# IBM Cloudant: Database as a Service Fundamentals

Understand the basics of NoSQL document data stores

Access and work with the Cloudant API

Work programmatically with Cloudant data

Christopher Bienko
Marina Greenstein
Stephen E Holt
Richard T Phillips

Redpaper

ibm.com/redbooks

# Contents

# Notices

# Trademarks

# Preface

This IBM® Redpaper™ publication shows how the IBM Cloudant fully-managed *database as a service* (DaaS) enables applications and their developers to be more agile. As a part of its data layer, clients have access to multi-master replication and mobile device synchronization capabilities for occasionally connected devices. Applications can take advantage of Cloudant advance real-time indexing features for ad hoc full text search via Apache Lucene, online analytics via MapReduce, and advance geospatial querying.

Mobile applications can use a durable replication protocol for offline sync and global data distribution, as well as a geo-load balancing capability to ensure cross-data center availability and optimal performance. The Cloudant RESTful web-based application programming interface (API), flexible schema, and capacity to scale massively are what empowers clients to deliver applications to market faster in a cost-effective, DBA-free service model.

## Authors

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

**Christopher Bienko** covers the Information Management Cloud Solutions portfolio for the IBM Worldwide Technical Sales organization. He is a published author for *Big Data Beyond the Hype: A Guide to Conversations for Today's Data Center* (2014, Zikopoulos, deRoos, Bienko) for McGraw Hill Education. Over the better part of 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 polishing off his freshly-minted 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 DB2®. She has presented migration methodology at numerous DB2 technical conferences. She is also the author of multiple articles and 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 DaaS solutions. He has held technical management positions and has expertise in DB2, IBM Informix® database, Cloudant, along with other Information Management (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 - WW Information Management Sales, Client Value Engagement Program Manager IBM

Philip Monson - IBM Redbooks® Portfolio Manager

## Now you can become a published author, too!

Here's an opportunity to spotlight your skills, grow your career, and become a published author—all at the same time! Join an ITSO residency project and help write a book in your area of expertise, while honing your experience using leading-edge technologies. Your efforts will help to increase product acceptance and customer satisfaction, as you expand your network of technical contacts and relationships. Residencies run from two to six weeks in length, and you can participate either in person or as a remote resident working from your home base.

Find out more about the residency program, browse the residency index, and apply online at:

**ibm.com**/redbooks/residencies.html

## Comments welcome

Your comments are important to us!

We want our papers to be as helpful as possible. Send us your comments about this paper or other IBM Redbooks publications in one of the following ways:

- Use the online **Contact us** review Redbooks form found at:

  **ibm.com**/redbooks

- Send your comments in an email to:

  redbooks@us.ibm.com

- Mail your comments to:

  IBM Corporation, International Technical Support Organization
  Dept. HYTD Mail Station P099
  2455 South Road
  Poughkeepsie, NY 12601-5400

## Stay connected to IBM Redbooks

- Find us on Facebook:

  http://www.facebook.com/IBMRedbooks
- Follow us on Twitter:

  https://twitter.com/ibmredbooks
- 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

# Survey of the database landscape

Databases have been around for years, mainly hierarchical, networking, and relational, and over the past 40 years, the most prevalent have been relational databases that support Structured Query Language (SQL). The objective with relational databases was query flexibility over schema flexibility. This trade off made the most sense at the time from a business and use-case perspective. The expectation (and often the norm) was that large organizations would have database administrators in-house and on-hand for managing and structuring schemas tailored to a specific company's data storage and retrieval requirements. Issues of complexity and fine-tuning required in order to keep a relational solution viable were secondary to how specific queries performed against the stored data. This continues to be the case with many of the enterprise customers we see today.

In the following chapters, we introduce you to a relatively recent database type known as a *document store* where flexibility in writing the data is beneficial over the ease of reading the data. This type of database is a known as *schema-on-read* where much of the programming is done to get the data out, unlike the relational approach where the programming and formatting is done to get the data into the database. The database we are referring to here is *Cloudant*, an IBM cloud solution supporting a database as a service (DaaS) delivery model. Using IBM Cloudant, one can deploy the database as a service in a public cloud, private cloud, or combination of both referred to as a hybrid cloud by connecting Cloudant DaaS, Cloudant Local, or the on-premises edition of Cloudant. With Cloudant, many small or large databases can be local, remote, and even mobile. For security reasons, selected data can always be protected from leaving the premises. Clients no longer have to deploy any dependent middleware to deliver database requests and applications can connect directly to the database. One can start small, scale on demand, and choose to manage your cloud yourself or let Cloudant manage the DaaS environment for you. Cloudant allows clients to continuously optimize the data layer for cost, performance, security, and reach. Clients can break up their data, distribute it and move it closer to their users around the world.

If the traditional database store continues to hold true for much of the industry today, you may be wondering what is the "evolution" in the database landscape. Before we begin with our examination of IBM Cloudant, it is important that we take a moment to look at the development of this new paradigm of databases, the driver behind this shift in the data persistence models, known as “NoSQL”. The best way to broker that conversation around “What is NoSQL?” is to look at its counterpoint: SQL, the bread and butter of relational databases.

## 1.1 The fundamentals and evolution of relational databases

Relational database management systems (RDBMSs) are set-based systems. They are implemented as two-dimensional tables of the familiar rows and columns data structures. We describe these systems as “relations” between “tuples” (rows) and “attributes” (columns). Data values are typed and can be numeric, strings, or other types of data that your use cases might require; the defining characteristic of such data types is that they are enforced by the database. Furthermore, tables are able to join and be modified to generate more complex (composite) tables. An entire discipline, known in mathematics as *relational set theory*, provides the concrete mathematical basis and proofs to allow such table joins and morphologies to occur.

RDBMS prioritizes query flexibility over having flexible schemas. This may seem like a moot point. Of course, having a performant query apparatus is going to be valuable for organizations working with transactional records, but as we dive deeper, we will see that it is a key differentiator between the worlds of SQL and NoSQL. In order to minimize redundancy and prevent the propagation of errors throughout your tabular stores, it is necessary to employ normalization strategies as you populate your relational database. Relationships between tuples and attributes must be well-defined and tightly enforced (“respected”, as some DBAs will describe it). In other words, relational databases and their schemas are well ordered and rigidly defined.

Interestingly, the term used to describe many relational systems, SQL databases, actually refers to the manner in which users and administrators interact with their database. SQL, or “Structured Query Language”, is a highly flexible and structured query language for interacting with your relational data stores. It supports JOINs that are built upon relational set theory for combining multi-dimensional tables into more complex structures, GROUP BY operations for performing aggregations upon your data, and so on. Furthermore, transactional systems heavily leverage SQL-based relational stores because they are what we call “ACIDic” systems: we will examine in much greater detail what ACID refers to in a moment.

As this is a paper focused on exploring Cloudant (a NoSQL database) and because we assume that you will be well-versed in relational database systems ahead of time, this will be the extent of our coverage of SQL database fundamentals. With that brief primer behind us, and an entire new paradigm of databases to explore, let us jump right into examining the core principles of SQL in order to answer a key question: What is NoSQL?

### 1.1.1 The relational model

There are a handful of significant milestones along the journey towards the database and analytics landscape of the industry as it stands today, where advances within Information Management technologies have made new ways of interacting with and understanding our data possible. IBM has been the driving force behind many of these innovations.

Ubiquitous throughout the database world today, SQL, described as “Structured English Query Language,” was first developed by IBMers in the early 1970s and is credited to Donald Chamberlin and Raymond Boyce. SQL was purpose-built as a means of interacting with and retrieving data from the IBM System R: a relational database system forerunner that also emerged around this time. The 1970s were characterized by the availability of hard disk drives, and IBM capitalized on this new infrastructure to invent and refine the relational database as we understand it today. From the 1970s into the 1980s, IBM released further refinements on System R, the iconic System z, and the first deployments of IBM DB2. These technologies allowed IBM customers and businesses to store increasingly larger amounts of information within secure and reliable databases; warehousing technologies accompanied this development as analytic techniques kept apace of the changing database market. IBM continues the drive towards new ways of understanding data through leadership behind providing SQL access to data stored in distributed systems like Hadoop.

SQL has three defining characteristics that, no matter the flavor, are generally consistent for each variety of SQL in the marketplace today. These are: a tabular storage schema of rows and columns; a database schema which is rigid and strictly enforced; and a relational design. SQL is a well-known language owing to its pedigree and long history, and is ubiquitous within the modern IT world, from small businesses up to enterprise customers as a result. Furthermore, because SQL is highly standardized and rigorously defined, industry professionals that gain skills in administration of SQL systems have a skill set that is easily transferrable. The toolsets available for working with SQL-based systems are bountiful as well, due to the relatively simple (in so far that it can be easily digested by technical and non-technical professionals alike) and widespread popularity of SQL. Most importantly, the relational model upon which SQL is founded allows you to easily explore data relationships; for example, examining a company's sales database to compare sales by part number, customer classification, region, industry, and so on. Relations between data can be defined to closely model the way that we associate data and objects in their real-world context.

Having looked at the reasons why SQL has seen such widespread adoption and popularity over the last four decades, we turn now to peeling back the layers to determine the shortfalls of SQL relational models. Generally, SQL tables become more sparse with null fields as the number of columns and rows increase; such a use of storage is inefficient, and best practices need to be followed in order to mitigate the amount of sparse data populating your relational stores. Secondly, SQL's rigid and simplistic data model does not lend itself well to modeling more complex objects. The equivalent to mapping a single XML document relationally can require the creation of dozens of tables when working with SQL-based storage.

The process of shredding a single object across multiple columns also has the tendency of obscuring the original form of the object you are representing within the database. For example, the data points that make up your business card: your name, email address, phone number, workplace, and so forth, might be more intuitively stored as a single object (1:1 between your card and the database object). However, SQL systems require you to shred those fields over multiple columns, and potentially multiple tables which may need to be linked by a JOIN as well, which further complicates and obscures the representation. We explore in later chapters how NoSQL's JSON document stores allow you to more intuitively persist your data for exactly this type of scenario. With SQL, retrieving what may have once been a single object can now require a significant amount of work (multiple JOINs, understanding of the schema logic) in order to reconstruct the original entity.

Evolving data models under SQL's rigidly enforced schemas is often a non-trivial task, requiring skilled expertise from database administrators (who represent a significant ongoing expense for organizations that need to keep DBAs in-house for precisely these kinds of tasks) in order to manipulate schemas for fitting new data or relational models. Changing a column's type, for example, may require a complete rebuild of said table and potentially a rebuild of all dependent tables as well, should those exist. The relational model, therefore, can be an impediment to being truly agile in your application development strategy. The structure of all data being persisted within your data stores must be well known and rigorously defined before you can begin work in earnest towards building your SQL solution.

The take-away from the shortfalls of SQL can be summarized into three types of pressures that have shaped the development and use cases of these systems to this day. Technically cumbersome procedures for modifying existing table schemas and for defining new ones has been an impediment for developers looking to be more agile, both in terms of the types of data they persist, but also how easily they can modify their schema models. Budgetary constraints resulting from needing costly DBAs in-house to manage your database solution has slowed adoption of SQL-based systems from smaller organizations that may not have the capital to afford and manage their data layer. Finally, regulatory pressures, SLAs, archiving requirements, and data governance, have shaped a culture that favors consistency and availability of your data over things such as partition tolerance.

In the following section, we examine how new trends and technologies within the database landscape are altering those perceptions around consistency and availability. As we will see, the momentum driving this change of perspective will lead to the emergence of a new data persistence paradigm: NoSQL.

### 1.1.2 The CAP Theorem

Also known as Brewer's Theorem, the CAP Theorem is an acronym that stands for three things: constancy, availability, and partition tolerance. Discussions of IBM products never have a shortage of acronyms, so why add a new one into the mix in our examination of NoSQL? Essentially, "CAP" describes at a fundamental level the strengths and weaknesses of relational database systems; it is also the driving force behind the shift towards NoSQL alternatives to relational systems, and the new advantages (and limitations) those databases introduce. The CAP Theorem posits that it is not possible for any distributed systems to ever achieve all three qualities: consistency, availability, and partition tolerance, simultaneously for one system. Let us take a moment to define what each of these components are describing.

Consistency refers to the concept that all nodes belonging to a distributed database have the same view of the data stored, at the same time. We can simplify this by imagining a fictitious bank account, which Alice and Bob both share. The bank account might be housed in Switzerland, while Alice lives in Canada and Bob lives in Germany. Collectively, they share $100.00 deposited within their account. If the database supporting this bank's operations is consistent, Bob and Alice should be able to perform a look-up on their account balance, in both Canada and Germany, to find that they do indeed have exactly $100.00 in the bank. Should Bob perform a $10.00 withdrawal from the account (bringing the shared total to $90.00), Alice should see that change in balance reflected in her next total look-up.

However, if the bank continues without processing the transaction correctly and her bank account still has $100.00 to its name, such a data layer would be lacking consistency. It is possible the database will be "eventually consistent", which is to say that at the time that Alice performed her look-up, there has not been a sufficient amount of time to propagate the changed bank account state throughout the distributed system. This concept of eventual consistency is key to the Cloudant consistency model and NoSQL systems in general. We will examine it, and the implications it has upon your data and applications, in the following sections.

Availability, the "A" in CAP Theorem, refers to a guarantee that every request made to your database cluster will receive a response corresponding to either a success or a failure. The simplest way to think about this is the very primitive "ping" test. A ping will always return some sort of response, either the packet was successfully delivered, or it failed to reach its destination, which we use to determine whether or not our target endpoint is "available" or not. In the scope of relational systems that we have been discussing thus far, availability might refer to the ability for a database to process a financial transaction you are requesting. If the database is unable to process your request, for example when a table or row is locked, we say that it is unavailable. Scenarios such as these can have a significant impact financially for both the customer and the financial organization that operates the database. For that reason, relational systems that handle transactional records tend to prioritize availability with nearly the same importance as having consistent data.

The third and final piece of the pie, partition tolerance, or the “P” in CAP Theorem, stipulates that the database cluster as a whole continues to function and operate in the event of message loss or failure in some parts of the system. If one of the nodes in your cluster goes down, or traffic over a network is disrupted, other unaffected parts of the cluster should be able to carry on and operate (perhaps at diminished capacity) for customers needing access to the database. For distributed systems, the expectation is that parts of the system will fail. Having resiliency against these failures is what we describe as being partition tolerant.

The key to the CAP Theorem proposed by Brewer is that it posits no database can ever achieve all three characteristics: consistency, availability, and partition tolerance, within the same system. At most, you can achieve two of the three, to the loss of the third characteristic. Consider RDBMS for a moment again: these systems harken from the days where transactional systems for finance and federal institutions dominated the database market. Use cases such as these require that data is 100% consistent. At no point would you want an inconsistent view of your bank account balance. They also needed to be highly available: if we consider a banking system again, it would be unacceptable for most customers not to be able to make a checking account withdrawal or account balance query at a moment’s notice. For these reasons, RDBMS has historically favored consistency and availability, but at the cost of partition tolerance. RDBMS is notoriously challenging to scale horizontally into distributed clusters. In the context of Brewer’s CAP Theorem, relational databases have opted to prioritize consistency and availability, but to the detriment of being able to scale out horizontally.

There are numerous advantages to having a data layer that can scale over distributed clusters, of course. Aside from the high availability and resilience against failure, there are significant gains in performance when you are able to deliver data closer to users. We look at how the replication and sync functionality of Cloudant are able to achieve this later. However, the most significant advantage arguably has to do with cost. If you are not scaling your database infrastructure horizontally (in other words, over distributed systems of commodity hardware), you need to scale it vertically, a solution that becomes prohibitively expensive and still does not address issues of high availability.

At this stage, we are finally ready to make a flip of the coin and look at the other face of relational database systems: NoSQL. In many ways, NoSQL is the counterpoint to what SQL is, the name certainly implies as much. The key differentiator here is the CAP Theorem that we outlined here. If the industry moves away from RDBMS towards systems that can scale out horizontally, what do we gain? What do we lose? Let us explore those questions.

## 1.2 The NoSQL paradigm

With distributed database systems, we assume that a network partition will occur. Statistics on hardware failures and first-hand experience we all have with service disruption over the Internet are significant enough to make this a fairly safe assumption. Therefore, we can surmise that in order for our data layer to function over a distributed network, our data layer solution must exhibit partition tolerance. Given the CAP Theorem, which limits us to selecting only two of the three characteristics governing all databases, we must make a choice: will our data layer prioritize consistency or availability? The willingness to trade consistency for availability exists on a spectrum, of course, it is what we describe as “tuneable”. For example, selecting for consistency would not mean that your database is 100% unavailable. (This would not be very practical for your customers.)

Data normalization is critical for the elimination of redundancy in relational database systems. Without the ability to transform and normalize data before insertion into the database, it would not be possible to ensure master data consistency: a hallmark of RDBMS. For these reasons, consistency and data availability have helped RDBMS lend itself well to handling transactional, reporting, log, and warehouse data.

Instead of prioritizing consistency and availability, recent trends in database architecture have shifted focus towards a solution based on availability and partition tolerance solution. It is unlikely that you will be able to convince your customers or your organization of a scenario where loss of availability would be tolerable. Users of any system expect that system to be responsive at any point in time that they want to utilize it. What, then, is the alternative?

According to the CAP Theorem, if we assume partition tolerance is required for working on distributed clusters and that loss of availability is intolerable, prioritize these two attributes. Selecting for partition tolerance opens up the possibility to horizontal scalability, allowing the data layer to scale out over geographically vast regions. Our system, therefore, has greater resiliency to failure because of its distributed nature. As a side effect, we gain higher availability than would be possible with a vertically scaled system, which has a single point of failure. As you can see, selecting for availability (A) follows naturally when partition tolerance (P) is built into the system architecture.

By virtue of this architecture, we gained partition tolerance and availability, but at the cost of consistency (C). Data takes time to replicate and propagate across every facet of your distributed system. The laws of physics dictate that light can only travel at a finite speed (unfortunately).

Think back to the description earlier of consistency as it relates to the CAP Theorem: If our database is to be consistent, all users accessing the data layer from any endpoint, at any point in time, should have a consistent view of the data it stores. That is to say, if two users are checking their shared bank account balance, the amount of money belonging to Bob should agree with the amount of money that Alice sees inside of the same checking account. However, because of the nature of our distributed system, it may take time for example, should Alice withdraw $10 from the account, for Bob to see his shared account has dropped in value if Alice is accessing the database in Canada and Bob is doing the same in Germany. The change needs time to propagate to every instance of the database.

The kind of scenario we are describing here is what is termed “eventual consistency”, and it is the model followed by database systems that are partition tolerant and available, to the loss of consistency. It stipulates that your data layer can no longer guarantee that all nodes (and clients connected to those nodes) share identical versions of the same data point, at a given snapshot in time. Given enough time, however, all the nodes of the distributed system will eventually share the same versioning of all data, but only after a sufficient duration needed for all nodes to replicate and synchronize to the same state.

For database administrators and users, the implications of eventual consistency may not be obvious at first glance. However, as we demonstrate, it has radical consequences for the types of data you can now store within your database; it also requires a reconsideration on the types of data you may opt to store inside of a relational database versus a non-relational database. At its core, this boils down to a consideration of whether your data layer solution needs to be rigid or flexible. Put another way: Whether your use case requires an ACID or BASE solution.

## 1.2.1 ACID versus BASE systems

The origin of these terms ACID and BASE is a bit of a chemistry joke for those who know the differences between their acids and bases. The term ACID is one we suspect you have likely heard in your conversations around databases: it is, like so many terms in our industry, an acronym. ACID: "Availability, Consistency, Isolated, and Durable" is frequently used to describe traditional relational databases (RDBMS). These individual components are fairly self-descriptive and harken back to the conversations we had earlier about the CAP Theorem. Essentially, they guarantee that interactions with a transactional database will not have unexpected side-effects and can be relied upon to commit operations to and from the disk.

Where things become more interesting is with the rise of distributed systems, mentioned before, that prioritize availability and partition tolerance over the consistency of the data stored within such systems. In this case, it is no longer possible to describe these architectures as ACIDic. In keeping with the tradition of acronyms and to serve as a counterpoint to ACID, the term BASE was devised to describe systems that are "Basically Available, Soft State, and Eventually Consistent". (We concede that the name is slightly arbitrary, but the chemistry of acids and bases is such that the two neutralize one another and exist at opposite ends of the pH spectrum).

The first tenant, "Basically Available", is the premise that the system does not necessarily guarantee availability at all times: that is to say, partitions in the database may arise and not all nodes of a distributed system can be guaranteed to be online at any point in time.

The second, "Soft State", describes that the state of the distributed system may change from moment to moment. Put another way: these systems are prone to volatility (at least more so than the type of architecture you would require from a transactional RDBMS). This is connected to the Basically Available premise as well, since the expectation with distributed systems is that parts of the cluster are susceptible to failure at any time. We do not feel that this is a weakness. In fact, it can be prudent to assume that failures will happen, so as not to be taken by surprise when they do.

The final term, which we have seen before, "Eventually Consistent", reflects the expectation that not every instance or view into your database will be entirely consistent with the most up-to-date version. However, given enough time, the cluster will achieve a consistent view eventually.

Interestingly, for database architectures that prioritize availability and partition tolerance, they have the unique characteristic of being ACID-compliant at the node level: individual nodes in the cluster demonstrate availability, consistency, isolation, and durability. However, looked at in aggregate, across all of the nodes of a cluster, we can say that the cluster of nodes is a BASE system.

We covered quite a bit of material so far, but we are confident that the stage has been set for our most important discussion of all: answering the question of "What is NoSQL?". We have defined: the attributes of relational systems; the CAP Theorem and how this influences the characteristics and use cases for database systems; and finally the emergence of database architectures that prioritize availability and partition tolerance over consistency. What has emerged from this strategy is a differentiation between ACID and BASE database system architecture. Essentially, this divergence represents the split between relational and non-relational. Or put another way: the beginning of the NoSQL paradigm shift.

## 1.2.2 What is NoSQL?

NoSQL, “No SQL”, or “Not Only SQL”, started off by defining itself in contrast to SQL. Popular urban legend suggests that the term emerged from a technical professional meet-up in California, where the name NoSQL was coined the night before by a developer short on ideas for how to set apart their presentation for other SQL-focused talks. NoSQL, therefore, first identified itself by describing precisely what it is not. Over time, this meaning has changed to become more nuanced. More importantly, the sentiment and meaning of the name, “No SQL” or “Not Only SQL”, changes according to the developer or database practitioner you are speaking with.

As we explore the facets of NoSQL in this chapter, we believe you will find that the meaning of the word (and the technology it describes) becomes more nuanced the deeper your understanding becomes. The distinct line that NoSQL aimed to draw in the sand has, with time and a shifting database landscape, continued to blur.

So what is NoSQL? It is not a relational database: that seems intuitive enough. Like the BASE systems we described before that favor partition tolerance and availability over characteristics like consistency, NoSQL databases do not follow the relational model. In particular, they are designed to work over distributed systems and clusters, unlike relational databases. For that reason, they are typically employed for use as the data layer drivers of 21st century web applications. This is owing to their scalability and distributed architecture, of course; however, the flexibility of NoSQL database schemas means that they are more aptly suited for persisting the types of unstructured or semi-structured data often generated over the web today.

Most importantly, NoSQL’s key distinguishing feature over relational databases is that RDBMS is ACID-compliant (atomic, consistent, isolated, and durable) whereas NoSQL is a BASE system. With diminished emphasis on consistency of data, NoSQL is able to offer greater flexibility in terms of data schema. Instead of rigidly enforced relational schemas, non-relational databases like Cloudant can use JSON as their data interchange format. In turn, this allows for representation of a much wider variety of data types. For this reason, NoSQL is sometimes mistakenly described as a “schema-less” approach to data persistence. We say that it is inaccurate because such a name implies that there is no schema for JSON documents. This is not the case: instead, every JSON document has a unique schema (your data schema is defined on a per-document basis).

The upshot of this is that your schema can be modified document by document, without the need to lock down the entire database such as would be required with a relational data store. NoSQL document stores like Cloudant have no concept of locking, in fact. We explore more of these varieties of NoSQL data stores in a moment. For now, it is sufficient to understand that NoSQL data stores are “schema-flexible” instead of being “schema-first” (which is typical of rigidly enforced RDBMS schemas). Having no “schema-first” requirement lends itself well to the rapid and agile development strategies often practiced by developers for web and mobile applications, which favor the use of NoSQL databases as their data layer backbone.

In terms of performance and efficiency, NoSQL databases tend to be much more elastic, with regards to scale out and scale down, than relational systems. Having an elastic and scalable database architecture enables your organization to rapidly scale up your database to multiple racks and potentially hundreds of servers. NoSQL systems generally support more dynamic workloads, and are amenable to distributed analytics because of the distributed architecture of the data layer.

JSON documents and web-based APIs leveraged by many varieties of NoSQL databases translates to lower operational data management costs. All that is required to access your data and interact with your data layer are low-latency, low-overheard API calls. As a result, adopters of NoSQL find that their organizations are able to increase the volume of data processed by their data layer, all atop commodity hardware or pay-as-you-go infrastructure that is provisionable over the cloud. All built around an architecture that is fault-tolerant and highly available.

NoSQL databases follow relaxed consistency models that we have described as being "eventually consistent". This describes a willingness on the part of NoSQL adopters and developers to trade consistency for higher availability. As mentioned previously, these trade-offs exist on a spectrum. You can tune between consistency and availability over partitioned (or distributed) systems; it is never an all-or-nothing proposition, where a system guarantees consistency of all data but is never available. (A database that is guaranteed to never be available does not sound like an enticing solution).

Lastly, NoSQL database systems appeal to a wide audience of developers because of the low upfront cost for adopting the software. Typically, NoSQL databases are available under an open source license. In comparison to some of the costly relational database licenses available in the marketplace, NoSQL's open source approach certainly appeals to financially constrained developers. However, it is important that adopters of such technologies remain weary of the hidden costs associated with "free" software: maintenance, patching, monitoring, and scaling of your NoSQL database solution (if not managed by the vendor for you) can require having expensive database administrators (DBAs) or skilled technical experts on-hand.

The hidden costs associated with open source software may deter weary adopters, but that is not to dismiss the managed NoSQL data layer solutions such as Cloudant. We cover the immense value of a fully managed, database as a service solution in the following chapters. And perhaps the biggest elephant in the room that we have yet to mention, but is a major driving force behind NoSQL adoption, is the desire for developers and programmers to embrace an alternative to SQL. As we will see, the use cases of NoSQL and traditional SQL databases can vary significantly. There are also emerging technologies such as hybrid databases that aim to combine the best attributes of NoSQL with those of transactional stores.

However, before we delve into the future of NoSQL, let us take a moment to survey the NoSQL landscape as it stands today.

### 1.2.3 NoSQL database landscape

*Key-value (K|V)* stores represent an associative array of keys, mapping to a value. The objects stored within them are opaque to the user interacting with the database. That is to say, objects can only be referenced by their keys (as a look-up), and not directly referenced by their values. Key-value stores are stored internally as a hash map, and therefore are highly performant. The relationship between keys and values are fairly straightforward, and therefore developers can interact with such data with even the most simplistic CRUD operations (create, read, update, delete). Complex relations and interconnections between data points that exist in alternative NoSQL implementations are not a concern for key value stores. However, they are poor at handling complex queries and aggregations of mixed data types. As we will see, there are NoSQL varieties, such as document data types of Cloudant, that are ideally suited for mapping these more objects and data structures.

A second version of NoSQL database, *Columnar*, is derived from Google's BigTable papers on distributed storage systems. Columnar databases exhibit the type of flexible ("schema-less") schema that is typical of many NoSQL databases. Users and database administrators are able to define (in an ad hoc manner) as many columns as they want within a given row of a database table. For this reason, rows belonging to the same table do not necessarily exhibit identical columnar properties (in terms of the number of columns, as well as the number of columns). The name “BigTable” and its origins with Google would seem to imply that this type of NoSQL database is well suited for handling large volumes of data, and you would be correct for thinking so. Columnar databases are frequently leveraged when handling high volume, low latency writes to disk. They consist of “column families”, which are addressable by a combination of a row key and column-specific family name. In essence, a column family is a combination of columns that fit together. Since a Columnar database can store millions of rows, and the process of adding new columns to a row is relatively inexpensive, it is possible for Columnar tables to remain sparse without incurring a storage penalty cost for NULL values (which often plague relational databases).

*Graph databases* represent one of the least-used varieties of NoSQL databases. However, that is not to suggest it is without merit. Graph databases excel at dealing with highly interconnected data. In fact, one of the unique strengths of Graph databases is that you can trace the relationship between data points by tracing the linkages between data nodes. Graphs are composed of nodes; linkages between these nodes are called “edges”. The schema of a graph database is easily modified by simply adding new nodes and defining edges between new and pre-existing nodes. The self-describing nature of Graph databases allow developers and administrators to query interconnected data efficiently, which is a great boon for users looking to persist and query their data in new and innovative ways. However, they are equally susceptible to over complication. The more nodes and edges belonging to a graph, the more challenging it becomes to tease apart and trace relationships. If we consider again our first point that Graph databases represent one of the least used varieties of NoSQL databases in the marketplace today, we feel that slower adoption of Graph speaks to their complexity and unique style of persisting data, but not necessarily because of their lack of utility. In fact, we feel that as this style of NoSQL database matures and skill sets build up around it, Graph databases will be leveraged as a data layer for powering applications in new and exciting ways that were not possible previously.

The fourth version of NoSQL databases, and by far the most popular, accounting for what we estimate to be over 50% of the NoSQL marketplace today, are *Document data stores*, such as IBM Cloudant. Document data stores are represented in JSON, or JavaScript Object Notation, and have the characteristic that every document is itself a complex data structure. Documents can contain nested structures of various data types (we cover these shortly in the section on JSON), including other objects. Users of a Document database are able to query into these complex structures, retrieving or updating portions of the document (or the entire document) without the need to lock down the database to do so. Documents are stored and retrieved via a primary key that is unique to every document (similar to the key in a key-value store). As you will see in the following chapters, documents in Cloudant also store a revision identifier in order to perform conflict resolution and versioning control for distributed systems. This structure lends itself to persisting objects from the real world inside of your database in an intuitive and descriptive way. Rather than shredding every field of a business card, your name, address, phone number, and so on, into multiple columns across a relational database row, it is possible with JSON and Document databases to represent that business card as a single object of nested fields. This translates much more intuitively between our understanding of the object as it exists in the real world, and how it is modeled and persisted within our NoSQL database.

