Indexing and querying overview

Types of indexes

IBM Cloudant supports several types of indexes. You can perform basic create-read-update-delete operations on documents in a Cloudant database by directly referencing the document ID. Below are definitions of each of the indexes supported in Cloudant database.

- **Primary:** Created out-of-the-box and stored in a b-tree data structure; uses the document ID as the primary key; used to get a list of documents by ID.
- **Cloudant Query:** Uses “mongo-style” querying and provides you with a declarative way to define and query indexes on your database.
- **Secondary:** Also called a view; built via the MapReduce paradigm; used for analytics such as counts, sums, averages, or other mathematical functions.
- **Search:** Built using Apache Lucene search; used for ad hoc queries on one or more fields, searches involving blocks of text, or queries that require additional Lucene syntax like wildcards, fuzzy search, and facets.
- **Geospatial:** Stored in the most efficient way to allow for 4D querying; used for advanced geospatial queries beyond a bounding box or 4D.

© Copyright IBM Corp. 2017
Course materials may not be reproduced in whole or in part without the prior written permission of IBM.

Using the primary index

As mentioned previously, the primary index is created out-of-the-box and uses the document ID as the primary key.

The URL for the Cloudant API is made up of an account name, the database within that account, and the endpoints to manipulate data within that database.

```
http://ablanks.cloudant.com/employee_directory/_all_docs?include_docs=true
```

The previous example references the ablanks account, the employee_directory database, and for all documents in the database, show the document body. You could use this programmatically to populate a web page that shows all employees.

_all_docs is the primary index which displays a list of documents by ID.

The following table lists the query parameters supported by the primary index.

Parameter	Description	Example
include_docs	Includes the document contents when set to true.	Include_docs=true
key	Retrieves a specific document.	key="32b0bb7f30a61deee45b26aa9cd"
startkey and endkey	Retrieves a range of documents; reads from the top beginning at the startkey, and returns results through the endkey sorted by the key.	startkey="32"&endkey="34"
descending	Sort the results in descending order when set to true; reverses the reading direction. When combined with startkey and endkey, it reads from the bottom beginning at the startkey, and returns the results through the endkey sorted by the key.	Descending=true
skip	Skip over the first *n* documents in the results.	skip=10
limit	Limit the results to the first *n* documents in the results.	limit=100

© Copyright IBM Corp. 2017
Course materials may not be reproduced in whole or in part without the prior written permission of IBM.

Using Cloudant Query

Cloudant Query overview

Cloudant Query is the best way to get started with querying Cloudant databases. With Cloudant Query, you define an index by posting JSON text listing the fields you want to index. Indexes are implemented behind the scenes as either a primary index, a secondary index, or a search index. There's no need to write a JavaScript function to create an index as you will learn with other indexes later in this course. It is the logical starting point for developers new to Cloudant and Apache CouchDB. For developers who are familiar with SQL, Cloudant Query is also easy to learn and use.

The following JSON object defines a new index.

```
{     "index" : {"fields" : ["foo"]},
      "name" : "foo-index",
      "type" : "json"
}
```

The following JSON object is an example of a request body to find documents using that index.

```
{     "selector" : {"bar" : {"$gt" : 1000000}},
      "fields" : ["_id", "_rev", "foo", "bar"],
      "sort" : [{"bar" : "asc"}],
      "limit" : 10,
      "skip" : 0
}
```

Index types

There are two types of indexes: json and text. There are advantages to using each type of Cloudant Query index.

json: Leverages the Map phase of MapReduce with secondary indexes, and will build and query faster than a text type for a fixed key. But this type of index restricts the operators you can use as a basis for query in a selector and might end up doing more work in memory for complex queries. Secondary indexes are covered in more detail later.

type: Leverages an Apache Lucene search index under the covers. It permits indexing all fields in documents automatically with a single simple command. It also provides more flexibility to perform adhoc queries and sort across multiple keys, and permits you to use any operator as a basis for query in a selector. Search indexes are covered in more detail later.

From the dashboard, you can automatically index all fields using this syntax.
POST to _index with:

```
{ "index": {}, "type": "text" }
```

© Copyright IBM Corp. 2017
Course materials may not be reproduced in whole or in part without the prior written permission of IBM.

