Tema 7: Cassandra

Introducción

Apache Cassandra es una base de datos distribuida altamente escalable y de alto rendimiento diseñada para manejar grandes cantidades de datos en muchos servidores básicos, proporcionando alta disponibilidad sin un punto único de falla. Es un tipo de base de datos NoSQL.

> If you can’t split it, you can’t scale it.
>
> Randy Shoup, Distinguished Architect, eBay

Recordemos: NoSQL

Una base de datos NoSQL (a veces llamada Not Only SQL) es una base de datos que proporciona un mecanismo para almacenar y recuperar datos distintos de las relaciones tabulares utilizadas en las bases de datos relacionales.

Estas bases de datos tienen esquemas flexibles, admiten una fácil replicación, tienen una API simple, eventualmente son consistentes y pueden manejar grandes cantidades de datos.

Las bases de datos NoSql utilizan diferentes estructuras de datos en comparación con las bases de datos relacionales. Hace que algunas operaciones sean más rápidas en NoSQL. La idoneidad de una base de datos NoSQL dada depende del problema que debe resolver.

Una clasificación de modelos NOSQL

- Key-value stores
- Column stores
- Document stores
- Graph databases
- Object databases
- XML databases
- Multimodel databases

Una clasificación de modelos NOSQL

Key-value stores

En un almacén de clave-valor, los elementos de datos son claves que tienen un conjunto de atributos. Todos los datos relevantes para una clave se almacenan con la clave; los datos se duplican con frecuencia. Las tiendas clave-valor populares incluyen Dynamo DB, Riak y Voldemort de Amazon. Además, muchas tecnologías populares de almacenamiento en caché actúan como almacenes de valores clave, incluidos Oracle Coherence, Redis y Memcached.

- Column stores
- Document stores
- Graph databases
- Object databases
- XML databases
- Multimodel databases

Una clasificación de modelos NOSQL

Column stores

En un modelo de columnas, también conocido como modelo de columnas anchas o modelo orientado a columnas, los datos se almacenan por columna en lugar de por fila. Por ejemplo, en un almacén de columnas, todas las direcciones de los clientes pueden almacenarse juntas, lo que les permite recuperarlas en una sola consulta. Las modelos de columnas populares incluyen HBase de Apache Hadoop, Apache Kudu y Apache Druid.

- Document stores
- Graph databases
- Object databases
- XML databases
- Multimodel databases

Una clasificación de modelos NOSQL

Document stores

La unidad básica de almacenamiento en una base de datos de documentos es el documento completo, a menudo almacenado en un formato como JSON, XML o YAML. Las bases de datos de documentos populares incluyen MongoDB, CouchDB y varias ofertas de nube pública.

- Graph databases
- Object databases
- XML databases
- Multimodel databases

Una clasificación de modelos NOSQL

Graph databases

Las bases de datos de grafos representan los datos como un gráfico: una red de nodos y aristas que conectan los nodos. Tanto los nodos como las aristas pueden tener propiedades. Debido a que otorgan mayor importancia a las relaciones, las bases de datos de grafos como Neo4j, JanusGraph y DataStax Graph han demostrado ser populares para crear redes sociales y aplicaciones de web semántica.

- Object databases
- XML databases
- Multimodel databases

Una clasificación de modelos NOSQL

Object databases

Las bases de datos de objetos almacenan datos no en términos de relaciones, columnas y filas, sino en términos de objetos tal como se entienden desde la disciplina de la programación orientada a objetos. Esto facilita el uso de estas bases de datos desde aplicaciones orientadas a objetos. Las bases de datos de objetos como db4o e InterSystems Caché le permiten evitar técnicas como procedimientos almacenados y herramientas de mapeo relacional de objetos (ORM). La base de datos de objetos más utilizada es el Servicio de almacenamiento simple (S3) de Amazon Web Services.

- XML databases
- Multimodel databases

Una clasificación de modelos NOSQL

XML databases

Las bases de datos XML son una forma especial de bases de datos de documentos, optimizadas específicamente para trabajar con datos descritos en el lenguaje de marcado extensible (XML). Las bases de datos denominadas "XML nativas" incluyen BaseX y eXist.

- Multimodel databases

Una clasificación de modelos NOSQL

- Multimodel databases
  - Las bases de datos que admiten más de uno de estos estilos han ido ganando popularidad. Estas bases de datos "multimodelo" se basan en una base de datos principal subyacente (la mayoría de las veces, un almacén relacional, de clave-valor o de columnas) y exponen modelos adicionales como API además de esa base de datos subyacente. Ejemplos de estos incluyen Microsoft Azure Cosmos DB, que expone las API de documentos, columnas anchas y gráficos sobre un almacén de clave-valor, y DataStax Enterprise, que ofrece una API de gráficos sobre el modelo de columna ancha de Cassandra. Las bases de datos multimodelo a menudo se promocionan por su capacidad para admitir un enfoque conocido como persistencia políglota, en la que diferentes microservicios o componentes de una aplicación pueden interactuar con los datos utilizando más de uno de los modelos que hemos descrito aquí.

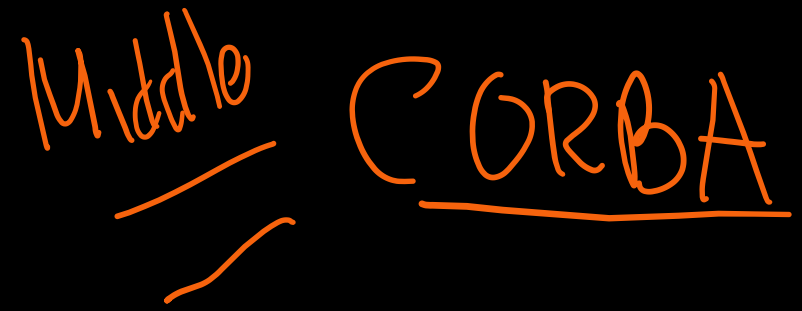
LEARNING MORE ABOUT NOSQL DATABASES

For a comprehensive list of NoSQL databases, see the NoSQL site. The DB-Engines site also provides popularity rankings of popular databases by type, updated monthly.