This is by no means an exhaustive list of every variety of NoSQL databases that exist in the marketplace today, but it is a comprehensive list of its largest players, with document data stores like Cloudant clearly leading the pack. We would not be surprised if the NoSQL landscape looks radically different at the time you are reading this paper, given how quickly this area is evolving and with the trend towards emerging hybridizations between RDBMS and NoSQL.

The key to a successful NoSQL database application is correctly fitting the data layer solution to the appropriate problem. Key-value stores, for example, are best applied when data is not highly related. In scenarios where all that a developer requires are basic create, read, update, delete operators to interact with their data, a key value store should be the obvious first choice. For document stores: schemas do not need to be tightly defined or rigorously enforced. The schema of your data later is likely to be subject to frequent change and revision; and your data layer needs to remain flexible. If handling big data or sparse data, where compression or versioning is required, a Columnar database is aptly suited. Finally, Graph databases are best applied to use cases where highly interconnected data can be modeled for tracing relationships between data points.

### 1.2.4 JSON and schema-less model

As described earlier, JavaScript Object Notation (JSON) is a serialized form of JavaScript objects and has rapidly become the de facto lightweight data interchange format in use on the web today. It was a minimal, portable, and textual subset of JavaScript. Consequently, JSON's rise in popularity and adoption by web and mobile developers have essentially made JSON the most commonly used means for exchanging data between programs written in nearly even modern programming language.

JSON's power lies in its self-describing and easy to read (by humans and machines alike) schema. Associative arrays called objects are less verbose than XML-based equivalents, and their nested structure more intuitively persist and model objects from the real world within your database. There are various different data types that can be stored using JSON, including: strings, numbers (integers, real and scientific notation), arrays (ordered sequences of values, including nested arrays), and complex objects (nested JSON data types), Booleans, and Null values.

Part of JSON's immense appeal with developers is because its schema can be rapidly evolved without intervention by database administrators. Developers typically resist solutions that require delays in order to synchronize up with DBA change windows. JSON offers a simple and elegant model for persisting Java or JavaScript objects (thanks to the tight affinity that JSON has with JavaScript). This allows developers to construct a data layer solution without the need for heavy-weight persistence solutions such as OpenJPA or Hibernate. Finally, there is an elegance to JSON's data modeling that relational databases cannot replicate: storing a single JSON document to represent a real-world object is much more intuitive than storing N numbers of rows in a relational database as a normalized object.

Developers and applications continue to shift towards JSON as these audiences recognize the potential and power of the NoSQL paradigm. As we explore in this paper, new Systems of Engagement, powering web and mobile applications, that continue to flood the marketplace have less stringent ACID requirements than applications that emerged in decades past. Instead, these new mediums favor speed and flexibility over consistency, an ideal fit for easy-to-manage, scalable, and configurable NoSQL database solutions. Mobile and web applications require solutions that offer near-continuous integration of application changes for agile development strategies, at a pace with the ever growing "Internet of Things" and the data economy driving today's industries.

NoSQL databases continue to be the data management solution standard of choice for supporting applications operating at web scale. JSON document stores deliver greater data flexibility, scaling to large concurrent users and data volumes, and ease of data movement. In the next chapter, we demonstrate how IBM Cloudant delivers a scalable, global, fault-tolerant data layer, provided as both a fully managed cloud service and as an on-premises product, for enabling companies to develop next-generation Systems of Engagement.

2

# Build more, grow more, and sleep more with IBM Cloudant

Many competing database vendors that have adopted the term of “Hosted” or “Managed” Database Services and misleadingly market it as a catch-all for truly managed databases. In reality, these “hosted” data layers are only one step removed from “Roll-Your-Own” (do-it-yourself) practitioners. Granted, the infrastructure layer is provisioned and set up by the vendor; however, the remainder of the data layer’s database stack administration, configuration, and maintenance of software remain the burden of the customer.

These vendors will openly market themselves as a “managed database service”. As a simple analogy, such companies deliver the car and the keys to the ignition. It remains the sole responsibility of the customer, however, to drive their machine and know how to get to their destination. In fact, keeping with the car analogy, the client has to fill the tank with gas, change the oil, schedule time to do a recall on any defects, and more.

In the same way that Google’s self-driving cars handle everything along the way between your point of origin and final destination, IBM’s fully managed database as a service (DaaS) with IBM Cloudant handles the complete database stack (from provisioning, to administration, to scale-out and growth) for the customer “as-a-service”. Cloudant unburdens the enterprise customer from the mundane tasks of database administration, eliminates the need for having expensive DBAs trained and available in-house, and allows developers of mobile-first and born-on-the-web apps to focus entirely on building better applications with a data layer solution that is managed for them.

Cloudant represents a strategic acquisition by IBM that extends the company’s Big Data and Analytics portfolio to include a fully managed, NoSQL cloud service. Cloudant simplifies the development cycle for creators of fast-growing web and mobile applications, by alleviating the burdens of mundane database administration tasks. Developers are then able to focus on building the next generation of systems of engagement, social and mobile applications, without losing time, money, or sleep managing their database infrastructure and growth. Critically, Cloudant is an enterprise-ready service that supports this infrastructure with guaranteed performance and availability.

Built on the open source database Apache CouchDB, the Cloudant fully managed DaaS enables applications and their developers to be more agile. As a part of its data layer, customers have access to multi-master replication and mobile device synchronization capabilities for occasionally connected devices. Applications can take advantage of Cloudant advanced real-time indexing for ad hoc full text search via Apache Lucene, online analytics via MapReduce, and advanced geospatial querying. Mobile applications can leverage a durable replication protocol for offline sync and global data distribution, as well as a geo-load balancing capability to ensure cross-data center availability and optimal performance. The Cloudant RESTful web-based application programming interface (API), flexible schema, and capacity to scale massively are what empower customers to deliver applications to market faster in a cost-effective, DBA-free service model as shown in Figure 2-1.

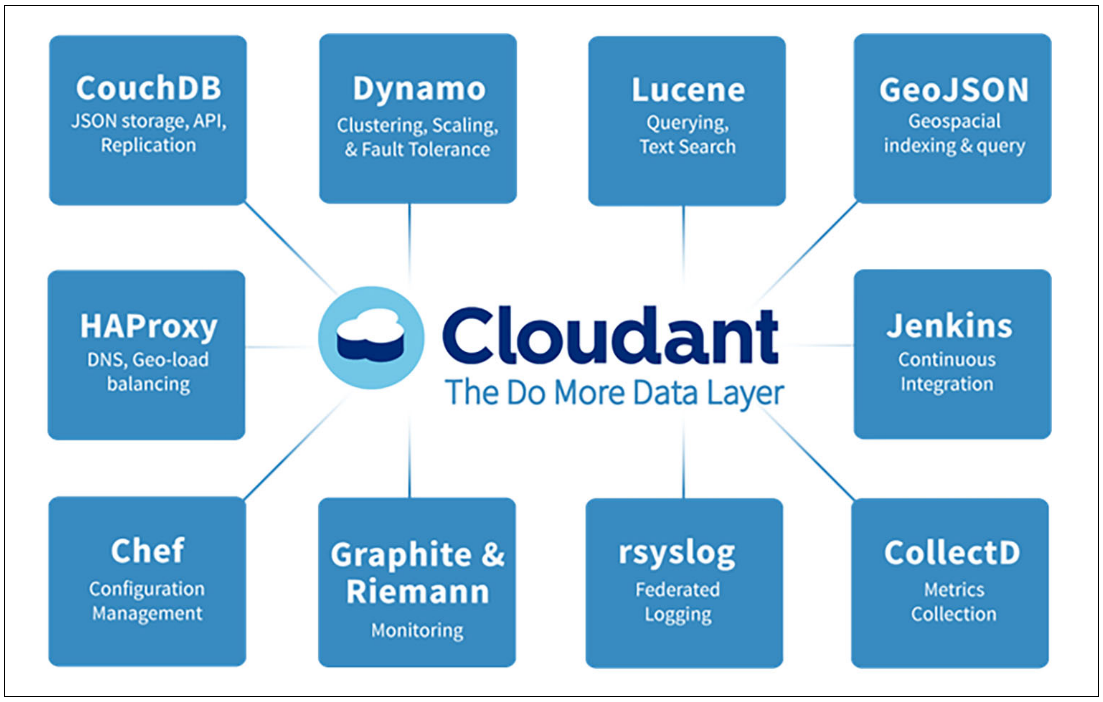


*Figure 2-1 Cloudant delivers the most flexible, scalable, and always-available solution for developers of big mobile and "Internet of Things" applications, via a fully managed DaaS*

## Did you know?

The NoSQL data management market is burgeoning: forecasts expect the value of the market to grow to USD $14 billion 2014 - 2018. Additionally, over 50% of NoSQL solutions use JavaScript Object Notation (JSON)-based document data stores (including Cloudant).

The rise of the traditional database store, which we will refer to as relational databases, harkens back to days where query flexibility was valued more over having schemas that were flexible. For the time in which they were developed, and throughout the last 40 years or so, this trade-off simply made the most sense from a business and use-case perspective. The expectation (and often the norm) was that large organizations would have database administrators (DBAs) in-house and on-hand for managing schemas to fit your company's data. Issues of complexity and fine-tuning required in order to keep a relational solution viable are secondary to how performant and specific the queries you run on your data can be, particularly when there are highly trained DBAs maintaining the data layer for your business. This continues to be the case with many of the enterprise customers we see today.

If this continues to hold true for much of the industry today, then you might be wondering what is the "evolution" in the database landscape. Before we begin with our examination of Cloudant, it is important that we take a moment to look at the development of this new paradigm of databases, the driver behind this shift in the data persistence models, known as

“NoSQL”. The best way to broker that conversation around “What is NoSQL?” is to look at its counterpoint: SQL, the standard of relational databases.

## 2.1 Business value

It is clear that the momentum behind big data, mobile, social, Internet of Things (IoT), and cloud initiatives are transforming the modern IT profession. We are already seeing an explosion of new products and applications developed for these platforms, and the expectation is that these products will experience ever-increasing consumption by established and emerging markets. Agility and elasticity are key in the mobile application environment, where being able to rapidly scale up performance is necessary to accommodate fluctuations in usage and load on your infrastructure. In the mobile space, the demand on these systems can change on a day-to-day basis, and it is necessary for developers to have a solution in place that allows them to scale up without having to scale their internal resources as well. Web and mobile application databases, and their administrators, must be prepared to face issues of cost, performance, scalability, availability, and security head-on; all of which carry uncertainty and risk.

IBM Cloudant eliminates this complexity by enabling developers to focus on building next-generation applications without the need to manage their database infrastructure or growth. A fully managed cloud data layer service, Cloudant guarantees its customers the high availability, scalability, simplicity, and performance that modern web and mobile applications demand. The scalability of a fully managed cloud DaaS solution simplifies the application development cycle and offers Cloudant customers greater agility for launching new products or responding to an ever-changing market: build more, grow more, and sleep more.

Cloudant provides the following benefits:

- Offers a NoSQL data layer, delivered as a fully managed service. Liberates developers from the cost, complexity, and risk of do-it-yourself data layer solutions.
- Monitored and managed 24x7 by Cloudant big data and database administration experts.
- Top-tier security, backed by IBM, for fine grained authentication and authorized access to your data.
- Leverages self-describing JSON “document” storage schemas to allow for flexible and agile application development.
- Mobile device and web replication and synchronization support for offline and occasionally connected devices.
- Built using a master-master (also known as, “master-less”) clustering framework that can span multiple racks, data centers, cloud providers, or devices.
- High availability and enhanced performance for customer applications that require data to be local to the user. Supplied by global data distribution and geo-load balancing technologies.
- Delivers real-time indexing for online analytics, ad hoc full-text search via integrated Apache Lucene, and advanced geospatial querying.
- Supplies a RESTful API for ease of access and compatibility with developers that live and work on the modern web.
- Based on open standards including: Apache CouchDB, Apache Lucene, GeoJSON, and others.

## 2.2 Solution overview

The data layer solution offered by IBM Cloudant outshines other NoSQL database competitors by delivering a fully managed cloud service that is always-on, fast, and scalable. Cloudant provides all the database administration for their client's applications, providing a fast-growing and scalable framework on which clients are able to focus purely on development of their next generation of applications.

IBM Cloudant provides database solutions tailored to address the following scenarios for its customers:

- Inadequate database performance is (or has the potential down the road) to hamper user base or business growth.
- Unreliable service availability has negatively impacted user experience or resulted in lost revenue opportunities.
- Require access to application features and data on sometimes-offline (mobile) devices, where network connectivity is poor or unavailable.
- Seeking to perform embedded analytics on customer data, full-text search, and geospatial querying: all within a single database.
- Storage solutions need to leverage “variable” or multi-structure JSON data for maximum schema flexibility.
- No in-house database administration solutions; company does not want to hire DBAs.

Developers of mobile and web applications can host their business on a global network of service providers, including: IBM SoftLayer®, Rackspace, Microsoft Azure, and Amazon Web Services. Regardless of the service provider, the data layer of Cloudant gives assurances that a company's underlying services are fully supported by a scalable and flexible NoSQL solution. Full-text search, advanced analytics technologies, and mobile data replication and synchronization further extend how clients interact with and leverage their data. For this reason, Cloudant typically targets verticals in the areas of: online gaming, mobile development, marketing analytics, software as a service (SaaS) companies, online education providers, social media, networking sites, and data analytics firms. Figure 2-2 on page 17 shows how the IBM Cloudant fully managed database as a service solution fits into the database market.

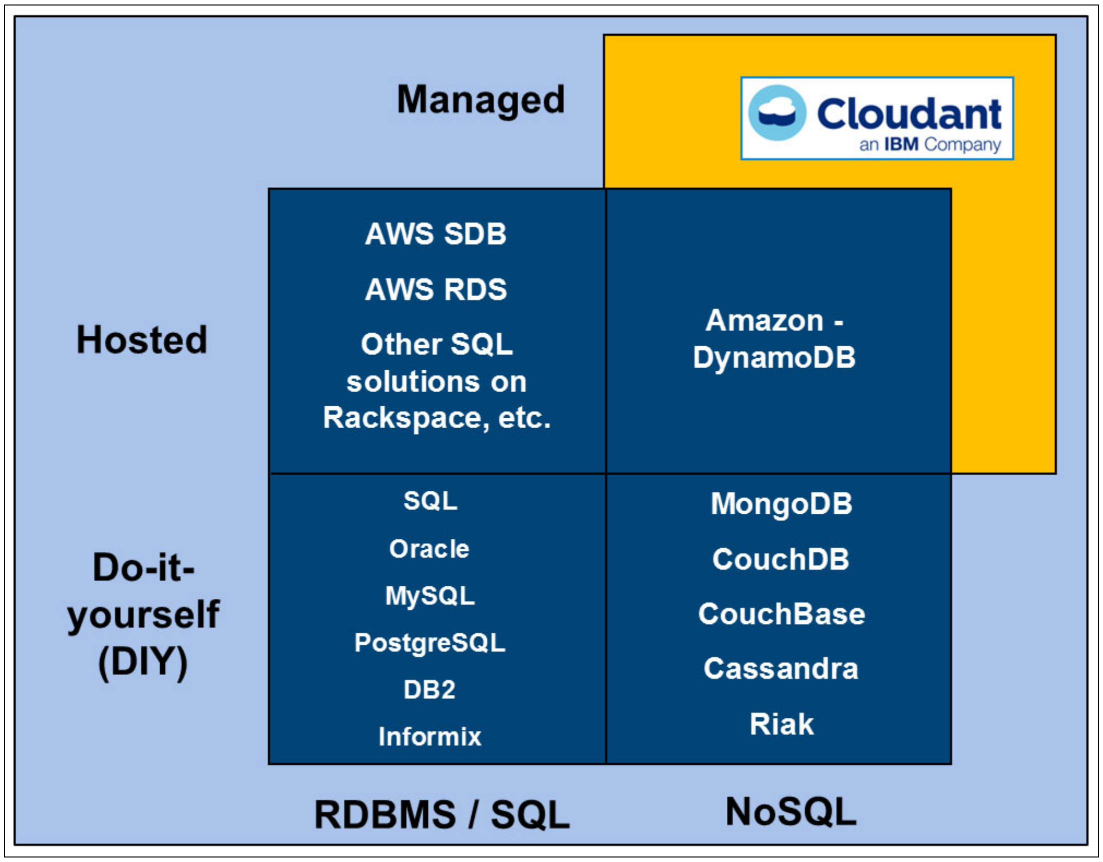


*Figure 2-2 How the Cloudant fully-managed DaaS solution fits into the database market*

**Note:** At the time of this writing, the Cloudant service grants customers access between four major infrastructure as a service (IaaS) providers, across 35 data centers, in 13 countries around the world (and growing). IBM Cloudant allows for total flexibility in document and database design, as well as geo-location choices, to ensure maximum control and security over customer data. Cloudant is able to scale out these deployments up to millions of databases.

Furthermore, you can instantiate individual databases to isolate data on an individual user level. This combination of scaled development and deployment across geospatial locations, as well as partitioning of data across individual databases, enables the customer to isolate and tightly control how data is persisted in the network.

Figure 2-3 on page 18 shows the IBM Cloudant NoSQL database, services, and API layer view.

## 2.3 Solution architecture

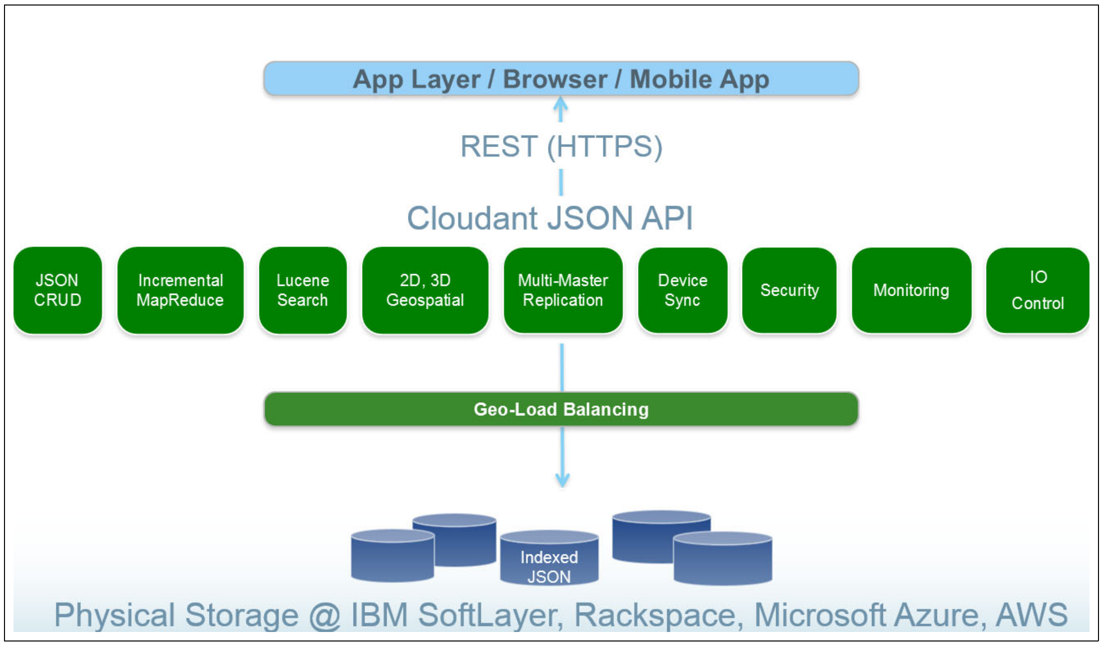


*Figure 2-3 The IBM Cloudant NoSQL database, services, and API layer view*

**Note:** Notice in Figure 2-3 the fully integrated capabilities that are inherent in the Cloudant API and available without the need for third-party integrations. IBM Cloudant was designed to reduce the complexity of tasks and services that developers otherwise would be required to manage themselves: Complete synchronization and geo-load balancing features are tuneable within Cloudant toolsets, and its replication API is consistent across both Cloudant and CouchDB. No additional services or components between your device and storage endpoints are required to take advantage of these features. The upshot: being able to source database synchronization activity to only particular subsets of your data reduces network load and increases performance via targeted, geospatial-specific synchronization tasks.

The following key HTTP API features set Cloudant apart from its competitors:

- JSON document create, read, update, delete (CRUD) options
- Primary indexing
- MapReduce-built secondary indexes
- Full-text search

JSON is a lightweight data interchange format that has become the de facto data interchange format on the web because of its language independence and self-describing data structures. Data representation and structure can vary from document to document. This schema flexibility allows you to describe all the aspects of data (in any formatting that you might encounter); moreover, JSON allows you to avoid the use of NULL values, such as you would find in relational databases. Consequently, IBM Cloudant can be described as a “flexible schema” approach to data storage: This is not meant to imply that there is no schema; rather that the schema varies across subsets of documents and their data. No database downtime or table locks are required to alter a single document’s schema, and because of this Cloudant is aptly suited for scenarios where database schema flexibility is key. Cloudant comes embedded with various real-time indexing options to query your data.

Secondary indexes built via MapReduce, also known as *views*, are ideal for searching for secondary keys or ranges of keys, and for doing heavy online analytics. Search indexes built using Apache Lucene are excellent for performing ad hoc or full text search; additionally, Cloudant indexing supports search facets, groups, as well as search by both distance and bounding box. Finally, advanced geospatial indexing allows for querying against complex structures such as polygons and calculating advanced relations like overlap or intersection. All of these features are accessed by a RESTful web-based API, which is natural and intuitive to programmers familiar with developing for the web.

Figure 2-4 shows how the sole concern for customers of a DaaS solution is the design and development of their application.

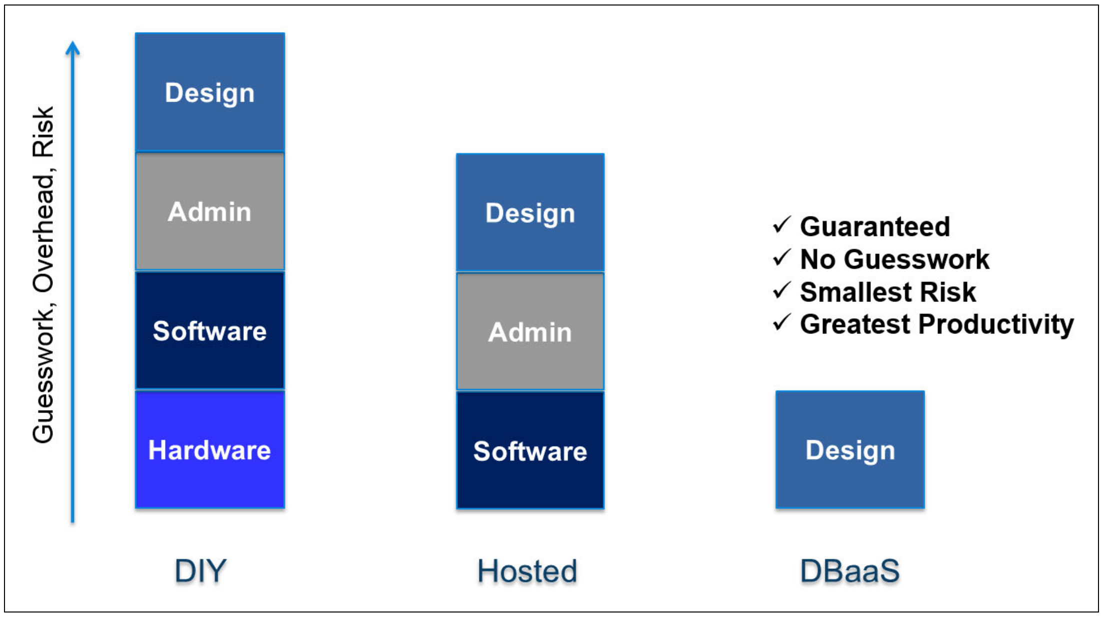


*Figure 2-4 The sole concern for customers of a DaaS solution is the design and development of their application. IBM Cloudant guarantees availability, eliminates risk, and ensures that service is able to scale out as clients (and their applications) grow*

For modern web and mobile applications, the speed of deployment onto a database is critical: Databases need to adequately support usage requirements, scale (and downsize) rapidly, and provide high availability. A distinguishing feature of Cloudant is the delivery of these necessary services and rich portfolio of proprietary tools via a cloud-distributed, DaaS solution. Unlike do-it-yourself (DIY) solutions, which ask developers to handle everything from provisioning the hardware to database administration at the top of the stack, DaaS handles cloud database provisioning, management, and scaling as a paid service to the customer as depicted in Figure 2-4. The client receives guaranteed availability and reliability of their business, hardware provisioning that can grow elastically as required, and a rapid time to value with the greatest mitigation of risk. Hosted services simply cannot claim to offer the same degree of comprehensive services: hosted cloud solutions provision hardware and instantiate an image, then turn the keys over to you, the developer. Only a fully-managed service like IBM Cloudant can liberate developers from the burdens of database administration and allow clients to focus their energy on what really matters: building the next generation of web and mobile applications for their customers.

## 2.4 Usage scenarios

Consider the scenario of a small start-up developer for mobile and web-browser games with a problem: the run-away success of their newest mobile game. This company is of modest size and equally modest budget, with a game that is named a "Featured App" in the App Store the day of launch. The unexpected success of the application required the studio to rapidly scale out their service in order to accommodate the increasing demand for the application. At present, this development studio and their application are at a stand-still; the load on their inadequate database architecture is so great that the mobile game has become unusable. Despite their best efforts to prepare for scaling, including a soft launch of the game, no one on the team has ever faced such an onslaught of users. Negative customer reviews are piling up and the company's ability to conduct business is gridlocked. Without experts in database administration on-hand to support the product's demand, the financial repercussions of this ongoing service outage could prove disastrous.

The design studio had five criteria that needed to be satisfied before they would be able to commit to a solution:

- The improved database backend would need to scale massively and elastically (up and down) in response to fluctuating demand on the App Store.
- It would need to be available non-stop so as not to interrupt the delivery of entertainment to their users around the world.
- They'd need to be up and running on it fast, while there was still a chance to capitalize on the initial popularity of the game.
- The solution would need to be managed. Hiring DBAs did not make sense for the company's long-term objectives of developing better games for their customers.
- The solution would require improved tools and techniques for data management over the messy and frustrating relational database management systems (RDBMSs) that had been used previously.

Enter IBM Cloudant, which delivered the robust scalability needed by the game developer, who was able to migrate to Cloudant within just a few days and without hiring a DBA. Cloudant addressed challenges of availability, synchronization, and geography. Millions of new users were able to interact with the online game's world, without requiring the studio to hire (or have on staff previously) DBAs to administer the solution. Cloudant also provided monitoring of user activity and monetization analytics: tools that allowed the developer to track how customers were purchasing through in-application markets and ecosystems, as well as trace usage and application exposure. Key decision makers in this process were the company's chief technology officer, as well as the lead application development team, for whom a fully managed database as a service such as Cloudant was able to alleviate key pain points and drive new business.

As you might expect, IBM Cloudant is as equally appealing to enterprise developers as it is for small start-ups companies. Nearly any system that needs to elastically scale concurrent access to data or manage multi-structured data can benefit from Cloudant.

Some examples of enterprise success stories include:

- **Elastic Scaling** - A major consumer financial services company created a customer-facing web and mobile app for storing and sharing personal financial data. It was intended to serve as a digital safety deposit box. Providing this to customers generated better self-service and brand loyalty. However, the company did not have the experience or know-how that is needed to scale a system that would serve up to 20 million users. They chose Cloudant over MongoDB, Couchbase, and Amazon DynamoDB due to the superior service, performance, and security of IBM Cloudant.

- **Messy Data** - One of the world's largest pharmaceutical companies uses the Cloudant DaaS to stage and transform clinical trial data for a large data warehousing and analytics project they operate. Cloudant can handle the wide variety of clinical studies data (via self-describing JSON schemas and Cloudant indexing), and ability to index it incrementally as new data is loaded, reduced the time needed for data processing from 18 hours (in Oracle) to just a few minutes. It also eliminated the substantial expense of Oracle's hardware and software overhead.
- **Internet of Things** - A fitness metrics company collects data from Internet-enabled fitness devices (including mobile phones) to collect information about product usage and workout information. Users can subsequently tap into and monitor their fitness metrics online. It also collects product "health" readings to determine whether the devices collecting this data might require maintenance. The customer relies on Cloudant to handle the large volume of data being concurrently collected and read by its products and users.
- **Social learning** – A publicly held developer of desktop language learning software wants to deliver their software as an online service. Additionally, they want to enable language learners to connect and communicate with each other in order to practice their newly learned languages with other users. They use Cloudant to handle the large scale-up of course material and user data, including: connections between users, states of conversations, full-text indexing, search of curriculum and correspondence information, and more. Cloudant provided the scalability and eliminated the need to use separate databases for structured data, graph (connections) data, and full text.

## 2.5 Intuitively interact with data using Cloudant Dashboard

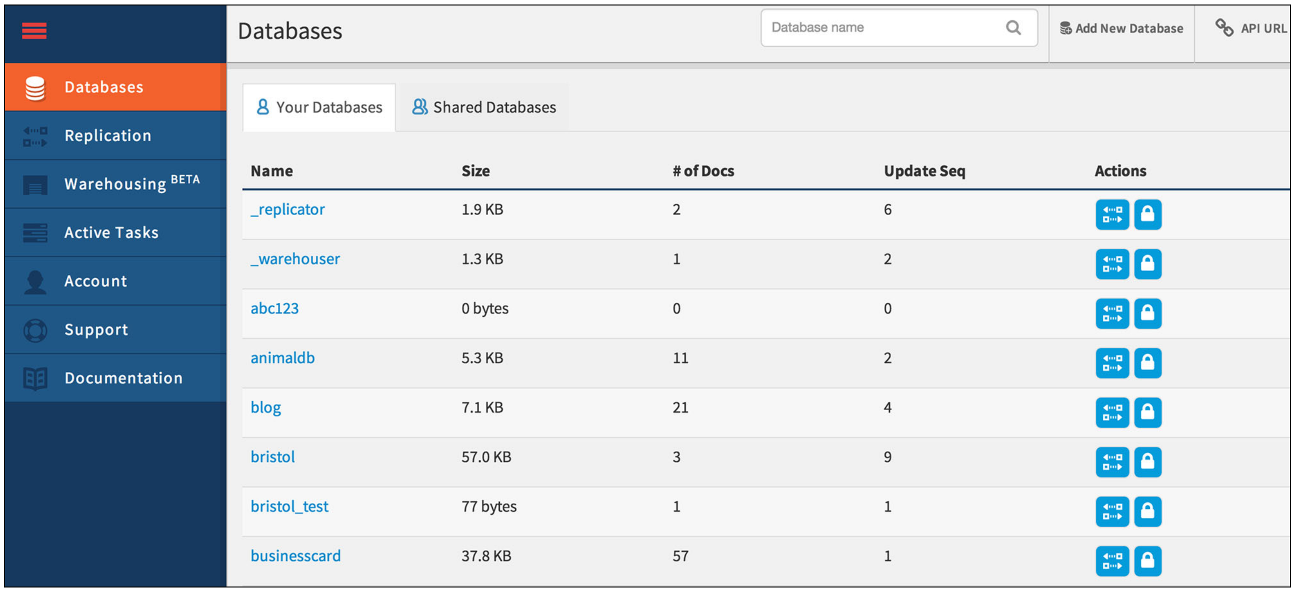
Signing up for a multi-tenant account to get started developing and building applications with Cloudant is incredibly simple. No credit card is required: You can use Cloudant multi-tenant deployment without a fee if your activity accumulates under $50.00 USD worth of charges after every monthly cycle.

