Data Topics

Analytics | Database | Data Architecture | Data Literacy | Data Science | Data Strategy | Data Modeling | EIM | Governance & Quality | Smart Data

A Brief History of Non-Relational Databases

By **Keith D. Foote** on **June 19, 2018**

First came relational databases, which provide a useful comparison for understanding non-relational databases. Invented by Edgar F. Codd in 1970, the relational database arranges data into different rows and columns by associating a specific key for each row. Almost all relational database systems use Structured Query Language (SQL) and are remarkably complex. They are traditionally more rigid or controlled systems and have a limited or restricted ability to translate complex data such as unstructured data. That said, SQL systems are still used extensively and are quite useful for maintaining accurate transactional records, legacy data

sources, and numerous other use cases within organizations of all sizes.

In the mid-1990s, the internet gained extreme popularity, and relational databases simply could not keep up with the flow of information demanded by users, as well as the larger variety of data types that occurred from this evolution. This led to the development of non-relational databases, often referred to as NoSQL. NoSQL databases can translate strange data quickly and avoid the rigidity of SQL by replacing “organized” storage with more flexibility.

The Evolution of NoSQL

The acronym NoSQL was first used in 1998 by Carlo Strozzi while naming his lightweight, open-source “relational” database that did not use SQL. The name came up again in 2009 when Eric Evans and Johan Oskarsson used it to describe non-relational databases. Relational databases are often referred to as SQL systems. The term NoSQL can mean either “No SQL systems” or the more commonly accepted translation of “Not only SQL,” to emphasize the fact some systems might support SQL-like query languages.

NoSQL developed at least in the beginning as a response to web data, the need for processing unstructured data, and the need for faster processing. The NoSQL model uses a distributed database system, meaning a system with multiple computers. The non-relational system is quicker, uses an ad-hoc approach for organizing data, and processes large amounts of differing kinds of data. For general research, NoSQL databases are the better choice for large, unstructured data sets compared with relational databases due to their speed and flexibility.

Not only can NoSQL systems handle both structured and unstructured data, but they can also process unstructured Big Data quickly. This led to organizations such as Facebook, Twitter, LinkedIn, and Google adopting NoSQL systems. These organizations process tremendous amounts of unstructured data, coordinating it to find patterns and gain business insights. Big Data became an official term in 2005.

The CAP Theorem

The CAP Theorem, also known as Brewer’s theorem (after its developer, Eric Brewer), is an important part of non-relational databases. It states that a distributed data store “cannot” simultaneously offer more than “two of three” established guarantees. Brewer, at the University of California, presented the theory in the fall of 1998, and it was published in 1999 as the *CAP Principle*. The three guarantees that cannot be met simultaneously are:

- **Consistency:** The data within the database remains consistent, even after an operation has been executed. For instance, after updating a system, all clients will see the same data.
- **Availability:** The system is constantly on (always available), with no downtime.
- **Partition Tolerance:** Even if communication among the servers is no longer reliable, the system will continue to function. This is because the servers can be partitioned off, into multiple groups which can't communicate with each other.

In 2002, a formal proof of Brewer's concept was published by Nancy Lynch and Seth Gilbert of MIT, turning it into a "true theorem."

ACID and BASE Provide Consistency

The two most popular consistency models use the acronyms ACID and BASE . Both models have advantages and disadvantages, with neither being a consistent perfect fit. The acronym ACID stands for Atomicity, Consistency, Isolation, and Durability. It was created in 1983 by Theo Härder and Andreas Reuter. The strength of ACID is the guarantee it will provide a safe environment for processing data. This means data is consistent and stable and may use multiple memory locations. Most NoSQL Graph Databases use ACID constraints to ensure data is safely and consistently stored.

The term BASE seems to have become popular in 2008 as an alternative to the ACID model. Availability for scaling purposes is an important feature for BASE data stores. However, it doesn't offer the guarantee of consistency for replicated data during write time. The BASE model , generally speaking, provides less assurance than ACID. BASE is used primarily by aggregate stores , which includes column, document, and key value stores.

Non-Relational Data Storage Design

Non-relational data storage is often open source , non-relational, schema-less, horizontally scalable, and uses BASE for consistency. The term "elasticity" is used for data storage that is scalable, schema-free, and allows for rapid changes and rapid replication. Generally speaking, these features have been accomplished by designing NoSQL data storage from the bottom up and optimized for horizontal scaling. These systems often support only low-level, simplistic APIs (such as "get" and "put" operations). As a consequence, modeling with non-relational systems feels completely different from the modeling used in the relational world and follows a different philosophy.

NoSQL uses data stores optimized for specific purposes. Normally, NoSQL stores data in one of four categories :

- Key-Value storage
- Document storage

- Wide Column storage
- Graph database

A Key-Value Store , also called a Key-Value Database, is a data storage system designed for storage, retrieval, and managing “associative arrays.” A Key-Value Store works very differently than a relational database. A relational database pre-defines the data structure, using a series of tables containing fields with well-defined data types. This data store chooses from a variety of optimal options when classifying the data types.

A Document Store , also called a Document-Oriented Database, is a system designed for storage, retrieval, and managing “document-oriented information,” which is also referred to as semi-structured data. Document Stores have some similarities to Key-Value Stores, but differ in the way the data gets processed. A document-oriented system uses the internal structure of the document for identification and storage. Document Stores save all information for a given item as a single instance in a database (rather than spread out over tables, as with relational systems). This makes it easy to map items into the database.

A Wide Column Store uses tables, rows, and columns, but unlike relational databases, names and formats of the columns can change from row to row within the same table. They are more flexible. Wide Column Stores often support column families, which are stored separately. Each column family normally contains several columns used together. Within a specific column family, data is stored row-by-row, with columns for a specific row being stored together instead of each column being stored individually. Wide Column Stores supporting column families are also called column family databases.

A Graph Database is essentially a collection of relationships. Each memory (a node) symbolizes an entity (a business, person, or object). Each memory/node is connected to another. The connection is called an “edge” and represents a relationship between two nodes. Each node within a Graph Database includes a unique identifier, a set of incoming edges and/or outgoing edges, and characteristics that are represented as “key-value pairs.” Each edge also comes with a unique identifier, an ending and/or a starting place node, and a collection of properties.

Non-Relational Databases vs. Relational Databases

Relational and non-relational databases both have their pros and cons . Relational databases come with the limitation of each item containing only one attribute. Using a sales example, each feature of a client’s relationship is saved as a separate row of items within separate tables. The client’s master details use one table, while account details use another table. These tables are all linked by way of relations, such as foreign and primary keys.

