Couchbase

Why NoSQL Databases?

Why successful enterprises rely on NoSQL database applications

What is NoSQL?

NoSQL is a modern type of database management system that stores information in JSON documents instead of the columns and rows used by relational databases. It delivers data for applications in ways that make development easier and more robust.

As a result, NoSQL databases are built from the ground up to be flexible, scalable, and capable of rapidly responding to the data management demands of modern, digital businesses.

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

Customer experience drives enterprise to NoSQL solutions

The NoSQL revolution has been driven by the increased demands for engaging customer interactions and the need to develop a competitive edge, especially with real-time web applications.

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, resilient, and responsive.

Today's enterprises are interacting digitally – not only with their customers, but also with their employees, partners, vendors, and even their products – at an unprecedented scale.

These network, edge, and web-powered applications are at the heart of the NoSQL revolution: the Internet of Things (IoT), social media, big data analytics, a company's internal or external cloud, mobile/edge computing, and more.

Bringing together all of these new systems requires more flexibility, performance, and scalability that allows multiple types of systems to be consolidated into a single platform.

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, and today's 5G-enabled, distributed, digital economy. 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 servers – processors, memory, and storage – to scale up as Moore's Law allowed.

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, and Amazon Dynamo research paper in 2007. Efficient distributed 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** and to **operate at any scale**.

Agility meant providing flexible schemas, useful APIs, robust SQL-based querying, text analytics, and more. While scalability allowed data volumes to grow and without sacrificing performance and stability.

Supporting SQL and NoSQL developers

Traditional relational systems simply manage tabular data and return it as rows and columns. NoSQL databases can do this as well, however, modern NoSQL applications do not force application developers into using a static schema that has to be reworked 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 in the form of full or partial JSON data structures, full-text search matches, tabular SQL query rows, key-based objects, or even big data analytic systems.

This convergence of technologies simplifies the information architecture of enterprises and helps developers deliver applications more efficiently without needing to learn a dozen different platforms.

Scaling beyond SQL databases

To operate at scale, the new NoSQL systems approach required cluster-based computing with efficient node interconnects to keep data synchronized and flowing at high speeds.

For example, technical leaders are now expected to build enterprise data management infrastructure that includes the following characteristics:

- 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
- Rapidly adapt to changing requirements with frequent updates and new features

SQL-based relational databases are unable to meet these new requirements whereas NoSQL databases can.

Consider just a few examples of Global 2000 enterprises that are deploying NoSQL for mission-critical applications that have been featured in recent news reports:

- **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
- **Viber,** processes over 15B events per day for their over 1B customers
- **GE** deploys NoSQL for its Predix platform to help manage the industrial internet

Today's trends – tomorrow's challenges

Today's customer experience goals depend on tightly aligned technical integration more than ever before, but it must fit perfectly with the digital economy trends going forward or risk becoming outdated.

Some of the high-level technical goals include:

- Consolidated platforms that work together efficiently
- Simplified system architectures that are easy to manage
- Effective data "plumbing" for real-time web applications and low latency
- Act as a service layer that pushes data as close to the customer as possible

Five advantages of NoSQL databases in today's digital economy

The broader digital business economy trends have introduced new challenges that push the above goals even further. Ambitious customers with big ideas for how to use data have unleashed a new set of technology requirements for CIOs and technical leaders.

Here are five trends that play into the advantages of NoSQL databases for addressing the challenges of operating in a digital economy.

Digital Economy Trends	Requirements
1. Customer shift continues to online	• Scaling to support thousands or even millions of users • Meeting UX requirements with consistently high performance • Maintaining availability 24 hours a day, 7 days a week
2. The internet is connecting everything	• Supporting many different applications with different data structures • Ensuring software is "always on" with no excuse for downtime • Supporting continuous streams of data from the real-time web
3. Big data is getting bigger	• Storing customer-generated semi-structured and unstructured data • Storing different types of data from different sources in the same infrastructure or even the same cluster • Storing data generated by thousands or millions of customers and things

4. Applications are moving to the cloud	• Scaling on demand to support more customers, and store more data • Operating fully managed applications on a global scale to support customers worldwide • Minimizing infrastructure costs, achieving a faster time to market
5. The world has gone mobile	• Creating "offline-first" apps – network connection not required • Synchronizing mobile/edge data with remote databases in the cloud • Supporting multiple mobile platforms with a single backend

The above requirements are extensive and challenge even the best systems to do even more with less. Today's extreme requirements can be loosely grouped into two categories that impact two different levels of end users:

- Providing agile platforms for application developers to excel
- Supporting scalable system architectures that outperform others

Develop with agility

To remain competitive in the digital economy, enterprises must innovate – and now they have to do it faster than ever before. As this innovation centers on the development of modern web, mobile, and IoT applications, developers are under extraordinary pressure. Speed is critical, but so is agility, since these applications evolve far more rapidly than legacy applications like ERP. Relational databases require a relatively restricted flat data structure and don't respond well to frequent changes in the data model. This simply doesn't meet the needs of modern, frequently changing applications and business requirements.

