2. BASES DE DATOS CLAVE-VALOR

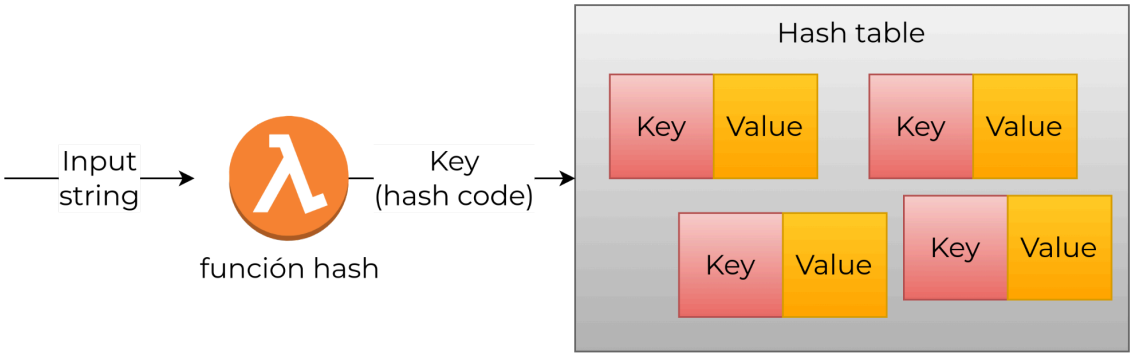
2.1. Key-Value Store

Representa a uno de los *patrones de arquitectura de datos* NoSQL más simple, empleado para representar datos a través de una cadena o llave (key). La Key es empleada para hacer referencia a cierta cantidad de datos a los que se les suele hacer referencia como el *valor* asociado a la Key. De aquí la pareja Key-Value.

Este patrón es similar al concepto de ***Hash table***. La key es generada a partir del resultado producido por una función llamada ***función hash***. La función hash ejecuta un algoritmo encargado de producir un valor que será asignado a la Key, conocido como ***hash code***.

El algoritmo es el encargado de generar valores que cumplan con ciertos requerimientos. Entre los más destacados:

- Producir valores únicos (pudieran existir colisiones)
- Producir valores cuya distribución permite realizar búsquedas eficientes al interior del hash table.
- Seguridad. Los valores generados por la función pueden ser empleados como llaves de cifrado.



Key-Value stores en realidad no definen un lenguaje para realizar consultas. Las operaciones que ofrecen para administrar los datos son simples:

- `add(key,value)`
- `update(key,value)`
- `remove(key)`
- `search(key) → value`

Otra analogía con este tipo de almacenamiento es un diccionario:

- Key: Palabra
- Value: El texto que explica su significado. Al almacenarse la estructura o el detalle de este texto es irrelevante para la base de datos, puede representarse como un simple conjunto de bytes (BLOB).

La key es generalmente ***indexada*** de tal forma que las búsquedas a partir del valor de la key se procesan de forma eficiente sin importar el tamaño del hash table o del store.

Al no especificar detalles en cuanto al contenido del valor, este puede almacenar cualquier cantidad de datos sin importar su estructura o su tipo. La aplicación es la encargada de decodificar su contenido.

La Key es una cadena que puede ser formada por diversos formatos, depende del caso de estudio:

- Paths, URLs o URIs que representan algún recurso: imágenes, archivos, etc.
- Cadenas artificiales generadas por una función hash.
- Llamadas a servicios Web tipo REST.
- Sentencias SQL.

Ejemplos:

	Key	Value
Nombre del archivo	`img-21302-jpg`	Archivo binario de una imagen
URL	`https://docs.google.com/d/mydoc.html`	Página web
Sistema de archivos	`c:/mydocs/tasks/task01.doc`	Documento en word
MD5 Hash	`92sdsew4324dsdw4d3rdwe3`	esteEsElPassword
REST call	`get-student-list?school=123&format=json`	`[ {id=1,nombre”paco”}, {id=2,nombre”paco”} ]`
SQL query	`select * from product;`	`<prod> <id>1</id> <clave>pr-005</clave> </prod>`

Key-Value store fue reconocido como un ***patrón de arquitectura de datos NoSQL*** alrededor de los 1990s, popularizado inicialmente por una base de datos llamada Berkley DB.

2.2. Beneficios de los Key-Value Store

La simplicidad de un Key-Value Store permite ofrecer una serie de beneficios y una amplia gama de usos prácticos.
Esta simplicidad permite generar diseños de aplicaciones (software) mucho más adecuados que permitan optimizar el uso de los recursos de hardware, así como reducir

los costos de operación para las empresas que hacen uso de los servicios en la nube. Dentro de los principales beneficios se encuentran:

- Mayor precisión para configurar niveles de servicio
- Mayor precisión de monitoreo y notificaciones
- Escalabilidad y confiabilidad
- Portabilidad a bajo costo.

2.2.1. Mayor precisión en niveles de servicio

Debido a la simplicidad de esta arquitectura, el diseño de la aplicación permite enfocarse en definir ciertos niveles de servicio que ofrecen como beneficio la obtención de una aplicación de alta calidad: servicios confiables que pueden operar adecuadamente bajo diversas condiciones de carga, etc.

