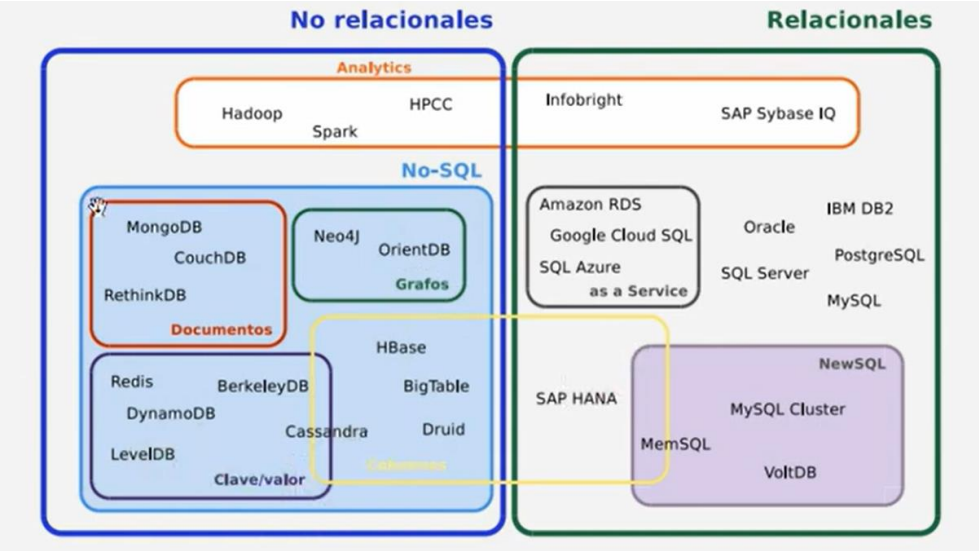
BASES DE DATOS NO SQL



Para BDD No-SQL de tipo documentos, clave-valor y columnas (wide columns) comparten la definición del ***agregado:*** es un conjunto de objetos relacionados entre si que se agrupan en una especie de unidad, y se guardan en un mismo lugar. Por ejemplo: el conjunto de datos personales de un cliente de una empresa, un post de Facebook con todos sus comentarios, etc.

SGBD distribuido

Un *sistema de gestión de bases de datos distribuido* es aquel que corre como sistema distribuido en distintas computadoras (nodos) que se encuentran sobre una red (ya sea local, a través de Internet, o como un servicio de *cloud computing*), aprovechando las ventajas de la computación distribuida y brindando a las aplicaciones la abstracción de ser un único sistema coherente.

Conceptos sobre SGBD's distributidos

1. ***Fragmentación:*** Es la tarea de dividir un conjunto de agregados entre un conjunto de nodos.

   - ✓ Almacenar conjuntos muy grandes de datos que de lo contrario no podrían caber en un único nodo
   - ✓ Paralelizar el procesamiento, permitiendo que cada nodo ejecute una parte de las consultas para luego integrar los resultados.

Según la manera de fragmentar, podemos distinguir entre

Fragmentación ***horizontal:*** se reparten los agregados entre los nodos, tal que cada nodo almacena un subconjunto de agregados, pero el agregado se almacena en X nodo en su totalidad. En general se hashea la clave del agregado y eso me dice a que nodo mando ese agregado.

- ✓ Fragmentación ***vertical:*** distintos nodos guardan un subconjunto de atributos de cada agregado. Todos suelen compartir los atributos que conforman la clave. Ejemplo vertical: tengo columnas que no son muy utilizadas y las mando a un nodo con menor capacidad de procesamiento.

2. ***Replicación:*** proceso por el cual se almacenan múltiples copias de un mismo dato en distintos nodos del sistema.

A veces me conviene tener un dato en mas de un lado. Por ejemplo, cuando se cae un nodo, o cuando un nodo esta muy ocupado porque lo matan a consultas.

Cuando una replica es ***solo para backup,*** se las suele denominar ***replica secundaria.*** Cuando también se las puede usar para realizar consultas y hacer procesamiento, se las conoce como ***replicas primarias.***

La replicación nos genera el problema de la **CONSISTENCIA de los datos.** Puede suceder que distintas replicas almacenen (al menos temporalmente) distintos valores para un mismo dato.

3. ***Búsqueda*** (lookup). Búsqueda de un dato, por ejemplo en una bdd clave valor, según su clave. Para ello, hay distintos métodos, como tablas de hash distribuidas
4. ***Modelos de consistencia:*** Qué pasa si no todos los nodos ven el mismo dato a la vez. A veces podemos aceptar que no todos vean el mismo dato a la vez.
    - ✓ Consistencia secuencial
    - ✓ Consistencia casual
    - ✓ Consistencia eventual
5. ***Método de acceso:*** una vez que ya estamos en un nodo particular, como accedemos al dato en sí. Muchas veces se usa B-trees o como alternativa los arboles LSM

Bases de datos clave-valor

Es una especie de Hashmap, pero pasado a bases de datos. SOLO guardan elementos de tipo clave-valor. Su objetivo es guardar y consultar grandes cantidades de datos, pero no de interrelaciones entre los datos

Las claves deben ser ***UNICAS,*** no puedo tener dos pares con la misma clave.

Operaciones elementales

- ✓ ***Put:*** insertar un nuevo par
- ✓ ***Delete:*** eliminar un par existente
- ✓ ***Update:*** actualizar el valor de un par
- ✓ ***Get:*** encontrar un par asociado a una clave particular

Sus ventajas son:

Simplicidad

- ✓ No se define un esquema
- ✓ No hay restricciones de integridad ni dominio
- ✓ El agregado es mínimo y está limitado al par (yo siempre trabajo con un par clave valor)

Velocidad

- ✓ Se prioriza la eficiencia de acceso antes que la integridad de los datos

Escalabilidad

- ✓ Proveen replicación en general y permiten repartir las consultas entre los nodos

Dynamo

Un ejemplo de BDD clave-valor es Dynamo, de Amazon, que utiliza un método lookup llamado hashing consistente.