An interesting fact about Cloudant is that many of the Apache CouchDB committers, which as you remember, CouchDB forms the open source backbone of Cloudant, actually work at Cloudant. They are a company that actively contributes back to the community on which they were founded, and make great efforts to commit back tools and features to CouchDB.

One of these features was the Cloudant Dashboard as shown in Figure 2-5 on page 22. The Dashboard front end was primarily built on the JavaScript backbone.js framework.

The process of creating a database via the Cloudant Dashboard, a user-friendly GUI for interacting with your Cloudant data layer, is performed by simply clicking the "Add New Database" button from within the Databases tab. Databases can vary dramatically in size, according to the size of your documents (which can individually reach a maximum size of 64 MB each). From this tab, you can also access quick action icons for tasks such as security management, access permissions, and replication jobs.

The Dashboard can be used to interact with your Cloudant database for a multitude of tasks. Keep in mind, however, that these are a subset of the complete API that is accessible through Cloudant RESTful endpoints. Users are able to create, read, update, and delete JSON documents. They can also monitor active or ongoing tasks, replicate or share databases, create virtual hosts, and manage access control at the database level.



| Name | Size | # of Docs | Update Seq | Actions |
|---|---|---|---|---|
| _replicator | 1.9 KB | 2 | 6 | |
| _warehouser | 1.3 KB | 1 | 2 | |
| abc123 | 0 bytes | 0 | 0 | |
| animaldb | 5.3 KB | 11 | 2 | |
| blog | 7.1 KB | 21 | 4 | |
| bristol | 57.0 KB | 3 | 9 | |
| bristol_test | 77 bytes | 1 | 1 | |
| businesscard | 37.8 KB | 57 | 1 | |

*Figure 2-5 Overview of databases as presented by the Cloudant Dashboard*

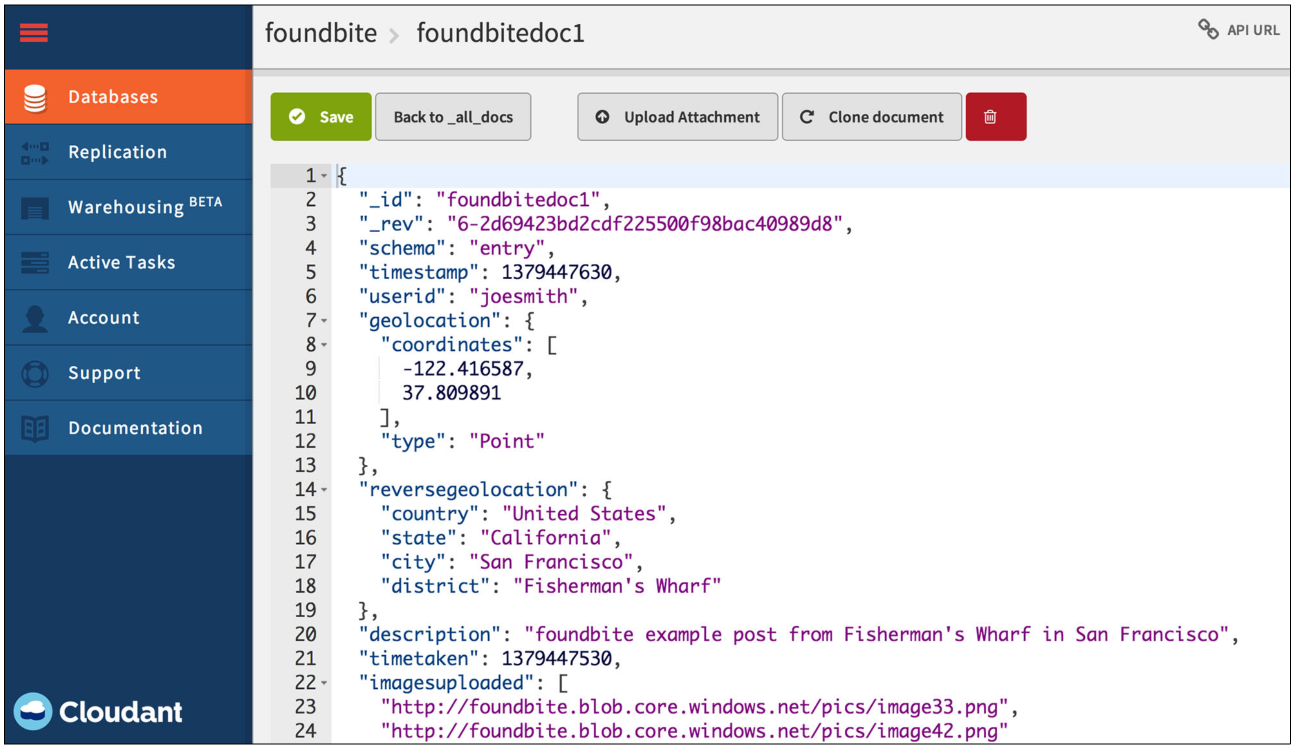
Let us briefly take a moment to describe the other tabs that users can select from the drop-down menu along the left side of the Dashboard. Replication tasks can be set up and executed from within the Replication tab. Warehousing is the newest addition to Cloudant, from which you can provision a DashDB analytics warehouse environment (which as of the time of this writing is available as a multi-tenant beta deployment only). Active Tasks is where you can monitor replication or compaction jobs, the progress towards building an index, and so on. Account is where you can track your usage, particularly: how many heavy and light API requests; how much data volume has been used; how much storage is taken up by raw data versus search indexes or secondary indexes; and so on. It is also where you can review your total metered usage and charge for that month's billing cycle.

Multi-tenant customers can request to change the location of their data from a predefined list of data centers hosted on IBM SoftLayer, Rackspace, and Microsoft Azure. Remember: dedicated clients are able to deploy to one of over 40 SoftLayer data centers, as well as database clusters hosted by Rackspace or Amazon Web Services. Opting to migrate your data triggers a support case for your account (without incurring any downtime. The DNS will carry you over to the new data center after you make that request), which is carried out by Cloudant engineers. Such requests can be submitted as either a support ticket from directly within the Cloudant Dashboard, or in response to an email request delivered to Cloudant support staff. If you are a dedicated enterprise customer, you receive a response from a Cloudant engineer within a 1-hour window. If you are a multi-tenant customer, it is within 1 day.

## 2.5.1 Editing JSON documents using Cloudant Dashboard

Customers typically use the Dashboard from an administrative perspective, but what many developers opt to do is develop and code their applications against the API programmatically. If you click the API URL button from the Cloudant Dashboard, you can see the RESTful API equivalent to the interactions you are performing via the Dashboard.

The Dashboard offers a slick and intuitive JSON editor, entirely accessible within your web browser window. You can use your own development tool if you choose. Cloudant Dashboard's built in editor will automatically validate your JSON and throw errors in response to incorrect syntax. You can define a unique schema for every JSON document, but that schema must still be syntactically correct before you can commit it to your database. This is shown in Figure 2-6.



```
{
  "_id": "foundbitedoc1",
  "_rev": "6-2d69423bd2cdf225500f98bac40989d8",
  "schema": "entry",
  "timestamp": 1379447630,
  "userid": "joesmith",
  "geolocation": {
    "coordinates": [
      -122.416587,
      37.809891
    ],
    "type": "Point"
  },
  "reversegeolocation": {
    "country": "United States",
    "state": "California",
    "city": "San Francisco",
    "district": "Fisherman's Wharf"
  },
  "description": "foundbite example post from Fisherman's Wharf in San Francisco",
  "timetaken": 1379447530,
  "imagesuploaded": [
    "http://foundbite.blob.core.windows.net/pics/image33.png",
    "http://foundbite.blob.core.windows.net/pics/image42.png"
```

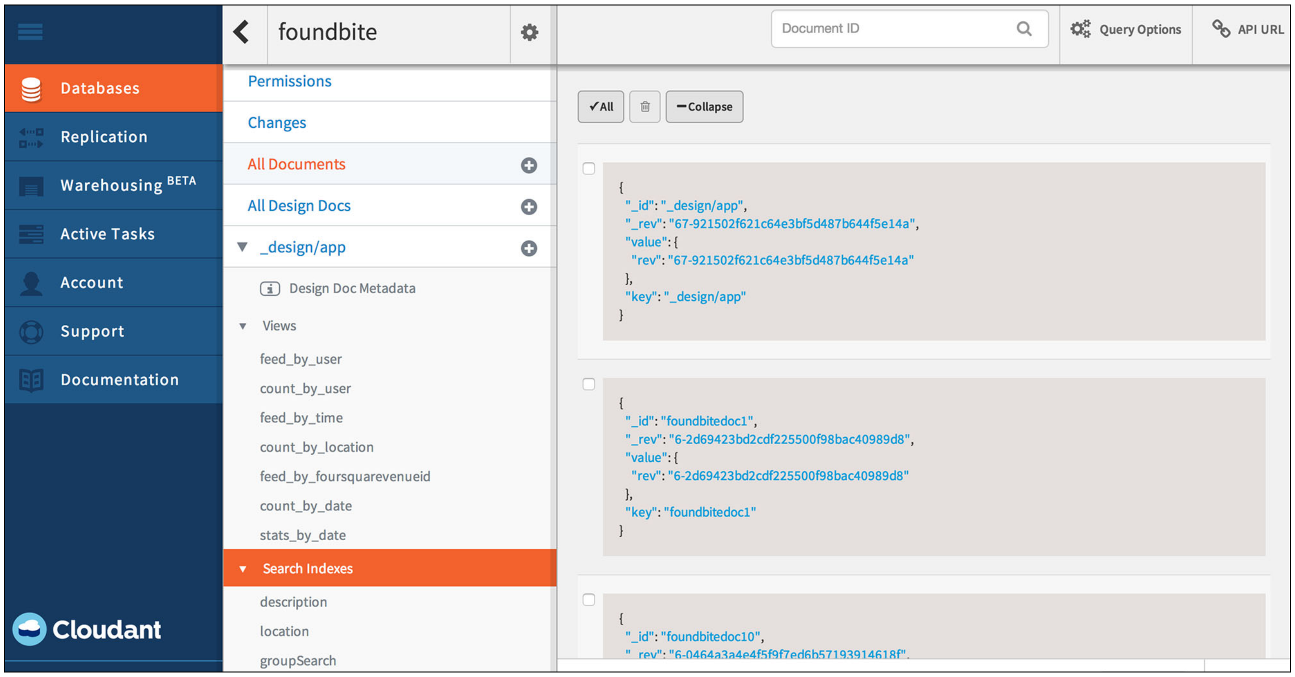
*Figure 2-6 Drilling down to examine the contents of a JSON document within the Dashboard JSON editor*

Whenever you create a document, the Dashboard populates a new universally unique identifier (UUID) within the contents of that document for you. You can leave the UUID unaltered, or replace it with your own customized ID (which must be guaranteed as unique across the entire database).

You may add to your document any information you want (remember that it is a flexible schema). When your JSON syntax has been validated and you are prepared to post the document to the database, click the “Save” button within the Dashboard. You can see that the ID is fixed now, and you are given a revision consisting of a 1 and the hash of the document contents. If you continue to add fields to the document, the integer appended to the front of the rev ID is incremented and the hash is updated (because the document contents have changed). Looking at the contents of your database via the Dashboard is equivalent to querying the Cloudant primary index: all results are ASCII sorted by their key (_id) fields. If you save a change to the document and do not change anything in terms of the content of the document, it still updates the _rev.

When updating the document via cURL (which we explore later), keep in mind that Cloudant multi-version concurrency control (MVCC) architecture requires that you supply both the _id and the _rev of the document; when editing documents via the Cloudant Dashboard, that information is supplied automatically by the editor for you.

Figure 2-7 on page 24 shows examining the views and search indexes belonging to the “foundbite” database via the Cloudant Dashboard. We will be covering the steps involved with creating and querying Cloudant indexes in the chapters to follow.



```
{
 "_id": "_design/app",
 "_rev": "67-921502f621c64e3bf5d487b644f5e14a",
 "value": {
  "rev": "67-921502f621c64e3bf5d487b644f5e14a"
 },
 "key": "_design/app"
}

{
 "_id": "foundbitedoc1",
 "_rev": "6-2d69423bd2cdf225500f98bac40989d8",
 "value": {
  "rev": "6-2d69423bd2cdf225500f98bac40989d8"
 },
 "key": "foundbitedoc1"
}

{
 "_id": "foundbitedoc10",
 "_rev": "6-0464a3a4e4f5f9f7ed6b57193914618f",
```

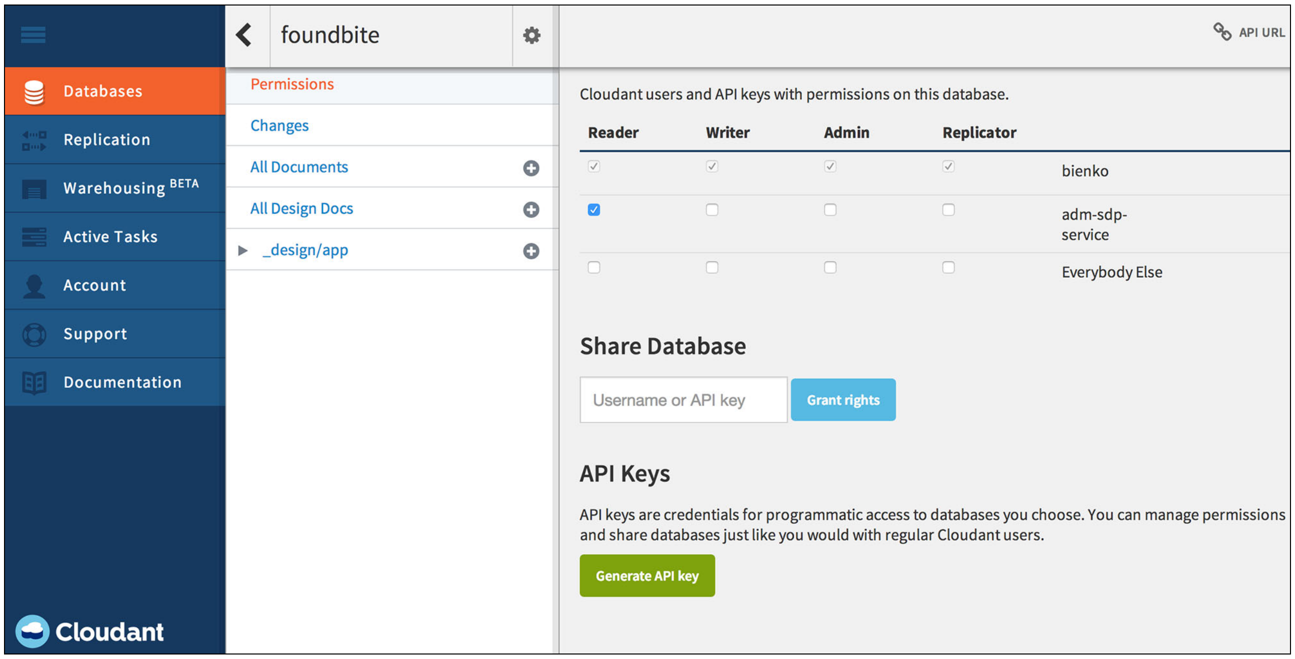
*Figure 2-7 Examining the views (secondary indexes) and search indexes belonging to the “foundbite” database via the Cloudant Dashboard*

## 2.5.2 Configuring access permissions and replication jobs within Cloudant Dashboard

You can get started right away with replicating data by going to `https://cloudant.com` and navigating to their “For Developers” resources. Within these pages, you find tutorials that explain the fundamentals of working with Cloudant, including interactive tutorials that ask you to replicate sample databases to your personal Cloudant account.

Under the Databases tab within the Cloudant Dashboard, you can change access control permissions on a per-database level. This determines which users, or the world in general, can perform tasks such as read, write, or replicate from the databases belonging to your Cloudant account.

For example, in Figure 2-8 on page 25, we navigated into my foundbite database and selected the Permissions tab. By default, your account is automatically set with administrator-level permissions for every database that you create. Setting these permissions to include Read and Write permissions for “Everybody Else” opens your database to being globally read and writeable. Changing permission settings do not require you to click “Save”; changes are committed immediately. If you want to generate a user login and password for a specific application or individual, you can generate an API key. These are created as a one-off. Therefore, it is important to save and store this information in a secure way afterward, as you cannot look it up again once generated. Cloudant users can also share databases with other dedicated or multi-tenant customers, with synchronization enabled between both replicates to allow for collaboration if the users choose to do so.



Cloudant users and API keys with permissions on this database.

| Reader | Writer | Admin | Replicator | |
|---|---|---|---|---|
| ✓ | ✓ | ✓ | ✓ | bienko |
| ✓ | ☐ | ☐ | ☐ | adm-sdp-service |
| ☐ | ☐ | ☐ | ☐ | Everybody Else |

Share Database

Username or API key | Grant rights

API Keys

API keys are credentials for programmatic access to databases you choose. You can manage permissions and share databases just like you would with regular Cloudant users.

Generate API key

*Figure 2-8 Setting access control permissions for the “foundbite” database via the Cloudant Dashboard*

## 2.6 A continuum of services working together on cloud

A defining attribute of cloud distribution models today is that cloud allows customers to completely bypass the costs and risks associated with on-premises infrastructure investments. You can burst data up to the cloud as needed and integrate with existing cloud services as required. This allows you to be much more experimental and innovative with what applications you choose to develop over the cloud, now that the risk that would have otherwise been associated with an on-premises infrastructure are now gone. Overall, we think it is fair to say that customers are expecting to consume services over the cloud, without having to make costly investments in DBAs to manage their data layer and back-end services.

One point we need to stress and highlight with our examination of cloud distribution models is how tightly integrated and unified the IBM portfolio is on the cloud today. Take a moment to examine Figure 2-9 on page 26.

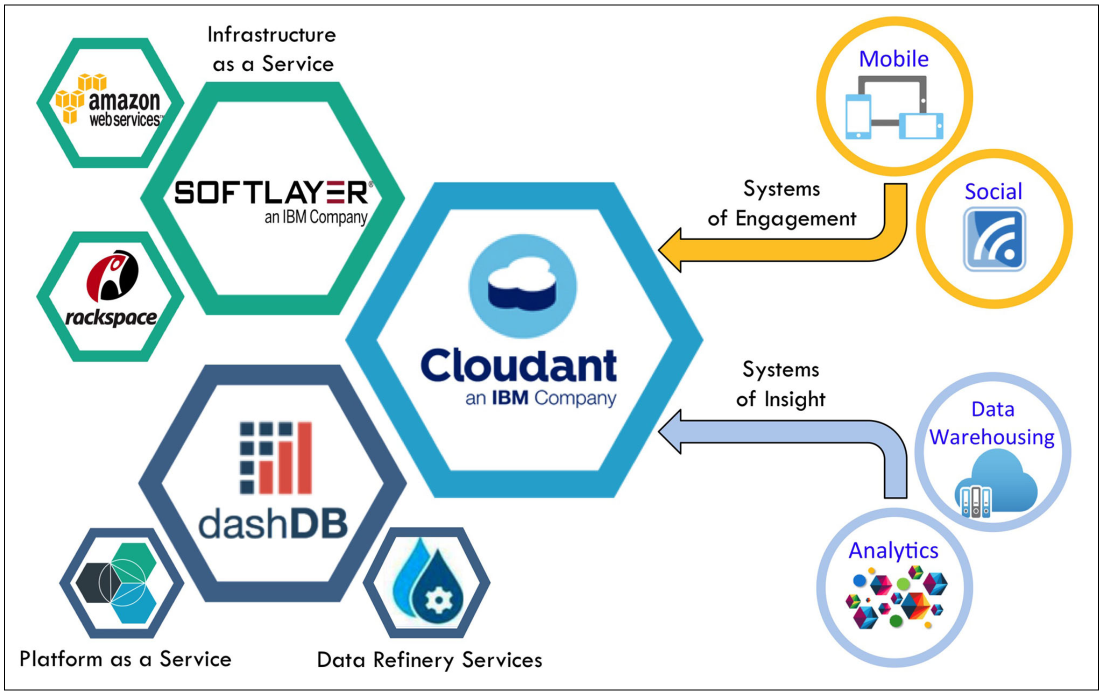


*Figure 2-9 Cloudant forms the central lynchpin between a continuum of services,* from infrastructure as a service such as IBM SoftLayer and platform as a service such as IBM Bluemix™ and IBM dashDB, *that interact with the Systems of Engagement and Systems of IBM Insight™ that power the modern enterprise cloud*

None of the competing vendors in this space are able to showcase the same holistic approach to deeply integrated and cross-service interconnectivity that IBM does. Cloudant is very much a key anchor point at the center of this cloud delivery model. One of the most significant benefits from trusting your data layer to Cloudant, beyond the managed database service, the scalability, robustness, and flexibility, is the rich tapestry of services from across the IBM portfolio you can integrate with. You can conceptualize Cloudant as the binding between application and data layers that holds this tapestry of cloud services together.

## 2.6.1 Provisioning an analytics warehouse with IBM dashDB

An exciting new entry that broadens Cloudant capabilities is what we codename “dashDB”, or what was formerly known as the “Analytics Warehouse” service on IBM platform as a service (PaaS): Bluemix. dashDB is a marriage of DB2 BLU in-memory analytics, along with advanced analytics and statistical tools from IBM Netezza® (PDA: IBM PureData™ for Analytics). Together, these services are offered as a single product, delivered and consumable as-a-service over the cloud. Its first launching point is as an integrated service alongside Cloudant, bringing together unrivalled analytics capabilities (that previously were only available to customers through on-premises infrastructure) delivered on the cloud. In four easy take-away points, dashDB is a high performance analytic database, in a “cloud easy” form factor, with best in class simplicity, and built-in security for enterprise customers. All of which is delivered as-a-service in the cloud.

In the Cloudant Dashboard, the user provisions a warehouse (a new warehouse account is provisioned in BLU). Because of the multi-tenancy model deployed, it means that we are standing up a new schema for the existing Cloudant account. An operational database in Cloudant is provisioned and called the _warehouse DB: This database is used to preserve the state of integration, and also used for monitoring and tracking the number of provisions we created (in addition to the state of every provision). See Figure 2-10.

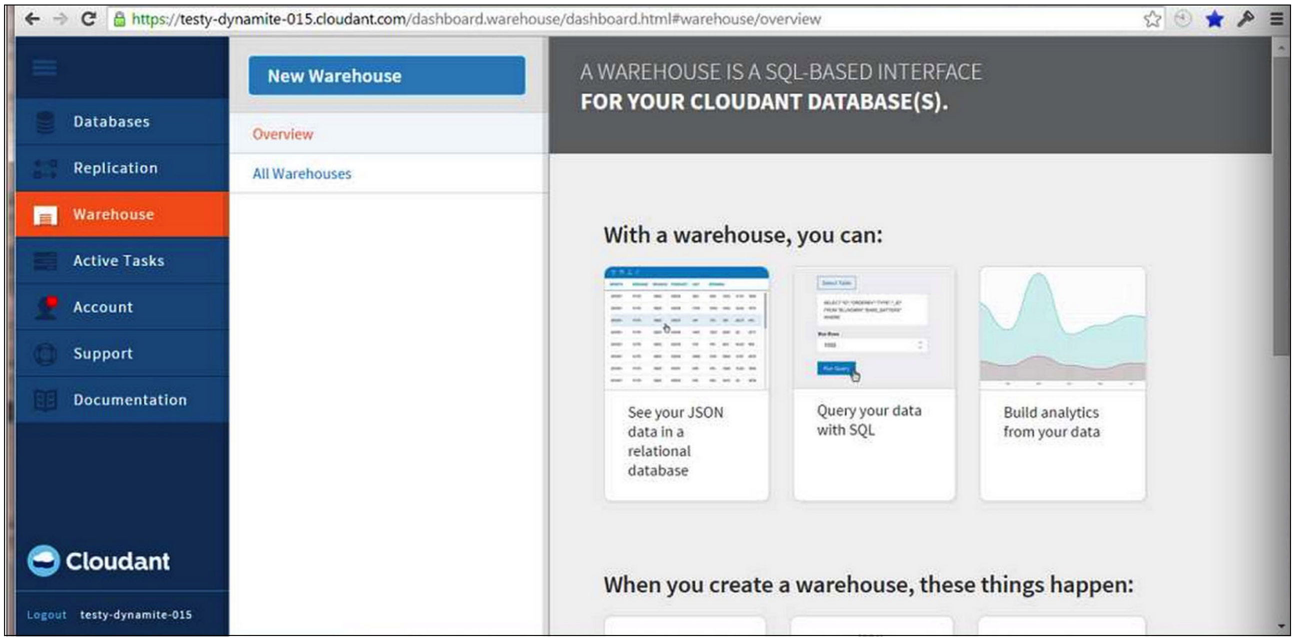
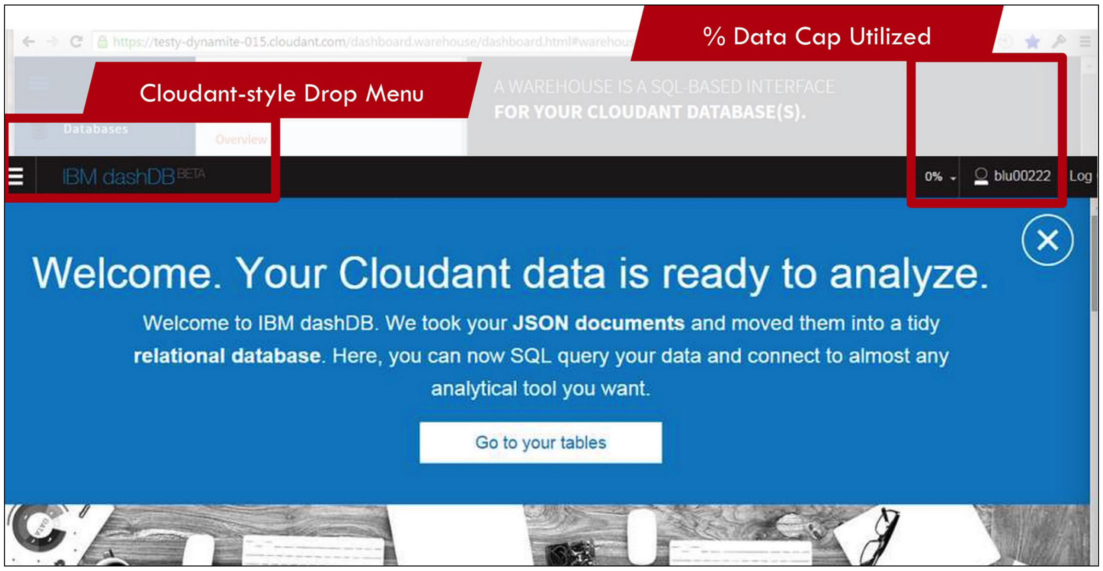

https://testy-dynamite-015.cloudant.com/dashboard.warehouse/dashboard.html#warehouse/overview
Databases
Replication
Warehouse
Active Tasks
Account
Support
Documentation
Cloudant
Logout testy-dynamite-015
New Warehouse
Overview
All Warehouses
A WAREHOUSE IS A SQL-BASED INTERFACE
FOR YOUR CLOUDANT DATABASE(S).
With a warehouse, you can:
See your JSON data in a relational database
Query your data with SQL
Build analytics from your data
When you create a warehouse, these things happen:

*Figure 2-10 Provisioning a dashDB analytics warehouse via the Cloudant Dashboard*

Specifying the database that will be used to populate dashDB and clicking Create Warehouse will initiate the process of data movement (from Cloudant NoSQL document data) into the dashDB data warehouse. Clicking the new “Warehouse” tab in Cloudant, you assign a name to your warehouse and add one or many Cloudant databases associated with your account (or public accounts) as a source as seen in Figure 2-10 above. The schema discovery process, via the IBM DataWorks data refinery services, takes over from here. This performs two things: First, it discovers the source database and JSON documents in order to infer a schema; second (and this is the more challenging and expensive component of the migration operations), is to move every document into the inferred schema. This schema must be created on the DB2 side of the equation, building tables and loading every document from Cloudant into BLU columnar representation.

One-time load (“initial load”) and continuous replication are the two models for ingesting data that DataWorks supports at the time of this writing. Since the general availability of dashDB, Cloudant now supports continuous synchronization between dashDB and Cloudant to ensure that changes in your Cloudant documents are reflected in dashDB’s tables, without needing to schedule a complete rescan of the database.

As dashDB matures, it is possible to interoperate this data with external services such as IBM Watson™ Analytics for further exploration and insight into your analytics warehouse and Systems of Engagement data. IBM dashDB can support multiple Cloudant databases as data sources, but only a single target warehouse can be instantiated to house this data. Furthermore, only 1 GB of decompressed data for the target data warehouse is supported, and no continuous ingestion is available for the Beta release of dashDB. See Figure 2-11 on page 28.



Welcome. Your Cloudant data is ready to analyze.

