Redbooks
ibm.com/redbooks

IBM Cloudant
Database as a Service Advanced Topics

Christopher D. Bienko

Marina Greenstein

Stephen E Holt

Richard T Phillips

Analytics

Information Management

Redpaper

International Technical Support Organization

IBM Cloudant: Database as a Service Advanced Topics

April 2015

REDP-5189-00

Note: Before using this information and the product it supports, read the information in “Notices” on page v.

First Edition (April 2015)

This edition applies to IBM Cloudant.

This document was created or updated on April 17, 2015.

Contents

Notices

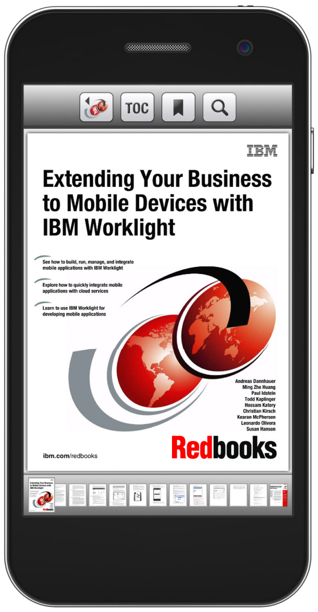
Trademarks

Find and read thousands of IBM Redbooks publications

- Search, bookmark, save and organize favorites
- Get up-to-the-minute Redbooks news and announcements
- Link to the latest Redbooks blogs and videos

Get the latest version of the Redbooks Mobile App

iOS **Download Now** Android


TOC
IBM
Extending Your Business to Mobile Devices with IBM Worklight
Redbooks
ibm.com/redbooks

Promote your business in an IBM Redbooks publication

Place a Sponsorship Promotion in an IBM® Redbooks® publication, featuring your business or solution with a link to your web site.

Qualified IBM Business Partners may place a full page promotion in the most popular Redbooks publications. Imagine the power of being seen by users who download millions of Redbooks publications each year!

ibm.com/Redbooks
About Redbooks → Business Partner Programs

Preface

This IBM® Redpaper™ publication describes advanced topics for IBM Cloudant, a NoSQL JSON document store that is optimized for handling heavy workloads of concurrent reads and writes in the cloud, a workload that is typical of large, fast-growing web and mobile apps. You can use Cloudant as a fully-managed DBaaS running on public cloud platforms like IBM SoftLayer or via an on-premise version called Cloudant Local that you can run yourself on any private, public, or hybrid cloud platform.

This paper is the third in a series of IBM Redbooks® publications on Cloudant. Be sure to read the others: *IBM Cloudant: The Do-More NoSQL Data Layer*, TIPS1187 and *IBM Cloudant: Database-as-a-service Fundamentals*, REDP-5126.

Authors

This paper was produced by a team of specialists from around the world working at the International Technical Support Organization, Poughkeepsie Center.

Christopher D. Bienko covers the Information Management Cloud Solutions portfolio for IBM's World-Wide Technical Sales organization. He is a published author of *Big Data Beyond the Hype: A Guide to Conversations for Today's Data Center* (2014, Zikopoulos, deRoos, Bienko) for McGraw Hill Education. For two years, Christopher has navigated the cloud computing domain, enabling IBM customers and sellers to stay abreast of these rapidly evolving technologies. Before this, Christopher was completing his Bachelor of Computer Science degree from Dalhousie University. Christopher also holds a Bachelor of Science degree from Dalhousie University, with a double major in Biology and English.

Marina Greenstein is an Executive IT Software Specialist with the IBM Core Database Technical Team. She is an IBM Certified Solutions Expert who joined IBM in 1995 with experience in database application architecture and development. During the 19 years Marina has been with IBM, she assisted customers in their migrations from Microsoft SQL Server, Sybase, and Oracle databases to IBM DB2®. She has presented migration methodology at numerous DB2 technical conferences. She is also the author of multiple articles and IBM Redbooks publications about DB2 migration

Stephen E Holt is an advanced technical pre-sales specialist in the New York City area. He has over 20 years of experience working with DB2 ranging from development, deployment, and performance tuning, along with assorted customer facing roles. Stephen now concentrates on leveraging Information Management use of cloud infrastructure for his Wall Street customers.

Richard T Phillips is an IBM Certified Consulting Senior IT Specialist on the IBM New York Metro Technical Team. He has a background in Electrical Engineering, and 32 years of experience in the IT industry, plus three years as a manager of information systems for a laser communication company. Rich has worked on many BI and database projects, providing support in architecture design, data modeling, consulting, and implementation for DW, Analytics, and DBaaS solutions. He has held technical management positions, and has expertise in DB2, IBM Informix® database, Cloudant, and other IM solutions. Rich has worked for IBM for over 14 years.

Thanks to the following people for their contributions to this project:

Lisa Squire Harnett
IBM Analytics, WW Client Success

Philip Monson
IBM Redbooks Software Portfolio Manager

Michael Breslin
Product Manager - Cloud Data Services

Now you can become a published author, too!

Comments welcome

Stay connected to IBM Redbooks

- Look for us on LinkedIn:

  `http://www.linkedin.com/groups?home=&gid=2130806`

- Explore new Redbooks publications, residencies, and workshops with the IBM Redbooks weekly newsletter:

  `https://www.redbooks.ibm.com/Redbooks.nsf/subscribe?OpenForm`

- Stay current on recent Redbooks publications with RSS Feeds:

  `http://www.redbooks.ibm.com/rss.html`

1

IBM Cloudant replication and sharding

This chapter describes IBM Cloudant replication and sharding.

This chapter includes the following sections:

- Cloudant clusters are available, scalable, and fully managed
- Overview of node failures
- The concept of scalability
- Cloudant sync: supporting replication for mobile devices
- Conceptualizing database expansion and rebalancing
- Cloudant clustering and quorum parameters
- Cloudant cluster rebalancing
- Cloudant tuned for high availability and data durability

1.1 Cloudant clusters are available, scalable, and fully managed

One of the key topics described in *IBM Cloudant: Database-as-a-Service Fundamentals*, REDP-5126, was how Cloudant uses NoSQL JSON to persist your data. You might ask *What is the reason for customers making the migration from relational database management systems (RDBMS) to NoSQL?* The reasoning is often that it is complicated. These solutions are often fraught with scalability issues derived from a common limitation for relational databases. Another situation that occurs is that the customer's IT team can be overwhelmed by continually tuning their database to fit their data. Generally, customers are looking for an alternative solution that offers greater flexibility and rapid time-to-market.

Scalability and data locality (data that is closest to the user or their mobile endpoints) are essential to building a successful application for web or mobile devices. Developers building applications for mobile and web-connected devices need to place data as close to their users as possible. This approach lowers latency for reads and writes, which translates into improved performance of the applications and more satisfied customers. It is important to remember that IBM Cloudant offers geo-load balancing as a part of its software stack making this capability immediately available.

Cloudant can manage data over distributed clusters, across both bare-metal and virtualized data centers, to power applications and deliver content to mobile devices worldwide. Developers of mobile and web applications can host their business on a global network of service providers, including the IBM SoftLayer, Rackspace, Microsoft Azure, and Amazon Web Services. Currently, the Cloudant service is hosted in over 35 data centers around the world.

Cloudant supports total flexibility in document and database design including jurisdiction over where that data is hosted. Cloudant is able to scale out these deployments up to millions of databases. You can instantiate individual databases to isolate data on an individual database level. This combination of scaled development and deployment across data centers worldwide and as partitioning of data across individual databases enables the customer to isolate and tightly control how data is persisted on the network.

Cloudant is able to achieve a high level of scalability because of a multi-master architecture that is designed for Dynamo-style horizontal clustering. This master-less framework ensures that Cloudant is able to scale and replicate across multiple systems, racks, data centers, and cloud operations. It also helps ensure that any node can be used to run read/write capabilities during hardware failures within your cluster. In this context, you can think of the process of synchronization between two nodes as a matter of deterministically achieving the same state between nodes.

A replication task can be submitted for one node, such that the contents of node X are replicated (either continuously or as a snapshot in time) to node Y. The ballpark latency for achieving this is in the order of seconds, which means that when you write a document into a database, the primary key index and the _changes feed are atomically updated. The _changes feed maintains a list of all documents that have changed since the last time you checked in. Cloudant uses incremental check-in to achieve this capability. Ultimately, the largest contributing factor for the time it takes a write request to update the _changes feed (triggering responses such as replication and sync) is the physical latency between your data centers.

Cloudant's Database as a Service (DBaaS) is provisioned across multiple servers and maintains redundant copies throughout the cluster for durable and highly available storage. What looks like a logical database to the customer can be data that is partitioned across multiple systems. To maintain the same state across multiple nodes, it is necessary to instantiate replication in both directions. To accomplish this, you need to establish bidirectional replication, which is possible with the Cloudant architecture. With replication underway in both directions, each database will deterministically agree upon the contents and state of each node.

Every Cloudant cluster has a minimum size of three database nodes and a single load balancer node as explained in “Cloudant clustering and quorum parameters” on page 7. For now, it is important understand that it is an architectural requirement.

1.2 Overview of node failures

Multi-master (or master-less) cluster architectures are sometimes mistakenly described as being immune to downtime because the loss of a single node will not take down the entire cluster and each node can operate independently. One caveat to this statement is the load balancer node. If your Cloudant cluster only has a single load balancer node and that node is subject to failure, your database can suffer downtime. Redundant node balancers can be provisioned to provide a higher degree of continuous availability. Figure 1-1 shows that the Cloudant clusters require a minimum of three database nodes and one load balancer. For higher availability and resilience against hardware failure, consider adding redundant load balancer and database nodes consider adding redundant load balancer and database nodes.

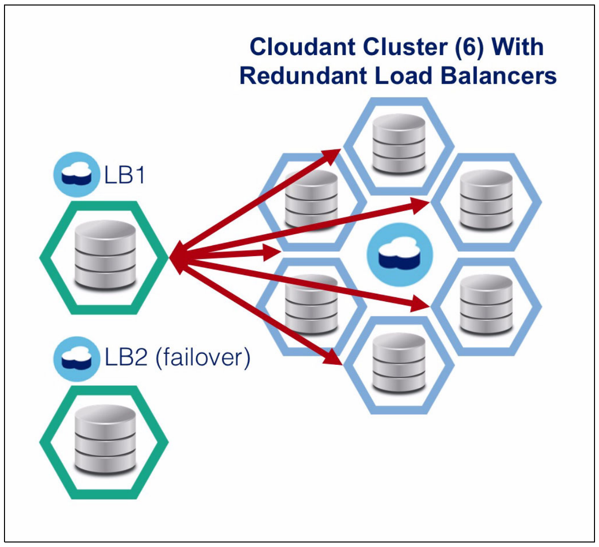


Figure 1-1 Cloudant cluster with redundant load balancers

Cloudant generally recommends that customers provision two load balancer nodes (although only one is required) as a back-up during a failure. There are also concerns around quorum parameters, which might prevent your database from satisfying read or write requests from clients should enough nodes in the database go down.

During a failure, Cloudant operators manage restoration of downed nodes by first returning these failure points to life in maintenance mode. Then the node receives writes and continues to run internal replication tasks to bring it into coordination with the rest of the cluster. However, the node will not respond to coordination requests. The term ‘coordination requests’ refers to the read and write quorum acknowledgements as explained in “Cloudant clustering and quorum parameters” on page 7 in response to client-submitted requests. After the node matches the replication state of the rest of the cluster, the Cloudant operators turn off maintenance mode and the node is available for servicing coordination tasks.

One node being out of date is typically not a problem because redundant nodes will still be able to service requests without the client seeing an impact. However, for Search and MapReduce-built indexes, Cloudant is not able to enforce the same types of quorum constraints. For example, there is no way to know which node contributed to a particular value of an aggregate result from a reduce function. For that reason, Cloudant engineers engage maintenance mode during a node failure. This approach enables Cloudant to preserve and minimize the consistency-related responses that you might achieve with secondary index queries.

1.3 The concept of scalability

Cloudant cluster database nodes can be scaled out horizontally by adding more nodes to the cluster. There is no need to add more software components or change the existing sharding strategy for the database to facilitate new database node additions. However, a rebalance command is run after a new node is added to move shards between the new nodes and redistribute the data according to your database sharding and replication parameters.

Cloudant clusters support replication both between database nodes within a single cluster (cross-cluster) and across entire data center, and support cross-data center replication. They can also span cloud providers. For example, a Cloudant cluster on Amazon Web Services (AWS) is able to replicate with a cluster hosted on IBM SoftLayer, or even with a Cloudant local deployment available on the premises.

1.4 Cloudant sync: supporting replication for mobile devices

Replication is the ability to redundantly store data across the cluster at the entire database level (or through filters) replicate a subset of documents within one database to another. Within Cloudant, database replication can be any of these types:

- Bidirectional: Two databases replicate with each other, such that each database contains a replica of the other's data.
- Continuous: All participating databases agree to maintain a continually up-to-date replica of each other's database. Changes that are made to one instance of the replicated database are propagated to all other replicas to actively maintain synchronization.
- Point-in-time: Databases can be configured to provide a snapshot backup of a target database. Changes that are made to one instance of the replicated database are not propagated until the next manual replication job. This type of back-up is useful as a back-up and restore point if a roll-back becomes necessary.
- Filterable: An entire database might be replicated or a subset of the documents within a database might be replicated instead.
- Durable: Cloudant databases can pick up and resume where they left off if a replication job is interrupted (due to a network outage, for example).

Figure 1-2 shows bidirectional synchronization between Cloudant clusters. Replication jobs between Cluster A to Cluster B and vice versa must be defined to establish this type of mega-cluster. Cloudant Sync libraries extend this synchronization to occasionally connected mobile devices as well.