Operator syntax

Selector syntax can be used to express conditional logic using specially named fields. If you are familiar with MongoDB querying then this will seem similar.

A simple selector might match any documents with the name “Paul”:

```
{“name” : “Paul”}
```

An extended selector might match documents with the name “Paul” and location “Boston”:

```
{“name” : “Paul”, “location” : “Boston”}
```

The period (.) character denotes subfields. These two expressions are equivalent.

```
{“location” : {“city” : “Omaha”}}  ==    {“location.city” : “Omaha”}
```

The dollar sign ($) character denotes operators. For example, this expression matches all documents with age greater than 20.

```
{“age” : {“$gt” : 20}}
```

Selector requirements

The “selector” requires at least one of the following relational operators:

- Equal to (“$eq”)
- Greater than (“$gt”)
- Greater than or equal to (“$gte”)
- Less than (“$lt”)
- Less than or equal to (“$lte”)

Cloudant assumes EQUALITY when there is no operator present. For example, the following are equivalent:

```
{“foo”: “bar”}      ==      {“foo” : {“$eq” : “bar”}}
```

The implied equality applies to nested values as well. The following are equivalent:

```
{“foo” : {“bar” : “baz”}}  ==   {“foo” : {“$eq” : {“bar” : “baz”}}}
```

Cloudant assumes AND when there are multiple selectors. For example, the following are equivalent:

```
{“foo” : “bar”, “baz” : true} == {“$and” :[{“foo” : “bar”},{“baz” : true}]}
```

Note: If the user submits a query that does not have a suitable index (for example, if no index exists for the submitted selector), then it is up to the developer to decide at the application layer how such errors are handled. Cloudant does not automatically re-index the whole data set.

© Copyright IBM Corp. 2017
Course materials may not be reproduced in whole or in part without the prior written permission of IBM.

Operator types

There are two core types of operators in the selector syntax:

- Combination operators: used to combine conditions, or to create combinations of conditions, into one selector.
  - $and: Matches if all selectors in the array match
  - $or: Matches if any selectors in the array match
  - $not: Matches if the given selector does not match
  - $nor: Matches if none of the selectors (multiple) match
  - $all: Matches an array value if it contains all element of argument array
  - $elemMatch: Returns first element (if any) matching value of argument
- Condition operators: check for a specific condition to be true; they are specified on a per-field basis, and apply to the value indexed for that field.
  - $lt: Less than
  - $lte: Less than or equal to
  - $eq: Equal to
  - $ne: Not equal to
  - $gt: Greater than
  - $gte: Greater than or equal to
  - $exists: Boolean (exists or it does not)
  - $type: Check document field's type
  - $in: Field must exist in the provided array of values
  - $nin: Field must not exist in the provided array of values
  - $size: Length of array field must match this value
  - $mod: [Divisor, Remainder]. Returns true when the field equals the remainder after being divided by the divisor.
  - $regex: Matches provided regular expression

© Copyright IBM Corp. 2017

Course materials may not be reproduced in whole or in part without the prior written permission of IBM.

Sort options

Sort is an array of field name and direction pairs. This is the standard format for the sort syntax:

```
"sort" : [ { "fieldName1": dir1 }, { "fieldName2": dir2 }, … ]
```

fieldName can be any field and dir can be “asc” or “desc” and indexes must be created ahead of time for sorting based on fields.

The following example selects Robert De Niro movies and sorts in ascending order of the person’s name with a secondary sort on the year the movie was released.

```
{
   "selector": {"Person_name": "Robert De Niro"},
   "sort" : [{"Person_name": "asc"}, {"Movie_year": "asc"}]
}
```

For the “json” type, one of the sorting fields must be included as part of the “selector” and all sorting fields must be indexed.

For the “text” type, you need to append **:number** or **:string** to the sort field as in the following example.

```
{
   "selector": {"Person_name": "Robert De Niro"},
   "sort" : [{"Movie_name:string": "asc"}, {"Movie_runtime:number": "asc"}]
}
```

© Copyright IBM Corp. 2017
Course materials may not be reproduced in whole or in part without the prior written permission of IBM.

