Top 5 Considerations When Evaluating NoSQL Databases

June 2018

Table of Contents

Introduction

Relational databases have a long-standing position in most organizations, and for good reason. Relational databases underpin existing applications that meet current business needs; they are supported by an extensive ecosystem of tools; and there is a large pool of labor qualified to implement and maintain these systems.

But organizations are increasingly considering alternatives to legacy relational infrastructure. In some cases the motivation is technical — such as a need to handle new, multi-structured data types that don't fit the relational database's tabular data model, or scale beyond the capacity constraints of existing systems. In other cases the motivation is driven by the desire to identify viable alternatives to expensive proprietary database software and hardware. A third motivation is agility or speed of development, as companies look to adapt to the market more quickly and embrace agile development methodologies, DevOps, and microservices.

Development teams exert strong influence in the technology selection process. This community tends to find that the relational, tabular data model is not well aligned with the needs of their applications. Consider:

- Developers are working with applications that create new, rapidly changing data types — structured, semi-structured, unstructured and polymorphic data — and massive volumes of it.
- Long gone is the twelve-to-eighteen month waterfall development cycle. Now small teams work in agile sprints, iterating quickly and pushing code every week or two, some even multiple times every day.
- Applications that once served a finite audience are now delivered as services that must be always-on, accessible from many different devices, and scaled globally with data distributed close to it's users for low latency experiences and compliance with data sovereignty regulation.
- Organizations are now turning to scale-out architectures using open source software, running on commodity servers, and cloud computing platforms, instead of large monolithic server and storage infrastructure.

When compared to relational databases, many non-tabular "NoSQL" systems share several key characteristics including a more flexible data model, higher scalability, and

superior performance. But most of these NoSQL databases also discard the very foundation that has made relational databases so useful for generations of applications – expressive query language, secondary indexes, and strong consistency. In fact, the term "NoSQL" is often used as an umbrella category for all non-tabular databases. As we will see, this term is far too wide and loosely defined to be truly useful. It often ignores the trade-offs NoSQL databases have made to achieve flexibility, scalability, and performance.

In this paper, we hope to help you navigate the complex and rapidly evolving domain of NoSQL and non-tabular databases. We describe five critical dimensions organizations should use to evaluate these databases as they determine the right choice for their applications and their businesses.

Data Model

The primary way in which non-tabular databases differ from relational databases is the data model. Although there are dozens of non-tabular databases, they primarily fall into one of the following three categories:

Document Model

Whereas relational databases store data in rows and columns, document databases store data in documents. These documents typically use a structure that is like JSON (JavaScript Object Notation), a format popular among developers. Documents provide an intuitive and natural way to model data that is closely aligned with object-oriented programming – each document is effectively an object. Documents contain one or more fields, where 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 and its associated (i.e., related) data are typically stored together 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, multi-record transactions.

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 unstructured and polymorphic data. It also makes it easier to evolve an application during its lifecycle, such as adding new fields. Additionally, some document databases provide the query expressivity that developers have come to expect from relational databases. In particular, data can be queried based on any combination of fields in a document, with rich secondary indexes providing efficient access paths to support almost any query pattern.

Applications: Document databases are general purpose, 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 data model to objects in modern programming languages.

Examples: MongoDB and CouchDB.

Graph Model

Graph databases use graph structures with nodes, edges and properties to represent data. In essence, data is modeled as a network of relationships between specific elements. While the graph model may be counter-intuitive and takes some time to understand, it can be useful for a specific class of queries. Its main appeal is that it makes it easier to model and navigate relationships between entities in an application.

Applications: Graph databases are useful in cases where traversing relationships are core to the application, like navigating social network connections, network topologies or supply chains.

Examples: Neo4j and AWS Neptune.

Key-Value and Wide Column Models

From a data model perspective, key-value stores 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 only be queried by the key. This model can be useful for representing polymorphic and unstructured data, as the database does not enforce a set schema across key-value pairs.

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

Applications: Key value stores and wide column stores are useful for a narrow set of applications that only query data by 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 and AWS DynamoDB (Key-Value); HBase and Cassandra (Wide Column).

TAKEAWAYS

- All of these data models provide schema flexibility.
- The key-value and wide-column data model is opaque in the system - only the primary key can be queried.
- The document data model has the broadest applicability.
- The document data model is the most natural and most productive because it maps directly to objects in modern object-oriented languages.
- The wide column model provides more granular access to data than the key value model, but less flexibility than the document data model.

Query Model

Each application has its own query requirements. In some cases, it may be acceptable to have a very basic query model in which the application only accesses records based on a primary key. For most applications, however, it is important to have the ability to query based on several different values in each record. For instance, an application that stores data about customers may need to look up not only specific customers, but also specific companies, or customers by a certain size, or aggregations of customer sales value by zip code or state.

