MARCH 2023

7 NoSQL Considerations

MongoDB.

Table of Contents

Table of Contents

Introduction

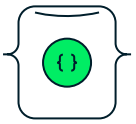
Data and software are at the heart of business today. But for many organizations, realizing the full potential of the digital economy remains a significant challenge. Since the inception of MongoDB, we've understood that the biggest challenge organizations face is working with data:

- Demands for higher productivity and faster time to market are being held back by rigid relational data models that are mismatched to modern code and impose complex interdependencies among engineering teams.
- Organizations are unable to work with, and extract insights from, massive increases in the new and rapidly changing structured, semi-structured, and polymorphic data generated by today's applications.
- Monolithic and fragile legacy databases inhibit the wholesale shift to distributed systems and cloud computing that deliver the resilience and scale businesses need, making it harder to satisfy new regulatory requirements for data privacy.
- Previously separate transactional, analytical, search, and mobile workloads are converging to create rich, data-driven applications and customer experiences. However, each workload traditionally has been powered by its own database, creating duplicated data silos stitched together with fragile ETL pipelines accessed by different APIs.

To address these limitations, several non-tabular alternatives to relational databases have emerged. Generally referred to as NoSQL databases, these systems discard the foundation that has made relational databases so useful for generations of applications: expressive query language, secondary indexes, and strong consistency. NoSQL databases share several key characteristics, including a more flexible data model, higher scalability, and superior performance.

Although the term NoSQL often is used as an umbrella category for all non-tabular databases, it's too vague and poorly defined to be a useful descriptor of the underlying data model. Primarily, it neglects the trade-offs NoSQL databases make to achieve flexibility, scalability, and performance.

To help technology decision-makers navigate the complex and evolving domain of NoSQL and non-tabular databases, we've highlighted the key differences between them in this white paper. We also explore critical considerations based on seven dimensions that define these systems: data model; query model; consistency and transactional model; APIs; mobile data; data platform; and commercial support, community strength, and freedom from lock-in.



Key Consideration #1: Data Model

The primary way in which NoSQL databases differ from relational databases is the data model. Although there are dozens of NoSQL databases, they generally fall into three categories: key-value or wide-column, graph, and document.

Key-Value and Wide-Column Databases

From a data model perspective, key-value databases are the most basic type of non-tabular database. Every item in the database is stored as an attribute name or key, together with its value. The value, however, is entirely opaque to the system – data can be queried only by the key. This model can be useful for representing polymorphic and unstructured data because the database does not enforce a set schema across key-value pairs.

Wide-column databases use a sparse, distributed, multidimensional, sorted map to store data. Each record can vary in the number of columns that are stored. Columns can be grouped together into column families or spread across multiple families. Data is retrieved by primary key per column family.

Applications

Key-value and wide-column databases are useful for a specialized set of applications that query data by using a single key value. The appeal of these systems is their performance and scalability, which can be highly optimized due to the simplicity of the data access patterns and opacity of the data itself.

Examples

Redis, Amazon DynamoDB (key-value); Apache HBase, Apache Cassandra (wide-column)

Graph Database

Graph databases use graph structures with nodes, edges, and properties to represent data relationships. In essence, data is modeled as a network of relationships among specific elements. Their main appeal is in their ability to model and navigate relationships among entities in an application. This makes graph databases incredibly efficient for finding patterns, making predictions, and creating solutions. Flexible schemas allow developers to easily make changes to graph databases as requirements change. This is especially valuable for agile teams building modern applications.

Applications

Graph databases are useful in cases where traversing relationships is core to the application, such as navigating social network connections, network topologies, or supply chains. Other use cases include detecting fraud, building recommendation engines, managing IT networks, and computing graph algorithms between data.

Examples

Neo4j, Amazon Neptune

Document Database

Whereas relational databases store data in rows and columns, document databases store data in documents by using JavaScript Object Notation (JSON), a text-based data interchange format popular among developers. Documents provide an intuitive and natural way to model data that closely aligns with object-oriented programming – each document is effectively an object that matches the objects developers work with in code. Documents contain one or more fields, and each field contains a typed value such as a string, date, binary, decimal value, or array. Rather than spreading out a record across multiple columns and tables connected with foreign keys, each record is stored along with its associated (i.e., related) data in a single, hierarchical document. This model accelerates developer productivity, simplifies data access, and, in many cases, eliminates the need for expensive join operations and complex abstraction layers such as object relational mapping (ORM).