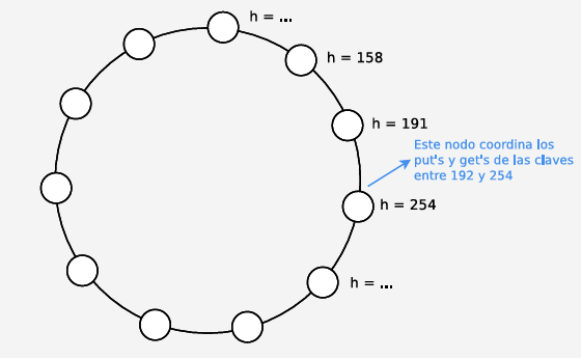
Hashing consistente

Dada una función de hash h( ), y una clave, k, h(k) devuelve un valor, para un par dado que **determina en cuál de los S nodos se va a almacenar el par (k,v).** esto se conoce como una DHT (distributed hash table)

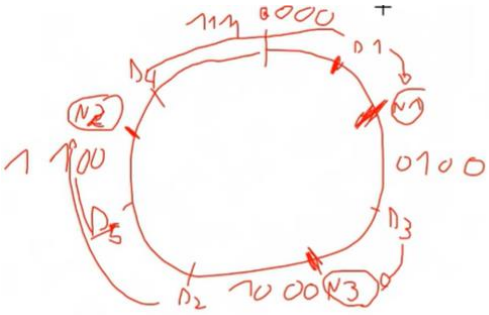
En algunas DHT's, el nodo asignado se determina como h(k) mod S. El problema con esto es que cada vez que agrego un nodo, tengo que rehashear los elementos que tenía guardados, porque si se cambia el S, la gran mayoría de los valores se tienen que modificar.

En Dynamo se usa hashing consistente. Es otra técnica de hashing, que busca que agregar/quitar nodos, tenga un impacto bastante menor.

- Al id de cada nodo (que en general suele ser su IP), se le aplica la misma función de hash
- Con el resultado, se organizan los nodos en una estructura de anillo por hash creciente, y asigno cada nodo a un valor de hash h.
  Por ejemplo, el nodo N1 me da un valor 180, entonces lo asigno al valor de hash h=191

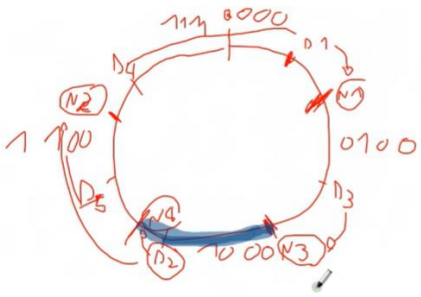


- Luego, hasheo los datos con su clave. Cada dato se va a guardar en el siguiente nodo siguiendo las agujas del reloj (datos = D_i):



La ventaja de esto, es que si yo agrego un nuevo nodo, no tengo que cambiar todos los datos.

Por ejemplo, si agrego un Nodo4 entre los nodos N2 y N3, el único que debería reubicar es el D2:



Ahora… ¿Qué pasa si se cae un nodo? Lo que se sule hacer es no solo guardar los datos en el nodo que le corresponde (el siguiente en sentido horario), sino guardarlo como replica en los N siguientes nodos:

Replicación: un par (k, v) se replica en los N servidores siguientes a h(k)

Consistencia secuencial

Lo que buscamos es que se vea un mismo valor en todos los nodos al mismo tiempo. En el caso clave-valor, es que para una sola clave halla un único valor. Esto no es fácil de lograr.

Se flexibiliza el concepto de que todos vean lo mismo (por ejemplo que desde un nodo se vea un poco mas desactualizado), utilizando un modelo de consisetencia denominado consistencia eventual, que tolera pequeñas inconsistencias en los valores almacenados en distintas réplicas.

- Partimos de una serie de procesos que ejecutan instrucciones de lectura, $R_{P_i}(X)$ y de escritura, $W_{P_i}(X)$, sobre una base de datos distribuida. → ¡Los procesos están en distintas máquinas!
- Se dice que una base de datos distribuida tiene consistencia secuencial cuando *"el resultado de cualquier ejecución concurrente de los procesos es equivalente al de alguna ejecución secuencial en que las instrucciones de los procesos se ejecutan una después de otra"*. → Queremos ver si el conjunto de órdenes locales de las instrucciones se corresponde con algún hipotético orden global.
  - Atención! Esto no quiere decir que los procesos se ejecuten uno después de otro, sino que una instrucción no comienza hasta que otra no haya terminado de aplicarse en todas las réplicas.

Nosotros vamos a tener que un proceso hizo varias lecturas/escrituras, otro proceso hizo varias lecturas/escrituras y así… y cada uno va a tener un orden local. Lo que queremos ver es de lograr que siempre, esos ordenes locales se correspondan con algún hipotético orden global. Es decir, que si agarramos de cada computadora, qué operaciones se fueron dando, si es posible que haya un orden global en el cual cada una de esas operaciones son coherentes.

- Para indicar el valor leído/escrito utilizaremos esta notación:
  - $R(X)a$ indica que el proceso leyó el valor a del ítem X.
  - $W(X)b$ indica que el proceso escribió el valor b en el ítem X.
- Ejemplo:
  - El valor inicial de a es 30, y el de b es 12.

P_1	R(b)12	R(a)3	W(a)20	
P_2	R(a)30	R(b)12	W(a)3	W(b)8
P_3	R(a)20			

Tiempo local →

Nota: Ninguna columna no implica mismo tiempo físico. ¡Los procesadores pueden no tener sus relojes sincronizados!

  - Esta ejecución es equivalente a la siguiente ejecución secuencial de las instrucciones:

P_1				R(b)12	R(a)3		W(a)20	
P_2	R(a)30	R(b)12	W(a)3			W(b)8		
P_3								R(a)20

Tiempo universal →

En este ejemplo tenemos 3 procesos. Primero tenemos los tiempos locales, luego intentamos armar un proceso local. Vamos entrelazando los procesos y viendo que puede pasar después de que, acorde a lo que paso en cada proceso individual.

- Pero en la siguiente ejecución, en cambio:

P_1	R(b)8	R(a)30	W(a)20	
P_2	R(a)30	R(b)12	W(a)3	W(b)8
P_3	R(a)20			

Tiempo local →

- No existe una ejecución secuencial equivalente. Evidentemente, P_1 está leyendo un valor de a desactualizado.
- Esta ejecución no tiene consistencia secuencial.

Garantizar consistencia secuencial es muy costoso, ya que requiere de mecanismos de sincroizacion fuertes que aumentan los tiempos de respuesta.

Es totalmente descentralizado. Los nodos son peers entre sí (Carece de un punto único de falla).

Consistencia causal

En este modelo se busca capturar eventos que pueden estar causalmente relacionados.

- Ejemplo: Supongamos que un proceso P_1 escribe un ítem X. Simultáneamente, un proceso P_2 lee el ítem X y escribe el ítem Y. Las dos escrituras están causalmente relacionadas, porque operan sobre el mismo ítem. Entonces, el modelo requiere que todos las vean en el mismo orden.