Using the secondary index

MapReduce

As mentioned previously, the Cloudant Query json type index creates a secondary index. The secondary index, also known as a view, uses the MapReduce paradigm and provides a way for you to query data in more ways than just a simple primary key lookup. MapReduce is a two-phase paradigm for crunching large data sets in a distributed system. Review the video "Cloudant: MapReduce explained" (https://youtu.be/0iGUCyMIEqk) which uses a simple example.

The map phase

During the map phase, Cloudant scans the JSON documents in the database, emits a list of keys and values, and builds an index sorted by key. You can use IF statements to restrict documents and validate that certain fields exist.

The reduce phase

The reduce phase is optional unless you want to do analytics on the values from the map. If you are only doing lookups that are based on a key or range of keys, you can omit the reduce phase. The reduce phase aggregates the values that are emitted in the map, and includes _count, _sum, and _stats built-in.

© Copyright IBM Corp. 2017
Course materials may not be reproduced in whole or in part without the prior written permission of IBM.

Secondary index examples

Secondary indexes are JavaScript functions that are stored in design documents. The following table provides some examples of map and reduce functions.

Map	Reduce	Results

function(doc) {
if(doc.type === 'entry') {
emit(doc.userid, null);
}
}
	None	Documents sorted by userid

function(doc) {
if(doc.type === 'entry') {
emit(doc.userid, doc.timetaken, null);
}
}
	None	Documents sorted first on the userid, then on timetaken

function(doc) {
if(doc.type === 'entry') {
emit(doc.userid, null);
}
}
	_count	A count of the number of documents returned by the map

function(doc) {
if(doc.salesperson === 'entry') {
emit(doc.salesperson, doc.amount);
}
}
	_sum	The sum of the sales amount for each salesperson

Parameters for secondary indexes

There are a few additional parameters that are specific to a Secondary Index.

- **stale**: Speeds up the response as it does not wait for the index build process to complete
- **reduce**: Turns reduce on or off. By default, reduce=true. When you add reduce=false, the result of the Map is shown with the emit having the key of the user ID and value being null. The id of the document is also shown in this view.
- **group**: Groups results by unique key. The value is reduced via the _count function to show the total number by unique key. It's only valid when reduce is turned on. By default, group=false.

© Copyright IBM Corp. 2017
Course materials may not be reproduced in whole or in part without the prior written permission of IBM.

Complex keys

A secondary Index is automatically sorted by the key. However, the key does not have to be just a single field and can be more complex, such as an array or an object.

Emitting an array of fields as the key in a secondary Index is also referred to as using a complex key. Complex keys are a useful design pattern when you must aggregate results across a fixed hierarchy, and you want to be able to toggle the granularity of results.

For example, an HR department might want to calculate the vacation days that are taken for all company employees. The HR manager would like to get the vacation days broken down by employee, group, department, and the whole company. In this case, the key would be [doc.company, doc.department, doc.group, doc.employee].

When querying a complex key, you can toggle which level in the array you want the aggregate to be grouped by. To toggle the level of granularity, use the query parameter group_level and specify a number from 1 to the number of elements in the array. So, in the above example, group_level=1 would group by company, group_level=2 would group by department, and so on. Using group=true will group by each unique key and is equivalent to using group_level=4 in the example.

© Copyright IBM Corp. 2017
Course materials may not be reproduced in whole or in part without the prior written permission of IBM.

Using the search index

Search index overview

The Cloudant Query text type index creates a search index. The search index is built using Apache Lucene search, and is useful to perform ad-hoc queries, find documents based on their contents, or work with groups, facets, or geographies.

Like secondary indexes, search indexes are JavaScript functions stored in design documents. The following example indexes nine fields: email address, name, each part of the address field, an array of products and languages, hire date, and their geolocation.

```
function(doc) {
    index("email", doc._id);
    if (doc.name) {
        index("name", doc.name, {"store": true});
    }
    if (doc.address.city) {
        index("city", doc.address.city, {"store": true});
    }
    if (doc.address.state) {
        index("state", doc.address.state, {"store": true});
    }
    if (doc.address.country) {
        index("country", doc.address.country, {"store": true});
    }
    if (doc.products) {
        for (var i in doc.products) {
            index("product", doc.products[i], {"store": true, "facet":
true, "index": true});
        }
    }
    if (doc.languages) {
        for (var j in doc.languages) {
            index("language", doc.languages[j], {"store": true, "facet":
true});
        }
    }
    if (doc.HireDate) {
        index("hiredate", doc.HireDate, {"store": true, "facet": true});
    }
    if (doc.geolocation.coordinates) {
        index("long", doc.geolocation.coordinates[0]);
        index("lat", doc.geolocation.coordinates[1]);
    }
}
```

© Copyright IBM Corp. 2017
Course materials may not be reproduced in whole or in part without the prior written permission of IBM.