The schema of a relational database is defined by tables; in a document database, the notion of a schema is dynamic – each document can contain different fields. This flexibility can be particularly helpful for modeling data where structures can change between each record – i.e., polymorphic data. It also makes it easier to evolve an application during its life cycle, such as by adding new fields.

False assumptions about NoSQL databases are prevalent. One of the most common assumptions is that NoSQL databases are schemaless and that data modeling is not necessary. This impression comes from the fact that NoSQL databases are ideally suited for storing unstructured data, and because they dispense with the tabular structure of relational databases. This is why NoSQL databases are often referred to as non-relational databases. Ideally, schema design and data modeling in a document database are based on the access patterns for the data you're working with. In essence, data that is accessed together should be stored together. While a document database does allow you to store data without defining what it is, the shape of that data matters if you plan to do more than simply retrieve whole documents by keys. In some cases, schema design is irrelevant – for example, if you're simply storing pre-existing documents – and a simple key-value store would suffice. On the other hand, if you expect to filter, modify, and retrieve data efficiently, schema design and data modeling are essential.

Developers are often in the best position to know the data access patterns for their applications. Document schemas can increase performance for a given set of hardware by reducing computation, I/O operations, and contention between users. With today's pay-as-you-go cloud pricing, that's an important consideration. What really differentiates a document database from relational databases is the ability to co-locate related data in the atomic unit of storage so multiple values for an attribute can exist within a single record rather than being broken up into rows and stored independently. A document database with a properly designed schema enables you to filter and retrieve data with minimal computational overhead and in a single I/O operation. This can make finding and retrieving data far faster and less expensive.

Applications

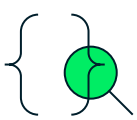
Document databases are useful for a wide variety of applications due to the flexibility of the data model, the ability to query on any field, and the natural mapping of the document model to objects in modern programming languages.

Examples

MongoDB, Azure CosmosDB, Apache CouchDB

Takeaways

- Documents are a superset of other data models, so they support a wider variety of data types and use cases.
- Key-value and wide-column databases are opaque to the system – only the primary key can be queried.
- The wide-column model provides more granular access to data than the key-value model but is less flexible than the document model.
- Key-value and wide-column databases are desired for their simplicity, performance, and scalability.
- Graph databases use nodes to represent relationships such as parent-child, actions, and ownership.
- Graph databases are most useful for navigating social connections, network topologies, and supply chains.
- The document data model has the broadest applicability.
- The document data model is the most natural because it maps directly to objects in modern object-oriented languages.
- Despite common assumptions, data modeling and schemas are critical elements of document databases if you expect to filter, modify, and retrieve data efficiently.



Key Consideration #2: Query Model

Each application has its own query requirements. In some cases, a basic query model may be appropriate, where the application accesses records based on a primary key. For most applications, however, it's important to have the ability to query based on several different values in each record. For example, an application that stores data about customers may need to query by customer name, company name, size, sales value, ZIP code, state, or aggregations of multiple values. It's also common for applications to update multiple records, including one or more individual fields. To satisfy these requirements, the database needs to be able to perform queries based on secondary indexes. In these cases, a document database often will be the most appropriate solution.

Key-Value and Wide-Column Databases

Key-value and wide-column databases provide the ability to retrieve and update data based on a single or limited range of keys. For querying other values, developers need to build and maintain their own indexes. Some products provide limited support for secondary indexes, but with caveats. To perform an update in these systems, multiple round trips may be necessary – first to find the record, then to update it, and then to update the index. The index, therefore, may not be consistent with the base data, which can then return stale or deleted data, dramatically increasing application complexity and decreasing the accuracy of query results. In these systems, the update may be implemented as a complete rewrite of the entire record at the client, regardless of whether a single attribute or the entire record has changed.

Graph Database

Graph databases provide rich query models in which simple and complex relationships can be interrogated to make direct and indirect inferences about the data in the system. Although relationship analysis tends to be efficient, other types of analysis are less optimal. As a result, graph databases are rarely used for general-purpose, operational applications. Rather, they're often coupled with document or relational databases to surface graph-specific data structures and queries. For use cases involving multiple query patterns, there's an option to employ a multimodel database where different data models and query types are available within a single platform. For example, MongoDB offers the $graphLookup aggregation stage for graph processing natively within the database. $graphLookup enables efficient traversals across graphs, trees, and hierarchical data to uncover patterns and surface previously unidentified connections.