Apache Cassandra en 50 palabras o menos

- “Apache Cassandra es una base de datos de open source, distribuida, descentralizada, elásticamente escalable, altamente disponible, tolerante a fallas, sintonizable y orientada a filas. Cassandra basa su diseño de distribución en Dynamo de Amazon y su modelo de datos en Bigtable de Google, con un lenguaje de consulta similar a SQL.

Cassandra: Distributed



- Cassandra es distribuida , lo que significa que es capaz de ejecutarse en varias máquinas mientras se muestra a los usuarios como un todo unificado.
- De hecho, no tiene mucho sentido ejecutar un solo nodo de Cassandra. Aunque puede hacerlo, y eso es aceptable para ponerse al día sobre cómo funciona, rápidamente se da cuenta de que necesitará varias máquinas para obtener realmente algún beneficio al ejecutar Cassandra.
- Gran parte de su diseño y base de código está diseñado específicamente no solo para que funcione en muchas máquinas diferentes, sino también para optimizar el rendimiento en múltiples racks de centros de datos, e incluso para un solo clúster de Cassandra que se ejecuta en centros de datos dispersos geográficamente. Puede escribir datos con confianza en cualquier parte del clúster y Cassandra los obtendrá.

Cassandra: Decentralized

- El hecho de que Cassandra esté descentralizada significa que no hay un único punto de falla. Todos los nodos en un clúster de Cassandra funcionan exactamente igual. Esto a veces se denomina "simetría del servidor". Debido a que todos están haciendo lo mismo, por definición, no puede haber un host especial que coordine las actividades, como ocurre con la configuración primaria/secundaria que se ve en MySQL, Bigtable y muchas otras bases de datos.

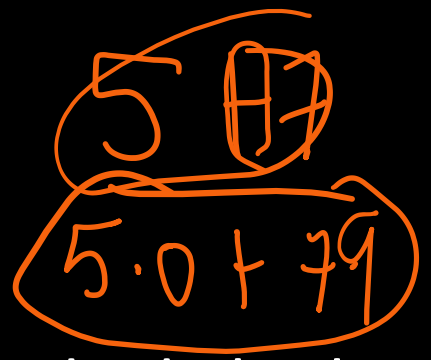
Cassandra: Elastic Scalability

- La escalabilidad elástica se refiere a una propiedad especial de la escalabilidad horizontal. Significa que su clúster puede escalar hacia arriba y hacia abajo sin problemas.
- Para hacer esto, el clúster debe poder aceptar nuevos nodos que puedan comenzar a participar al obtener una copia de algunos o todos los datos y comenzar a atender las solicitudes de nuevos usuarios sin una interrupción importante o una reconfiguración de todo el clúster. No tienes que reiniciar tu proceso.
- No tiene que cambiar las consultas de su aplicación. No tiene que reequilibrar manualmente los datos usted mismo. Simplemente agregue otra máquina: Cassandra la encontrará y comenzará a enviarle trabajo.

Cassandra: High Availability and Fault Tolerance

- En términos generales de arquitectura, la disponibilidad de un sistema se mide según su capacidad para cumplir con las solicitudes. Pero las computadoras pueden experimentar todo tipo de fallas, desde fallas en los componentes de hardware hasta la interrupción de la red y la corrupción.
- Cualquier computadora es susceptible a este tipo de fallas.
- Por lo tanto, para que un sistema tenga alta disponibilidad, normalmente debe incluir varias computadoras en red, y el software que ejecutan debe ser capaz de operar en un clúster y tener alguna facilidad para reconocer fallas de nodos y conmutar solicitudes a otra parte del sistema.

Cassandra: Tuneable Consistency



- La consistencia es un término sobrecargado en el mundo de las bases de datos, pero para nuestros propósitos usaremos la definición de que una lectura siempre devuelve el valor escrito más recientemente.
- Considere el caso de dos clientes que intentan colocar el mismo artículo en sus carritos de compras en un sitio de comercio electrónico.
- Si coloco el último artículo en stock en mi carrito un instante después de que usted lo haga, debe agregar el artículo a su carrito y se me debe informar que el artículo ya no está disponible para la compra.
- Se garantiza que esto suceda cuando el estado de una escritura sea consistente entre todos los nodos que tienen esos datos.

Un compromiso inevitable

- Escalar los almacenes de datos significa hacer ciertas concesiones entre la **consistencia de los datos, la disponibilidad de los nodos y la tolerancia a la partición**.
- A Cassandra se la llama con frecuencia "eventualmente consistente", aunque es más adecuado decir queCassandra intercambia algo de consistencia para lograr una disponibilidad total.
- Cassandra se denomina con mayor precisión "sintonizablemente coherente", lo que significa que le permite decidir fácilmente el nivel de coherencia que necesita, en equilibrio con el nivel de disponibilidad.

