Couchbase

WHITEPAPER

Why NoSQL Databases?

A Developer's Guide

Why successful projects rely on NoSQL database applications

Contents

WHAT IS NOSQL?

RELATIONAL VS. NOSQL: WHAT'S THE DIFFERENCE?

Relational databases were born in the era of mainframes and back-office business applications—long before the internet, the cloud, big data, mobile, etc.

NoSQL is an umbrella term for database management systems that store information in a variety of formats and structures that cater to specific data access and processing requirements that traditional relational databases have trouble addressing. Relational structures were necessary to minimize duplication during an era when storage solutions were expensive. And for decades, database and application functionality flourished until application functionality and large-scale data volumes demanded alternative methods of data access. Google-style search, for example, requires special indexing and scanning capabilities that relational databases had trouble supporting.

Requirements like this are what spawned the creation of NoSQL databases. NoSQL has also been called "Not only SQL," and encompasses a variety of models including key/value, document, column, time series, and graph. As NoSQL capabilities evolved, they began to not only incorporate traditional relational database capabilities, they have also begun to converge their data access models, creating today's modern, multi-model NoSQL databases.

What is a modern multi-model NoSQL database?

Modern NoSQL databases incorporate multiple data access methods, making them useful across a wide variety of use cases. As they have evolved, JSON (JavaScript Object Notation) has become a common data format from which these databases are designed. Today's NoSQL databases can deliver data for applications in ways that make development easier and more robust.

NoSQL databases are built from the ground up to be fast, flexible, and support modern cloud computing, distribution, and data management needs for modern applications.

This paper introduces the modern challenges that NoSQL databases address and shows when to choose NoSQL over relational and why.

Customer experience drives enterprises to NoSQL solutions

Customer experience is the most important competitive differentiator and businesses are working to meet these new expectations with services that are on demand, real time, and responsive. Customer experiences are dynamic, operating across multiple devices and interfaces, changing and evolving constantly. Keeping up with this pace of change is the current challenge for all databases, including NoSQL.

Meeting the demands of these new systems requires flexibility, performance, and scalability, and benefits from the ability to consolidate multiple types of systems to reduce database sprawl.

Relational vs. NoSQL: What's the difference?

Relational databases were born in the era of mainframes and back-office business applications—long before the internet, the cloud, big data, mobile, etc. In fact, the first commercial implementation was released by Oracle in 1979. 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 server—processors, memory, and storage—to scale up as Moore's Law allowed (also known as "vertical" scaling).

NoSQL databases emerged as a result of the exponential growth of the internet and the rise of web applications. Google Bigtable research was released in 2006, as well as an Amazon Dynamo research paper in 2007. Efficient, distributed, highly scalable key-value (KV) engines were essential to this evolutionary step and have propelled the technology much further.

New databases were engineered to meet the next generation of enterprise requirements, which companies like Couchbase have taken even further to meet needs going into the future—the need to **develop with agility**.

Agility means providing flexible schemas, APIs, robust SQL-based querying, text search, analytics, and more.

Supporting SQL and NoSQL developers

Traditional relational systems manage tabular data and return it as rows and columns. NoSQL databases can do this without forcing application developers into using a static schema that has to be altered every time there is a change. Instead, NoSQL databases give developers the flexibility they need to help them excel at their work.

NoSQL systems hold hierarchical JSON data, but return it to the application as full or partial JSON data structures, full-text search matches, SQL query results, key-based values, or even big data analytics systems.

This convergence of the best of relational and the best of modern NoSQL simplifies the information architecture of enterprises and helps developers deliver applications more efficiently with familiar concepts and tooling, without needing to learn a dozen different platforms.

Beyond relational database management systems (RDBMS)

To meet modern development styles, NoSQL databases provide the option to store data in flexible formats while also delivering high speeds and keeping data synchronized as systems scale.

For example, teams are now expected to build data management layers that include the following characteristics:

- Deliver highly responsive experiences, through the web and mobile
- Handle semi- and unstructured data
- Adapt rapidly to changing user requirements with frequent updates
- Deliver new features with shorter times to market
- Support multiple data type and data access methods
- Be always available—no downtime

It's very difficult for SQL-based relational databases to meet these requirements efficiently and cost-effectively, especially when it comes to uptime, scalability, and responsiveness.