Document Database

Document databases provide the ability to query and update any field within a document, although capabilities in this domain vary. Some databases, such as MongoDB, provide a rich set of indexing options to optimize a wide variety of queries and to automate data management, including text, geospatial, compound, sparse, wildcard, time to live (TTL), and unique indexes. Some document databases support real-time analytics against data in place without having to replicate to a dedicated analytics application or search engine. MongoDB, for instance, provides an aggregation framework for developers to create processing pipelines for data analytics and transformations via faceted search, joins, unions, geospatial processing, materialized views, and graph traversals. To update data, MongoDB provides expressive update methods that enable developers to perform complex manipulations against matching elements of a document – including elements embedded in nested arrays – all in a single transactional update operation.

Takeaways

- The biggest difference between non-tabular databases lies in the ability to query data efficiently.
- Key-value databases and wide-column stores provide a single means of accessing data: primary keys. Although fast, they offer limited query functionality and may impose additional development costs and application-level requirements to support more complex query patterns.
- Document databases provide the richest query functionality, which allows them to address a wide variety of operational and real-time analytics applications.

Key Consideration #3: Consistency and Transactional Model

Most NoSQL systems maintain multiple copies of data for availability and scalability purposes. These databases can impose different guarantees on the consistency of data across copies. NoSQL databases tend to be categorized as either strongly consistent or eventually consistent. With a strongly consistent system, writes by the application are immediately visible in subsequent queries. With an eventually consistent system, the visibility of writes depends on which data replica is serving the query. For example, when reflecting inventory levels for products in a product catalog, with a consistent system each query will see the current inventory as it's updated by the application. With an eventually consistent system, the inventory levels may not be accurate for a query at a given time but will eventually become accurate as data is replicated across all nodes in the database cluster. For this reason, application code can be different for eventually consistent systems – rather than updating the inventory by taking the current inventory and subtracting one, for example, developers are encouraged to issue idempotent queries that explicitly set the inventory level. Developers also need to build additional control logic in their apps to handle potentially stale or deleted data.

Most NoSQL systems offer atomicity guarantees at the level of an individual record. Atomicity is one of four transaction properties that constitute ACID transactions. The four properties in an ACID transaction are:

- Atomicity
- Consistency
- Isolation
- Durability

The point of ACID transactions is to guarantee data validity despite errors, power failures, and other mishaps. Atomicity is an assurance that database operations are indivisible or irreducible such that either all operations complete or none complete. Because these databases can combine related data that otherwise would be modeled across separate parent-child tables in a tabular schema, atomic single-record operations provide transaction semantics that meet the data integrity needs of the majority of applications.

It's important to note that some developers and database administrators have been conditioned by 40 years of relational data modeling to assume multirecord transactions are a requirement for any database, regardless of the underlying data model. Some are concerned that although multidocument transactions aren't needed by their apps today, they might be in the future. And for some workloads, support for ACID transactions across multiple records is required.

MongoDB added support for multidocument ACID transactions in 2018 so developers could address a wider range of use cases with the familiarity of how transactions are handled in relational databases. Through snapshot isolation, transactions provide a consistent view of data and enforce all-or-nothing execution. MongoDB is relatively unique in offering the transactional guarantees of traditional relational databases with the flexibility and scale that come from NoSQL databases.

Consistent Systems

Applications can have different requirements for data consistency. For many applications, it's imperative for data to be consistent at all times. Because development teams have worked under a model of consistency with relational databases for decades, this approach is more natural and familiar. In other cases, eventual consistency is an acceptable trade-off for the flexibility it allows in the system's availability.

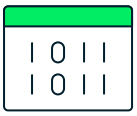
Document and graph databases can be consistent or eventually consistent. MongoDB provides tunable consistency. By default, data is consistent – all writes and reads access the primary copy of the data. As an option, read queries can be issued against secondary copies where data may be eventually consistent if the write operation has not yet been synchronized with the secondary copy; the consistency choice is made at the query level.

Eventually Consistent Systems

With eventually consistent systems, there is a period of time during which copies of data are not synchronized. This may be acceptable for read-only applications and data stores that do not change often, such as historical archives or write-intensive use cases where the database is capturing logs that will be read at a later point in time, often after it has been moved into another system that offers richer query capabilities. Typically, key-value and wide-column databases are considered eventually consistent. Eventually consistent systems must be able to accommodate conflicting updates in individual records. Because writes can be applied to any copy of the data, it is possible and not uncommon for writes to conflict with one another when the same attribute is updated on different nodes. Some systems use vector clocks to determine the order of events and ensure the most recent operation prevails in the case of a conflict. However, the older value may already have been committed back to the application. Other systems retain all conflicting values and push the responsibility of resolving conflicts back to the user. For these reasons, inserts tend to perform well in eventually consistent systems, but updates and deletes can involve trade-offs that complicate the application.