Solo garantizamos el orden cuando están relacionados entre si. Dos eveots no causalmente correlacionades se denominan concurrentes, y no es necesario que sean vistos por todos en el mismo orden.

- Analicemos la siguiente situación en que los valores iniciales son $a = 5$, $b = 7$ y $c = 9$:

P_1	W(a)5	R(c)20	R(b)7
P_2	W(b)10	R(b)10	R(c)9
P_3	W(c)20		

Tiempo local →

- Aquí las operaciones $W(b)10$ y $W(c)20$ son concurrentes, y por lo tanto es consistente que sean vistas por P_1 y P_2 en órdenes distintos.
- La ejecución tiene entonces consistencia causal. Sin embargo, no tiene consistencia secuencial.

- Veamos en cambio este ejemplo con valores iniciales $a = 5$ y $b = 7$:

P_1	W(a)20		
P_2		R(a)20	W(b)3
P_3	R(b)3	R(a)5	

Tiempo local →

- Ahora las operaciones $W(a)20$ y $W(b)3$ tienen relación causal, y deben ser vistas por todos en el mismo orden.
- Sin embargo, P_3 las ve en el orden $W(b)3 \rightarrow W(a)20$ mientras que P_2 las ve en el orden $W(a)20 \rightarrow W(b)3$.
- Esta ejecución no tiene consistencia causal.

Consistencia eventual

Una ejecución tiene consistencia eventual cuando “si en el sistema no se producen modificaciones (escrituras) por un tiempo suficientemente grande, entonces eventualmente todos los procesos verán los mismos valores”.

Dynamo provee un modelo de consistencia eventual, que permite que las actualizaciones se propaguen a las réplicas de forma asincrónica. Gracias a ésto, las lecturas y escrituras pueden devolver el control rápidamente. Cuando un nodo recibe un put sobre una clave, no necesita propagarlo a las N −1 réplicas antes de confirmar la escritura. Dado que las operaciones de get pueden realizarse sobre cualquier réplica, es posible leer un valor no actualizado.

Se definen dos parámetros adicionales:

- W <= N: Quorum de escritura

- Un nodo puede devolver un resultado de escritura exitosa luego de recibir la confirmación de escritura de otros $W - 1$ nodos del listado de preferencia.
- $W = 2$ ofrece un nivel de replicación mínimo.

Me dice que la escritura esta bien cuando el que mando el put grabo ese valor, y un nodo mas también.

- R <= N: Quorum de lectura

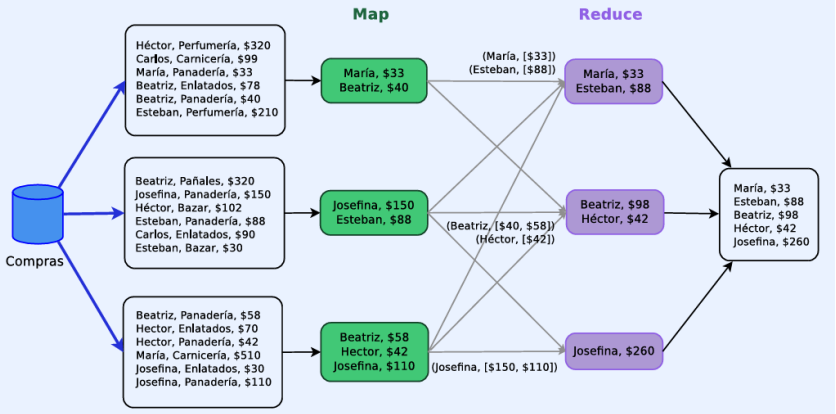
- Un nodo puede devolver el valor de una clave leída luego de disponer de la lectura de R nodos distintos (incluído él mismo).
- En muchas situaciones $R = 1$ es ya suficiente.
- Valores mayores de R brindan tolerancia a fallas como corrupción de datos ó ataques externos, pero hacen más lenta la lectura.

Esto lo mismo. Ya cuando R nodos lo leen esta Ok. Suele alcanzar con un solo nodo.

Modelo Map-Reduce

- Un pipeline MapReduce está formado por una secuencia de etapas de los siguientes tipos:
  - Map: Recibe un par clave/valor, (k, v), y devuelve un conjunto de pares clave/valor, $[(k_1, v_1), ...]$.
  - Reduce: Recibe un par clave/[valores], $(k, [v_1, ...])$, (es decir, una clave y un conjunto de valores asociados a esa clave) y devuelve un conjunto de valores $[v_1, ...]$ (típicamente es un único valor).

- En el siguiente ejemplo, queremos calcular cuál fue el gasto mensual de cada cliente en el rubro “Panadería”.



Teorema CAP

- En 1998 el científico E. Brewer postuló la imposibilidad de que un sistema de bases de datos distribuido garantice simultáneamente el máximo nivel de:
  - (C) Consistencia (*consistency*)
  - (A) Disponibilidad (*availability*)
  - (P) Tolerancia a particiones (*partition tolerance*)
- En 2002 Seth Gilbert y Nancy Lynch presentaron una prueba formal del resultado.

- Consistencia: propiedad de que en un instante determinado el sistema muestre un único valor de cada ítem de datos a los usuarios.
- Disponibilidad: consiste en que toda consulta que llega a un nodo del sistema distribuido que no está caído reciba una respuesta efectiva, sin errores

- Tolerancia: el sistema pueda responder una consulta aun cuando algunas conexiones entre algunos pares de nodos estén caídas

Lo que dice el teorema es que es imposible que un sistema distribuido garantice el máximo nivel de estas tres cosas.

- El Teorema CAP dice entonces que a lo sumo podremos ofrecer 2 de las 3 garantías:
  - AP: Si la red está particionada, podemos optar por seguir respondiendo consultas aún cuando algunos nodos no respondan. Garantizaremos disponibilidad, pero el nivel de consistencia no será el máximo.
  - CP: Con la red particionada, si queremos garantizar consistencia máxima no podremos garantizar disponibilidad. Es posible que no podamos responder una consulta en forma efectiva porque esperamos mensajes de confirmación desde nodos que no pueden comunicarse.
  - CA: Si queremos consistencia y disponibilidad, entonces no podremos tolerar que una cantidad indeterminada de enlaces se caiga.

Bases de datos orientadas a documentos

En vez de filas con datos vamos a tener un documento que almacene con cierta estructura datos del tipo clave-valor.

La estructura de un documento típicamente se describe con un lenguaje de intercambio de datos (data Exchange language). Por ejemplo: XML, HTML; JSON, YAML, etc.