Estos niveles de servicio se emplean también para reducir costos de operación en un sistema que corre sobre Cloud, ya que permiten identificar con precisión la cantidad de recursos que se van contratar y por tanto, la reducción de costos.

Ejemplos de niveles de servicio

- El tiempo máximo que debe tardar una búsqueda
- El tiempo máximo que debe tardar una escritura
- Cantidad máxima de lecturas por segundo que debe soportar
- Cantidad máxima de escrituras por segundo que debe soportar
- Cantidad de copias (respaldos) para garantizar confiabilidad
- Nivel de replicación en cada nodo
- Configuración de comportamiento transaccional: consistencia vs disponibilidad.

Algo interesante de estos niveles de servicio es que son configurables y se pueden ajustar de forma dinámica a través de los servicios en Cloud. Esta característica permite:

- La aplicación solicitará más recursos cuando se encuentre en un periodo de carga máxima (aumento de costo)
- Los recursos solicitados pueden liberarse cuando exista una carga baja y por lo tanto los costos de operación bajarán.

2.2.2. Mayor precisión en monitoreo y notificaciones

Permite el uso de diversas herramientas que pueden generar reportes muy adecuados y precisos que pueden mostrar en tiempo real la forma en la aplicación se está desempeñando así como mostrar los niveles de servicio esperados bajo una situación de alta carga.

Otro servicio importante es el envío de notificaciones, emails cuando alguno de estos niveles se sale de los valores máximos tolerables que en conjunto con el uso de

herramientas de monitoreo, permitirán tomar medidas necesarias para garantizar la disponibilidad de la aplicación.

2.2.3. Escalabilidad y confiabilidad

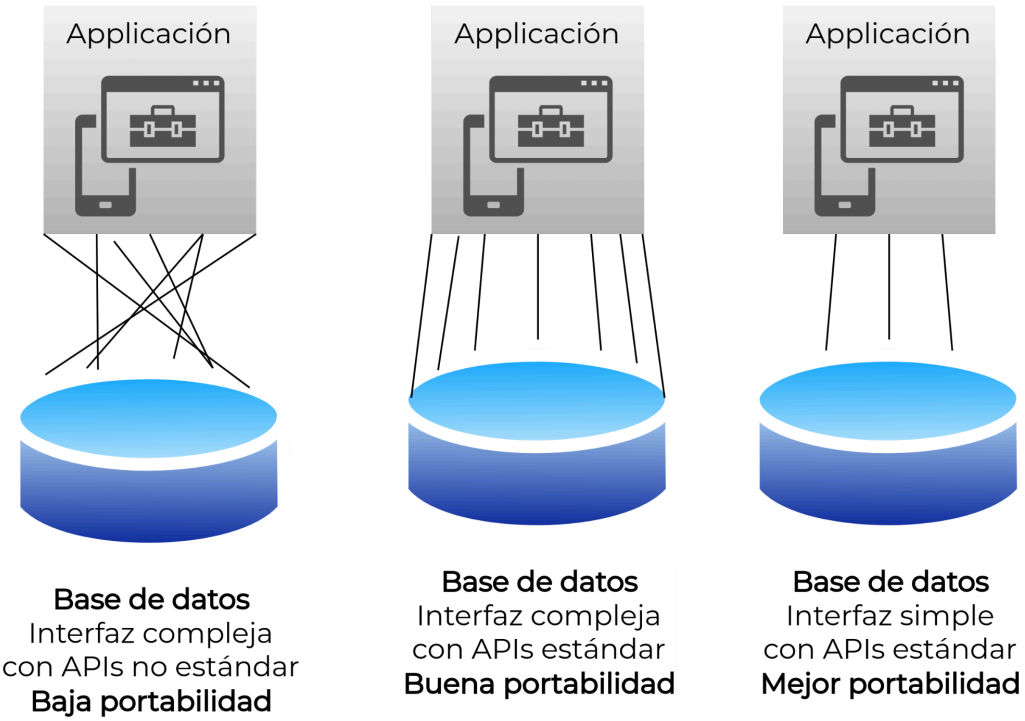
Regla importante: Entre más simple sea la arquitectura e interfaz de acceso a datos, las aplicaciones serán altamente escalables y confiables. Hacer ***tuning*** de una aplicación generalmente es una tarea constante y complicada sobre todo si la arquitectura de los datos es compleja.

2.2.4. Portabilidad y costos operacionales bajos

Una forma de reducir costos operacionales y mejorar desempeño es la posibilidad de migrar o de evolucionar una aplicación hacia otras tecnologías o herramientas. Esto dependerá de qué tan portable (flexible) sea una aplicación para migrar o evolucionar. Si la interfaz o capa de persistencia de una aplicación es compleja, tendrá una baja portabilidad.

Complejidad significa que la base de datos ofrece una gama muy amplia de operaciones (interfaz con muchas funciones) , por qué migrar a otras soluciones implicaría encontrar un equivalente para cada una de las funciones u operaciones que la aplicación utilice. En algunos casos, no existirán equivalentes. Por su simplicidad, un Key-Value Store ofrece alta portabilidad.

La interfaz de base de datos pudiera ofrecer una interfaz muy amplia y ser portable a la vez, siempre y cuando su interfaz o API pertenezca a un ***estándar***. De esta forma la aplicación puede ser migrada a otra solución sin mayor problema ya que se sigue un mismo estándar.



