Top 5 Considerations When Evaluating NoSQL Databases

November 2016

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 or scale beyond the capacity constraints of existing systems — while 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.

These motivations apply both to analytical and operational applications. Companies are shifting workloads to Hadoop for their bulk analytical workloads, and they are building online, operational applications with a new class of so-called "NoSQL" or non-relational databases.

Development teams exert strong influence in the technology selection process. This community tends to find that the relational 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.
- Organizations are now turning to scale-out architectures using open source software, commodity servers and cloud computing instead of large monolithic servers and storage infrastructure.

When compared to relational databases, many 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-relational 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-relational 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-relational databases differ from relational databases is the data model. Although there are dozens of non-relational 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, 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 document. This 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 development, such as adding new fields. Additionally, document databases generally provide the query robustness that developers have come to expect from relational databases. In particular, data can be queried based on any combination of fields in a document.

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 Giraph.

Key-Value and Wide Column Models

From a data model perspective, key-value stores are the most basic type of non-relational 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: Riak and Redis (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 provide the ability to query on any field within a document. Some products, such as MongoDB, provide a rich set of indexing options to optimize a wide variety of queries, including text indexes, geospatial indexes, compound indexes, sparse indexes, time to live (TTL) indexes, 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 both the Aggregation Framework for providing real-time analytics (along the lines of the SQL GROUP BY functionality), and a native MapReduce implementation for other types of sophisticated analyses. To update data, MongoDB provides a find and modify method so that values in documents can be updated in a single statement to the database, rather than making multiple round trips.

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.

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 within the same platform, thereby serving diverse application requirements. For example, MongoDB 3.4 introduces graph computing natively within the database, enabling 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 primary 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 irrespective of whether a single attribute has changed, or the entire record.

TAKEAWAYS

- The biggest difference between non-relational 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 Model

Most non-relational 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-relational databases tend to be categorized as either consistent or eventually consistent.

With a consistent system, writes by the application are immediately visible in subsequent queries. With an eventually consistent system writes are not immediately visible. 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. 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.

Consistent Systems

Each application has 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. Some systems like Riak use vector clocks to determine the ordering of events and to ensure that the most recent operation wins in the case of a conflict. Other systems like CouchDB retain all conflicting values and push the responsibility to resolving conflict back to the user. Another approach, followed by Cassandra, is simply to assume the latest value is the correct one. 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 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.

APIs

There is no standard for interfacing with non-relational 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. 30+ other drivers are supported by the community.

Thrift or 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. Similarly, some systems provide a Thrift interface, a very low level paradigm that shifts the burden to developers to develop more abstract interfaces within their applications.

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.

- Most support queries only, with no support for write operations. Therefore developers will still need to learn the database's native query 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-relational 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 NoSQL technologies. To address this, organizations turn to OBDC drivers that provide industry-standard connectivity between their NoSQL 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.

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 and Community Strength

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-relational systems are relatively new, and while there are many options in the market, a small number of products 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.

TAKEAWAYS

- Community size and commercial strength is an important part of evaluating non-relational databases.
- MongoDB has the largest commercial backing; the largest and most active community; support teams spread across the world providing 24x7 coverage; user-groups in most major cities; and extensive documentation.

The Nexus Architecture

MongoDB's design philosophy is focused on combining the critical capabilities of relational databases with the innovations of NoSQL technologies. Our vision is to leverage the work that Oracle and others have done over the last 40 years to make relational databases what they are today. Rather than discard decades of proven database maturity, MongoDB is picking up where they left off by combining key relational database capabilities with the work that Internet pioneers have done to address the requirements of modern applications.

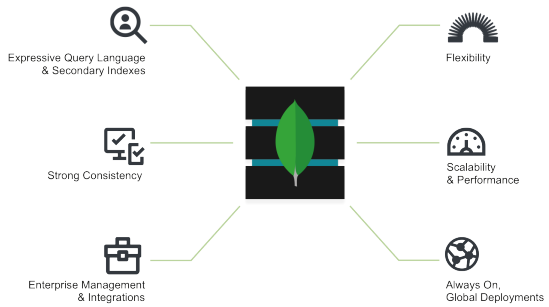

Figure 1: MongoDB Nexus Architecture, blending the best of relational and NoSQL technologies

Relational databases have reliably served applications for many years, and offer features that remain critical today as developers build the next generation of applications:

- **Expressive query language & secondary Indexes**. Users should be able to access and manipulate their data in sophisticated ways to support both operational and analytical applications. Indexes play a critical role in providing efficient access to data, supported natively by the database rather than maintained in application code.
- **Strong consistency.** Applications should be able to immediately read what has been written to the database. It is much more complex to build applications around an eventually consistent model, imposing significant work on the developer, even for the most sophisticated engineering teams.
- **Enterprise Management and Integrations.** Databases are just one piece of application infrastructure, and need to fit seamlessly into the enterprise IT stack. Organizations need a database that can be secured, monitored, automated, and integrated with their existing technology infrastructure, processes, and staff, including operations teams, DBAs, and data engineers.

However, modern applications impose requirements not addressed by relational databases, and this has driven the development of NoSQL databases which offer:

- **Flexible Data Model.** NoSQL databases emerged to address the requirements for the data we see dominating modern applications. Whether document, graph, key-value, or wide-column, all of them offer a flexible data model, making it easy to store and combine data of any structure and allow dynamic modification of the schema without downtime or performance impact.
- **Scalability and Performance.** NoSQL databases were all built with a focus on scalability, so they all include some form of sharding or partitioning. This allows the database to be scaled out across commodity hardware deployed on-premises or in the cloud, enabling almost unlimited growth with higher throughput and lower latency than relational databases.
- **Always-On Global Deployments.** NoSQL databases are designed for continuously available systems that provide a consistent, high quality experience for users all over the world. They are designed to run across many nodes, including replication to automatically synchronize data across servers, racks, and geographically-dispersed data centers.

While offering these innovations, NoSQL systems have sacrificed the critical capabilities that people have come to expect and rely upon from relational databases. MongoDB offers a different approach. With its Nexus Architecture, MongoDB is the only database that harnesses the innovations of NoSQL while maintaining the foundation of relational databases.

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-relational 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 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 2,000 organizations rely on our commercial products, including startups and more than a third of the Fortune 100. 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 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 Professional helps you manage your deployment and keep it running smoothly. It includes support from MongoDB engineers, as well as access to MongoDB Cloud Manager.

Development Support helps you get up and running quickly. It gives you a complete package of software and services for the early stages of your project.

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.

Contact us to learn more, or visit mongodb.com.

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

New York • Palo Alto • Washington, D.C. • London • Dublin • Barcelona • Sydney • Tel Aviv
US 866-237-8815 • INTL +1-650-440-4474 • info@mongodb.com