Index function parameters

The index function takes three parameters.

- The **first** is the field name. If you use “default" for this parameter, then this field will be queried if no field name is specified in the query syntax. In the example below, the first parameter is "product".
- The **second** parameter is the data to index. In the example below, the second parameter is the array doc.products[i].
- The **third** parameter, which is optional, can contain the fields store, index, and facet. In the example below, the third parameter is {"store": true, "facet": true, "index": true}.
    - When the store field is set to true, then the value will be returned in search results, otherwise, it will only be indexed.
    - When facet is set to true, then faceting will be turned on for the data being indexed.
    - When index is set to false, the field will not be searchable in the index, but can still be retrieved from the index if store is set to true.

```
index("product", doc.products[i], {"store": true, "facet": true, "index": true})
```

You can put search indexes and secondary indexes in the same design document, however, it is a best practice to separate search and secondary indexes into different design documents.

Analyzers

There are six types of analyzers which define how to recognize terms within text.

- **Standard**: The default analyzer. It implements the Word Break rules from the Unicode Text Segmentation algorithm.
- **Classic**: The standard Lucene analyzer uses a defined set of stop words for tokenization.
- **Email**: Similar to the standard analyzer, but matches 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.

© Copyright IBM Corp. 2017

Course materials may not be reproduced in whole or in part without the prior written permission of IBM.

Each analyzer has a specific way of tokenizing the fields. For example, the standard analyzer tokenizes a date stored as “2014-11-03” in parts as follows:

```
{
    "tokens": [
        "2014",
        "11",
        "03",
    ]
}
```

Which would yield unexpected results when you try to query that field. Whereas the keyword analyzer tokenizes the same date as a whole as follows:

```
{
    "tokens": [
        "2014-11-03"
    ]
}
```

Query parameters

In addition to the **stale**, **include_docs**, and **limit** parameters that you have already used, the following query parameters are available with a search index.

sort: Specifies the sort order of the results. sort=["name<string>"]

include_fields: Specifies a JSON array of field names to include in search results. Any fields included must have been indexed with the "store":true option.

highlight_fields: Specifies which fields should be highlighted.

bookmark: Specifies the bookmark that was received from a previous search. This allows you to page through the results. If there are no more results after the bookmark, you get a response with an empty rows array and the same bookmark. That way you can determine that you have reached the end of the result list.

Faceted search parameters

The following query parameters are specific to a search index with faceting turned on.

ranges: Defines ranges for faceted, numeric search fields. For example, name:[b TO d]

drilldown: Can be used several times. Each use defines a pair of a field name and a value. The search only matches documents that have the given value in the field name. It differs from using "fieldname:value" in the q parameter only in that the values are not analyzed.

counts: Defines an array of names of string fields, for which counts should be produced. The response contains counts for each unique value of this field name among the documents matching the search query.

© Copyright IBM Corp. 2017

Course materials may not be reproduced in whole or in part without the prior written permission of IBM.

Group search parameters

The following parameters are used to group documents in the search results.

group_field: Specifies the field by which to group search matches.

group_limit: Specifies the maximum group count. This field can only be used if group_field is specified.

group_sort: Defines the order of the groups in a search using group_field. The default sort order is relevance.

© Copyright IBM Corp. 2017
Course materials may not be reproduced in whole or in part without the prior written permission of IBM.

Using a geospatial index

Cloudant Geospatial overview

Cloudant Geospatial combines the advanced geospatial queries of Geographic Information Systems with the flexibility and scalability of Cloudant's database-as-a-service. Web and mobile developers can enhance their applications with geospatial operations that go beyond simple bounding boxes. Cloudant Geospatial integrates with existing GIS applications so they can scale for data size, concurrent users, and multiple locations.

Here is the high level process for using Geospatial in Cloudant.