Para un Key-Value store el API o interfaz para realizar la manipulación de datos es simple y estándar

- `put($key as xs:string, $value as item())`
  - Agrega un nuevo elemento o lo actualiza si existe
- `get($key as xs:string) as item()`
  - Obtiene un elemento con base al valor de su Key
- `delete($key as xs:string)`
  - Elimina un elemento con base a su Key, o error en caso de no encontrar al elemento.

Notar el nombre de las funciones: `put`, `get` y `delete`, corresponden al estándar REST API (Representational State Transfer Application Programming Interface) empleada en los servicios web tipo REST.

Tarea: Investigar características principales de los siguientes estándares de arquitectura de datos empleados ampliamente en la industria de las bases de datos.

- REST API
- XQuery
- Estándar SQL/JSON

2.3. Principales funcionalidades de un Key-Value Store

Las funcionalidades que se muestran a continuación se enfocan como en todos los sistemas NoSQL a los conceptos de consistencia, transacciones, consultas, estructura de los datos y escalabilidad.

2.3.1. Consistencia

- Se aplica únicamente a nivel de una operación simple (`put`, `delete` o `get`), es decir, si 2 transacciones intentan modificar un mismo elemento, la BD se encarga de controlar este evento.
- Escrituras con enfoque optimista podrían implementarse por la aplicación ya que la BD no tiene conocimiento del ***value***, pero es complicado y costoso. Recordando, escritura optimista permite que 2 transacciones actualicen un mismo dato de forma concurrente (consistencia eventual).
- Riak implementa *consistencia eventual distribuida* (permite resolver conflictos de consistencia cuando ocurre una replicación parcial).

2.3.2. Transacciones

Cada producto tiene especificaciones diferentes, pero a nivel general, no hay garantías en las escrituras. Riak utiliza un concepto llamado Quorums.

2.3.3. Búsquedas

- Las búsquedas siempre se realizan empleando la Key. Si se desea realizar una búsqueda basada en el value, se deberá implementar en la aplicación, la BD no se puede emplear para este propósito.
- ***Riak search*** ofrece funcionalidades para realizar búsquedas en el ***Value*** haciendo uso de *Lucene indexes*.

2.3.4. Escalabilidad

- La mayoría de los Key-Value store hacen uso de sharding. El valor de la Key determina el nodo en donde el objeto será almacenado.
- De forma similar, la mayoría de los productos ofrecen opciones y configuraciones para controlar los 2 extremos del teorema de CAP.

2.4. Casos de estudio Key-Value Store

- Google Web crawler
- Amazon simple storage service (S3)
- Almacenamiento de sesiones de usuario en sitios web
- Recomendaciones (advertising) en tiempo real de sitios web
- In-memory data caching
- User profiles, user preferences
- Shopping cart (carritos de compra)

2.4.1. ¿Dónde no emplear Key-Value stores?

- Si se requiere establecer relaciones entre conjuntos de datos, o realizar correlaciones entre set de datos
- Si se requiere transaccionalidad entre múltiples operaciones, por ejemplo, atomicidad entre varias operaciones `put`.
- Realizar búsquedas con criterios basados en el contenido o ***value*** asociado a la key. Recordar que los datos almacenados (value) asociados a cada key, se tratan como si fueran datos BLOB, (caja negra).
- Operaciones que requieran manejar listas, por ejemplo, obtener una lista de sesiones, etc. La recuperación de datos siempre regresa un solo elemento a la vez.

2.4.2. Ejemplos de Key-Value Stores

- Riak
- Redis
- Memcached DB
- Berkeley DB
- Hamster DB
- Amazon DynamoDB
- Project Voldemort

2.5. Riak

Riak está formada por un conjunto de bases de datos distribuidas que en lo general ofrecen alta confiabilidad y escalabilidad.

- Riak KV Representa a una base de datos NoSQL
- Riak TS (Time Series database) optimizada para IoT
- Riak S2 Empleada para optimizar almacenamiento masivo de datos así como su integración con otros servicios de datos como son: Apache Spark, Redis Caching, Apache Solr, etc.

Las principales características de Riak:

- Alta disponibilidad
- Escalabilidad
- Tolerancia a fallas
- Simplicidad operacional

2.5.1. Riak KV

Es una base de datos NoSQL distribuida altamente disponible y escalable. Ofrece sharding automático. El diseño de Riak KV se centra en implementar cuatro funciones: *escalabilidad masiva*, *simplicidad*, *tolerancia a fallo y sencillez en las operaciones*.

Riak KV tiene una arquitectura multinodos sin maestro que proporciona lecturas y escrituras rápidas incluso en el caso de fallas de red o hardware.

Para garantizar la disponibilidad de los datos y la tolerancia de partición, Riak KV replica los datos de forma predeterminada en tres nodos del clúster.

No hay un solo maestro, no hay un solo punto de falla, y cualquier nodo puede atender cualquier solicitud entrante. Con su arquitectura maestra, Riak KV es fácil.