La BDD orientada a documentos con la que vamos a trabajar es MongoDB. Es una BDD que utiliza JSON como formato para los documentos. Almacena por clave/valor.

```
Hamburguesa = {
    "nombre": "BigBacon",
    "ingredientes": ["pan", "carne", "lechuga", "salsa", "pan"
        "carne", "tocino", "queso", "pepinillos", "salsa", "pan"
            ],
    "precio": 129.99,
    "calorías": 930
  }
```

Los documentos que vamos a almacenar en la base de datos se van a identificar con un hash (clave) que denominamos **_id.**

Si no lo indicamos, MongoDB asignará como _id un *hash de 12 bytes.* La función ***ObjectId(h)*** convierte un hash en una referencia al documento que dicho hash identifica. El hash también asegura que no se puede insertar dos veces el mismo documento en una colección.

Cuando hacemos insert_one() o insert() obtenemos una lista de los hashes de los documentos creados

Ejemplo: Creación de documento

```
from pymongo import MongoClient
conn = MongoClient()

conn.database_names()

# Creamos una nueva base de datos
bd_empresa = conn.base_empresa

# Le agregamos una colección
col_clientes = bd_empresa.clientes

# Y le agregamos un documento a la colección
cliente1 = {
    "nombre": "Mario",
    "apellido": "Wilkerson",
    "domicilio": "Av. Entre Ríos 1560" }
id_cliente1 = col_clientes.insert_one(cliente1).inserted_id
```

Consultas

- Las consultas se realizan con la función *find*() sobre la colección.
- El resultado es un cursor que debe ser iterado.

```
#Buscamos todos los clientes que son de Morón
respuesta_query = col_clientes.find({"localidad": "Morón"})

for c in respuesta_query:
    pprint.pprint(c)
```

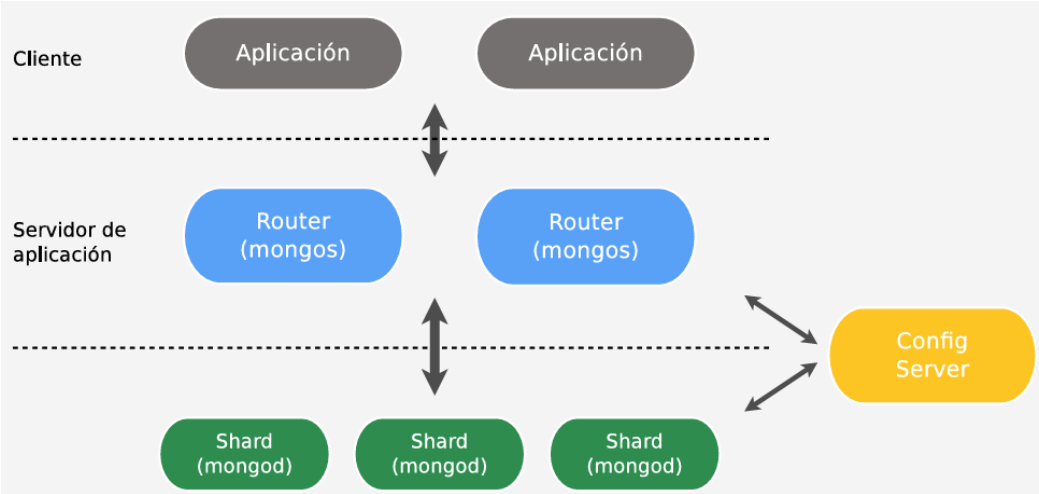
```
1  {'_id': ObjectId('592086269757902143 70fc98'),
2   'apellido': 'Fonseca',
3   'localidad': 'Morón',
4   'nombre': 'Horacio'}
5  {'_id': ObjectId('592086269757902143 70fc9a'),
6   'apellido': 'Findo',
7   'localidad': 'Morón',
8   'nombre': 'Diego'}
```

- MongoDB implementa la agregación a través de un *pipeline* secuencial que combina etapas de agrupamiento, selección, etc.
- La función *aggregate()* opera a partir de un vector de documentos JSON, en donde cada documento describe una operación (p. ej., *group* ó *match*) del *pipeline*.

```
#Calculamos la cantidad de clientes que viven en cada localidad,
#y mostramos sólo aquellas en que vive a lo sumo un cliente:
result = col_clientes.aggregate( [
    { "$group": {"_id": "$localidad", "cantidad": { "$sum": 1 } } },
    { "$match": {"cantidad": { "$lte":  1 } } } ])
for cliente in result:
    pprint.pprint(cliente)
```

MongoDB utiliza un modelo distribuido de procesamiento conocido como ***sharding.***

- Se basa en el particionamiento horizontal de las colecciones en chunks que se distribuyen en nodos denominados shards. Cada shard contendrá un subconjunto de los documentos de cada colección.
- Un sharding cluster de MongoDB está formado por distintos tipos de nodos de ejecución:
  - ✓ Los shards (fragmentos): Son los nodos en los que se distribuyen los chunks de las colecciones. Cada shard corre un proceso denominado mongod.
  - ✓ Los routers: Son los nodos servidores que reciben las consultas desde las aplicaciones clientes, y las resuelven comunicándose con los shards. Corren un proceso denominado mongos.
  - ✓ Los servidores de configuración: Son los que almacenan la configuración de los routers y los shards.



El particionado de las colecciones se realiza a partir de una ***shard key.***

- ✓ La shard key es un atributo o conjunto de atributos de la colección que se escoge al momento de construir el sharded cluster.
- ✓ La asignación de documentos a shards se hace dividiendo en rangos los valores de la shard key (range-based sharding), o bien a partir de una función de hash aplicada sobre su valor (hashed sharding).

Ejemplo: Acá hago el sharding de una colección por localidad.

```
sh.shardCollection("db_empresa.col_clientes",
                   {"localidad": "hashed"}, unique = False)
```

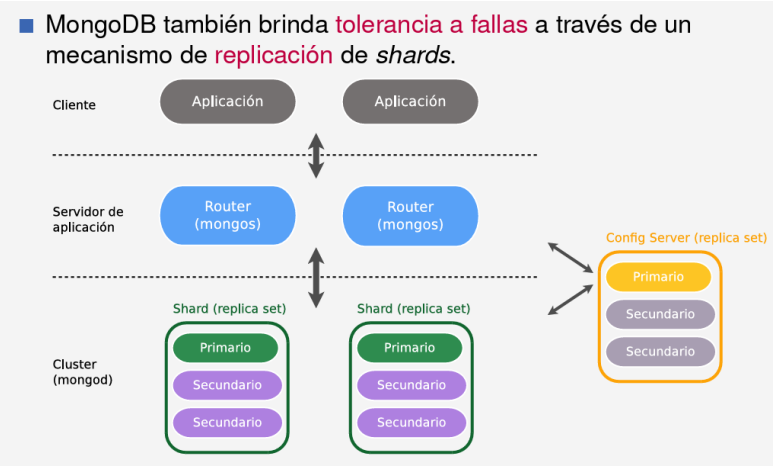
Importante: MongoDB no permite des-shardear una colección. Una vez que la distribuimos no podemos volver a ponerla en un solo nodo.

