NoSQL for Games and Gaming

Couchbase

TABLE OF CONTENTS

Couchbase

NoSQL for Games and Gaming

Executive summary

The world's leading game developers and publishers – including Blizzard Entertainment, PopCap Games, Playtika, Jagex, Kabam, SGN, and many more – are using NoSQL databases to better compete in the Digital Economy. They're not only improving player experience, they're improving player engagement – becoming online platforms with downloadable content (DLC), microtransactions, and social networking.

Reach hundreds of millions of users – with chart-topping games going from zero to 10, 20, 30 million daily active users in a matter of weeks.

The success of digital distribution platforms for PC (e.g., Steam), console (e.g., Xbox Games Store), and mobile (e.g., App Store) not only lowered the barrier of entry, it enabled developers and publishers to reach hundreds of millions of users – with chart-topping games going from zero to 10, 20, 30 million daily active users in a matter of weeks. However, in order to increase retention and monetization, developers must deliver updates faster, and more frequently.

It all leads to higher performance, scalability, and agility requirements, and this is where relational databases fall short. It's why leading game developers and publishers are turning to NoSQL databases. They need an operational database that can:

- Perform reads and writes with no perceptible delay
- Scale to millions of users on demand, and with ease
- Guarantee 24x7x365 access to data with zero downtime
- Support a global user base with multisite deployments
- Handle iterative, evolving data models with a flexible schema
- Stream data to Spark and Kafka in real time for constant analysis

The operational database must also be versatile enough to support many use cases – e.g., user profile management, session storage, player and game data services and messaging, to name just a few.

This paper provides an overview for CIOs, CTOs, architects, developers, and operations engineers interested in building and supporting game servers with NoSQL:

- Why relational databases (Oracle, SQL Server, MySQL, PostgreSQL, etc.) can no longer meet gaming requirements
- How NoSQL – specifically Couchbase Server, an open source document database – provides a better solution
- Examples of gaming use cases supported by NoSQL, showing how data can be modeled and queried with Couchbase Server

We conclude with a section on how to introduce NoSQL into a relational environment, emphasizing two approaches:

- NoSQL in a microservices architecture
- NoSQL as the primary source of engagement

Game developers and publishers improve player experience and engagement with NoSQL

In the Digital Economy, game developers and publishers compete based on player experience and engagement – underscoring the importance of retention and monetization. Gamers expect to play whenever they want to for as long as they want to, and that requires 100% uptime. Further, they have a low tolerance for noticeable delays, lag being one of them. In the Digital Economy, game developers and publishers win by creating an experience that keeps players engaged, playing longer, and spending more.

However, with competition growing and a new game topping the charts every month, retention is no longer desirable, it's necessary. It requires an active community and frequent updates – whether it's downloadable content, maps and characters, or achievements and rewards – and that requires shorter development life cycles. With Steam Early Access and Apple TestFlight, for example, game developers and publishers can now iterate faster – trying new things, testing updates, and soliciting feedback.

For architects, developers, and operations engineers, the pressure to deliver great player experiences with higher engagement has never been more important. As a result, game developers and publishers, from PC to consoles to mobile – companies like Blizzard Entertainment, PopCap Games, Playtika, Jagex, Kabam, SGN, and many more – are embracing NoSQL to improve player experience and engagement.

NoSQL is a modern database technology developed about a decade ago by leading internet companies, including Google, Facebook, Amazon, and LinkedIn, to overcome the limitations of relational databases.

Database performance and availability are key to delivering a great player experience

When it comes to player experience, performance and availability are critical. Whether gamers know it or not, they're interacting with a database. They're accessing player data, game data, and session data – and if the data is not readily available, their experience suffers.

Gamers expect every request, whether it's logging in, viewing their achievements, seeing if their friends are online, or joining a match, to be handled immediately – not within seconds, but within milliseconds. It doesn't matter what time it is, or what time zone they're in. It doesn't matter if it's 12 p.m. or 12 a.m., if they're at home or traveling abroad – gamers expect to be able to play 24 hours a day, 365 days a year.

Historically, relational databases have been the bottleneck, as DBAs and developers try to squeeze out every last drop of performance and work around the clock to maintain availability. However, with an ever-growing number of gamers and rising player expectations, they're fighting an uphill battle.

It's one reason why game developers and publishers are embracing NoSQL: They require a database that's engineered for a higher level of performance and availability – capable of meeting the expectations of an online, 24/7, global player base.

The pressure to deliver great player experiences with higher engagement has never been more important.

