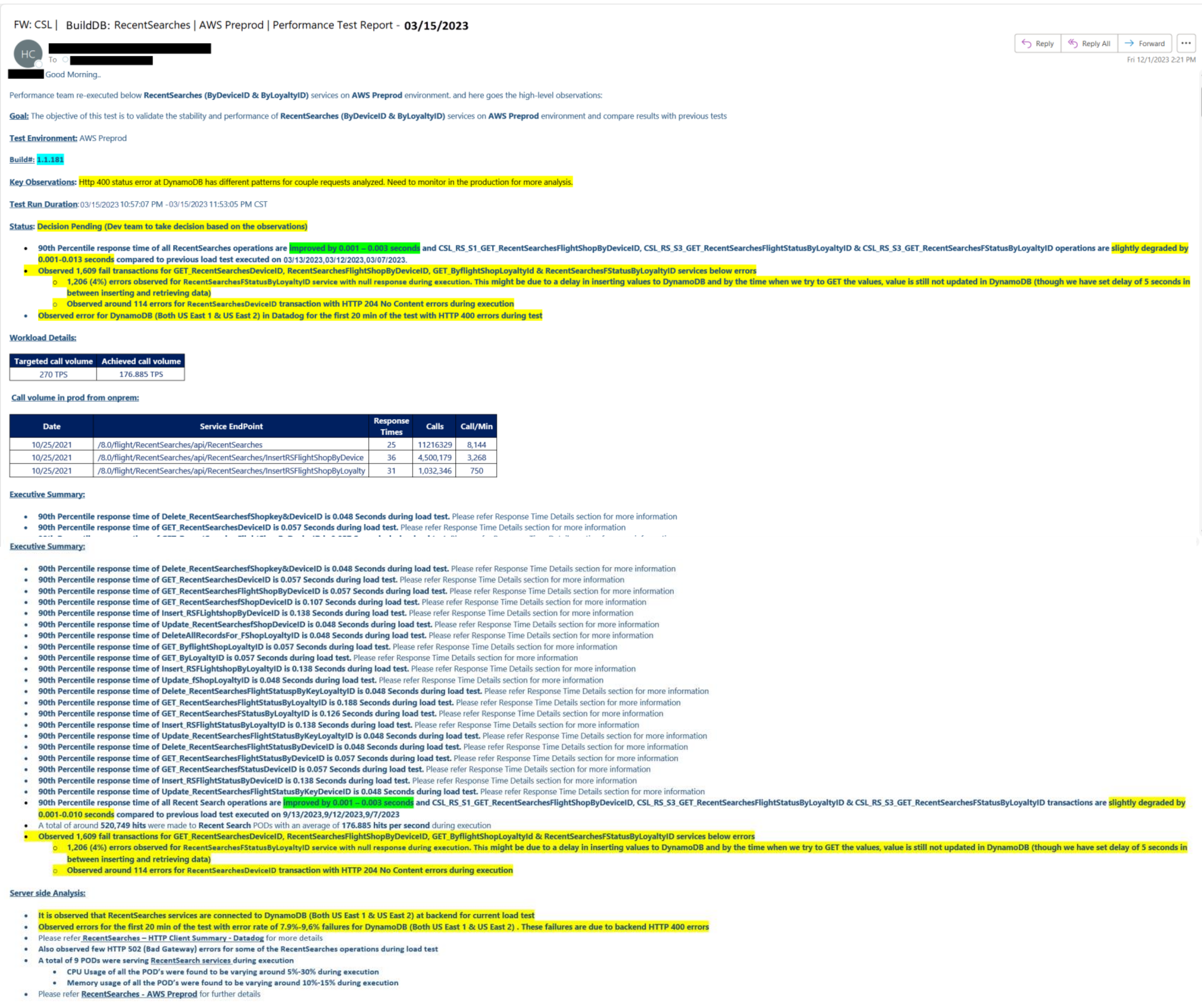
A database administrator never wants to see database utilization spike over 100%. Therefore, administrators must provide a buffer against traffic spikes by 'overprovisioning' hardware. The degree to which hardware must be overprovisioned depends on the scaling characteristics of the database. In general, NoSQL databases have a flatter and more predictable performance curve. Therefore, NoSQL databases tend to require administrators to minimize overprovisioning without compromising safety.

Performance tuning can be used to minimize overprovisioning, but it can only go so far in preventing full utilization. When performance tuning hits a wall, the modern database must be able to accommodate load automatically in a way that alleviate the costs common to cloud providers.

BUILDDB™ PERFORMANCE METRICS

BuildDB™ has been evaluated at a major US-based airline as a replacement to AWS DynamoDB. The test application used within BuildDB™ is the RecentSearches service. The RecentSearches service is a backend service called from the airline's homepage and thus receives some of the highest amount of traffic and calls from both the website and mobile devices.

This evaluation occurred on **3/15/2023** and included performance tests where AWS DynamoDB was removed and replaced with BuildDB™. The following report highlighted improvement over test conducted previously using AWS DynamoDB on **3/7/2023, 3/12/2023 and 3/13/2023**.