Otras características:

- Replicación multicluster automática
- Consultas a texto empleando Apache Solr
- Global object expiration
- Riak distributed data types
- APIs Robustas y diversas librerías cliente
- Integración con Redis

2.5.2. Instalación de Riak KV en Ubuntu/Mint

- Versión y Sitio web: https://www.tiot.jp/riak-docs/riak/kv/3.2.0
- Sitio de descargas https://www.tiot.jp/riak-docs/riak/kv/3.2.0/downloads

A. Instalar dependencias

```
apt-get install libc6 libc6-dev libc6-dbg
```

B. Obtener paquete Ubuntu Jammy Jellyfish (22.04). Si se cuenta con una versión diferente, descargar el paquete del sitio antes mencionado y almacenarlo en el escritorio.

```
cd ~/Desktop
mkdir riak
cd riak
wget https://files.tiot.jp/riak/kv/3.2/3.2.0/ubuntu/jammy64/riak_3.2.0-OTP25_amd64.deb
```

C. Instalar Riak

```
sudo dpkg -i riak_3.2.0-OTP25_amd64.deb
```

D. Configurar parámetro del Kernel para aumentar el número de archivos abiertos que pueden existir en el sistema

- Agregar las siguientes líneas en `/etc/security/limits.conf`

```
riak soft nofile 65536
riak hard nofile 200000
```

E. Iniciar el servidor

```
sudo riak start
```

F. Validar la instalación, verificar que se obtenga la siguiente salida: `pong`

```
sudo riak ping
```

G. Obtener los properties de la BD empleando el comando `curl`

```
curl -v http://127.0.0.1:8098/types/default/props
```

2.5.3. Conceptos básicos en Riak

2.5.3.1. Buckets

- Buckets son empleados para definir un e*spacio virtual de almacenamiento* de objetos Riak.
- Un bucket puede considerarse como el equivalente a una tabla en el modelo relacional

2.5.3.2.Bucket types

- Un bucket puede ser configurado empleando ***bucket types***.
- Un bucket Type permite crear y modificar un conjunto de configuraciones las cuales pueden ser aplicadas a un conjunto de buckets.

Ejemplos de parámetros

Parámetro	**Descripción**
`n_val`	Número de copias de cada objeto que serán almacenadas en el cluster. Por default es 3.
`last_write_wins`	Indica si el timestamp de un objeto será utilizado para decidir qué cambio es el que se persiste bajo un ambiente de concurrencia (múltiples y simultáneas escrituras).
`precommit, postcommit`	Funciones escritas en Erlang que pueden ser invocadas antes y después de realizar una escritura de un objeto.
`datatype`	En caso de emplear Riak data types, indica el tipo de dato que será empleado en el bucket. Posibles valores: counter, set, map

De forma adicional, es posible crear nuevos bucket types

```
riak admin bucket-type create robert
riak admin bucket-type create earth '{"props":{"n_val":2}}'
riak admin bucket-type create fire '{"props":{"n_val":2}}'
riak admin bucket-type create wind '{"props":{"n_val":2}}'
```

Para que el datatype pueda emplearse posterior a su creación, este debe ser activado.

```
riak admin bucket-type activate <bucket_type>
```

Mostrar el status del bucket type

```
riak admin bucket-type status <bucket_type>
```

Otro uso de un bucket type es agregar un tercer nivel de Jerarquía para organizar e identificar objetos.

Cada objeto puede ser identificado a 3 niveles: `type -> bucket -> key`. A este tercer nivel que agrega el bucket type se le conoce como ***Additional Namespace.***

2.5.3.3. Keys

Cadenas empleadas para identificar objetos. Cada bucket representa a un ***keyspace*** diferente. La pareja Key- Objeto se considera como una entidad simple.

2.5.3.4. Objects

Representan la única unidad de almacenamiento. Un objeto representa a una estructura identificada por un Key. Cada objeto puede estar formado por los siguientes elementos:

- Bucket al que pertenece
- key
- Vector clock
- Metadatos (parejas key - value)

2.5.4. Modelado de objetos en Riak

- El mejor desempeño se obtiene cuando una aplicación es construida en términos de operaciones CRUD (`create`, `read`, `update`, `delete`).
- Para localizar a un objeto se consideran 3 niveles llamados locators:
  - La Key del objeto
  - El bucket al que pertenece

    - El tipo de dato del bucket en cual determina la configuración de replicación y otras propiedades.
- Para navegar o explorar el contenido de un bucket, se puede aplicar el concepto de ***nested key/value hash***

Ejemplo:

```
simpsons = {
  'season 1': {
    { 'episode 1': 'Simpsons Roasting on an Open Fire' },
    { 'episode 2': 'Bart the Genius' },
    # ...
  },
  'season 2': {
    { 'episode 1': 'Bart Gets an "F"' },
    # ...
  },
  # ...
}
```

Cada nivel de anidamiento puede definirse por una key. En este ejemplo `Season ->episode`.

- SI se desea obtener los datos de la temporada 2 se escribe lo siguiente:

```
simpsons['season 4']['episode 12']
```

Ambos valores son tratados como keys. A esta técnica se le llama ***lookup operations***. Esta característica es muy rápida ya que no se tiene que hacer búsquedas entre renglones y columnas para encontrar al objeto: `bucket/key address`.

