DHBW Informatik Semester 4
Wahlpflicht Cloud Computing
Chapter 5 NoSQL Database, Object Storage

Juergen Schneider

Apache CouchDB

CouchDB
relax

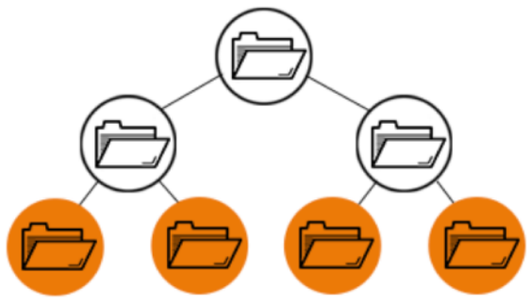
IBM Cloud
Object Storage

Amazon S3

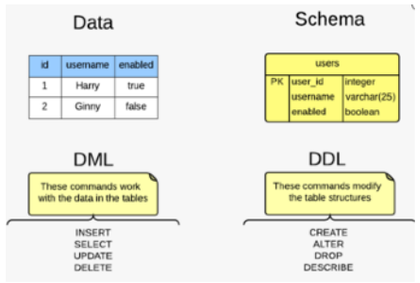
Agenda

- Cloud Object Storage, Amazon S3 Storage
  - Characteristics, Access, Use Cases
- Document Centric Storage
  - NoSQL versus SQL
  - CouchDB and IBM Cloudant
  - Node.JS package usage

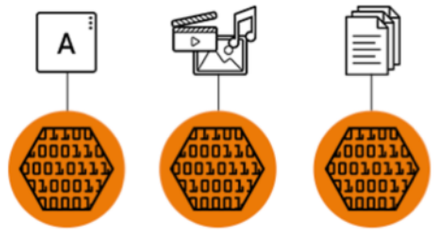
Many Storage Options and Architectures



Files in folders (hierarchical)



Tables, Rows, Columns



Single data objects

e.g Data (unstructured in JSON)

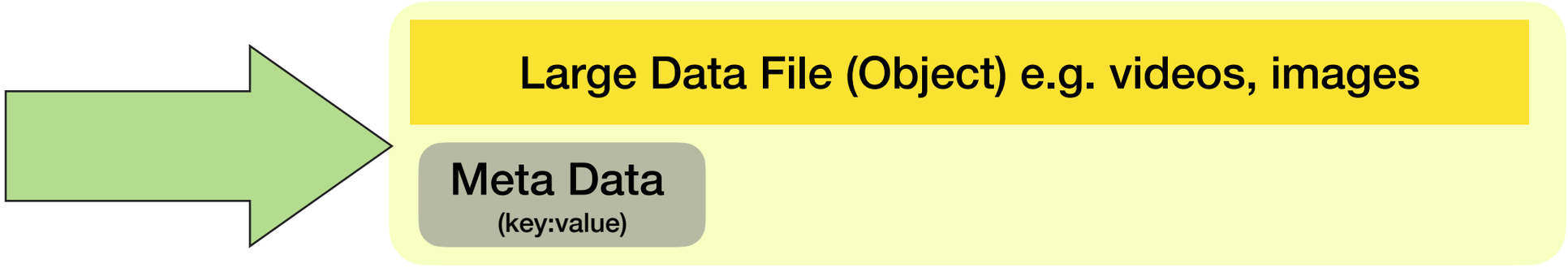
OS APIs	SQL APIs	HTTP APIs	HTTP APIs

Capability to persist data on a medium (typically in blocks)

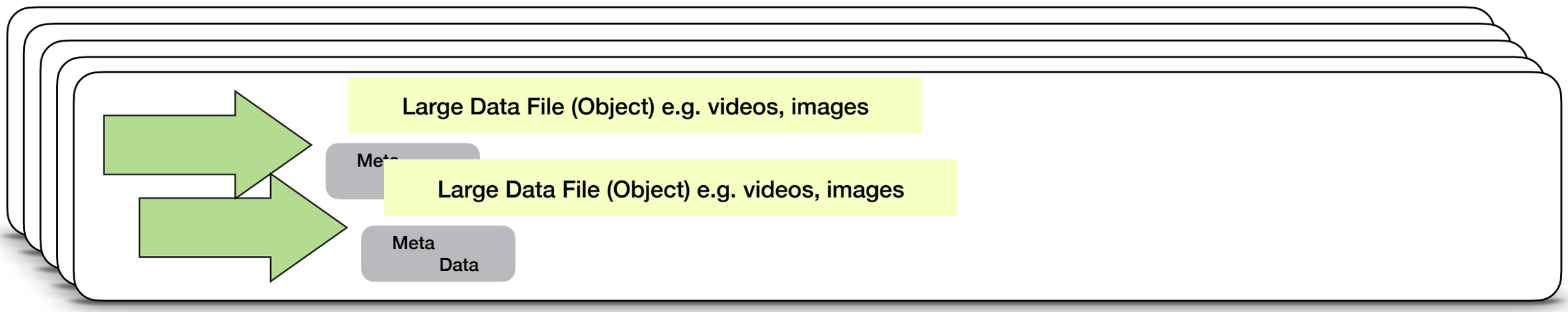
References : https://www.redhat.com/en/topics/data-storage/file-block-object-storage ,
https://launchschool.com/books/sql_first_edition/read/review

Object Storage Principles

Object



Buckets = Container for Objects (Namespace, Storage Classes)



Storage Classes (per bucket, profile for different usage and price)