FIVE ADVANTAGES OF NOSQL DATABASES

NOSQL ALLOWS YOU TO DEVELOP WITH AGILITY

- Adaptable NoSQL schema requirements
- Flexibility for faster development
- Simplicity for easier development

Here are five areas that demonstrate the advantages of NoSQL databases.

Advantage	Details
Structure data to match the application	• JSON documents align more to the way applications are structured and can be serialized/deserialized directly from objects and structures in code • Logically design, organize, and reorganize JSON documents in scopes/collections per requirements
Less rules, faster data	• JSON documents don't need to denormalize all data like in traditional RDBMS • When denormalization is allowed, faster access to related information stored in a single document is enabled
Supports ACID in a smart way	• Data integrity can still be built in if desired or needed for part of an application or microservice • NoSQL doesn't mean ACID transactions can't be supported (anymore)
More freedom to evolve faster	• Structures can change, reducing the burden on DBAs • Agile development can move and evolve quicker
Supports mobile	• Create "offline-first" apps that work even when the network is down • Automatically sync mobile/edge data with remote databases in the cloud • Support multiple mobile platforms with a single backend

DEVELOP WITH AGILITY

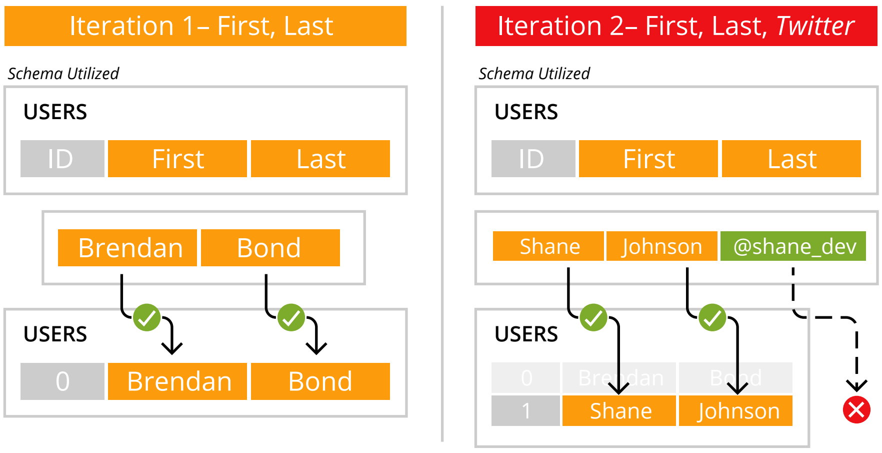
To remain competitive, enterprises must innovate. Changing requirements can put developers under pressure. Speed is critical, but so is agility, since these applications can evolve far more rapidly than legacy applications. Relational databases have a restricted, flat data structure and don't respond well to frequent changes in the data model.

Adaptable NoSQL schema requirements

A core principle of agile development is responding to change. When the requirements change, the data model also changes. This is a problem for relational databases because the data model can be cumbersome to change. In order to change the data model, developers have to modify the schema, or in some organizations, formally request a "schema change" from the database administrators. This slows down or stops development, not only because it is a manual, time consuming process, but because it also impacts other applications and services.