- Estructura de la URL:

```
GET/PUT/DELETE /bucket/<season>/keys/<episode number>
```

El ejemplo anterior puede extenderse con un bucket type:

```
simpsons = {
  'good': {
    'season X': {
      { 'episode 1': '<title>' },
      # ...
```

```
    }
  },
  'bad': {
    'season Y': {
      { 'episode 1': '<title>' },
      # ...
    }
  }
}
```

- URL

```
GET/PUT/DELETE /types/<good or bad>/buckets/<season>/keys/<episode number>
```

- Búsqueda

```
simpsons['good']['season 8']['episode 6']
```

2.5.5. Creando objetos en Riak

- Forma básica de un URL para realizar escrituras

```
PUT /types/<type>/buckets/<bucket>/keys/<key>
```

Para efectos del curso solo se cubrirá el uso del API Rest de Riak (HTTP API) en cual se revisa en la siguiente sección. Los siguientes ejemplos muestran la forma de creación de objetos empleando algunos de los diversos lenguajes de programación que tienen soporte para manipular datos con Riak.

Almacenar un objeto que contiene información de un perro llamado `Rufus` con key `rufus`, en el bucket `dogs` con el tipo de bucker `animals`.

Ejemplo con Java

```
String quote = "WOOF!";
Namespace bucket = new Namespace("animals", "dogs");
Location rufusLocation = new Location(bucket, "rufus");
RiakObject rufusObject = new RiakObject()
  .setContentType("text/plain")
  .setValue(BinaryValue.create(quote));
StoreValue storeOp = new StoreValue.Builder(rufusObject)
```

```
  .withLocation(rufusLocation)
   .build();
client.execute(storeOp);
```

Ejemplo en JavaScript

```
var riakObj = new Riak.Commands.KV.RiakObject();
riakObj.setContentType('text/plain');
riakObj.setValue('WOOF!');
client.storeValue({
  bucketType: 'animals', bucket: 'dogs', key: 'rufus',
  value: riakObj
}, function (err, rslt) {
    if (err) {
        throw new Error(err);
    }
});
```

Ejemplo con el API Rest

Almacenar un objeto JSON. Para realizar peticiones hacia los servicios Rest de Riak se emplea el comando `curl`.

```
curl -v -X POST -d '
  { "clienteId" : 210839028,
    "nombre":  "Diego",
    "codigoPais": "mx"
  }
'
-H "Content-type: application-JSON"
http://localhost:8098/buckets/session/keys/21983we29e
```

2.5.6. HTTP API

Riak define un API HTTP que permite realizar una gran variedad de operaciones. La lista de URLs puede consultarse en https://www.tiot.jp/riak-docs/riak/kv/3.2.0/developing/api

El API puede emplearse haciendo uso del comando `curl`.

El comando `curl` se emplea para transferir datos de o hacia un servidor empleando alguno de los siguientes protocolos: HTTP, FTP, IMAP, POP3, SCP, SFTP, SMTP, TFTP, TELNET, LDAP, o FILE.

2.5.6.1. Almacenando objetos con HTTP

Estructura del URL

```
POST /types/type/buckets/bucket/keys        # Riak-defined key
PUT /types/type/buckets/bucket/keys/key     # User-defined key
POST /buckets/bucket/keys                   # Riak-defined key
PUT /buckets/bucket/keys/key                # User-defined key
```

Headers obligatorios

- Content-type
- X-Riak-Vclock
- X-Riak-Meta-*
- X-Riak-Index-*

Headers opcionales, sólo para operaciones PUT

- If-None-Match
- If-Match, I
- f-Modified-Since
- If-Unmodified-Since

Query params opcionales

- w
- dw
- pw
- return body

Ejemplo sin Key

```
curl -v http://127.0.0.1:8098/buckets/test/keys \
       -H "Content-Type: text/plain" -d 'this is a test'
```

Salida:

```
*   Trying 127.0.0.1:8098...
* Connected to 127.0.0.1 (127.0.0.1) port 8098 (#0)
> POST /buckets/test/keys HTTP/1.1
> Host: 127.0.0.1:8098
```

```
> User-Agent: curl/7.81.0
> Accept: */*
> Content-Type: text/plain
> Content-Length: 14
>
* Mark bundle as not supporting multiuse
< HTTP/1.1 201 Created
< Vary: Accept-Encoding
< Server: MochiWeb/2.20.0 WebMachine/1.11.1 (greased slide to failure)
< Location: /buckets/test/keys/7x9bX2FnXkbp5v2x5Qi7DK3jeKz
< Date: Wed, 22 Feb 2023 06:57:07 GMT
< Content-Type: text/plain
< Content-Length: 0
<
* Connection #0 to host 127.0.0.1 left intact
```

Ejemplo con key

```
 curl -v -XPUT -d '{"bar":"baz"}' -H "Content-Type: application/json" -H
"X-Riak-Vclock: a85hYGBgzGDKBVIszMk55zKYEhnzWBlKIniO8mUBAA=="
http://127.0.0.1:8098/buckets/test/keys/doc?returnbody=true
```

2.5.6.2.Consultar objetos con HTTP

Estructura del URL

```
GET /types/type/buckets/bucket/keys/key
GET /buckets/bucket/keys/key
```

Ejemplo

```
curl -v http://127.0.0.1:8098/buckets/test/keys/doc2
```

2.5.7. Riak Secondary Indexes (2i)

Secondary Indexes (2i) permiten agregar ***tags*** a objetos almacenados en Riak al momento de almacenarlos. Las Tags permiten realizar búsquedas con 2 criterios:

- Igualdad: Mostrar a todos los objetos que comparten un tag en específico
- Rango: Para un tag numérico, mostrar los objetos cuyo tag esté en el rango [x,y].