Access: frequently, unknown (measure yourself), infrequently, archiving, archiving for years…
Endpoint: single AZ, single Region, Multiregion, (Outage Scope)
Link: public, private, VPN

Object Storage Characteristics

- **Objects**
  - Objects consist of object data and metadata. The data portion is opaque to Amazon S3. The metadata is a set of name-value pairs that describe the object.
- **Keys**
  - A key is the unique identifier for an object within a bucket. Every object in a bucket has exactly one key. The combination of a bucket, key, and version ID uniquely identify each object.
  - A Key can be organized as folder structure like /path-x/path-y/file to simulate a folder structure
- **Versions (AWS) (Bucket Level)**
  - **Default** : no versioning each {HTTP Put key, data} will replace an existing object with the same key
  - **Versioning:** {HTTP Put key, data}. will generate a new version of that object. {HTTP Delete key, versionid} would permanently delete this version. To retrieve all versions of an Object you need a {HTTP Get/?versions&prefix=key
- **Buckets**
  - are used to enable an object key Namespace.
  - refer to the account responsible for storage and data transfer charges.
  - are used as entity to protect in access control.
  - Enable / disable versioning of objects
  - serve as the unit of aggregation for usage reporting
- ‘Special cases’
  - SQL Interface to query content of object (where it make sense, with JDBC driver)
  - Integration with DNS and CDN Services to easily build a static Webpage

References : https://docs.aws.amazon.com/AmazonS3/latest/userguide/Welcome.html#features

Object Storage Characteristics

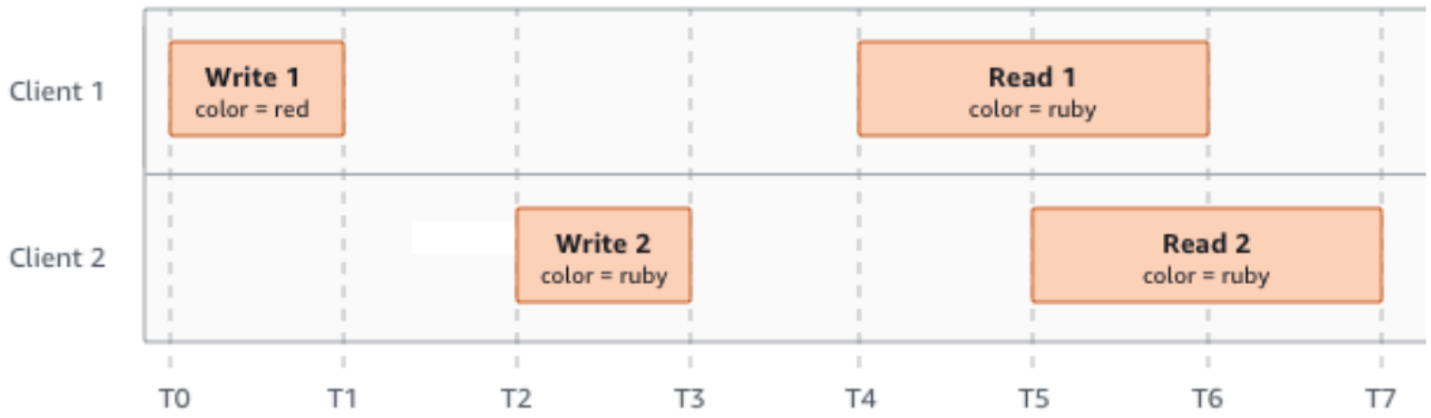
- **Data Consistency**
  - **Full consistency** : data read and write in a time sequence are consistent. For example a later read will always get the write before.
  - Problem in **distributed data system** such operation are running on different nodes and not serialized, therefore time sequence is not always guaranteed. (e.g. a later read will not get the latest ((full completed) update) **(Eventual Consistency)**

  - AWS S3 Object Updates are fully consistent ! (Update visibility is atomic (serialized), that is concurrent reads see either the old or the new update, but never the old update when the new has been completed. At the time of the update commit, **all reads to replicated versions are locked.)**
  - AWS Bucket Configuration Updates are eventual consistent (takes time to replicate, no serialization)

References : https://docs.aws.amazon.com/AmazonS3/latest/userguide/Welcome.html#features

Object Storage Characteristics

- **Full Data Consistency Examples**



Full Data Consistency ensure that each read **arrived** after T3 will get ruby.