Lo que si es posible es tener algunas colecciones shared (fragmentadas) y otras unshared (no fragmentadas). Las colecciones unsharded de una base de datos se almacenarán en un shard particular del cluster, que será el shard primario para esa base de datos.

Para qué sirve el sharding

- Disminuir el tiempo de respuesta en un sistema con alta carga de consulta, al distribuir el trabajo de procesamiento entre varios nodos
- Ejecutar consultas sobre conjuntos de datos muy grandes que no podrían caber en un único servidor

Replicación de Shards



El esquema de réplicas es de master-slave with automated failover (maestro-esclavo con recuperación automática):

- ✓ Cada shard pasa a tener un servidor mongod primario (master), y uno o más servidores mongod secundarios (slaves). El conjunto de réplicas de un shard se denomina replica set.
- ✓ Las réplicas eligen inicialmente un master a través de un algoritmo distribuido.
- ✓ Cuando el master falla, los slaves eligen entre sí a un nuevo master.

También los servidores de configuración se implementan como replica sets.

Todas las operaciones de escritura sobre el shard se realizan en el master. Los slaves sólo sirven de respaldo.

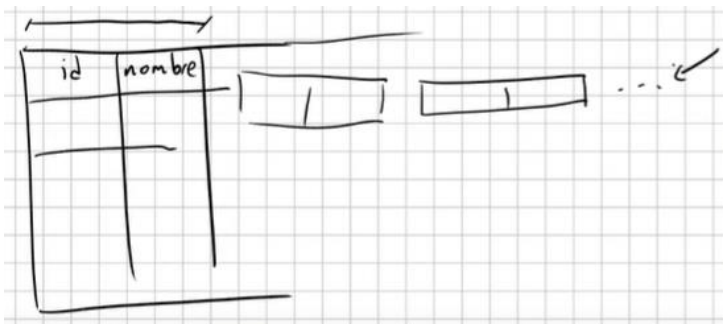
Los clientes pueden especificar una read preference para que las lecturas sean enviadas a nodos secundarios de los shards.

Bases de datos Wide Column

La idea es una tabla que normalmente se diseñaría en dos partes con un join, la vamos a guardar toda junta y vamos a guardar cerca los datos que joinearíamos muy seguido.

Son una evolución de las bases de datos clave/valor, ya que agrupan los pares vinculados a una misma entidad como columnas asociadas a una misma clave primaria.

En nuestras filas de la BDD vamos a tener ciertas columnas que son fijas y después vamos a tener algunas columnas que se pueden repetir varias veces, que van a ser algún tipo de dato que va a estar relacionado con los 1ros fijos que están en las 1ras columnas, pero por alguna razón los quiero tener en un formato de tipo lista a la que voy agregando cosas… como un hash abierto:



Un valor particular de la clave primaria junto con todas sus columnas asociadas forma un agregado análogo a la fila de una tabla. Pero además, estas bases permiten agregar conjuntos de columnas en forma dinámica a una fila, convirtiéndola en un agregado llamado fila ancha (wide row, aunque se llama wide columna historicamente)

Ejemplo: yo tengo una biblioteca y voy a estar prestando libros… yo quiero ver que clientes se llevaron X libro con tal id.

Nosotros vamos a trabajar con la BDD NoSQL de tipo wide columna llamada **Cassandra**

Cassandra

Key spaces y column families

SGBD's relacionales	Cassandra
Esquema	*Keyspace*
Tabla	*Column family*
Fila	Fila
-	*Wide row*

Sinónimos en SGBD con Cassandra. Acá el concepto análogo de la tabla es el de ***columna family.***

Una ***fila*** está formada por

- ✓ Una clave compuesta (un atributo o un conjunto de atributos)
- ✓ Un conjunto de pares clave-valor o columnas.

Para Cassandra, cada columna no es más que un par clave-valor asociado a una fila.

Cada keyspace (esquema) puede estar distribuido en varios nodos de nuestro cluster

- Creamos un nuevo *key space* con la siguiente instrucción en CQL:

```
CREATE KEYSPACE empresa_db
WITH replication = {
  'class': 'SimpleStrategy',
  'replication_factor': 1 };
```

- Asignamos este *key space* como aquél por defecto:

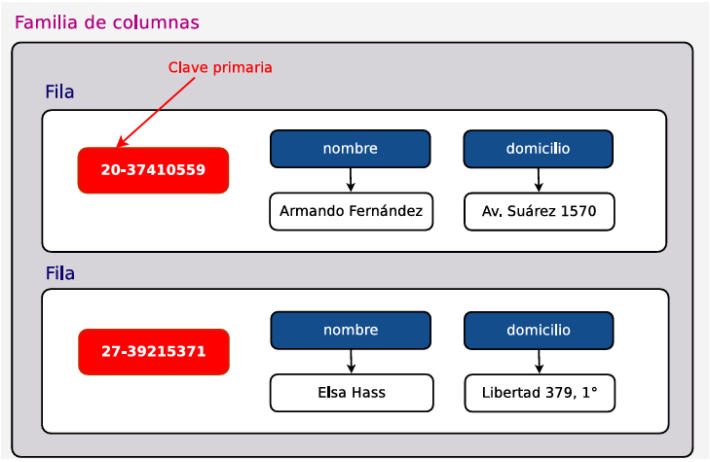
```
USE empresa_db;
```

- Creación de un *column family*:

```
CREATE COLUMNFAMILY clientes (
  cuit int,
  nombre text,
  domicilio text,
  primary key (cuit));
```

  - ¡Es obligatorio definir una clave primaria!