NoSQL distributed databases provide better performance and availability than relational

Typically, relational databases run on a single server, so the resources available to the database – processing, storage, and memory – are not only limited, they're fixed. This was not a problem when relational databases powered internal-facing applications with a limited number of users and predictable workloads. However, games with thousands, if not millions, of players can overwhelm relational databases with increasing, even volatile workloads.

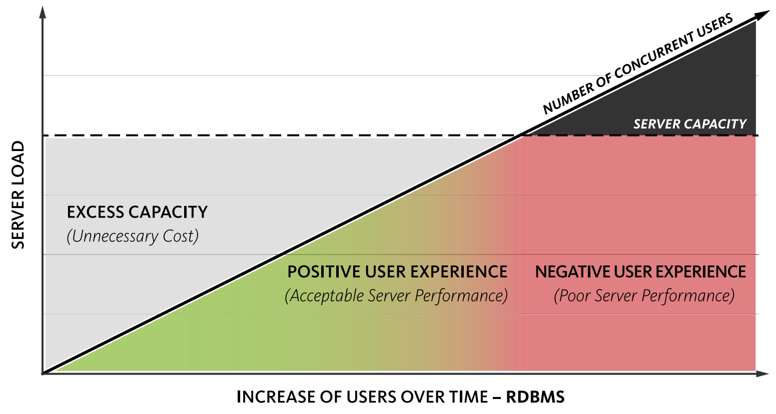

Figure 1: Relational databases suffer from under/over-provisioning, leading to performance and cost issues.

The only way to maintain performance as the number of players grows, especially when a game becomes a hit, is to add more resources – often on demand. With relational databases, this is both difficult and costly, because you have to replace the RDBMS server with a bigger, more expensive server. But with many NoSQL databases, it's simply a matter of adding more servers to scale out. That's because they often run on a cluster of servers – i.e., they're distributed databases.

The only way to maintain performance as the number of players grows, especially when a game becomes a hit, is to add more resources – often on demand.

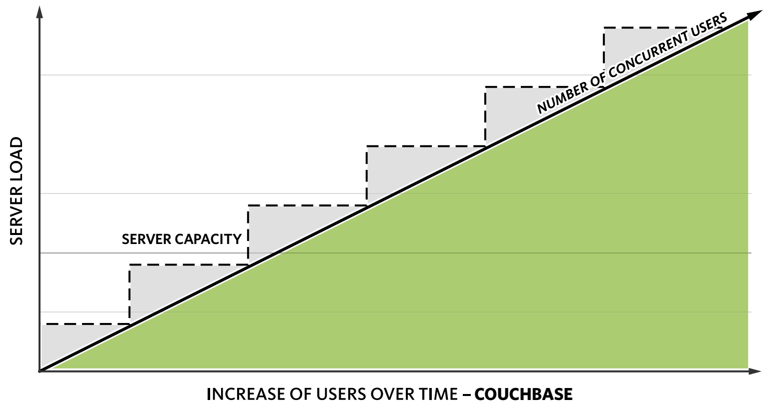

Figure 2: NoSQL databases can increase capacity in small increments for more efficient use of hardware.

The same is true for availability. If a relational database is running on a single server, and that server fails, the data becomes unavailable. NoSQL databases, when running on a cluster of servers, not only tolerate hardware failure, they expect it. That's why the data is replicated to multiple nodes. If one fails, the data remains available on the others.

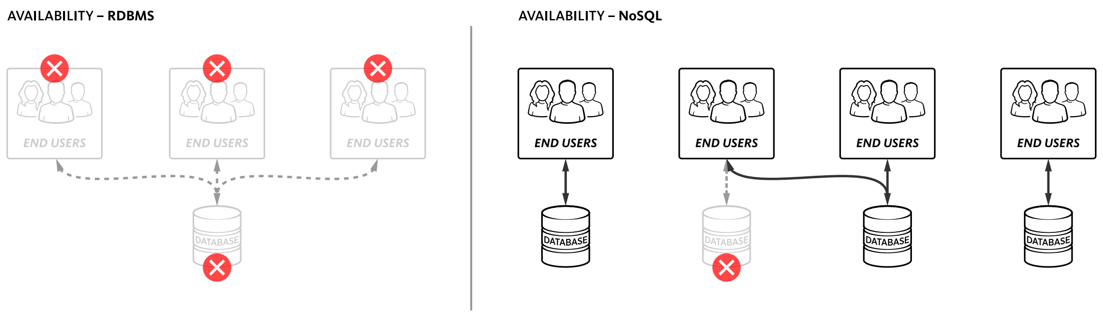

Figure 3: Relational databases suffer from a single point of failure, but NoSQL databases do not.