Non-relational databases, on the other hand, are quite different, especially regarding key-value pairs, or Key-Value Stores. Key-value pairs allow several related items to be saved in one “row” within the same table.

It should be noted a non-relational “row” is not the same as a row in a relational table. For example, in a non-relational table, each row would have the client’s details, in addition to their account, sales, and payment history. All the data of one client can be saved together as one convenient record.

While the non-relational database has certain strengths when storing data, it also comes with a significant drawback – key-value stores cannot “enforce” the relationships between items. This means a client’s details (name, address, payment history, etc.) would all be saved as one data record. In a relational model, data would be stored in several tables, providing redundancy and enforcement. This means the relational model comes with a built-in, foolproof way of ensuring business logic and trustworthiness at the database layer. For example, the use of primary and foreign keys will show a payment in the appropriate client account. This is why relational databases continue to be popular.

However, the highest priority for web-based applications is the ability to service large numbers of user requests, which is the strength of non-relational databases. eBay, for example, allows users to browse and view posted items. Only a small number of these users will actually bid on or reserve an item, but millions, sometimes billions, of pages will be viewed per day. eBay is interested in a quick response time and wants to assure fast page loading, rather than enforcing strict business rules.

The Future of Non-Relational Databases

Non-relational databases have their own strengths and weakness, as do relational databases. As the NoSQL revolution continues, it is important to remember “the right tool for the right job” is a useful philosophy. Relational databases support accuracy and redundancy, while non-relational databases support research.

Currently, efforts are being made to merge the two database systems. Hybrid systems are adding SQL-type features like transactional support, joins , and customizable consistency. Additionally, SQL databases (for example SQL Server) are adding NoSQL features which allow more transparent tactics when using horizontal scaling.

It seems reasonable to predict non-relational and relational databases will continue to merge eclectically, adding strengths and minimizing weaknesses.

Photo Credit: Bakhtiar Zein/Shutterstock.com

Overview

To help you better understand NoSQL, this page covers:

- **What is NoSQL and what is a NoSQL database?**
- **Why use NoSQL?**
- **How does NoSQL work?**
- **Conclusion**

NoSQL databases store data in documents rather than relational tables. Accordingly, we classify them as "not only SQL" and subdivide them by a variety of flexible data models. Types of NoSQL databases include pure document databases, key-value stores, wide-column databases, and graph databases. NoSQL databases are built from the ground up to store and process vast amounts of data at scale and support a growing number of modern businesses.

What is NoSQL and what is a NoSQL database?

SQL" at all. This means a NoSQL JSON database can store and retrieve data using literally "no SQL." Or you can combine the flexibility of JSON with the power of SQL for the best of both worlds. Consequently, NoSQL databases are built to be flexible, scalable, and capable of rapidly responding to the data management demands of modern businesses. The following defines the four most-popular types of NoSQL database:

- **Document databases** are primarily built for storing information as documents, including, but not limited to, JSON documents. These systems can also be used for storing XML documents, for example.
- **Key-value stores** group associated data in collections with records that are identified with unique keys for easy retrieval. Key-value stores have just enough structure to mirror the value of relational databases while still preserving the benefits of NoSQL.
- **Wide-column databases** use the tabular format of relational databases yet allow a wide variance in how data is named and formatted in each row, even in the same table. Like key-value stores, wide-column databases have some basic structure while also preserving a lot of flexibility
- **Graph databases** use graph structures to define the relationships between stored data points. Graph databases are useful for identifying patterns in unstructured and semi-structured information.

Why use NoSQL?

Customer experience has quickly become the most important competitive differentiator and ushered the business world into an era of monumental change. As part of this revolution, enterprises are interacting digitally – not only with their customers, but also with their employees, partners, vendors, and even their products – at an unprecedented scale. This interaction is powered by the internet and other 21st century technologies – and at the heart of the revolution are a company's cloud, mobile, social media, big data, and IoT applications.

How are these applications different from legacy enterprise applications like ERP, HR, and financial accounting? Today's web, mobile, and IoT applications share one or more (if not all) of the following characteristics. They need to:

- Support large numbers of concurrent users (tens of thousands, perhaps millions)
- Deliver highly responsive experiences to a globally distributed base of users
- Be always available – no downtime
- Handle semi- and unstructured data

Building and running these massively interactive applications has created a new set of technology requirements. The new enterprise technology architecture needs to be far more agile than ever before, and requires an approach to real-time data management that can accommodate unprecedented levels of scale, speed, and data variability. Relational databases are unable to meet these new requirements, and enterprises are therefore turning to NoSQL database technology.

Global 2000 enterprises are rapidly embracing NoSQL databases to power their mission-critical applications:

- **Tesco**, Europe's No. 1 retailer, deploys NoSQL for e-commerce, product catalog, and other applications
- **Ryanair**, the world's busiest airline, uses NoSQL to power its mobile app serving over 3 million users
- **Marriott** deploys NoSQL for its reservation system that books $38 billion annually
- **Gannett**, the No. 1 U.S. newspaper publisher, uses NoSQL for its proprietary content management system, Presto
- **GE** deploys NoSQL for its Predix platform to help manage the Industrial Internet

Five trends creating new technical challenges that NoSQL DBs address

These companies and hundreds more like them are turning to NoSQL because of five trends that present technical challenges that are too difficult for most relational databases.

Digital Economy trends	1. More customers are going online
Requirements	• Scaling to support thousands, if not millions, of users • Meeting UX requirements with consistent high performance • Maintaining availability 24 hours a day, 7 days a week

What about SQL?

developers and data analysts around the globe to find and report on data stored in relational systems such as Oracle.

Why relational databases fall short

Relational DBMS (database management systems) were born in the era of mainframes and business applications – long before the internet, the cloud, big data, mobile, and today's massively interactive enterprise. These databases were engineered to run on a single server – the bigger, the better. The only way to increase the capacity of these databases was to upgrade the servers – processors, memory, and storage – to scale up.

NoSQL databases emerged as a result of the exponential growth of the internet and the rise of web applications. Google released **the Bigtable research paper** in 2006, and Amazon released **the Dynamo research paper** in 2007. These databases were engineered to meet a new generation of enterprise requirements:

The need to **develop with agility**, to **meet changing requirements**, and to **eliminate data transformation**.