It is also common for applications to update records, including one or more individual fields. To satisfy these requirements, the database needs to be able to query data based on secondary indexes. In these cases, a document database will often be the most appropriate solution.

Document Database

Document databases generally provide the ability to query and update any field within a document, though capabilities in this domain vary. Some products, such as MongoDB, provide a rich set of indexing options to optimize a wide variety of queries, including text, geospatial, compound, sparse, time to live (TTL), unique indexes, and others. Furthermore, some of these products provide the ability to analyze data in place, without it needing to be replicated to dedicated analytics or search engines. MongoDB, for instance, provides the Aggregation Framework for developers to create sophisticated processing pipelines for data analytics and transformations, through to faceted search, JOINs, geospatial processing, and graph traversals. It also provides native visualisation capabilities with MongoDB Charts, along with connectors for Apache Spark and BI tools. 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.

Graph Database

These systems tend to provide rich query models where simple and complex relationships can be interrogated to make direct and indirect inferences about the data in the system. Relationship-type analysis tends to be very efficient in these systems, whereas other types of analysis may be less optimal. As a result, graph databases are rarely used for more general purpose operational applications, but instead are coupled with regular document or relational databases to materialize graph-specific data structures and queries.

To try and tame the complexity that would come from using a multitude of storage technologies, the industry is moving towards the concept of “multimodel” databases. Such designs are based on the premise of presenting multiple data models and query types within a single data platform,

thereby serving diverse application requirements. For example, MongoDB offers the $graphLookup aggregation stage for graph processing natively within the database. $graphLookup enables efficient traversals across across graphs, trees, and hierarchical data to uncover patterns and surface previously unidentified connections.

Key Value and Wide Column Databases

These systems provide the ability to retrieve and update data based only on a single or a limited range of keys. For querying on other values, users are encouraged to build and maintain their own indexes. Some products provide limited support for secondary indexes, but with several caveats. To perform an update in these systems, multiple round trips may be necessary: first find the record, then update it, then update the index. In these systems, the update may be implemented as a complete rewrite of the entire record at the client, irrespective of whether a single attribute has changed, or the entire record.

TAKEAWAYS

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

Consistency and Transactional Model

Most non-tabular systems typically maintain multiple copies of the data for availability and scalability purposes. These databases can impose different guarantees on the consistency of the data across copies. Non-tabular 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, writes are not immediately visible, depending on which data replica is serving the query. As an example, when reflecting inventory levels for products in a product catalog, with a consistent system each query will see the current inventory as inventory levels are updated by the application, whereas 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 “eventually” replicated across all nodes in the database cluster. For this reason application code tends to be somewhat 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 non-tabular systems offer atomicity guarantees at the level of an individual record. Because these databases can bring together related data that would otherwise 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.

However, some developers and DBAs have been conditioned by 40 years of relational data modeling to assume multi-record transactions are a requirement for any database, irrespective of the data model they are built upon. Some are concerned that while multi-document 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.

It is for these reasons that MongoDB added support for multi-document ACID transactions. This makes it even easier for developers to address a complete range of use cases with MongoDB. They feel just like the transactions developers are familiar with from relational databases – multi-statement, similar syntax, and easy to add to any application. 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 comes from non-tabular databases.

Consistent Systems

Applications can have different requirements for data consistency. For many applications, it is imperative that the data be consistent at all times. As 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 databases 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 maybe 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 in which all copies of the data are not synchronized. This may be acceptable for read-only applications and data stores that do not change often, like historical archives. By the same token, it may also be appropriate for write-intensive use cases in which the database is capturing information like logs, which will only be read at a later point in time. Key-value and wide column stores are typically 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 ordering of events and to ensure that the most recent operation wins 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 to resolving conflict 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 significantly.

TAKEAWAYS

- Most applications and development teams expect strongly consistent systems.
- Different consistency models pose different trade-offs for applications in the areas of consistency and availability.
- 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 through read repairs and compactions.
- Most non-tabular databases provide single record atomicity. This is sufficient for many applications, but not for all. MongoDB provides multi-document ACID guarantees, making it easier to address a complete range of use-cases with a single data platform.

APIs

There is no standard for interfacing with non-tabular systems. Each system presents different designs and capabilities for application development teams. The maturity of the API can have major implications for the time and cost required to develop and maintain the application and database.

Idiomatic Drivers

There are a number of popular programming languages, and each provides different paradigms for working with data and services. Idiomatic drivers are created by development teams that are experts in the given language and that know how programmers prefer to work within that language. This approach can also benefit from its ability to

leverage specific features in a programming language that might provide efficiencies for accessing and processing data.