FW: CSL | BuildDB: RecentSearches | AWS Preprod | Performance Test Report - **03/15/2023**

[redacted]
To [redacted]

Fri 12/1/2023 2:21 PM

[redacted] Good Morning..

Performance team re-executed below **RecentSearches (ByDeviceID & ByLoyaltyID)** services on **AWS Preprod** environment. and here goes the high-level observations:

Goal: The objective of this test is to validate the stability and performance of **RecentSearches (ByDeviceID & ByLoyaltyID)** services on **AWS Preprod** environment and compare results with previous tests

Test Environment: AWS Preprod

Build#: **1.1.181**

Key Observations: Http 400 status error at DynamoDB has different patterns for couple requests analyzed. Need to monitor in the production for more analysis.

Test Run Duration: 03/15/2023 10:57:07 PM - 03/15/2023 11:53:05 PM CST

Status: **Decision Pending (Dev team to take decision based on the observations)**

- **90th Percentile response time of all RecentSearches operations are improved by 0.001 – 0.003 seconds and CSL_RS_S1_GET_RecentSearchesFlightShopByDeviceID, CSL_RS_S3_GET_RecentSearchesFlightStatusByLoyaltyID & CSL_RS_S3_GET_RecentSearchesFStatusByLoyaltyID operations are slightly degraded by 0.001-0.013 seconds compared to previous load test executed on** 03/13/2023,03/12/2023,03/07/2023.
- **Observed 1,609 fail transactions for GET_RecentSearchesDeviceID, RecentSearchesFlightShopByDeviceID, GET_ByflightShopLoyaltyId & RecentSearchesFStatusByLoyaltyID services below errors**
    - **1,206 (4%) errors observed for RecentSearchesFStatusByLoyaltyID service with null response during execution. This might be due to a delay in inserting values to DynamoDB and by the time when we try to GET the values, value is still not updated in DynamoDB (though we have set delay of 5 seconds in between inserting and retrieving data)**
    - **Observed around 114 errors for RecentSearchesDeviceID transaction with HTTP 204 No Content errors during execution**
- **Observed error for DynamoDB (Both US East 1 & US East 2) in Datadog for the first 20 min of the test with HTTP 400 errors during test**

Workload Details:

Targeted call volume	Achieved call volume
270 TPS	176.885 TPS

Call volume in prod from onprem:

Date	Service EndPoint	Response Times	Calls	Call/Min
10/25/2021	/8.0/flight/RecentSearches/api/RecentSearches	25	11216329	8,144
10/25/2021	/8.0/flight/RecentSearches/api/RecentSearches/InsertRSFlightShopByDevice	36	4,500,179	3,268
10/25/2021	/8.0/flight/RecentSearches/api/RecentSearches/InsertRSFlightShopByLoyalty	31	1,032,346	750

Executive Summary:

- **90th Percentile response time of Delete_RecentSearchesfShopkey&DeviceID is 0.048 Seconds during load test.** Please refer Response Time Details section for more information
- **90th Percentile response time of GET_RecentSearchesDeviceID is 0.057 Seconds during load test.** Please refer Response Time Details section for more information

Executive Summary:

- **90th Percentile response time of Delete_RecentSearchesfShopkey&DeviceID is 0.048 Seconds during load test.** Please refer Response Time Details section for more information
- **90th Percentile response time of GET_RecentSearchesDeviceID is 0.057 Seconds during load test.** Please refer Response Time Details section for more information