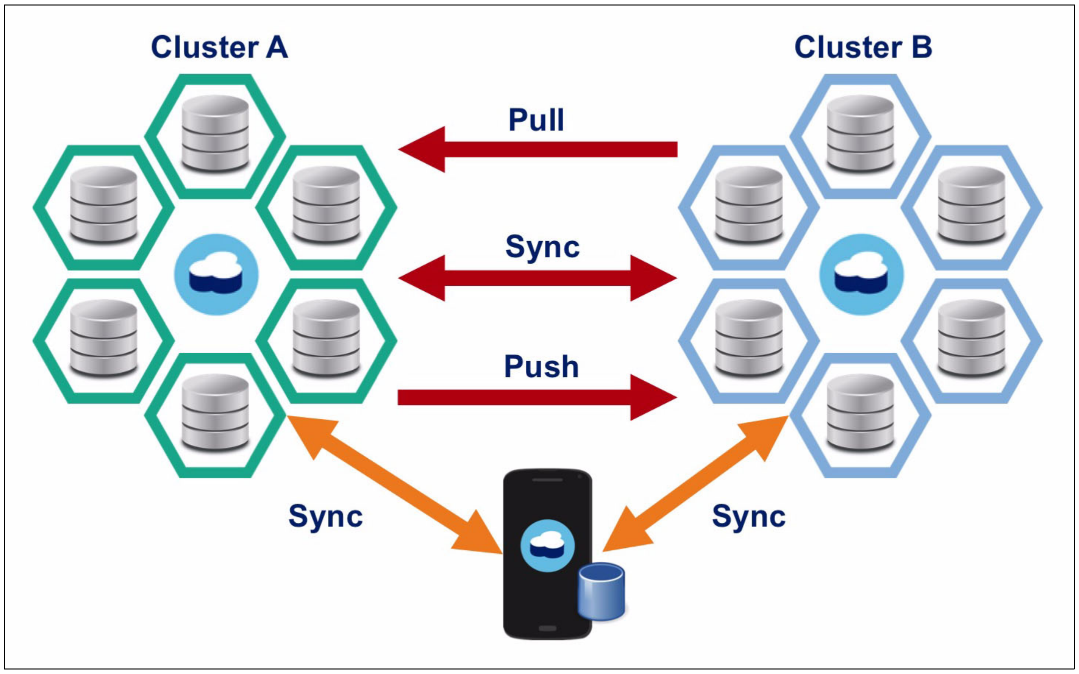


Figure 1-2 Bidirectional synchronization between Cloudant clusters

For developers who are building applications that are mobile-centric, Cloudant Sync delivers local read, write, and index query access to data on your mobile devices. The results are incredibly performant applications that only need to rely on your local device to do the computations, instead of waiting for server-side calls and responses.

SQLite is a relational database that is ubiquitous across nearly every mobile phone and tablet. It has a data layer that is familiar to many of today's application developers and is a proven technology. The SQLite implementation in Cloudant abstracts the migration process (by using Cloudant Sync libraries) from the perspective of users and developers. SQLite is incredibly well-proven in production (familiar to both users and developers) and is available on all major platforms including mobile operating systems. SQLite easily builds indexes with fast indexing of data from JSON. Each document in the database can be indexed in any number of ways, allowing for flexible querying semantics that are more labor-intensive to achieve with CouchDB's MapReduce views.

Cloudant Sync is now available across two client libraries: One for iOS and one for Android. Both are able to synchronize data directly with either Cloudant or CouchDB over HTTP(S). No additional runtimes or software is needed in support of these libraries. On iOS devices, their their replication is based on TouchDB (which is available on iOS already). For Android phones, Cloudant uses custom own code to efficiently use the Java platform. Both libraries are wrapped in a native application-friendly API.

Cloudant Sync works well in use cases where you have high levels of write and critically where your application and device rely on connectivity. Imagine a shipping truck leaving a warehouse to make its deliveries. The more processing the operator can do on his tablet, the better the experience for both the driver and the customers along the route. You only want to sync those moments that are of interest, such as an airbag being deployed in the truck.

Cloudant can create very small JSON documents. These documents trigger other applications on a short turn-around, such as piping telemetrics data off to another analytics or reporting tool.

Cloudant Sync libraries handle the mapping of JSON documents to SQLite relational data store on your mobile devices, without intervention by the developer. Cloudant SQLite-based Sync libraries follow a multi-version concurrency control data model to build indexes locally on your mobile device. Replication is still handled by Cloudant, but during connectivity outages your local reads, writes, and index queries are managed by the SQLite data layer.

Data is stored using storage local to your mobile device, with the native SQLite client library handling the synchronization between your device and the Cloudant database cluster. If that connectivity is lost, you retain the ability to perform local reads, writes, and queries against indexes. You can continue to create, update, and delete database documents, all as though you were connected to the Cloudant database cluster.

When your device regains connection to the network, the Sync library handles pushing that data up to the Cloudant cluster. After your device's data enters the cluster, your edits are propagated through the traditional Cloudant replication and synchronization mechanisms to the rest of the nodes (according to the quorum parameters you have defined for your database). As before, no additional libraries are required for Cloudant Sync to interact with your database cluster. All requests are passed over HTTP(S) through the Cloudant RESTful API.

Cloudant can also replicate to mobile endpoints by using the Cloudant Sync libraries. This configuration allows you to push databases from your Cloudant cluster on the cloud directly to your mobile device for local read and write operations. This even works when connectivity is lost with the network. Mobile replication and synchronization libraries handle the process of restoring a consistent view of the database across the entire Cloudant cluster (including your local database copy) upon restoration of connectivity.

1.5 Conceptualizing database expansion and rebalancing

This section describes how data is replicated and synchronized across the nodes of your cluster. Replication can occur between separate Cloudant clusters or from within your own cluster (with itself). You can also replicate to other database endpoints, if they have the same replication protocol as your Cloudant database. For example, CouchDB has the same protocol as Cloudant, and as such Cloudant customers have in the past frequently replicated their database with CouchDB if they were seeking on-premises replicas of their data sets. Similarly, Cloudant Local allows Cloudant DBaaS customers to replicate their databases from cloud to ground, or burst their data off-premises to the cloud. Both examples with Cloudant Local provide point-in-time and continuous synchronization options.

Understand that the terms ‘shards’ and ‘partitions’ are interchangeable in this context. Sharding is the result of applying a consistent hash function to a JSON document (and its contents) and mapping that to a particular key space (called shard or partition). It is an automated process that is based on the CRC32 hash function applied to a combination of your document's ID and contents.

1.6 Cloudant clustering and quorum parameters

It is useful at this stage to introduce the four parameters that govern a logical Cloudant database, which are Q, N, and W:

- Q is the number of individual database partitions over which your data is distributed.

  If a database is split into Q=4 shards, then any document submitted to the database will need to pass through a consistent hashing function. Each document is ensured to fall into the key-space of that hash function (it can be visualized as a ring), where each key-space represents a shard. If you have a three node cluster (regardless of the Q value), every document in that cluster exists on each of the nodes. Q determines the number of copies that are required to exist across each cluster. Only under circumstances where the number of nodes belonging to a cluster exceeds the sharding parameter will there be the potential for certain nodes to host more replicas of a shard than others. Selecting a good consistent hashing function generates an even distribution of the number of documents per shard across your key-space.

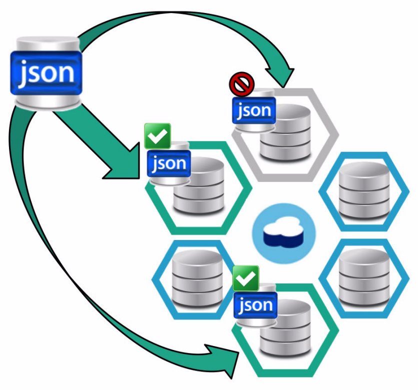
- N is the number of copies (replicas) of every data shard that will be generated across your cluster of nodes to ensure high availability and redundancy.

  Typically N=3 for Cloudant multi-tenant customers. In this scenario where N=3 and Q=4, there would be 12 (N x Q = 3 x 4) shards distributed across your cluster.

- W is the quorum parameter that must be satisfied in order for a write request to be considered durably committed to disk.

  A client submitting a write operation to a three node cluster submits the request to nodes one, two, and three. However, only a majority of these nodes need to acknowledge the request (in a response back to the client) for that write to be successful. For the example in Figure 1-3 on page 8, an operator would consider the write request successful if two out of the three total nodes reported back a successful response code. If N=3 for this cluster, the third node that did not report back a write (either because it was the last to respond, or due to some error) an additional replica of the data being written will eventually be replicated to that node. This scenario describes the Cloudant model of *eventual consistency*. This premise is that, given enough time, the replication state of the cluster and all views into the data will be consistent across the cluster. One last consideration on writes is that Cloudant does not have the notion of a *roll-back*. Instead, write requests that do not satisfy the simple majority quorum requirements (for example, writing successfully to only one-third of the nodes) return a 202 status code. This event signals to the application developer that the write did not meet the quorum requirements, but was still committed to one of the nodes. You cannot roll back failed writes, but you can design program logic to handle such cases after the fact.

Figure 1-3 shows the parameters and counts for the example.



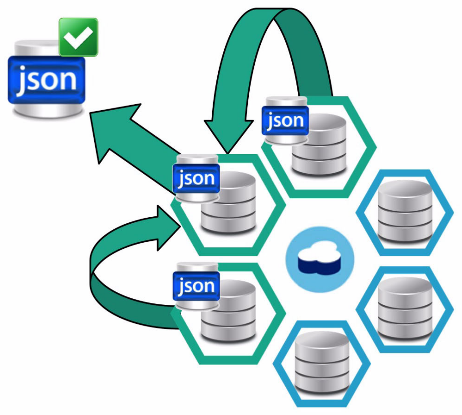
Parameters	Count	PUT Request
Nodes in Cluster	6	Minimum for any cluster is 3.
N	3	Every shard is stored on 3 different nodes (for redundancy).
Q	4	Every database is sharded into 4 keys.
W	2	Majority (2/3) must report successful write for a durable commit.

Figure 1-3 Parameters and counts

This example conceptualizes a *write* request from a client to a Cloudant cluster of six nodes. In this example, the write quorum is W=2 and every document is sharded across three nodes (N=3). Although one of the three nodes did not return a successful write response to the client, the two-thirds majority is sufficient to ensure a durable write to disk.

- R is the quorum parameter that denotes the number of nodes that must return to the requesting client copies of data that *agree* (match).

  Similar to the W parameter, R is by default a simple majority of the total number of nodes in the cluster. If an insufficient number of nodes (fewer than the simple majority of two if N=3) are able to return a replica of the requested data, the read request fails. In general, the consistency that can be expected from secondary indexes is much lower than what would be expected from individual document lookups based on a document's _id field. For the example in Figure 1-4, all three nodes are able to produce versions of the client-requested JSON document that *agree*, and therefore the read request is satisfied.



Parameters	Count	GET Request
Nodes in Cluster	6	Minimum for any cluster is 3.
N	3	Every shard is stored on 3 different nodes (for redundancy).
Q	4	Every database is sharded into 4 keys.
R	2	Majority (2/3) must agree on returned doc for a successful read.

Figure 1-4 Conceptualizing a read request from a client to a Cloudant cluster of six nodes

In the example in Figure 1-4, the read quorum R=3 and every document is sharded across three nodes. The node coordinating the request must receive three documents that agree and match the client-requested document (having the same content and revision value). One copy of the data happens to be on the coordinating node and two replicas are shared by other host nodes in the cluster. If these three replicas agree, quorum is reached and the document is returned to the client.

As part of the value behind the Cloudant DBaaS, experienced technical engineers will select the optimal values of N, Q, R, W, and other parameters for you. Because changing the replication factor and sharding key are non-trivial tasks after the database is up and running, it is essential to choose the correct values before working with your database. In many ways, selecting the best value is as much an art form as it is a science. The value of the Cloudant fully managed approach to database provisioning, configuration, and maintenance is quickly apparent.

1.7 Cloudant cluster rebalancing

Cloudant has no notion of a master node, unlike other solution providers who use a master-slave architecture. Cloudant follows a multi-master (*master-less*) cluster architecture. If a client wants to write a document to a Cloudant database, that write request can land anywhere in the cluster. The request does not necessarily need to hit any of the nodes that host the document in question.

The node that receives the request (defined as the coordinating node) performs the following actions:

- Consults the partition table
- Performs a look-up to determine which nodes are host to the requested document
- Simultaneously submits requests to store it on three hosts (the default replication parameter)

After quorum is reached (in other words, all three, or a majority of nodes, have reported a successful write to disk), a response is delivered to the client to signify the request has been successfully carried out.

Figure 1-5 shows that a document submitted by a client for commit to disk is passed through a load balancer, which then selects the coordinator node for the write request within the Cloudant cluster. According to the replication factor set by the database, the coordinator node ensures that the document is replicated to N database nodes within the Cloudant cluster for redundancy.

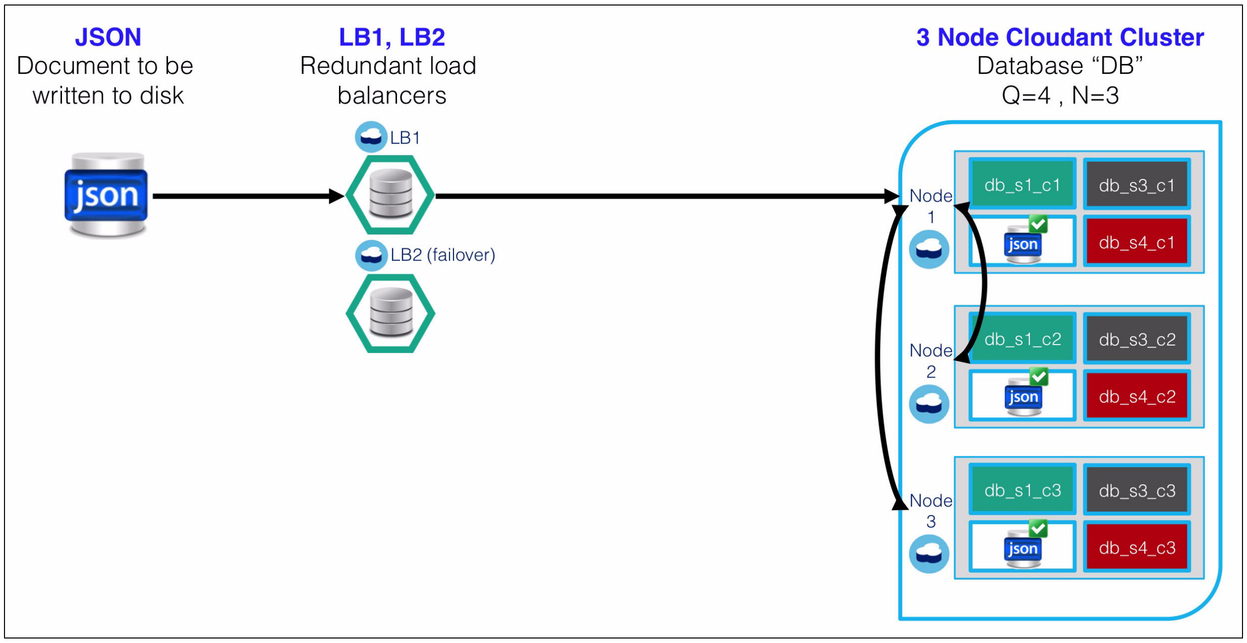


Figure 1-5 Using redundant log balancers

Determining where to distribute the document shards is a purely functional computation of the document ID. There is no look-up table that says "for this document ID 123, host on partition X". Instead, it is determined as a pure hash function of document ID to a segment in the hash key-space. The only state that Cloudant needs to maintain is the mapping of partitions (shards) to nodes. This information is persisted in a separate database that is continuously replicated to every node in the cluster.