Eventual Consistency could mean that a read after T3 still gets red

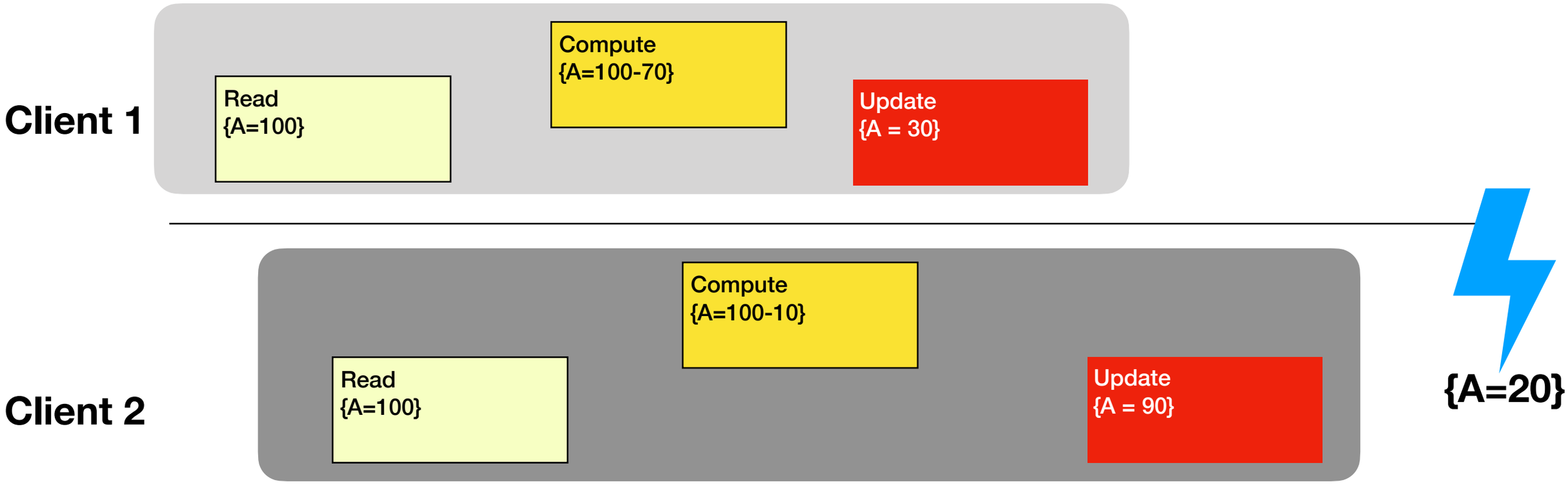
While Write 2 is not complete, Read 1 which arrived at T3 either gets ruby or garnet. This depends on when Write 2 makes garnet visible (commit the change between T3 and T4). It is fully consistent, since W2 has not completed before Read 1 started and the Lock is close to the commit.

Two concurrent Writes. The latest commit always wins, however that could also be the first started.

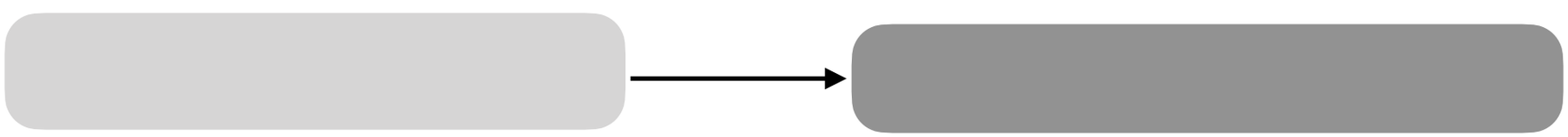
References : https://docs.aws.amazon.com/AmazonS3/latest/userguide/Welcome.html#features

Logical (In)Consistency

Application Level Dependency between read and write (logical unit of work)



Application View : lock A until update or detect the update in between (e.g. changed revision)



- This App scenario is not at all a typically S3 use case. S3 is more one place to upload {HTTP Put} and many places to download {HTTP Get}
- However this App scenario may apply to document centric DBs such as Couch DB

Object Storage Programming

AWS S3 follows the REST API programming Pattern

- {HTTP Put} to create a resource (e.g. to create a new object)
- {HTTP Delete} to delete a resource (e.g. a written object)
- {HTTP Get/Head to read a resource (e.g. an object, or meta data of an object)
- Request Header are used to carry API Keys and other (meta) data
- HTTP return codes are used to communicate success or failures

Examples. cURL

- Upload and Object

```
curl        -X "PUT" "https://(endpoint)/(bucket-name)/(object-key)" \
            -H "Authorization: bearer (token)" \
            -H "Content-Type: (content-type)" \
            -d "(object-content)"
```

Examples. cURL

- Download Object

```
curl          "https://(endpoint)/(bucket-name)/(object-key)"
            -H "Authorization: bearer (token)"
```

References : https://docs.aws.amazon.com/AmazonS3/latest/API/Type_API_Reference.html

Object Storage Characteristics

- **Service Levels (SLO/SLA) e.g AWS S3** (https://aws.amazon.com/s3/sla/)

For all requests not otherwise specified below:

Monthly Uptime Percentage	Service Credit Percentage
Less than 99.9% but greater than or equal to 99.0%	10%
Less than 99.0% but greater than or equal to 95.0%	25%
Less than 95.0%	100%

For requests to S3 Intelligent-Tiering, S3 Standard-Infrequent Access, and S3 One Zone-Infrequent Access:

Monthly Uptime Percentage	Service Credit Percentage
Less than 99.0% but greater than or equal to 98.0%	10%
Less than 98.0% but greater than or equal to 95.0%	25%
Less than 95.0%	100%

Node.js Package (IBM Example)

```
var ibm = require('ibm-cos-sdk');
var util = require('util');

var config = {
    endpoint: '<endpoint>',
    apiKeyId: '<api-key>',
    serviceInstanceId: '<resource-instance-id>',
};

var cos = new ibm.S3(config);

function doCreateBucket() {
    console.log('Creating bucket');
    return cos.createBucket({
        Bucket: 'my-bucket',
        CreateBucketConfiguration: {
          LocationConstraint: 'us-standard'
        },
    }).promise();
}

function doCreateObject() {
    console.log('Creating object');
    return cos.putObject({
        Bucket: 'my-bucket',
        Key: 'foo',
        Body: 'bar'
    }).promise();
}

function doDeleteObject() {
    console.log('Deleting object');
    return cos.deleteObject({
        Bucket: 'my-bucket',
        Key: 'foo'
    }).promise();
}

function doDeleteBucket() {
    console.log('Deleting bucket');
    return cos.deleteBucket({
        Bucket: 'my-bucket'
    }).promise();
}

doCreateBucket()
    .then(doCreateObject)
    .then(doDeleteObject)
    .then(doDeleteBucket)
    .then(function() {
        console.log('Finished!');
    })
    .catch(function(err) {
        console.error('An error occurred:');
        console.error(util.inspect(err));
    });
```