- **90th Percentile response time of GET_RecentSearchesFlightShopByDeviceID is 0.057 Seconds during load test.** Please refer Response Time Details section for more information
- **90th Percentile response time of GET_RecentSearchesfShopDeviceID is 0.107 Seconds during load test.** Please refer Response Time Details section for more information
- **90th Percentile response time of Insert_RSFLightshopByDeviceID is 0.138 Seconds during load test.** Please refer Response Time Details section for more information
- **90th Percentile response time of Update_RecentSearchesfShopDeviceID is 0.048 Seconds during load test.** Please refer Response Time Details section for more information
- **90th Percentile response time of DeleteAllRecordsFor_FShopLoyaltyID is 0.048 Seconds during load test.** Please refer Response Time Details section for more information
- **90th Percentile response time of GET_ByflightShopLoyaltyID is 0.057 Seconds during load test.** Please refer Response Time Details section for more information
- **90th Percentile response time of GET_ByLoyaltyID is 0.057 Seconds during load test.** Please refer Response Time Details section for more information
- **90th Percentile response time of Insert_RSFLightshopByLoyaltyID is 0.138 Seconds during load test.** Please refer Response Time Details section for more information
- **90th Percentile response time of Update_fShopLoyaltyID is 0.048 Seconds during load test.** Please refer Response Time Details section for more information
- **90th Percentile response time of Delete_RecentSearchesFlightStatuspByKeyLoyaltyID is 0.048 Seconds during load test.** Please refer Response Time Details section for more information
- **90th Percentile response time of GET_RecentSearchesFlightStatusByLoyaltyID is 0.188 Seconds during load test.** Please refer Response Time Details section for more information
- **90th Percentile response time of GET_RecentSearchesFStatusByLoyaltyID is 0.126 Seconds during load test.** Please refer Response Time Details section for more information
- **90th Percentile response time of Insert_RSFlightStatusByLoyaltyID is 0.138 Seconds during load test.** Please refer Response Time Details section for more information
- **90th Percentile response time of Update_RecentSearchesFlightStatusByKeyLoyaltyID is 0.048 Seconds during load test.** Please refer Response Time Details section for more information
- **90th Percentile response time of Delete_RecentSearchesFlightStatusByDeviceID is 0.048 Seconds during load test.** Please refer Response Time Details section for more information
- **90th Percentile response time of GET_RecentSearchesFlightStatusByDeviceID is 0.057 Seconds during load test.** Please refer Response Time Details section for more information
- **90th Percentile response time of GET_RecentSearchesfStatusDeviceID is 0.057 Seconds during load test.** Please refer Response Time Details section for more information
- **90th Percentile response time of Insert_RSFlightStatusByDeviceID is 0.138 Seconds during load test.** Please refer Response Time Details section for more information
- **90th Percentile response time of Update_RecentSearchesFlightStatusByKeyDeviceID is 0.048 Seconds during load test.** Please refer Response Time Details section for more information
- **90th Percentile response time of all Recent Search operations are improved by 0.001 – 0.003 seconds and CSL_RS_S1_GET_RecentSearchesFlightShopByDeviceID, CSL_RS_S3_GET_RecentSearchesFlightStatusByLoyaltyID & CSL_RS_S3_GET_RecentSearchesFStatusByLoyaltyID transactions are slightly degraded by 0.001-0.010 seconds compared to previous load test executed on 9/13/2023,9/12/2023,9/7/2023**
- A total of around **520,749 hits** were made to **Recent Search** PODs with an average of **176.885 hits per second** during execution
- **Observed 1,609 fail transactions for GET_RecentSearchesDeviceID, RecentSearchesFlightShopByDeviceID, GET_ByflightShopLoyaltyId & RecentSearchesFStatusByLoyaltyID services below errors**
    - **1,206 (4%) errors observed for RecentSearchesFStatusByLoyaltyID service with null response during execution. This might be due to a delay in inserting values to DynamoDB and by the time when we try to GET the values, value is still not updated in DynamoDB (though we have set delay of 5 seconds in between inserting and retrieving data)**
    - **Observed around 114 errors for RecentSearchesDeviceID transaction with HTTP 204 No Content errors during execution**

Server side Analysis:

- **It is observed that RecentSearches services are connected to DynamoDB (Both US East 1 & US East 2) at backend for current load test**
- **Observed errors for the first 20 min of the test with error rate of 7.9%-9,6% failures for DynamoDB (Both US East 1 & US East 2) . These failures are due to backend HTTP 400 errors**
- Please refer **RecentSearches – HTTP Client Summary - Datadog** for more details
- **Also observed few HTTP 502 (Bad Gateway) errors for some of the RecentSearches operations during load test**
- **A total of 9 PODs were serving RecentSearch services during execution**
    - **CPU Usage of all the POD's were found to be varying around 5%-30% during execution**
    - **Memory usage of all the POD's were found to be varying around 10%-15% during execution**
- Please refer **RecentSearches - AWS Preprod** for further details

The performance report results of BuildDB™ are as follows:

Statistical Summary:

Please find the below statistical summary of **Recent Search** operations in **AWS Preprod** environment:

Statistics	Recent Search
Total Test period (CST)	03/15/2023 10:57:07 PM - 03/15/2023 11:53:05 PM CST
Total Hits	520,749
Average Hits/Sec	176.885
Total Pass Transactions	519,298
Total Fail Transactions	1,463
Run ID	16210

CSL_RS_S4_GET_RecentSearchesfStatusDeviceID	0.036	0.05	0.478	**0.057**	29,496	0	0
CSL_RS_S4_Insert_RSFlightStatusByDeviceID	0.049	0.128	3.124	**0.138**	29,491	0	0
CSL_RS_S4_Update_RecentSearchesFlightStatusByKeyDeviceID	0.029	0.041	0.406	**0.048**	29,498	0	0

Response Time Details:

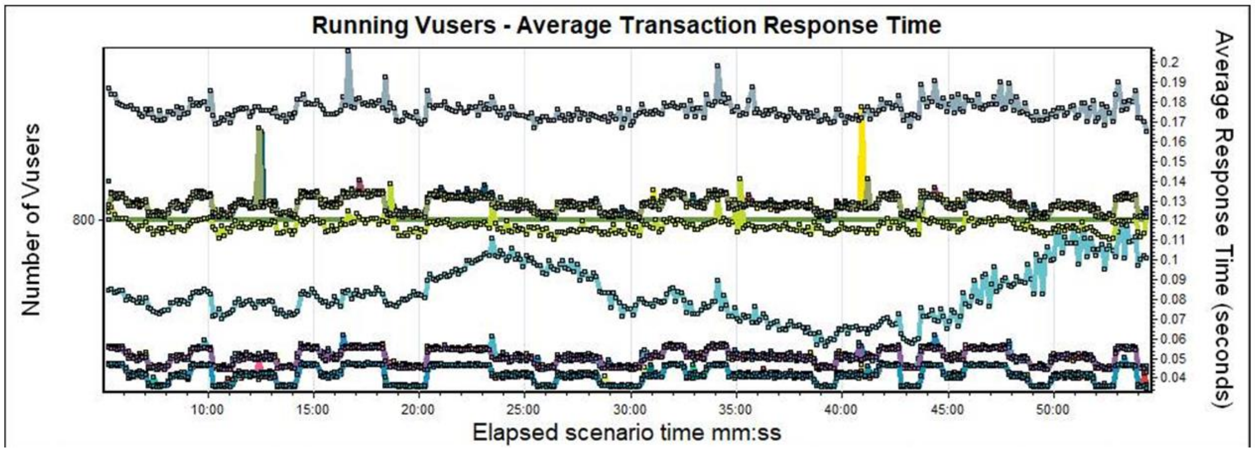
Following table depicts the 90th Percentile response time and transaction details of **Recent Search** service executed on **AWS Preprod** environment.