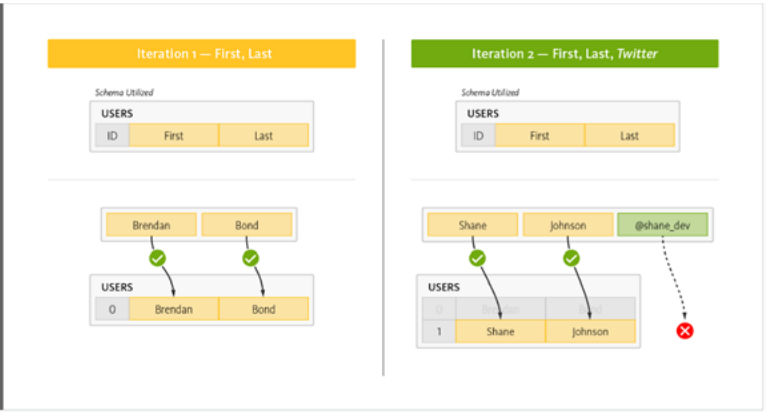
All agile platforms provide a flexibility that allows faster, easier, application development. Some of these benefits are in how the platform handles data, others are in how applications can interact with the database.

Adaptable NoSQL schema 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 is a manual, time-consuming process, but it also impacts other applications and services.

Figure 1

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

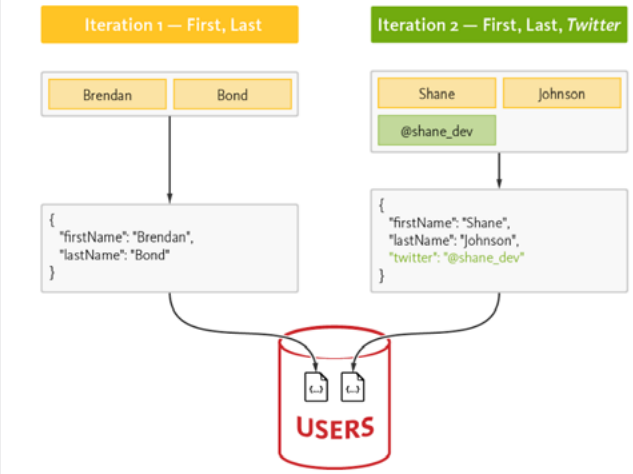


Flexibility for faster development

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.



Simplicity for easier development

Applications and services model data as objects (e.g., employee profile), multi-valued data as collections (e.g., roles), and related data as nested objects or collections (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. 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."

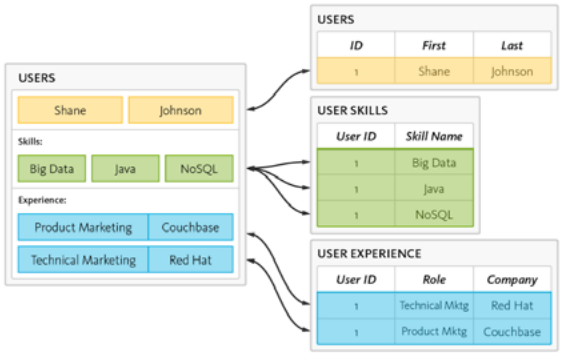
The workaround is object-relational mapping (ORM) frameworks, which are inefficient at best, problematic at worst.

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**.

Figure 3

RDBMS – Applications "shred" objects into rows of data stored in multiple tables.



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

Figure 4

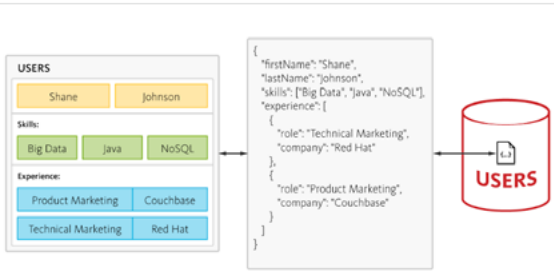
RDBMS – Queries return duplicate data, applications have to filter it out.

Shane	Johnson	Big Data	Product Marketing	Couchbase
Shane	Johnson	Big Data	Technical Marketing	Red Hat
Shane	Johnson	Java	Product Marketing	Couchbase
Shane	Johnson	Java	Technical Marketing	Red Hat
Shane	Johnson	NoSQL	Product Marketing	Couchbase
Shane	Johnson	NoSQL	Technical Marketing	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 eliminates the object-relational impedance mismatch, it 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**.