NoSQL delivers faster, easier, more affordable scalability

For a NoSQL database to effectively scale, it requires two things: a flat topology and bidirectional replication across data centers.

Single node type simplifies scaling

It's not enough for a NoSQL database to be able to scale – it has to scale on demand and with ease. This can be difficult for NoSQL databases that rely on complex topologies with multiple moving parts, like master-slave or other specialized node type topologies, for example. Such complexity introduces barriers to easy scaling, because it requires careful configuration of different nodes with different roles and different relationships.

Rather, every node should be capable of performing the same roles – a single node type. A flat topology with a single node type is easy to scale because it requires little more than adding a node to the cluster.

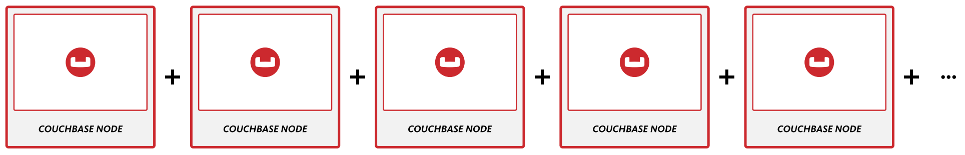

Figure 4: It's easy to scale with a single-node architecture – start more nodes, add them to the cluster.

Bidirectional replication improves availability and data locality

It's not enough for a NoSQL database to be able to scale – it has to scale on demand and with ease.

Bidirectional replication between data centers is essential because players are often not confined to a single geography; they can be in different states, countries, or regions. In order to provide the highest level of performance and availability to a geographically distributed player base, the database must also be geographically distributed.

However, being geographically distributed isn't enough: the database must be deployed in an active-active configuration using bidirectional replication. This enables games to read and write to the data center closest to the player – providing local data access for the best possible performance. For example, player data for U.S. gamers could be updated in a North America data center while player data for U.K. gamers could be updated in an Europe data center. It would be far more efficient than updating all player data in a single, possibly geographically distant, higher latency, data center.

In addition, with active-active configurations, an entire data center can fail without interrupting availability. That's because applications can simply write to a different data center, or requests can be automatically rerouted by routers and load balancers. For example, if the North America data center failed, U.S. gamers would access their player data from the Europe data center – and they wouldn't even know it.

Providing this kind of seamless failover without interrupting player experience is a challenge for NoSQL databases that rely on unidirectional replication (master-slave) between data centers. They can provide local access to mostly read-only data, like user profiles, but they can't do it for things like unlocking achievements, using rewards, or sending messages – i.e., data services for frequent, user-generated data. In addition, because writes can only be handled by the data center with the primary node, applications have to wait for a new primary to be elected should the original fail – resulting in temporary loss of availability, potentially on a global scale.

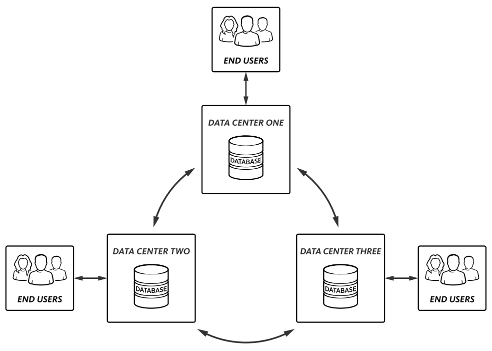

Figure 5: Deploy to multiple data centers in an active/active configuration with bidirectional replication.

Couchbase Server combines single node type and bidirectional replication

Among leading NoSQL databases, Couchbase Server stands out as the only one that combines a single node type architecture (i.e., flat topology) and bidirectional replication between data centers, to deliver fast and easy scalability, along with high performance and availability.

NoSQL drives increased operational efficiency: lowers costs and accelerates time to market

In addition to improving the player experience with greater performance and availability, NoSQL databases improve operational efficiency and shorten development life cycles – reducing costs and enabling a faster time to market.

Elastic scaling optimizes hardware usage

Database workloads for game servers can be highly volatile and unpredictable – exponential growth within weeks of a launch, as well as the introduction of new downloadable content, highly anticipated sequels, or special editions. As a result, it can be challenging to maintain operational efficiency. When a relational database server is configured for peak workloads, and it's not operating under one, then it's over-provisioned.

Hardware, like any other resource, must be elastic – it must be possible to use more or less of it depending on demand. NoSQL databases can scale out *and* scale in. Capacity can be adjusted simply by increasing or decreasing the number of nodes in a NoSQL database cluster, depending on workload. With a NoSQL database, it's possible to always use only as much hardware as needed – never too much, never too little. Because hardware is always optimized for the current workload, NoSQL is more cost efficient than relational.