For programmers, idiomatic drivers are easier to learn and use, and they reduce the onboarding time for teams to begin working with the underlying 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 over ten languages: Java, .NET, Ruby, Node.js, Perl, Python, PHP, C, C++, C#, Javascript, and Scala. Dozens of other drivers are supported by the community.

RESTful APIs

Some systems provide RESTful interfaces. This approach has the appeal of simplicity and familiarity, but it relies on the inherent latencies associated with HTTP. It also shifts the burden of building an interface to the developers; and this interface is likely to be inconsistent with the rest of their programming interfaces.

SQL-Like APIs

Some non-relational databases have attempted to add a SQL-like access layer to the database, in the hope this will reduce the learning curve for those developers and DBAs already skilled in SQL. It is important to evaluate these implementations before serious development begins, considering the following:

- Most of these implementations fall a long way short compared to the power and expressivity of SQL, and will demand SQL users learn a feature-limited dialect of the language.
- SQL-based BI, reporting, and ETL tools will not be compatible with a custom SQL implementation.
- While some of the syntax may be familiar to SQL developers, data modeling will not be. Trying to impose a relational model on any non-tabular database will have disastrous consequences for performance and application maintenance.

Visualization and Reporting

Many companies conduct data visualization, analytics, and reporting using SQL-based BI platforms that do not natively integrate with non-tabular technologies. To address this, organizations turn to OBDC drivers that provide industry-standard connectivity between their non-tabular databases and 3rd party analytics tools. For example, the MongoDB Connector for BI allows analysts, data scientists, and business users to seamlessly visualize semi-structured and unstructured data managed in MongoDB, alongside traditional data from their SQL databases, using the most popular BI tools. MongoDB Charts allows users to quickly and easily create and share visualisations of their MongoDB data in real time, without needing to move your data into other systems, or leverage third-party tools. Because Charts natively understands the MongoDB document model, users can create charts from data that varies in shape or contains 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.
- Not all SQL is created equal. Carefully evaluate the SQL-like APIs offered by non-relational databases to ensure they can meet the needs of your application and developers

Commercial Support, Community Strength, Freedom from Lock-In

Choosing 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 database. 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. Non-tabular systems are still relatively new, and

while there are many options in the market, a small number of technologies and vendors will stand the test of time.

Commercial Support

Users should consider the health of the company or project when evaluating a database. It is important not only that the product continues to exist, but also to evolve and to provide new features. Having a strong, experienced support organization capable of providing services globally is another relevant consideration.

Community Strength

There are significant advantages of 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 that 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. Lastly, a strong community encourages other technology vendors to develop integrations and to participate in the ecosystem.

Freedom from Lock-In

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

It is important to evaluate the licensing and availability of any major new software investment. Open source licensing is an important start. Also, having the freedom to run the database wherever its needed – from a developer's laptop in early stage adoption, to running it on your own infrastructure as you go into production, and the flexibility to consume the database as a service from any public cloud if desired.

MongoDB is open source, giving developers and DevOps teams the freedom to download and run the database on their own infrastructure. With MongoDB Atlas, it is also available as a fully-managed cloud service, on top of all of the leading cloud providers. Wherever you choose to run MongoDB, you enjoy complete platform portability – you are using the same codebase, APIs, and management tooling.

TAKEAWAYS

- Community size and commercial strength is an important part of evaluating non-relational databases.
- MongoDB is one of the very few non-relational database providers to be a publicly traded company; it has the largest and most active community; support teams spread across the world providing 24x7 coverage; user-groups in most major cities; and 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.

Why MongoDB?

MongoDB is designed to meet the demands of modern apps with a technology foundation that enables you through:

Best Way To Work With Data

- Easy: Work with data in a natural, intuitive way, while providing ACID guarantees to ensure data integrity
- Fast: Get great performance without a lot of work
- Flexible: Adapt and make changes quickly
- Versatile: Supports a wide variety of data and queries

Intelligently Put Data Where You Want It

- Availability: Deliver globally resilient platforms through sophisticated replication and failover
- Scalability: Grow horizontally through native sharding
- Workload Isolation: Run operational and analytical workloads in the same cluster