Takeaways

- Different consistency models pose different trade-offs for applications in the areas of consistency, availability, and performance.
- MongoDB provides tunable consistency, defined at the query level.
- Eventually consistent systems provide some advantages for inserts at the cost of making reads, updates, and deletes more complex, while incurring performance overhead via read repairs and compactions.
- Most NoSQL databases provide single-record atomicity. This is sufficient for many applications but not all.
- MongoDB provides multidocument ACID guarantees, making it easier to address a range of use cases with a single data platform.



Key Consideration #4: Interfaces

There is no single standard for interfacing with NoSQL databases. Each presents different designs and capabilities for application developers. The maturity of APIs can affect the time and cost required for developing and maintaining the application and database.

Idiomatic Drivers

Programming languages provide different paradigms for working with data. Idiomatic drivers are created by development teams that are experts in a given language and know how programmers prefer to work within a language. This approach can also provide efficiencies for accessing and processing data by leveraging specific features in a programming language. Because idiomatic drivers are easier for developers to learn and use, they reduce the onboarding time required for teams to begin working with a database. For example, idiomatic drivers provide direct interfaces to set and get documents or fields within documents. With other types of interfaces, it may be necessary to retrieve and parse entire documents and navigate to specific values in order to set or get a field.

MongoDB supports idiomatic drivers in more than a dozen languages including Java, .NET, Ruby, Node.js, Python, PHP, C, C++, C#, JavaScript, Go, Rust, and Scala. Dozens of other drivers are supported by the developer community.

APIs

Some systems provide representational state transfer (RESTful) interfaces. This approach has the appeal of simplicity and familiarity, although it also relies on the inherent latencies associated with HTTP. For our multi-cloud developer data platform, MongoDB Atlas, the MongoDB Atlas Data API is a fully managed REST-like API that enables developers to access their MongoDB Atlas data and perform CRUD operations and aggregations. With the Atlas Data API, you can read and write data in Atlas with standard HTTPS requests.

SQL-like APIs help reduce the learning curve for non-developers already skilled in SQL, such as business analysts and data scientists. The MongoDB Atlas SQL Interface enables users to leverage existing SQL knowledge and familiar tools to query and analyze Atlas data live. The Atlas SQL Interface uses mongosql, a SQL-92-compatible dialect that's designed for the document model. It also leverages Atlas Data Federation functionality for running queries across Atlas clusters and cloud storage, like S3.

Command Line Interface (CLI)

CLIs are text-based interfaces for interacting with a database, application, file, or piece of hardware. CLIs are often the interaction method of choice by advanced developers who prefer control and speed over a more visual interface like a graphical user interface (GUI). The MongoDB Atlas CLI is the fastest way to create and manage an Atlas database, automate ongoing operations, and scale a deployment for the full application development lifecycle. The Atlas CLI gives users a streamlined experience for both onboarding and ongoing management of an Atlas database in the cloud.

Visualization and Reporting

Many companies conduct data visualization, analytics, and reporting using SQL-based BI platforms that do not natively integrate with NoSQL technologies. To address this, organizations turn to an Open Database Connectivity interface, or ODBC driver, to provide industry-standard connectivity between their NoSQL databases and third-party analytics tools. For example, the MongoDB Connector for BI allows analysts, data scientists, and business users to visualize semi-structured and unstructured data managed in MongoDB alongside traditional data from SQL databases using popular BI tools. MongoDB Atlas Charts allows users to create and share visualizations of their MongoDB data in real time without needing to move data into other systems or rely on third-party tools. Because Charts natively understands the MongoDB document model, users can create charts from data that vary in shape or contain nested documents and arrays without needing to first map the data into a flat, tabular structure.

Takeaways

- The maturity and functionality of APIs vary significantly across non-relational products.
- MongoDB's idiomatic drivers minimize onboarding time for new developers and simplify application development.
- Carefully evaluate the SQL-like APIs offered by non-relational databases to ensure they can meet the needs of applications and developers.