Una clasificación aceptable de consistencia

- Strict consistency
- Causal consistency
- Weak (eventual) consistency

Una clasificación aceptable de consistencia

- Strict consistency
  - Esto a veces se denomina consistencia secuencial y es el nivel más estricto de consistencia. Requiere que cualquier lectura siempre devuelva el valor escrito más recientemente.
  - ¿Más recientemente a quién? En una máquina de un solo procesador, esto no es un problema de observar, ya que la secuencia de operaciones es conocida por el reloj único. Pero en un sistema que se ejecuta en una variedad de centros de datos dispersos geográficamente, se vuelve mucho más resbaladizo.
  - Lograr esto implica algún tipo de reloj global que puede marcar con fecha y hora todas las operaciones, independientemente de la ubicación de los datos o del usuario que los solicita o de cuántos servicios (posiblemente dispares) se requieren para determinar la respuesta.
- Causal consistency
- Weak (eventual) consistency

Una clasificación aceptable de consistencia

- Causal consistency
  - Esta es una forma ligeramente más débil de consistencia estricta. Elimina la fantasía del reloj global único que puede sincronizar mágicamente todas las operaciones sin crear un cuello de botella insoportable.
  - En lugar de depender de las marcas de tiempo, la coherencia causal adopta un enfoque más semántico e intenta determinar la causa de los eventos para crear cierta coherencia en su orden.
  - Significa que las escrituras que están potencialmente relacionadas deben leerse en secuencia. Si dos operaciones diferentes no relacionadas escriben repentinamente en el mismo campo al mismo tiempo, entonces se infiere que esas escrituras no están causalmente relacionadas. Pero si una escritura ocurre después de otra, podríamos inferir que están causalmente relacionadas. La consistencia causal dicta que las escrituras causales deben leerse en secuencia.
- Weak (eventual) consistency

Una clasificación aceptable de consistencia

- Weak (eventual) consistency
  - La consistencia eventual significa en la superficie que todas las actualizaciones se propagarán a través de todas las réplicas en un sistema distribuido, pero que esto puede llevar algún tiempo.
  - Eventualmente, todas las réplicas serán consistentes.
  - La consistencia eventual se vuelve repentinamente muy atractiva cuando consideras lo que se requiere para lograr formas más fuertes de consistencia.

Puntos notables de Apache Cassandra

Es escalable, tolerante a fallas y consistente.

Es de tipo llave-valor así como una base de datos orientada a columnas.

Su diseño de distribución se basa en Dynamo de Amazon y su modelo de datos en Bigtable de Google.

Creado en Facebook, difiere marcadamente de los sistemas de gestión de bases de datos relacionales.

Cassandra implementa un modelo de replicación estilo Dynamo sin un único punto de falla, pero agrega un modelo de datos de "familia de columnas" más potente.

Cassandra está siendo utilizado por algunas de las empresas más importantes, como Facebook, Twitter, Cisco, Rackspace, ebay, Twitter, Netflix y más.

Resumiendo:
¿Qué es Apache Cassandra?

Apache Cassandra es un sistema de almacenamiento (base de datos) de código abierto, distribuido y descentralizado/distribuido, para administrar grandes cantidades de datos estructurados repartidos por todo el mundo. Proporciona un servicio de alta disponibilidad sin un único punto de falla.

Features importantes

- Escalabilidad elástica: Cassandra es altamente escalable; permite agregar más hardware para acomodar a más clientes y más datos según el requisito.
- ● Arquitectura siempre activa: Cassandra no tiene un único punto de falla y está continuamente disponible para aplicaciones críticas para el negocio que no pueden permitirse una falla.

Features importantes

- Rápido rendimiento de escala lineal: Cassandra es linealmente escalable, es decir, aumenta su rendimiento a medida que aumenta la cantidad de nodos en el clúster. Por lo tanto mantiene un tiempo de respuesta rápido.
- Almacenamiento de datos flexible: Cassandra se adapta a todos los formatos de datos posibles, incluidos: estructurados, semiestructurados y no estructurados. Puede adaptarse dinámicamente a los cambios en sus estructuras de datos según sus necesidades.

Features importantes

- Fácil distribución de datos: Cassandra proporciona la flexibilidad para distribuir datos donde los necesite mediante la replicación de datos en varios centros de datos.
- Compatibilidad con transacciones: Cassandra admite propiedades como Atomicidad, Consistencia, Aislamiento y Durabilidad (ACID)
- Escrituras rápidas: Cassandra se diseñó para ejecutarse en hardware básico económico. Realiza escrituras increíblemente rápidas y puede almacenar cientos de terabytes de datos, sin sacrificar la eficiencia de lectura.

Arquitectura de Cassandra

Distribución de nodos en cluster

- Cassandra tiene un sistema distribuido peer-to-peer en sus nodos, y los datos se distribuyen entre todos los nodos en un clúster.
- Todos los nodos en un clúster juegan el mismo rol. Cada nodo es independiente y a la vez interconectado con otros nodos.
- Cada nodo en un clúster puede aceptar solicitudes de lectura y escritura, independientemente de dónde se encuentren realmente los datos en el clúster.
- Cuando un nodo deja de funcionar, las solicitudes de lectura/escritura se pueden atender desde otros nodos en la red.