URL addressing and API credentials

S3 JavaScript Object

createBucket method

The more or less useful examples are executed here as sequences of promise objects.

https://cloud.ibm.com/apidocs/cos/cos-compatibility

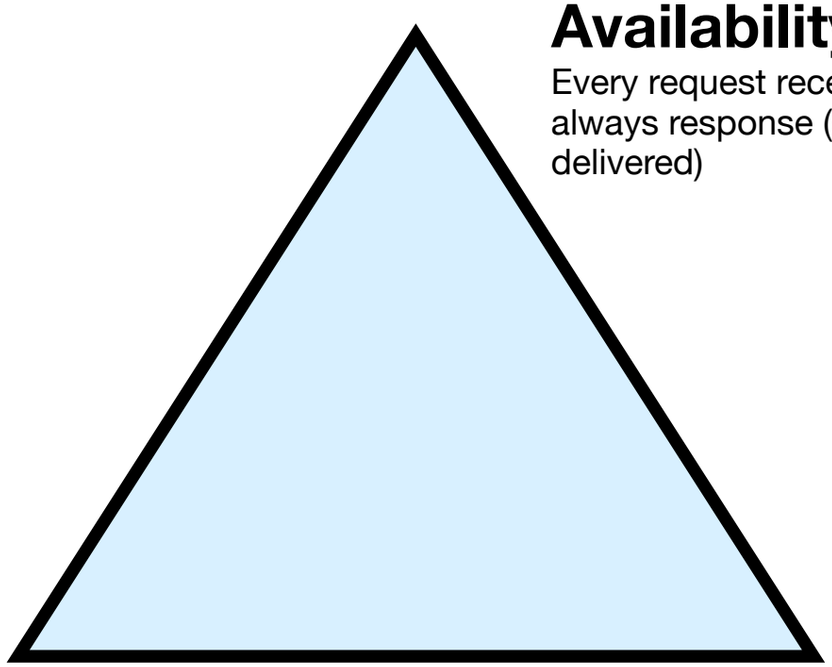
Document Centric Data Services

NoSQL and SQL DBs

	No SQL	SQL
Model	(un)structured data (JSON) (key value, graph, wide column, document centric) schema-less	Tables (Rows/Columns) and Joins (fixed schema)
Standard Query Language	No, most support a HTTP REST Interface, filters and views	SQL (and the variants)
Focus	Availability and Scalability (horizontally scaleable)	Full Data Integrity (only vertically scaleable, or sharding)
Data Consistency Transaction Context	Eventual Consistency (distributed)	central instance fully transaction context
Maturity	Still growing, focus at the big Internet Companies	Developed over decades, in almost every IT Enterprise
Examples	MongoDB, DynamoDB, CouchDB,Redis	DB2, MySQL, Oracle,PostgreSQL

CAP Theorem

- Basic rule for distributed Data Services (Brewers Theorem)
  - Only two can be supported to the full (reasonable) extent



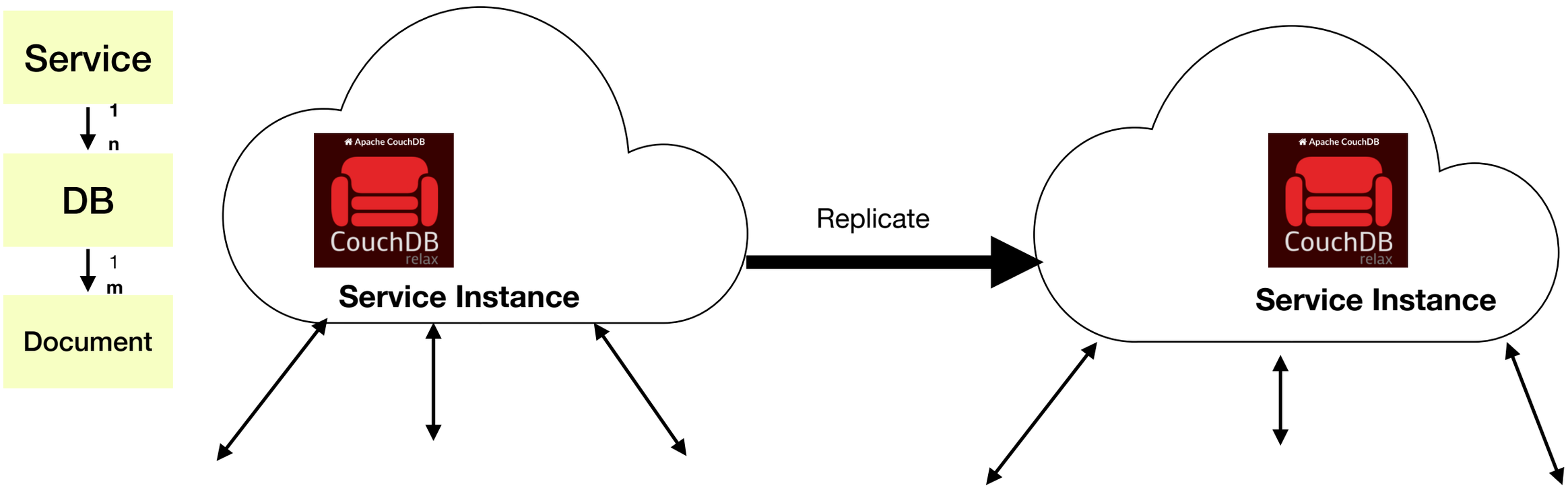
Availability
Every request receives a response (no outage). The data service will always response (data could be ‘inconsistent’ but a response will be delivered)