Key Consideration #5: Mobile Data

The performance of mobile applications is just as important as the performance of server-based architectures. But mobile apps introduce the added challenge of not always being connected to the network. Application developers need a solution for keeping all of their customers' apps in sync with the backend database, no matter where they are in the world and what kind of network connection they have. The solution also needs to scale easily and quickly as more users download an app, and to support the cutting edge of mobile development technologies as they evolve. NoSQL databases – which are engineered to scale out on demand by leveraging less expensive commodity hardware or cloud infrastructure – are ideally suited to the extra demands placed on the backend by mobile applications that sync to it.

Schema Flexibility

Because new features are always being added in mobile apps, making schema changes in relational databases for new situational relationships becomes increasingly time-consuming. Mobile applications also present more use cases than relational databases are designed to handle, including device type, operating system, firmware, and location. For NoSQL databases, adding features or updating objects to account for new use cases is simply a matter of entering new lines of code. NoSQL databases also are ideal for handling frequent application updates that are a continual part of the app development life cycle. There's no need to overhaul the logic just to fix a bug. And making changes in one part of the database is not likely to affect other parts of the application.

Edge-to-Cloud Synchronization

MongoDB Atlas Device Sync is a fully managed service that syncs mobile data and MongoDB Atlas. This solution addresses the unique technical challenges of mobile and offline-first development, allowing organizations to rapidly build responsive applications for their customers and remote workforces that drive user adoption, improve productivity, and deliver ROI. Device Sync enables teams to take advantage of robust bidirectional data sync between devices and Atlas without having to write complex conflict resolution and networking code.

Takeaways

- The same flexible data model, higher scalability, and superior performance found in NoSQL databases for server environments make NoSQL an ideal solution for mobile applications and data.
- NoSQL databases are engineered to scale out on demand by leveraging less expensive commodity hardware or cloud infrastructure.
- The lack of rigid relational schemas makes NoSQL development more agile and better equipped to add new features, update apps, and fix bugs without having to overhaul the entire database.
- MongoDB Atlas Device Sync is a fully managed service that syncs mobile data and MongoDB Atlas.

Key Consideration #6: Data Platform

Relational databases have a long and successful history of running with proprietary software and hardware as part of an on-premises ecosystem of applications, servers, and endpoints. But modern infrastructure has moved to the cloud. Server workloads are widely distributed across multi-cloud architectures that continually expand the edge of the network far beyond the confines of traditional on-premises environments. Widely distributed workloads place high demands on databases that must fulfill their role as the single source of truth, where truth is measured in microseconds. The simplicity of NoSQL databases makes them better suited for the velocity and volume of modern data transactions. And their portability enables organizations to transform the traditional centralized data repository into a highly flexible and responsive data platform capable of distributing workloads closer to where applications need them.

As data privacy regulations expand to include data sovereignty requirements, and local application servers require the most relevant data to be close by to ensure low-latency reads and writes, organizations need more control over where they deploy their data. MongoDB Atlas cloud database gives organizations this level of control over where they deploy their data, whether for regulatory or for performance purposes.

The Data and Innovation Recurring Tax (DIRT)

Working with data is a critical part of building and evolving applications. Although developers are always finding new ways to build applications, most continue to use the same underlying data infrastructure that's been in use for decades. Legacy relational databases can inhibit innovation due to the rigid nature of tabular structures, which tend to clash with modern data and object data types developers are used to working with. This makes experimenting and iterating on applications harder. Complicating this further is a sprawl of single-purpose data technologies, each designed to solve for a small slice of a growing set of necessary use cases as organizations look to expand their digital footprint. Organizations as a whole are spending more energy trying to figure out how to move data across systems, designing, testing, monitoring, and maintaining an exponential increase in the number of system component interactions. We refer to this as the Data and Innovation Recurring Tax (DIRT), and it can lead to a fragmented developer experience, significant data integration efforts, and unnecessary data duplication.

The Superset of All Data Models

The way to eliminate DIRT is by using a developer data platform that simplifies and accelerates how developers work with data. MongoDB has built a developer data platform that reduces the need for niche databases and the associated costs of deploying and maintaining a complicated sprawl of data technologies. It makes it faster and easier for teams to work with data to support the demands of modern applications while helping to massively simplify an organization's data infrastructure.

In many cases, the relationships between data is more natural to model with documents and subdocuments than in separate tables. Documents map directly to objects in modern object-oriented languages, so the developer experience more closely resembles how they already think and code. This makes it an ideal platform to build upon. The document model is a superset of other data models because it can be used to support graph workloads, key-value, time-series, and geospatial data. So there's no need for additional niche NoSQL databases.