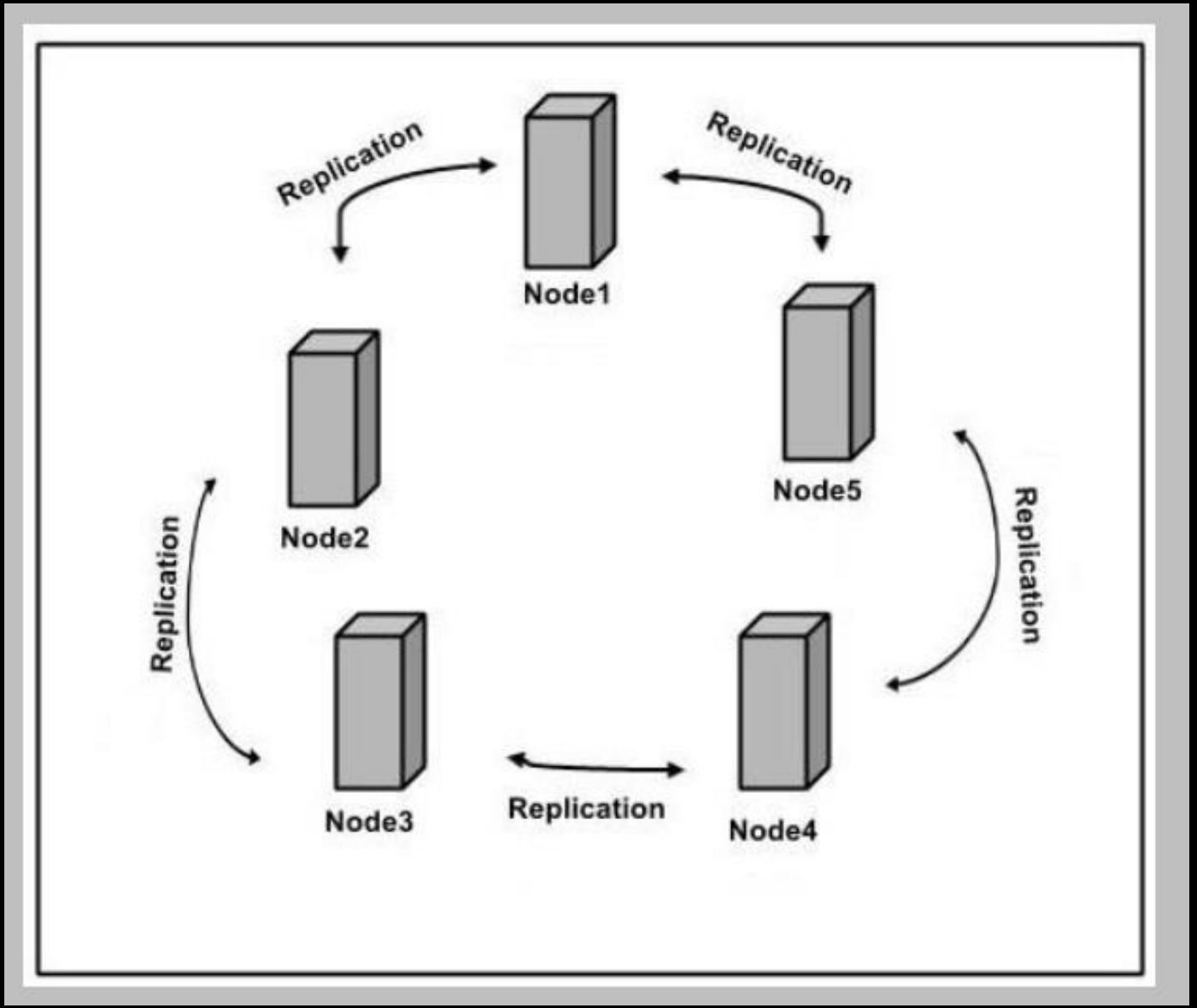
Replicación de datos

- En Cassandra, uno o más de los nodos de un clúster actúan como réplicas de un dato determinado. Si se detecta que algunos de los nodos respondieron con un valor desactualizado, Cassandra devolverá el valor más reciente al cliente.
- Después de devolver el valor más reciente, Cassandra realiza una reparación de lectura en segundo plano para actualizar los valores obsoletos.

Replicación de datos

Cassandra usa el protocolo Gossip en segundo plano para permitir que los nodos se comuniquen entre sí y detecten cualquier nodo defectuoso en el clúster.

[Gossip Dissemination (martinfowler.com)]



Componentes de Cassandra

Nodo: Es el lugar donde se almacenan los datos.

Centro de datos: es una colección de nodos relacionados.

Clúster: un clúster es un componente que contiene uno o más centros de datos.

Registro de confirmación: el registro de confirmación es un mecanismo de recuperación de fallas en Cassandra. Cada operación de escritura se escribe en el registro de confirmación.

Mem-table: una mem-table es una estructura de datos residente en la memoria. Después del registro de confirmación, los datos se escribirán en la tabla mem. A veces, para una familia de una sola columna, habrá varias tablas de memoria.

SSTable: es un archivo de disco al que se descargan los datos de la tabla mem cuando su contenido alcanza un valor de umbral.

Filtro Bloom: estos no son más que algoritmos rápidos y no deterministas para probar si un elemento es miembro de un conjunto. Es un tipo especial de caché. Se accede a los filtros Bloom después de cada consulta.

Cassandra Query Language

- Los usuarios pueden acceder a Cassandra a través de sus nodos usando Cassandra Query Language (CQL). CQL trata la base de datos (Keyspace) como un contenedor de tablas. Los programadores usan cqlsh: un shell para trabajar con CQL o controladores de lenguaje de aplicación separados.
- Los clientes acceden a cualquiera de los nodos para sus operaciones de lectura y escritura. El nodo coordinador funge como proxy entre el cliente y los nodos que contienen los datos.

Operaciones de escritura

- Cada actividad de escritura de los nodos es capturada por los registros de confirmación (commit log) escritos en los nodos. Posteriormente, los datos se capturan y almacenanen la mem_table..
- Una vez que se llena esa tabla los datos se escriben al archivo Sstabel.
- Todas las escrituras se particionan y replican automáticamente en todo el clúster.
- Cassandra consolida periódicamente las SSTables, descartando datos innecesarios.

Operaciones de lectura

- Durante las operaciones de lectura, Cassandra obtiene valores de la tabla mem y verifica el filtro bloom para encontrar la SSTable adecuada que contenga los datos requeridos.