Consistency
Every read will get the latest write (update) or an error

Partitioning
The system continues to operate despite an arbitrary number of messages being dropped (or delayed) by the network between nodes

CouchDB guarantees **eventual consistency** to be able to provide both availability and partition tolerance. A later read will eventually (not always) get the latest write.

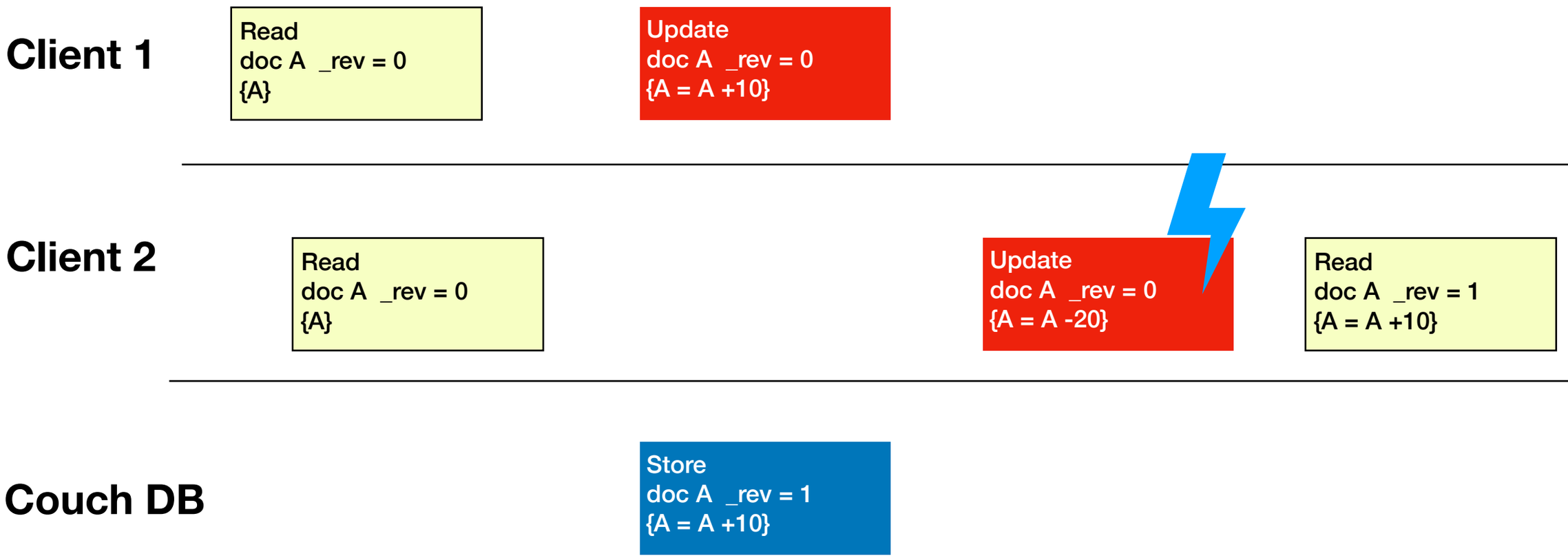
Distributed Data Service Topology



- Only one node will run all updates (other nodes can be selected in case of an outage, but there will always be one)
- All data is stored (e.g.) in three different places to allow parallel (high scale) reads. (Since this takes time or there is outage in the replication, a later read may not catch the latest write)
- Sharding (document updates are distributed by the system) or partitioning (document updates, driven by a document key) is possible. Purpose is to balance the load among all servers
- Everything is redundant, but since we have a distributed system, we have eventual consistency ..

Couch DB Main Features

- ACID Semantics (**A**tomicity, **C**onsistency, **I**solation und **D**urability)
  - CouchDB provides **some** ACID semantics. It does this by implementing a form of **Multi-Version Concurrency Control**, meaning that CouchDB can handle a high volume of concurrent readers and less writers without conflict. (**Each write will generate a new version (_rev)**, while reads can work on the older versions
  - Updates (per document) are serialized for one Couch DB node (so always consistent there).
  - It does not support when a couple of updates should be committed (made available) in one transaction (do it for all or don't do it for all). This is up to the App to recover from here.
  - mySQL gives the control to the App already with SELECT ... FOR UPDATE to lock potential reads while updating a selected row