FIGURE 1

RDBMS – An explicit schema prevents the addition of new attributes on demand

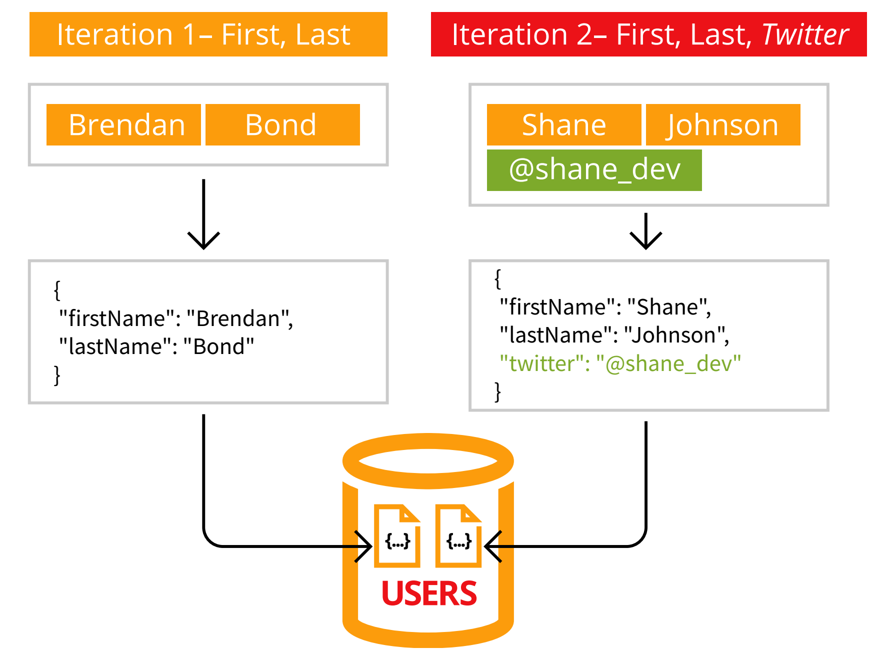


Flexibility for faster development

By comparison, a NoSQL document database fully supports agile development, because it is schema-less and does not statically define how the data must be modeled. Instead, it 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.

FIGURE 2

JSON – The data model evolves as new attributes are added on demand



```
{
 "firstName": "Brendan",
 "lastName": "Bond"
}
```

```
{
 "firstName": "Shane",
 "lastName": "Johnson",
 "twitter": "@shane_dev"
}
```

Simplicity for easier development

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

One problem with relational data modeling is that data is read and written by disassembling, or “shredding,” (like a paper shredder turns a single document into multiple smaller strips) and reassembling objects. This is the object-relational “impedance mismatch.” The workaround is object-relational mapping (ORM) frameworks, which can be effective in simple scenarios, but can become problematic in more complex situations.

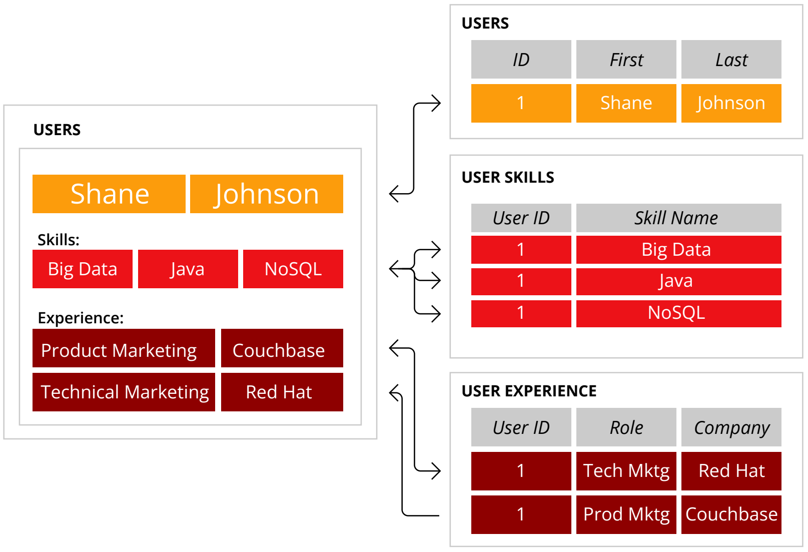
Consider an application for managing resumes. It interacts with resumes as an object of user objects. It contains an array for skills and a collection for positions. However, writing a resume to a relational database requires the application to “shred” the user object.

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

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

FIGURE 3

RDBMS – Applications “shred” objects into rows of data stored in multiple tables



USERS

ID	*First*	*Last*
1	Shane	Johnson

USER SKILLS

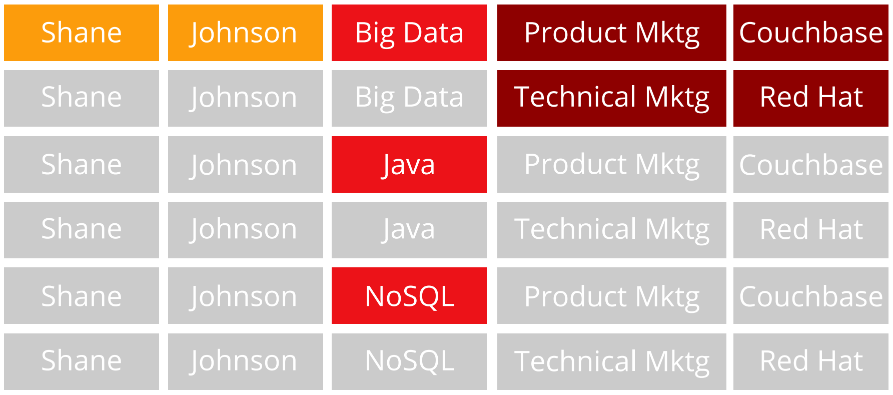
User ID	*Skill Name*
1	Big Data
1	Java
1	NoSQL