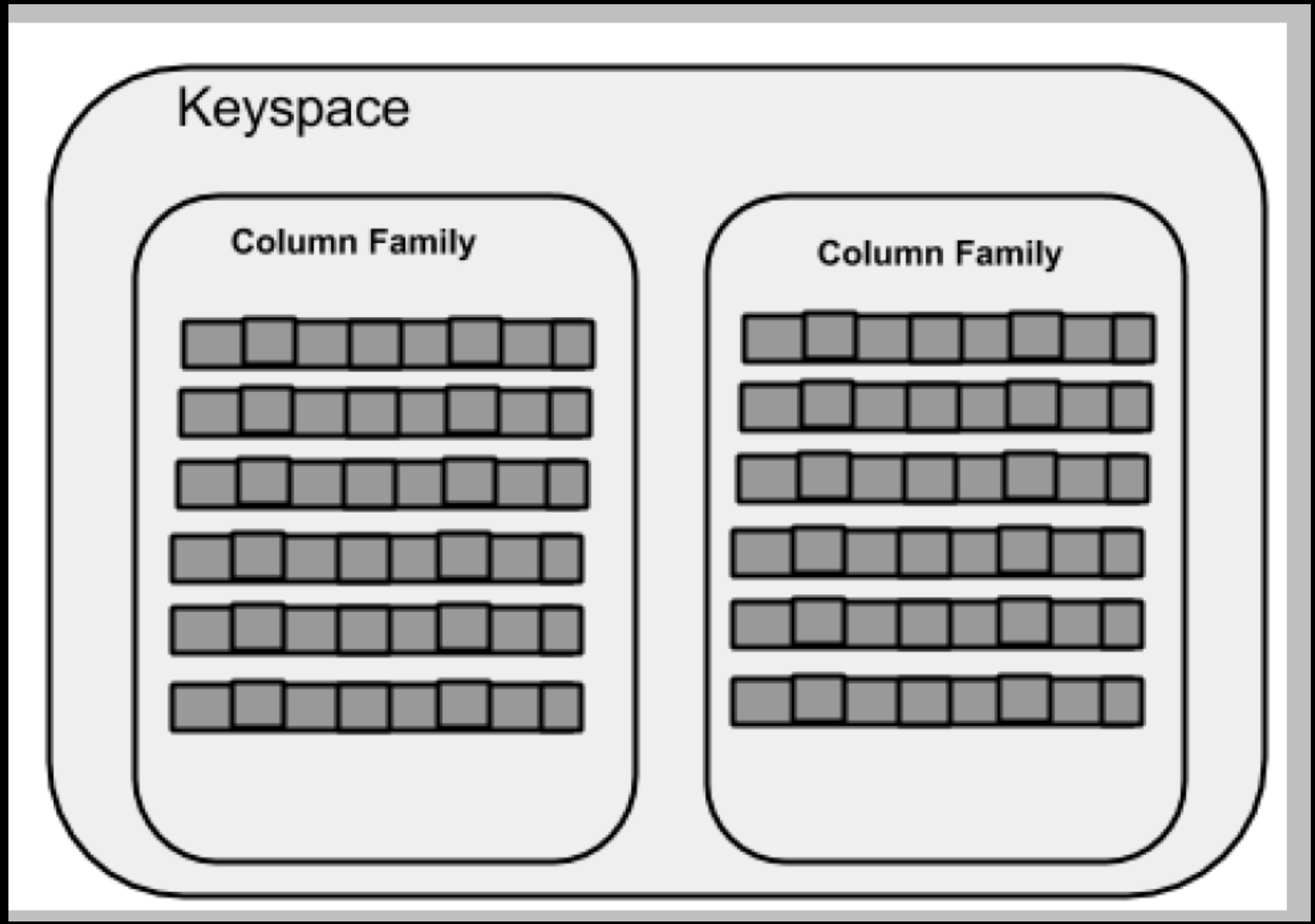
Modelo de datos

- Cluster
- La base de datos de Cassandra se distribuye en varias máquinas que funcionan juntas. El contenedor más externo se conoce como el Clúster. Para el manejo de fallas, cada nodo contiene una réplica y, en caso de falla, la réplica se hace cargo. Cassandra organiza los nodos en un grupo, en formato de anillo, y les asigna datos.

Modelo de datos

- Keyspace
- Keyspace es el contenedor más externo para datos en Cassandra. Los atributos básicos de un Keyspace en Cassandra son:
- Factor de replicación: Es la cantidad de máquinas en el clúster que recibirán copias de los mismos datos.
- ● Estrategia de colocación de réplicas: No es más que la estrategia de colocar réplicas en el ring.
- ●Familias de columnas: Keyspace es un contenedor para una lista de una o más familias de columnas. Una familia de columnas, a su vez, es un contenedor de una colección de filas. Cada fila contiene columnas ordenadas. Las familias de columnas representan la estructura de sus datos. Cada keyspace tiene al menos una y, a menudo, muchas familias de columnas.