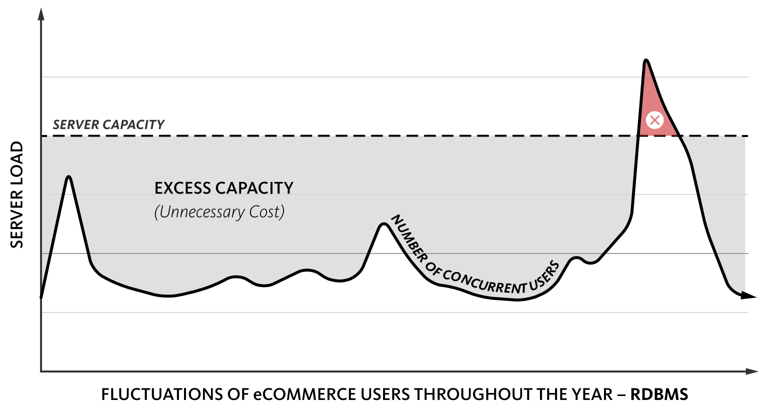

Figure 6: Relational databases with fixed capacities cannot support volatile workloads with high peaks.

Data flexibility speeds innovation and time to market

Innovation is key to competitive success – for game developers and publishers, that means a steady flow of new games, new features, new services, and more.

Innovation is key to competitive success – for game developers and publishers, that means a steady flow of new games, new features, new services, and more. And when it comes to innovation, and in the interests of player acquisition and retention, time to market is critical. Relational databases, and wide-column stores, limit the pace of development due to their static schemas. Adding a new feature often requires changing the database schema, which in turn requires planning and coordination – both time-consuming tasks.

However, NoSQL databases – in particular key-value stores and document databases – leverage dynamic schemas. Rather than defining a static schema, they defer to the application. As a result, developers can create new features without having to wait for a schema change. When the database enables this kind of agile development instead of hindering it, new features can be released far more quickly, resulting in a faster time to market for game developers and publishers.

NoSQL can support numerous video game and gaming use cases

Innovative game developers and publishers are successfully leveraging NoSQL for use cases that involve players, games, and interactions – for everything from player and game data, to user profiles and sessions, to friends and messages, and more. All of these use cases benefit from one or more of the advantages of NoSQL: dynamic schema, low-latency throughput, scalability, and availability.

For example, when stored in a relational database, player data requires a complex schema across multiple tables and complex queries to access it. However, when stored in a NoSQL document database, all of the information for a single player can be stored together in a single JSON document – making it much easier and faster to access.

When managing millions of players, storing their profiles and all of their interactions requires a database that can easily and affordably scale. It's a matter of volume, as well as velocity. Ingesting player interactions and session data to improve engagement and create personalized experiences requires a database capable of sustaining high throughput while maintaining low latency.

Below are some of the common video game and gaming use cases where companies are leveraging NoSQL for greater performance, availability, scalability, and agility:

User profiles
Store user profile data to enable multiple services (e.g., authentication, personalization). User profiles contain data that's specific to the user (e.g., location, preferences, friends) enabling them to play multiple games, often across different platforms and devices.

User sessions
Manage user sessions containing information related to current player state – whether they are playing or between games, how long they've been logged in, where they are and what they're doing – in memory for low-latency performance, and enabling them to resume play if they're interrupted.

Innovative game developers and publishers are successfully leveraging NoSQL for use cases that involve players, games, and interactions.

Player data
Maintain data for player achievements and rewards, statistics and points, leaderboards and rankings, social activity feeds, teams and clans, and more. It's data about a player, or all players, within the context of a game – all the signs of an active, social, and competitive community.

Game data
Handle game data for everything from a marketplace to matches to item state. It begins with the status (e.g., players, games, and game servers) and it ends with the results (e.g., winners and scores), all while capturing progression and resource events in between (e.g., mission completed, item used).

Messaging
Enable players to message their friends, invite them to join games, and receive notifications about changes to their state (e.g., offline, online, playing) – all things necessary to create a social environment. Further, use messaging to provide in-game support, engaging players who are having problems.

NoSQL is well suited to many data types

NoSQL document databases are well suited to many types of data – making them very effective as a general-purpose database for game servers. They are ideal for user profiles and users, player and game data, and social/messaging data – all part of today's player experience, whether PC, console, or mobile.

In the end, there is more than one way to model game data with a relational database. However, they all require workarounds to get past the limitations of relational data models.