The process of rebalancing the cluster when adding a node does not require splitting partitions. Instead, as new nodes are added to a cluster, the data layer employs a module inside the cluster that computes a set of moves to yield a balanced distribution of shard files.

The movement process itself is done operationally by Cloudant. New shards are copied while old shards remain online for performing look-ups. When the process of replicating shards is complete, Cloudant engineers change the shard mapping to point to the newly replicated shards.

Indexes: Indexes need to be copied first (before the main database files) because if Cloudant detects an index that is out of step with the documents it is indexing, that index is considered to be corrupted and must be rebuilt. While the process of rebalancing your cluster can be a time-consuming effort (particularly if you are working with a sizeable data set), no downtime is suffered during the rebalancing procedures. This approach presents a significant advantage over relational and other NoSQL database solutions, which would mandate downtime to perform rebalancing and would not offer a fully managed service to handle these operations for you.

To visualize this process, consider the illustrations of the process of adding a database to your existing Cloudant cluster. This is not the same as adding a Cloudant node and rebalancing the entire cluster. The outcome of adding this new database is that the data stored within it needs to be sharded and replicated across the nodes belonging to the cluster.

Beginning with Figure 1-6, a sharding factor of 4 is set in the cluster (Q = 4), and the Cloudant database node is partitioned into four shard maps (Parts 1 - 4).

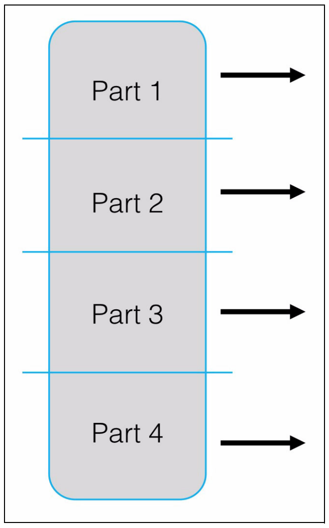


Figure 1-6   Cloudant database with a sharding factor of Q = 4 visualized in part 1 - 4

Figure 1-7 conceptualizes the process of sharding the database as partitioning it into four chunks of data: db_s1, db_s2, and so on. These are logical mappings to where the data is on disk, according to the CRC32 hash function used to generate the shard map.

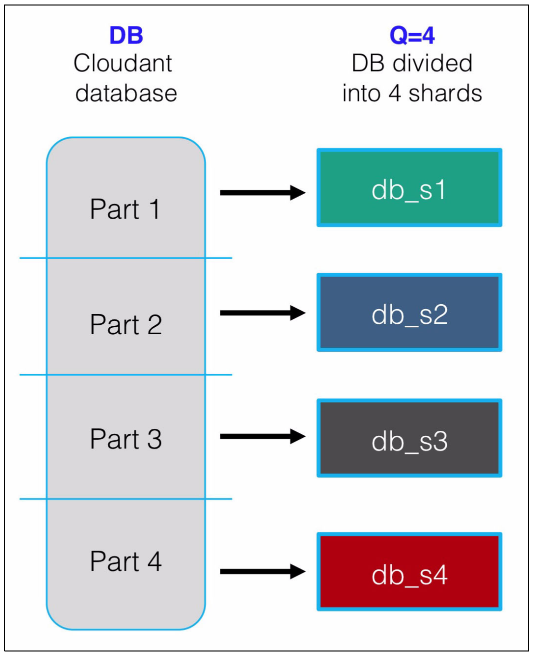


Figure 1-7   Every partition (or shard) within the database corresponds to a range mapped by a CRC32 hash function

In Figure 1-7 on page 11, every document within a database falls within one (or more, depending on the number of shards) of these shard mappings.

In this example, the Cloudant cluster has a replication factor of three (N = 3), meaning that every piece of data is stored redundantly across the cluster three times. In Figure 1-8, there are three replica copies of each shard. Therefore, the total number of shards in your Cloudant database is equal to the product of Q and N. If the sharding factor is four (Q = 4) and the replication factor is three (N = 3), then there are 3 x 4 = 12 shards across the database.

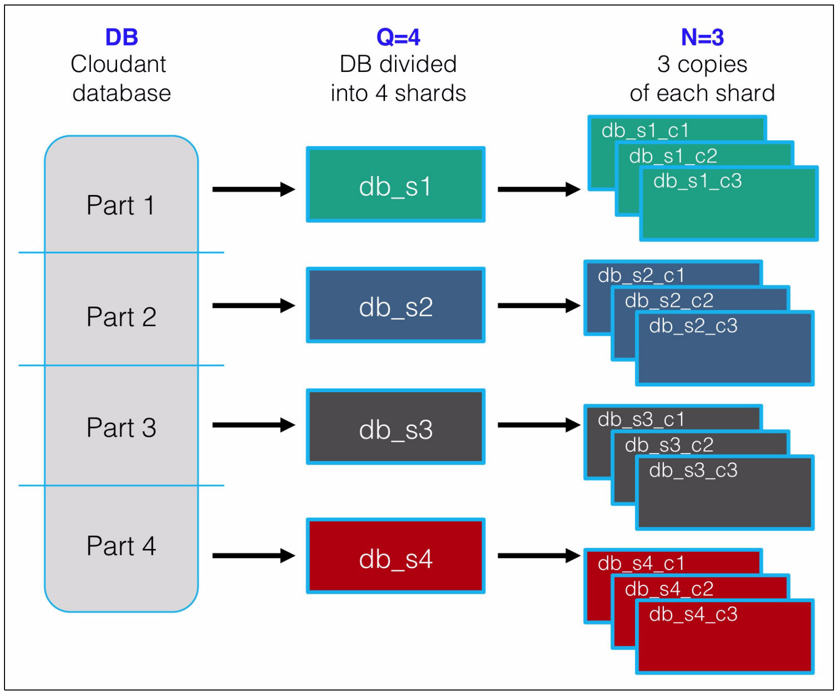


Figure 1-8 A sample database with a replication factor of N=3

In Figure 1-8, every document present within the database is being stored in triplicate. The total number of shards in this case is equal to the product of N and Q, or 3 x 4 = 12 total shards.

Figure 1-9 concludes the example by illustrating the distribution of shards across the three nodes of the example Cloudant database. One copy of every shard is stored on each node and every node persists four database shards.

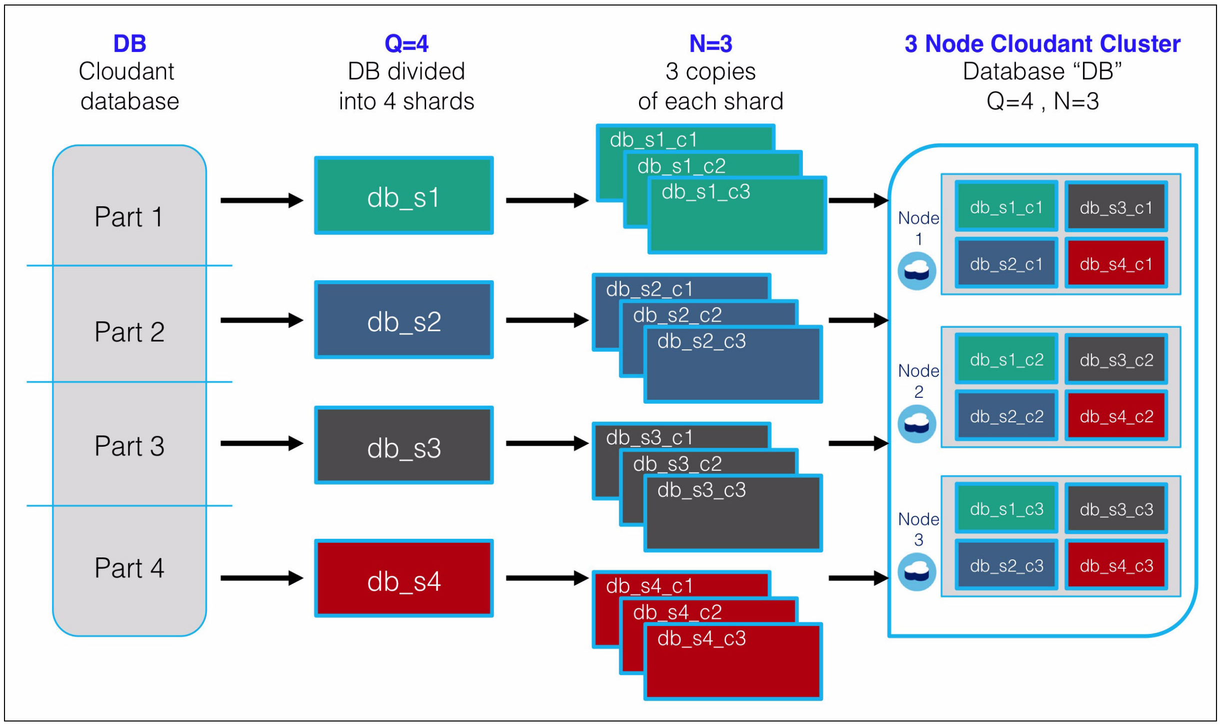


Figure 1-9 Shards that are evenly distributed across the three database nodes

In Figure 1-9 the shards are evenly distributed across the three database nodes, such that every node persists one replica of every database shard.

Distribution of labor on a Cloudant cluster is equal opportunity. That is, Cloudant instructs all available nodes to work on every job submitted to the cluster. This design decision emerged from the need to ensure that all indexes remain available during a hardware failure. Cloudant passes any client-submitted request to all the nodes on the cluster, which run in parallel. For example, if a database has four partitions and three replicas per partition, there are at any time 12 workers that are indexing that database. The work is being done in triplicate.

There is some redundancy in that each node is operating on their own independent copy of the data. The concern for Cloudant engineers is that if they defer or de-prioritize construction of redundant copies of an index, then the database runs the risk of having applications made unavailable from the loss of a singular index replica. The coordinator follows a similar strategy in response to index queries submitted to the cluster. Unlike the quorum parameters for reads and writes that must be satisfied by a simple majority, the coordinator does not assemble answers from each response to an index query. Instead, the first answer that is received by the coordinator is accepted as the correct response.

If your data is distributed over three nodes with a replication factor of four (N = 3, Q = 4) and a query is submitted to the secondary index, the coordinator passes along that request to all 12 replicas. The first index query response to be received by the coordinator is returned to the client making the request. All subsequent responses that follow from the other nodes are ignored. Cloudant sends a *cancel* response to those nodes that have not yet submitted a response when the coordinator receives its answer. Using this approach cancels streaming jobs being run on those nodes, but will not cancel background indexing tasks.

1.8 Cloudant tuned for high availability and data durability

Before describing the nuances of shard key values, this section defines the default (minimum) values for sharding with Cloudant databases. For multi-tenant customers on Cloudant DBaaS, Q = 4 is the minimum sharding factor for these accounts. Multi-tenant customers cannot adjust their shard key value. Cloudant Dedicated (DBaaS) and Cloudant Local customers must have a minimum of Q=8 shards. These customers can tune their database for a higher sharding factor, depending on their workloads and use cases.

The number of shards into which a database is logically partitioned must be determined before creating your database. Changing the value of Q requires rebalancing of all shards across each database associated with your Cloudant cluster. One of the key considerations when selecting a Q value is that indexes are constructed locally to the individual partition files. This becomes very nuanced when you understand that only one copy of each partition is consulted to satisfy a single view query. To be more specific, the first copy is given to the coordinator in response to the query request

Therefore, there are two factors you are trying to combat when selecting for Q:

- Keep the required throughput on a partition (the number of writes the partition must juggle) low enough so that the index can be computed fast enough to give a response with low latency.
- Minimize the amount of network traffic generated across your cluster. If you are building an index that you expect will be queried against thousands of times per second, you need to consider that each of those queries will be sent to every partition. The shard file size should be 10 GB or less, with no more than a maximum of 100 GB of data associated per shard. Exceeding these parameters will affect the performance of compaction within the database.

Determining an optimal shard key is as much an art as it is a science. One key advantage of choosing Cloudant as your data layer is that, as part of the DBaaS, Cloudant engineers advise customers on the optimal shard key value for their cluster. Choosing an inappropriate sharding key from the beginning can potentially saddle your applications and database with significant performance issues. This is a common issue for MongoDB customers, who need to choose a sharding key themselves if they choose to provision their own instance.

When deciding on a shard key, there are two general rules to consider:

- If an application is read-heavy, specify a lower number of shards (small Q value).
- If an application is write-heavy, specify a higher number of shards (large Q value).

The higher the shard count, the smaller the data set size that is mapped to each partition, resulting in more efficient inserts and index updates. If there are multiple writes submitted by many users to the same shard (partition), then the time spent waiting for each job to complete is shortened if the shard count (Q) is higher. Increasing the number of partitions significantly increases the load that you place on both the CPUs supporting your database and the network itself. If an application is read-heavy, the database should favor a lower number of shards.