Develop with agility

To remain competitive in today's experience-focused digital economy, enterprises must innovate – and they have to do it faster than ever before. And because this innovation centers on the development of modern web, mobile, and IoT applications, developers have to deliver applications and services faster than ever before. Speed and agility are both critical because these applications evolve far more rapidly than legacy applications like ERP. Relational databases are a major roadblock because they don't support agile development very well due to their fixed data model.

"

Scoping for changing requirements

A core principle of agile development is adapting to evolving application requirements: when the requirements change, the data model also changes. This is a problem for relational databases because the data model is fixed and defined by a static schema. So in order to change the data model, developers have to modify the schema, or worse, request a "schema change" from the database administrators. This slows down or stops development, not only because it's a manual, time-consuming process, but also because it impacts other applications and services.

By contrast, a **NoSQL database** fully supports agile development and does not statically define how the data must be modeled. Instead, NoSQL defers to the applications and services, and thus to the developers as to how data should be modeled. With NoSQL, the data model is defined by the application model. Applications and services model data as objects.

Eliminate data transformation

Applications and services model data as objects (e.g., employee), multi-valued data as collections (e.g., roles), and related data as nested objects or collections (e.g., manager). However, relational databases model data as tables of rows and columns – related data as rows within different tables, and multi-valued data as rows within the same table.

The problem with relational databases is that data is read and written by disassembling, or "shredding," and reassembling objects. This is the object-relational "impedance mismatch." The workaround is transforming data via object-relational mapping frameworks, which are inefficient at best, and problematic at worst. **NoSQL databases generally handle data as it is presented**.

How does NoSQL work?

How does NoSQL compare? Let's take a closer look. The following NoSQL tutorial illustrates an application used for managing resumes. It interacts with resumes as an object (i.e., the user

Storing this resume would require the application to insert six rows into three tables, as illustrated in **Figure 3**.

And, reading this profile would require the application to read six rows from three tables, as illustrated in **Figure 4**.

By contrast, in a document-oriented database defined as NoSQL, JSON is the de facto format for storing data – helpfully, it's also the de facto standard for consuming and producing data for web, mobile, and IoT applications. JSON not only eliminates the object-relational impedance mismatch, it also eliminates the overhead of ORM frameworks and simplifies application development because objects are read and written without "shredding" them (i.e., a single object can be read or written as a single document), as illustrated in **Figure 5**.

What about querying and SQL?

to handle structured and unstructured data equally well, and new tools allow for faster querying than ever before.

Couchbase Server 4.0 introduced N1QL (pronounced "nickel"), a powerful query language that extends SQL to JSON, enabling developers to leverage both the power of SQL and the flexibility of JSON. It not only supports standard SELECT / FROM / WHERE statements, it also supports aggregation (GROUP BY), sorting (SORT BY), joins (LEFT OUTER / INNER), as well as querying nested arrays and collections. In addition, query performance can be improved with composite, partial, covering indexes, and more.

SQL

SQL

```
SELECT RTRIM(p.FirstName) + ' ' + LTRIM(p.LastName) AS Name, d.C
FROM AdventureWorks2016.Person.Person AS p
INNER JOIN AdventureWorks2016.HumanResources.Employee e ON p.Bus
INNER JOIN
    (SELECT bea.BusinessEntityID, a.City
    FROM AdventureWorks2016.Person.Address AS a
    INNER JOIN AdventureWorks2016.Person.BusinessEntityAddress
    ON a.AddressID = bea.AddressID) AS d
ON p.BusinessEntityID = d.BusinessEntityID
ORDER BY p.LastName, p.FirstName;
```

Learn about querying

N1QL tutorial > **What is N1QL? >** **Run your first query >** **Try free >**

Operate at any scale

Databases that support web, mobile, and IoT applications must be able to operate at any scale. While it's possible to scale a relational database like Oracle (using, for example, Oracle RAC), doing so is typically complex, expensive, and not fully reliable. With Oracle, for example, scaling out using RAC technology requires numerous components and creates a single point of failure

Elasticity for performance at scale

Applications and services have to support an ever-increasing number of users and data – hundreds to thousands to millions of users, and gigabytes to terabytes of operational data. At the same time, they have to scale to maintain performance, and they have to do it efficiently.

The database has to be able to scale reads, writes, and storage.

This is a problem for relational databases that are limited to scaling up (i.e., adding more processors, memory, and storage to a single physical server). As a result, the ability to scale efficiently, and on demand, is a challenge. It becomes increasingly expensive because enterprises have to purchase bigger and bigger servers to accommodate more users and more data. In addition, it can result in downtime if the database has to be taken offline to perform hardware upgrades.

A distributed NoSQL database, however, leverages commodity hardware to scale out – i.e., add more resources simply by adding more servers. The ability to scale out enables enterprises to scale more efficiently by (a) deploying no more hardware than is required to meet the current load; (b) leveraging less expensive hardware and/or cloud infrastructure; and (c) scaling on demand and without downtime.

In addition to being able to scale effective and efficiently, distributed NoSQL databases are easy to install, configure, and scale. They were engineered to distribute reads, writes, and storage, and they were engineered to operate at any scale – including the management and monitoring of clusters small and large.

Availability for always-on, global deployment

As more and more customer engagements take place online via web and mobile apps, availability becomes a major, if not primary, concern. These mission-critical applications have to be available 24 hours a day, 7 days a week – no exceptions. Delivering 24x7 availability is a challenge for relational databases that are deployed to a single physical server or that rely on clustering with shared storage. If deployed as a single server and it fails, or as a cluster and the shared storage fails, the database becomes unavailable.

like Oracle require separate software for replication (e.g., Oracle Active Data Guard), NoSQL databases do not – it's built in and it's automatic. In addition, automatic failover ensures that if a node fails, the database can continue to perform reads and writes by sending the requests to a different node.

As customer engagements move online, the need to be available in multiple countries and/or regions becomes critical. While deploying a database to multiple datacenters increases availability and helps with disaster recovery, it also has the benefit of increasing performance, because all reads and writes can be executed on the nearest datacenter, thereby reducing latency.

Ensuring global availability is difficult for relational databases in cases where the requirement of separate add-ons increases complexity (e.g., Oracle requires Oracle GoldenGate to move data between databases) – or when replication between multiple datacenters can only be used for failover because only one datacenter is active at a time. Also, when replicating between datacenters, applications built on relational databases can experience performance degradation or find that the datacenters are severely out of sync.