Transaction Name	Minimum	Average	Maximum	90 Percent	Pass	Fail	Stop
CSL_RS_S1_Delete_RecentSearchesfShopkey&DeviceID	0.029	0.041	0.345	**0.048**	19,625	0	0
CSL_RS_S1_GET_RecentSearchesDeviceID	0.037	0.051	0.198	**0.057**	19,801	114	0
CSL_RS_S1_GET_RecentSearchesFlightShopByDeviceID	0.037	0.051	0.429	**0.057**	19,627	180	0
CSL_RS_S1_GET_RecentSearchesfShopDeviceID	0.039	0.083	1.034	**0.108**	19,620	0	0
CSL_RS_S1_Insert_RSFLightshopByDeviceID	0.053	0.128	0.501	**0.138**	19,909	0	0
CSL_RS_S1_Update_RecentSearchesfShopDeviceID	0.029	0.041	0.108	**0.048**	19,630	0	0
CSL_RS_S2_DeleteAllRecordsFor_FShopLoyaltyId	0.029	0.041	0.349	**0.048**	23,668	0	0
CSL_RS_S2_GET_ByflightShopLoyaltyId	0.037	0.051	0.26	**0.057**	23,674	2	0
CSL_RS_S2_GET_ByLoyaltyID	0.037	0.051	0.352	**0.057**	23,670	0	0
CSL_RS_S2_Insert_RSFLightshopByLoyaltyID	0.052	0.128	3.145	**0.138**	23,661	0	0
CSL_RS_S2_Update_fShopLoyaltyId	0.029	0.041	0.42	**0.048**	23,675	0	0
CSL_RS_S3_Delete_RecentSearchesFlightStatuspByKeyLoyaltyID	0.029	0.041	0.493	**0.048**	28,748	0	0
CSL_RS_S3_GET_RecentSearchesFlightStatusByLoyaltyID	0.056	0.176	1.011	**0.188**	28,746	0	0
CSL_RS_S3_GET_RecentSearchesFStatusByLoyaltyID	0.083	0.117	1.651	**0.126**	28,752	1,206	0
CSL_RS_S3_Insert_RSFlightStatusByLoyaltyID	0.049	0.128	3.132	**0.138**	29,949	0	0
CSL_RS_S3_Update_RecentSearchesFlightStatusByKeyLoyaltyID	0.029	0.041	0.379	**0.048**	28,752	0	0
CSL_RS_S4_Delete_RecentSearchesFlightStatusByDeviceID	0.029	0.041	0.089	**0.048**	29,488	0	0
CSL_RS_S4_GET_RecentSearchesFlightStatusByDeviceID	0.036	0.05	0.371	**0.057**	29,499	0	0
CSL_RS_S4_GET_RecentSearchesfStatusDeviceID	0.036	0.05	0.478	**0.057**	29,496	0	0
CSL_RS_S4_Insert_RSFlightStatusByDeviceID	0.049	0.128	3.124	**0.138**	29,491	0	0
CSL_RS_S4_Update_RecentSearchesFlightStatusByKeyDeviceID	0.029	0.041	0.406	**0.048**	29,498	0	0

Comparison with Previous Tests:

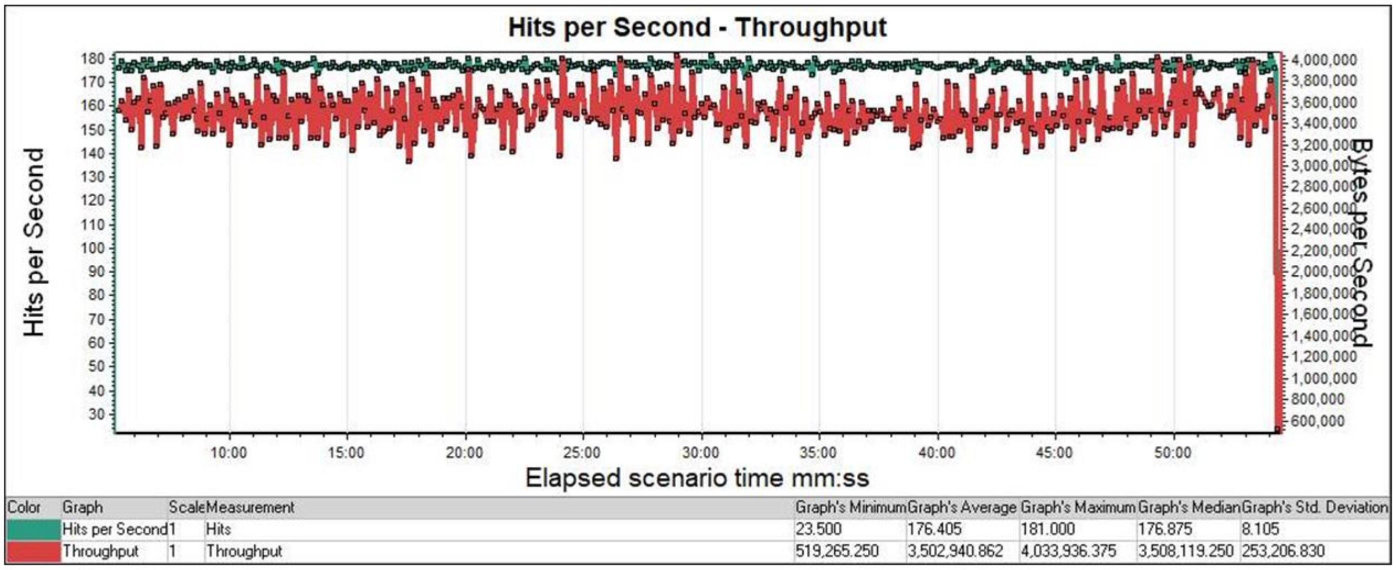
Transaction Name	90 Percentile				Difference		
	3/15/2023 BuildDB	3/13/2023 DynamoDB	3/12/2023 DynamoDB	3/7/2023 DynamoDB	3/15/2023 vs 3/13/2023	3/15/2023 vs 3/12/2023	3/15/2023 vs 3/7/2023
CSL_RS_S1_Delete_RecentSearchesfShopkey&DeviceID	**0.048**	0.049	0.049	0.048	-0.001	-0.001	0
CSL_RS_S1_GET_RecentSearchesDeviceID	**0.057**	0.058	0.058	0.057	-0.001	-0.001	0
CSL_RS_S1_GET_RecentSearchesFlightShopByDeviceID	**0.057**	0.058	0.058	0.057	-0.001	-0.001	0
CSL_RS_S1_GET_RecentSearchesfShopDeviceID	**0.108**	0.097	0.097	0.095	0.011	0.011	0.013
CSL_RS_S1_Insert_RSFLightshopByDeviceID	**0.138**	0.141	0.141	0.139	-0.003	-0.003	-0.001
CSL_RS_S1_Update_RecentSearchesfShopDeviceID	**0.048**	0.049	0.049	0.048	-0.001	-0.001	0
CSL_RS_S2_DeleteAllRecordsFor_FShopLoyaltyId	**0.048**	0.049	0.049	0.048	-0.001	-0.001	0
CSL_RS_S2_GET_ByflightShopLoyaltyId	**0.057**	0.058	0.058	0.067	-0.001	-0.001	-0.01
CSL_RS_S2_GET_ByLoyaltyID	**0.057**	0.059	0.059	0.091	-0.002	-0.002	-0.034
CSL_RS_S2_Insert_RSFLightshopByLoyaltyID	**0.138**	0.141	0.141	0.138	-0.003	-0.003	0
CSL_RS_S2_Update_fShopLoyaltyId	**0.048**	0.049	0.049	0.048	-0.001	-0.001	0
CSL_RS_S3_Delete_RecentSearchesFlightStatuspByKeyLoyaltyID	**0.048**	0.049	0.049	0.048	-0.001	-0.001	0
CSL_RS_S3_GET_RecentSearchesFlightStatusByLoyaltyID	**0.188**	0.186	0.186	0.186	0.002	0.002	0.002
CSL_RS_S3_GET_RecentSearchesFStatusByLoyaltyID	**0.126**	0.125	0.125	0.125	0.001	0.001	0.001
CSL_RS_S3_Insert_RSFlightStatusByLoyaltyID	**0.138**	0.14	0.14	0.139	-0.002	-0.002	-0.001
CSL_RS_S3_Update_RecentSearchesFlightStatusByKeyLoyaltyID	**0.048**	0.049	0.049	0.048	-0.001	-0.001	0
CSL_RS_S4_Delete_RecentSearchesFlightStatusByDeviceID	**0.048**	0.049	0.049	0.048	-0.001	-0.001	0
CSL_RS_S4_GET_RecentSearchesFlightStatusByDeviceID	**0.057**	0.058	0.058	0.057	-0.001	-0.001	0
CSL_RS_S4_GET_RecentSearchesfStatusDeviceID	**0.057**	0.058	0.058	0.057	-0.001	-0.001	0
CSL_RS_S4_Insert_RSFlightStatusByDeviceID	**0.138**	0.141	0.141	0.139	-0.003	-0.003	-0.001
CSL_RS_S4_Update_RecentSearchesFlightStatusByKeyDeviceID	**0.048**	0.049	0.049	0.048	-0.001	-0.001	0