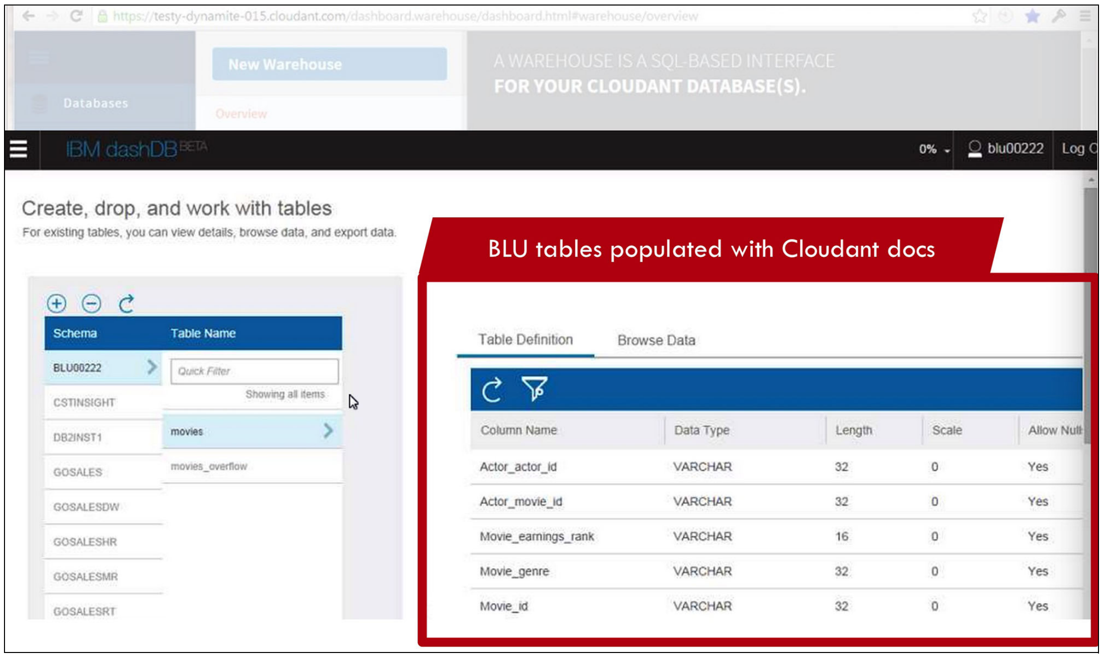
Welcome to IBM dashDB. We took your **JSON documents** and moved them into a tidy **relational database**. Here, you can now SQL query your data and connect to almost any analytical tool you want.

*Figure 2-11 IBM dashDB represents an exciting opportunity for Cloudant NoSQL customers to access the power and flexibility of a cloud-based analytics warehouse, built upon IBM BLU columnar performance and Netezza in-memory analytics*

As your documents are migrated from Cloudant to dashDB, you are given immediate feedback via the Cloudant Dashboard for monitoring the status of your data migration, such as progress towards finishing, the number of jobs running, and so on. IBM dashDB at this time supports 0.5 GB per hour throughput rates. The first documents to be migrated appear within the dashDB tables within 1-2 seconds of commencing the schema discovery job. All of this is carried out in the background using Cloudant RESTful API and JSON-encoded responses. Migration is performed using “batch” inserts, up to 10,000 lines at a time.

In Figure 2-11, IBM dashDB represents an exciting opportunity or Cloudant NoSQL customers to access the power and flexibility of a cloud-based analytics warehouse.

In Figure 2-12 on page 29, IBM dashDB migrates your NoSQL JSON documents from Cloudant into BLU columnar format automatically.


https://testy-dynamite-015.cloudant.com/dashboard.warehouse/dashboard.html#warehouse/overview

Databases
New Warehouse
Overview
A WAREHOUSE IS A SQL-BASED INTERFACE
FOR YOUR CLOUDANT DATABASE(S).
IBM dashDB BETA
0%
blu00222
Log O
BLU tables populated with Cloudant docs
Table Definition
Browse Data
Quick Filter

Create, drop, and work with tables

For existing tables, you can view details, browse data, and export data.

| Schema | Table Name |
|---|---|
| BLU00222 | |
| CSTINSIGHT | Showing all items |
| DB2INST1 | movies |
| GOSALES | movies_overflow |
| GOSALESDW | |
| GOSALESHR | |
| GOSALESMR | |
| GOSALESRT | |

| Column Name | Data Type | Length | Scale | Allow Nul |
|---|---|---|---|---|
| Actor_actor_id | VARCHAR | 32 | 0 | Yes |
| Actor_movie_id | VARCHAR | 32 | 0 | Yes |
| Movie_earnings_rank | VARCHAR | 16 | 0 | Yes |
| Movie_genre | VARCHAR | 32 | 0 | Yes |
| Movie_id | VARCHAR | 32 | 0 | Yes |

*Figure 2-12 IBM dashDB migrates your NoSQL JSON documents from Cloudant into BLU columnar format automatically, with performance demonstrated to be equally as efficient, and in some circumstances even faster, than the time needed to replicate a Cloudant database between nodes.*

Once schema discovery and data migration has been carried out, you have the option to visit the warehouse and navigate over to the dashDB console. It looks similar to the BLU Acceleration console on Bluemix, but has been modified to give a more Cloudant look and feel. There is a utilization meter at the top indicating how much storage you have consumed out of your 1 GB decompressed allocation, a representation of your Cloudant ID, and a drop-down navigational menu to the left.

At this stage, your BLU columnar tables have been populated with your Cloudant JSON document data as seen in Figure 2-12. This is supported by the schema discovery and data movement IBM WebSphere® application we just described, which discovers the JSON schema. Schema discovery runs as a synchronous process, and therefore it is necessary to wait for the complete schema to be available and computed. The schema discovery processes cannot continue until the schema has been determined and hardened.

Once dashDB has determined the schema, it maps the JSON schema and builds out an equivalent relational schema within DB2 BLU columnar representation. The rest of the migration job from this point onwards can be done asynchronously. Multiple users can be pushing data into the defined schema and mapping it into the relational table. IBM dashDB's performance is equivalent to the Cloudant-to-Cloudant database replication speed, in terms of how much time is required to complete the job. In some circumstances, dashDB's migration process is even able to exceed the replication performance of Cloudant. The upshot: users can push JSON documents into dashDB's data warehouse at a rate that exceeds Cloudant replication out-of-the-box.

At the time of this publication, IBM dashDB is available as a multi-tenant beta deployment from within IBM Cloudant and IBM Bluemix. Soon, and potentially by the time you are reading this, dashDB will expand to included dedicated, single tenant deployments via these platforms and others.

## 2.6.2 Data refinement services on-premises

The traditional strategy for data refinement and cleansing has been to take on-premises data through IBM InfoSphere® IIG (Information Integration & Governance) for refinement, and deliver that to on-premises targets or targets on the cloud for storage and for use with applications. Take a moment to consider Figure 2-13 for an illustration of how these components fit together.

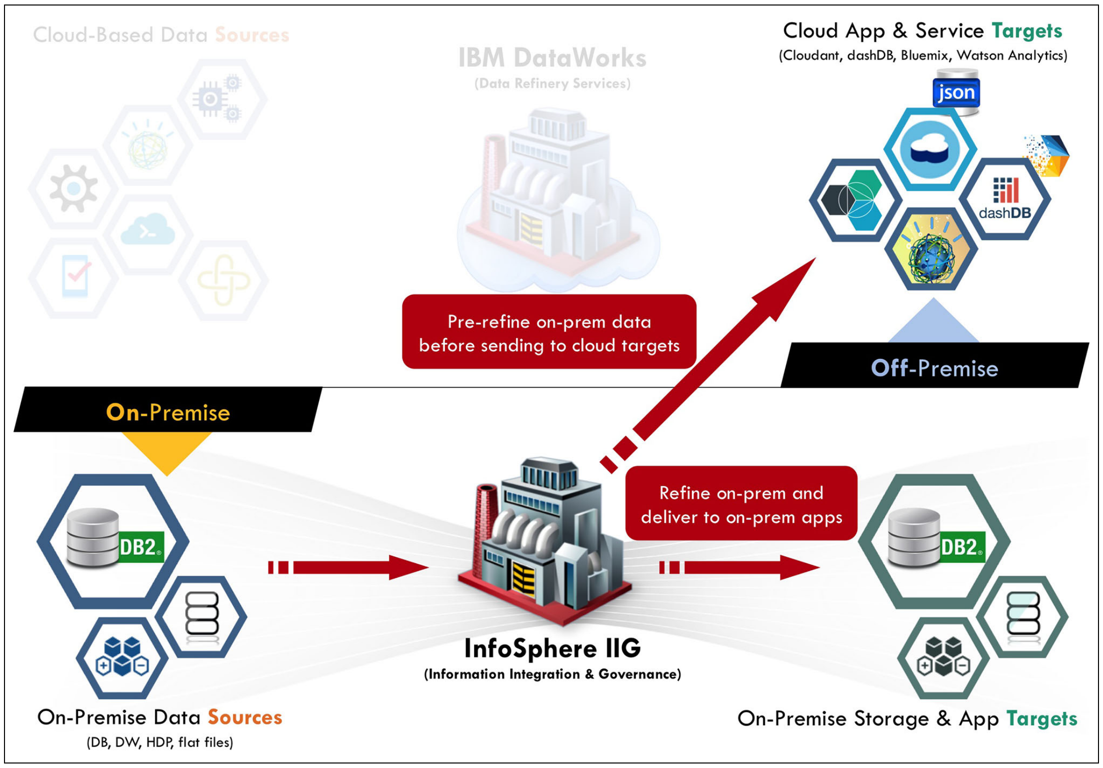

Cloud-Based Data Sources
IBM DataWorks
(Data Refinery Services)
Cloud App & Service Targets
(Cloudant, dashDB, Bluemix, Watson Analytics)
json
dashDB
Pre-refine on-prem data before sending to cloud targets
Off-Premise
On-Premise
DB2
Refine on-prem and deliver to on-prem apps
DB2
InfoSphere IIG
(Information Integration & Governance)
On-Premise Data Sources
(DB, DW, HDP, flat files)
On-Premise Storage & App Targets

*Figure 2-13 Data refinement services available on-premises*

## 2.6.3 Data refinement services on the cloud

The new landscape centers around IBM DataWorks v1.1, which independently refines on the cloud and simultaneously has access to on-premises data from the cloud via a secure gateway as shown in Figure 2-14 on page 31. The 2014 release of DataWorks is focused on-cloud to on-cloud interactions, or on-cloud to on-premises. In 2015, IBM will be able to pull from on-premises and bring that into on-cloud much more readily. In addition, publishing to on-premises (and not simply performing “read” operations) needs to consider the logic built around inserting data into on-premises relational stores. Solutions that commit data from the cloud back to relational on-premises systems need to respect and adhere to that data insertion logic.

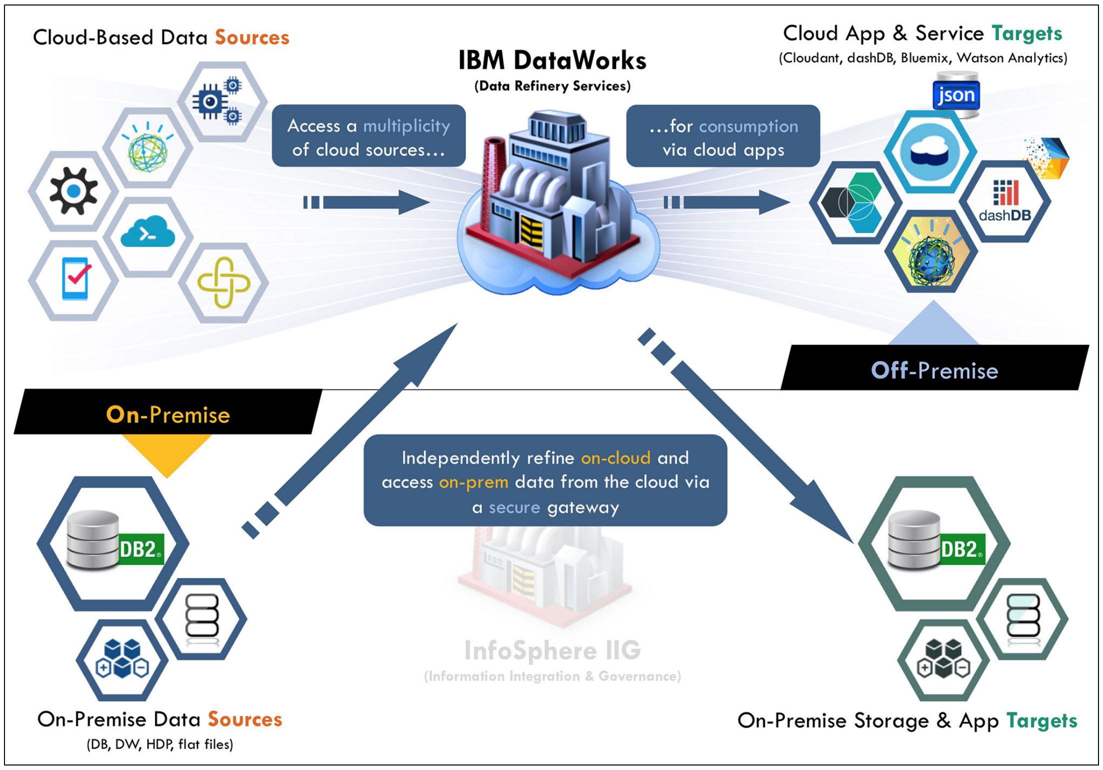

Cloud-Based Data Sources
IBM DataWorks
(Data Refinery Services)
Cloud App & Service Targets
(Cloudant, dashDB, Bluemix, Watson Analytics)
Access a multiplicity of cloud sources…
…for consumption via cloud apps
json
dashDB
Off-Premise
On-Premise
Independently refine on-cloud and access on-prem data from the cloud via a secure gateway
DB2
DB2
InfoSphere IIG
(Information Integration & Governance)
On-Premise Data Sources
(DB, DW, HDP, flat files)
On-Premise Storage & App Targets

*Figure 2-14 Data refinement services available on the cloud*

Going forward into 2015, DataWorks (on-cloud) will have a hybrid solution with InfoSphere IIG (on-premises) to give access from IIG to DataWorks for common governance, common lineage of the data, and so on. We believe that this is the vision of data refinement services between on-premises and on cloud.

## 2.6.4 Hybrid clouds: Data refinement across on/off–premises

When we open the discussion to the future of relational and cloud-centric databases, we are really talking about the evolution of "hybrid databases": multiple databases used together inside of an organization's ecosystem to solve complex data management use cases. These types of challenges could not be tackled before with yesterday's on-premises only infrastructure. The goal is to offer a comprehensive set of services for bringing together the transactional, ACID-compliant data stores of traditional DB2 with the operational data store capabilities of Cloudant. Of course, many of our customers have invested in on-premises infrastructure (like DB2) as well, so this solution needs to bridge both the on-premises and off-premises (on-the-cloud) worlds.

Figure 2-15 on page 32 shows that hybrid clouds require data refinement across both on-premises and off-premises infrastructure.

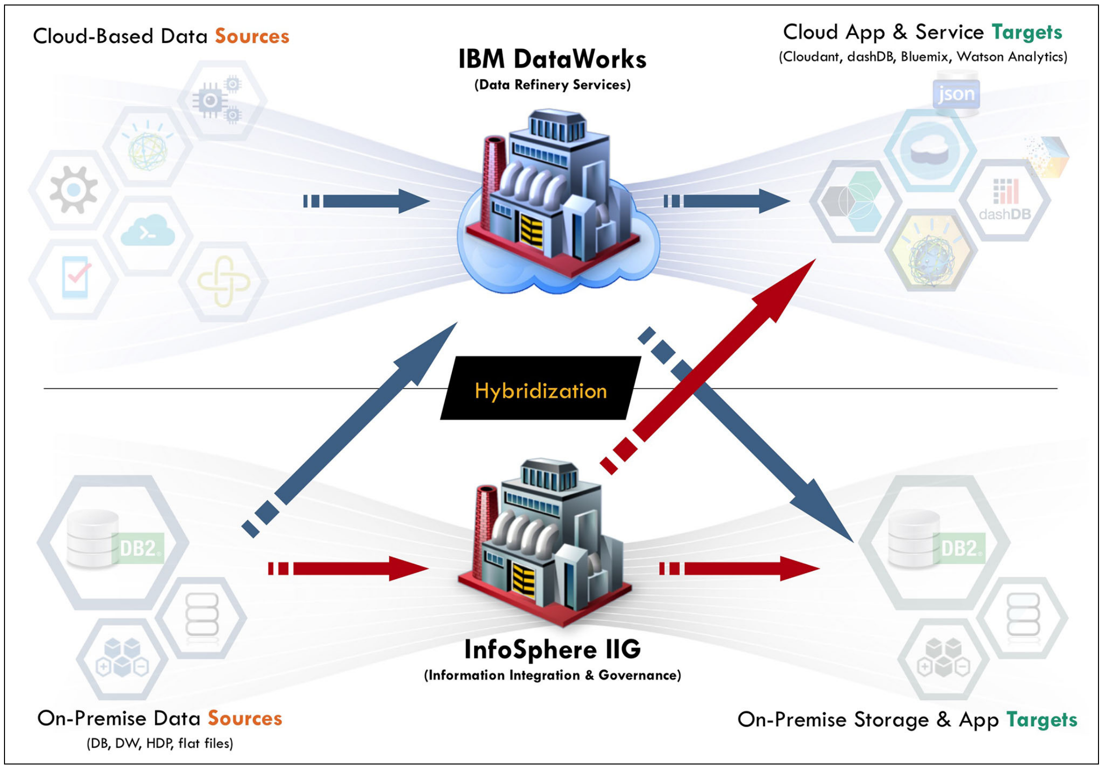

Cloud-Based Data Sources
IBM DataWorks
(Data Refinery Services)
Cloud App & Service Targets
(Cloudant, dashDB, Bluemix, Watson Analytics)
json
dashDB
Hybridization
DB2
DB2
InfoSphere IIG
(Information Integration & Governance)
On-Premise Data Sources
(DB, DW, HDP, flat files)
On-Premise Storage & App Targets

*Figure 2-15 Hybrid clouds require data refinement across both on-premises and off-premises infrastructure*

The shape of the data, knowing where it is appropriate to store data in tabular rows and columns, or persisting it in a flexible JSON document schema, is a key consideration. However, even more important are the considerations that you make about the type of workloads you need to perform on that data. Using a JSON store for high performance transactional workloads, for example, does not necessarily play to the strengths and advantages of using JSON documents. The key is storing data, according to type and how you want to use that data in the appropriate silo, and then providing services to interoperate between those different stores as needed.

IBM's enterprise-ready Big Data and Analytics portfolio enables customers to address the full spectrum of challenges across areas of mobile, social, big data, and the cloud. Cloudant extends these capabilities by providing another leading solution to an already market-leading portfolio.

3

# Introduction to JSON and fundamentals of document data stores

Using JavaScript Object Notation (JSON) as a means of transferring data between source and target systems is simple and easy given the nature of JSON itself. In this chapter, we explore the structure of JSON, and how easy it is to create, update, and delete JSON data within IBM Cloudant.

## 3.1 What is a JSON?

JSON stands for JavaScript Object Notation, and has become a standard method of transferring data between systems. It is a text-based open standard designed for human-readable data interchange. It is derived from the JavaScript scripting language for representing simple data structures and associative arrays, called objects.

JSON example:

```
{
     "firstName": "John",
     "lastName" : "Smith",
     "age"      : 25,
     "address"  :
     {
         "streetAddress": "21 2nd Street",
         "city"         : "New York",
         "state"        : "NY",
         "postalCode"   : "10021"
     },
     "phoneNumber":
     [
         {
           "type"  : "home",
           "number": "212 555-1234"
         },
         {
           "type"  : "fax",
           "number": "646 555-4567"
         }
     ]
 }
```

### History of JSON

Long ago (in the Internet age) a man named Douglas Crawford wanted an easier way of transferring information between browsers and web servers. He wanted to do it in a way that did not require the use of frameworks provided by browser plug-ins like Flash or Java Applets.

Douglas found that while he kept the http port open between the web server and browser, he could transfer JavaScript data between the source and target repeatedly. While this ability had been around for a while, no one had named this feature of data interchange, so he called it *JavaScript Object Notation*, and created a website in 2001 called JSON.org to create the specification.

### JSON types

Number is a signed decimal number that can use exponential notation E notation using the double-precision floating-point format.

#### *String type*

The String type is simply a set of zero or more characters delimited by double quotation marks. The String syntax also supports the backslash escaping syntax for special characters.

Examples:

```
{ "firstName": "John"}
```

Assigns the value of “John” to the string variable name firstName.

```
{ “doubleQuote” : ”\””}
```

Assigns the single double quote character to the string name doubleQuote)

### Boolean type

The Boolean type assigns a true or false value to a key name. Such as:

```
{ "isResident": true}
```

Assigns the variable name isResident the value true.

### Array type

The Array type is an ordered list of any type, and can have zero or greater members in the array. The Array type is bracketed by the Square bracket **]**.

Example:

```
{
"tableware" : [
    {"itemName":"cups", "itemCount":12 },
    {"itemName":"saucers", " itemCount ":11 },
    {"itemName":"spoons", " itemCount ":8 }
]
}
```

In the preceding example, the object “tableware” is an array containing three objects. Each object is a record of an item with an associated number of that item.

### Null Type

The Null type is the universal indicator of an empty value.

Example:

```
{ “initialString”: null )
```

Clears the variable “initialSting” and places nothing (an empty value) within it.

### The Object type

The object type is what we later define as a JSON Document, but for now we define it as any unordered set of key value pairs. Objects are delimited with curly brackets `http://en.wikipedia.org/wiki/Bracket#Braces` and use commas to separate each pair, while within each pair the colon “:” character separates the key or name from its value. All keys must be strings and should be distinct from each other within that object.

This object set can consist of numbers, strings, arrays, and even have these objects nested within each other. However, for readability, we typically keep the object relatively flat.

Example:

```
{
  "firstName": "Chuck",
  "lastName": "Taylor",
  "isResident": true,
  "age": 45,
  "address": {
    "streetAddress": "2 Boylsten Street",
    "city": "Boston",
    "state": "MA",
    "postalCode": "01902"
  },
  "phoneNumbers": [
    {
      "type": "home",
      "number": "617 555-1234"
    },
    {
      "type": "office",
      "number": "781 555-4567"
    }
  ],
  "children": [],
  "spouse": null
}
```

JSON generally ignores any whitespace around or between syntactic elements (values and punctuation, but not within a string value). However, JSON only recognizes four specific whitespace characters: the space, horizontal tab, line feed, and carriage return. JSON does not provide or allow any sort of comment syntax.

## 3.2 Document Data Stores

Document Data Stores are computer programs that store data in a semi-structured form, specifically the JSON structure type. The Document Data Store is one type of NoSQL database system that had become very popular as an easy to use and develop type of database system as opposed to the SQL-based relational database systems such as IBM DB2.

While Document Data Stores dominate the NoSQL market with over 50%, the other types of NoSQL Data Stores include:

- Column family Stores
- Key-value Stores
- Graph Type Stores

Document Data Stores hold values in a document in a semi-structured relationship. However, there are several markup languages to define these structures. These include:

- XML
- YAML
- JSON
- BSON

JSON Document databases are popular because the data is stored in a natural, readable format. Unlike other NoSQL database formats, fields typically require operators to define the relationship between various fields within a NoSQL database. However, with the JSON Document Data Store, each document entry is a key value set of readable natural language concepts.

## 3.3 Basics of create, read, update, and delete operations

CRUD is an acronym for:

- Create
- Read
- Update
- Delete

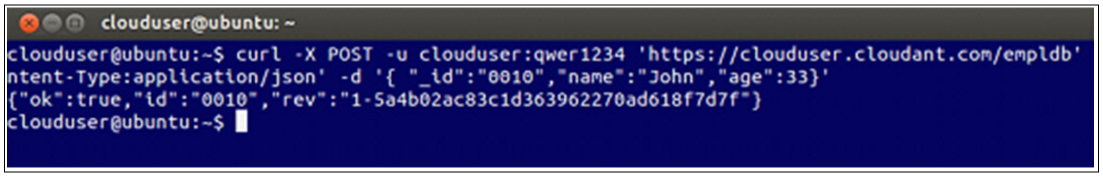
Using curl and a Cloudant database, we can start with trying out these basic CRUD operations.

### Creating new document

Use the following command to create a document:

```
curl -X POST -u <username>:<password> 'https://<username>.cloudant.com/empldb' -H
'Content- Type:application/json' -d '{ "_id":"0010","name":"John","age":33}'
```

Creating a document is shown in Figure 3-1.



```
clouduser@ubuntu:~$ curl -X POST -u clouduser:qwer1234 'https://clouduser.cloudant.com/empldb'
ntent-Type:application/json' -d '{ "_id":"0010","name":"John","age":33}'
{"ok":true,"id":"0010","rev":"1-5a4b02ac83c1d363962270ad618f7d7f"}
clouduser@ubuntu:~$
```

*Figure 3-1 Creating a document*

| | |
|---|---|
| ***curl*** | Is command called "Command-Line URL". It is a command-line tool for getting or sending files using URL syntax. |
| ***-X POST*** | Specifies that a custom request is being sent, and the default should not be used. Typically, the default is a GET operation, but in this case we are performing a POST. The POST operation is telling the server to initiate an action. In this case it is creating a new document with the additional JSON data. |
| ***-u*** | Is the username and password combination. |
| | `'https://<username>.cloudant.com/empldb'` is the specified location of where to send this request. |
| ***-H*** | This specifies an extra header to identify the request. In this case we are specifying the header as type: `'Content- Type:application/json'` |
| ***-d*** | This specifies the data to be sent for the request. In this case we are sending a complete JSON document enclosed in single quotation marks: `d '{ "_id":"0010","name":"John","age":33}'` |

The Service returns a JSON document back to indicate whether our create request worked or not. In this case we received:

{“ok”:true, “id”:”0010”,”rev”:”1-5a4b02ac83c1d363962270ad618f7d7f”}

Where:

| | |
|---|---|
| ***“ok”:true*** | Indicates a successful POST operation. |

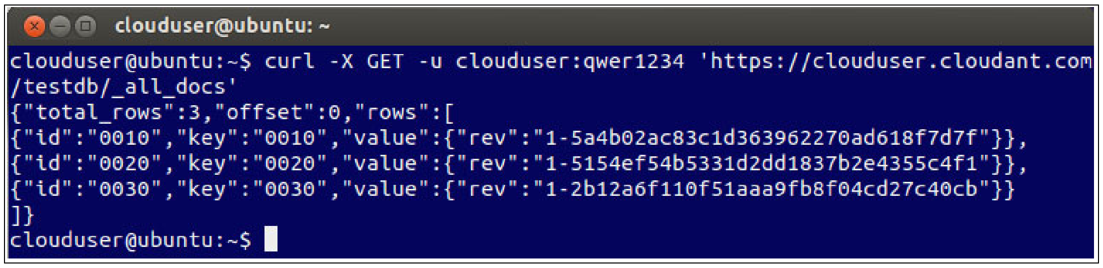
| | |
|---|---|
| ***“id”:”0010”*** | Indicates the document ID assigned to the newly created document. |
| ***”rev”:”1-5a4b02ac83c1d363962270ad618f7d7f”*** | Returns the unique value of the revision number assigned to your creation request. Any further updates to your created document will require this revision number and a new revision number will be assigned. |

## Read documents

Use the following command to read documents:

*curl -X GET -u <username>:<password> ‘https://<username>.cloudant.com/testdb/_all_docs'*

Reading a document is shown in Figure 3-2.



```
clouduser@ubuntu:~$ curl -X GET -u clouduser:qwer1234 'https://clouduser.cloudant.com
/testdb/_all_docs'
{"total_rows":3,"offset":0,"rows":[
{"id":"0010","key":"0010","value":{"rev":"1-5a4b02ac83c1d363962270ad618f7d7f"}},
{"id":"0020","key":"0020","value":{"rev":"1-5154ef54b5331d2dd1837b2e4355c4f1"}},
{"id":"0030","key":"0030","value":{"rev":"1-2b12a6f110f51aaa9fb8f04cd27c40cb"}}
]}
clouduser@ubuntu:~$
```

*Figure 3-2 Reading a document*

Here we are reading or getting a collection of documents from the document store.

***curl –X GET*** means that we are executing a **GET** command.

-u specifies the username and password.

‘`https://<username>.cloudant.com/testdb/_all_docs`' specifies the location of the GET request. As you can see, we added _all_docs to the end of the location to indicate that we are using the function “all_docs” to read the current document entries.

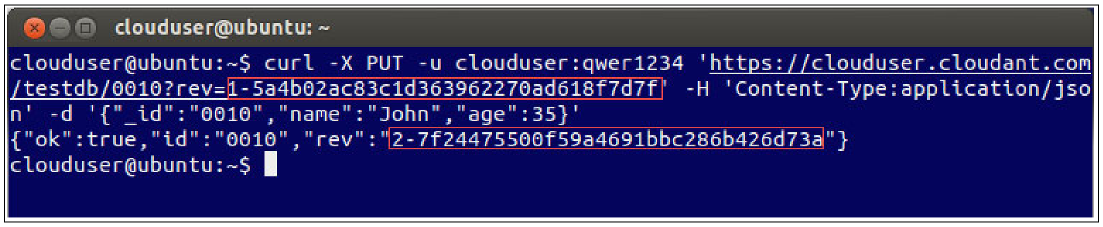
The return values indicate the number of rows, and the actual JSON documents themselves.

## Update documents

To update a document in the Cloudant database, create another document with the same *_id* and *_rev*. Note that we are using PUT (instead of POST). To update the age for *_id* “0010”, run the following command:

```
curl -X PUT -u <username>:<password>
'https://<username>.cloudant.com/testdb/0010?rev=1
-5a4b02ac83c1d363962270ad618f7d7f' -H 'Content-Type:application/json' -d
'{"_id":"0010","name":"John","age":35}'
```

Updating a document is shown in Figure 3-3 on page 39.



```
cloudduser@ubuntu: ~
clouduser@ubuntu:~$ curl -X PUT -u clouduser:qwer1234 'https://clouduser.cloudant.com
/testdb/0010?rev=1-5a4b02ac83c1d363962270ad618f7d7f' -H 'Content-Type:application/jso
n' -d '{"_id":"0010","name":"John","age":35}'
{"ok":true,"id":"0010","rev":"2-7f24475500f59a4691bbc286b426d73a"}
clouduser@ubuntu:~$ 
```

Figure 3-3   Updating a document

This is similar to the *create* that we did earlier, however there are a few key differences described here.

***curl –X PUT*** we are specifying a **PUT** command instead of a **POST** command. Post is used to submit data to a request. PUT is used to forcefully place data in a specific location. In this case, we are specifying that location by including the document ID and the revision number on the location request:

```
'https://<username>.cloudant.com/testdb/0010?rev=1-5a4b02ac83c1d363962270ad618f7d7
f'
```

We receive the return code indicating that it was “ok” and we also receive the new revision number associated with that document ID. Now any further updates to that row must include the new system generated revision ID.