A distributed, NoSQL database includes built-in replication between datacenters – no separate software is required. In addition, some include both unidirectional and bidirectional replication enabling full active-active deployments to multiple datacenters, which allow the database to be deployed in multiple countries and/or regions and to provide local data access to local applications and their users. This not only improves performance, it also enables immediate failover via hardware routers – applications don't have to wait for the database to discover the failure and perform its own failover.

Conclusion

So what are NoSQL databases and why do they matter now? As enterprises shift to the Digital Economy – enabled by cloud, mobile, social media, and big data technologies – developers and operations teams have to build and maintain web, mobile, and IoT applications faster and faster, and at greater scale. Flexible, high-performance NoSQL is increasingly the preferred database technology to power today's web, mobile, and IoT applications.

Hundreds of Global 2000 enterprises, along with tens of thousands smaller businesses and startups, have adopted NoSQL. For many, the use of NoSQL started with a cache, proof of concept, or a small application, then expanded to targeted mission-critical applications, and is now the foundation for all application development.

With NoSQL, enterprises are better able to both develop with agility and operate at any scale – and deliver the performance and availability required to meet the demands of Digital Economy businesses.

Related resources

Whitepapers
Moving from Relational to NoSQL: How to Get Started

Whitepapers
NoSQL Database Evaluation Guide: Eight Core Requirements

See All Resources

Couchbase

English (US)

COMPANY

ABOUT

LEADERSHIP

NEWS & PRESS

INVESTOR RELATIONS

CAREERS

EVENTS

LEGAL

CONTACT US

QUICKLINKS

BLOG

DOWNLOADS

GET STARTED

ONLINE TRAINING

RESOURCES

WHY NOSQL

PRICING

SUPPORT

DEVELOPER PORTAL

DOCUMENTATION

FORUMS

PROFESSIONAL SERVICES

SUPPORT LOGIN

SUPPORT POLICY

TRAINING

KEEP IN TOUCH

TWITTER

LINKEDIN

YOUTUBE

FACEBOOK

GITHUB

STACKOVERFLOW

/thoughtworks Menu

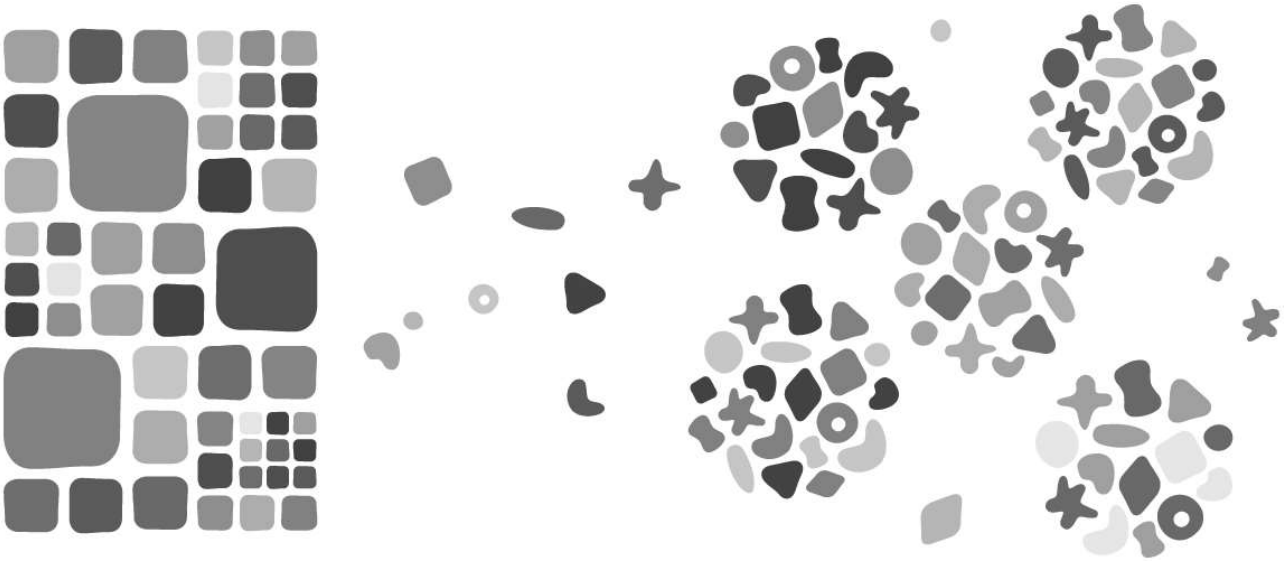
What we do >
Go to overview >
Customer Experience, Product and Design >
Data and AI >
Digital Transformation and Operations >
Enterprise Modernization, Platforms and Cloud >
Who we work with >
Go to overview >
Automotive >
Healthcare >
Public Sector >
Cleantech, Energy and Utilities >
Media and Publishing >
Retail and E-commerce >
Financial Services and Insurance >
Not-for-profit >
Travel and Transport >
Insights >
Go to overview >
Featured
Technology
An in-depth exploration of enterprise technology and engineering excellence
>
Business
Keep up to date with the latest business and industry insights for digital leaders
>
Culture
The place for career-building content and tips, and our view on social justice and inclusivity
>
Digital Publications and Tools
Technology Radar
An opinionated guide to technology frontiers
>
Perspectives
A publication for digital leaders
>
Digital Fluency Model
A model for prioritizing the digital capabilities needed to navigate uncertainty
>
Decoder
The business execs' A-Z guide to technology
>
Looking Glass
Bringing the tech-led business changes into focus
>
All Insights
Articles
Expert insights to help your business grow
>
Blogs

NoSQL Databases: An Overview

Data science and analytics Blog

By **Pramod Sadalage**

Published: October 02, 2014



Over the last few years we have seen the rise of a new type of databases, known as NoSQL databases, that are challenging the dominance of relational databases. Relational databases have dominated the software industry for a long time providing mechanisms to store data persistently, concurrency control, transactions, mostly standard interfaces and mechanisms to integrate application data, reporting. The dominance of relational databases, however, is cracking.