1. Include a GeoJSON geometry object in your JSON documents.

```
"geolocation": {
          "coordinates": [
               -73.985248,
               40.759093
           ],
           "type": "Point"
 }
```

2. Index the geometry object using "st_index"

```
{
  "_id": "_design/geodd",
  "views": {},
  "language": "javascript",
  "st_indexes": {
     "geoidx": {
        "index": "function(doc){if (doc.geometry &&
doc.geometry.coordinates)
   {st_index(doc.geometry);}}"
     }
  }
}
```

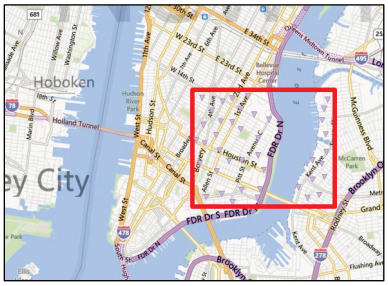
3. Query with various geometries and geometric relationships.

```
relation=contains&g=POLYGON ((-71.0537124 42.3681995 0,-71.054399
42.3675178 0,-71.0522962 42.3667409 0,-71.051631 42.3659324 0,-71.051631
42.3621431 0,-71.0502148 42.3618577 0,-71.0505152 42.3660275 0,-
71.0511589 42.3670263 0,-71.0537124 42.3681995 0))
```

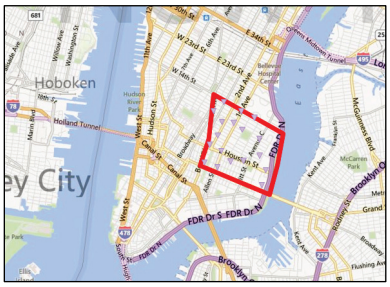
© Copyright IBM Corp. 2017
Course materials may not be reproduced in whole or in part without the prior written permission of IBM.

Defined regions

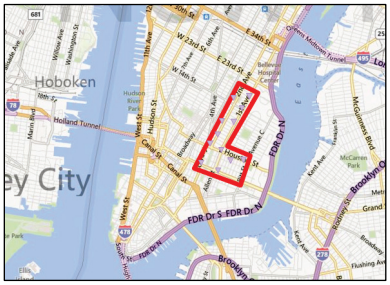
Bounding box queries return local results, but may not be all that compelling because a simple rectangle may cross impassable barriers such as highways or rivers.



Multipoint polygons deliver better results for users. Here the results are limited to a Lower East Side neighborhood in Manhattan.



Users can define their own polygons. Here a user has drawn a polygon around their commute to work so they can search for dry cleaners within the Lower East Side of Manhattan that are along the streets she most frequently uses.



GeoJSON formatted data

Since GeoJSON is a standard for storing geographic data in JSON format, it is a best practice to adhere to this format when storing geo-coordinates in Cloudant.

GeoJSON format consists of three parts:

- The geometry section contains two fields: the coordinates and the type, such as Point, LineString, or Polygon.
- The properties section contains other optional data.
- The type section must be set to “Feature”.

© Copyright IBM Corp. 2017
Course materials may not be reproduced in whole or in part without the prior written permission of IBM.

The coordinates must adhere to the GeoJSON standard, but the properties object and type field are not required. The following is an example of GeoJSON formatted data.

```
{"_id": "79f14b64c57461584b152123e38a58ca",
   "_rev": "1-d9518df5c255e4bfa06516802faa2a16",
   "geometry": {
        "coordinates": [-71.05987446,
             42.28339928
        ],
        "type": "Point"
   },
   "properties": {
        "compnos": "142035012",
        "domestic": true,
        "fromdate": 1412208240000,
        "main_crimecode": "Argue",
        "naturecode": "DISTRB",
        "reptdistrict": "C11",
        "shooting": false,
        "source": "Boston"
   },
   "type": "Feature"}
```

Geospatial indexes

Just like with secondary and search indexes, you create a geospatial Index by saving a JavaScript function in a _design document in the database. It's a best practice to use a separate design document for geospatial indexes.

In the following geodd design document example, there is an "st_indexes" object, which includes a geoidx index along with the "index" function which checks for a valid geometry object, and then runs the st_index function on doc.geometry.

```
{
  "_id": "_design/geodd",
  "views": {},
  "language": "javascript",
  "st_indexes": {
    "geoidx": {
      "index": "function(doc){if (doc.geometry && doc.geometry.coordinates)
{st_index(doc.geometry);}}"
    }
  }
}
```

© Copyright IBM Corp. 2017

Course materials may not be reproduced in whole or in part without the prior written permission of IBM.