Sharding strategy uses consistent hashing to evenly distribute documents into partitions. For any individual document lookup, Cloudant can go directly to the partition that hosts the document. Any node can compute the partition that hosts a particular document by the ID supplied by the client. If a user wants to perform lookups based on attributes other than _id, the system does not have a priori knowledge about which partitions in the cluster can contribute to that response.

Another consideration is that Cloudant indexes are built locally for each shard. Every time a document is updated or inserted into the database, the index must be incrementally updated. The "incremental" component (which relies on Cloudant's built-in incremental MapReduce) ensures that the index can be rebuilt without needing to reprocess and rescan all data that are associated with that index. Increasing the number of shards decreases the amount of data that each shard must index (and reindex, if documents are added or altered). Figure 1-10 shows that if an application is write-intensive, set the shard count (Q) to a large value to parallelize work more efficiently.

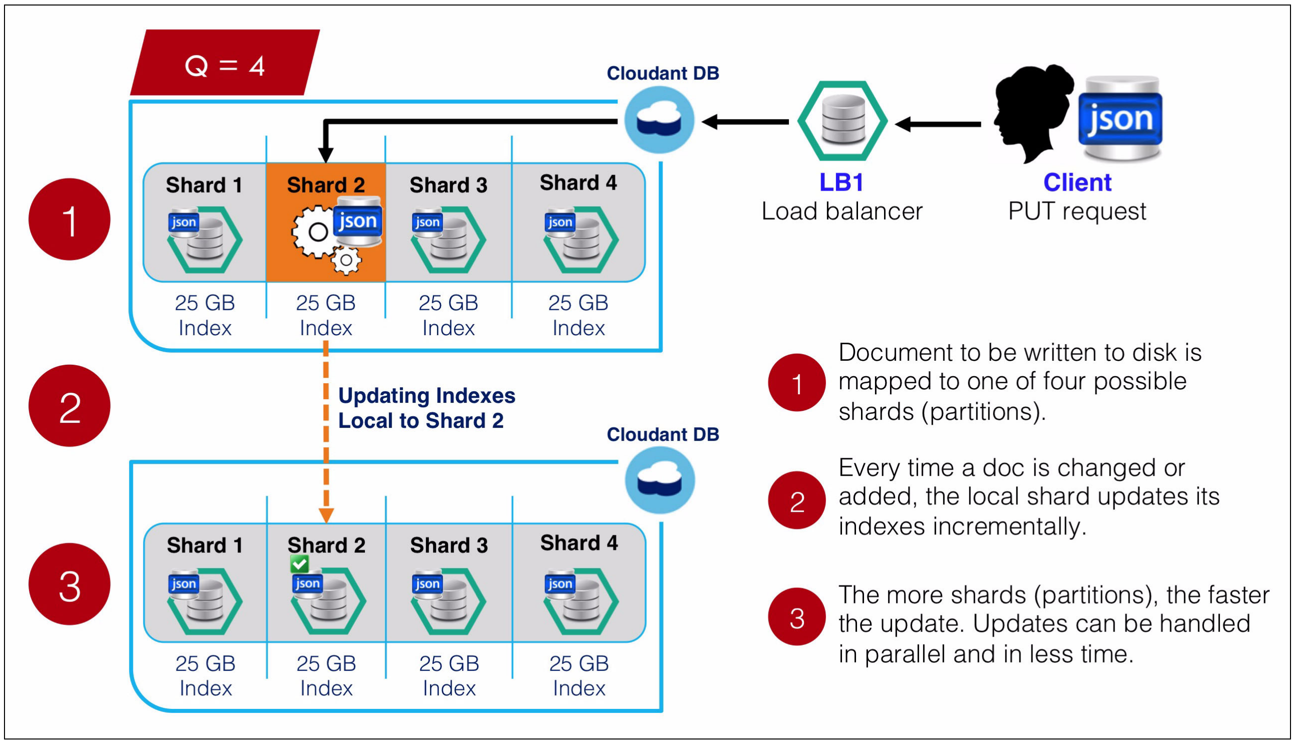

Q = 4
Cloudant DB
LB1
Load balancer
json
Client
PUT request
1
Shard 1
Shard 2
Shard 3
Shard 4
json
25 GB Index
25 GB Index
25 GB Index
25 GB Index
2
Updating Indexes Local to Shard 2
Cloudant DB
3
Shard 1
Shard 2
Shard 3
Shard 4
25 GB Index
25 GB Index
25 GB Index
25 GB Index
1 Document to be written to disk is mapped to one of four possible shards (partitions).
2 Every time a doc is changed or added, the local shard updates its indexes incrementally.
3 The more shards (partitions), the faster the update. Updates can be handled in parallel and in less time.

Figure 1-10 Example with shard count (Q) set to a large value

All of the indexing that Cloudant does today is done locally to a specific partition. When you build a MapReduce (secondary) index or a Search index, each of those are built in parallel. Each partition builds its own secondary index, and at query time Cloudant merges the results from the individual partitions, sorts them as needed, and delivers the cluster-wide response out to the client.

This approach is good from a scalability standpoint. Cluster indexing performance can scale in support of increased throughput requirements. However, Cloudant is heavily dependent on sufficient cross-network bandwidth to support the network traffic that is required for receiving data from every partition. This situation is true even if that data is "I have no results relevant to your query request". The key take-away is that Cloudant is able to scale up in support of good throughput, and the SoftLayer Infrastructure as a Service (IaaS) is well-suited for this task.

Increasing the number of shards (Figure 1-10) has the added benefit of parallelization of reindexing jobs across smaller amounts of data, improving performance. For this reason, you can think of Q as the degree of parallelism available to the Cloudant cluster. If your application is write-heavy, you want a higher number of shards to balance out and distribute the work of running write operations. In general, the shard number should be a multiple of the number of nodes in the cluster to distribute the number of shards per node evenly.

It is important that you define only as many shards as you have spindles and CPUs within your cluster. The more shards that you have, the greater the demand on CPU resources. The larger the value of Q, the larger the request from a client will be. For example, a single document might span multiple shards (Figure 1-11). Extra processing time is needed to stitch these shards together into the requested document. There are performance implications associated with the time needed to compile and stitch together the shards.

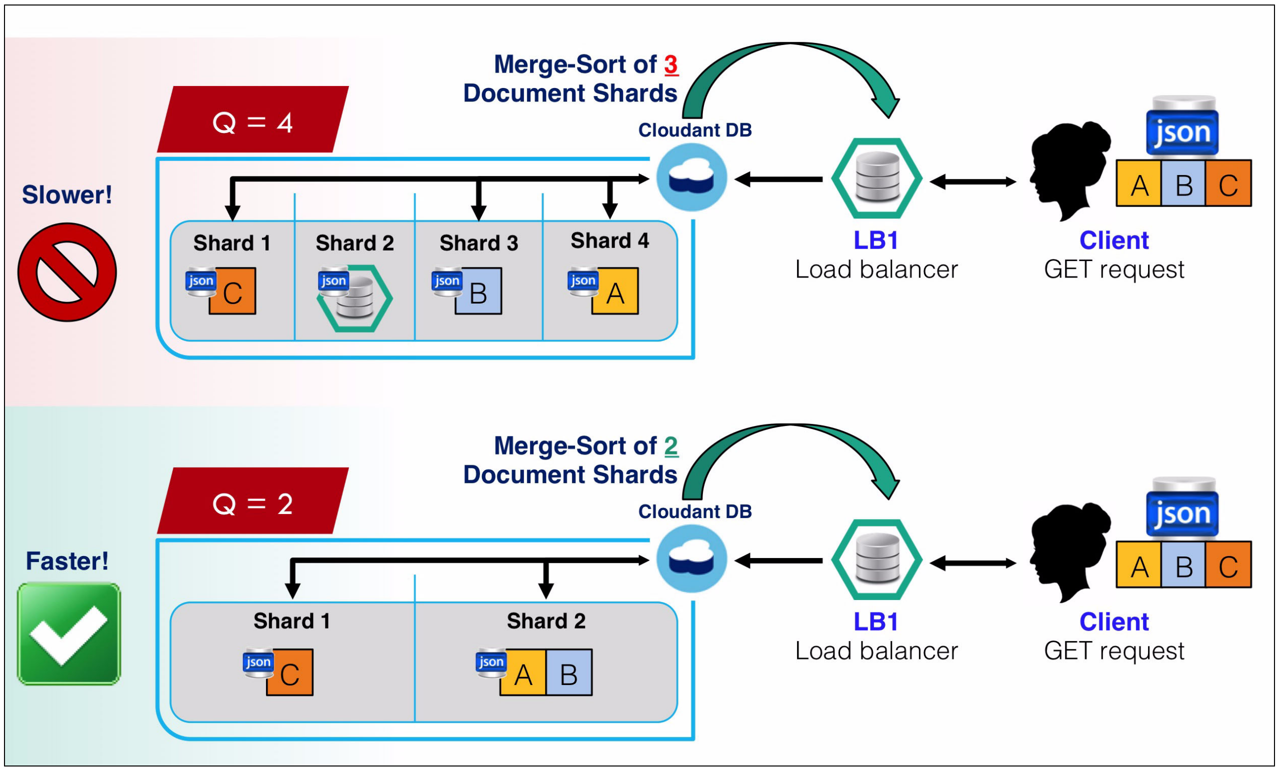


Figure 1-11 Example of a read-intensive application

As shown in Figure 1-11, if it is anticipated that an application will be read-intensive, it is recommended that the shard count (Q) be set to a small value. This approach reduces the number of merge-sort tasks that are required to compile documents for complex queries. This generally only applies to complex queries. Look-ups based on document _id carry a negligible performance effect.

Consider a database of documents sorted on the field 'name' with an additional field 'salary'. A client requests to find all employees that have a salary less than $100,000 a year, sorted alphabetically. If Q = 10 (shards), then the cluster needs to run 9 merge-sorts. If Q = 2, then only a single merge-sort is required for the same result. Also, keep in mind that all of these operations need to be transferred over the network, which can increase latency if data centers are not located near each other or the client.

2

Cloudant multiversion concurrency control (MVCC) and document conflict resolution

Multiversion concurrency control (MVCC) is how Cloudant databases ensure that all of the nodes in a database's cluster contain only the newest version of a data. Replication is a procedure that synchronizes the state of two databases so that their contents are identical.

This chapter includes the following section:

- Resolving conflict through document history

2.1 Resolving conflict through document history

With distributed systems, there is always the potential that you and another client might be introducing different updates to the same document at the same time. A data layer that supports writing to a database on a device and server simultaneously or across clusters, increases the likelihood of document conflict scenarios. Your database solution needs to be able to track all of the edits being made to an individual document and have mechanisms to reconcile these changes and persist them in some logical order.

Cloudant uses MVCC to resolve versioning conflicts that result from create, read, update, and delete operations and synchronization tasks. Every Cloudant JSON document contains both an _id (document ID) and _rev (revision) field. The _rev field is a combination of both an integer and a hash string, which represent:

- ► The number of edits a document has undergone. A newly committed document has an integer of 1 appended to the front of hash string.
- ► A string resulting from a CRC32 hash function applied to the document's contents, and the location relative to other documents in the database.

For example, document A with _rev = 1-32ef42a… has not been updated from its original state. Document B with _rev = 4-3a45bc… has been updated three times, with content and position within the B-tree that is different from document A. The _rev value is automatically incremented when a document is updated. Recall that while a developer can set _id to any value they want (if it is unique within the database), a user cannot set the value of _rev manually.

These hash values are injected at commit time. The edit history in this case is important. It is necessary that these concepts are deeply embedded into the core of the database architecture to ensure that replications are performed correctly and that consistency is maintained. With Cloudant, the revision and hash history of every document is persisted internally as a “B-tree” and continues to be stored alongside the document as it is replicated across servers. Without this revision history, Cloudant would be unable to detect and repair divergences within a cluster.

Figure 2-1 shows Cloudant databases cloudantDB and mobileDB with the same version of document “foo” present at the onset of this replication example.

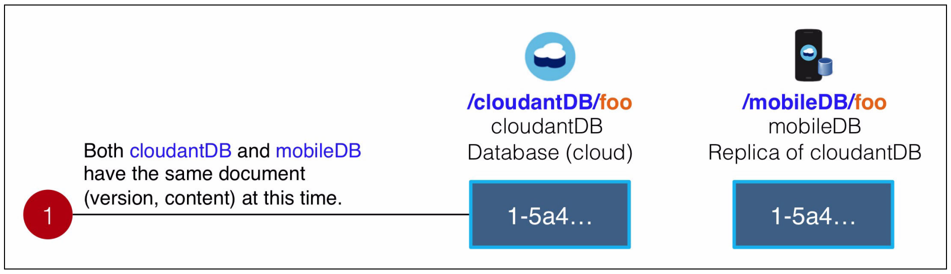


Figure 2-1 Example Cloudant databases with the same document

In Figure 2-1 there are two databases “foo” existing in different parts of the Cloudant database cluster:

- ► One replica in node cloudantDB (part of the Cloudant cluster on the cloud)
- ► The other replica in a mobileDB endpoint (on a user's mobile phone)

To update or delete a document, the _rev value of the most recent version of said document (that exists in the database) must be supplied in the PUT request to the API. If the supplied _rev does not match or is outdated, an Error 409 HTML response is returned. The default behavior of the API today is that if it detects a newer version of a document when trying to update, it rejects the update and alerts the user that a newer version exists.

Minimizing write-contention is rule number one for avoiding conflicts. By doing so, Cloudant minimizes the number of conflicts that the user needs to manage. If a request to update a document is found immediately to be in conflict with an update that already exists on the cluster, the write (and thus the conflict) can be prevented before it happens. This prevents concurrent updates from overwriting updates made by other users that might have shared access to the same data.

Figure 2-2 shows a client submitting a PUT request to update document "foo" on cloudantDB. This revision is replicated to mobileDB automatically.