Esquema lógico de una fila

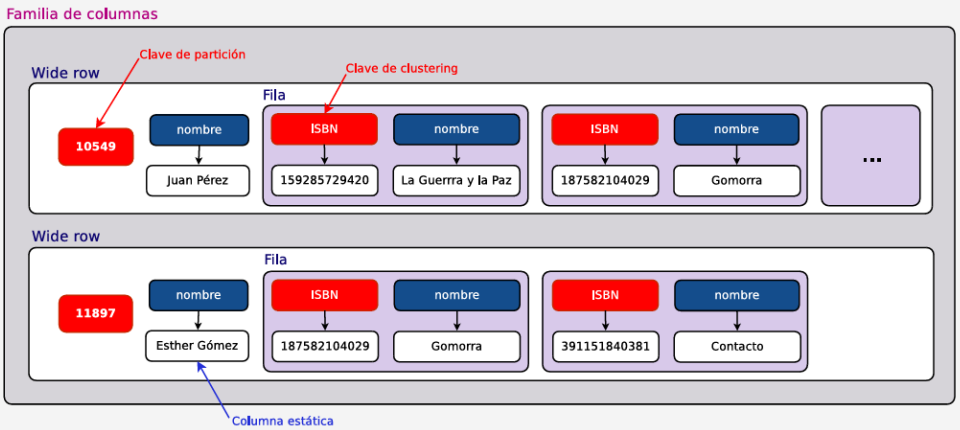


Esquema lógico de una wide row

Vamos a tener partes de nuestras columnas que se van a poder repetir más de una vez:

Ejemplo: Si un cliente nos compra Libros, quisiéramos agregar por cada libro comprado el ISBN y el nombre. Pero ¿cómo es esto posible? Durante la creación del column-family tenemos que definir en el esquema cuáles van a ser las columnas.

En realidad, lo que definimos es un listado de columnas, que cada fila puede instanciar muchas veces. Esto se resuelve seleccionando a una o más de las columnas como parte de la clave. Cuando en una fila las columnas se repiten identificadas por el valor que toman las columnas clave, se dice que la fila se convirtió en una wide row (fila ancha).



- Nuestra clave primaria queda ahora dividida en dos partes: una clave de partición y una clave de clustering.
- Adicionalmente, podemos tener columnas estáticas, que sean únicas por cada *partition key* (por ejemplo, el nombre del Cliente).

- La definición de esta *column family* es:

```
CREATE COLUMNFAMILY clientes (
  nro_cliente int,
  nombre text static,
  ISBN bigint,
  nombre_libro text,
  primary key ((nro_cliente), ISBN));
```

La clave primaria en Cassandra se divide en dos partes:

- ✓ La clave de particionado (partition key): por si sóla debe alcanzar para identificar a la wide-row (en el ejemplo anterior, al Cliente). Vamos a pedir que sea **minimal.**

La clave de particionado determina el/los nodo/s del cluster en que se guardará la wide-row (se utiliza hashing consistente para el lookup).

Toda la wide-row se almacenará contigua en disco, y la clave de clustering nos determina el ordenamiento interno de las columnas dentro de ella.

- ✓ La clave de clustering (clustering key): pediremos que permita identificar a la fila. No puede haber dos filas de una column family con igual valor en la clave primaria.

Restricciones de la clave primaria

El diseño físico de los datos en Cassandra impone algunas restricciones sobre la elección de la clave primaria de cada column family. Estas se usan para filtrar las tablas.

1. Las columnas que forman parte de la partition key **deben** ser comparadas por igual contra valores constantes en los predicados.

2. Si una columna que forma parte de la clustering key es utilizada en un predicado, también deben ser utilizadas todas las restantes columnas que son parte de la clustering key y que preceden a dicha columna en la definición de la clave primaria.

3. En particular, si una columna que forma parte de la clustering key es comparada por rango en un predicado, entonces todas las columnas de la (clustering key) que la preceden deben ser comparadas por igual, y las posteriores no deben ser utilizadas.

Tipos de dato

- Los tipos de dato básicos son similares a los de SQL:
  - **int**, **smallint**, **bigint**, **float**, **decimal**
  - **text**, **varchar**
  - **timestamp**, **date**, **time**
  - **uuid** (clave surrogada)
  - **boolean**
- Adicionalmente, Cassandra permite trabajar con colecciones como tipos de dato:
  - **set** (conjunto de elementos)
  - **list** (lista ordenada de elementos)
  - **map** (conjunto de pares clave/valor)

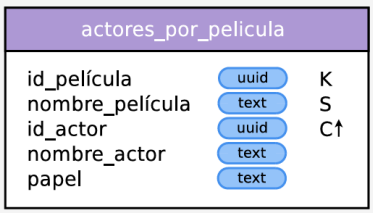
Reglas de diseño

Para diseñar una base de datos en Cassandra debemos tener en cuenta los siguientes puntos.

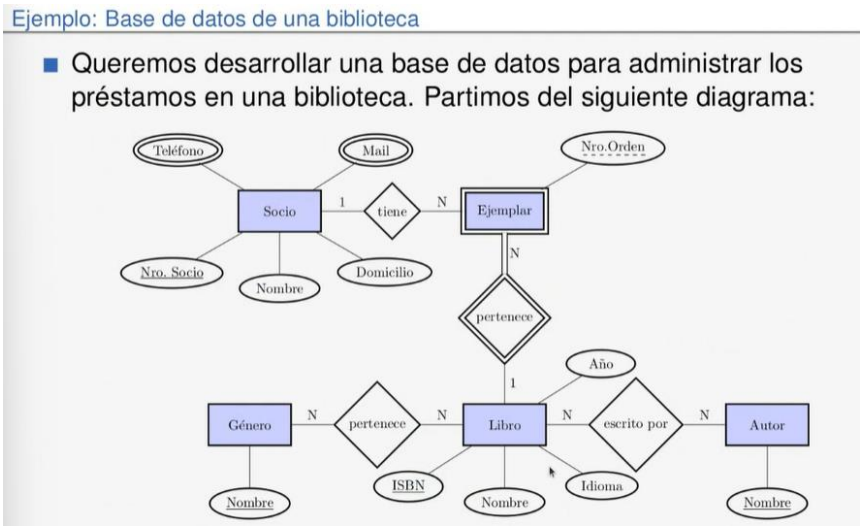
1. No existe el concepto de junta. Si para alguna consulta típica necesitamos el resultado de una junta, entonces debemos guardarla como una tabla desnormalizada más desde el comienzo.
2. No existe el concepto de integridad referencial. Si la necesitamos, debe ser manejada desde el nivel de aplicación.
3. Desnormalización de datos. En las bases de datos NoSQL el uso de tablas no normalizadas está a la orden del día, y básicamente por un único motivo: performance.
4. Diseño orientado a las consultas. ¿Cómo saber qué tablas crear si no sabemos cuáles son las consultas a hacer?
   - ✓ En los SGBD's relacionales ésto no es un problema. El modelo lógico contiene toda la información, que luego extraeremos haciendo juntas.
   - ✓ Aquí, en cambio las consultas preceden al modelo de datos: debemos pensar de antemano qué consultas haremos para poder diseñar las tablas.