## Delete Operation

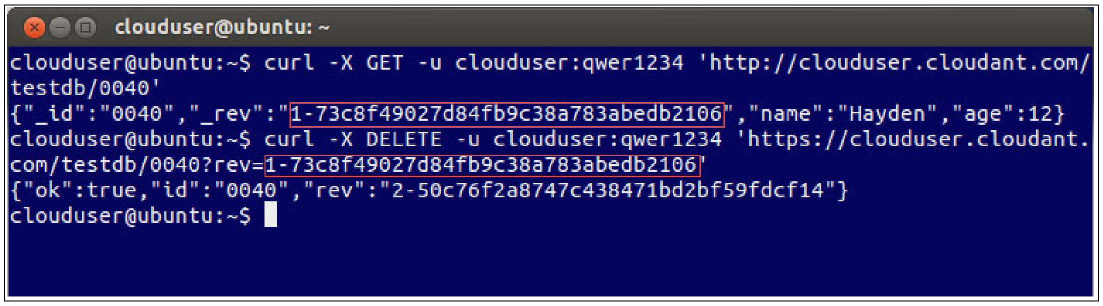
To delete a document, for example, with ID “0040”, we must first retrieve the *_rev* number for the document ID we want to delete:

```
curl –X GET -u <username>:<password> 'https://<username>.cloudant.com/testdb/0040'
```

Then, use the returned *_rev* to complete the DELETE request:

```
curl -X DELETE -u <username>:<password>
'https://<username>.cloudant.com/testdb/0040?rev=2-f50c76f2a8747c438471bd2bf59fdcf
14'
```

The delete operation is shown in Figure 3-4.



```
clouduser@ubuntu: ~
clouduser@ubuntu:~$ curl -X GET -u clouduser:qwer1234 'http://clouduser.cloudant.com/
testdb/0040'
{"_id":"0040","_rev":"1-73c8f49027d84fb9c38a783abedb2106","name":"Hayden","age":12}
clouduser@ubuntu:~$ curl -X DELETE -u clouduser:qwer1234 'https://clouduser.cloudant.
com/testdb/0040?rev=1-73c8f49027d84fb9c38a783abedb2106'
{"ok":true,"id":"0040","rev":"2-50c76f2a8747c438471bd2bf59fdcf14"}
clouduser@ubuntu:~$ 
```

Figure 3-4   Deleting a document

We are returned with an ok, and a revision number. The revision number is used only in the case that a conflict needs to be resolved in the future. Otherwise, the revision number is associated with no data, and the row is now effectively deleted.

4

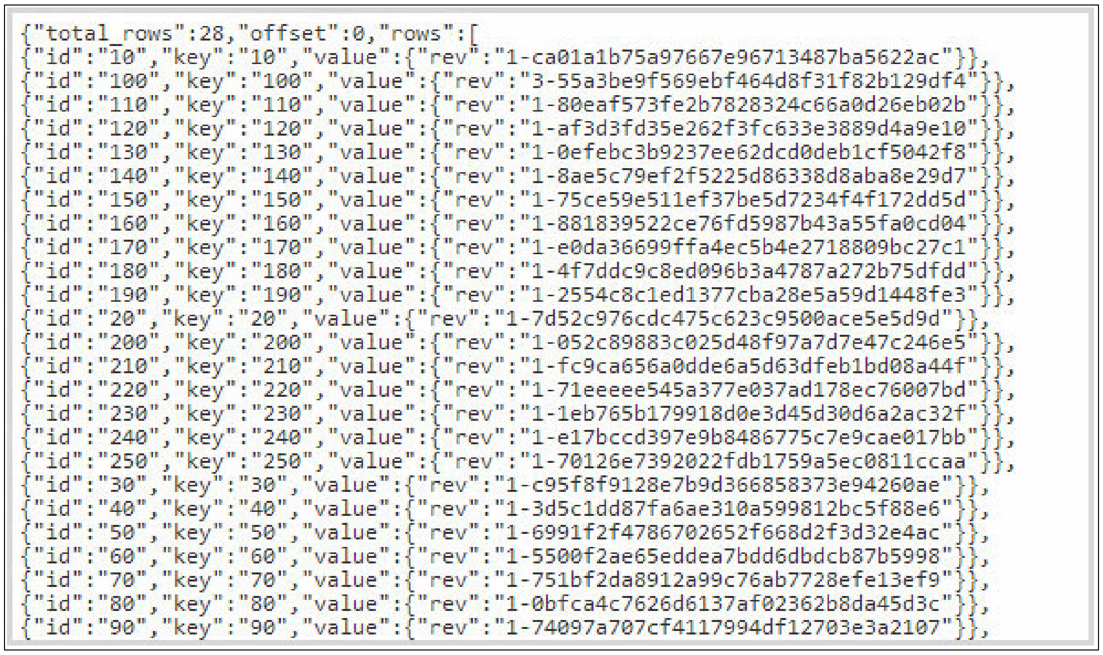
# Primary index

The fastest and simplest way to query an IBM Cloudant database is using the primary index. Each document has an *_id* field that serves as the primary key; it can be retrieved using the *../db/_all_docs* endpoint. Invoking this endpoint will return a key/value pair where key is *_id* and the value is the *_rev* field. Note that the *_id* field of every document in a database must be unique (no two documents can share the same *_id*). The *_rev* value is unique for every document, and is produced by applying a hash function to a combination of the documents contents and the path to where that document resides in the database. The *_all_docs* endpoint reports on the total number of documents belonging to a database and any offset used to query the index. Results will be sorted by the keys.

The following command that uses Representational State Transfer (REST) application programming interfaces (APIs) will return all rows for ***staff*** database:

```
GET https://redbookid.cloudant.com/staff/_all_docs
```

Figure 4-1 on page 42 shows the result of this command:



```
{"total_rows":28,"offset":0,"rows":[
{"id":"10","key":"10","value":{"rev":"1-ca01a1b75a97667e96713487ba5622ac"}},
{"id":"100","key":"100","value":{"rev":"3-55a3be9f569ebf464d8f31f82b129df4"}},
{"id":"110","key":"110","value":{"rev":"1-80eaf573fe2b7828324c66a0d26eb02b"}},
{"id":"120","key":"120","value":{"rev":"1-af3d3fd35e262f3fc633e3889d4a9e10"}},
{"id":"130","key":"130","value":{"rev":"1-0efebc3b9237ee62dcd0deb1cf5042f8"}},
{"id":"140","key":"140","value":{"rev":"1-8ae5c79ef2f5225d86338d8aba8e29d7"}},
{"id":"150","key":"150","value":{"rev":"1-75ce59e511ef37be5d7234f4f172dd5d"}},
{"id":"160","key":"160","value":{"rev":"1-881839522ce76fd5987b43a55fa0cd04"}},
{"id":"170","key":"170","value":{"rev":"1-e0da36699ffa4ec5b4e2718809bc27c1"}},
{"id":"180","key":"180","value":{"rev":"1-4f7ddc9c8ed096b3a4787a272b75dfdd"}},
{"id":"190","key":"190","value":{"rev":"1-2554c8c1ed1377cba28e5a59d1448fe3"}},
{"id":"20","key":"20","value":{"rev":"1-7d52c976cdc475c623c9500ace5e5d9d"}},
{"id":"200","key":"200","value":{"rev":"1-052c89883c025d48f97a7d7e47c246e5"}},
{"id":"210","key":"210","value":{"rev":"1-fc9ca656a0dde6a5d63dfeb1bd08a44f"}},
{"id":"220","key":"220","value":{"rev":"1-71eeeee545a377e037ad178ec76007bd"}},
{"id":"230","key":"230","value":{"rev":"1-1eb765b179918d0e3d45d30d6a2ac32f"}},
{"id":"240","key":"240","value":{"rev":"1-e17bccd397e9b8486775c7e9cae017bb"}},
{"id":"250","key":"250","value":{"rev":"1-70126e7392022fdb1759a5ec0811ccaa"}},
{"id":"30","key":"30","value":{"rev":"1-c95f8f9128e7b9d366858373e94260ae"}},
{"id":"40","key":"40","value":{"rev":"1-3d5c1dd87fa6ae310a599812bc5f88e6"}},
{"id":"50","key":"50","value":{"rev":"1-6991f2f4786702652f668d2f3d32e4ac"}},
{"id":"60","key":"60","value":{"rev":"1-5500f2ae65eddea7bdd6dbdcb87b5998"}},
{"id":"70","key":"70","value":{"rev":"1-751bf2da8912a99c76ab7728efe13ef9"}},
{"id":"80","key":"80","value":{"rev":"1-0bfca4c7626d6137af02362b8da45d3c"}},
{"id":"90","key":"90","value":{"rev":"1-74097a707cf4117994df12703e3a2107"}},
```

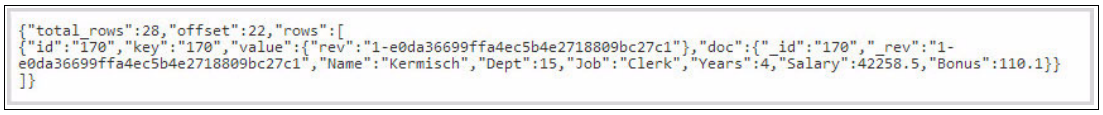
*Figure 4-1 The result of a call made to the all_docs endpoint*

Now, let us assume that we want to get data for the staff member with ID 170 and we want to return the full document.

The following command will do just that:

```
GET https://redbookid.cloudant.com/staff/_all_docs?key="170"&include_docs=true
```

Figure 4-2 shows the result from this command.

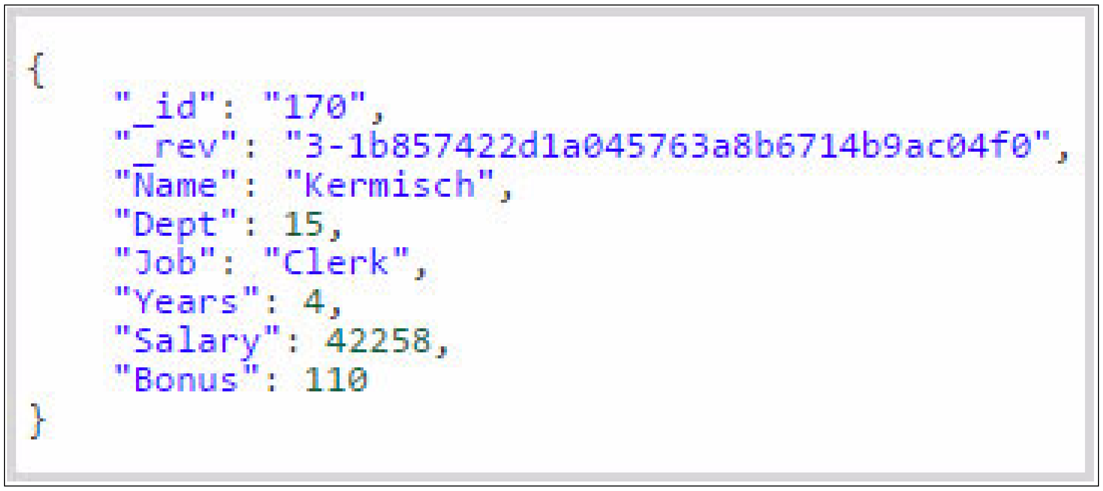


```
{"total_rows":28,"offset":22,"rows":[
{"id":"170","key":"170","value":{"rev":"1-e0da36699ffa4ec5b4e2718809bc27c1"},"doc":{"_id":"170","_rev":"1-
e0da36699ffa4ec5b4e2718809bc27c1","Name":"Kermisch","Dept":15,"Job":"Clerk","Years":4,"Salary":42258.5,"Bonus":110.1}}
]}
```

*Figure 4-2 Retrieving document using _all_docs endpoint based on the primary key with option include_docs=true*

Alternatively, we can retrieve the same document using the unique *_id* as an argument supplied to the URL. The full document will be returned:

```
GET https://redbookid.cloudant.com/staff/170
```



```
{
    "_id": "170",
    "_rev": "3-1b857422d1a045763a8b6714b9ac04f0",
    "Name": "Kermisch",
    "Dept": 15,
    "Job": "Clerk",
    "Years": 4,
    "Salary": 42258,
    "Bonus": 110
}
```

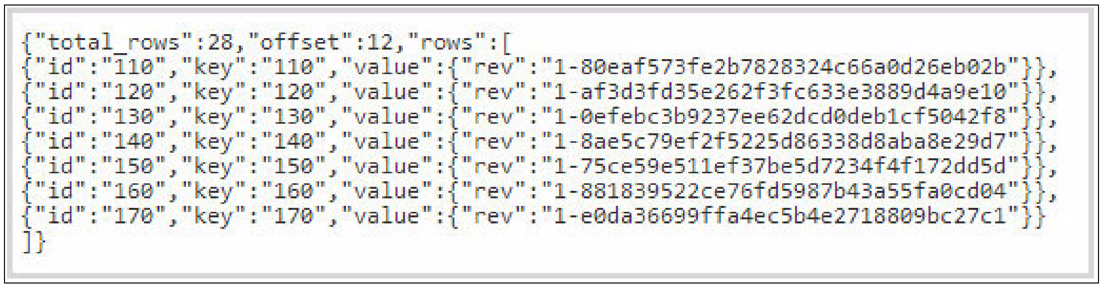
*Figure 4-3 Retrieving a document based on the primary key value*

Note that we do not have to put quotes around _id in this case.

We can find all documents where the primary key ranges between two values using **startkey** and **endkey**.

The following command will return documents with IDs greater than 110, up to and including 170 as seen in Figure 4-4.

```
GET https://redbookid.cloudant.com/staff/_all_docs?startkey="110"&endkey="170"
```



```
{"total_rows":28,"offset":12,"rows":[
{"id":"110","key":"110","value":{"rev":"1-80eaf573fe2b7828324c66a0d26eb02b"}},
{"id":"120","key":"120","value":{"rev":"1-af3d3fd35e262f3fc633e3889d4a9e10"}},
{"id":"130","key":"130","value":{"rev":"1-0efebc3b9237ee62dcd0deb1cf5042f8"}},
{"id":"140","key":"140","value":{"rev":"1-8ae5c79ef2f5225d86338d8aba8e29d7"}},
{"id":"150","key":"150","value":{"rev":"1-75ce59e511ef37be5d7234f4f172dd5d"}},
{"id":"160","key":"160","value":{"rev":"1-881839522ce76fd5987b43a55fa0cd04"}},
{"id":"170","key":"170","value":{"rev":"1-e0da36699ffa4ec5b4e2718809bc27c1"}}
]}
```

*Figure 4-4 Invoking the all_docs endpoint with parameters specifying a minimum start key value and maximum end key value*

The following table (Table 4-1) provides a list of the optional parameters that can be supplied to call to the *_all_docs* endpoint.

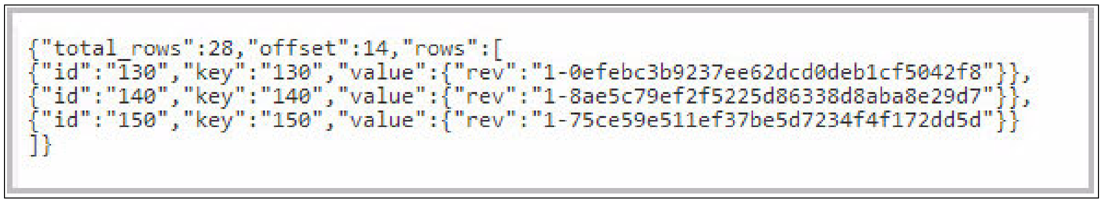
*Table 4-1 A list of all option for _all_docs*

| **key** | **Retrieve document with specific _id** |
|---|---|
| startkey, endkey | Retrieve range of documents based on the primary key |
| include_docs=(true I false) | Include the document body into results, default is “false” |
| descending=(true I false) | Sort in descending order, default is “false” |
| limit=N | Limit result to N documents |
| skip=N | Skip over the first N documents |

For example, we can narrow the results of the previous query using skip and limit options, such as with the following:

```
GET
https://redbookid.cloudant.com/staff/_all_docs?startkey="110"&endkey="170"&skip=2&
limit=3
```

As you can see in Figure 4-5, the resulting document contains only three results (as specified by the argument *limit=3*). Furthermore, the first two results that were returned by the command issued in Figure 4-3 on page 43 are missing. This is because of the parameter skip=2, which instructs Cloudant to not return the first two results that would otherwise be selected by such a query made with *startkey=110* and *enkey=170.*



```
{"total_rows":28,"offset":14,"rows":[
{"id":"130","key":"130","value":{"rev":"1-0efebc3b9237ee62dcd0deb1cf5042f8"}},
{"id":"140","key":"140","value":{"rev":"1-8ae5c79ef2f5225d86338d8aba8e29d7"}},
{"id":"150","key":"150","value":{"rev":"1-75ce59e511ef37be5d7234f4f172dd5d"}}
]}
```

*Figure 4-5 Performing the same query as before, but with additional limitations on for rejecting the first two values returned, and limiting the number of results to three*

5

# Building and querying the secondary index

One of the ways you can do reporting and querying in IBM Cloudant is by using “Views.” Views are defined via MapReduce functions written in Java Script and are a very powerful way to index and aggregate data. Views are also often referred as *secondary indexes*. In this chapter, we discuss them in depth.

## 5.1 Secondary indexes or MapReduce

The benefits of MapReduce include the ability to process large data sets in parallel fashion. Views are updated incrementally, as JavaScript Object Notation (JSON) docs are created, updated, or deleted in Cloudant. Any views that reference them are also updated instantly and without requiring a complete regeneration of the view.

By design, MapReduce operations consist of two parts: a *map* phase that reads each document and emits one or more key-value pairs for each document, and a *reduce* phase that processes the output of the map operation. By being one of the forms of aggregation, MapReduce can specify a query condition to select the input documents as well as sort and limit the results.

Figure 5-1 demonstrates the concept of MapReduce strategy.

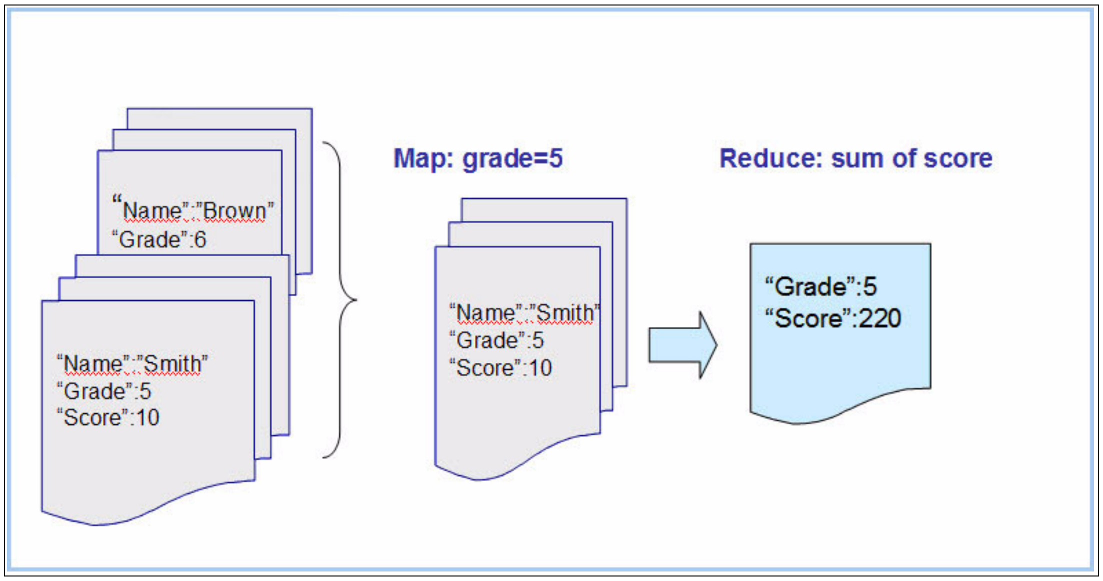


*Figure 5-1 Simplistic view of how MapReduce works. Map all documents with Grade 5, and reduce them with the "Summary" function*

Both map and reduce could be written as a JavaScript function. Cloudant provides several built-in reduce functions and because they are written in Cloudant native Erlang rather than JavaScript, they run much faster than custom functions. So whenever it is possible, do not write custom JavaScript reduce functions, use built-in ones.

### 5.1.1 Defining secondary index or map function

Secondary indexes, or views, are defined in a map function, which pulls out data from your documents and with an optional reduce function aggregates the data emitted by the map.

These functions are written in JavaScript and held in "design documents". These are special documents that the database knows contain the reduce functions and other indexes. You can think of them as documents that define secondary indexes.

You can define secondary indexes using the Cloudant Dashboard. In the database view, select "**+**" next to "All Design Docs" line and select "**New View**" from the drop-down menu as shown in Figure 5-2 on page 47.

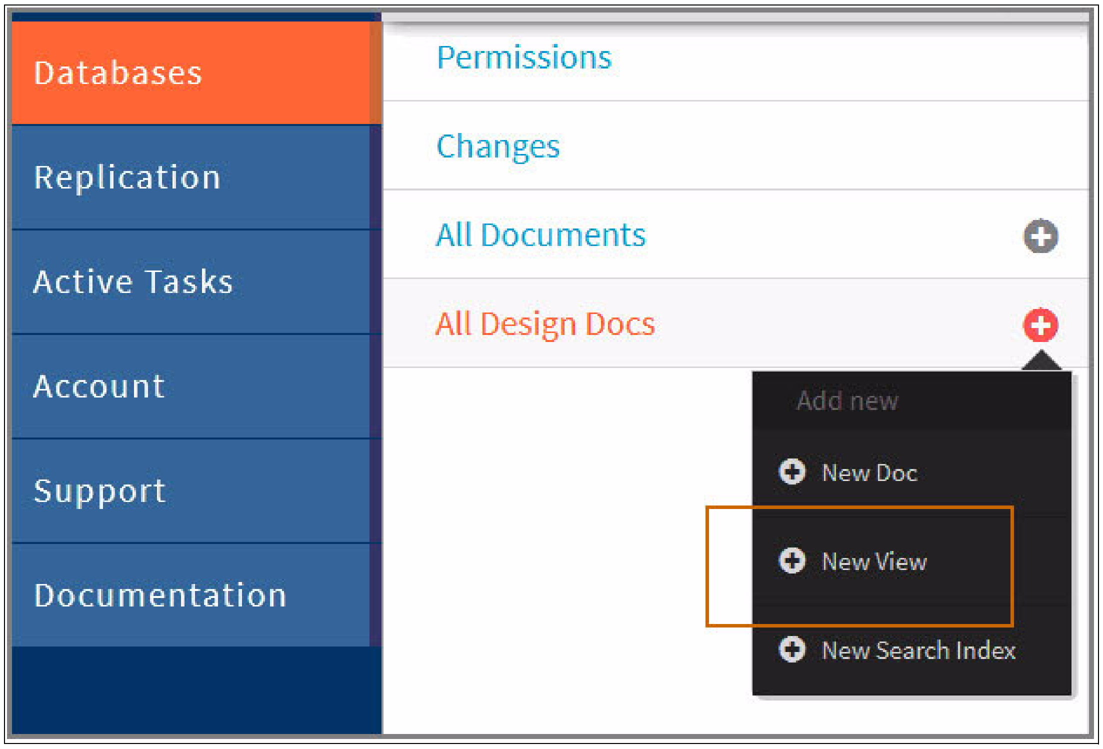


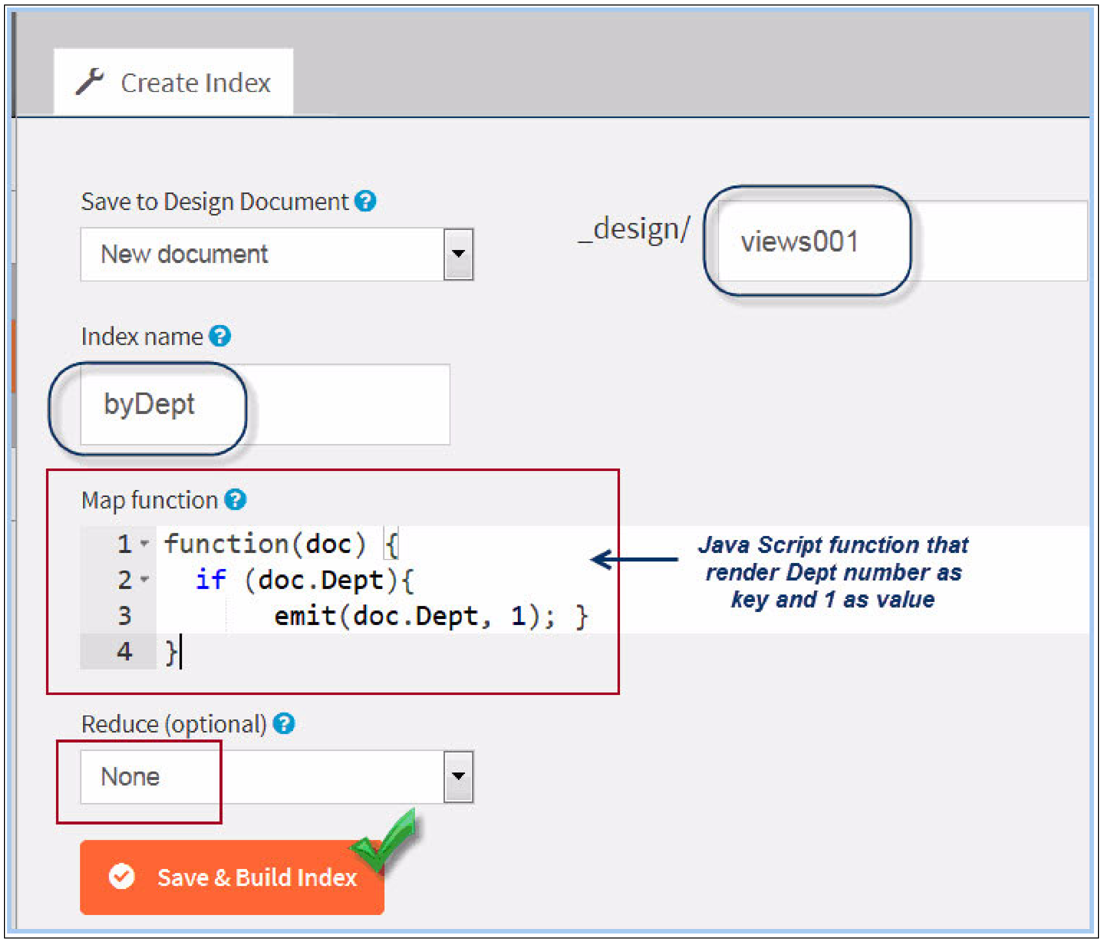
*Figure 5-2 Using Database View of Cloudant Dashboard to create a new secondary index*

As a result, you see the “Create Index” editor window, which is where you can specify the name of the design document, the name of index, and the Java Script function for both map and reduce. The reduce function is optional and we omit this for now.

Let us create an index that renders our staff department number as key with 1 as the value. The JavaScript map function will look like:

```
function(doc) {
if (doc.Dept){
emit(doc.Dept, 1); }
}
```

Specify “views001” for the design document name and “byDept” for the index name as shown in Figure 5-3 on page 48. Edit the default map function to emit department number. Then, press **Save & Build Index** to build this index on the staff database.



```
function(doc) {
  if (doc.Dept){
      emit(doc.Dept, 1); }
}
```

*Figure 5-3 Building secondary index "byDept" in design doc "views001"*

As a result, for each document we have a key-value pair that includes Dept number as key and 1 as value. Note that each result includes the original document _id:

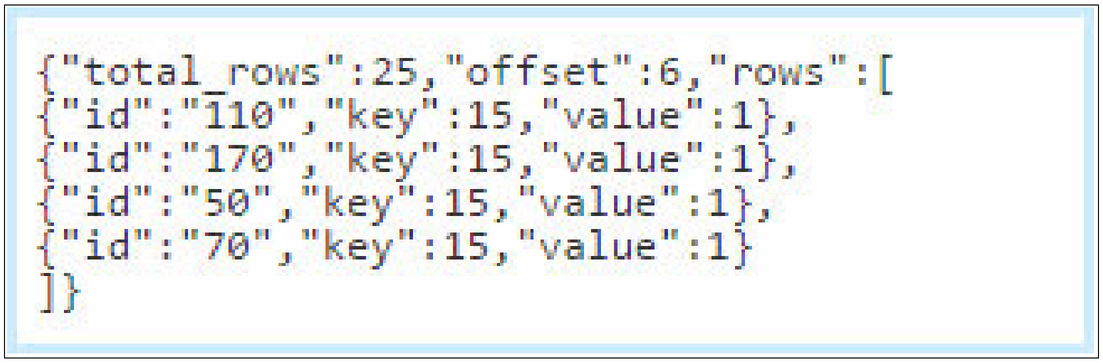
```
{
  "id": "160",
  "key": 10,
  "value": 1
}
```

## 5.1.2 Querying the database using secondary index

Now, search the database using the byDept index to find all documents where Dept number equals 15. To do that, send the following request:

```
GET https://redbookid.cloudant.com/staff/_design/views001/_view/byDept?key=15
```

Figure 5-4 on page 49 shows the response from the database server.