Modelo de datos



Modelo de datos

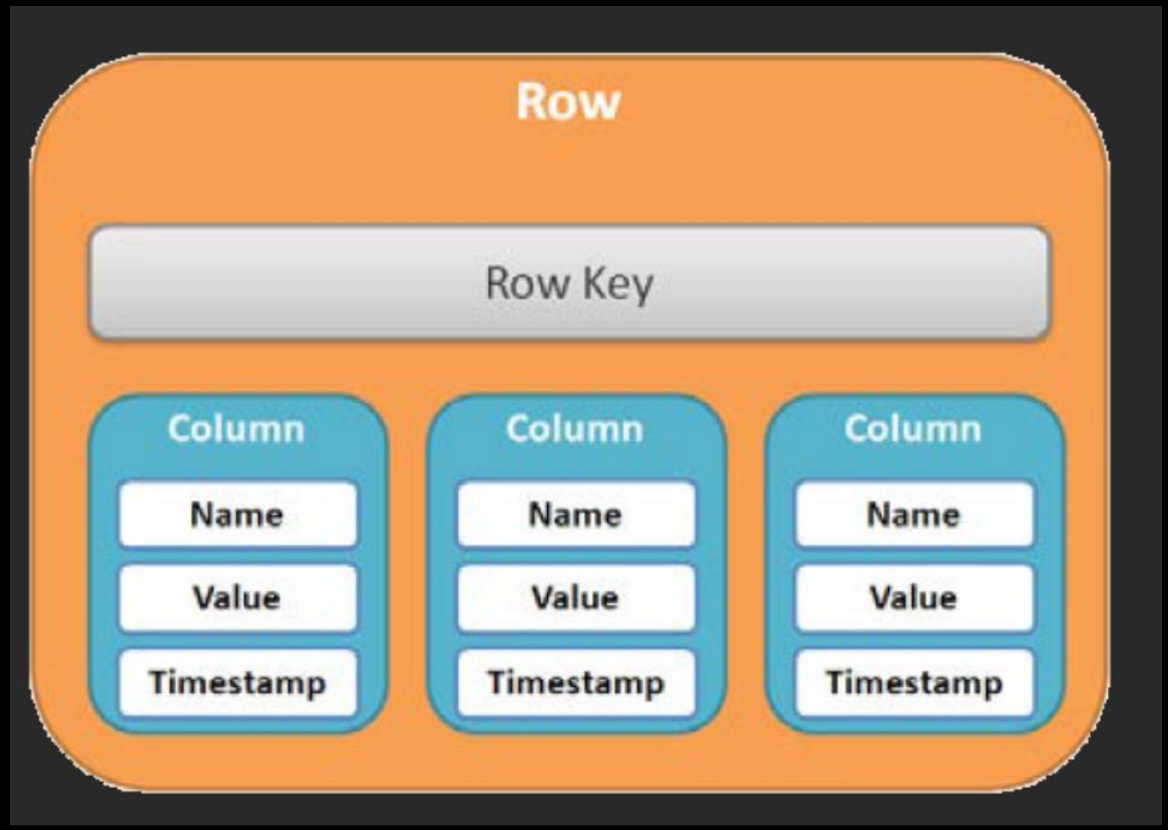
- Column Family
- Una familia de columnas es un contenedor para una colección ordenada de filas. Cada fila, a su vez, es una colección ordenada de columnas.

Modelo de datos: Columna

- Una columna es el elemento de dato más pequeño de Cassandra. Cada columna consiste de un nombre, el valor de la columna, un timestamp y un ttl.
- Timestamp para resolución de conflictos durante operaciones de escritura.
- TTL: valor opcional usado para marcar columnas que se eliminan después de expirar.

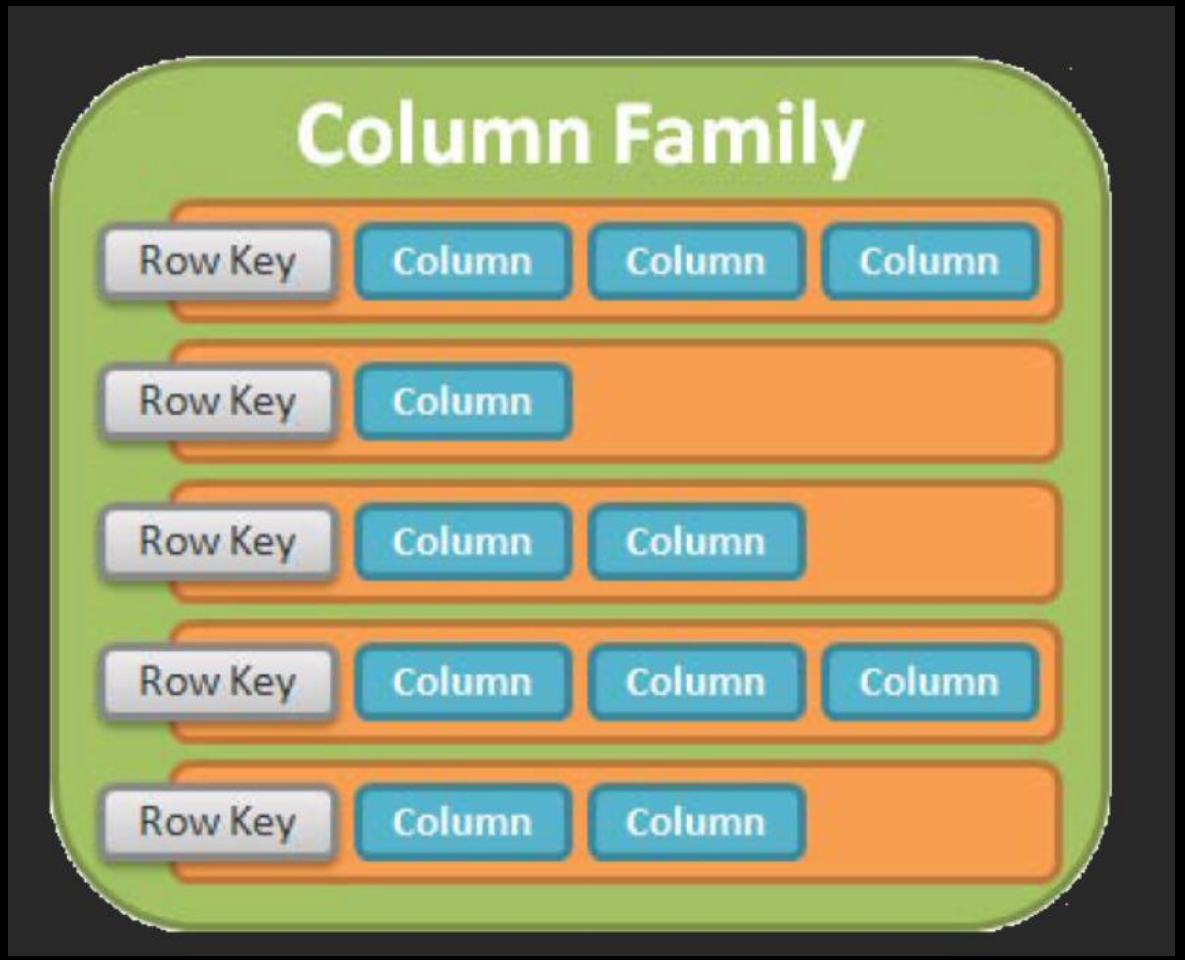
Modelo de datos: Fila

- Cada fila consiste de una llave de la fila (su llave primaria y un conjunto de columnas. Cada fila puede tener diferentes nombres de columnas (orientado a filas y orientado a columnas)