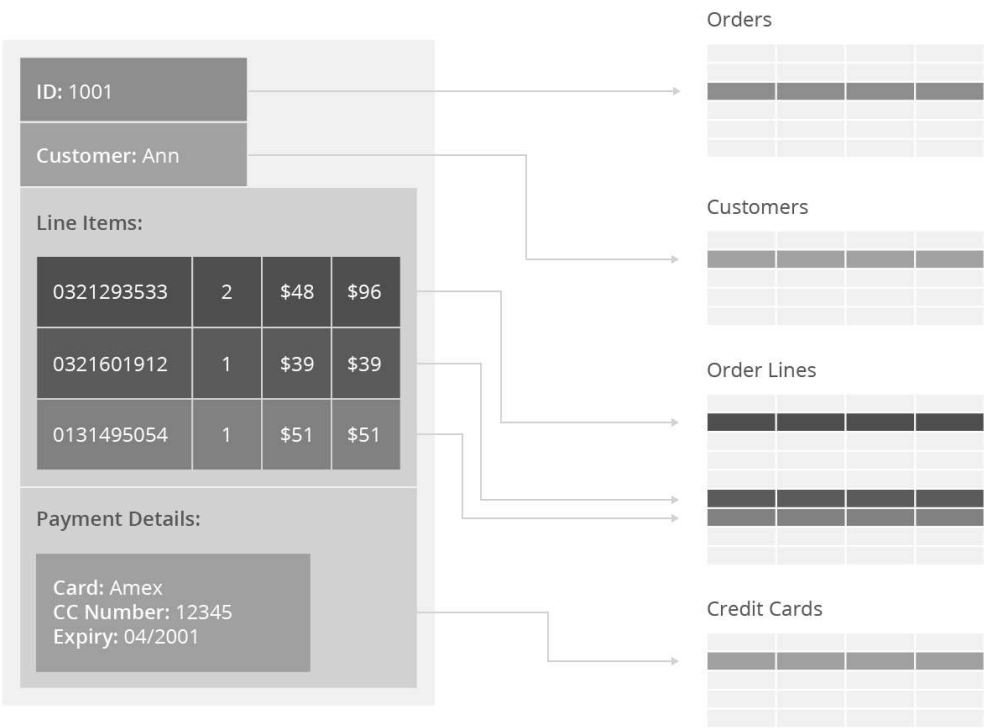
NoSQL what does it mean

What does NoSQL mean and how do you categorize these databases? NoSQL means Not Only SQL, implying that when designing a software solution or product, there are more than one storage mechanism that could be used based on the needs. NoSQL was a hashtag (#nosql) choosen for a meetup to discuss these new databases. The most important result of the rise of NoSQL is Polyglot Persistence. NoSQL does not have a prescriptive definition but we can make a set of common observations, such as:

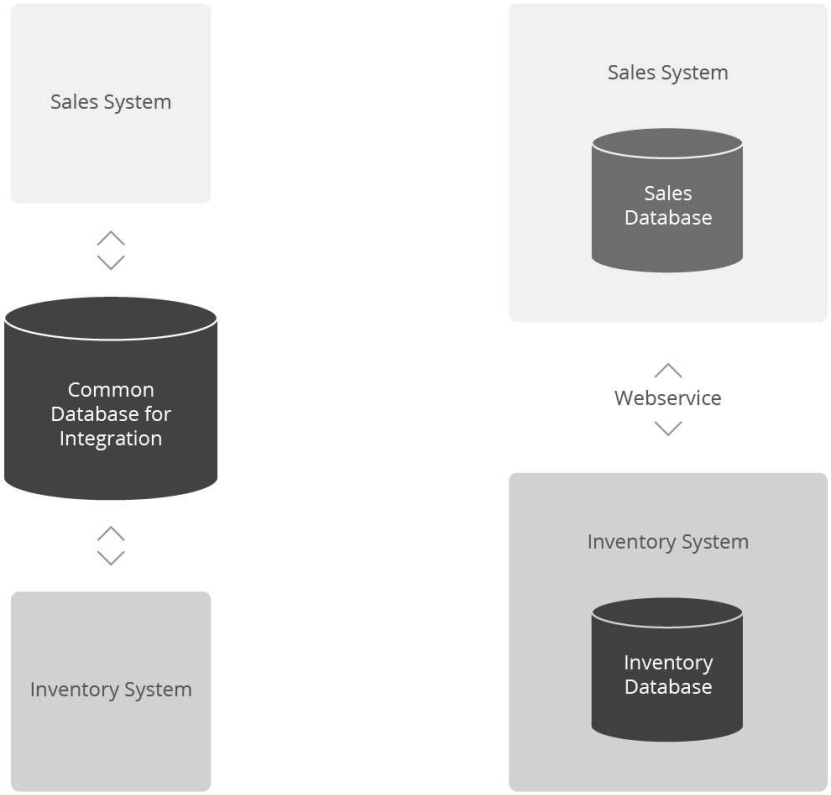
- Not using the relational model
- Running well on clusters
- Mostly open-source

- Built for the 21st century web estates
- Schema-less

Why NoSQL Databases



Application developers have been frustrated with the impedance mismatch between the relational data structures and the in-memory data structures of the application. Using NoSQL databases allows developers to develop without having to convert in-memory structures to relational structures.



There is also movement away from using databases as integration points in favor of encapsulating databases with applications and integrating using services.

The rise of the web as a platform also created a vital factor change in data storage as the need to support large volumes of data by running on clusters.

Relational databases were not designed to run efficiently on clusters.

The data storage needs of an ERP application are lot more different than the data storage needs of a Facebook or an Etsy, for example.

Aggregate Data Models:

Relational database modelling is vastly different than the types of data structures that application developers use. Using the data structures as modelled by the developers to solve different problem domains has given rise to movement away from relational modelling and towards aggregate models, most of this is driven by ***Domain Driven Design***, a book by Eric Evans. An aggregate is a collection of data that we interact with as a unit. These units of data or aggregates form the boundaries for ACID operations with the database, Key-value, Document, and Column-family databases can all be seen as forms of aggregate-oriented database.

Aggregates make it easier for the database to manage data storage over clusters, since the unit of data now could reside on any machine and when retrieved from the database gets all the related data along with it. Aggregate-oriented databases work best when most data interaction is done with the same aggregate, for example when there is need to get an order and all its details, it better to store order as an aggregate object but dealing with these aggregates to get item details on all the orders is not elegant.

Aggregate-oriented databases make inter-aggregate relationships more difficult to handle than intra-aggregate relationships. Aggregate-ignorant databases are better when interactions use data organized in many different formations. Aggregate-oriented databases often compute materialized views to provide data organized differently from their primary aggregates. This is often done with map-reduce computations, such as a map-reduce job to get items sold per day.