USER EXPERIENCE

User ID	*Role*	*Company*
1	Tech Mktg	Red Hat
1	Prod Mktg	Couchbase

FIGURE 4

RDBMS – Queries return duplicate data, applications have to filter it out

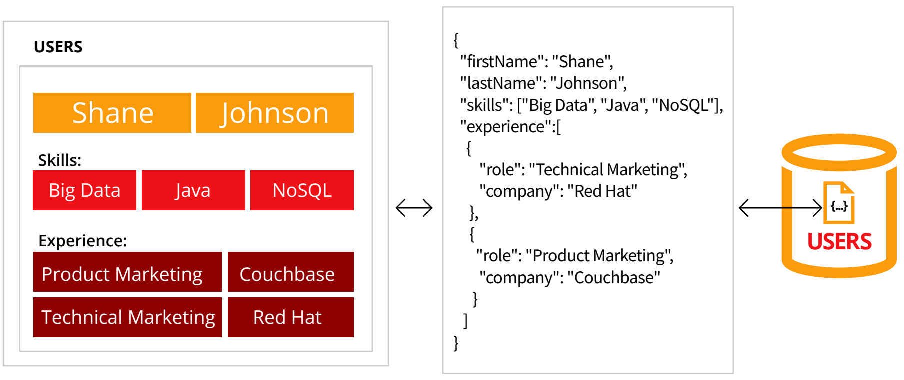


Shane	Johnson	Big Data	Product Mktg	Couchbase
Shane	Johnson	Big Data	Technical Mktg	Red Hat
Shane	Johnson	Java	Product Mktg	Couchbase
Shane	Johnson	Java	Technical Mktg	Red Hat
Shane	Johnson	NoSQL	Product Mktg	Couchbase
Shane	Johnson	NoSQL	Technical Mktg	Red Hat

In contrast, a document-oriented NoSQL database reads and writes data formatted in JSON—which is the de facto standard for consuming and producing data for web, mobile, and IoT applications. It not only can eliminate the object-relational impedance mismatch, it can eliminate 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.

FIGURE 5

JSON – Applications can store objects with nested data as single documents



```
{
 "firstName": "Shane",
 "lastName": "Johnson",
 "skills": ["Big Data", "Java", "NoSQL"],
 "experience":[
  {
    "role": "Technical Marketing",
    "company": "Red Hat"
  },
  {
   "role": "Product Marketing",
    "company": "Couchbase"
   }
  ]
}
```

Grouping documents to ease access

Unlike using predefined sets of schemas to differentiate tables from one another, NoSQL databases have a concept, such as buckets, that serve as a general holding area for all documents. A database can have many logical, named buckets for various purposes. The name is provided while connecting or requesting data and allows applications to have their own area in the system.

Within those buckets are additional hierarchical logical groupings that can be restricted to particular users or roles. These are called collections and/or scopes, allowing subsets of documents in a bucket to be named. Because this flexibility helps segregate data of one user or application from another, the developer does not have to build their own security and reliability code, but can instead let the underlying database do it.

Querying using SQL

Application developers that are used to querying with SQL can continue to use the same language in NoSQL platforms but operate against the JSON data that is stored. For example, Couchbase provides a SQL-based query standard known as SQL++ (previously known as N1QL) that returns results in JSON with sets of rows and subdocument components where appropriate. This is in contrast to the vast majority of other NoSQL databases (like MongoDB™) that don't use SQL and require developers to climb a new language learning curve.

Standard statements are supported including SELECT … FROM … WHERE syntax. SQL++ also supports aggregation, sorting, and joins (GROUP BY … SORT BY … LEFT OUTER/INNER JOIN). Querying collections, scopes, and even nested arrays is supported. Query performance can be improved with composite, partial, covering indexes, and more.

There are minimal changes for SQL experts to move to SQL++ where desired, with many basic queries working out of the box.

"We wanted a solution that seamlessly works across server and mobile, and that the developers could use without lots of retraining. None of the other solutions came even close to Couchbase's broad enterprise capabilities."

—AVIRAM AGMON, CTO, MACCABI HEALTH CARE

Case study:
https://www.couchbase.com/customers/maccabi

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

What about ACID transactions in NoSQL?