- **Below snapshots depicts the average response time trend of Recent Search services executed during execution. Response time of most of the services are consistent throughout the execution**

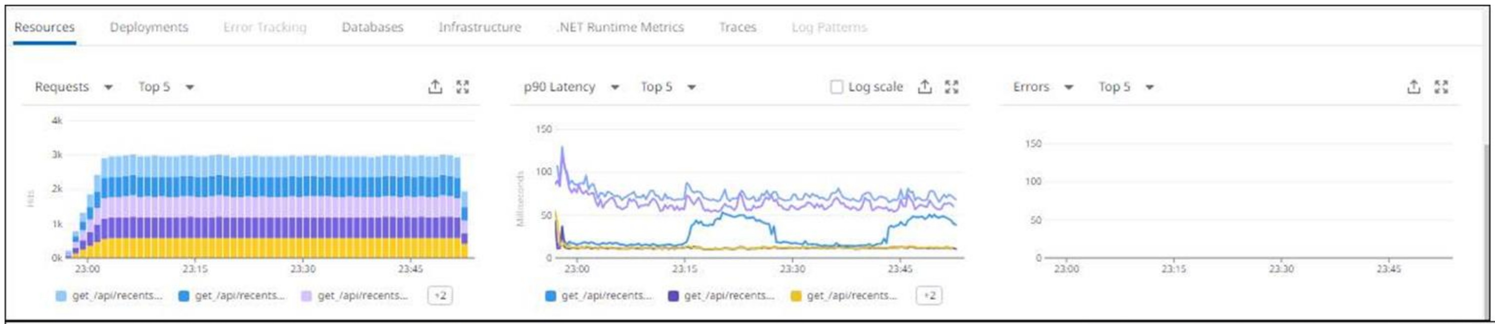


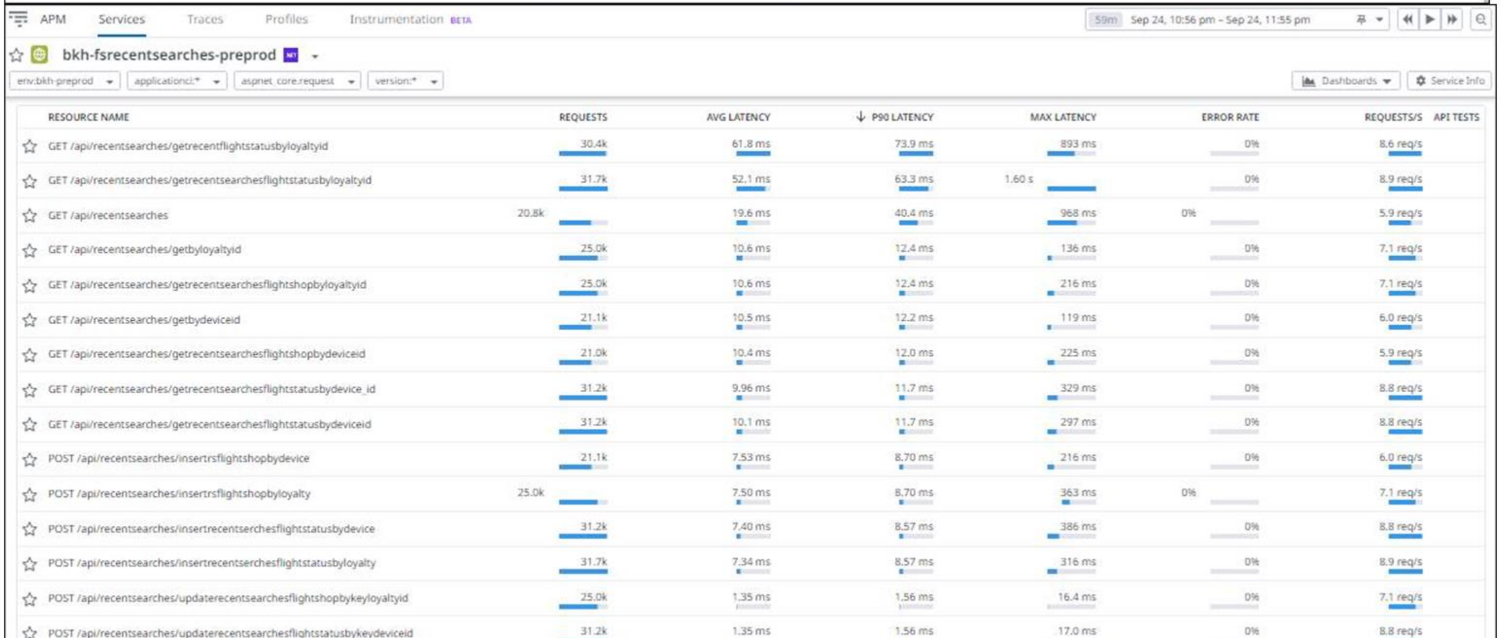
Hits per second vs Throughput:

- **Hits per second and Throughput is proportional throughout the test and constant during the steady state.**
    - A total of around **520,749 hits** were made to **Recent Search** PODs with an average of **176.885 hits per second** during execution