Database as a Service

A modern developer data platform enabled through a database-as-a-service capability gives developers the freedom and flexibility to work seamlessly with data wherever their applications and users need it, and build integrated search features on top of cloud data across all the major public cloud platforms. Rather than rigid tabular schemas and complex relationships, the Atlas developer data platform provides a fully elastic data infrastructure that can be updated as needed via idiomatic drivers that developers are already familiar with. This allows developers more time to focus on their applications rather than managing databases themselves.

Additional Data Workloads

Atlas enables fully integrated full-text search, eliminating the need for a separate search engine. Flexible local datastore offers seamless edge-to-cloud sync for mobile and IoT devices. You can perform in-place, real-time analytics with workload isolation and native data visualization. You can also run federated queries across operational or transactional databases and cloud object storage. And it allows global data distribution for data sovereignty and faster access to data because it resides closer to where it's being used. Atlas also offers industry-leading data-privacy controls with client-side field-level encryption, and it can be deployed globally in over 90 regions.

Takeaways

- Modern multi-cloud environments require flexibility, speed, and elasticity not found in relational databases with tabular schemas.
- Rigid tabular structures lead to the Data and Innovation Recurring Tax (DIRT).
- Distributed databases deployed in the cloud to the edge of the network give organizations the ability to create a resilient, high-availability data platform that puts data closer to the applications that need it.
- Database-as-a-service capabilities allow developers to spend less time managing databases and more time building applications and rich query experiences.

Key Consideration #7: Commercial Support, Community Strength, Freedom From Lock-In

A database is a major investment. Once an application has been built on a given database, it is costly, challenging, and risky to migrate it to a different one. Companies usually invest in a small number of core technologies so they can develop expertise, integrations, and best practices that can be amortized across many projects. NoSQL databases are still a relatively emergent technology. Although there are many new options in the market, only a subset of technologies and vendors will stand the test of time.

Commercial Support

Consider the health of the vendor or product when evaluating a database. It is important not only that the product continues to exist, but also that it evolves and adds new features as the needs of users dictate. Having a strong, experienced support organization capable of providing services globally is another relevant consideration.

Community Strength

There are significant advantages to having a strong community around a technology, particularly databases. A database with a strong community of users makes it easier to find and hire developers who are familiar with the product. It makes it easier to find best practices, documentation, and code samples, all of which reduce risk in new projects. It also helps organizations retain key technical talent. A strong community encourages other technology vendors to develop integrations and participate in the ecosystem.

Freedom From Lock-In

Many organizations have been burned by database lock-in and abusive commercial practices. The use of open-source software and commodity hardware has provided an escape route for many, but organizations also have concerns that as they move to the cloud, they may end up trading one form of lock-in for another.

It's important to evaluate the licensing and availability of any major new software investment. Also critical is having the flexibility to run the database wherever it's needed – whether it's from a developer's laptop in early-stage adoption, on your own infrastructure as you go into production, or in the cloud under a database-as-a-service consumption model.

MongoDB Atlas database enables you to deploy data across AWS, Google Cloud, and Microsoft Azure. In addition, you can create a multi-cloud cluster to enable applications that make use of two or more clouds at the same time. MongoDB Enterprise Advanced gives developers and DevOps teams the option to download and run the database on their own infrastructure. Wherever you choose to run MongoDB, it uses the same codebase, APIs, and management tooling.

Takeaways

- Community size and commercial strength are important for evaluating NoSQL databases.
- MongoDB is one of the very few NoSQL database companies to be publicly traded, it has the largest and most active community, its support teams spread across the world provide 24/7 coverage, it boasts user groups in most major cities, and it provides extensive documentation.
- MongoDB is available to run on your own infrastructure or as a fully managed cloud service on all of the leading public cloud platforms.

Conclusion

As the technology landscape evolves, organizations increasingly find the need to evaluate new databases to support changing application and business requirements. Considering the media hype around NoSQL databases and the commensurate lack of clarity in the market, it's important to make clear distinctions between the available solutions when possible. As discussed in this white paper, there are several key criteria to consider when evaluating these technologies. Many organizations find that document databases such as MongoDB are best suited to meet these criteria.

Resources

- Customer Success Stories
- Presentations
- Free Online Training
- Webinars and Events
- Try MongoDB

For more information, please visit MongoDB.com.