NoSQL databases also operate as operational systems with large numbers of transactions. When you flatten out a business entity into multiple separate tables, you require a transaction for almost every update. With NoSQL databases, you don't need to flatten out the entity, but can usually contain it in a single document. Updates to a single document are atomic and don't require a transaction.

However, there may be updates that span multiple documents and require a check to ensure "all or nothing" of the transaction occurs. For instance, a transfer of credits from one user's account to another.

This is why NoSQL databases like Couchbase support transactions.

```
START TRANSACTION;
UPDATE customer SET balance = balance + 100 WHERE cid = 4872;
SELECT cid, name, balance from customer;
SAVEPOINT s1;
UPDATE customer SET balance = balance – 100 WHERE cid = 1924;
SELECT cid, name, balance from customer;
ROLLBACK WORK TO SAVEPOINT s1;
SELECT cid, name, balance from customer;
COMMIT ;
```

The combination of transactions and SQL greatly expand the number of use cases where a NoSQL database can be considered. In the past, the inability to join or handle transactional operations meant that NoSQL databases were only chosen for the highest volume and scale use cases. But the option of using SQL and transactions means that NoSQL databases can also be chosen for traditional RDBMS cases that need more flexibility and power.

Additionally, by selecting a transactional NoSQL database, many traditionally complex applications can be simplified because there is no need for an ORM tool.

"For many, many years, we said, 'Wouldn't it be nice to have a data store where we could go from the Java object right into the database and back without a big translation and lots of overhead?' Well, this is it."

—THOMAS VIDNOVIC,
SOLUTIONS ARCHITECT,
MARRIOTT INTERNATIONAL

Case study:
https://www.couchbase.com/customers/marriott

One data source—multiple access methods

NoSQL databases operate as a primary content store, meaning you enter the data in one application but can access it multiple ways depending on the use case. For example, developers can use direct API calls to access a specific document using a key or through a SQL query that returns multiple rows of data in a JSON response. This is known as "multi-model."

Other access methods are available depending on the database, including full-text search systems that allow natural language search requests. Requests can be made for full or partial "fuzzy" matches, geographic ranges, or wildcard searches. The response includes a JSON document with lists of matching document IDs, contextual information, and a relevancy score.

Full-text search systems are often separate from a database but NoSQL databases like Couchbase include them as part of the underlying system, resulting in a simpler overall architecture.

Big data analytics are possible as well, using complementary subsystems that process larger volumes of historical data. Using advanced indexing and query capabilities, also based on SQL++, more advanced analytics can be done in the same database without needing a separate, external OLAP system.

Because the data is stored and indexed all within the one database product, it allows developers to connect to one system and pass through the relevant requests.

OPERATE AT ANY SCALE

Databases that support web, mobile, and IoT applications must be able to operate at any scale. While it is possible to scale a relational database like Oracle (using, for example, Oracle RAC), doing so is typically complex, expensive, and not fully reliable. NoSQL distributed databases—designed with a scale-out architecture and no single point of failure—provides compelling operational advantages.

A distributed NoSQL database runs on commodity hardware to **scale out**—i.e., add more resources simply by adding more servers to a cluster (sometimes known as “horizontal scaling”). 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) applying less expensive hardware and/or cloud infrastructure; and (c) scaling on demand and without downtime.

Availability is also important. 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 the single server or shared storage fails, the database becomes unavailable, applications stop, and customers get frustrated.

Being available in constrained environments means being fast. NoSQL databases sometimes need an additional in-memory layer to provide sub-millisecond response time. Some (like Couchbase) natively integrate an in-memory layer, making it easier to operate at no additional cost.

A distributed NoSQL database includes built-in replication between data centers—no separate software is required. In addition, some (like Couchbase) include bidirectional replication enabling full active-active deployments to multiple data centers. This enables the database to be deployed in multiple countries or regions while providing local data access to local applications and their users.

DATABASE-AS-A-SERVICE

BENEFITS OF DBAAS

- Rapid setup
- Easily scale or evolve configurations
- High service levels
- Security automation

Many organizations are looking to reduce their operational efforts and costs of running software and hardware, and databases are no exception. Typically a Database-as-a-Service, or DBaaS, streamlines and improves operations and reduces the amount of work that teams have to do, for example: IaaS setup and configuration, database provisioning, database provisioning, operations management, scaling automation, monitoring, and security.

Operational management reduces many of the tasks of maintaining a database environment, allowing companies to focus more time and effort on core business activities. These operational processes can include ongoing configuration, patching, upgrades, backup and recovery activities, and overall system monitoring.