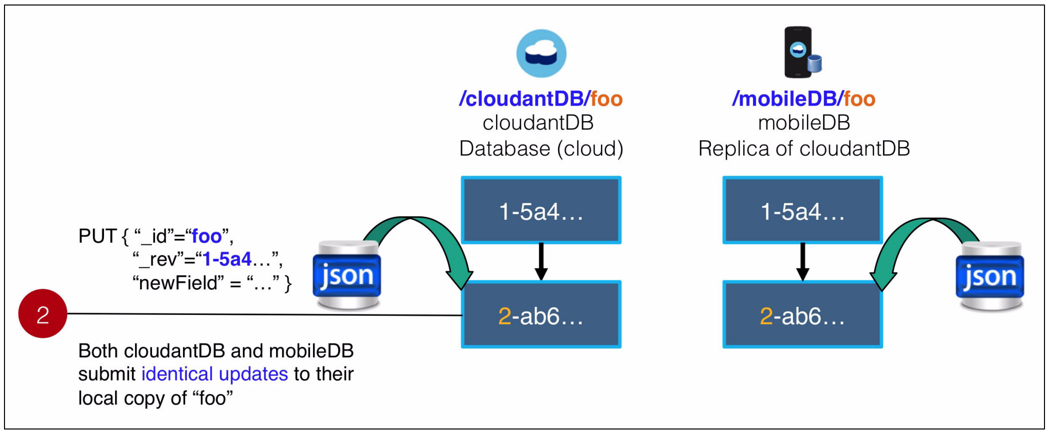


Figure 2-2 Example of a client submitting a PUT request to update document "foo" on cloudantDB

The update to document "foo" in Figure 2-2 is only submitted to the cloudantDB endpoint. However, the replicator libraries do the following steps:

1. Ask cloudantDB for the most recent version of the document from mobileDB
2. MobileDB reciprocates with a request from the most recent version from cloudantDB that need to be replicated to its local copy

This process ensures that the updated document is replicated across and present within both the cloudantDB and mobileDB databases.

Figure 2-3 shows two clients submitting a document update to revision 2-ab6 at the same time on cloudantDB. Both revisions (submitted at nearly identical points in time) need to be persisted by the cloudantDB database until the conflict can be reconciled at the application layer. The mobileDB database begins the process of replicating the updates made to cloudantDB. Given sufficient time, it replicates both 3-f57 and 3-085 versions. However, before it can replicate both revisions, connectivity is lost with the network.

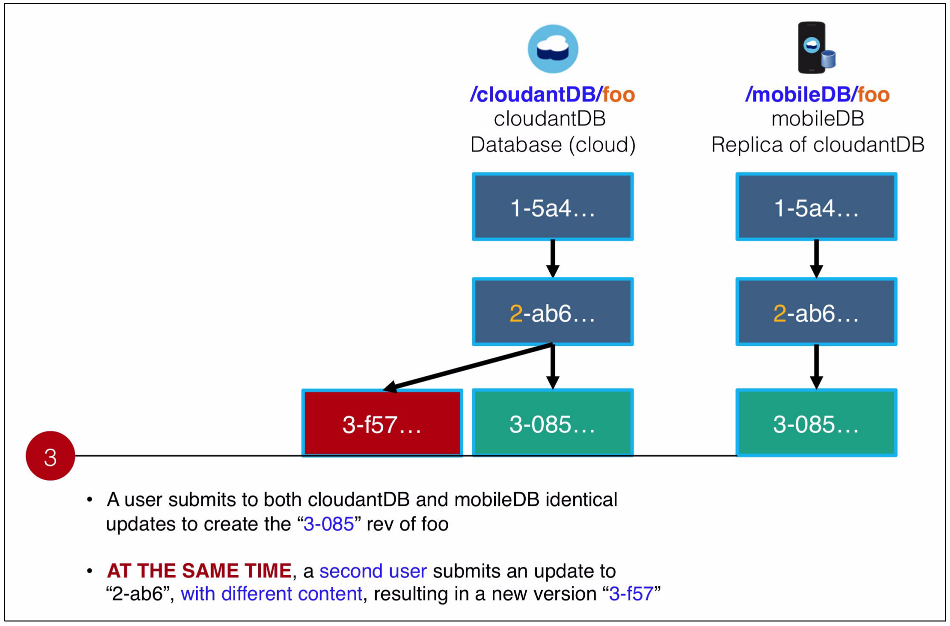

/cloudantDB/foo
cloudantDB
Database (cloud)
/mobileDB/foo
mobileDB
Replica of cloudantDB
1-5a4…
1-5a4…
2-ab6…
2-ab6…
3-f57…
3-085…
3-085…
3
• A user submits to both cloudantDB and mobileDB identical updates to create the "3-085" rev of foo
• AT THE SAME TIME, a second user submits an update to "2-ab6", with different content, resulting in a new version "3-f57"

Figure 2-3 Example of two users submitting updates to the same item

The synchronization and replication procedure between cloudantDB and mobileDB is complicated when multiple users submit revisions to revision 2-ab6 on cloudantDB at the same time. In Figure 2-3, two different sets of updates by two unique users are submitted to cloudantDB at nearly identical points in time. As such, the two versions are committed to disk before Cloudant is able to detect that both PUT requests are for the same version of the document. What results is a versioning conflict that is shown in Figure 2-3 by the branch formed in the document's history B-tree.

Versioning and document conflicts will inevitably rise with any large distributed system. While Cloudant employs numerous strategies to avoid document update conflicts, there must always be a contingency for resolving conflicts. An example requirement is that all updates to documents include the document ID and the latest revision ID of said document.

The Cloudant solution is to never delete conflicts. The approach is conflicting documents are both persisted inside the database and are left to the application layer to deterministically resolve. Requiring updates to documents to supply both the document ID and revision value certainly helps reduce the number of conflicts that arise. However, immutable documents (the approach to creating documents in place of doing updates to existing documents) are a best practice for minimizing conflicts that Cloudant will often recommend to customers.

With conflicting documents, the cluster coordinator detects that one of the document hosts has responded to a request with an older version of the document. It does so by detecting divergence in the document history. It can then determine (by using the conflicting document's hash history) how the conflicted document fits into the complete revision tree, allowing a developer to later programmatically commit that document back.

Concurrently, the mobileDB database is actively replicating the changes committed to the cloudantDB database. The 3-085 revision of “foo” is replicated without error to mobileDB. However, Figure 2-4 shows the connectivity between the Cloudant cluster on the network and the mobile device is lost before the 3-f57 revision in cloudantDB can be replicated into the mobileDB database.

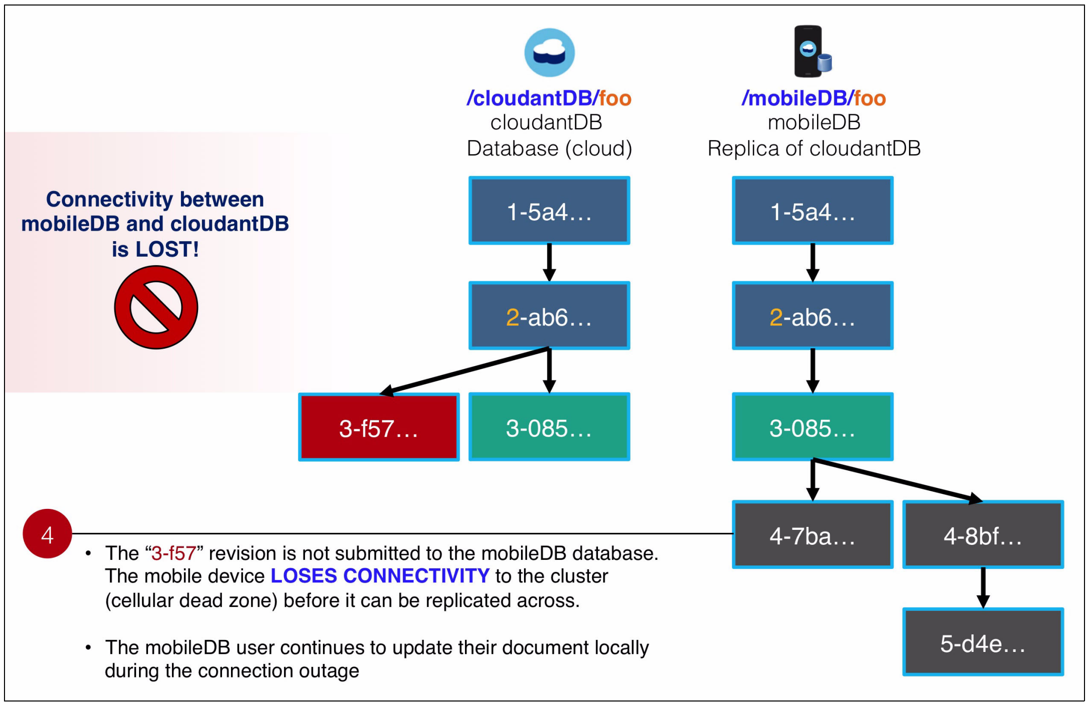

/cloudantDB/foo
cloudantDB
Database (cloud)
/mobileDB/foo
mobileDB
Replica of cloudantDB
Connectivity between mobileDB and cloudantDB is LOST!
1-5a4…
2-ab6…
3-f57…
3-085…
1-5a4…
2-ab6…
3-085…
4-7ba…
4-8bf…
5-d4e…
4
• The “3-f57” revision is not submitted to the mobileDB database. The mobile device LOSES CONNECTIVITY to the cluster (cellular dead zone) before it can be replicated across.
• The mobileDB user continues to update their document locally during the connection outage

Figure 2-4 Example of lost connectivity between Cloudant cluster and mobile device

As shown in Figure 2-4, without connectivity to the network, the mobileDB user continues to write updates to their local database. This revision history needs to be propagated to other nodes in the Cloudant database after connectivity with the network is restored.

In Figure 2-4, the client accessing mobileDB continues to make updates and revisions to their local database during the period that network connectivity is lost. Local reads and writes are made possible by SQLite and the Cloudant Sync libraries for mobile devices. Upon restoration of connectivity with the Cloudant cluster, these revisions need to be replicated and reconciled with any revisions that might have occurred across other nodes in the cluster.

If a user deliberately chooses to delete a document from the example's revision tree, the body of the document (the content fields) is removed. However, a tombstone indicating the _id and _rev fields remains forever. If you are constantly deleting documents, there will be an ever-growing collection of tombstones persisted within the database. In terms of consuming disk space, this should only become a problem when the number of tombstone objects reach into the billions, given that these objects have a small storage footprint. It is necessary to maintain tombstones to ensure that records of deletions are replicated correctly throughout the nodes of a Cloudant cluster.

Figure 2-5 shows connectivity restored to the Cloudant cluster. The changes made on mobileDB's local database instance are replicated with the changes made in the Cloudant cluster databases. As before, the conflicting document versioning needs to be reconciled at the application layer. Cloudant never deletes conflicting versions of documents by default.

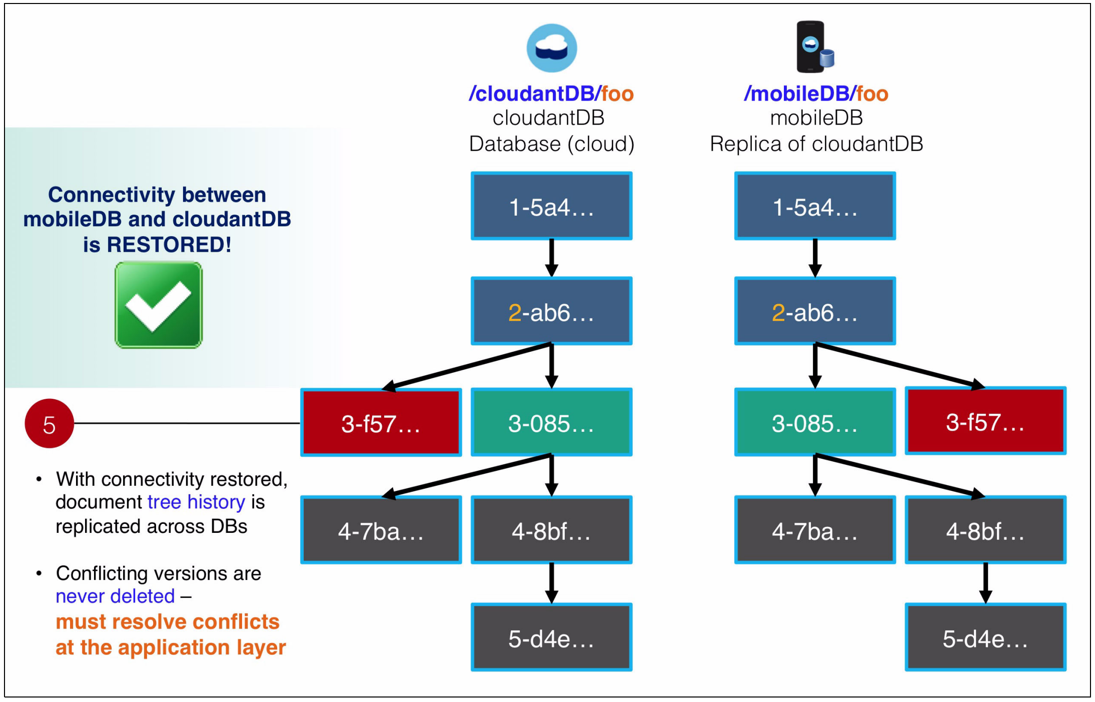

/cloudantDB/foo
cloudantDB
Database (cloud)
/mobileDB/foo
mobileDB
Replica of cloudantDB
Connectivity between mobileDB and cloudantDB is RESTORED!
5
With connectivity restored, document tree history is replicated across DBs
Conflicting versions are never deleted – must resolve conflicts at the application layer
1-5a4…
2-ab6…
3-f57…
3-085…
4-7ba…
4-8bf…
5-d4e…

Figure 2-5 Example of restored connectivity

After replication is performed, every leaf revision in cloudantDB will be present within mobileDB. Cloudant actively uses hash history technology to preserve the integrity and consistency of the cluster as a whole. Append, merge, drop, or pick the winning revision - it is up to you as the developer to determine how to resolve document conflicts. Cloudant has no tooling to do that automatically. Conflicts are never automatically resolved within the database. It is up to the developer to resolve at the application layer.