Color	Graph	Scale	Measurement	Graph's Minimum	Graph's Average	Graph's Maximum	Graph's Median	Graph's Std. Deviation
🟩	Hits per Second	1	Hits	23.500	176.405	181.000	176.875	8.105
🟥	Throughput	1	Throughput	519,265.250	3,502,940.862	4,033,936.375	3,508,119.250	253,206.830





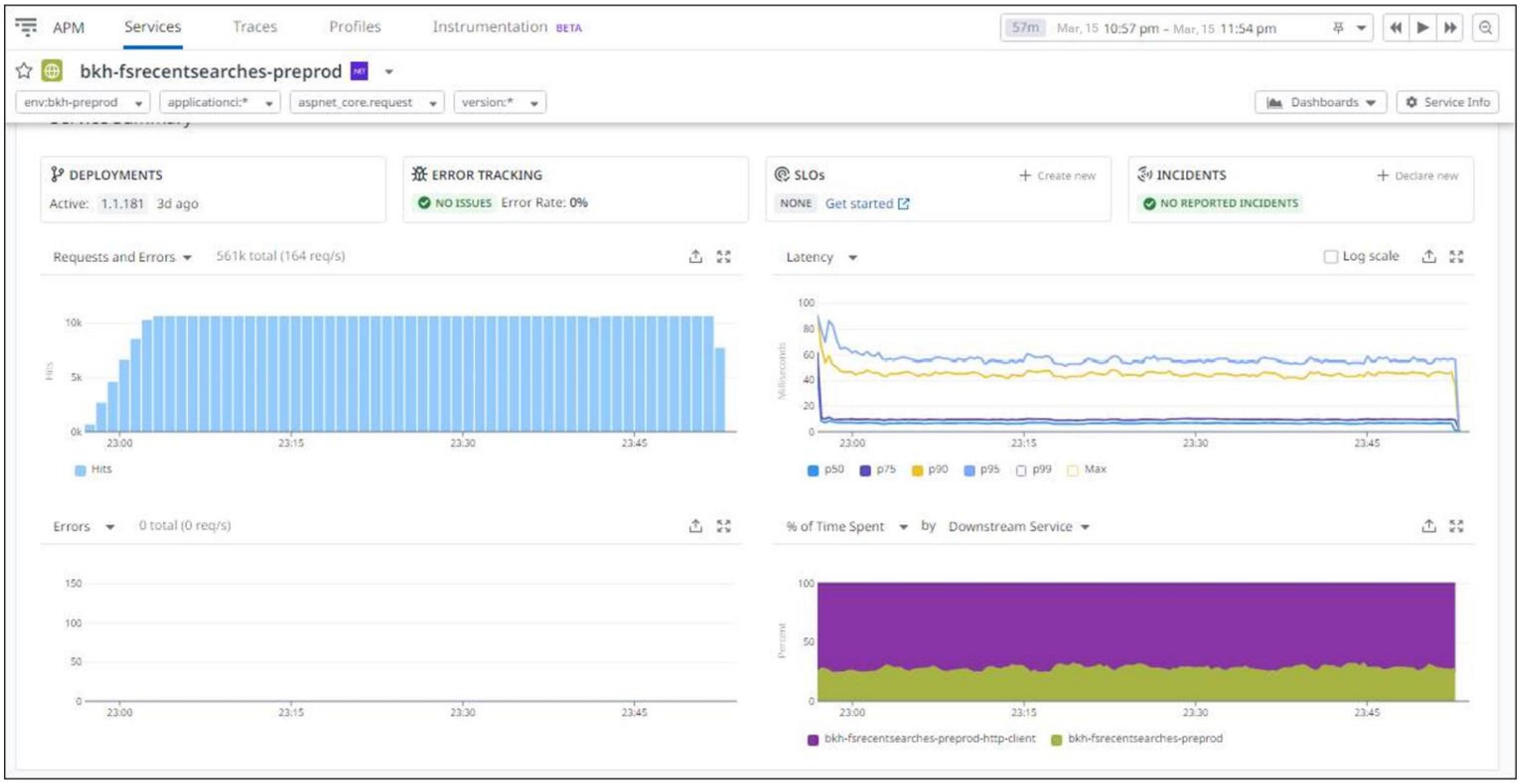
bkh-fsrecentsearches-preprod

env:bkh-preprod | applicationci:* | aspnet_core.request | version:*

RESOURCE NAME	REQUESTS	AVG LATENCY	↓ P90 LATENCY	MAX LATENCY	ERROR RATE	REQUESTS/S	API TESTS
GET /api/recentsearches/getrecentflightstatusbyloyaltyid	30.4k	61.8 ms	73.9 ms	893 ms	0%	8.6 req/s	
GET /api/recentsearches/getrecentsearchesflightstatusbyloyaltyid	31.7k	52.1 ms	63.3 ms	1.60 s	0%	8.9 req/s	
GET /api/recentsearches	20.8k	19.6 ms	40.4 ms	968 ms	0%	5.9 req/s	
GET /api/recentsearches/getbyloyaltyid	25.0k	10.6 ms	12.4 ms	136 ms	0%	7.1 req/s	
GET /api/recentsearches/getrecentsearchesflightshopbyloyaltyid	25.0k	10.6 ms	12.4 ms	216 ms	0%	7.1 req/s	
GET /api/recentsearches/getbydeviceid	21.1k	10.5 ms	12.2 ms	119 ms	0%	6.0 req/s	
GET /api/recentsearches/getrecentsearchesflightshopbydeviceid	21.0k	10.4 ms	12.0 ms	225 ms	0%	5.9 req/s	
GET /api/recentsearches/getrecentsearchesflightstatusbydevice_id	31.2k	9.96 ms	11.7 ms	329 ms	0%	8.8 req/s	
GET /api/recentsearches/getrecentsearchesflightstatusbydeviceid	31.2k	10.1 ms	11.7 ms	297 ms	0%	8.8 req/s	
POST /api/recentsearches/insertrsflightshopbydevice	21.1k	7.53 ms	8.70 ms	216 ms	0%	6.0 req/s	
POST /api/recentsearches/insertrsflightshopbyloyalty	25.0k	7.50 ms	8.70 ms	363 ms	0%	7.1 req/s	
POST /api/recentsearches/insertrecentserchesflightstatusbydevice	31.2k	7.40 ms	8.57 ms	386 ms	0%	8.8 req/s	
POST /api/recentsearches/insertrecentserchesflightstatusbyloyalty	31.7k	7.34 ms	8.57 ms	316 ms	0%	8.9 req/s	
POST /api/recentsearches/updaterecentsearchesflightshopbykeyloyaltyid	25.0k	1.35 ms	1.56 ms	16.4 ms	0%	7.1 req/s	
POST /api/recentsearches/updaterecentsearchesflightstatusbykeydeviceid	31.2k	1.35 ms	1.56 ms	17.0 ms	0%	8.8 req/s	

Datadog Observations:

- Please find below analysis report with screenshots from Datadog for the test run duration. Please refer **RecentSearches - AWS Preprod** for further details



BUILDDB™ NOSQL: SCALE-UP OF RDBMS

Arriving at a database solution that is highly available and resilient but queryable like relational databases is challenging. BuildDB™ is the result of a database that extracts maximum performance from modern hardware to deliver predictable, low latency data results with advanced features like filtering, grouping and aggregate count functions.

BuildDB™ minimizes operational overhead and significantly reduces TCO.

Many IT organizations have adhered to the following tradeoffs between SQL and NoSQL databases.

	SQL	NoSQL
Orientation	Relational	Generally non-relational