BENEFITS OF DBAAS

DBaaS offerings limit the work of often overstretched IT teams, providing convenience and opportunities to focus on more high-value projects. New databases can often be spun up in minutes instead of a traditional multi-week provisioning process.

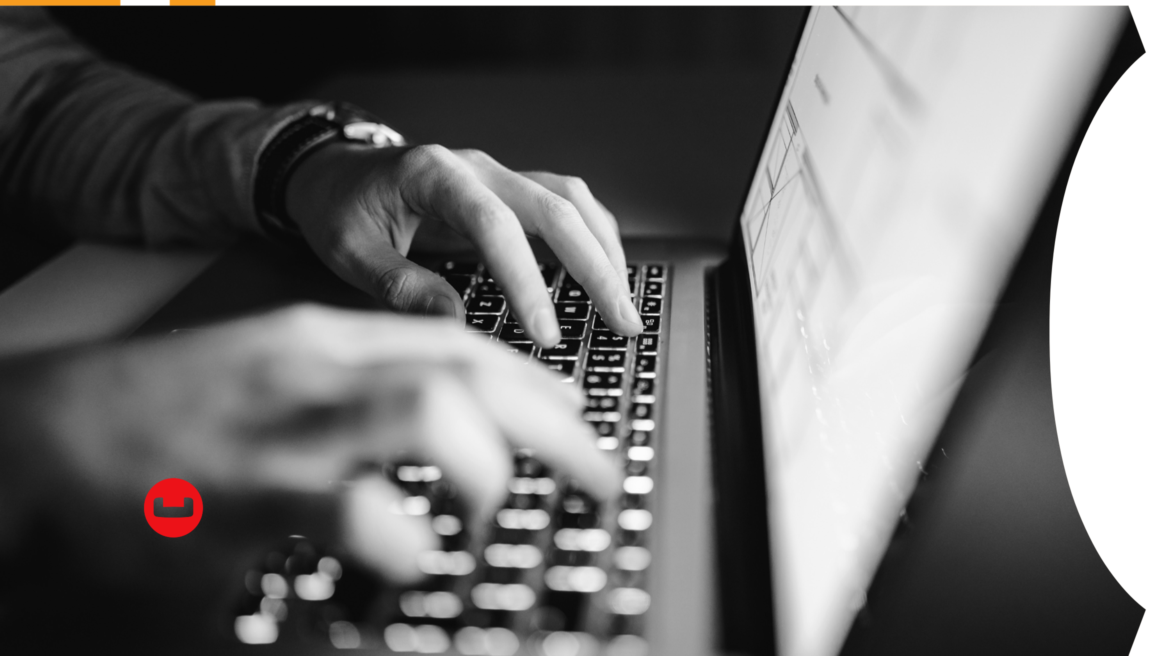
From both a financial and operations perspective, companies see benefits like:

- **Rapid setup –** DBaaS capabilities allow users to provision new database instances when needed in a self-service, highly automated fashion. Developers can more quickly test out new projects to drive company innovation.
- **Easily scale or evolve configurations –** As the needs of users and applications change, modify the configuration of clusters to match those needs. The database makes it easy to match regional needs and keep up with regulatory changes.
- **High service levels –** Most DBaaS systems provide at least a 99% uptime SLA, and often much higher. Replication and redundancy architecture improve quality even further as deployments go global.
- **Security automation –** Advanced DBaaS systems build in multiple levels of security and encryption to protect data at rest, in transit, and throughout the data's lifecycle.

NOSQL IS A BETTER FIT FOR LARGE-SCALE REQUIREMENTS

Tens of thousands of organizations have adopted NoSQL. For many, the use of NoSQL started with a proof of concept, or a single use case, then expanded to more critical applications. Today, the Couchbase NoSQL database is used in thousands of use cases and workloads.

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 businesses.



Couchbase

Modern customer experiences need a flexible database platform that can power applications spanning from cloud to edge and everything in between. Couchbase's mission is to simplify how developers and architects develop, deploy and consume modern applications wherever they are. We have reimagined the database with our fast, flexible and affordable cloud database platform Capella, allowing organizations to quickly build applications that deliver premium experiences to their customers—all with best-in-class price performance. More than 30% of the Fortune 100 trust Couchbase to power their modern applications.

For more information, visit www.couchbase.com and follow us on Twitter.