Figure 5

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



Grouping documents to ease access

Unlike using predefined sets of schemas to differentiate tables from one another, NoSQL databases will 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 language 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. N1QL 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 N1QL where desired, with all the basic queries working out of the box.

SQL	N1QL
SELECT p.FirstName + ' ' + p.LastName AS Name, d.City FROM AdventureWorks2016.Person.Person AS p INNER JOIN AdventureWorks2016.HumanResources.Employee e ON p.BusinessEntityID = e.BusinessEntityID INNER JOIN (SELECT bea.BusinessEntityID, a.City FROM AdventureWorks2016.Person.Address AS a INNER JOIN AdventureWorks2016.Person.BusinessEntityAddress AS bea ON a.AddressID = bea.AddressID) AS d ON p.BusinessEntityID = d.BusinessEntityID ORDER BY p.LastName, p.FirstName;	SELECT p.FirstName || ' ' || p.LastName AS Name, d.City FROM AdventureWorks2016.Person.Person AS p INNER JOIN AdventureWorks2016.HumanResources.Employee e ON p.BusinessEntityID = e.BusinessEntityID INNER JOIN (SELECT bea.BusinessEntityID, a.City FROM AdventureWorks2016.Person.Address AS a INNER JOIN AdventureWorks2016.Person.BusinessEntityAddress AS bea ON a.AddressID = bea.AddressID) AS d ON p.BusinessEntityID = d.BusinessEntityID 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.