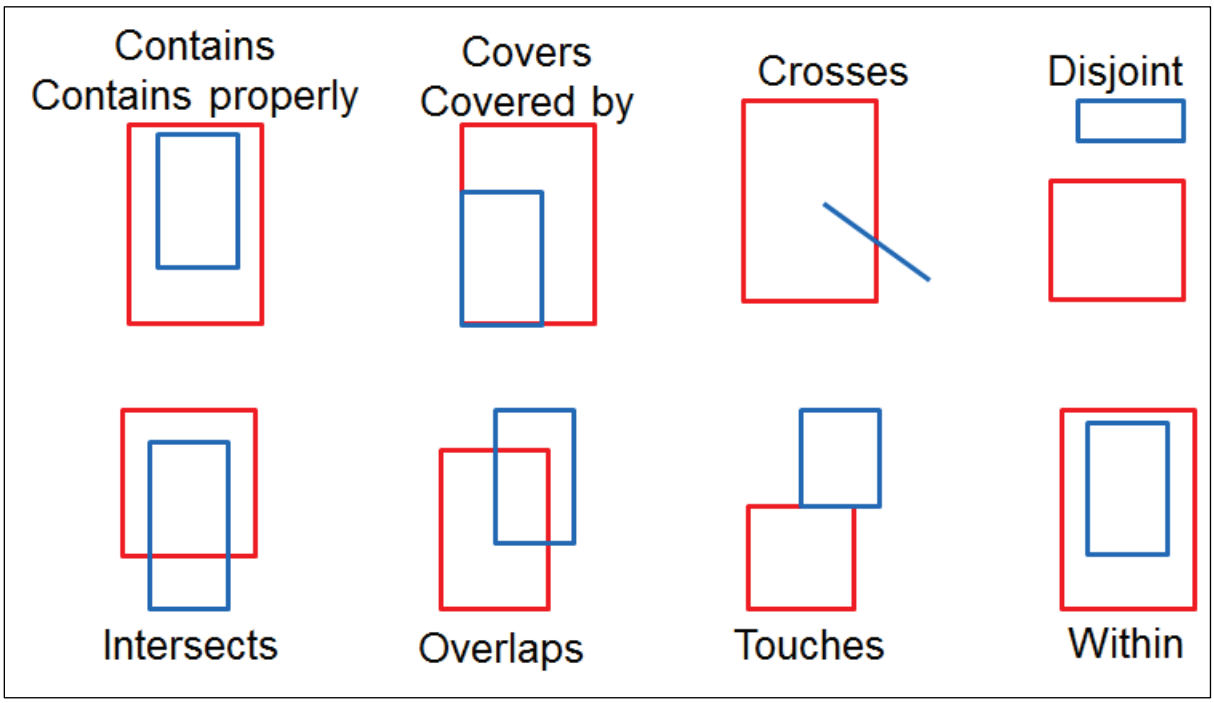
Types of geospatial queries

There are three types of queries:

- radius: specifies latitude, longitude, and radius which creates a circle centered at that latitude and longitude of the given radius in meters, and compares each geometry in the index to that circle using the given relation.
- ellipse: specifies a latitude, a longitude, and two radii: rangex and rangey measured in meters
- g: specifies a geometry value such as polygon, linestring, point, and so on. It also requires a relation parameter such as "contains".

Geospatial relations

In Cloudant, geospatial relations are specified by the relation query parameter. Cloudant Geo supports the following standard geospatial relations: contains, contains properly, covers, covered by, crosses, disjoint, intersects, overlaps, touches, and within. For more information on spatial relations, refer to https://en.wikipedia.org/wiki/Spatial_relation.



Mapbox integration

The Cloudant dashboard includes integrated map visualizations via Mapbox.js. The activities in this course show this integration. For more information, refer to Geospatial Development with Cloudant & Mapbox at https://developer.ibm.com/clouddataservices/2016/04/11/geojson-database-cloudant-mapbox/.

© Copyright IBM Corp. 2017
Course materials may not be reproduced in whole or in part without the prior written permission of IBM.