Existe otra herramienta más sofisticada llamada Riak Search. Su función se basa en el parseo tanto del key como del value y construye un índice empleando un esquema tipo Solr. Esta funcionalidad se considera como ***deprecated*** , por lo que se sugiere emplear 2i.

2.5.7.1. Funcionalidades de 2i

- Permite 2 tipos de atributos secundarios: enteros y cadenas.
- Permite realizar búsquedas por igualdad o por rango sobre un índice.
- Permite paginación de resultados
- Permite realizar streaming de resultados.

2.5.7.2. Escenarios de uso de 2i

- Se requiere localizar objetos más allá de los 3 niveles: bucket type, bucket, key.
- El value es un objeto binario por lo que carece de criterios de búsqueda
- Los criterios de búsqueda son simples (por igualdad o por rango).

2.5.7.3. Ejemplos

- Agregar un objeto y asignarle tags a un índice

```
curl -XPOST localhost:8098/types/default/buckets/users/keys/john_smith \
  -H 'x-riak-index-twitter_bin: jsmith123' \
  -H 'x-riak-index-email_bin: jsmith@riak.info' \
  -H 'Content-Type: application/json' \
  -d '{"userData":"data"}'
```

- Realizar consultas por valor

```
curl localhost:8098/buckets/users/index/twitter_bin/jsmith123
```

- Realizar consultas por rango

```
curl localhost:8098/types/indexes/buckets/people/index/field2_int/1002/1004
```

Para poder ejecutar estos comandos, el tipo de storage de Riak debe ser configurado con un backend que soporte manejo de índices: ***LevelDB*** o ***Memory***.

Ejemplo, editar `/etc/riak.conf`

```
storage_backend = memory
```

2.5.8. Riak Search (deprecated)

Permite realizar búsquedas empleando el contenido del ***value*** empleando *Lucene Indexes*.
Riak Search representa una integración entre los siguientes 2 elementos:

- Solr: Para realizar indexado y búsquedas haciendo uso del value
- Riak: Empleada para realizar el almacenamiento y distribución de los datos

Existen 3 conceptos importantes para almacenar datos y realizar búsquedas de forma adecuada:

- ***Schemas***: Le permiten a Solr como realizar el indexado de atributos
- ***Índices***: Empleados para realizar búsquedas
- ***Asociación bucket-índice***: Permite notificar a Riak cuando deben aplicarse índices al *value*.

Riak Search debe ser configurado con un *esquema* de Solr que le permita conocer cómo indexar atributos del *value*. De no existir se emplea un esquema llamado `_yz_default`.

El siguiente paso es crear un índice, el cual representa una colección de datos similares empleados para realizar consultas. Al crear un índice se puede especificar un *esquema*. De no especificarlo, se emplea el esquema por default.

Ejemplo
Creando un índice con el esquema por default.

```
export RIAK_HOST="http://localhost:8098"
curl -XPUT $RIAK_HOST/search/index/famous
```

Para especificar el esquema se emplea la siguiente sintaxis:

```
curl -XPUT $RIAK_HOST/search/index/famous \
     -H 'Content-Type: application/json' \
     -d '{"schema":"_yz_default"}'
```

El último paso para configurar las búsquedas es la asociación del índice ya sea con un *bucket type* (todos los buckets al que se le apliquen este tipo tendrán asociado al índice) o con un bucket específico. Para ambos casos el índice se asocia empleando la propiedad llamada `search_index`.

Ejemplo

- Asociación con un bucket

```
curl -XPUT $RIAK_HOST/types/animals/buckets/cats/props \
     -H 'Content-Type: application/json' \
     -d '{"props":{"search_index":"famous"}}'
```

- Asociación con un bucket type

```
riak admin bucket-type create animals '{"props":{"search_index":"famous"}}'
riak admin bucket-type activate animals
```

2.5.9. Realizar búsquedas

Antes de realizar búsquedas se debe realizar una configuración de seguridad que limita el uso de esquemas e índices.

```
riak admin security grant search.admin on schema to username
riak admin security grant search.admin on index to username
riak admin security grant search.query on index to username
riak admin security grant search.query on index famous to username
```

Ejemplo

- Insertar datos

```
curl -XPUT $RIAK_HOST/types/animals/buckets/cats/keys/liono \
     -H 'Content-Type: application/json' \
     -d '{"name_s":"Lion-o", "age_i":30, "leader_b":true}'

curl -XPUT $RIAK_HOST/types/animals/buckets/cats/keys/cheetara \
     -H 'Content-Type: application/json' \
     -d '{"name_s":"Cheetara", "age_i":28, "leader_b":false}'

curl -XPUT $RIAK_HOST/types/animals/buckets/cats/keys/snarf \
     -H 'Content-Type: application/json' \
     -d '{"name_s":"Snarf", "age_i":43}'

curl -XPUT $RIAK_HOST/types/animals/buckets/cats/keys/panthro \
     -H 'Content-Type: application/json' \
     -d '{"name_s":"Panthro", "age_i":36}'
```

Para realizar el indexado de datos se emplean ***extractors***.