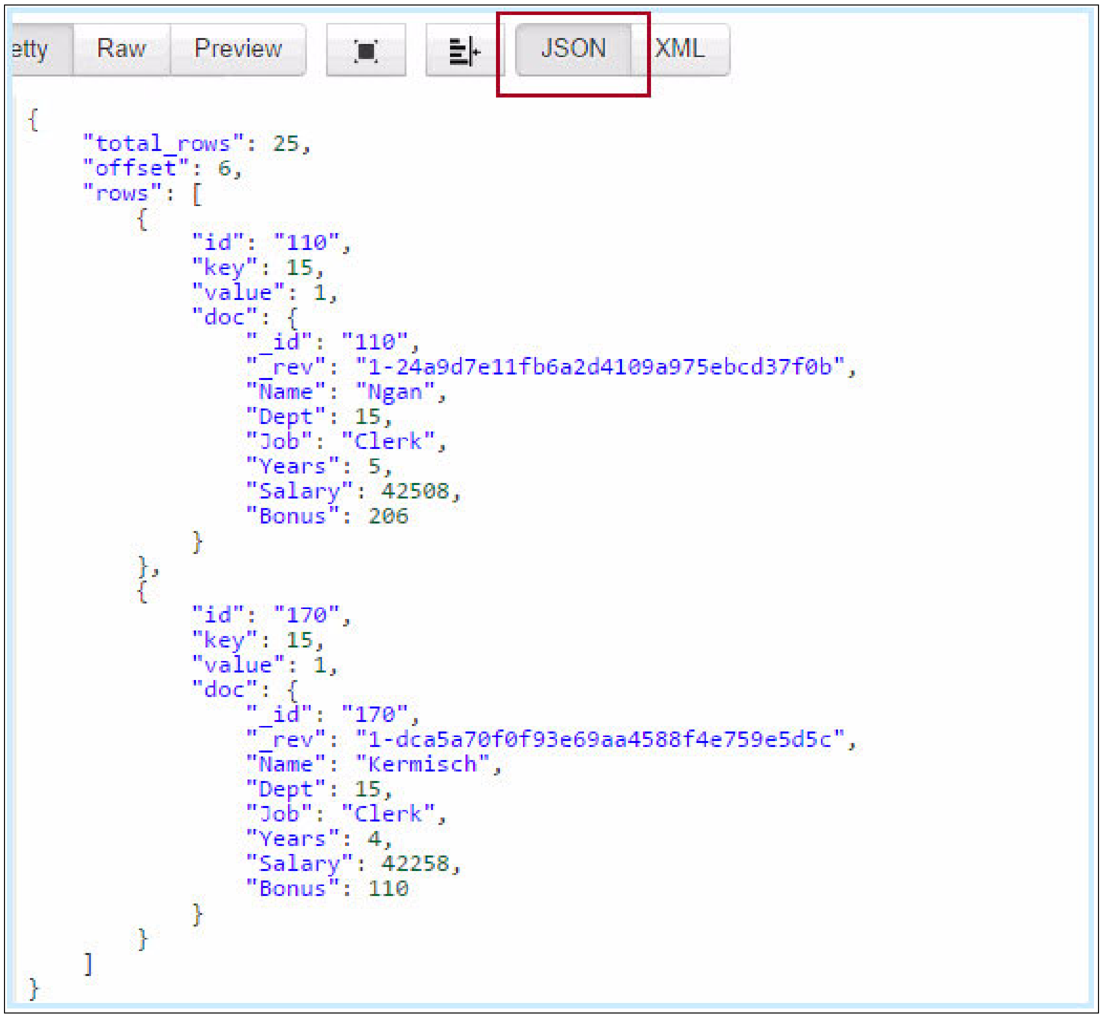
```
{"total_rows":25,"offset":6,"rows":[
{"id":"110","key":15,"value":1},
{"id":"170","key":15,"value":1},
{"id":"50","key":15,"value":1},
{"id":"70","key":15,"value":1}
]}
```

*Figure 5-4 Result of search for documents with Dept number = 15 using MapReduce index byDept*

If you want to get the same result but see all fields of documents, send the same request with the include_docs option equal to TRUE. Also, limit the result to 2:

```
GET
https://redbookid.cloudant.com/staff/_design/views001/_view/byDept?key=15&include_
docs=true&limit=2
```

Figure 5-5 on page 50 shows the response from the database server using JSON format.



```
{
    "total_rows": 25,
    "offset": 6,
    "rows": [
        {
            "id": "110",
            "key": 15,
            "value": 1,
            "doc": {
                "_id": "110",
                "_rev": "1-24a9d7e11fb6a2d4109a975ebcd37f0b",
                "Name": "Ngan",
                "Dept": 15,
                "Job": "Clerk",
                "Years": 5,
                "Salary": 42508,
                "Bonus": 206
            }
        },
        {
            "id": "170",
            "key": 15,
            "value": 1,
            "doc": {
                "_id": "170",
                "_rev": "1-dca5a70f0f93e69aa4588f4e759e5d5c",
                "Name": "Kermisch",
                "Dept": 15,
                "Job": "Clerk",
                "Years": 4,
                "Salary": 42258,
                "Bonus": 110
            }
        }
    ]
}
```

*Figure 5-5   Result of search using MapReduce index byDept and option include_docs=true and limit=2*

When you search using secondary indexes, you can use all options from primary indexes search for _all_docs endpoint plus some other option specific for MapReduce views.

Table 5-1 lists options for secondary searches.

*Table 5-1   Options for secondary index searches*

| **key** | **Retrieve document with specific _id** |
|---|---|
| startkey, endkey | Retrieve range of documents based on the primary key |
| include_docs=(**true** \| false) | Include the document body into results, default is “false” |
| descending=(**true** \| false) | sort in descending order, default is “false” |
| limit=N | Limit result to N documents |
| skip=N | Skip over the first N documents |
| stale=ok | Speed out process, as it does not wait for index building to complete |
| reduce=(true \| **false**) | Turn off reduce step, default is “true” |
| group= (**true** \| false) | Returns results per key. Invalid for map-only or reduce=false views |

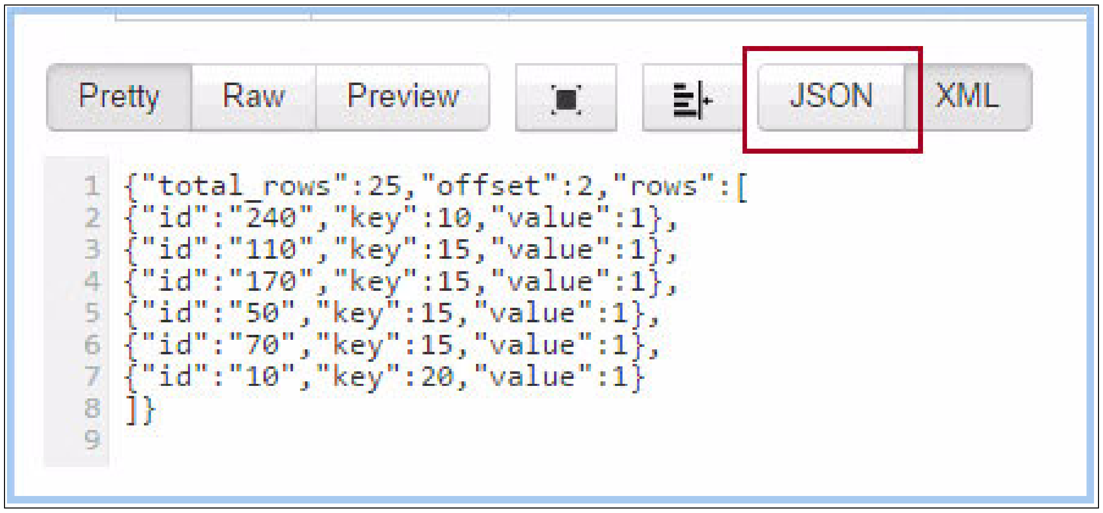
| key | Retrieve document with specific _id |
|---|---|
| group_level = N | Determine group level for complex key |

**Table 5-1. A list of all options for MapReduce searches**

For example, if you want to search for Dept 10 - 20, but only return six rows, skipping the first two, you can submit the following request:

```
GET
https://redbookid.cloudant.com/staff/_design/views001/_view/byDept?startkey=10&end
key=20&skip=2&limit=6
```

Figure 5-6 shows the response from the database server using JSON format.



```
{"total_rows":25,"offset":2,"rows":[
{"id":"240","key":10,"value":1},
{"id":"110","key":15,"value":1},
{"id":"170","key":15,"value":1},
{"id":"50","key":15,"value":1},
{"id":"70","key":15,"value":1},
{"id":"10","key":20,"value":1}
]}
```

*Figure 5-6 Result of search using MapReduce index byDept with multiple option*

## 5.1.3 Reduce function

The Reduce function processes the output of the map operation to aggregate results according to business requirements.

Reduces are called with three parameters: keys, values, and rereduce.

- *keys* will be a list of keys as emitted by the map or, if rereduce is true, null.
- *values* will be a list of values for each element in keys, or if rereduce is true, a list of results from previous reduce functions.
- *rereduce* will be true or false.

If possible, do not use customs reduce functions. We recommend the use of the built-in functions outlined in the next section. They are simpler, faster, and save you time.

## 5.1.4 Built-in reduces

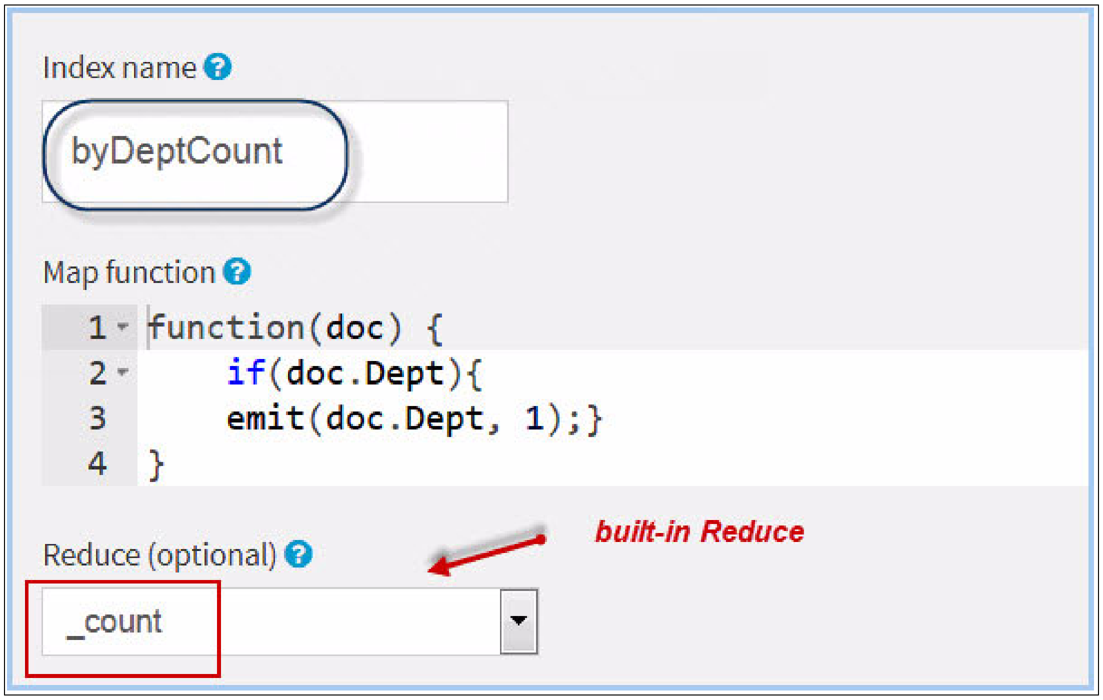
Cloudant exposes several built-in reduce functions, which because they are written in Cloudant native Erlang rather than JavaScript, run much faster than custom functions. Those reduces are _count, _sum, _stats:

| | |
|---|---|
| **_count** | Reports the number of docs emitted by the map function, regardless of the emitted values' types. |

| | |
|---|---|
| **_sum** | Given an array of numeric values, _sum, sums them up. We are going to use this function to demonstrate how you can report a summary of salaries for each department. |
| **_stats** | Produces a JSON structure containing the sum, count, min, max, and sum squared. |

Also, like _sum, the _stats function only deals with numeric values and arrays of numbers; you get an error if you pass a string or an object as the argument for the function.

To demonstrate how it works, create an index byDeptCount that uses the same map function as the previously created byDept index but with built-in reduce. In the database view of Cloudant Dashboard, select “New” to create a new secondary index, specify a name for the index, and use the same Java Script function to emit Dept, but also select _count for the Reduce function. This is shown in Figure 5-7.



byDeptCount

```
function(doc) {
     if(doc.Dept){
     emit(doc.Dept, 1);}
}
```

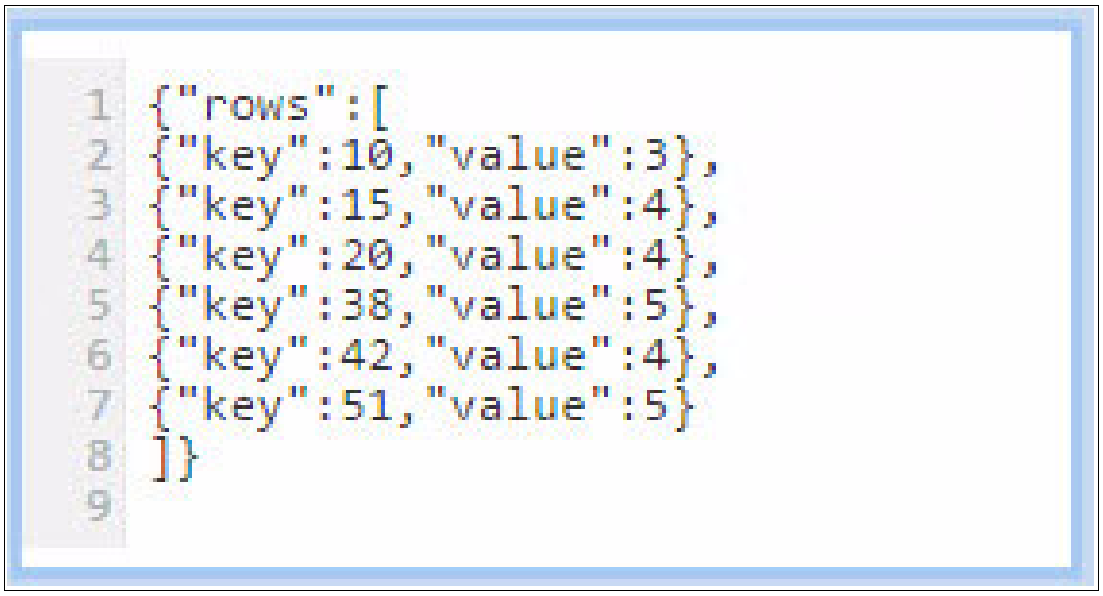
_count

*Figure 5-7 Creating secondary index byDeptCount with built-in Reduce function*

## 5.1.5 Querying data

Now you can use the byDeptCount index to count the number of employees for each department. If you group results by key it returns the number of documents that are emitted for each key (Department number in this case). This is depicted in Figure 5-8 on page 53. Send the following request:

```
GET
https://redbookid.cloudant.com/staff/_design/views001/_view/byDeptCount?group=true
```

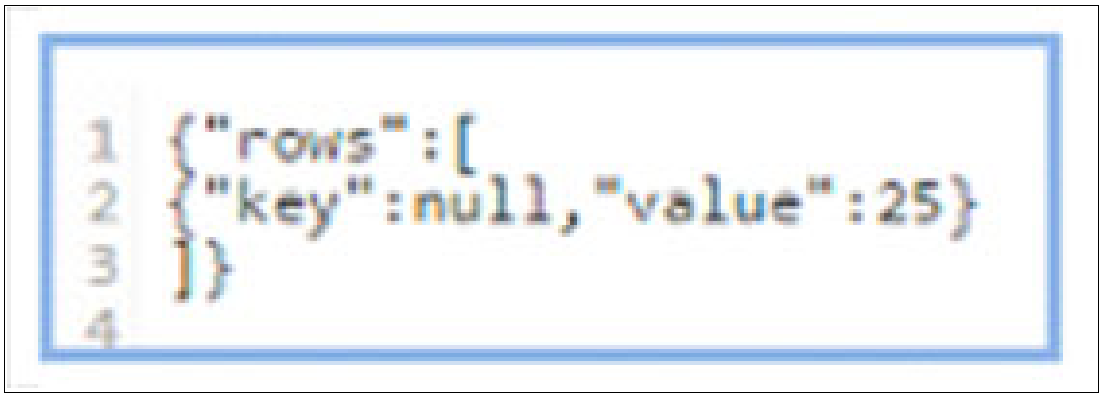


```
{"rows":[
{"key":10,"value":3},
{"key":15,"value":4},
{"key":20,"value":4},
{"key":38,"value":5},
{"key":42,"value":4},
{"key":51,"value":5}
]}
```

*Figure 5-8 Shows the database server response and returns the number of employees in each department*

If you send the same request without group=true option, the database server returns the number of documents in the database, as shown in Figure 5-9.

```
GET https://redbookid.cloudant.com/staff/_design/views001/_view/byDeptCount
```

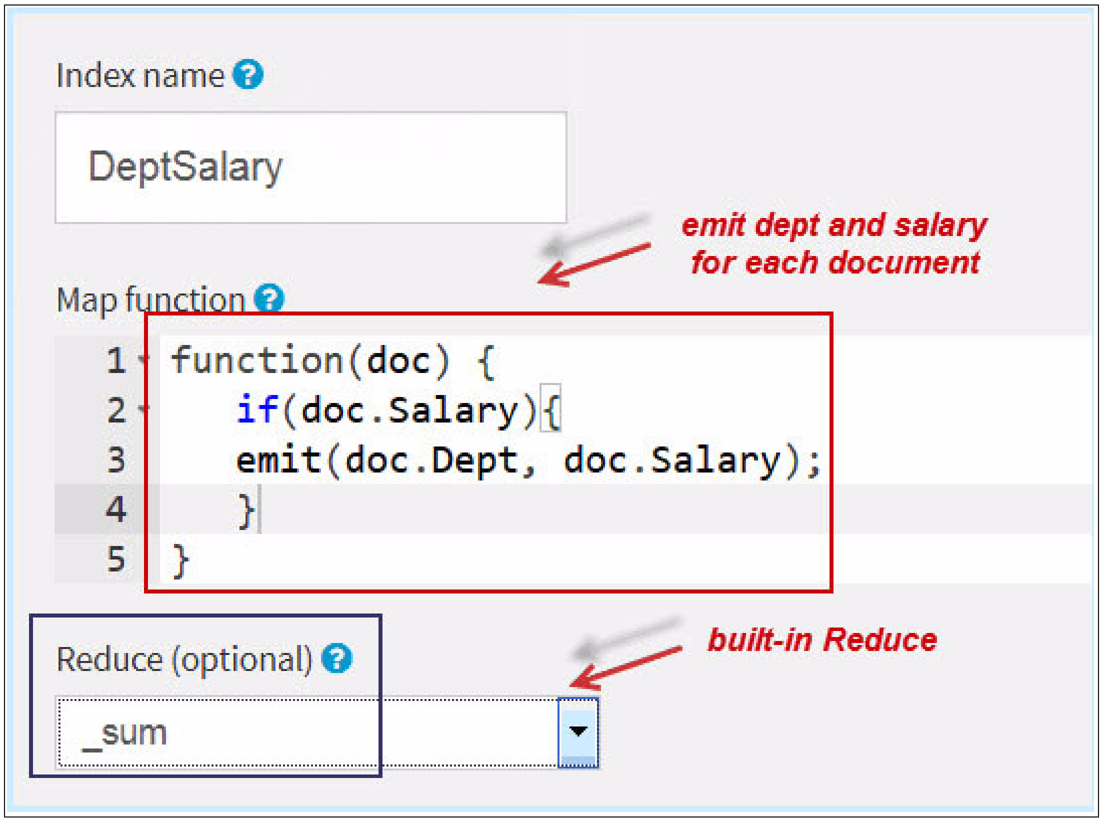


```
{"rows":[
{"key":null,"value":25}
]}
```

*Figure 5-9 Shows the database server response and returns the number of documents in the database*

Let us examine another example of using MapReduce with the _sum function for Reduce. For example, we want to report a summary of salaries for each department. To do so, we can create a DeptSalary index using the _sum Reduce function.

Figure 5-10 on page 54 shows the DeptIndex create window.


Index name

DeptSalary

Map function

```
function(doc) {
    if(doc.Salary){
    emit(doc.Dept, doc.Salary);
    }
}
```

Reduce (optional)

_sum

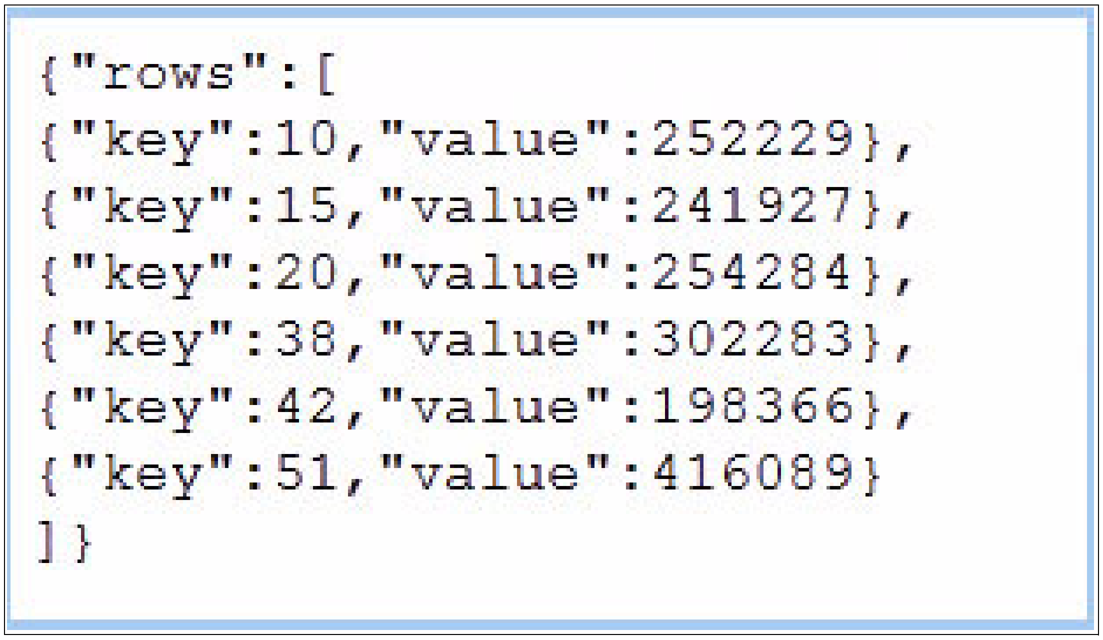
*Figure 5-10* MapReduce index

Use this index to get a summary of salaries for each department.

To do that, issue the following request:

```
GET
https://redbookid.cloudant.com/staff/_design/views001/_view/DeptSalary?group_level
=1
```

Figure 5-11 on page 55 shows the database response.



```
{"rows":[
{"key":10,"value":252229},
{"key":15,"value":241927},
{"key":20,"value":254284},
{"key":38,"value":302283},
{"key":42,"value":198366},
{"key":51,"value":416089}
]}
```

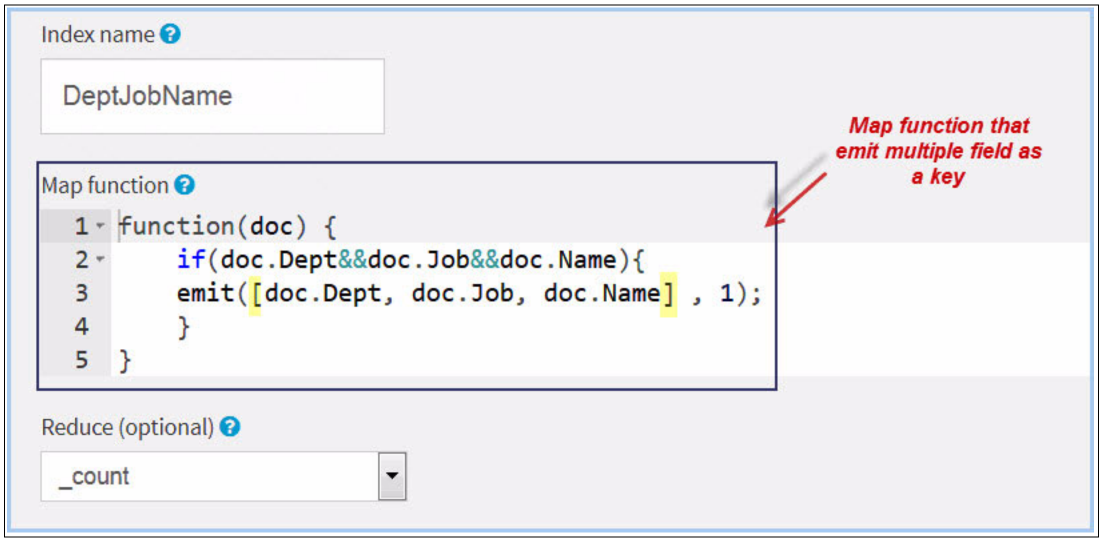
*Figure 5-11 Shows the database server response and returns the salaries sum for each department*

**Reduce=false**

You can use MapReduce views when you want only the results of the map function, even though you added a reduce function to your view. You do not need to write another view for that. Add the **reduce=false** option to the query to turn off the reduce function.

## 5.1.6 Complex indexes

In our previous example, we used a view with an emitted single field key. A view's key can be any valid JSON data structure. We can define a view that uses multiple fields to use as a secondary index. Create a view that will not only derive the number of employees per department, but also group employees by job title. Figure 5-12 on page 56 shows how you create the index DeptJobName.



```
function(doc) {
    if(doc.Dept&&doc.Job&&doc.Name){
    emit([doc.Dept, doc.Job, doc.Name] , 1);
    }
}
```

*Figure 5-12 DeptJobName index*

If you want to emit multiple fields as a key to the view, you need to use square brackets as shown in Figure 5-12.

Now, query this view with the ***reduce=false*** option. To do that, send the following request:

```
GET
https://redbookid.cloudant.com/staff/_design/views001/_view/DeptJobName?reduce=fal
se
```

The results are shown in Figure 5-13 on page 57.

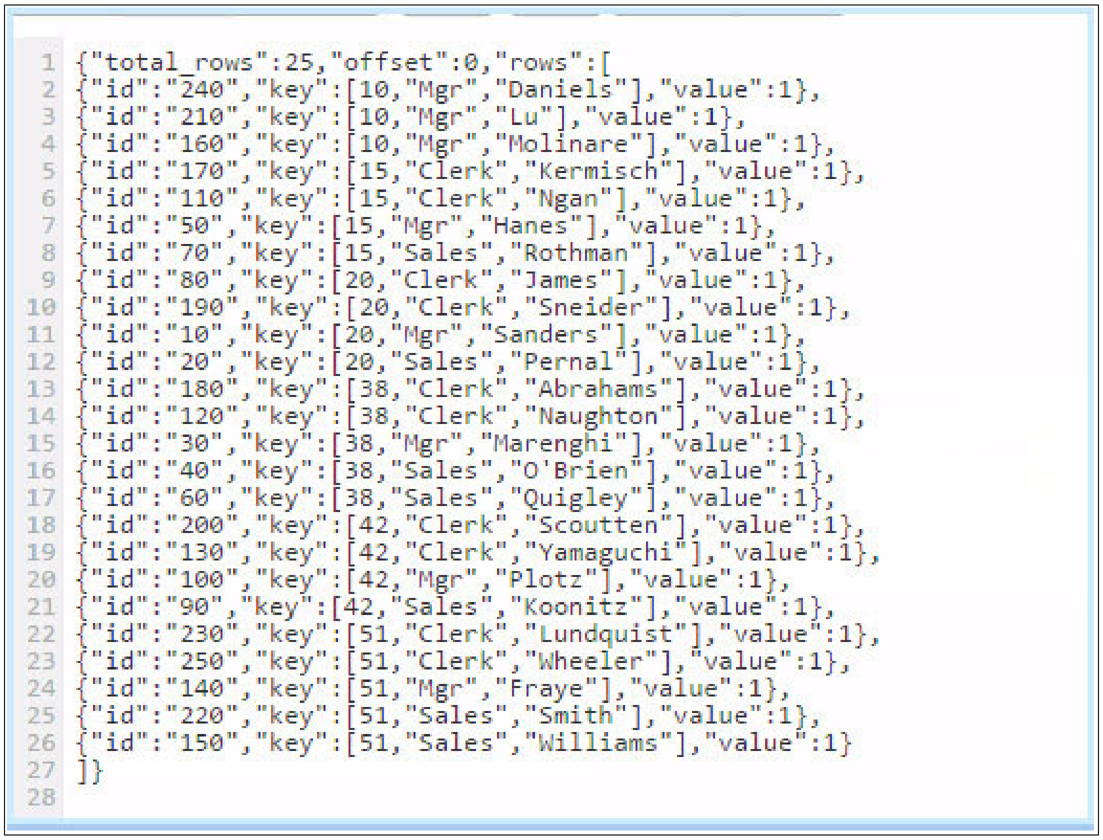

```
{"total_rows":25,"offset":0,"rows":[
{"id":"240","key":[10,"Mgr","Daniels"],"value":1},
{"id":"210","key":[10,"Mgr","Lu"],"value":1},
{"id":"160","key":[10,"Mgr","Molinare"],"value":1},
{"id":"170","key":[15,"Clerk","Kermisch"],"value":1},
{"id":"110","key":[15,"Clerk","Ngan"],"value":1},
{"id":"50","key":[15,"Mgr","Hanes"],"value":1},
{"id":"70","key":[15,"Sales","Rothman"],"value":1},
{"id":"80","key":[20,"Clerk","James"],"value":1},
{"id":"190","key":[20,"Clerk","Sneider"],"value":1},
{"id":"10","key":[20,"Mgr","Sanders"],"value":1},
{"id":"20","key":[20,"Sales","Pernal"],"value":1},
{"id":"180","key":[38,"Clerk","Abrahams"],"value":1},
{"id":"120","key":[38,"Clerk","Naughton"],"value":1},
{"id":"30","key":[38,"Mgr","Marenghi"],"value":1},
{"id":"40","key":[38,"Sales","O'Brien"],"value":1},
{"id":"60","key":[38,"Sales","Quigley"],"value":1},
{"id":"200","key":[42,"Clerk","Scoutten"],"value":1},
{"id":"130","key":[42,"Clerk","Yamaguchi"],"value":1},
{"id":"100","key":[42,"Mgr","Plotz"],"value":1},
{"id":"90","key":[42,"Sales","Koonitz"],"value":1},
{"id":"230","key":[51,"Clerk","Lundquist"],"value":1},
{"id":"250","key":[51,"Clerk","Wheeler"],"value":1},
{"id":"140","key":[51,"Mgr","Fraye"],"value":1},
{"id":"220","key":[51,"Sales","Smith"],"value":1},
{"id":"150","key":[51,"Sales","Williams"],"value":1}
]}
```

*Figure 5-13 Results of the query with the* ***reduce=false*** *option. Department number, job title, and name of employee are emitted as key for each document*

## 5.1.7 Using group_level option

Now query that index, but use the reduce=true (you can omit it as reduce=true is the default) option and **group_level** option. First, use **group_level = 1**.

```
GET
https://redbookid.cloudant.com/staff/_design/views001/_view/DeptJobName?group_leve
l=1
```

The results are shown in Figure 5-14.