Schema	Strict and rigid schema design and data Normalization.	Loose and more varied designs for unstructured and semi-structured data; data is generally denormalized
Language	Structured Query Language (SQL) for defining, reading and manipulating data. Supports JOIN statements to relate data across tables.	There are different languages for querying, some quite similar to SQL, such as Cassandra Query Language (CQL) for wide column databases, or others radically different, such as using object-oriented JSON for document databases.
Scalability	Vertically scalable. Loads on a single server can be increased with CPU, RAM or SSD.	Generally designed for horizontal scalability. Increased traffic can be handled by adding more servers in the database. This is useful for large and frequently changing datasets.
Structure	Table-based, which is efficient for applications using multi-row transactions or systems that were built with a relational structure.	NoSQL database structure is variable, and can be based on documents, key-value pairs, graph structures or wide-column stores.

Easy Integration

Unlike other SQL-based or CQL-based databases, BuildDB™ provides simple, yet powerful language called QueryChain™ to encapsulate filters and functions to retrieve and aggregate data results. Leveraging filters and functions can be achieved either through the API Client or via Postman or OpenAPI requests.

ABOUT BuildDB™

BuildDB™ is the real-time blockchain-enabled big data NoSQL database. BuildDB™ operates as an excellent replacement for storage solutions like:

- AWS DynamoDB
- Azure CosmoDB
- Google Cloud Datastore
- Oracle NoSQL Database
- Redis
- MongoDB
- Couchbase
- RavenDB
- SupaBase

BuildDB™ solves the performance challenges of the aforementioned storage solutions due to its unique architecture that alleviates indexes and the need to tune pre-provisioned and auto-scaling resources common to cloud offerings. Adopting BuildDB™ allows organizations to realize an order-of-magnitude performance improvements while reducing total cost of ownership (TCO) by a minimum of 40%.
BuildDB is available as a Software as a Service (SaaS) offering on the AWS Marketplace in three distinct deployment sizes.

- BuildDB™ provides an API client accessible via a Nuget package under the name: **BuildDB™Client:** https://www.nuget.org/packages/EBBuildClient/

- BuildDB™ is available for deployment from the AWS Marketplace under the name: **BuildDB™** and is located under SaaS/Blockchain-enabled Storage:

https://aws.amazon.com/marketplace/pp/prodview-7ulzw27yljmlu?sr=0-1&ref_=beagle&applicationId=AWSMPContessa

Suite 101
United States Headquarters
12574 Flagler Center Blvd

Jacksonville, FL 32258
Email: support@everythingblockchain.io

EverythingBlockchain.IO