Settling on the *right* version of a document is something that Cloudant deterministically returns for you by default. If you want to know about the existence of all conflicts, you need to ask for them. This action is accomplished by enabling the parameter conflicts=true as part of your GET call to the Cloudant API. The API returns a list of revision identifiers that are siblings to the document you specified. Currently, Cloudant will not, in the database metadata, provide the number of conflicts that exist. However, you can use the built-in MapReduce functions to build a list of documents that have revision conflicts.

Indexes within Cloudant do get sparse over time with tombstone objects left in place of documents that have since been updated or deleted from the database. Compaction trims the content from these document records marked as *deleted*, but preserves the _id and _rev fields and is done automatically within Cloudant. This activity is handled internally and users do not need to manually configure the process themselves. Users cannot rely on the existence of ancestor revision bodies for restoring earlier document versions or content as a result of this compaction process.

If a developer suspects that query indexes within a database are corrupted, they can opt to have Cloudant rebuild them. Views are defined in design documents (and search indexes). Those indexes are stored on disk according to the signature of the design document that defined it. If you are creating an identical design document with the exact same function, the system outsmarts you and realizes that you are pointing to an index that already exists. Instead, a developer can implement an arbitrary change to the function (such as adding a comment line) to fool the system into building a new index. If you delete the design document, you delete the index as well. Secondary (view) indexes are built using the same append-only B-tree that is used for other Cloudant indexes. As such corruption is not really something observed except during hardware failures.

3

IBM Cloudant Sync

This chapter describes IBM Cloudant Sync for disconnected applications, its functions, and API. Cloudant Sync is used in the Cloudant replication process that was described in Chapter 1, “IBM Cloudant replication and sharding” on page 1. Cloudant Sync supports massive scaling of applications by allowing data to be accessible anytime and anywhere. The task of managing users, systems, and environmental data when there are massive networks of mobile users or remote devices can be a scalability and availability nightmare. Cloudant addresses that situation by providing developers with a simplified solution to address that complexity.

This chapter includes the following sections:

- Cloudant Sync overview
- Cloudant Sync for large-scale and disconnected apps
- Architecture
- Cloudant open source libraries for mobile devices
- Supporting replication for many devices
- Client libraries

3.1 Cloudant Sync overview

Mobile and web applications are expected to manage a variety of structured and unstructured data. This data is accessed by massive networks of users, devices, and business locations, or even sensors, vehicles, and Internet-enabled devices. With IBM Cloudant as your Database-as-a-Service (DBaaS) data layer, you can enable your applications to deliver the availability, elasticity, and reach required by mobile and web applications.

Cloudant delivers a scalable enterprise JavaScript Object Notation (JSON) database with easy-to-use administration and management, rich developer support, and powerful mobile and web capabilities. It is best suited for apps that require an operational data store to handle a massively concurrent mix of low latency reads and writes. Its data replication and synchronization technology enables continuous data availability and offline application usage for mobile or remote users. As a JSON document store, Cloudant is ideal for managing multi-structure or unstructured data.

It is clear that the future for mobile devices is having all your data available all the time, even when network connectivity is unavailable. With Cloudant Sync, developers can synchronize data across devices by using Apache CouchDB and Cloudant. IBM Cloudant Sync libraries have these two main differentiating features:

- A new API designed for the needs of mobile developers
- A new indexing and query layer that closely matches the requirements and expectations of application developers over what the CouchDB view model does

The Cloudant HTTP interface is straightforward for those who are well versed in HTTP. Cloudant Sync provides a native API for devices. The Conflicts API is an example of this capability in Android docs.

Cloudant also addresses the querying requirements on local data stores, which often differ from those required for data stored in the cloud. Customers often run large analytics tasks over their data using the Cloudant incremental map-reduce platform. On a device, the queries are usually simpler so the objective was to build an easy-to-use, more familiar indexing and query framework (such as for Android and iOS). Cloudant Sync is being released under the Apache license so that it can be used by both proprietary and open source projects.

3.2 Cloudant Sync for large-scale and disconnected apps

Most databases require updates to occur in a central, “master” database. This situation can result in performance bottlenecks and also prevent client apps from running when the connection to the master database is unavailable.

Cloudant Sync enables you to push database access to mobile devices, remote facilities, sensors, and Internet-enabled devices, so that you can enable client apps to continue running offline. Cloudant Sync enables mobile and distributed apps to scale by replicating and syncing data between multiple readable, writable copies even on mobile iOS and Android devices. This approach is much easier and more powerful than having to connect to a single, central database to handle all data collection.

Cloudant Sync simplifies large-scale mobile development by enabling you to create a single database for every user. You simply replicate and sync the copy of this database in Cloudant with a local copy on their phone or tablet (or other items such as vehicles, sensors, and appliances). This approach can reduce round-trip database requests with the server. When there is no network connection, the app runs off the database on the device. When the network connection is restored Cloudant resyncs the device and server. This local database can and usually is a subset of the online database. Some of the Cloudant groups of the largest mobile developers, such as Samsung developers, have scaled into the millions of databases.

Exposing a consistent API on all device platforms supports an architecture that enables multi-master syncing with eventual consistency rather than just cluster-to-cluster exchanges between data centers as is already done today. Interaction between data centers, clusters, and phones is now possible. This approach could eventually use different forms of connectivity between the devices themselves to create new paths back to the data center. Think of it as *phone-to-iPad-to-the-data-center*. With one consistent API, Cloudant can create new pathways for data to reach more users. This approach makes it easy for both web developer and mobile developers to jump-start their application development.

Using Cloudant Sync libraries brings added value to help the developer build on top of their environment. Cloudant Sync also eliminates the need for the developer to manage the storage of data on the local device and the server. There is no need for the developer to think about how, where, and when to store their data. This concept is a simple solution for storing querying and synchronizing the data on a local device. Synchronizing data between devices is important in today's *Internet of Things* world. Being fully compatible with Apache CouchDB the data can be isolated easily at the individual level, group level, regional level, and other levels.

The API for the mobile developer attempts to remove most of the backend complexity. Creating, reading, updating, and deleting documents is supported. Querying and indexing can be accomplished on the local device as needed.

Cloudant Sync libraries handle the mapping of JSON documents to the SQLite relational data store on your mobile devices, without intervention by the developer. The Cloudant SQLite-based “sync” libraries follow a multiversion concurrency control (MVCC) data model to build indexes locally on your mobile device. Replication is still handled by Cloudant, but during connectivity outages, your local reads, writes, and index queries are managed by the SQLite data layer.

Data is stored using local storage on your mobile device, with the native SQLite client library handling the synchronization back and forth between your device and the Cloudant database cluster. If connectivity is lost with the cluster, you retain the ability to perform local reads, writes, and queries against indexes. You can continue to create, update, and delete database documents all as though you were connected to the Cloudant database cluster.

When your device regains connection to the network, the Cloudant Sync library handles pushing that data up to the Cloudant cluster. After your device's data arrives at the cluster, your edits are propagated through traditional Cloudant replication and synchronization mechanisms to update the rest of the nodes. This action is done according to the quorum parameters that you have defined for your database. As before, no additional libraries are required for Cloudant Sync to interact with your database cluster. All requests are passed over HTTP(S) by using the Cloudant RESTful API.

Cloudant can also replicate to mobile endpoints by using the Cloudant Sync libraries. This capability allows you to push databases from your Cloudant cluster on the cloud directly to your mobile device for local read and write operations, even if connectivity is lost with the network. Mobile replication and synchronization libraries handle the process of restoring a consistent view of the database across the entire Cloudant cluster (including your local database copy) upon restoration of connectivity. Data protection, security, and geo-load balancing are handled transparently across the server after it is configured.

3.3 Architecture

The goal of this solution was to make it as easy as possible to develop applications. Figure 3-1 shows how Cloudant communicates over a RESTful API (HTTPS) using your browser or mobile application. Cloudant Sync adds value by adding a number of elements to your service that is built on top of a full Apache Lucene search engine for text indexed searching, security, and incremental map reduction. All this functionality is built on top of a geo-load balancer that can use any one of the available infrastructure offerings such as IBM SoftLayer or Rackspace, Amazon Web Services (AWS), and Microsoft Azure.

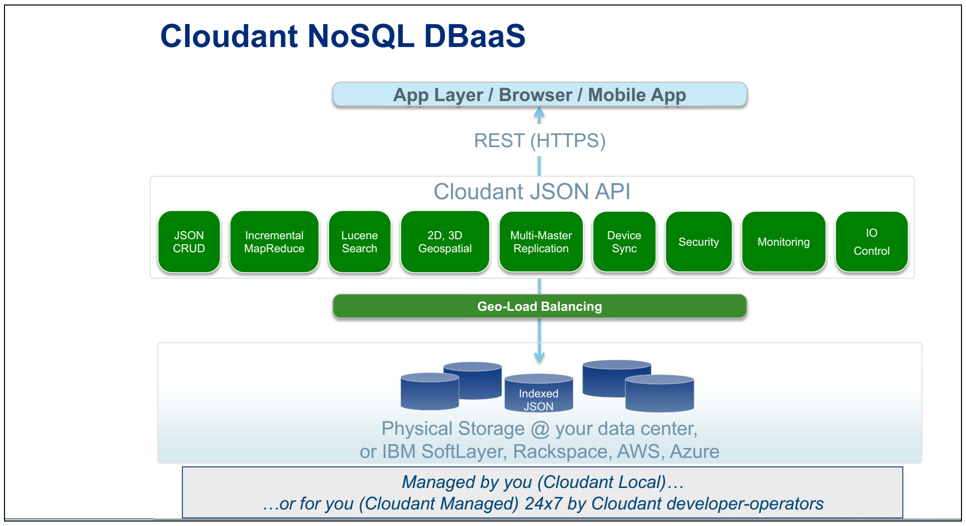

Cloudant NoSQL DBaaS
App Layer / Browser / Mobile App
REST (HTTPS)
Cloudant JSON API
JSON CRUD
Incremental MapReduce
Lucene Search
2D, 3D Geospatial
Multi-Master Replication
Device Sync
Security
Monitoring
IO Control
Geo-Load Balancing
Indexed JSON
Physical Storage @ your data center, or IBM SoftLayer, Rackspace, AWS, Azure
Managed by you (Cloudant Local)…
…or for you (Cloudant Managed) 24x7 by Cloudant developer-operators

Figure 3-1 Cloudant NoSQL DBaaS

3.4 Cloudant open source libraries for mobile devices

Cloudant has announced the open sourcing of Android and iOS sync libraries. The future for mobile devices is the concept of having all your data available all the time, even when network connectivity is unavailable.

The Cloudant Sync libraries build on a number of battle-tested open source libraries. Figure 3-2 shows a conceptual diagram of the architecture for both the Android and iOS libraries.

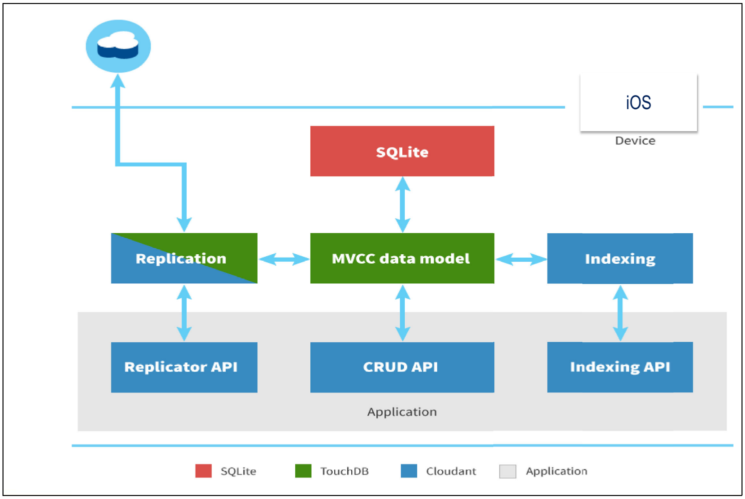


Figure 3-2 Basic mobile device architecture

It might seem like an odd choice to build a NoSQL database on top of SQLite. SQLite is well proven in production, and is available on all of the platforms IBM is targeting. Using SQLite as the durability layer was a simple choice. MVCC code was used from TouchDB, which layers MVCC semantics onto SQLite. Again, this code is well-tested, particularly on iOS, so creating something new did not make sense.

The library is able to synchronize data directly with either Cloudant or CouchDB over HTTP(S) without the need for any extra software. iOS replication code was based on code from TouchDB. However, Cloudant created its own on Android to make better use of the facilities that the Java platform offers.

On top of these core pieces, Cloudant wrapped a native-application-friendly API. The final code again used SQLite, allowing developers to easily build indexes around the JSON data. This approach allows developers to easily use SQLite fast indexing with the data from their JSON documents. Each document in the database can be indexed in any number of ways, allowing for flexible querying semantics, which are harder to achieve with the CouchDB map-reduce views. Cloudant found this to be a more natural fit to application use cases in their testing.

Cloudant Sync uses some of the Couchdb libraries (built on Java Script), which works well for browsers. It also used a native code implementation of two languages: Objective-C iOS and OS X, and Java for both Android and other Java platforms. Cloudant reused existing open source code and platforms where possible. Cloudant hosts SQLite, a library that has a long heritage and is a proven technology that works well. The CouchDB MVCC was implemented on top of the SQL tables. Cloudant handles merging changes, indexing, and finding the data on your device. For example, rather than use views, Cloudant uses a more search-related API.