USERS	PROFILE	PREFERENCES	HISTORY	DEVICES
SESSIONS	STATUS	STATE	DURATION	LOCATION
PLAYERS	ACHIEVEMENTS	MEDALS	REWARDS	STATISTICS
GAMES	STATUS	PROGRESS	RESOURCES	ACTIONS
MESSAGES	DIRECT	GROUP	SUPPORT	ATTACHMENTS (META)
MARKETPLACE	CURRENCY	ITEMS	FAVORITES	FILES (META)

Figure 7: The types of use cases and data that game servers use NoSQL databases for.

Use case spotlight: game data

As a concrete example of how NoSQL can be a better fit for game servers than relational databases, let's consider a common use case: game data.

Game data is challenging for relational databases
Game data can be very complex. In addition to basic information, games can have significantly more data associated with them. An instance may be a match. Matches may have teams, and multiple rounds. Teams have players. The game data can include game type, game status, teams, players, events, and more.

In the end, there is more than one way to model game data with a relational database. However, they all require workarounds to get past the limitations of relational data models. No one approach is ideal. They all introduce new problems and challenges for developers.

Game data can easily be modeled with a NoSQL database
It's much easier to model game data with a NoSQL document database, because all data for a single game can be stored together. It can all be stored in a single document instead of multiple rows, often in multiple tables. Not only is it easier to model the data, it's simpler and faster to access – there's no need to perform a query with multiple joins or to pivot the results.

key:
games::1234566

document:
```
{
  "type" : "match",
  "server": "127.0.0.1",
  "date": "09-01-2016",
  "status": "active",
  "startTime": "05:00",
  "gameType": "One Flag CTF",
  "map": "Area 51",
  "winner": "red",
  "rounds": [
    {
      "score": "red",
      "player": "players::Ali",
      "minutes": 3},
    {
      "minutes": 5},
    {
      "score": "red",
      "player": "players::David",
      "minutes": 4}],
  "teams": [
    {
      "name": "red",
      "players": [
        "players::Ali",
        "players::David",
        "players::Gary",
        "players::Tim"]},
    {
      "name": "blue",
      "players": [
        "players::Becky",
        "players::Gretchen",
        "players::Kari",
        "players::Stacy"]}]
}
```

Example #1: List the top five scorers

```
SELECT r.player, COUNT(r.player) AS scores FROM matches m
UNNEST m.rounds r
WHERE r.player is NOT MISSING GROUP BY r.player LIMIT 3;
```

player	*scores*
Shane	5
Ali	4

Example #2: Count the number of team wins for Ali

```
SELECT "Ali" AS player, COUNT(*) AS wins FROM matches m
UNNEST m.teams t WHERE m.winner = t.name
AND ARRAY_CONTAINS(t.players, "Ali");
```

player	*wins*
Ali	12

Example #3: Find open matches

```
SELECT  META(m).id AS match,
            TOSTRING(SUM(ARRAY_COUNT(t.players))) || "/12"
AS players FROM matches m UNNEST m.teams t
WHERE m.status = "waiting" AND ARRAY_COUNT(t.players) < 6
GROUP BY m LIMIT 2;
```

match	*players*
matches::1234567	8/12
matches::4567890	5/12

Unique to Couchbase, game data can be queried via a SQL-based query language, N1QL ("nickel"), or via simple key-value operations.

The following examples highlight the **subdocument API** in Couchbase Server. It enables applications to read or write specific fields within a document for maximum performance. This includes updating fields, incrementing counters, adding elements to arrays, and more.

Example #4: Change the game status for match 1234567 to complete

```
bucket.mutateIn("matches:1234567").replace("status", "complete").doMutate();
```

Example #5: Add a player to the red team in match 4567890

```
bucket.mutateIn("matches:4567890").pushBack("teams[0].players", "players::Peter").
doMutate();
```

Unique to Couchbase, game data can be queried via a SQL-based query language, **N1QL** ("nickel"), or via simple key-value operations. In the following examples, the results are displayed as tables for simplicity. However, within applications, the results can be JSON documents, providing much more flexibility – especially when querying multiple product types.

All sorts of player data can easily be modeled in a NoSQL database

It’s easy to model other types of data with NoSQL databases, too – player data, messages, notifications, and more. As with game data, this data is easier and faster to access with a NoSQL database.