Un extractor acepta un Riak Value de un determinado Content Type y realiza una conversión de una lista de atributos que pueden ser indexados. Existen extractors específicos a cada Content Type.
Para determinar las columnas a indexar se revisan los nombres de los atributos en el documento JSON, XML, etc. Se emplean los siguientes sufijos para identificar a la lista de atributos:

- `_s` representa a un string
- `_i` representa a un número entero
- `_b` representa un dato binario, etc.

Ejemplo de una búsqueda simple

```
curl "$RIAK_HOST/search/query/famous?wt=json&q=name_s:Lion*" | json_pp
```

La respuesta de la consulta anterior no es un Riak Value. En su lugar se obtiene un documento JSON

```
{
  "numFound": 1,
  "start": 0,
  "maxScore": 1.0,
  "docs": [
    {
      "leader_b": true,
      "age_i": 30,
      "name_s": "Lion-o",
      "_yz_id": "default_cats_liono_37",
      "_yz_rk": "liono",
      "_yz_rt": "default",
      "_yz_rb": "cats"
    }
  ]
}
```

2.5.10.Clusters en Riak

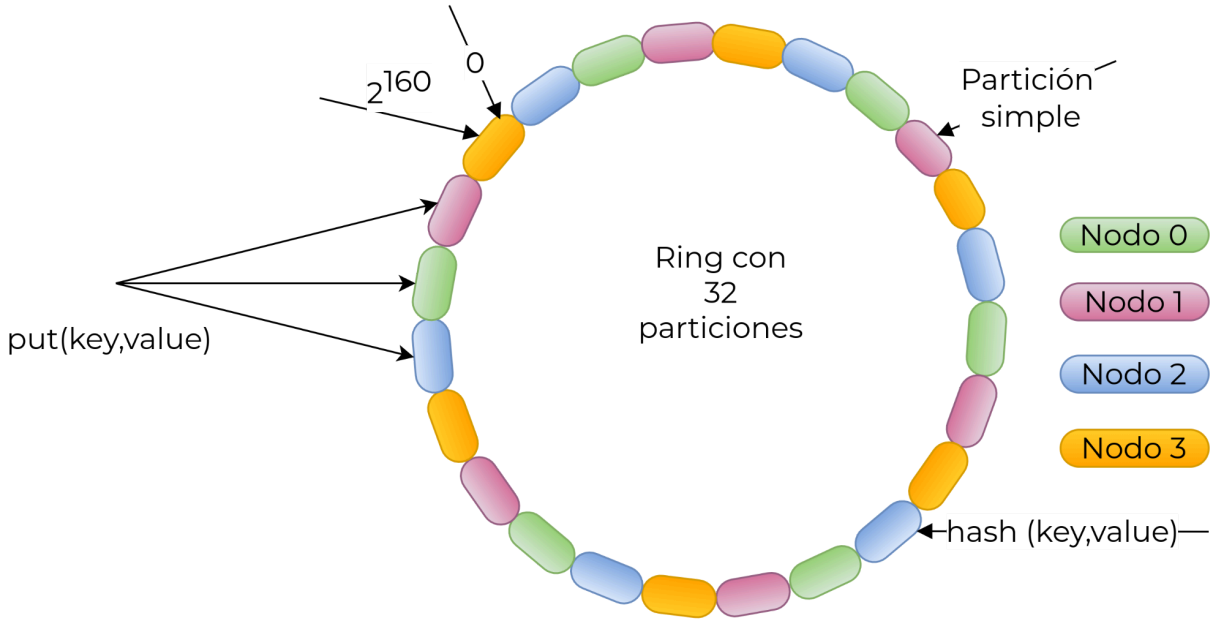
Como se mencionó anteriormente, un cluster está formado por un conjunto de nodos. En cada nodo se ejecuta una una instancia de Riak llamada ***Riak node.***

- Cada Riak node puede administrar a un conjunto de nodos virtuales (vnodes) encargados de almacenar cierta parte de los objetos key/value del cluster
- Los nodos en Riak no son considerados como clones. En una operación no necesariamente participarán los nodos.

2.5.10.1. Ring

Internamente Riak realiza el cálculo de un hash de 160 bits para cada objeto key/value. El resultado se mapea a una posición específica de un ***anillo*** ordenado.

- El anillo está dividido en particiones. A cada partición se le asocia un ***vnode.***



- A nivel general cada nodo del cluster es responsable de un cierto número de vnodes, es decir, `vnodes = #particiones / #nodos,` en este caso `vnodes = 32/4 = 8` ***vnodes***.
- El número de particiones se puede configurar.

2.5.10.2. Replicación inteligente

Operación PUT

- Cuando un objeto se almacena en el cluster, cualquiera de los nodos puede actuar como ***nodo coordinador*** para esa petición en específico.
- El nodo coordinador consulta el estado del anillo para determinar el vnode y la partición a la cual pertenece el objeto Key/value por lo que se enviará una petición de escritura al ***vnode*** correspondiente así como a las N-1 particiones en el ring donde se hará una copia. N es un valor configurado que indica el número de copias del objeto a almacenar.
- Existe otro parámetro W (W < N) que indica el número mínimo de copias que deben ser almacenadas de forma exitosa.