Distribution Models:

Aggregate oriented databases make distribution of data easier, since the distribution mechanism has to move the aggregate and not have to worry about related data, as all the related data is contained in the aggregate. There are two styles of distributing data:

- Sharding: Sharding distributes different data across multiple servers, so each server acts as the single source for a subset of data.
- Replication: Replication copies data across multiple servers, so each bit of data can be found in multiple places. Replication comes in two forms,
    - Master-slave replication makes one node the authoritative copy that handles writes while slaves synchronize with the master and may handle reads.
    - Peer-to-peer replication allows writes to any node; the nodes coordinate to synchronize their copies of the data.

Master-slave replication reduces the chance of update conflicts but peer-to-peer replication avoids loading all writes onto a single server creating a single point of failure. A system may use either or both techniques. Like Riak database shards the data and also replicates it based on the replication factor.

CAP theorem:

In a distributed system, managing consistency(C), availability(A) and partition toleration(P) is important, Eric Brewer put forth the CAP theorem which states that in any distributed system we can choose only two of consistency, availability or partition tolerance. Many NoSQL databases try to provide options where the developer has choices where they can tune the database as per their needs. For example if you consider **Riak** a distributed key-value database. There are essentially three variables r, w, n where

- r=number of nodes that should respond to a read request before its considered successful.
- w=number of nodes that should respond to a write request before its considered successful.
- n=number of nodes where the data is replicated aka replication factor.

In a Riak cluster with 5 nodes, we can tweak the r,w,n values to make the system very consistent by setting r=5 and w=5 but now we have made the cluster susceptible to network partitions since any write will not be considered successful when any node is not responding. We can make the same cluster highly available for writes or reads by setting r=1 and w=1 but now consistency can be compromised since some nodes may not have the latest copy of the data. The CAP theorem states that if you get a network partition, you have to trade off availability of data versus consistency of data. Durability can also be traded off against latency, particularly if you want to survive failures with replicated data.

NoSQL databases provide developers lot of options to choose from and fine tune the system to their specific requirements. Understanding the requirements of how the data

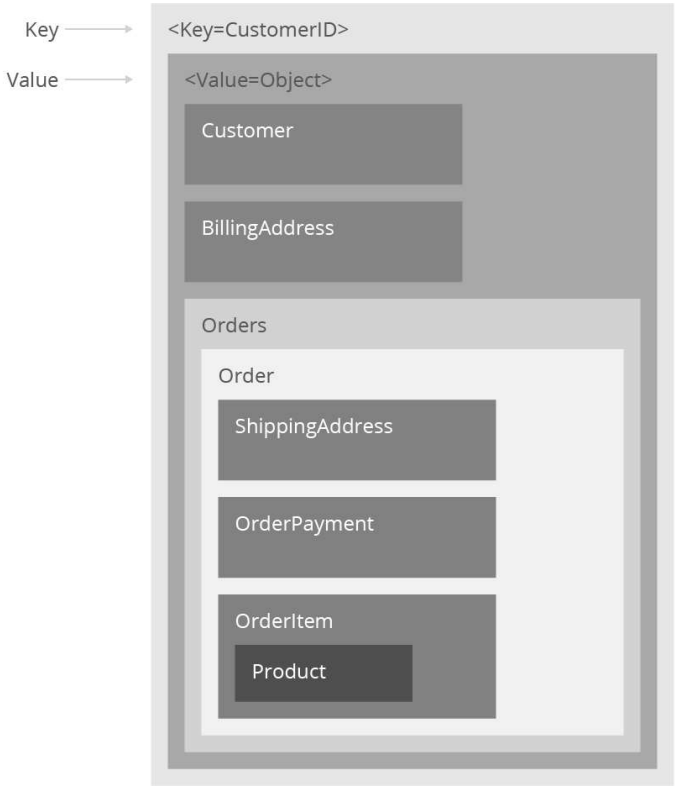
is going to be consumed by the system, questions such as is it read heavy vs write heavy, is there a need to query data with random query parameters, will the system be able handle inconsistent data.

Understanding these requirements becomes much more important, for long we have been used to the default of RDBMS which comes with a standard set of features no matter which product is chosen and there is no possibility of choosing some features over other. The availability of choice in NoSQL databases, is both good and bad at the same time. Good because now we have choice to design the system according to the requirements. Bad because now you have a choice and we have to make a good choice based on requirements and there is a chance where the same database product may be used properly or not used properly.

An example of feature provided by default in RDBMS is transactions, our development methods are so used to this feature that we have stopped thinking about what would happen when the database does not provide transactions. Most NoSQL databases do not provide transaction support by default, which means the developers have to think how to implement transactions, does every write have to have the safety of transactions or can the write be segregated into “critical that they succeed” and “its okay if I lose this write” categories. Sometimes deploying external transaction managers like **ZooKeeper** can also be a possibility.

Types of NoSQL Databases:

NoSQL databases can broadly be categorized in four types.



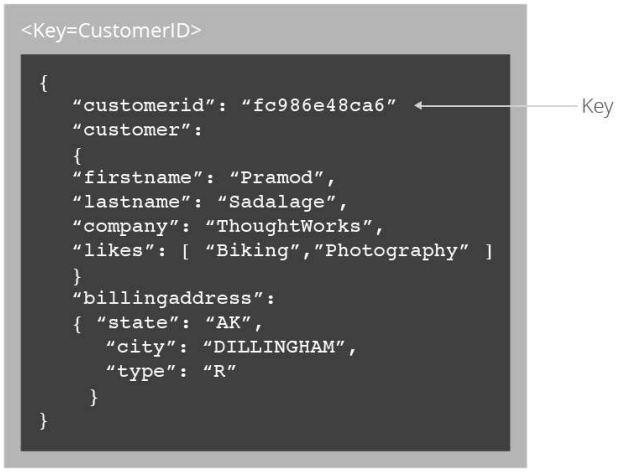
Key-Value databases

Key-value stores are the simplest NoSQL data stores to use from an API perspective. The client can either get the value for the key, put a value for a key, or delete a key from the data store. The value is a blob that the data store just stores, without caring or knowing what's inside; it's the responsibility of the application to understand what was stored. Since key-value stores always use primary-key access, they generally have great performance and can be easily scaled.

Some of the popular key-value databases are **Riak**, **Redis** (often referred to as Data Structure server), **Memcached** and its flavors, **Berkeley DB**, upscaledb (especially suited for embedded use), Amazon DynamoDB (not open-source), Project Voldemort and **Couchbase**.