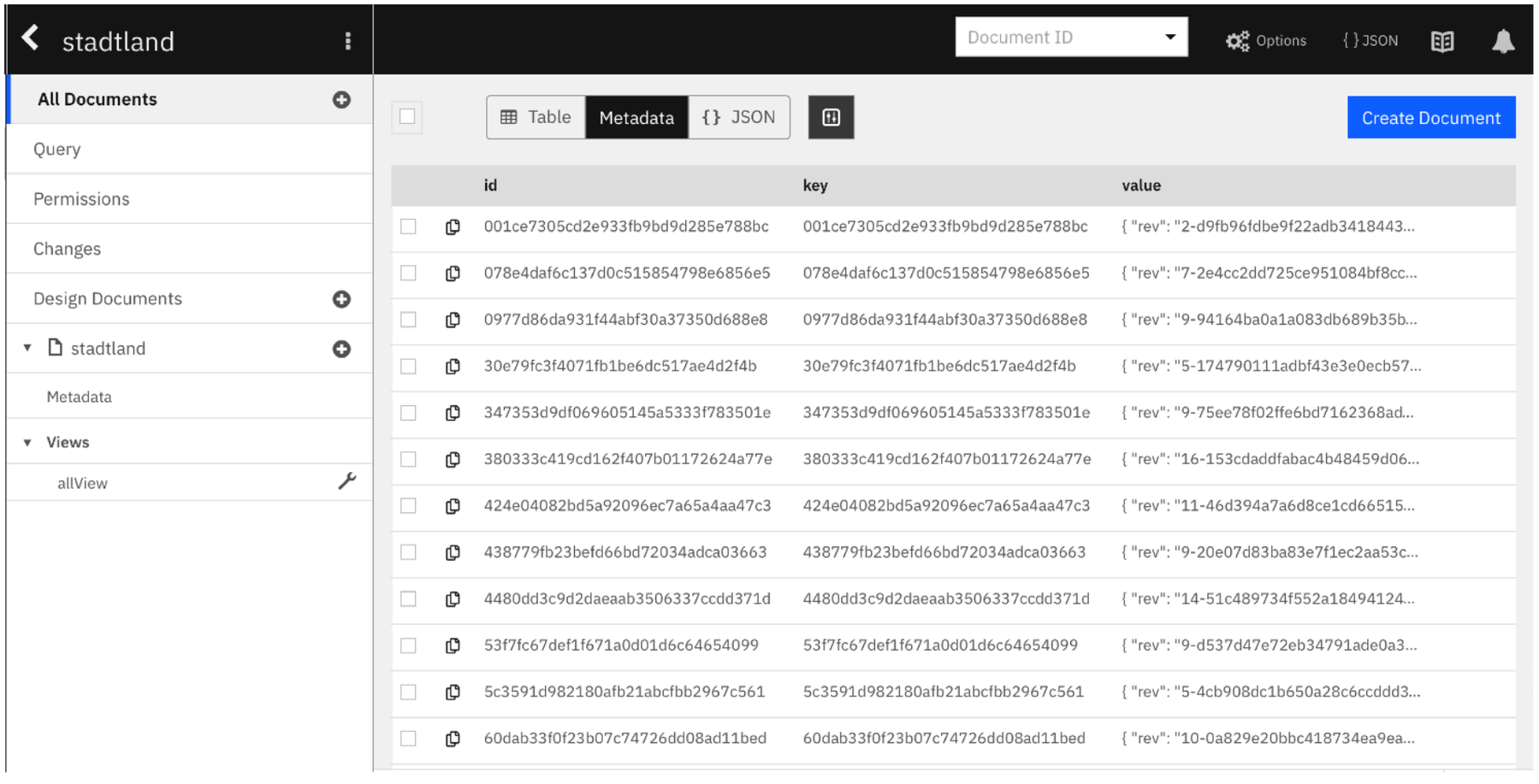


Couch DB Main Features

- Eventual Consistency (CAP)
  - CouchDB guarantees **eventual consistency** to be able to provide both availability and partition tolerance. A later read will eventually (not always) get the latest write.
- Distributed Architecture with Replication
  - Couch DB **implements sharding** or partitioning of documents of a DB to different instances
  - CouchDB was designed with bi-direction replication (or synchronization) and off-line operation in mind. That means multiple replicas can have their own copies of the same data, modify it, and then sync those changes at a later time.
- Document Storage
  - CouchDB stores **data as "documents"**, as one or more field/value pairs **expressed as JSON**. Field values can be simple things like strings, numbers, or dates; but ordered lists and associative arrays can also be used. Every document in a CouchDB database has a unique id and there is no required document schema..
- **Views and Indexes**
  - Views are mechanisms for working with document content in databases. A view can selectively filter documents and speed up the search for content. It can be used to pre-process the results before they're returned to the client. Views are simply JavaScript functions. With the emit Method we can also build new indexes
  - Reduce : the data returned by this function is a ‘row’ for each different key
    - either counting the documents with this key
    - building the sum of the returned values (numeric data)
    - Other statistics
  - Example in StadtLand DB in the IBM Cloud
- HTTP API
  - All items have a unique URI that gets exposed via HTTP. It uses the HTTP methods POST, GET, PUT and DELETE for the four basic CRUD (Create, Read, Update, Delete) operations on all resources.
- Reference : https://en.wikipedia.org/wiki/Apache_CouchDB