Estos parámetros se explican en la siguiente sección

Operación GET

- Opera de forma similar, se envía la petición al vnode correspondiente el cual está asociado con la key del objeto. De forma adicional se envían peticiones a los N-1 particiones donde hay una copia.
- Existe un parámetro R (R <= N) que indica el número de vnodes que debe responder antes de regresar una respuesta.

Ejemplo

- Suponer el objeto `o1` y un valor de `N=3`. La key de `o1` es asignada a 3 particiones de las 32 disponibles.
- Cuando se realiza una lectura de `o1` el estado del ring será empleado para determinar a las particiones involucradas para realizar la lectura. Existen otros parámetros que le indican a Riak que hacer en caso de que el objeto no sea encontrado de forma inmediata.

2.5.11. Replicación en Riak

Riak está pensada para trabajar en cluster multinodos. Los datos son distribuidos a través de múltiples servidores ofreciendo características como alta disponibilidad y tolerancia a fallas. Riak implementa el teorema de CAP de 2 formas diferentes:

- AP: Disponibilidad y tolerante a la partición a través de ***consistencia eventual***
- CP: Consistente y tolerante a la partición a través de ***consistencia estricta***

RIak puede ser configurado para implementar estas 2 opciones con base a las necesidades de la aplicación.

La replicación en Riak puede operar de 2 formas principales:

- A nivel de petición. Se realizan configuraciones de replicación empleando properties por cada objeto
- A través del uso de bucket types. Los properties se configuran al definir el bucket type. Estas configuraciones se aplicarán a todos los objetos que contenga. Ejemplo:

```
riak admin bucket-type create custom_props '{"props":{"n_val":5,"r":3,"w":3}}'
riak admin bucket-type activate custom_props
```

2.5.11.1. Principales parámetros de configuración

Parámetro	Nombre común	Valor por default	Descripción
n_val	N	3	Factor de replicación. Indica el número de nodos donde se almacenará el objeto.
r	R	quorum	Número de servidores que deben responder en

Parámetro	Nombre común	Valor por default	Descripción
			una operación de lectura.
w	W	quorum	Número de servidores que deben responder en una operación de escritura. Quorum significa que la mayoría de los nodos deben responder (por lo menos la mitad más 1).
pr	PR	0	Número de vnodes primarios que deben responder en una operación de lectura
dw	DW	0	Número de vnodes primarios que deben responder en una operación de escritura
notfound_ok		true	Determina el comportamiento de Riak cuando un objeto no se encuentra. Si su valor es `true`, Riak regresará un error `not found` si el objeto no existe en el primer nodo que responda indicando que el objeto no existe. Si su valor es `false`. Riak seguirá intentando encontrar el objeto en los otros N nodos restantes (N = factor de replicación).

- ¿ Qué pasa si N, R y W tienen valores altos ? Mencionar Ventajas y desventajas
- ¿Qué pasa si R tiene un valor muy pequeño? Por ejemplo, 1. Mencionar ventajas y desventajas.

Para configurar estos parámetros de forma correcta se debe considerar los 2 extremos del teorema de CAP: AP y CP.

Acerca de los valores R, N y W

- Valores altos de R, N y W significa una mayor consistencia ya que un mayor número de nodos son consultados para verificar que los datos existen y son correctos para operaciones de lectura y en escrituras, los datos son escritos en un mayor número de nodos.
- Por otro lado, valores altos pueden afectar el nivel de disponibilidad ya que Riak tendría que esperar por un mayor número de respuestas por parte de los nodos.

Acerca del valor N

- Debe ser mayor a 0 y menor o igual al número de nodos.

- Su valor no debería modificarse posterior a su creación. ¿ Qué pasa con los datos previamente almacenados si su valor se incrementa ? Podrían esperarse más copias existentes de las que actualmente existen.

Acerca del valor R

R indica el número de nodos que deben regresar un resultado para una cierta lectura y ser considerada como exitosa. Esta característica ofrece alta disponibilidad para lecturas.

- Si su valor es bajo, se obtiene una respuesta rápida, pero al ser bajo, puede aumentar la probabilidad de que el objeto no se encuentre en el primer nodo consultado. Por ejemplo, pudiera darse el caso de que el primer nodo aún no ha recibido la copia o réplica del objeto. En este caso se obtendrá `not found`, incluso, el objeto pudiera existir en otros nodos que no respondieron en primer lugar.
- Si su valor es alto, Riak puede tardar más tiempo en encontrarlo.

Acerca del valor W

- Indica el número de nodos deben completar una escritura para ser considerada como exitosa. Ventajas y desventajas se aplican de forma similar a los del parámetro R.

Acerca de los valores PR y PW

- En el modelo de replicación de Riak existen N vnodes (explicado más adelante) llamados nodos primarios. RIak intenta utilizarlos en primer lugar. De existir una falla, Riak intentará con nodos auxiliares configurados con una opción llamada ***sloppy quorum*** (Quorum relajado)
- Los parámetros PR y PW indican cuántos nodos primarios deben responder para considerar a una petición como exitosa.
- Cuando su valor es mayor a 0, se habilita un modo llamado ***strict quorum*** (Quorum estricto). Este modo tiene la ventaja que el cliente recibirá datos muy actualizados (últimas versiones), pero puede poner en riesgo la disponibilidad si estos nodos primarios no están disponibles.