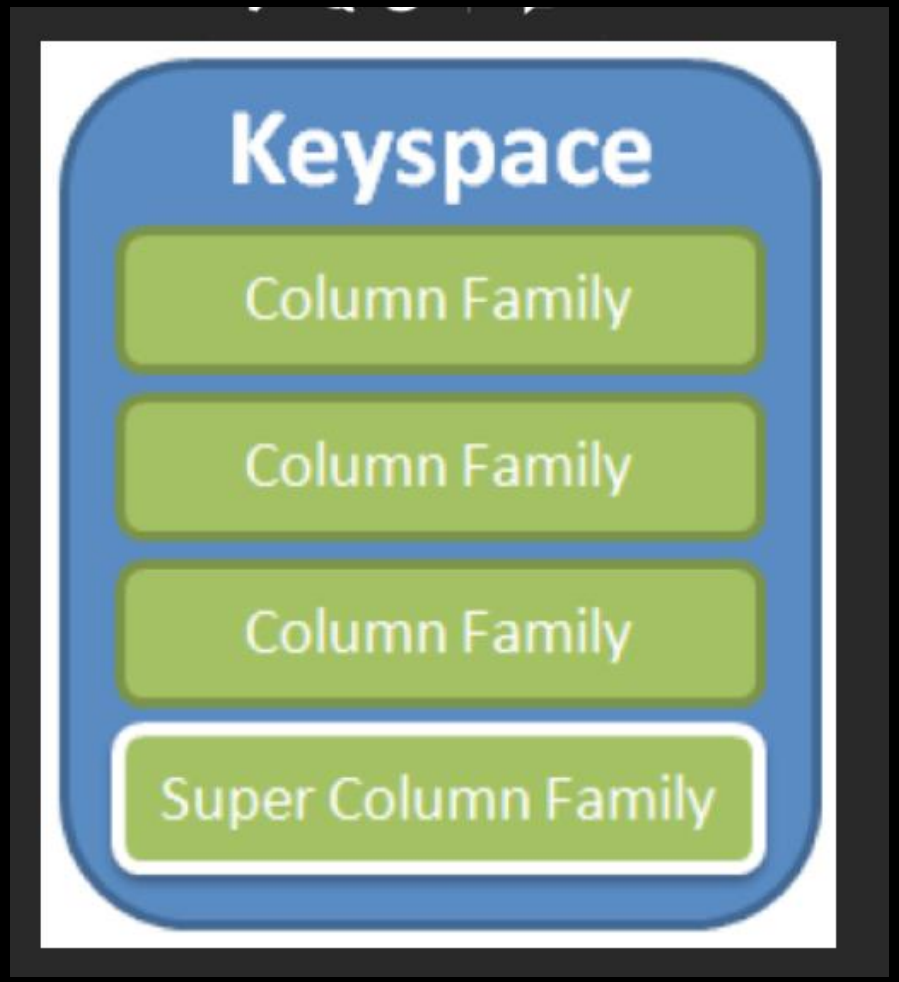
Modelo de datos: Familia de columnas

- Una llave de fila en la familia de columnas debe ser única y usarse para identificar filas. Aunque no es lo mismo, la familia de columnas puede ser análoga a una tabla en una base de datos relacional. Las familias de columnas brindan una mayor flexibilidad al permitir diferentes columnas en diferentes filas.



Modelo de datos: Keyspace

- El espacio de claves o llave es el elemento del modelo de datos más externo, un contenedor para una colección de familias de columnas y una familia de super columnas.