In addition, document databases provide a great deal of flexibility by supporting nested elements like arrays and objects. For example, achievements and medals – they could be embedded within the player document, stored as individual documents, or stored as a group of documents. In the example below, we model Halo 5 achievements and medals by storing them together in a separate document per game – making it easier and faster to access.

key:
awards::player::Shane

document:
```
{
  “type”: “awards”,
  “player”: “players::Shane”,
  “game”: “Halo 5”,
  “achievements”: {
    “campaign”: [
      {
        “name”: “Thanks A Killion”,
        “description”: “Kill 20,000 enemies.”,
        “points”: 20},
      {
        “name”: “The Long Haul”,
        “description”: “Complete 500 missions.”,
        “points”: 20},
      {
        “name”: “All Out of Bubblegum”,
        “description”: “Collection 1,000 medals.”,
        “points”: 20}],
    “multiplayer”: [
      {
        “name”: “Multiplayer Champion”,
        “description”: “Win 500 games.”,
        “points”: 20},
      {
        “name”: Sample Plate”,
        “description”: “Complete three playlists.”
        “points”: 20},
      {
        “name”: “Bro Hammer”,
        “description”: “Complete co-op on Heroic.”,
        “points”: 10}]},
  “medals”: [
      “Road Trip”,
      “Pineapple Express”,
      “Rocket Mary”,
      “Unfriggenbelievable”]
}
```

Example #1: List Shane’s multiplayer achievements

```
SELECT m.name, m.points FROM awards a
UNNEST a.achievements.multiplayer m
WHERE a.player = “players::Shane”;
```

name	*points*
Thanks A Killion	20
The Long Haul	20
All Out of Bubblegum	20

Example #2: List players with more than 500 campaign points

```
SELECT a.player, SUM(c.points) points
FROM awards a UNNEST a.achievements.campaign c
GROUP BY a.player HAVING SUM(c.points) > 500;
```

player	*points*
Ali	1000
Shane	800
Karl	660

Example #3: List the three most awarded medals

```
SELECT medals AS medal, count(medals) as count
FROM default a UNNEST a.medals
GROUP BY medals
ORDER BY count DESC LIMIT 2;
```

medal	*count*
Pineapple Express	200
Rocket Mary	120

Example #4: Add a medal

```
bucket.mutateIn(“awards::Shane”).pushBack(“achievements.medals”, “Awesome Sauce”).
doMutate();
```

Modeling messaging data

In addition to player data, modeling **messaging data** with a document database is easy and flexible, too. Below are two examples. The first models messages by sender, the second by conversation. For example, when there are multiple conversations with different people, you can store each conversation as a separate document.

Message (by sender)

Key:
inbox::Matt

Document:

```
{
  "type": "inbox",
  "messages": [
    {
      "messageId": "inbox::Matt::1234567",
      "from": "players::Doug",
      "status": "unread"},
    {
      "messageId": "inbox::Matt::3456789",
      "from": "players::Max",
      "status": "read"}]
}
```

Key:
inbox::Matt::1234567

Document:

```
{
  "type": "message",
  "date": "09-01-2016",
  "time": "22:00",
  "from": "players::Doug",
  "text": "Game over. Thanks for playing!",
  "screenshot": "doug_wins.jpg"}
```

Messages (by conversation)

Key:
conversations::24680

Document:

```
{
  "players": [
    "Gretchen",
    "Kari",
    "Becky"],
  "messages": [
    {
      "from": "players::Becky",
      "date": "09-01-2016",
      "time": "22:00",
      "text": "How can we defeat product
marketing?"},
    {
      "from": "players::Kari",
      "date": "09-01-2016",
      "time": "22:05",
      "text": "I wish I knew. Shane is too
good."},
    {
      "from": "players::Gretchen",
      "date": "09-01-2016",
      "time": "22:07",
      "text": "We need to work together."}]
}
```

Example #1: Display Matt's unread messages

```
SELECT LTRIM(m.`from`, "players::") `from`, ms.date, ms.time, ms.text
FROM inbox i UNNEST i.messages m INNER JOIN messages ms ON KEYS m.messageId
WHERE m.status = "unread";
```

from	*text*	*date*	*time*
Doug	Game over. Thanks for playing!	09-01-2016	22:00

Example #2: Add a message to a conversation

```
JsonObject msg =
  JsonObject.create().put("from", "players::Kari").put("date": "09-01-2016").put("time": "22:10").
put("text", "Really?");
```

```
bucket.mutateIn("conversations::24680").pushBack("messages", msg);
```

How to introduce a NoSQL database into a relational environment

NoSQL document databases are excellent general-purpose databases for game developers and publishers. However, that doesn't mean there isn't a place for relational databases. In particular, relational databases are well suited to legacy business management applications, such as enterprise resource planning (ERP) and supply chain management (SCM). Whereas these are employee-facing applications, NoSQL is better suited to meet the performance, scalability, availability, and agility requirements of games.