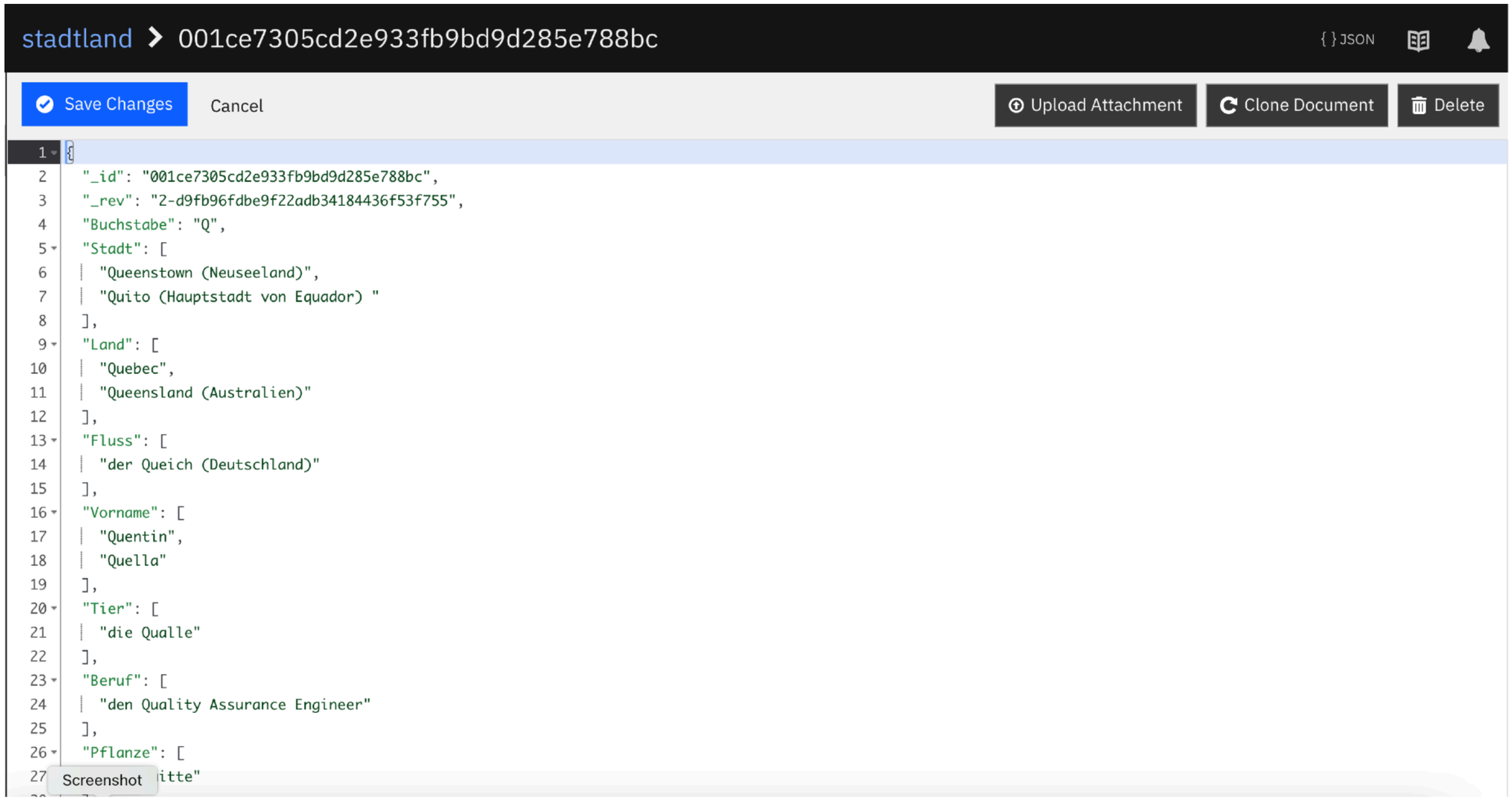
IBM Cloudant a managed Couch DB by IBM

- Cloudant Entities
  - A Service Instance can have 1 or more Databases, a Database can of one or more documents.
  - Data in Documents are described in JSON (it looks like a JSON data persistency service)
  - Documents in the same DB can theoretically be totally different
  - Documents have an own or system generated ID (_id) needed to identify the document and a Revision Code (always system generated) needed for updates (Multi Version Concurrency)
-