All key-value databases are not the same, there are major differences between these products, for example: Memcached data is not persistent while in Riak it is, these features are important when implementing certain solutions. Lets consider we need to implement caching of user preferences, implementing them in memcached means when the node goes down all the data is lost and needs to be refreshed from source system, if we store the same data in Riak we may not need to worry about losing data but we must also consider how to update stale data. Its important to not only choose a key-value database based on your requirements, it's also important to choose which key-value database.

Document databases



Documents are the main concept in document databases. The database stores and retrieves documents, which can be XML, JSON, BSON, and so on. These documents are self-describing, hierarchical tree data structures which can consist of maps, collections, and scalar values. The documents stored are similar to each other but do not have to be exactly the same. Document databases store documents in the value part of the key-value store; think about document databases as key-value stores where the value is examinable. Document databases such as MongoDB provide a rich query language and

constructs such as database, indexes etc allowing for easier transition from relational databases.

Some of the popular document databases we have seen are **MongoDB**, **CouchDB** , **Terrastore**, **OrientDB**, **RavenDB**, and of course the well-known and often reviled Lotus Notes that uses document storage.

Column family stores

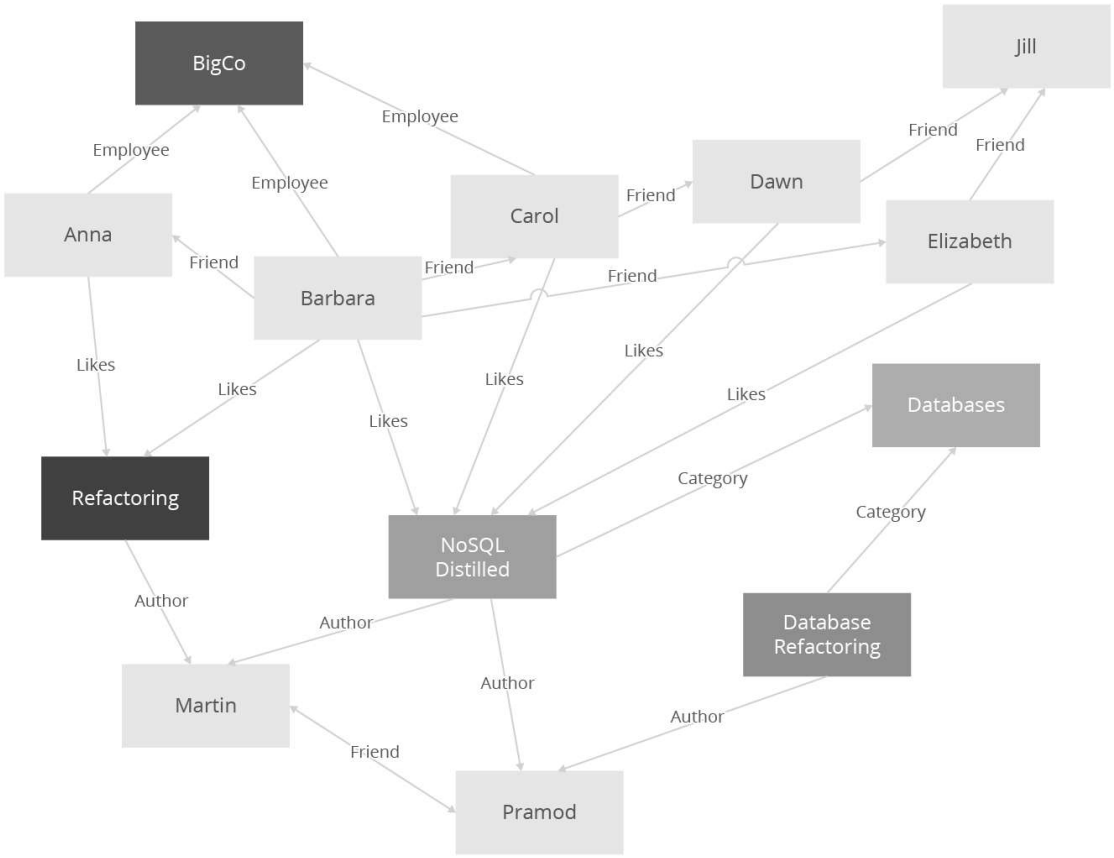
Column-family databases store data in column families as rows that have many columns associated with a row key (Figure 10.1). Column families are groups of related data that is often accessed together. For a Customer, we would often access their Profile information at the same time, but not their Orders.

Each column family can be compared to a container of rows in an RDBMS table where the key identifies the row and the row consists of multiple columns. The difference is that various rows do not have to have the same columns, and columns can be added to any row at any time without having to add it to other rows.

When a column consists of a map of columns, then we have a super column. A super column consists of a name and a value which is a map of columns. Think of a super column as a container of columns.

Cassandra is one of the popular column-family databases; there are others, such as **HBase**, **Hypertable**, and Amazon DynamoDB. Cassandra can be described as fast and easily scalable with write operations spread across the cluster. The cluster does not have a master node, so any read and write can be handled by any node in the cluster.