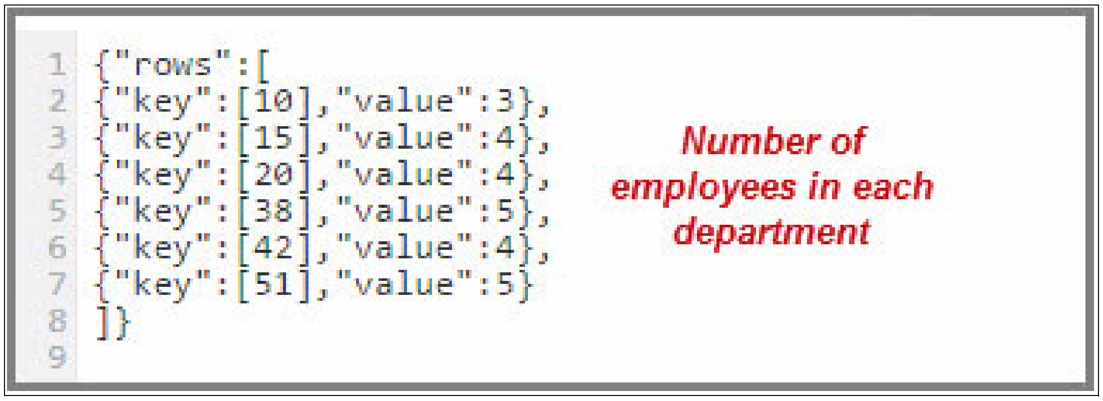
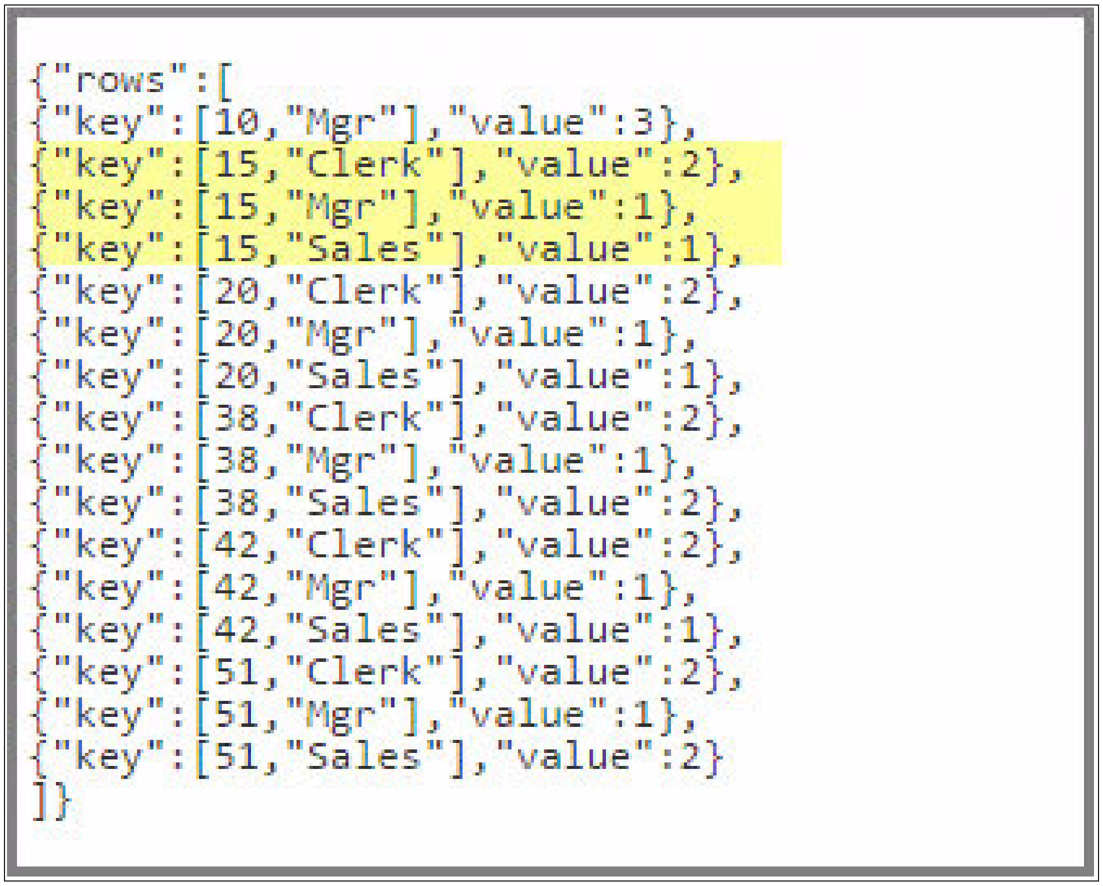

```
{"rows":[
{"key":[10],"value":3},
{"key":[15],"value":4},
{"key":[20],"value":4},
{"key":[38],"value":5},
{"key":[42],"value":4},
{"key":[51],"value":5}
]}
```

*Figure 5-14 Results of querying Dept_JobIndex with group_level=1*

Now see the results if you use group_level = 2. Send the following request:

```
GET
https://redbookid.cloudant.com/staff/_design/views001/_view/DeptJobName?group_leve
l=2
```

The results are shown in Figure 5-15.

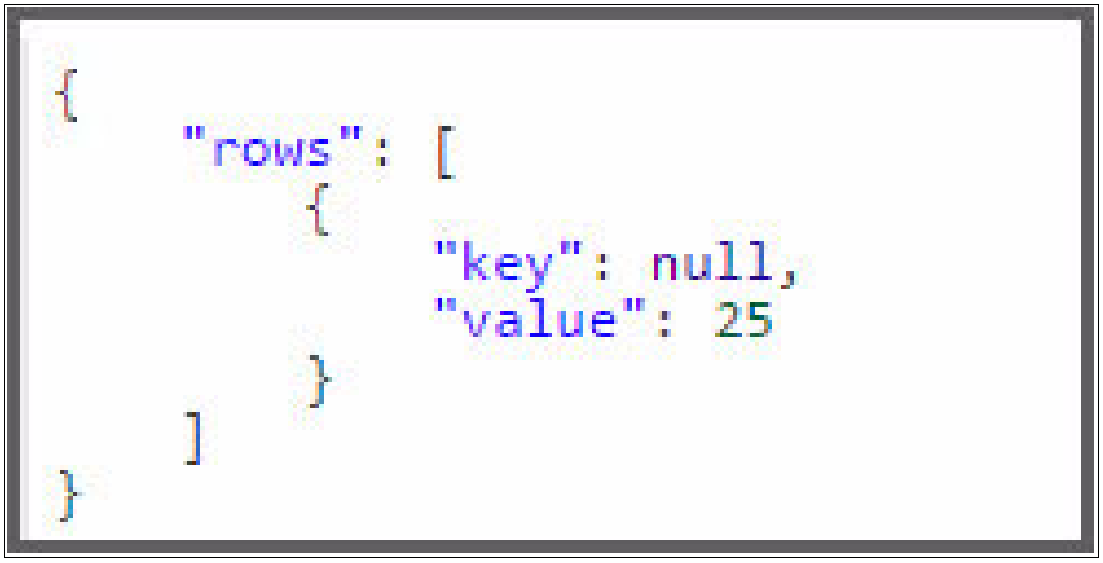


```
{"rows":[
{"key":[10,"Mgr"],"value":3},
{"key":[15,"Clerk"],"value":2},
{"key":[15,"Mgr"],"value":1},
{"key":[15,"Sales"],"value":1},
{"key":[20,"Clerk"],"value":2},
{"key":[20,"Mgr"],"value":1},
{"key":[20,"Sales"],"value":1},
{"key":[38,"Clerk"],"value":2},
{"key":[38,"Mgr"],"value":1},
{"key":[38,"Sales"],"value":2},
{"key":[42,"Clerk"],"value":2},
{"key":[42,"Mgr"],"value":1},
{"key":[42,"Sales"],"value":1},
{"key":[51,"Clerk"],"value":2},
{"key":[51,"Mgr"],"value":1},
{"key":[51,"Sales"],"value":2}
]}
```

*Figure 5-15 By using group_level parameter to group documents by different fields in complex keys, the database server returns the number of employees with a particular job for each department. For example, department 15 has one manager, two clerks, and 1 sales person*

Remember when querying complex indexes that **group_level=0** or **reduce=true** are default parameters and will return Reduce function for all documents. Figure 5-16 on page 59 shows the results for the DeptJobIndex view.

```
GET https://redbookid.cloudant.com/staff/_design/views001/_view/DeptJobName
```



```
{
    "rows": [
        {
            "key": null,
            "value": 25
        }
    ]
}
```

*Figure 5-16 Result of the DeptJobName view*

To summarize map functions are JavaScript functions that take a document as an argument and emit a key/value pair, where the key could contain one or many fields of that document. Cloudant stores emitted rows by constructing a sorted B-tree index, so lookups are efficient.

The Reduce function is run on every node in the tree in order to calculate the final reduce value.

6

# Building and querying the search index

In this chapter, we provide you with an overview of creating search indexes allowing you to search document text. We describe the foundation that Cloudant search is built on and provide a number of examples how to create and query a search index. We also describe what a search analyzer is, a few of the generic analyzers that are available, and how you can implement foreign language searches. We discuss query syntax, pagination, sorting, bookmarks, and use of the application programming interface (API) with a few examples. Cloudant stores data in self-describing JavaScript Object Notation (JSON) documents, which makes every document in the Cloudant database accessible as JSON via a URL making Cloudant very powerful for web and mobile applications.

## 6.1 The foundation of Cloudant Search

As you learned in previous chapters, the primary index functions as a key-value store: it uses a document's ID as its key. By default, the primary index is part of every Cloudant database. Secondary indexes use incremental MapReduce to build indexes, which can be used to query against any fields you define as a part of that index. Data can also be aggregated using an optional Reduce function that follows the mandatory Map phase. It is also possible to chain these MapReduce jobs, having the output of one MapReduce job feed as input to a subsequent Map (and optional Reduce) task, using the appropriately named Chainable MapReduce functionality supported within Cloudant.

Since Cloudant Search is built on Apache Lucene, we provide you with a brief overview of Lucene. Lucene gives you ad hoc query capability beyond what is possible with Cloudant primary and secondary indexes. Lucene's ability to create indexes based on all text within documents stored within the Cloudant database is extremely valuable for developers. Lucene is an open source, rich text search engine library (API) written in Java. Within the Cloudant Dashboard, you are provided with a simple interface for allowing users to define what fields or segments of their data should be indexed for query, as well as "power-user" features like language choice, specific Lucene analyzers, and so on. The interface is nearly identical to how one writes Map and Reduce functions, except you substitute index() in place of the regular emit() function for each property of a document that you want to be indexed.

An analyzer tokenizes text by performing any number of operations on it, which could include: extracting words, discarding punctuation, removing accents from characters, lower-casing (also called normalizing), removing common words, reducing words to a root form (stemming), or changing words into the basic form (lemmatization). This process is also called *tokenization*, and the chunks of text pulled from a stream of text are called *tokens*. For example, a search for "house" can also match results such as "houses", "housed", and "housing". Lucene analyzers also allow Cloudant to handle various languages. Once the data has been analyzed and search indexes have been created, you can query your data in various ways: use sort, group, and faceting to summarize the results according to your requirements.

Since Cloudant Search is a full text indexing and query system that is based on embedded Lucene libraries, Cloudant Search is ideal for arbitrary queries on high-dimensional data. Cloudant indexes your data in real time and provides a simple, scalable, and fast search engine that can process arbitrarily large volumes of data or concurrent queries without requiring users to worry about scaling concerns. Cloudant Search is an ideal replacement for the limited in-database search capabilities of other databases that "bolt-on" external search systems, or the multi-dimensional query languages like MongoDB.

Cloudant Search indexes are defined by developers using JavaScript and stored in _design documents. The index is built by scanning through all the documents in a database. One can index a single field or all the fields within a document. Cloudant Search is good for both ad hoc and multi-field queries. Keep in mind that Cloudant indexes take up storage space. When building a search index, you specify what query parameters can be used and whether or not data is returned in a search. Cloudant Search also supports faceted search by enabling optional index parameters at the time the index is built.

Search indexes are treated independently of one another: if the index function that defines the search index within the design document remains unchanged, the index will not need to be rebuilt. However, if the entire design document is replaced, or if the index function defining the Lucene search index is adjusted, the said index will need to be recomputed. Keep this in mind when making revisions to your design documents.

## 6.2 Examples of creating and querying a search index

Index Create example:

```
   function(doc) {
      index(‘name’,doc_id);
      index(‘age’,doc.age);
      for (var i in doc.information) {
index(‘info’,doc.information[i],
   {“store”:true},{facet”:true});
       }
                   }
```

Querying a Search Index:

```
   /<database>/_design/<name>/_search/<searchname>?
```

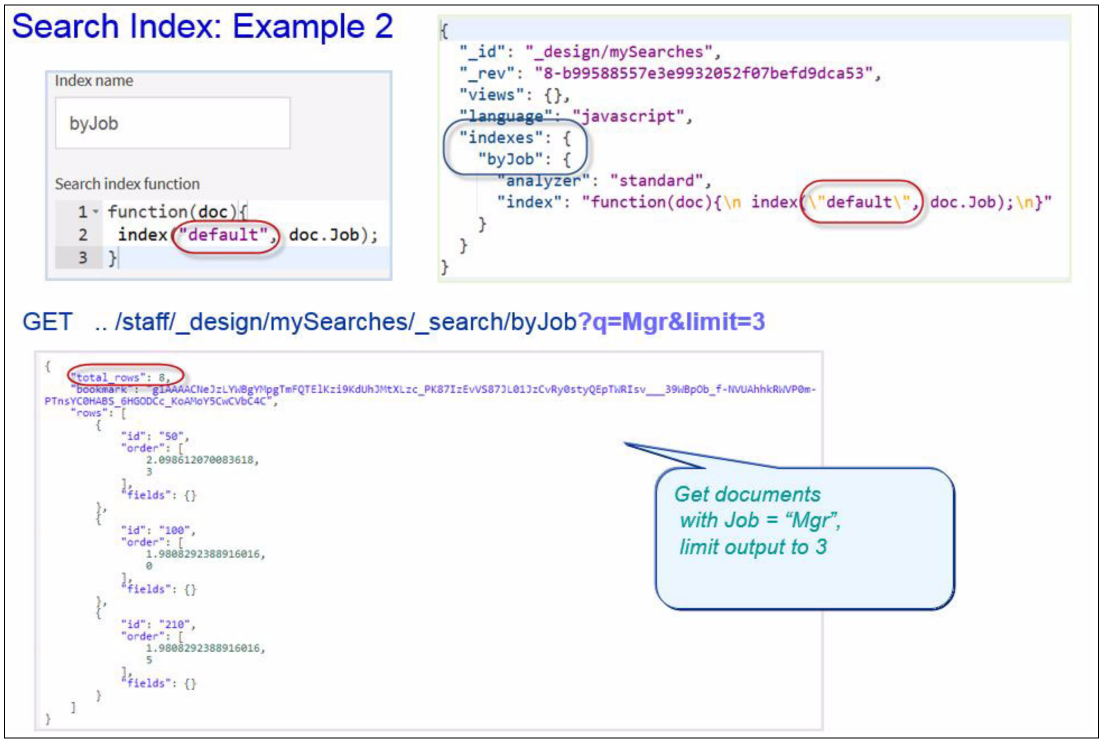
query or q with the following parameters:

- limit (Defaults to 200 as a maximum)
- include_docs
- bookmark (Allows for paging through results)
- stale=ok (Speed up response as it does not wait for index building to complete)
- sort
  - of form “fieldname<type>”or “-fieldname<type>”, for example, “foo<string>”or “bar<number>”
  - Default sort is by relevance
  - Can also sort by distance using Geo-Location Search not covered here

Given that Cloudant Search is powered by Apache Lucene, Lucene syntax is supported. Reference the Apache Lucene document (ref 1)*:

- Query syntax (q or query parameter):
- Default parameter: ?q=Boston
- Wildcards: ?q=passion:backpack*
- Match all: ?q=*:*
- Ranges: ?q=age:[30 TO 39] ?q=name:{Marina TO Steve}
- [ ] -Inclusive
- { } -Exclusive
- Fuzzy Searches: ?q=description:dbname~
- Proximity Searches: ?q=description:”CouchDB apache”~10
- Find fields where CouchDB and Apache are within 10 words of each other
- Boosting a Term: ?q=description:”CouchDB^4 apache”
- Boolean Operators: AND, “+”, OR, NOT, “-”(ALL CAPS)

Figure 6-1 on page 64 shows Search Example 2.


Search Index: Example 2

Index name

byJob

Search index function

```
function(doc){
 index("default", doc.Job);
}
```

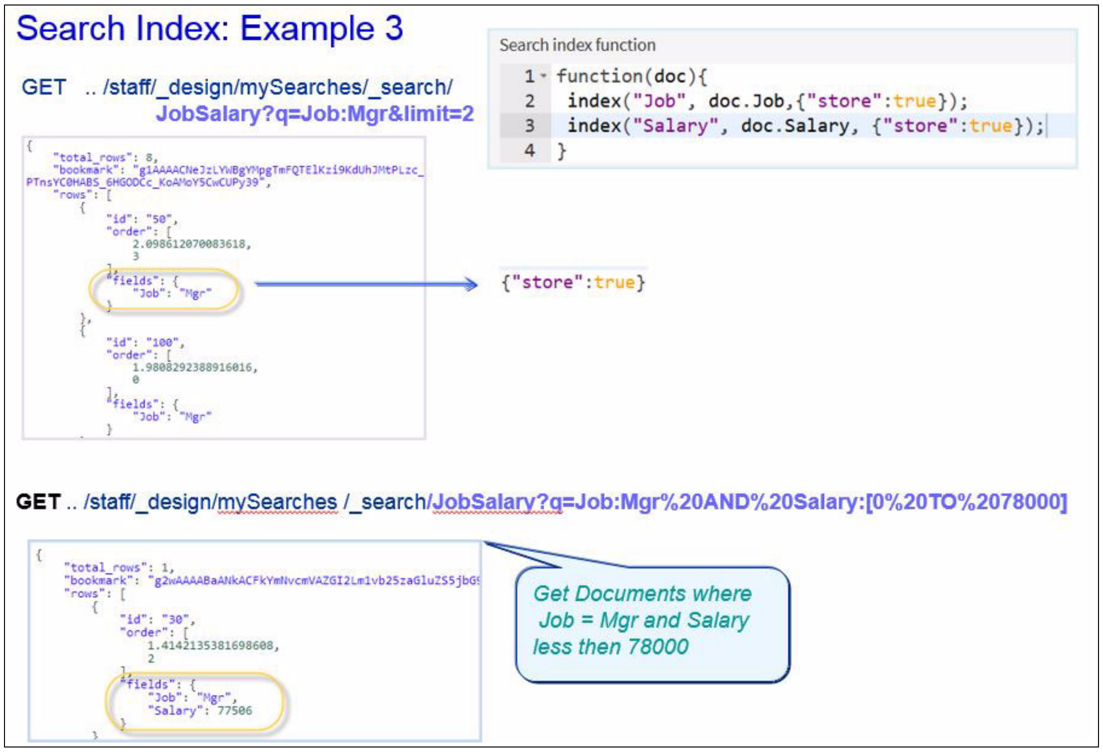
```
{
  "_id": "_design/mySearches",
  "_rev": "8-b99588557e3e9932052f07befd9dca53",
  "views": {},
  "language": "javascript",
  "indexes": {
    "byJob": {
      "analyzer": "standard",
      "index": "function(doc){\n index(\"default\", doc.Job);\n}"
    }
  }
}
```

GET .. /staff/_design/mySearches/_search/byJob?q=Mgr&limit=3

```
{
    "total_rows": 8,
    "bookmark": "g1AAAACNeJzLYWBgYMpgTmFQTElKzi9KdUhJMtXLzc_PK87IzEvVS87JL01JzCvRy0styQEpTWRIsv___39WBpOb_f-NVUAhhkRWVP0m-PTnsYC0HABS_6HGODCc_KoAMoY5CwCVbC4C",
    "rows": [
        {
            "id": "50",
            "order": [
                2.098612070083618,
                3
            ],
            "fields": {}
        },
        {
            "id": "100",
            "order": [
                1.9808292388916016,
                0
            ],
            "fields": {}
        },
        {
            "id": "210",
            "order": [
                1.9808292388916016,
                5
            ],
            "fields": {}
        }
    ]
}
```

Figure 6-1   Search Example 2

Figure 6-2 shows Search Example 3.


Search Index: Example 3

GET .. /staff/_design/mySearches/_search/JobSalary?q=Job:Mgr&limit=2

Search index function

```
function(doc){
 index("Job", doc.Job,{"store":true});
 index("Salary", doc.Salary, {"store":true});
}
```

```
{
    "total_rows": 8,
    "bookmark": "g1AAAACNeJzLYWBgYMpgTmFQTElKzi9KdUhJMtPLzc_PTnsYC0HABS_6HGODCc_KoAMoY5CwCUPy39",
    "rows": [
        {
            "id": "50",
            "order": [
                2.098612070083618,
                3
            ],
            "fields": {
                "Job": "Mgr"
            }
        },
        {
            "id": "100",
            "order": [
                1.9808292388916016,
                0
            ],
            "fields": {
                "Job": "Mgr"
            }
```

```
{"store":true}
```

GET .. /staff/_design/mySearches /_search/JobSalary?q=Job:Mgr%20AND%20Salary:[0%20TO%2078000]

```
{
    "total_rows": 1,
    "bookmark": "g2wAAAABaANkACFkYmNvcmVAZGI2Lm1vb25zaGluZS5jbG
    "rows": [
        {
            "id": "30",
            "order": [
                1.4142135381698608,
                2
            ],
            "fields": {
                "Job": "Mgr",
                "Salary": 77506
            }
        }
```

Figure 6-2   Search Example 3

Figure 6-3 on page 65 shows a Faceted Search Example.

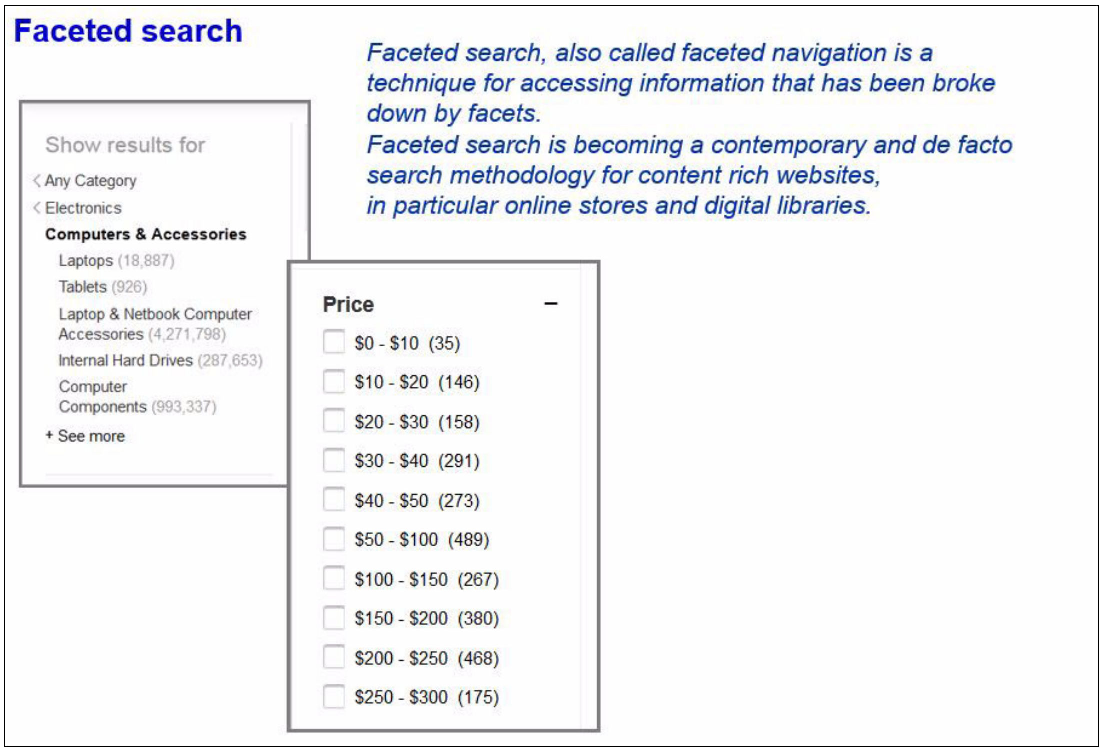

Faceted search
Faceted search, also called faceted navigation is a technique for accessing information that has been broke down by facets.
Faceted search is becoming a contemporary and de facto search methodology for content rich websites, in particular online stores and digital libraries.
Show results for
Any Category
Electronics
Computers & Accessories
Laptops (18,887)
Tablets (926)
Laptop & Netbook Computer Accessories (4,271,798)
Internal Hard Drives (287,653)
Computer Components (993,337)
+ See more
Price
$0 - $10 (35)
$10 - $20 (146)
$20 - $30 (158)
$30 - $40 (291)
$40 - $50 (273)
$50 - $100 (489)
$100 - $150 (267)
$150 - $200 (380)
$200 - $250 (468)
$250 - $300 (175)

*Figure 6-3* Faceted Search Example

```
Ref:
(1)Apache Lucene Doc –
https://Lucene.apache.org/core/2_9_4/queryparsersyntax.html
From online documentation
```

## 6.3 Further information and examples

The built-in indexing function takes three arguments: the Lucene field; the value for that field; and an optional “options” object.

```
function(doc){
  index("name", doc.name, {"store": true, "index": false});
}
```

The options object has two Boolean keys: store and index.

Option Description Values Default:

- store: If true, the value will be returned in the search result; if false, the value will not be returned in the search result. (Values: true, false) (Default = false)
- index: Whether the data is indexed. If set to false, the data cannot be used for searches, but it can still be retrieved from the index if store is set to true. (Values: true, false) (Default = true)
- facet: Whether to enable faceting. (Values: true, false) (Default = false)
- boost: To increase the relevance of the data being indexed in search results, supply a number greater than 1.0. Relevance is adjusted by the given factor. Any positive floating point number. 1.0 (no boosting)

Calling the index function with both options set to false has no effect.

## 6.3.1 Search analyzers

Analyzers define how to extract index terms from text, which you might need to do if your application needs to index Chinese, for example). Here is the list of generic analyzers supported by Cloudant Search. See further down for language-specific analyzers.

Here is the list of generic analyzers:

- *standard*: This is the default analyzer and implements the Word Break rules from the Unicode Text Segmentation algorithm, as specified in Unicode Standard Annex #29.
- *email*: Like standard, but tries harder to match an email address as a complete token.
- *keyword*: Input is not tokenized at all.
- *simple*: Divides text at non-letters.
- *whitespace*: Divides text at whitespace boundaries.
- *classic*: The standard Lucene analyzer circa release 3.1. You know whether you need it.

You can choose which analyzer is used within your index function by changing the index definition in the design document.

Defining an analyzer:

```
"indexes": { "mysearch" : {
  "analyzer": "whitespace", "index": "function(doc){ ... }" },
  }
```

**Note:** Changing the analyzer causes the index to be rebuilt. (Also note that queries against a given index are run with the same analyzer as defined by the function.)

## 6.3.2 Language-specific Analyzers

We provide many analyzers for specific languages. These analyzers omit words that are common to specific languages, as these tend to make poor indexing candidates and cause considerable index bloat. Many of these analyzers also perform stemming, where common word prefixes or suffixes are removed.

### Per-Field Analyzer

Sometimes a single analyzer is not enough. You can use the per-field analyzer to configure different analyzers for different field names:

*Per-field analysis*

```
"indexes": {
  "mysearch" : {
    "analyzer": {
      "name": "perfield",
      "default": "english",
      "fields": {
        "spanish": "spanish",
        "german": "german"
      }
    },
    "index": "function(doc){ ... }"
  }
 }
```

## 6.3.3 Stop words

You may want to define a set of words that do not get indexed. These are called *stop words*. You define stop words in the design document by turning the analyzer string into an object:

A simple stop words example

```
"indexes": {
  "mysearch" : {
    "analyzer": {"name": "portuguese", "stopwords":["foo", "bar", "baz"]},
    "index": "function(doc){ ... }"
  },
}
```

Note that keyword, simple, and whitespace analyzers do not support stop words.

## 6.3.4 API

As you probably noticed above, the search URL requires a q (or query) query string. This is the query that is passed on to the search index. There are two data types supported by search: string and number. The data type is auto detected. If you need to pass a number in as a string you will need to quote it, for example, q="12".

The search URL can optionally take limit, include_docs, stale (which have the same behavior as those in the primary and secondary indexes) sort, and bookmark.

## 6.3.5 Pagination, sorting, and bookmarks

- Pagination and sorting

  Bookmarks allow you to efficiently skip through results you have already seen. All search results include a bookmark in their JSON response. By passing this value to the search URL via the bookmark query parameter, you will see the next page of results.

  Search results can be sorted ascending or descending by any numeric or string field in the index. Sort order is set by the sort query parameter, which takes a JSON string or list as its parameter. If the field is a string field, you have to add <string> to the end of the string. If you wanted to sort by age, you would query your search index with ?sort="age". If you wanted to sort descending, you would use ?sort="-age". If you wanted to search by name, you would use ?sort="name<string>". Sorts can be applied to multiple fields, for instance ?sort=["-age", "height"] would sort by age descending then height ascending.

- Sorting by Relevance

  The default sort order (when you do not supply a sort parameter) is relevance, the highest scoring matches are returned first. If you specify a sort order, matches are returned in that order, ignoring relevance. If you want to include the relevance ordering in your sort order, you can use the special fields -<score> and <score>.

- Sorting By Distance

  In addition to sorting by indexed fields, you can sort by distance from a point chosen at query time. You will need to index two numeric fields (representing the longitude and latitude of whatever you are indexing):

  ```
  function(doc) {
    index("mylon", doc.longitude);
    index("mylat", doc.latitude);
  }
  ```

You can then query using the special <distance...> sort field, which takes five parameters:

- longitude field name

  The name of your longitude field ("mylon" in this example)
- latitude field name

  The name of your latitude field ("mylat" in this example)
- longitude of origin

  The longitude of the place you want to sort by distance from
- latitude of origin

  The latitude of the place you want to sort by distance from units
- The units to use (kilometers (km) and miles (mi)). The distance itself is returned in the order field

An example query to make this clear:

```
?sort="<distance,mylon,mylat,-0.14479689999996026,51.4964609,mi>"
```

You can combine sorting by distance with a bounding box query to perform simple geo operations.

## 6.3.6 Further examples on query syntax

The Cloudant Search query syntax is based on the Lucene syntax. Search queries take the form of name:value (unless the name is omitted, in which case they hit the default field as we demonstrated in the first example, above).

Queries over multiple fields can be logically combined; furthermore, groups and fields can be grouped. The available logical operators are: AND, +, OR, NOT, and -. These operators are case-sensitive. Range queries can run over strings or numbers.

If you want a fuzzy search, you can run a query with ~ to find similar results to, but not necessarily needing to exactly match the search term. For instance, look~ will find terms *book* and *took*.