This is why NoSQL databases like Couchbase support transactions.

```
Transactions transactions = Transactions.create(cluster,
        TransactionConfigBuilder.create()
                .durabilityLevel(TransactionDurabilityLevel.PERSIST_TO_
                MAJORITY)
                .logOnFailure(true, Event.Severity.WARN)
            .build());

   TransactionResult result = transactions.run((ctx) -> {
        // Inserting a doc:
        ctx.insert(collection, “doc-a”, JsonObject.create());

        // Getting documents:
        // Use ctx.getOptional if the document may or may not exist
        Optional<TransactionGetResult> docOpt =
                ctx.getOptional(collection, “doc-a”);

        // Use ctx.get if the document should exist, and the transaction
        // will fail if it does not
        TransactionGetResult docA = ctx.get(collection, “doc-a”);

        // Replacing a doc:
        TransactionGetResult docB = ctx.get(collection, “doc-b”);
        JsonObject content = docB.contentAs(JsonObject.class);
        content.put(“transactions”, “are awesome”);
        ctx.replace(docB, content);

        // Removing a doc:
        TransactionGetResult docC = ctx.get(collection, “doc-c”);
        ctx.remove(docC);

        ctx.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.

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.

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, allowing managers to simplify the overall architecture.

Big data analytics are possible as well, using complimentary subsystems that process larger volumes of historical data. Using advanced indexing and query capabilities, based on a language known as SQL++, more advanced analytics can be done in the same database without needing an 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. With Oracle, for example, scaling out using RAC technology requires numerous components and creates a single point of failure that jeopardizes availability.

By comparison, a NoSQL distributed database – designed with a scale-out architecture and no single point of failure – provides compelling operational advantages.

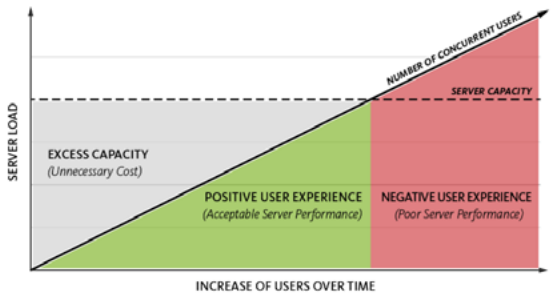
Elasticity for performance at scale

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., only being able to scale by 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.

Figure 6

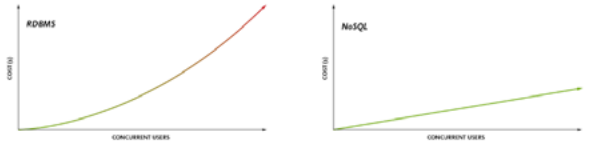
RDBMS – The server is too big or too small, leading to unnecessary costs or poor performance.



A distributed NoSQL database, however, leverages commodity hardware to scale out – i.e., add more resources simply by adding more servers to a cluster. 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.

Figure 7

NoSQL – Add commodity servers on demand so the hardware resources matche the application load.



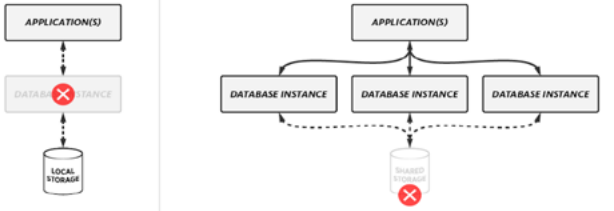
By distributing reads, writes, and storage across a cluster of nodes, NoSQL databases are able to operate at any scale. Additionally, they are designed to be easy to configure, install, and manage both small and large clusters.

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, applications stop, and customers disengage.

Figure 8

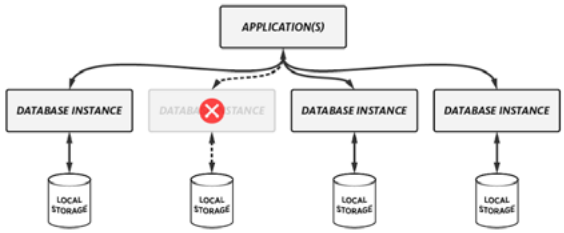
RDBMS – The failure of a server or storage device brings down the entire database.



In contrast to relational technology, a distributed, NoSQL database partitions and distributes data to multiple database instances with no shared resources. In addition, the data can be replicated to one or more instances for high availability (intercluster replication) and in different geographic locations. While relational databases like Oracle require separate software for replication, for example, 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.

Figure 9

NoSQL – If an instance fails, the application can send requests to a different one.

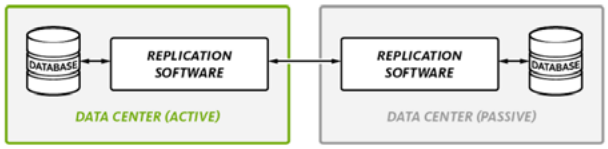


Customer behavior now requires organizations to support multiple physical, online and mobile channels in multiple regions and often multiple countries. While deploying a database to multiple data centers increases availability and helps with disaster recovery, it also has the benefit of increasing performance too. All reads and writes can be executed on the nearest data center, thereby reducing latency.

Ensuring global availability is difficult for relational databases where separate add-ons are required – which increase complexity – or where replication between multiple data centers can only be used for failover, because only one data center is active at a time. Oracle, for example, requires Oracle GoldenGate. When replicating between data centers, applications built on relational databases can experience performance degradation or find that the data centers are severely out of sync.

Figure 10

RDBMS – Requires separate software to replicate data to other data centers.

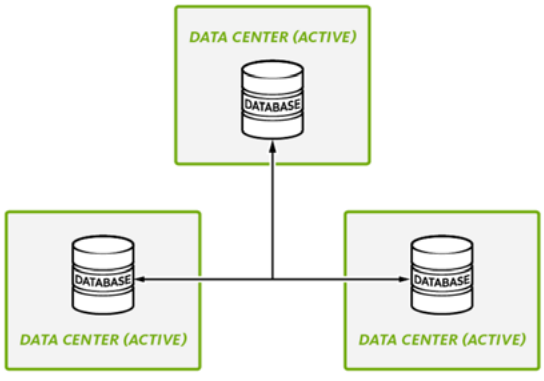


A distributed, NoSQL database includes built-in replication between data centers – no separate software is required. In addition, some include bidirectional replication enabling full active-active deployments to multiple data centers. This enables the database to be deployed in multiple countries or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

Deploying to multiple data centers not only improves performance, but enables immediate failover via hardware routers. Applications don't have to wait for the database to discover the failure and perform its own failover.

Figure 11

NoSQL – Replication between data centers is fully built-in, and can be bi-directional.



NoSQL is a better fit for digital economy requirements

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 a greater scale. 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 of smaller businesses and startups, have adopted NoSQL. For many, the use of NoSQL started with an open source cache, proof of concept or a small application, then expanded to targeted mission-critical applications, and is now the foundation for all application development. Today, the Couchbase NoSQL database serves thousands of these types of customers.

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

About Couchbase

Unlike other NoSQL databases, Couchbase provides an enterprise-class, multicloud to edge database that offers the robust capabilities required for business-critical applications on a highly scalable and available platform. As a distributed cloud-native database, Couchbase runs in modern dynamic environments and on any cloud, either customer-managed or fully managed as-a-service. Couchbase is built on open standards, combining the best of NoSQL with the power and familiarity of SQL, to simplify the transition from mainframe and relational databases.

Couchbase has become pervasive in our everyday lives; our customers include industry leaders Amadeus, American Express, Carrefour, Cisco, Comcast/Sky, Disney, eBay, LinkedIn, Marriott, Tesco, Tommy Hilfiger, United, Verizon, as well as hundreds of other household names. For more information, visit www.couchbase.com.

3250 Olcott Street | Santa Clara, California 95054 | 1-650-417-7500 | www.couchbase.com