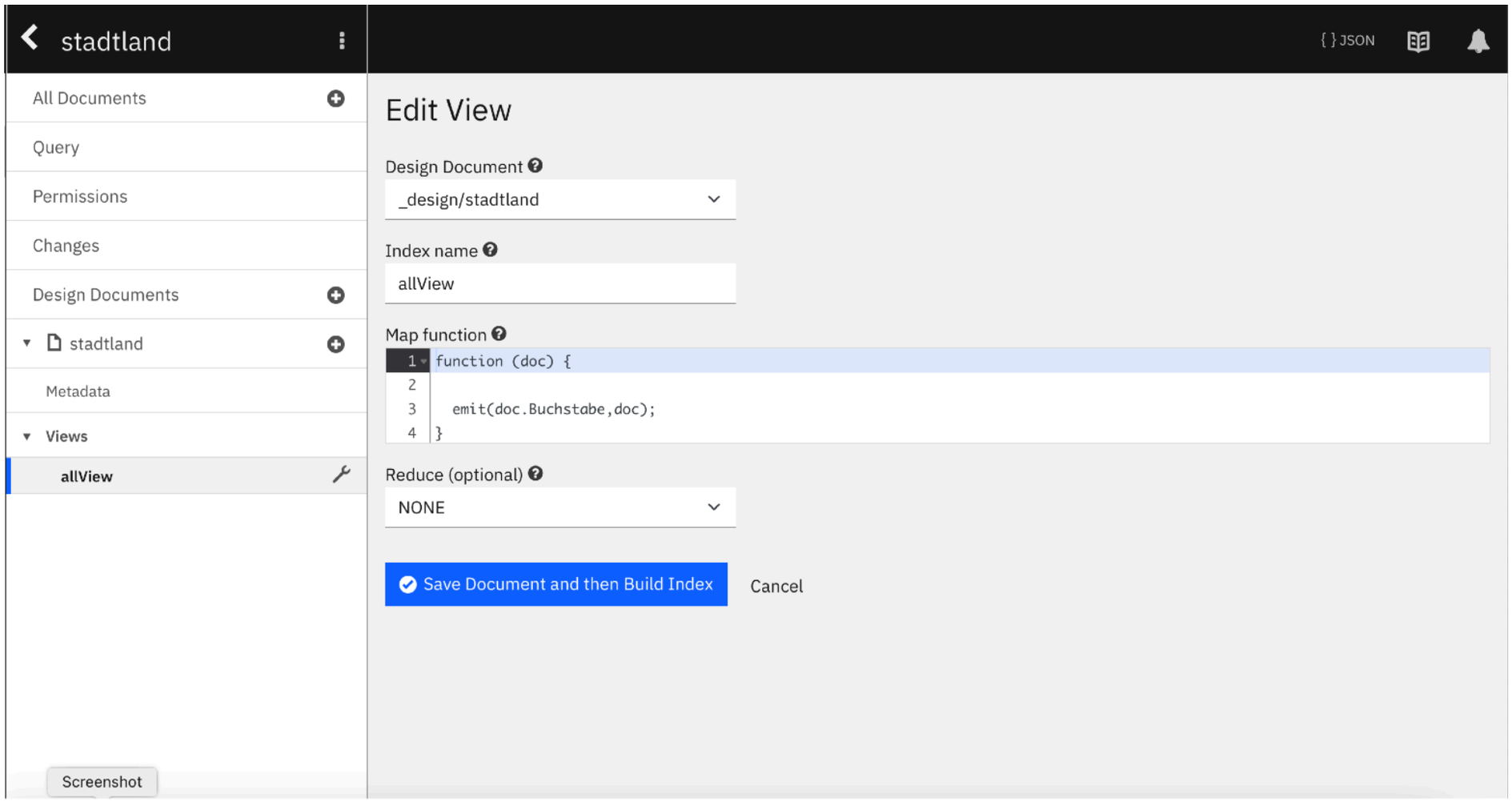
IBM Cloudant a managed Couch DB by IBM

- A Cloudant Document



IBM Cloudant a managed Couch DB by IBM

- Map Reduce (Views) and Index Search
  - A view is basically a JavaScript Function invoked by Couch DB for every document in the selected Database which emits (based on some logic) a subset of data mapped to a key (=Index)
  - Reduce is a function, that is used to produce aggregate results for that view. A reduce function is passed a set of intermediate values and combines them to a single value.
- For Completeness , There are also Geospatial indexes (assuming that the documents carries some geospatial informations)
  - For example, you might specify a polygon object that describes a housing district. You might then query your document database for people that reside within that district by requesting all documents where the place of residence is *contained* within the polygon object).



IBM Cloudant a managed Couch DB by IBM

- **Examples of using Node.JS**

```
// ----------------------------------------------------------------------------
// Setup Couch DB for StadtLand App
//
// cloudantaccess is the major Object for Couch DB
// stadtland is the database given using the .use method
// ----------------------------------------------------------------------------
var Cloudant = require('@cloudant/cloudant');
var cloudantaccess = Cloudant({account:process.env.Cloudant_account,
                        password:process.env.Cloudant_password});
var stadtlanddb = cloudantaccess.use('stadtland');
//----------------------------------------------------------------------------
//  Define the stadtlanddb in the own written stadtland.js package
// ----------------------------------------------------------------------------
stadtland.init(stadtlanddb);
//
```

- In a .env file we have stored the account and password of our Cloudant account.
- With cloudantaccess.use we set the major cloudant object and set the DB (stadtland)

```
  // ----------------------------------------------------------------------------
  // Function read stadtlandberg data from cloudant as promise
  // ----------------------------------------------------------------------------
  readCloudant : function (buchstabe) {
  return new Promise( function (resolve, reject) {
    stadtland.data.db.view('stadtland', 'allView', {key:buchstabe.toUpperCase().trim()},
       function(err, body) {
        if (!err) {
          var returnjson = [];
          body.rows.forEach(function(doc) {
            var singleentry = doc;
            returnjson.push(singleentry);
          });
          resolve(returnjson);
        } else { reject(err)}
      });
  });
}
```

- We invoke a map function (view, design document) and return the one document with the key. Key is set to the letter of the requested stadtdland game.
- Run the curl example (Folder internal/ cloudant/curlcopy)

Recap NoSQL DBs as Data Service

- Was versteht man grob unter dem CAP Theorem ?
- Was ist mit Eventual Consistency gemeint ?
- Nenne zwei Merkmale wo sich noSQL und SQL DBs unterscheiden ?
- Was wird bei Couch DB mit Versioning erreicht ?
- Was kann ich im S3 (COS) durch das Konzepts der Buckets steuern ?
- Beschreibe ein Anwendungsbeispiel für die Nutzung von COS
- Beschreibe ein Anwendungsbeispiel für die Nutzung von Couch DB
- Beschreibe kurz wie ich in einer Couch DB einen Zugriffsindex Index erzeuge.

Referenzen

- Couch DB
  - http://couchdb.apache.org/
- Cloudant
  - https://cloud.ibm.com/docs/Cloudant
  - https://www.npmjs.com/package/@ibm-cloud/cloudant