In fact, the most innovative game developers and publishers have embraced NoSQL by successfully introducing it into their relational environments. It's a matter of architecture, and there are two popular approaches to adding NoSQL into an existing relational database environment: microservices and caching.

NoSQL document databases are excellent general-purpose databases for game developers and publishers.

NoSQL in a microservices architecture

Many successful game developers and publishers are using microservices. In a microservices architecture, applications are deployed as a set of independent, full-stack services.

An important concept of microservices architecture is that of decentralized data management. Where a monolithic application would store all data in a single database, a set of microservices can and should store their own data in their own databases. This often leads to polyglot persistence – different services using different types of databases. While some services may continue to use a relational database, others may use NoSQL databases – some may use a document database, others may use a graph database.

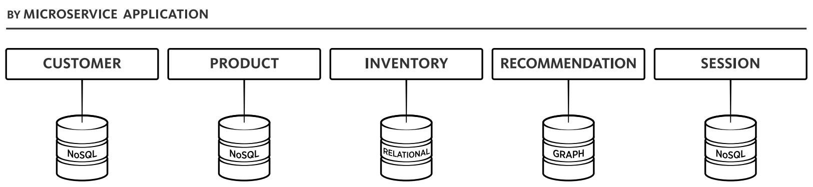

Figure 8: A microservices architecture supports different services using different types of databases.

With a microservices architecture, game developers and publishers can choose when and where to introduce a NoSQL database, enabling them to migrate to NoSQL one service at a time.

Caching with NoSQL

In some environments, it may be better to augment an existing relational database with a NoSQL database. In this architecture, data is loaded into a NoSQL database that is deployed between the game servers and the relational database. In a sense, the NoSQL database functions like a cache. For example, game data can be exported from the relational database to the NoSQL database. When the data changes in the relational database – for example, when new achievements are introduced – the NoSQL database is updated.

However, the data doesn't have to be read-only – game data, for example. The relational database may be the primary source of record, but games will interact with the NoSQL database for faster performance. While it's possible for the game data to become out of date for a brief time – until the relational and NoSQL databases are synchronized – the benefits of better performance, and thus a better player experience, far outweigh the drawbacks.

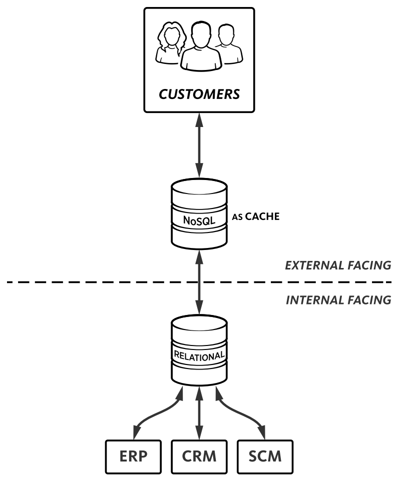

Figure 9: Deploy a NoSQL database on top of a relational database to improve performance and reduce costs.

Applications can leverage Database Change Protocol (DCP) in Couchbase Server to tap into a change stream, and apply these changes to the underlying relational database. In the reverse direction, with Oracle Database for example, applications can use Oracle GoldenGate to capture changes and apply them to Couchbase Server. Finally, applications can use the Couchbase Kakfa Connector to stream changes between Couchbase Server and a relational database in both directions.

In addition to providing better performance and availability, adding a NoSQL database can reduce the costs associated with maintaining relational databases. For example, a NoSQL database can be deployed to lessen the load on a mainframe – reducing MIPS and thus costs – or to avoid having to deploy the relational database on bigger hardware, and thereby avoid higher licensing and maintenance costs.

According to Newzoo's *Global Market Games Report*, mobile gaming will exceed PC in 2016 – 37% of the global market, generating $36.9 billion in revenue.

As video games and gaming go mobile, NoSQL can deliver a better player experience

Every year, more and more gamers are playing on smartphones and tablets. According to Newzoo's *Global Market Games Report*, mobile gaming will exceed PC in 2016 – 37% of the global market, generating $36.9 billion in revenue. However, many game developers and publishers have yet to realize the full potential of mobile platforms.

To create a mobile experience that is on par with, or exceeds, PC and console experiences requires a native mobile solution. A key component of that solution is an embedded database for mobile devices, providing mobile games with local data access. It not only improves performance, because the data is local – it also improves availability, because an embedded database ensures the game will work regardless of whether or not there's an active network connection. Furthermore, the solution must provide built-in synchronization – enabling the database to push data to the player, and vice versa.