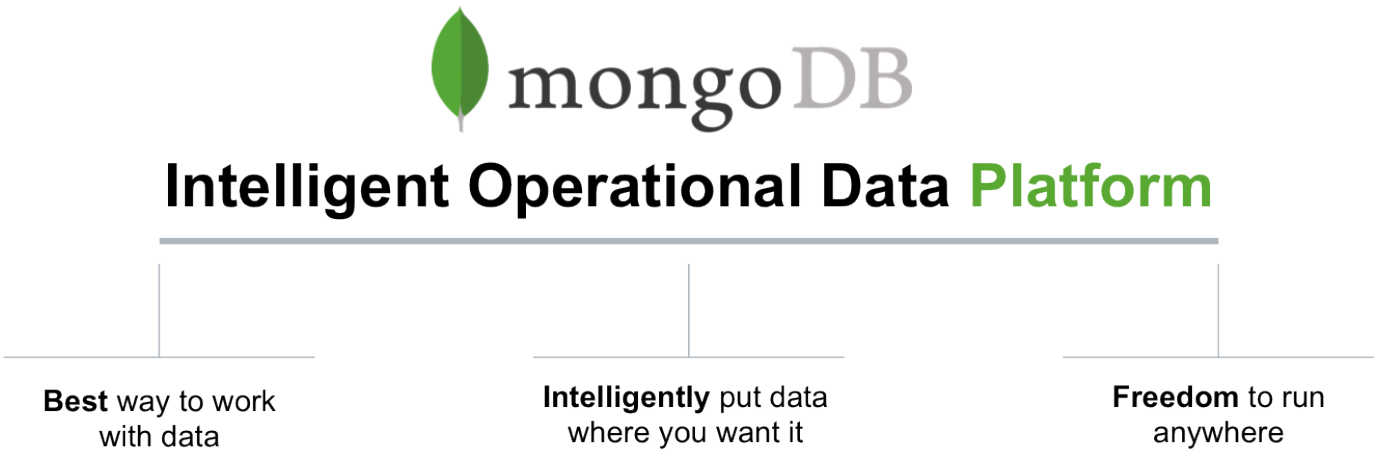

Figure 1: Intelligent Operational Data Platform, Built on MongoDB

- Locality: Place data on specific devices and in specific geographies for governance, class of service, and low-latency access

Freedom To Run Anywhere

- Portability: Database that runs the same everywhere
- Cloud Agnostic: Leverage benefits of multi-cloud strategy with no lock-in
- Global coverage: 50+ regions across the major providers

Learn more from the MongoDB Architecture Guide

Conclusion

As the technology landscape evolves, organizations increasingly find the need to evaluate new databases to support changing application and business requirements. The media hype around non-tabular databases and the commensurate lack of clarity in the market makes it important for organizations to understand the differences between the available solutions. As discussed in this paper, key criteria to consider when evaluating these technologies are the data model, query model, consistency and transactional model, and APIs, as well as commercial support and community strength. Many organizations find that document databases such as MongoDB are best suited to meet these criteria, though we encourage technology decision makers to evaluate these considerations for themselves.

We Can Help

We are the MongoDB experts. Over 5,700 organizations rely on our commercial products. We offer software and services to make your life easier:

MongoDB Enterprise Advanced is the best way to run MongoDB in your data center. It's a finely-tuned package of advanced software, support, certifications, and other services designed for the way you do business.

MongoDB Atlas is a database as a service for MongoDB, letting you focus on apps instead of ops. With MongoDB Atlas, you only pay for what you use with a convenient hourly billing model. With the click of a button, you can scale up and down when you need to, with no downtime, full security, and high performance.

MongoDB Stitch is a serverless platform which accelerates application development with simple, secure access to data and services from the client – getting your apps to market faster while reducing operational costs and effort.

MongoDB Mobile (Beta) MongoDB Mobile lets you store data where you need it, from IoT, iOS, and Android mobile devices to your backend – using a single database and query language.

MongoDB Cloud Manager is a cloud-based tool that helps you manage MongoDB on your own infrastructure. With automated provisioning, fine-grained monitoring, and continuous backups, you get a full management suite that reduces operational overhead, while maintaining full control over your databases.

MongoDB Consulting packages get you to production faster, help you tune performance in production, help you scale, and free you up to focus on your next release.

MongoDB Training helps you become a MongoDB expert, from design to operating mission-critical systems at scale. Whether you're a developer, DBA, or architect, we can make you better at MongoDB.

Resources

For more information, please visit mongodb.com or contact us at sales@mongodb.com.

Case Studies (mongodb.com/customers)
Presentations (mongodb.com/presentations)
Free Online Training (university.mongodb.com)
Webinars and Events (mongodb.com/events)
Documentation (docs.mongodb.com)
MongoDB Enterprise Download (mongodb.com/download)
MongoDB Atlas database as a service for MongoDB (mongodb.com/cloud)
MongoDB Stitch backend as a service (mongodb.com/cloud/stitch)

mongoDB

US 866-237-8815 ▪ INTL +1-650-440-4474 ▪ info@mongodb.com