Example 3-1 shows the following steps:

1. Create a document
2. Create an index
3. Run a query

Example 3-1 also shows the following capabilities:

- Other sample queries
- Other query options

Example 3-1 Code for creating and querying a database

```
1: Create a document

Docs
@{
@"country": @"UnitedStates",
@"area": @(752614)
},
!
@{
@"country": @"Canada",
@"area": @(390580)
}
"

2: Create an index

// Datastore is CouchDB database - pool
// of JSON documents
CDTDatastore *datastore = …;
// Standard alloc/init using datastore
CDTIndexManager *im = …;
!
[im ensureIndexedWithIndexName:@"country"
fieldName:@"country"
error:&error];

3: Run a query

NSDictionary *query;
CDTQueryResult *result;
!
query = @{@"country": @"UnitedStates"};
!
!
result =[im queryWithDictionary:q
error:&error];
!
!
for (CDTDocumentRevision *rev in result) {
NSLog(@"", rev.docId);
}

4: Other sample queries
!
```

```
@{
// Exact equality
@"country": @"UnitedStates",
!
// One of
@"country": @[@"UnitedStates", @"Canada"],
!
// >=
@"area": @{@"min": @(100000)},
!
// <=
@"area": @{@"max": @(1000000)}
};
"
```

5: Other query options

```
!
[im queryWithDictionary:@{@"area": @{@"min": @(10000)}}
options:@{
!
// Sort:
// by:
kCDTQueryOptionSortBy: @"area",
// order:
kCDTQueryOptionDescending: @YES,
!
// Paginate:
// start at result N:
kCDTQueryOptionOffset: @(20),
// return N results:
kCDTQueryOptionLimit: @(20)
!
}
error:&error];

?
```

You can find the source libraries and the source code in the GitHub Repositories:

- iOS/OSX, access this web address: https://github.com/cloudant/CDTDatastore
- Android/Java, access this web address: https://github.com/cloudant/sync-android

The libraries each contain this information:

- Source code
- Guides on how to include the code in your application
- Example applications
- Documentation

Included in an Objective-C application are the following items:

- iOS/OSX uses CocoaPods
- pod “CDTDatastore”
- #import <CloudantSync.h>

Example 3-2 shows the code to include in a Java application.

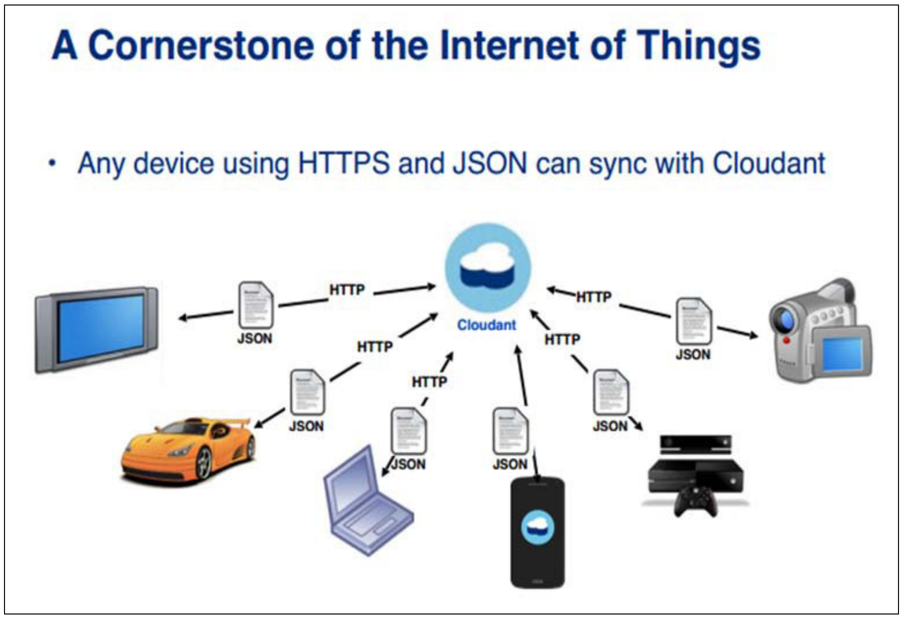
Example 3-2 Code for a Java application

```
// Add maven repo
repositories {
mavenLocal()
maven { url "http://cloudant.github.io/
cloudant-sync-eap/repository/" }
mavenCentral()
}
!
// Add dependency
dependencies {
compile group: 'com.cloudant',
name: 'cloudant-sync-datastore-android',
version:'0.3.2'
}
"
```

In summary, Cloudant replicates to mobile endpoints by using the Cloudant Sync libraries. It allows you to push databases from your Cloudant cluster on the cloud directly to your mobile device for local read and write operations. These mobile replication and synchronization libraries handle the process of restoring a consistent view of the database across the entire Cloudant cluster (including your local database copy) upon restoration of connectivity.

3.5 Supporting replication for many devices

Cloudant adheres to the normal web standards of mobile sync using HTTPS and JSON. As IBM becomes more involved in connecting to a world full of sensor data, the API provides support. Figure 3-3 shows the interactions of devices with the Internet of Things.



A Cornerstone of the Internet of Things

- Any device using HTTPS and JSON can sync with Cloudant

Figure 3-3 Devices and the Internet of Things

Cloudant Sync allows you to replicate your data and distribute readable (and writable) replicas of your data across multiple data centers, devices, and cloud providers, and keep changes to them synchronized. This approach increases up time and reduces access latency by connecting users to the closest copy of the data. This approach does for mutable application data what a content delivery network (CDN) does for static content.

The versatility in Cloudant Sync for replication between Cloudant Local databases and Cloudant in your data centers supports a hybrid environment where Cloudant is running as a managed DBaaS on public cloud platform. You can also replicate between Cloudant and Apache CouchDB to increase the size of your data cloud less expensively using free open source data stores in branch offices, points of sale, and other applications.

You have a lot of control over what gets replicated, how often it occurs, and how change conflicts are resolved if the same JSON document is changed in more than one replica. You can define filters, which are rules that control what documents are replicated where. You can set replications to occur intermittently (and how often), or continuously so that data moves in real time.

The Cloudant APIs make it easy to view changes at a database or an account level, to address change conflicts, and to set rules for resolving conflicts.

Cloudant customers use replication and sync to create copies (or replicas) of data in multiple locations for fault tolerance. That means if one source goes offline, there are others available to handle requests. You can push data to “Edge” databases such as data marts or spreadsheets, making it great for independent analytic projects.

Enable offline computing by using hub-and-spoke databases and mobile sync. Distribute subsets of data to specific participants such as partners and branch offices by using replication filters.

3.6 Client libraries

Client libraries are the tools that let you develop your own applications to work with Cloudant databases. Some client libraries are formally supported by Cloudant. These client libraries include mobile, Java, and Node.js. A supported library is one where you can contact Cloudant if you encounter a specific, reproducible problem with the library. Third-party client libraries are not maintained or supported by Cloudant. If you encounter a specific, reproducible problem with the third-party library, you should contact the library owner for assistance.

The supported client libraries are as follows:

- Mobile

  The Cloudant Sync library is used to store, index, and query local JSON data on a mobile device. It is also used to synchronize data between many devices. Synchronization is controlled by your application. The library also lets you manage and resolve conflicts easily, either in the local device or in the remote database.

  Two versions are available for mobile devices:

  – Cloudant Sync - Android / JavaSE
  – Cloudant Sync - iOS

- Java

  java-cloudant is the official Cloudant library for Java.

- Node.js

  nodejs-cloudant is the official Cloudant library for Node.js.

See these resources for further information about supported libraries and framework integration:

- Cloudant documentation:

  `https://docs.cloudant.com/`

- IBM Knowledge Center for Cloudant Data Layer Local Edition V1.0 documentation:

  `http://www-01.ibm.com/support/knowledgecenter/SSTPQH_1.0.0/com.ibm.cloudant.local.doc/SSTPQH_1.0.0_welcome.html?lang=en`

4

Building and querying the geospatial index

One of the unique advantages of Cloudant over other NoSQL databases is its ability to work with location data. This capability is accomplished by creating geospatial indexes that allow for advanced polygon and 4-D querying in real time.

This chapter includes the following sections:

4.1 Why is Cloudant geospatial an important feature?

The Cloudant geospatial feature (Figure 4-1) combines the advanced geospatial queries of GIS with the flexibility and scalability of the Cloudant Database-as-a-Service (DBaaS):

- Web and mobile developers enhance their applications with geospatial operations that go beyond simple bounding boxes
- Integrates with existing geographic information system (GIS) applications so the applications can scale for data size, concurrent users, and multiple locations
- Simple to implement as a standard because of well-defined data types, easy to read, and integrated into many development languages and platforms.

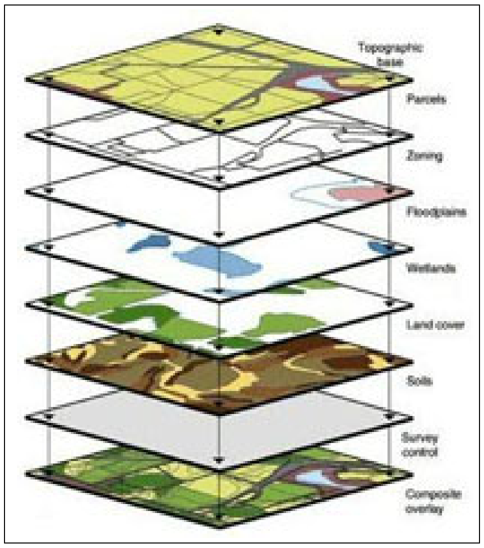


Figure 4-1 Cloudant geospatial feature

4.2 GeoJSON structure

GeoJSON is an open standard format for storing geographical features using JavaScript Object Notation (JSON). GeoJSON consists of three parts:

- Geometry
  - Must have type field with a value such as “Point”, “LineString”, or “Polygon”
  - Must have “coordinates”
- Properties
  - All the other data
  - Not required by Cloudant
- Type
  - Must be “Feature”

Example 4-1 shows example GeoJSON code.

Example 4-1 Example GeoJSON

```
{
“_id”: “79f14b64c57461584b152123e38c6449”,
"_rev": "1-e6b5ca2cd8047747co07cf36d290a4c8"
"geometry": {
"coordinates": [
-71.13687953,
42.34690635
],
"type": "Point"
},
"properties": {
"compnos": "142035014",
"domestic": false,
"fromdate": 1412209800000,
"main_crimecode": "MedAssist",
"naturecode": "EDP",
"reptdistrict": "D14",
"shooting": false,
"source": "boston"
},
"type": Feature"
}
```

In Example 4-1, you can see these characteristics:

- "_id" and "_rev" are Cloudant required JSON Document structures.
- "geometry" is the GeoJSON structure that holds the "point", and the following "coordinates" define that GeoJSON structure with longitude and latitude coordinate measurements. All GeoJSON structures in a Cloudant data source require this geometry definition.
- "properties" are additional data that describe the data point that might or might not be needed for an application. This data is not a requirement for Cloudant.
- "type" is required by Cloudant, and must always be defined as "Feature".

4.3 Cloudant: Going beyond bounding boxes

Many databases say they can handle the GeoJSON structure and are able to do primitive queries on geo-location data. You might have seen applications when, for example, selecting all the dry cleaners within a half mile of your location. These applications draw what is called a *bounding box*. A bounding box is a simple box that consists of four data points that define this box. The points are tested to be inside or outside the box (Figure 4-2).

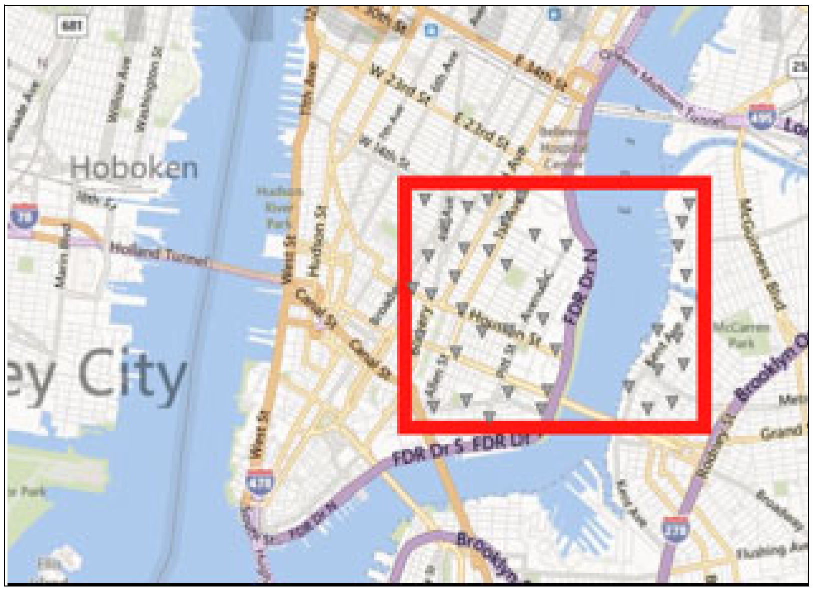


Figure 4-2 Bounding box example

If an individual is on the east side of the isle of Manhattan, a simple bounding box identifies all the dry cleaners that are within a half mile, even if the East Hudson river is in the way. Using a bounding box brings up a lot of data that is useless to the user. Cloudant allows you to expand on this approach by incorporating more complex geometries (Figure 4-3).

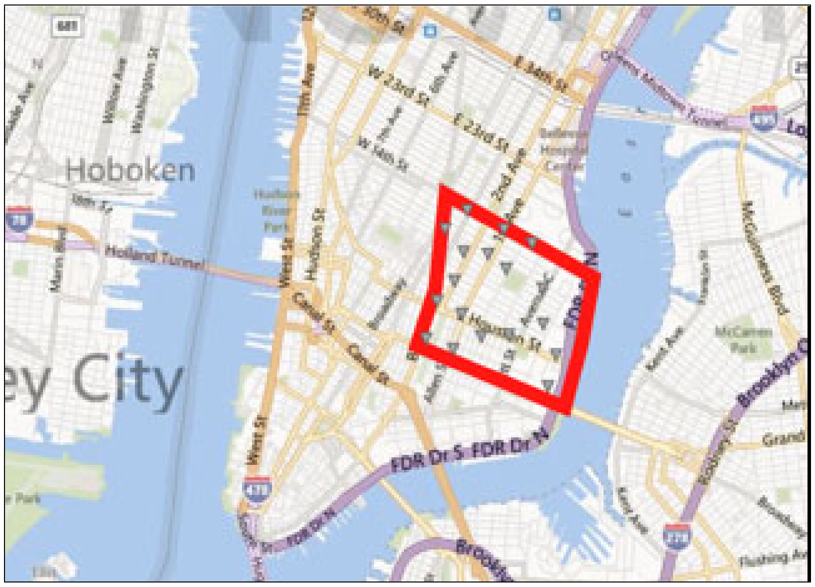


Figure 4-3 Advanced polygon definition

You can see that Cloudant can incorporate extra geometries from GIS to define a polygon that limits results to valid data within a half mile that are reachable without crossing a body of water.