A native mobile solution can not only improve PC and console experiences, it can improve real-world experiences too. After all, gamers are beginning to move freely between virtual worlds and the real world. For example, playing a mobile game to earn rewards for acquiring items in the PC/console version, or to receive digital coupons for purchasing toys and accessories in the real world, or to receive notifications when friends are online playing the PC/console version. By leveraging the full power of a native mobile solution, there are many possibilities for improving player experience – mobile, console, or PC.

Couchbase Mobile is the only native, NoSQL-based solution for mobile apps. It is comprised of Couchbase Lite and Couchbase Sync Gateway, and when paired with Couchbase Server, it's a complete platform for mobile games. In addition, it's a cross-platform solution for iOS, Android, .NET/Java, and Apple macOS.

Why Couchbase Server is a great NoSQL solution for video games and gaming

Couchbase Server is a powerful NoSQL solution for a broad range of use cases, delivering the high performance, scalability, availability, and agility today's games require. Numerous game developers and publishers – including Blizzard Entertainment, PopCap Games, Playtika, Jagex, Kabam, SGN, and many more – have chosen Couchbase Server for several key advantages.

Memory-centric architecture
Couchbase Server takes full advantage of all available memory to give your application the sub-millisecond responsiveness that today's shoppers expect.

Integrated cache
While other NoSQL databases like MongoDB require a third-party cache – adding to both cost and complexity – Couchbase Server has a fully integrated cache that delivers blazing performance. No need for a separate product to install and manage.

Powerful, SQL-based query language
Unique among all NoSQL document databases, Couchbase Server provides N1QL ("nickel") – a powerful query language that lets developers easily query JSON data using familiar, SQL-like expressions.

Built-in high availability and disaster recovery
Couchbase Server comes with high availability within a cluster and provides market-leading cross datacenter replication (XDCR) capabilities to support DR and data locality requirements. No need for complicated third-party systems. You have full control over the topology – unidirectional, bidirectional, or any configuration you need.

Complete, GUI-based admin console
Among NoSQL document databases, only Couchbase Server provides a fully integrated GUI-based management console, complete with hundreds of pre-built metrics and easy-to-use tools like push-button scaling, rebalancing, and memory tuning.

Always-on mobile support
Couchbase Mobile is a complete NoSQL solution for mobile application support. It includes Couchbase Lite – an embedded JSON database for devices – and Sync Gateway, a pre-built solution that syncs the device with the cloud. Couchbase Mobile lets you easily support use cases such as in-store personalized apps, point of sales systems, and mobile-optimized digital catalogs.

Conclusion: NoSQL is the right choice for many video game and gaming use cases

If you're running into scalability, performance, or availability challenges with your relational database – and you're looking to speed development and innovation – it's probably time to consider NoSQL technology.

If you're running into scalability, performance, or availability challenges with your relational database – and you're looking to speed development and innovation – it's probably time to consider NoSQL technology.

Many of the game developers and publishers have already made the move, deploying NoSQL to support a growing number of use cases from user profile management personalization to marketplace metadata, player and game data, messaging, and many more.

Getting started is easy

Download the software **and install it in a non-production corner of your IT environment.** Couchbase Server is available for Microsoft Windows, Apple macOS, and leading Linux platforms such as Red Hat, Debian, and CentOS.

Explore our sample travel reservation app**.** A ready-made testing environment is waiting for your input, or your can install it on your own machines and get intimate with the app's source code.

Talk to a Couchbase Solutions Engineer **about your video game or gaming use case.** We have dedicated sales and support offices in North America, Europe, and Asia Pacific, and trusted partners all over the globe.

About Couchbase

Couchbase delivers the database for the Digital Economy. Developers around the world choose Couchbase for its advantages in data model flexibility, elastic scalability, performance, and 24x365 availability to build enterprise web, mobile, and IoT applications. The Couchbase platform includes Couchbase, Couchbase Lite - the first mobile NoSQL database, and Couchbase Sync Gateway. Couchbase is designed for global deployments, with configurable cross datacenter replication to increase data locality and availability. All Couchbase products are open source projects. Couchbase customers include industry leaders like AOL, AT&T, Cisco, Comcast, Concur, Disney, Dixons Carphone, eBay, General Electric, Marriott, Nordstrom, Neiman Marcus, PayPal, Ryanair, Rakuten / Viber, Tesco, Verizon, Wells Fargo, as well as hundreds of other household names. Couchbase investors include Accel Partners, Adams Street Partners, Ignition Partners, Mayfield Fund, North Bridge Venture Partners, Sorenson Capital, and WestSummit Capital.

2440 West El Camino Real | Ste 600
Mountain View, California 94040

1-650-417-7500

www.couchbase.com