You can also increase the importance of a search term by using the boost character ^. This results in query matches that contain the term more relevant. For example, Cloudant "data layer"^4 will adjust results containing "data layer" to be four times more relevant. The default boost value is 1. Boost values must be positive, but can be less than 1 (for example, 0.5 to reduce importance).

Wildcard searches are supported for both single (?) and multiple (*) character searches. dat? would match date and data, dat* would match date, data, database, dates, and so on. Wildcards must come after a search term. You cannot do a query such as *base.

Result sets from searches are limited to 200 rows and return 25 rows by default. The number of rows returned can be changed via the limit parameter. The response contains a bookmark. If the bookmark is passed back as a URL parameter, skip through the rows that you have already seen and get the next set of results.

The following characters require escaping if you want to search on them;

```
+ - && || ! ( ) { } [ ] ^ " ~ * ? : \ /
```

Escape these with a preceding backslash character.

## 6.3.7 API demo example

The animals database contains a design document that, among other things, defines a search index over the animal name, diet, minimum length, Latin name, and class.

```
function(doc){
  index("default", doc._id);
  if(doc.min_length){
    index("min_length", doc.min_length, {"store": "yes"});
  }
  if(doc.diet){
    index("diet", doc.diet, {"store": "yes"});
  }
  if (doc.latin_name){
    index("latin_name", doc.latin_name, {"store": "yes"});
  }
  if (doc.class){
    index("class", doc.class, {"store": "yes"});
  }
}
```

With this index, you can run any of these queries:

- Desired result Query
- Birds class:bird
- Animals that begin with the letter "l" l*
- Carnivorous birds class:bird AND diet:carnivore
- Herbivores that start with letter "l" l* AND diet:herbivore
- Medium-sized herbivores min_length:[1 TO 3] AND diet:herbivore
- Herbivores that are 2m long or less diet:herbivore AND min_length:[-Infinity TO 2]
- Mammals that are at least 1.5m long class:mammal AND min_length:[1.5 TO Infinity]
- Find "Meles meles" latin_name:"Meles meles"
- Mammals who are herbivore or carnivore diet:(herbivore OR omnivore) AND class:mammal

**Note:** You can sign in or create a no-cost account to try these searches in the query field, below. (https://cloudant.com) (https://cloudant.com/sign-up)

### The Query

https://[username].Cloudant.com/animaldb/_design/views101/_search/animals?q=class:bird

### Grouping Results

In addition to basic searching, you can also group results by common values of a chosen field using the **`group_field`** parameter. For full details, see Docs.

### Faceted Search

Cloudant Search also supports faceted searching, which allows you to discover aggregate information about all your matches quickly and easily. You can even match all documents (using the special ?q=*:* query syntax) and use the returned facets to refine your query. Indexing a facet is straightforward and is applicable to both strings and numbers:

```
function(doc) {
  index("type", doc.type, {"facet": true});
  index("price", doc.price, {"facet": true});
```

Once indexed, you can find out how many documents you have of any string facet with the **`counts=`** parameter, in addition to any query string you like. Example output for ?q=*:*&counts=["type"] follows:

```
{"total_rows":100000, "bookmark":"g...", "rows":[...],
 "counts":{"type":{"sofa":10.0, "chair":100.0}}

}
```

You can also perform range facet queries on numeric facets using the **`ranges=`** parameter. For example:

```
?q=*:*&ranges={"price":{"cheap":"[0 TO 100]","expensive":"{100 TO Infinity}"}}
```

The range facet syntax reuses the standard Lucene syntax for ranges (inclusive range queries are denoted by square brackets, exclusive range queries are denoted by curly brackets). This will return output like:

```
"ranges":{"price":{"cheap":101.0,"expensive":99899.0}}
```

### Using POST

Some queries can get very long or it can be difficult to URL encode your query correctly. In these cases, you can use a POST instead:

```
{"query":"*:*", "limit":100}
```

The POST method uses the same parameters as we described thus far, but without needing to worry about URL length limits or URL escaping.

## 6.3.8 Example applications

To demonstrate the functionality of search, we pulled together a couple of example applications. If you would like to replicate them into your account you are welcome to do so, but they both contain sizable data sets and will use up a significant number of Cloudant units.

Full text indexing is what Lucene is built for, and Cloudant Search is no different. In this example, we have taken public lobbyist disclosure data set from the US senate. The data set consists of 757,123 individual documents. The decompressed XML documents are 2.5 GB on disk, and the corresponding Cloudant database is only 1.3 GB.

Geo indexing is possible with Cloudant Search. By combining location awareness with other queries, you can build applications that find what a user wants and where a user is located. In this example, we have taken the Simple Geo “places of interest” data set of over 20 million locations and combined it with searches over other values (for example, find restaurants near the office). A simple geo-indexer could not perform these “refined searches” because they require additional dimensions in the query.

## Querying example illustrating Cloudant Search features

Once the data has been analyzed and search indexes have been created, you can query your data in various ways as well as sort, group, and use faceting to summarize the results. In this section, you will find a quick tutorial. It also covers some of the advanced topics like Analyzers, Search Faceting, Sorting search results, Grouping search results, Sorting By Distance.

### *Sample Data*

In this tutorial, we use a database with information about best selling books to illustrate Cloudant Search features. The database contains about 100 JSON documents like this one:

```
{
  "title_English": "A Tale of Two Cities",
  "author": "Charles Dickens",
  "language": "English",
  "year": 1859,
  "sold": 200.0
}
```

### *Design Documents*

The database also contains design documents, which are used to define and create the search indexes.

```
{
    "indexes": {
        "books": {
            "analyzer": {
                "name": "perfield",
                "default": "english",
                "fields": {
                    "author": "whitespace",
                    "title_German": "german",
                    "title_Chinese": "chinese",
                    "title_Italian": "italian",
                    "title_Russian": "russian",
                    "title_Swedish": "swedish",
                    "title_Norwegian": "norwegian",
                    "title_Portuguese": "portuguese",
                    "title_Spanish": "spanish",
                    "title_Dutch": "dutch",
                    "title_French": "french",
                    "title_Japanese": "japanese",
                    "title_Czech": "czech"
                }
            },
            "index": "function(doc) { index('year', doc.year, {'facet': true});
index('sold', doc.sold, {'facet': true}); index('author', doc.author, {'facet':
true, 'store': true}); index('language', doc.language, {'facet': true});
index('title_' + doc.language, doc['title_' + doc.language], {'store': true}); }"
        }
    }
}
```

As you can see, the design document contains one index called “books”. The index has an analyzer and an index function. The analyzer is a per field analyzer, meaning different analyzers can be used for different search fields. Which analyzers are actually used is defined in the “fields” object. If the search uses a field that is not defined in the fields object, the default analyzer is used, in this case “english”. The index function is called for each new or updated document in the database, and defines what data is to be indexed. Since the index function is stored as a string in a JSON document, it cannot be split over multiple lines.

Here is a nicely formatted version:

```
function(doc) {
    index('year', doc.year, { 'facet': true });
    index('sold', doc.sold, { 'facet': true });
    index('author', doc.author, { 'store': true });
    index('language', doc.language, { 'facet': true });
    index('title_' + doc.language, doc['title_' + doc.language], { 'store': true
});
}
```

The function calls index with three arguments. The first is the name of the search field. The second is the data to be indexed and the third is an options object. 'facet': true turns on faceting and 'store': true stores the processed data within the index so that it will be returned with the search result. For more information about design documents and indexing, see the API reference in the online documentation.

### Search queries

Let us finally look at some search queries. Search queries are always HTTP GET requests. The URL for the request contains the database name, the design document ID, the index name, and the Lucene query. Assuming our design document is named “ddoc”, is stored in a database called “books” on docs.Cloudant.com, and we want to query the author field for “John”, this is the full URL of the books index:

`https://docs.Cloudant.com/examples/_design/ddoc/_search/books?q=author:John`

You can run this query in the form below or in a console using curl.

### Lucene query syntax

Lucene’s query syntax is quite powerful, so be sure to have a look at Lucene’s documentation if you need more than the basic features like those shown below.

Search queries take the form of name:value (unless the name is omitted, in which case they hit the default field).

Queries over multiple fields can be logically combined. The available logical operators are: AND, +, OR, NOT, and -. These operators are case-sensitive. Range queries can run over strings or numbers.

If you want a fuzzy search, run a query with ~ to find terms like the search term, for instance look~ will find terms *book* and *took*.

You can also increase the importance of a search term by using the boost character ^. This makes matches containing the term more relevant; for example, Cloudant "data layer"^4 will make results containing “data layer” four times more relevant. The default boost value is 1. Boost values must be positive, but can be less than 1 to reduce importance.

Wildcard searches are supported for both single (?) and multiple (*) character searches. dat? would match date and data; dat* would match date, data, database, dates, and so on. Wildcards must come after a search term; you cannot do a query like *base.

Result sets from searches are limited to 200 rows, and return 25 rows by default. The number of rows returned can be changed via the limit parameter. The response contains a bookmark.

If the bookmark is passed back as a URL parameter, skip through the rows that you have already seen and get the next set of results.

The following characters require escaping if you want to search on them;

```
+ - && || ! ( ) { } [ ] ^ " ~ * ? : \ /
```

Escape these with a preceding backslash character.

### *Search form*

Search query (q)

```
    "total_rows": 1,
    "bookmark":
"g2wAAAABaANkAB5kYmNvcmVAZGI1Lm1hbG9ydC5jbG91ZGFudC5uZXRsAAAAAm4EAAAAAIBuBAD___-_a
mgCRkACik3gAAAAYRtq",
    "rows": [
        {
            "id": "book91",
            "order": [
                2.3175313472747803,
                27
            ],
            "fields": {
                "author": "John Steinbeck",
                "title_English": "The Grapes of Wrath"
            }
        }
    ]
}
```

7

# Cloudant Query

Cloudant Query gives you a declarative way for defining and querying indices within your database. It provides you with a set of RESTful application programming interfaces (APIs) for performing these actions, creating indexes and querying data, from a Cloudant database source. Data can only be queried if there is an appropriate index available. Otherwise, the execution will result in an error informing you of the missing index. Since the index has been built ahead of time, queries made against these indices are guaranteed to have a good performance, even on larger data sets.

*Table 7-1 RESTful API methods used to interact with Cloudant data using Cloudant Query*

| Method | Path | Description |
|---|---|---|
| POST | /dbname/_index | Create new index |
| GET | /dbname/_index | List all Indexes |
| DELETE | /dbname/_index/<$designdoc/ $type/$name> | Delete an index |
| POST | /dbname/_find | Find document using index |

In the following sections, we detail how the API methods outlined in Table 7-1 can be leveraged, with specific examples, to interact with your Cloudant database.

## 7.1 Creating index

To create an index, you need to choose the fields that you want to be indexed and submit a POST request. The general syntax for this request is as follows:

```
POST  …/dbname/_index –d ‘
{ "index":
       { "fields": [" field_name"] },
      " ddoc " : " design_doc_name ",
      "name" : "index_name",
      "type" : "json" } ‘
```

In this scenario, **ddoc** is optional. We supply it to the POST request to specify the name of design document being created. By default, each index will be created in its own design document.

**Name** is also optional, and will be generated automatically if omitted. It is good practice to specify the name for an index in order to improve the clarity of your application. Having a computer-generated string of numbers and characters for a document identifier is less expressive (and not as easy to look up when querying an index) than specifying a custom identifier yourself.

***Type*** is an optional parameter; for now, only “json” type is supported.

For example, to create index on *Dept* field, the following document needs to be posted to the **STAFF** database:

```
POST https://redbookid.cloudant.com/staff/_index  -d ‘
{"index":
  {"fields":["Dept"]},
  "ddoc": "indexes123",
  "name":"DeptIndex"
}’
```

The database server will return the following in response:

```
{“result”:”created”}
```

## 7.2 Querying data

After index is created, you can query the database by POSTing a JavaScript Object Notation (JSON) document to the *_find* endpoint. The general syntax is:

```
POST  …/dbname/_find –d ‘
{ "selector":
       { "field_name": [" selection_criteria"] },
      "fields" : ["_id ", "_rev ", "field1 ", …]
      "sort" : ["field_name", "asc"],
      "limit" : n1,
      "skip" : n2
     }
 } ‘
```

Selector is a JSON object describing criteria used to select documents. It is similar to where statements in SQL. Some examples of selector's syntaxes are:

{"name":"John"} – where name field equals "John"

{"age": {"$gt": 20} } – where age field is greater than 20. Note that you need to use "$" sign to denote operator.

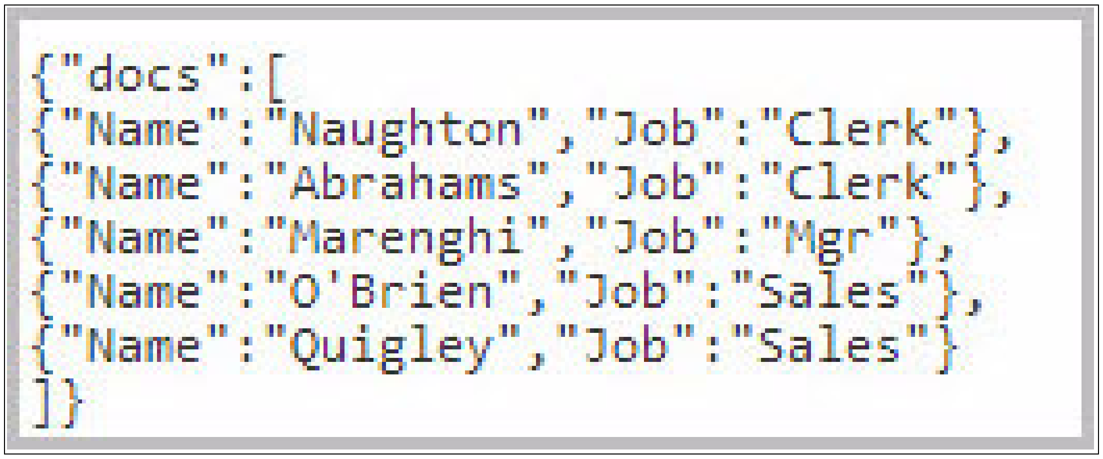
More information regarding selector syntax could be found at the following link:

https://docs.cloudant.com/api/cloudant-query.html#cloudant-query-selectors

In order to query the STAFF database to find all documents, where *Dept* is 38 and return the *Name* and *Job* fields, we can now use the previously built index *DepIndex* by POSTing the following JSON document:

```
POST https://redbookid.cloudant.com/staff/_find  -d '
{"selector":
 {"Dept": 38 },
 "fields":["Name","Job"]
}'
```

The resulting document is displayed in Figure 7-1.



```
{"docs":[
{"Name":"Naughton","Job":"Clerk"},
{"Name":"Abrahams","Job":"Clerk"},
{"Name":"Marenghi","Job":"Mgr"},
{"Name":"O'Brien","Job":"Sales"},
{"Name":"Quigley","Job":"Sales"}
]}
```

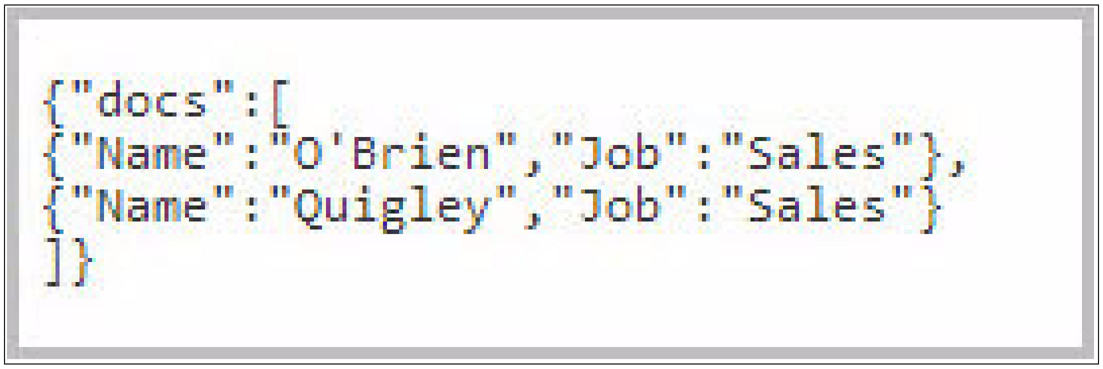
*Figure 7-1 The result of querying the _find endpoint with the field Dept is 38. The fields contained within the document correspond to the Name and Job fields of the complete document.*

We can further refine these results by selecting only for entries corresponding to Dept 38 with Job as "Sales". You can further refine queries by filtering on fields that are not in the index. At least one of the operators in the selector must be associated with a Cloudant Query index, while all other operators can be applied to dynamically filter the results further.

Our selector will look like:

```
{"selector":
 {"Dept": 38,
  "Job":"Sales"},
  "fields":["Name","Job"]
}
```

Figure 7-2 on page 78 shows the result of refined query.



```
{"docs":[
{"Name":"O'Brien","Job":"Sales"},
{"Name":"Quigley","Job":"Sales"}
]}
```

Figure 7-2 The resulting document from the same query made in Figure 7-1 on page 77, selecting only documents where field Job equal to "Sales"

Figure 7-2 shows the resulting document from the same query made in Figure 7-1 on page 77, selecting only documents where field Job equal to "Sales".

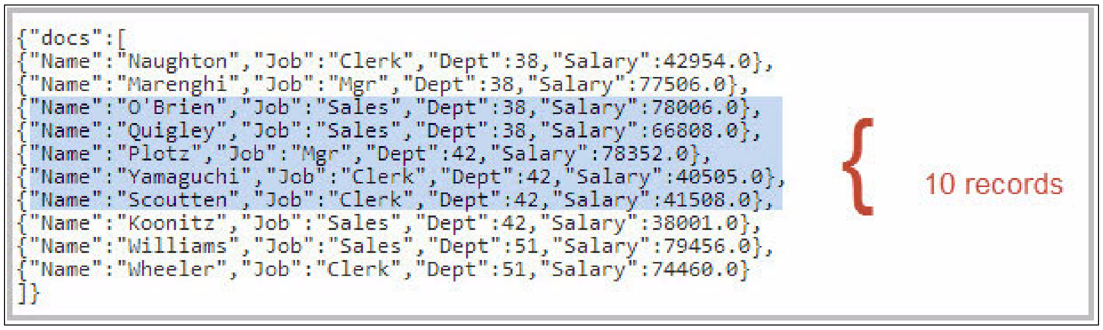
The above examples demonstrate how by creating index you can return list of the field you are interested in for documents that satisfy the selection criteria's.

## 7.3 More Cloudant Query options

You can limit the number of documents returned from a Cloudant Query using the ***limit*** option; additionally, you can skip the first N rows using the aptly-named ***skip*** option.

The following example demonstrates how we can narrow the results of the query using the two options we just described. Let us first return Name, Job, Dept, and Salary for all documents with Dept greater than 20 and Salary less than 80 K. We can do it using the DeptIndex created before.

```
POST https://redbookid.cloudant.com/staff/_find  -d '
{"selector":
 {"Dept":{"$gt":20},
  "Salary":{"$lt":80000}},
  "fields":["Name","Job","Dept","Salary"]
} '
```



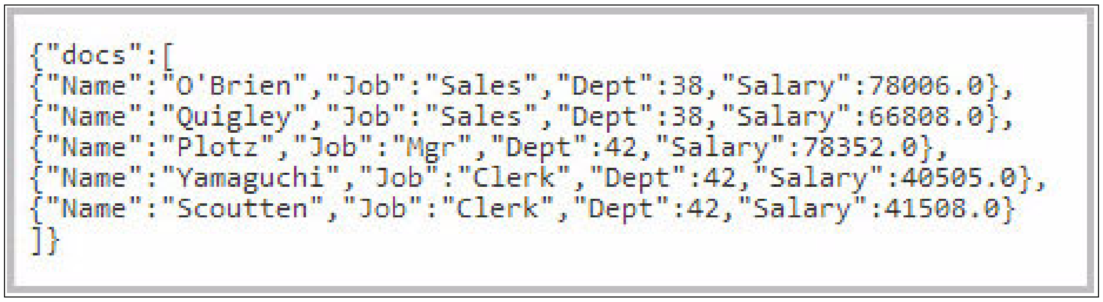
```
{"docs":[
{"Name":"Naughton","Job":"Clerk","Dept":38,"Salary":42954.0},
{"Name":"Marenghi","Job":"Mgr","Dept":38,"Salary":77506.0},
{"Name":"O'Brien","Job":"Sales","Dept":38,"Salary":78006.0},
{"Name":"Quigley","Job":"Sales","Dept":38,"Salary":66808.0},
{"Name":"Plotz","Job":"Mgr","Dept":42,"Salary":78352.0},
{"Name":"Yamaguchi","Job":"Clerk","Dept":42,"Salary":40505.0},
{"Name":"Scoutten","Job":"Clerk","Dept":42,"Salary":41508.0},
{"Name":"Koonitz","Job":"Sales","Dept":42,"Salary":38001.0},
{"Name":"Williams","Job":"Sales","Dept":51,"Salary":79456.0},
{"Name":"Wheeler","Job":"Clerk","Dept":51,"Salary":74460.0}
]}
```

Figure 7-3 The result of querying against the _find endpoint with restrictions on Dept and Salary, returning the Name, Job, Dept, and Salary fields of the matching documents

The query returns 10 documents that comply with our selection criteria. Now we only want to return documents highlighted in blue on Figure 7-3 on page 78. To do that, we can add “limit” and “skip” options to previous query to return only five rows, skipping the first two:

```
{"selector":
 {"Dept":{"$gt":20} ,
  "Salary":{"$lt":80000}},
"fields":["Name","Job","Dept","Salary"],
“limit”:5,
“skip”:2
 }
```

Figure 7-4 shows five “rows” returned from the query.



```
{"docs":[
{"Name":"O'Brien","Job":"Sales","Dept":38,"Salary":78006.0},
{"Name":"Quigley","Job":"Sales","Dept":38,"Salary":66808.0},
{"Name":"Plotz","Job":"Mgr","Dept":42,"Salary":78352.0},
{"Name":"Yamaguchi","Job":"Clerk","Dept":42,"Salary":40505.0},
{"Name":"Scoutten","Job":"Clerk","Dept":42,"Salary":41508.0}
]}
```

*Figure 7-4 The result of querying against the _find endpoint, with the same restrictions on Dept and Salary as in Figure 7-3 on page 78, but with additional parameters for skip and limit as well*

Note in Figure 7-4 that the first two rows are skipped, in comparison with the results illustrated in Figure 7-3 on page 78.

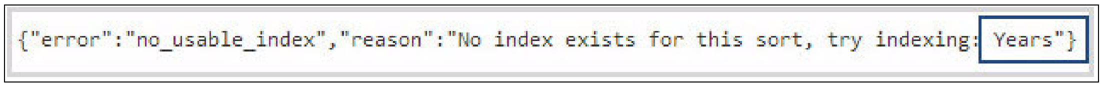
## 7.4 Using complex index

Now let us try to sort the result of the previous query according to years of service. We need to write a selector that returns *Name*, *Job*, *Dept*, and *Salary* for each document with *Dept* greater than 20 and *Salary* greater than 80 K, but we also want the results to be sorted by *Years* in ascending order.

We can send the following request to the *_find* endpoint:

```
POST https://redbookid.cloudant.com/staff/_find  –d ‘
{"selector":
 {"Dept":{"$gt":20} ,
  "Salary":{"$lt":80000}},
 "fields":["Name","Job","Dept","Salary"],
 "sort":[{"Years":"asc"}]
}’
```

We can see the result of the execution below with Figure 7-5.



```
{"error":"no_usable_index","reason":"No index exists for this sort, try indexing: Years"}
```

*Figure 7-5 The error response generated by an attempt to access a non-indexed document field; fields that are not defined as indexable in a database’s design documents cannot be queried in this fashion*

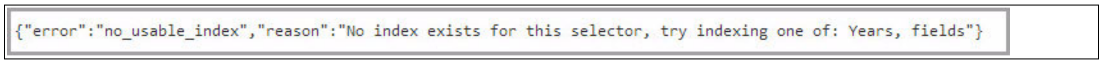
As you can see in Figure 7-5 on page 79, the database server returns an error because we did not yet create an index (as defined by this database's design document) on the Years field of our document. Note that it not only reported the error, but it also gives recommendations for additional indexes that need to be created. Let us follow this recommendation and create index on the Years field:

```
POST https://redbookid.cloudant.com/staff/_index  –d ‘
{"index":
  {"fields":["Years"]},
  "ddoc": “indexes123",
  "name":"YearsIndex"
}’
```

After database server has confirmed that the index on the field *Years* exists, we can resubmit the previous *_find* command:

```
POST https://redbookid.cloudant.com/staff/_find  –d ‘
{"selector":
 {"Dept":{"$gt":20} ,
  "Salary":{"$lt":80000}},
  "fields":["Name","Job","Dept","Salary"],
  "sort":[{"Years":"asc"}]
}’
```

Figure 7-6 shows response from database server.



```
{"error":"no_usable_index","reason":"No index exists for this selector, try indexing one of: Years, fields"}
```

*Figure 7-6   Error response from the database server following a submission to the _find endpoint*

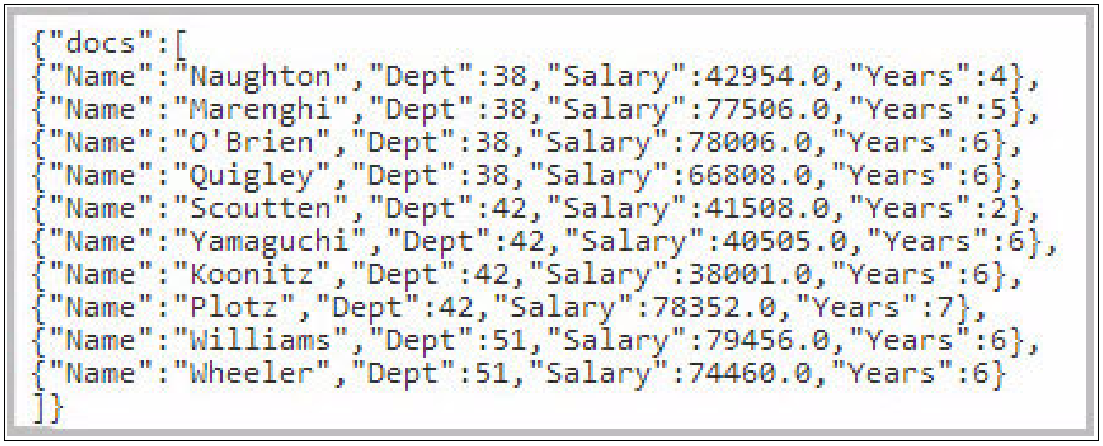
Apparently, we received another error in response to our request. Sorting of the results depends on the structure of your original index because Cloudant Query indexes are really Map Reduce views behind the scenes. In our query, let us call the Years portion of the selector the "primary operator", so as not to conflict with the YearsIndex identifier that Cloudant is trying to use to complete the request. In order for the sorting portion of the request to succeed, the primary operator must be supplied as a parameter to the sort instruction. In addition to that, all fields in the sort must already be a part of the index, meaning Dept must also be a part of the same index and in the same order you want to sort (for example, Years cannot be used to sort the result before first applying a sort on Dept). So, let us create the index that includes both Dept and Years:

```
POST https://redbookid.cloudant.com/staff/_index  –d ‘
{"index":
 {"fields":[{"Dept":"asc"},{"Years":"asc"}]},
  "ddoc": "indexes123",
  "name":"DeptYearsIndex"
}’
```

Now let us sort our results using the selector below. The field Years is included in the result for clarity:

```
{"selector":
 {"Dept":{"$gt":20} ,
  "Salary":{"$lt":80000}},
 "fields":["Name","Dept","Salary","Years"],
 "sort":[{"Dept":"asc"},{"Years":"asc"}]
}
```

The results of the sort request are outputted in Figure 7-7.



```
{"docs":[
{"Name":"Naughton","Dept":38,"Salary":42954.0,"Years":4},
{"Name":"Marenghi","Dept":38,"Salary":77506.0,"Years":5},
{"Name":"O'Brien","Dept":38,"Salary":78006.0,"Years":6},
{"Name":"Quigley","Dept":38,"Salary":66808.0,"Years":6},
{"Name":"Scoutten","Dept":42,"Salary":41508.0,"Years":2},
{"Name":"Yamaguchi","Dept":42,"Salary":40505.0,"Years":6},
{"Name":"Koonitz","Dept":42,"Salary":38001.0,"Years":6},
{"Name":"Plotz","Dept":42,"Salary":78352.0,"Years":7},
{"Name":"Williams","Dept":51,"Salary":79456.0,"Years":6},
{"Name":"Wheeler","Dept":51,"Salary":74460.0,"Years":6}
]}
```

*Figure 7-7 The result of requesting a sort request, identical to the one submitted for Figure 7-6 on page 80, with all fields predefined in the database design document ahead of time*

# Related publications

The publication listed in this section is considered particularly suitable for a more detailed discussion of the topics covered in this paper.

## IBM Redbooks

The following IBM Redbooks publication provides additional information about the topic in this document. Note that this publication might be available in softcopy only.

- Cloudant Solution Guide: *IBM Cloudant: The Do-More NoSQL Data Layer*, TIPS-1187

  `http://www.redbooks.ibm.com/Redbooks.nsf/RedbookAbstracts/tips1187.html`

You can search for, view, download or order this document and other Redbooks, Redpapers, Web Docs, draft and additional materials, at the following website:

**ibm.com**/redbooks

## Help from IBM

IBM Support and downloads

**ibm.com**/support

IBM Global Services

**ibm.com**/services

# IBM Cloudant: Database as a Service Fundamentals

Redpaper™

**Understand the basics of NoSQL document data stores**

**Access and work with the Cloudant API**

**Work programmatically with Cloudant data**

This IBM Redpaper publication shows how the IBM Cloudant fully-managed database as a service (DaaS) enables applications and their developers to be more agile. As a part of its data layer, clients have access to multi-master replication and mobile device synchronization capabilities for occasionally connected devices. Applications can take advantage of Cloudant advance real-time indexing features for ad hoc full text search via Apache Lucene, online analytics via MapReduce, and advance geospatial querying.

Mobile applications can use a durable replication protocol for offline sync and global data distribution, as well as a geo-load balancing capability to ensure cross-data center availability and optimal performance. The Cloudant RESTful web-based application programming interface (API), flexible schema, and capacity to scale massively are what empowers clients to deliver applications to market faster in a cost-effective, DBA-free service model.

REDP-5126-00