4.4 Cloudant geospatial functions

In the Cloudant geospatial feature, geospatial relations are specified by the relation query parameter. Cloudant geospatial supports the following standard geospatial features:

- Bbox: True if the geometry is within the envelope provided.
- Contains: True if the second geometry is wholly inside the first geometry.
- Crosses: True if the intersection of the two geometries results in a value whose dimension is less than the geometries. The maximum dimension of the intersection value includes points interior to both the geometries. The intersection value is not equal to either of the geometries.
- Disjoint: True if the two geometries do not touch or intersect.
- Equals: True if the two geometries are the same.
- Intersects: True if the two geometries intersect.
- Overlaps: True if the intersection of the geometries results in a value of the same dimension as the geometries that is different from both of the geometries.
- Touches: True if and only if the only common points of the two geometries are in the union of the boundaries of the geometries.
- Within: True if the first geometry is wholly inside the second geometry.

4.5 Conclusion

Using GeoJSON for geospatial location processing in Cloudant, customers can go further and develop broader user experiences that fine-tune results based on location and where they are going using routes and temporal features.

To learn more about this unique feature of Cloudant, see the documentation and examples at:

`http://docs.cloudant.com/guides.html#cloudant-geospatial`

5

Cloudant libraries, drivers, and open source integrations

Cloudant provides and has access to several libraries and drivers to support application development in various programming languages.

This chapter includes the following sections:

- Cloudant libraries for application development
- A node.js example
- Building an application on IBM Bluemix

5.1 Cloudant libraries for application development

Cloudant provides several libraries to support application development in various programming languages. Because Cloudant has an HTTP RESTful API, you could also use your languages' native HTTP and JSON libraries to develop Cloudant solutions. Most libraries are open source and typically have compatibility with both Apache CouchDB and Cloudant.

Apache CouchDB: Apache CouchDB drivers most likely will not have support for Cloudant-specific APIs similar to Lucene Search and Geospatial Indexes.

Table 5-1 shows the wide range of supported drivers.

Table 5-1

Language	Libraries and frameworks	Additional material
C#/.NET	► MyCouch: https://github.com/danielwertheim/mycouch ► LoveSeat: https://github.com/soitgoes/LoveSeat ► Divan: https://github.com/foretagsplatsen/Divan ► Relax: https://github.com/code-attic/Relax ► Hammock: https://code.google.com/p/relax-net/ ► EasyCouchDB: https://github.com/hhariri/EasyCouchDB ► WDK.API.CouchDB from Kanapes IDE: https://code.google.com/p/skitsanoswdk/source/browse/#svn%2Ftrunk%2FWDK10%2FWDK.API.CouchDb	Create, retrieve, update, and delete: https://github.com/cloudant/haengematte/tree/master/c%23

Language	Libraries and frameworks	Additional material
Java	► java-cloudant: https://github.com/cloudant/java-cloudant ► jcouchdb: https://code.google.com/p/jcouchdb/ ► jrelax: https://github.com/isterin/jrelax ► LightCouch: http://www.lightcouch.org/ ► Java Cloudant Web Starter boilerplate for IBM Bluemix™: https://console.ng.bluemix.net/?ace_base=true/#/store/cloudOEPaneId=store&appTemplateGuid=CloudantJavaBPTemplate&fromCatalog=true	► Create, retrieve, update, and delete with HTTP and JSON libraries: https://github.com/cloudant/haengematte/tree/master/java ► Create, retrieve, update, and delete with ektorp library: https://github.com/cloudant/haengematte/tree/master/java/CrudWithEktorp ► *Building apps using Java with Cloudant on IBM Bluemix*: https://cloudant.com/blog/building-apps-using-java-with-cloudant-on-ibm-bluemix/#.VP7vuGOVDG4 ► *Build a game app with Liberty, Cloudant, and Single Sign On* (a Bluemix example): http://www.ibm.com/developerworks/cloud/library/cl-multiservicegame-app/index.html?ca=drs- ► *Building a Java EE app on IBM Bluemix Using Watson and Cloudant* (a Bluemix example along with YouTube video): https://developer.ibm.com/bluemix/2014/10/17/building-java-ee-app-ibm-bluemix-using-watson-cloudant/
JavaScript	► Backbone.cloudant: https://github.com/cloudant-labs/backbone.cloudant ► *Backbone and Cloudant blog*: https://cloudant.com/blog/backbone-and-cloudant/#.VP7y72OVDG4 ► sag.js: http://www.saggingcouch.com/jsdocs.php ► PouchDB (JavaScript database for browser with offline sync): http://pouchdb.com/ ► If you require CORS support with Cloudant, see these websites: – Blog post: https://cloudant.com/blog/cors-and-faceted-search-coming-soon-to-a-cloudant-cluster-near-you/#.VP7z5mOVDG4 – Guide: https://gist.github.com/chewbranca/0f690f8c2bfad37a712a	► Create, retrieve, update, and delete using jQuery: https://github.com/cloudant/haengematte/tree/master/javascript-jquery ► CSVtoCloudant: – UI for importing .csv files into Cloudant): https://github.com/michellephung/CSVtoCloudant – Access the app: https://michellephung.github.io/CSVtoCloudant/ ► csv2couchdb (UI from Mango Systems to import .csv files to CouchDB/Cloudant): https://github.com/Mango-information-systems/csv2couchdb ► songblog (example app using JQuery): https://github.com/millayr/songblog ► PouchDB: *Getting Started Guide* (an example to do application that syncs from browser to Cloudant or CouchDB): http://pouchdb.com/getting-started.html ► locationtracker (example app to record and map location using PouchDB, CouchApp, and Cloudant): https://github.com/rajrsingh/locationtracker

Language	Libraries and frameworks	Additional material
Node.js	► nodejs-cloudant (npm): `https://github.com/cloudant/nodejs-cloudant` ► sag-js (also works in the browser): `https://github.com/sbisbee/sag-js` – See saggingcouch for more detail: `http://www.saggingcouch.com/` ► nano (a minimalist implementation): `https://github.com/dscape/nano` ► restler (delivers the best performance, but is really barebones): `https://github.com/danwrong/restler` ► cradle (a high-level client is also available if you absolutely need ease of use at the cost of lower performance): `http://cloudhead.io/cradle` ► cane_passport (a repository for a sample node app with Cloudant Angular Node and Express): `https://github.com/ddemichele/cane_passport` ► express-cloudant (a template for Node.js Express framework also using PouchDB and Grunt): `https://github.com/cloudant-labs/express-cloudant` ► Node.js Cloudant DB Web Starter (boilerplate for Bluemix): `https://www.ng.bluemix.net/docs/#starters/node-cloudant/index.html#node-cloudant`	► Create, retrieve, update, and delete: `https://github.com/cloudant/haengematte/tree/master/nodejs` ► *Using Cloudant with Node.js*: `https://cloudant.com/blog/using-cloudant-with-node-js/#.VP7832OVDG4` ► Cloudant-Uploader (utility to upload .csv files to Cloudant): `https://github.com/garbados/Cloudant-Uploader` ► couchimport (utility to import csv or tsv files into CouchDB or Cloudant): `https://github.com/glynnbird/couchimport` ► *Getting started with IBM Bluemix and Node.js*: `http://thoughtsoncloud.com/2014/07/getting-started-ibm-bluemix-node-js/` ► *A Cloud medley with IBM Bluemix, Cloudant DB and Node.js:* *`https://gigadom.wordpress.com/2014/08/15/a-cloud-medley-with-ibm-bluemix-cloudant-db-and-node-js/`* ► *Build a simple word game app using Cloudant on Bluemix* (uses Node.js): `http://www.ibm.com/developerworks/cloud/library/cl-guesstheword-app/index.html?ca=drs-` ► *Building a Real-time SMS Voting App Part 1: Node.js and Cloudant* (first in a six part series): `https://www.twilio.com/blog/2012/09/building-a-real-time-sms-voting-app-part-1-node-js-couchdb.html` ► *Tutorial – Building a Multi-Tier Windows Azure Web application use Cloudant's Couchdb-as-a-Service, node.js, CORS, and Grunt*: `https://msopentech.com/blog/2013/12/19/tutorial-building-multi-tier-windows-azure-web-application-use-cloudants-couchdb-service-node-js-cors-grunt-2/` ► *Do it yourself: Build a remote surveillance app using Bluemix, Cloudant, and Raspberry Pi*: `http://www.ibm.com/developerworks/library/ba-remoteservpi-app/index.html` ► *Analyze game data from the Oculus Rift using Bluemix* (Express, Node.js, and Cloudant): `http://www.ibm.com/developerworks/cloud/library/cl-oculus-app/index.html` ► *Conserve water with the Internet of Things: Part 3* (Bluemix, Node-RED/Node.js, D3.js, and Cloudant): `http://www.ibm.com/developerworks/cloud/library/cl-poseidon3-app/index.html` ► *Build a Where? app for web and Pebble users* (Bluemix, Node.js, Pitney Bowes geo services, Cloudant): `http://www.ibm.com/developerworks/library/mo-pebble-where-app/index.html`

Language	Libraries and frameworks	Additional material
PHP	► Sag for CouchDB: http://www.saggingcouch.com/ ► Doctrine CouchDB Client: https://github.com/doctrine/couchdb-client ► PHP-on-Couch: https://github.com/dready92/PHP-on-Couch	► Create, retrieve, update, and delete: https://github.com/cloudant/haengematte/tree/master/php
Python	► Cloudant-Python (with blog posts): https://github.com/cloudant-labs/cloudant-python ► CouchDB: http://pythonhosted.org/CouchDB/ ► *Requests: HTTP for Humans*: http://docs.python-requests.org/en/latest/ ► couchdbkit: http://couchdbkit.org/	► Create, retrieve, update, and delete (using requests): https://github.com/cloudant/haengematte/tree/master/python ► *Using Python with Cloudant*: https://cloudant.com/blog/using-python-with-cloudant/#.VP8FvmOVDG4 ► csv-import (script to import .csv files): https://github.com/claudiusli/csv-import ► flaskr (Python Flask example application Flaskr by using different data layers of Cloudant, Couchdbkit, SQLite, and JSON files): https://github.com/michaelbreslin/flaskr
Ruby	There are many CouchDB clients listed on The Ruby Toolbox: https://www.ruby-toolbox.com/categories/couchdb_clients	Create, retrieve, update, and delete for Ruby: https://github.com/cloudant/haengematte/tree/master/ruby
Client (Browser, iOS, and Android)	► Cloudant Sync - Android: https://github.com/cloudant/sync-android ► Cloudant Sync - iOS: https://github.com/cloudant/CDTDatastore ► PouchDB (JavaScript database for browser with offline sync): http://pouchdb.com/ ► Mobile Cloud (boiler plate for Bluemix includes: Node.js, Security, Push, and Mobile Data/Cloudant): https://console.ng.bluemix.net/?ace_base=true/#/store/cloudOEPaneId=store&appTemplateGuid=mobileBackendStarter&fromCatalog=true	► Cloudant Sync Overview: https://cloudant.com/product/cloudant-features/sync/ ► Cloudant Sync Resources: https://cloudant.com/cloudant-sync-resources/#.VP8LgGOVDG4 ► *IBM Worklight Powered Native Objective-C iOS Apps with Cloudant Adapter*: http://www.tricedesigns.com/2014/11/17/ibm-worklight-powered-native-objective-c-ios-apps/ ► locationtracker (example app to record and map location by using PouchDB, CouchApp, and Cloudant): https://github.com/rajrsingh/locationtracker

Official libraries: The java-cloudant, nodejs-cloudant (npm), Cloudant Sync - Android, and Cloudant Sync - iOS are official Cloudant libraries.

For additional information, go to this web address:
https://cloudant.com/for-developers/libraries-and-tutorials/

5.2 A node.js example

To install the Cloudant driver for Node.js, you would need to have package Cloudant installed in your environment. You can do this action by running the following command:

```
npm install -save cloudant
```

After the command has executed, you can use following code to connect to the Cloudant account and see all databases (Example 5-1).

Example 5-1 Connect to Cloudant

```
var Cloudant = require('cloudant')

var me = 'username' // Set this to your own account
var password = process.env.cloudant_password

Cloudant({account:me, password:password}, function(err, cloudant) {
  console.log('Connected to Cloudant')

  cloudant.db.list(function(err, all_dbs) {
    console.log('All my databases: %s', all_dbs.join(', '))
  })
})
```

The results of running the code (Example 5-1) returns the message shown in Example 5-2.

Example 5-2 Message returned

```
Connected to Cloudant
All my databases: _replicator, _warehouser, empldb, py_test, staff, staff1,
staff_old, testdb, twitterdb
```

If you run the code in Example 5-1 again, you see the database “alice” has been added (Example 5-3).

Example 5-3 Message returned with database alice

```
Connected to Cloudant
All my databases: _replicator, _warehouser, alice, empldb, py_test, staff, staff1,
staff_old, testdb, twitterdb
```

You can find the available APIs for this driver at the following web address:

https://github.com/cloudant/nodejs-cloudant

5.3 Building an application on IBM Bluemix

IBM Bluemix is an implementation of the IBM Open Cloud Architecture. This architecture and implementation use Cloud Foundry to enable developers to rapidly build, deploy, and manage their cloud applications, while tapping a growing ecosystem of available services and runtime frameworks. Bluemix provides a series of boilerplates to accelerate application development with the Cloudant NoSQL database.

IBM developerWorks® has multiple articles outlining how do develop different applications with Cloudant DBaaS. To access developerWorks, go to this web address:

https://www.ibm.com/developerworks/

For more information, see the following web addresses:

- IBM Cloudant For Developers Libraries and Tutorials

  https://cloudant.com/for-developers/libraries-and-tutorials/

- IBM Cloudant blog *Building apps using Java with Cloudant on IBM Bluemix*

  https://cloudant.com/blog/building-apps-using-java-with-cloudant-on-ibm-bluemix

REDP-5189-00

Printed in U.S.A.