Objetivos de diseño: Diagramas Chebotko

- Buscamos que:
  - Cada consulta se resuelva accediendo a una única *column family*.
  - Los resultados de una consulta estén en una única partición.
  - Se respeten las reglas del lenguaje CQL respecto al uso de la clave primaria.
- Se han propuesto distintos diagramas para el modelado lógico en Cassandra.
- Nosotros utilizaremos uno de los más conocidos: Chebotko.

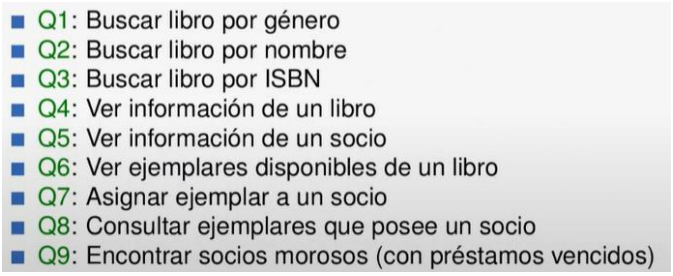


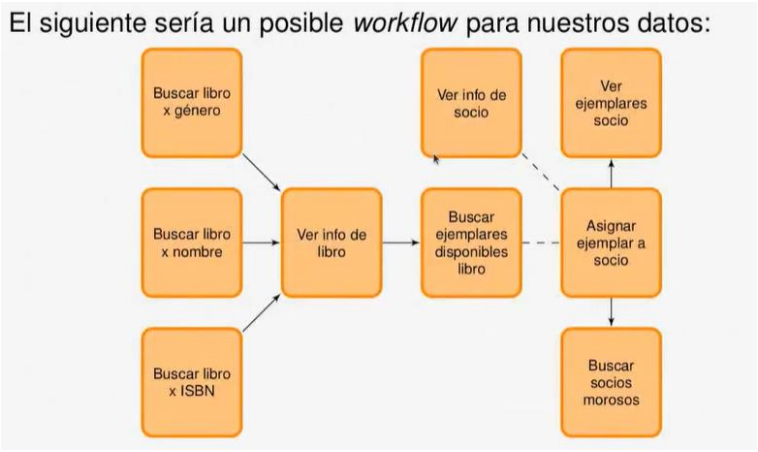
[Como modelamos una BDD de Cassandra]

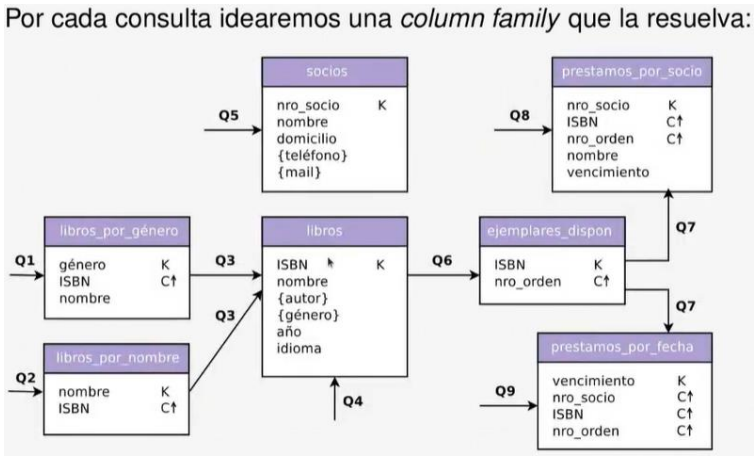


1. Definer el *workflow* de nuestra aplicación: Cuáles son nuestras necesidades y qué consultas queremos responder

Ejemplo: nos interesan responder las siguientes consultas:







Ejemplos de algunas consultas:

1. Dado un número de socio, encuentre los ejemplares que el socio posee, indicando el nombre del libro, el número de orden del ejemplar y la fecha de vencimiento del préstamo.

```
SELECT nombre, ISBN, nro_orden, vencimiento
FROM prestamos_por_socio
WHERE nro_socio = 751;
```

2. Dado el ISBN de un libro, encuentre los nombres de los autores del mismo.

```
SELECT autor
FROM libros
WHERE ISBN = 14292859338291;
```

3. Dado el ISBN y número de orden de un ejemplar disponible de un libro, y dado un número de socio, elimine el ejemplar del listado de ejemplares disponibles y asígnelo en préstamo al socio:

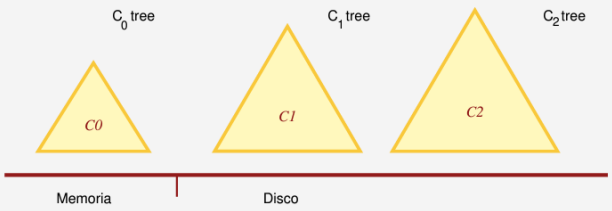
```
DELETE FROM prestamos_por_socio
WHERE nro_socio = 751 AND
      ISBN = 55102184963921 AND
      nro_orden = 2;

INSERT INTO ejemplares_dispon (ISBN, nro_orden)
VALUES (55102184963921, 2);
```

Cómo se acceden a los datos en Cassandra

Cassandra está optimizado para altas tasas de escritura. Utiliza una estructura de búsqueda denominada LSM-tree (log-structured merge tree), que mantiene parte de sus datos en memoria, para diferir los cambios sobre el índice en disco.

Se busca acceder en forma secuencial a disco, para mejorar el trade-off entre el costo de hacer un disk seek y el costo de un buffer en memoria. Esto ha sido bastante estudiado y se conoce como regla de los cinco minutos (Five-minute Rule).



- Consideraciones:
  - Desde que se inserta una entrada en C_0 hasta que se traslada a C_1 habrá una demora.
  - El costo de I/O de escritura en C_0 es nulo.
  - Cuando el tamaño de C_0 alcanza un umbral se inicia un proceso de *rolling merge (flush)*.
- El árbol C_1 suele tener una estructura similar a un B-tree.
- En cambio, como C_0 está en memoria no es relevante minimizar su profundidad → suelen emplearse árboles balanceados como el *2-3 tree* o el *árbol AVL*.

BDD Basadas en Grafos

En las bases de datos basadas en grafos los elementos principales son nodos y arcos (ejes).

Estas bases de datos resultan útiles para modelar interrelaciones complejas entre las entidades.

Las bases de datos relacionales modelan las relaciones entre entidades distintas utilizando claves foráneas.

- ✓ Ejemplo: Persona(dni, nombre, f_nac), HijoDe(dni, dni_padre)
- ✓ ¿Cómo hacemos si queremos encontrar a todos los descendientes de una persona?
- ✓ En bases de datos relacionales esta consulta tiene alto costo porque requiere de múltiples juntas.