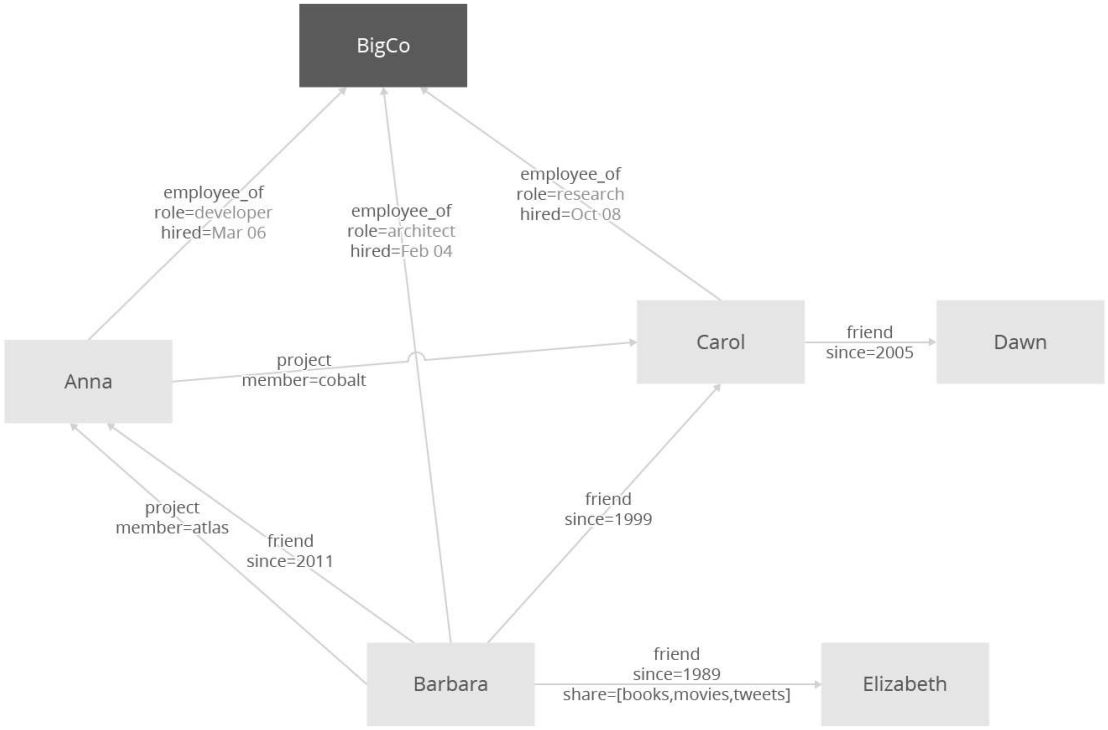
Graph Databases



Graph databases allow you to store entities and relationships between these entities. Entities are also known as nodes, which have properties. Think of a node as an instance of an object in the application. Relations are known as edges that can have properties. Edges have directional significance; nodes are organized by relationships which allow you to find interesting patterns between the nodes. The organization of the graph lets the data to be stored once and then interpreted in different ways based on relationships.

Usually, when we store a graph-like structure in RDBMS, it's for a single type of relationship ("who is my manager" is a common example). Adding another relationship to the mix usually means a lot of schema changes and data movement, which is not the case when we are using graph databases. Similarly, in relational databases we model the graph beforehand based on the Traversal we want; if the Traversal changes, the data will have to change.

In graph databases, traversing the joins or relationships is very fast. The relationship between nodes is not calculated at query time but is actually persisted as a relationship. Traversing persisted relationships is faster than calculating them for every query.



Nodes can have different types of relationships between them, allowing you to both represent relationships between the domain entities and to have secondary relationships for things like category, path, time-trees, quad-trees for spatial indexing, or linked lists for sorted access. Since there is no limit to the number and kind of relationships a node can have, they all can be represented in the same graph database.

Relationships are first-class citizens in graph databases; most of the value of graph databases is derived from the relationships. Relationships don't only have a type, a start node, and an end node, but can have properties of their own. Using these properties on the relationships, we can add intelligence to the relationship—for example, since when did they become friends, what is the distance between the nodes, or what aspects are shared between the nodes. These properties on the relationships can be used to query the graph.

Since most of the power from the graph databases comes from the relationships and their properties, a lot of thought and design work is needed to model the relationships in the domain that we are trying to work with. Adding new relationship types is easy; changing existing nodes and their relationships is similar to data migration, because these changes will have to be done on each node and each relationship in the existing data.

There are many graph databases available, such as **Neo4J**, **Infinite Graph**, **OrientDB**, or **FlockDB** (which is a special case: a graph database that only supports single-depth relationships or adjacency lists, where you cannot traverse more than one level deep for relationships).

Why choose NoSQL database

We've covered a lot of the general issues you need to be aware of to make decisions in the new world of NoSQL databases. It's now time to talk about why you would choose NoSQL databases for future development work. Here are some broad reasons to consider the use of NoSQL databases.

- To improve programmer productivity by using a database that better matches an application's needs.
- To improve data access performance via some combination of handling larger data volumes, reducing latency, and improving throughput.

It's essential to test your expectations about programmer productivity and/or performance before committing to using a NoSQL technology. Since most of the NoSQL databases are open source, testing them is a simple matter of downloading these products and setting up a test environment.

Even if NoSQL cannot be used as of now, designing the system using service encapsulation supports changing data storage technologies as needs and technology evolve. Separating parts of applications into services also allows you to introduce NoSQL into an existing application.

Choosing NoSQL database

Given so much choice, how do we choose which NoSQL database? As described much depends on the system requirements, here are some general guidelines:

- Key-value databases are generally useful for storing session information, user profiles, preferences, shopping cart data. We would avoid using Key-value databases when we need to query by data, have relationships between the data being stored or we need to operate on multiple keys at the same time.
- Document databases are generally useful for content management systems, blogging platforms, web analytics, real-time analytics, ecommerce-applications. We would avoid using document databases for systems that need complex transactions spanning multiple operations or queries against varying aggregate structures.
- Column family databases are generally useful for content management systems, blogging platforms, maintaining counters, expiring usage, heavy write volume such as log aggregation. We would avoid using column family databases for systems that are in early development, changing query patterns.
- Graph databases are very well suited to problem spaces where we have connected data, such as social networks, spatial data, routing information for goods and money, recommendation engines

Schema-less ramifications

All NoSQL databases claim to be schema-less, which means there is no schema enforced by the database themselves. Databases with strong schemas, such as relational databases, can be migrated by saving each schema change, plus its data migration, in a version-controlled sequence. Schema-less databases still need careful migration due to the implicit schema in any code that accesses the data.

Schema-less databases can use the same migration techniques as databases with strong schemas, in schema-less databases we can also read data in a way that's tolerant to changes in the data's implicit schema and use incremental migration to update data, thus allowing for zero downtime deployments, making them more popular with 24*7 systems.

Conclusion

All the choice provided by the rise of NoSQL databases does not mean the demise of RDBMS databases. We are entering an era of polyglot persistence, a technique that uses different data storage technologies to handle varying data storage needs. Polyglot persistence can apply across an enterprise or within a single application.

For more details, read ***NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*** by Pramod Sadalage and Martin Fowler.

Disclaimer: The statements and opinions expressed in this article are those of the author(s) and do not necessarily reflect the positions of Thoughtworks.

Related Blogs

ML and AI

Put Data Science Before Data Infrastructure

Learn more >

Data science and analytics

Introducing Agile Analytics

Learn more >

Technology strategy

Macro Trends in the Technology Industry

Learn more >

Keep up to date with our latest insights

Explore our content

Company

Insights

Site info

Connect with us