Las bases orientadas a grafos utilizan una estructura en que cada nodo mantiene una referencia directa a sus nodos adyacentes.

Organizar nuestra base de datos de esta forma nos provee ventajas para resolver problemas clásicos de grafos como:

- ✓ Encontrar patrones de nodos conectados entre sí.
- ✓ Encontrar caminos entre nodos.
- ✓ Encontrar la ruta más corta entre dos nodos.
- ✓ Calcular medidas de centralidad asociadas a los nodos.

En general, es una buena idea utilizarlas cuando en nuestro modelo conceptual encontramos que las instancias de los tipos de entidades mantienen interrelaciones con otras instancias de su mismo tipo de entidad.

Vamos a usar Neo4j

- Una base de datos Neo4j está formada por nodos.
- Un nodo puede tener distintos labels. Dentro de cada *label*, el nodo tendrá un conjunto de propiedades con determinados valores.

```
(tom:Persona {nombre: 'Tomás', color: 'Azul', prof: 'Estudiante'})
```

- No existe una estructura rígida respecto a qué propiedades deben tener los nodos con determinado *label*.

```
(edith:Persona {nombre: 'Tomás', color: 'Verde', prof: 'Músico'})
(maria:Persona {nombre: 'María', prof: 'Estudiante'})
(gaby:Persona {nombre: 'Gabriel', color: 'Verde', prof: 'Médico'})
```

- En Cypher, los nodos se crean con el comando **CREATE**:

```
CREATE (pepe:Persona {nombre: 'Pepe', color: 'Azul'})
```

- Para buscar un nodo ó conjunto de nodos utilizamos el comando **MATCH**:

```
1 MATCH (p:Persona {nombre: 'María'}) RETURN p.nombre, p.prof
```

- El resultado de la consulta es un conjunto de records, que podemos representar con una tabla:

nombre	profesión
María	Estudiante

- También se pueden aplicar condiciones de selección sobre las búsquedas con el comando **WHERE**:

```
1 MATCH (m:Persona) WHERE m.color='Verde' RETURN m.nombre, m.prof
```

nombre	profesión
Edith	Músico
Gabriel	Médico

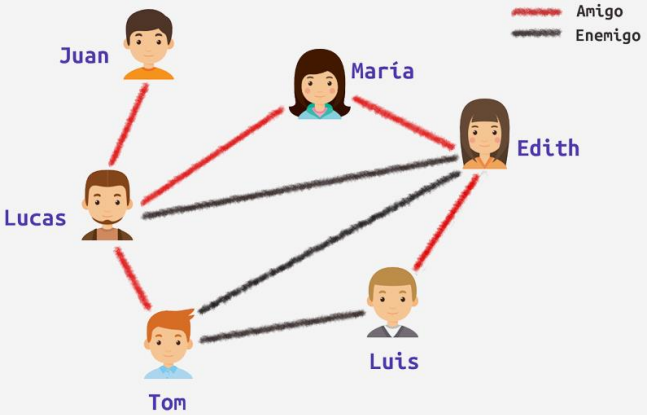
Direccionalidad

En Neo4j, los ejes son siempre direccionales. (Los grafos son dirigidos.)

Sin embargo, para trabajar con grafos no dirigidos no es necesario crear las interrelaciones en los dos sentidos.

Es posible indicar en la consulta Cypher si queremos prestar atención a la dirección de los ejes en la navegación, o no.

- ✓ ->:Requiere que se respete la dirección del eje.
- ✓ -:Pasa poralto la dirección del eje en la interrelación.



- Queremos descubrir, dentro de este grafo, si existen enemigos que tengan amigos en común.

¿Existen enemigos con amigos en común?

```
1 MATCH (n:Persona)-[:AMIGO_DE]-(m:Persona),
2       (m:Persona)-[:AMIGO_DE]-(o:Persona),
3       (n:Persona)-[:ENEMIGO_DE]-(o:Persona)
4 RETURN n.nombre, o.nombre
```

n.nombre	o.nombre
Lucas	Edith
Edith	Lucas

- Con un * en la interrelación podemos indicar una cantidad indeterminada de saltos.

¿A cuántos amigos de distancia están Juan y Luis?

```
1  MATCH (juan:Persona {nombre:'Juan'})
2        (luis:Persona {nombre:'Luis'})
3        p=(juan:Persona)-[:AMIGO_DE*]-(luis:Persona)
4  RETURN length(p)
5  ORDER BY length(p)
6  LIMIT 1
```

length(p)
4

- Lo que hicimos fue encontrar todos los *caminos*, ordenarlos por longitud y quedarnos con la longitud del menor.

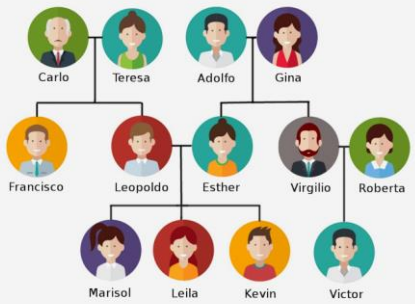
Esquemas de consultas en Cypher

- El esquema general de una consulta en Cypher es:

MATCH $p_1 = pattern_1, p_2 = pattern_2, ..., p_n = pattern_n$
WHERE $cond_1, cond_2, ..., cond_m$
RETURN $p_{i_1}, ..., agg(p_{j_1}), ...$
ORDER BY $p_{i_k}, agg(p_{j_{k'}}), ...$
LIMIT N;

- Un patrón *(pattern)* puede especificarse a través de un nodo y sus propiedades, una interrelación y sus propiedades, o un camino y sus propiedades. A cada patrón podemos darle un nombre.
- La cláusula **WHERE** dá aún más flexibilidad para filtrar los resultados basados en alguna condición.
- Las operaciones de agregación se realizan siempre en el **RETURN**, aplicando funciones como count(*), sum() ó max().

Ejemplo: Árbol Genealógico



- En este grafo tenemos interrelaciones de tipo *HIJO_DE* y *ESPOSO*.

¿Cómo se llaman los primos de Victor?

```
1  MATCH (victor:Persona {nombre:'Victor'}),
2    (primo:Persona)-[:HIJO_DE]->(t:Persona)-[:HIJO_DE]->
3    (a:Persona)<-[:HIJO_DE]-(p:Persona)<-[:HIJO_DE]-(victor)
4  WHERE p<>t
5  RETURN DISTINCT primo.nombre
```

nombre
Leila
Marisol
Kevin