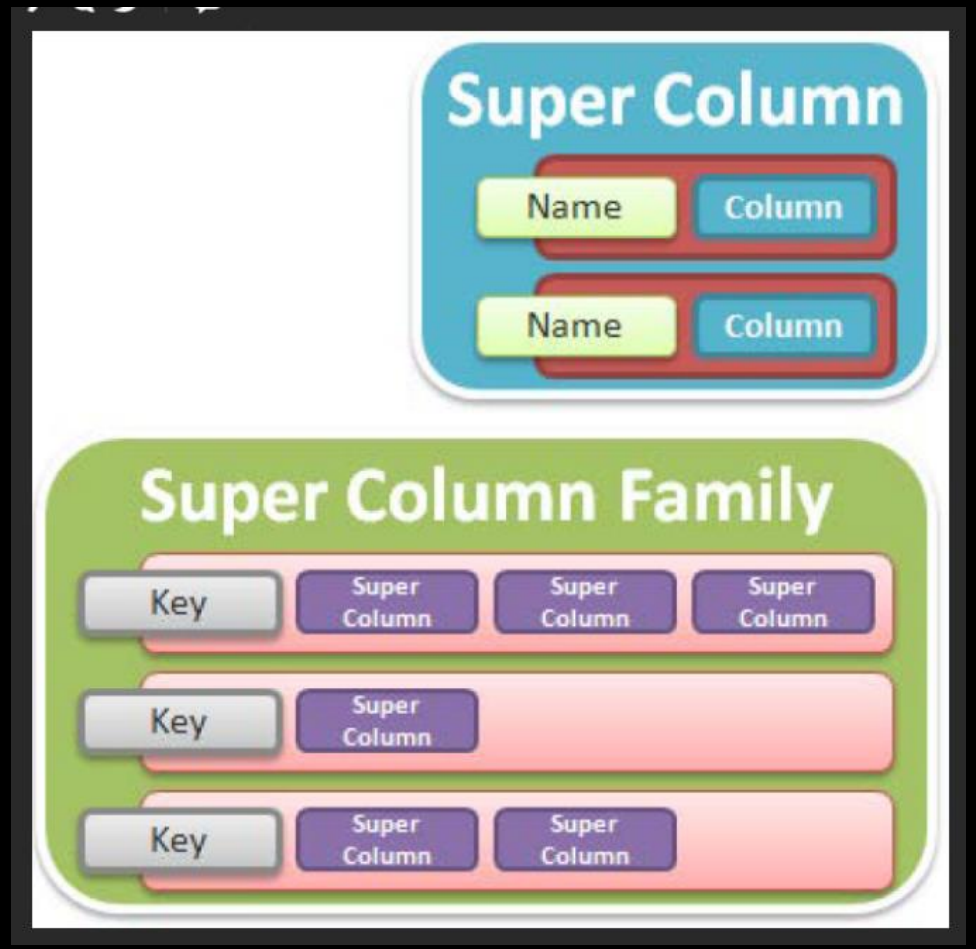


Modelo de datos: keyspace y el esquema

- Un espacio de claves es análogo a un esquema o base de datos en un modelo relacional.
- Cada clúster de Cassandra tiene un espacio de claves del sistema para almacenar metadatos de todo el sistema. Keyspace contiene configuraciones de replicación que controlan cómo se distribuyen y replican los datos en los clústeres. Por lo general, un clúster solo contiene un espacio de claves, pero un clúster puede contener más de un espacio de claves.

Modelo de datos: super columna y familia de super columnas

- Una super columna consiste de dos o más columnas y la familia de super columnas es una colección de super columnas.



Cassandra’s Data Model

- In this section, we’ll take a bottom-up approach to understanding Cassandra’s data model.

The simplest data store

- The simplest data store you would conceivably want to work with might be an array or list.
- An array is a clearly useful data structure, but not semantically rich.

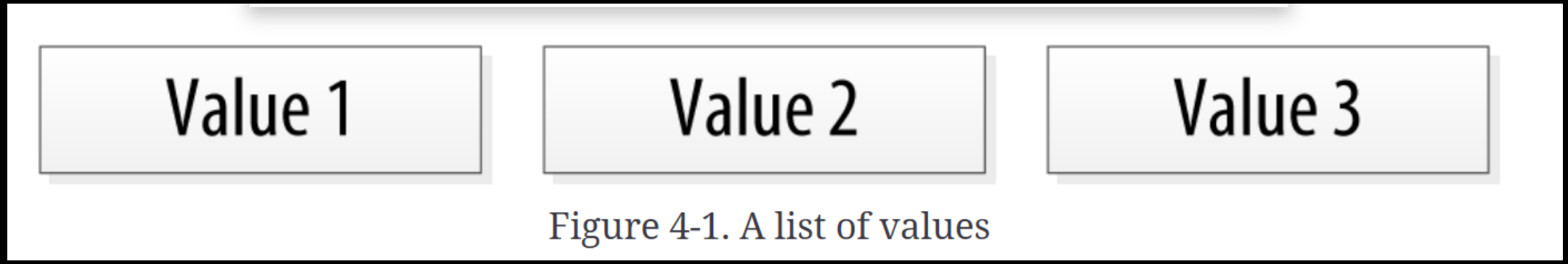


Figure 4-1. A list of values

Let’s add a second dimension to this list: names to match the values.

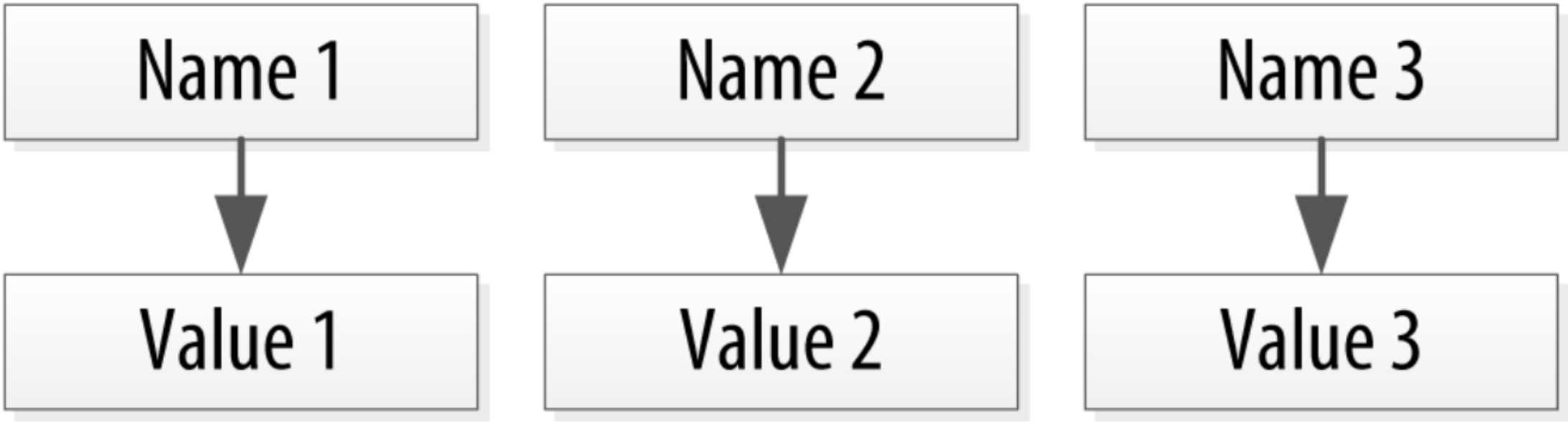


Figure 4-2. A map of name/value pairs

This is an improvement because you can know the names of your values. So if you decided that your map would hold user information, you could have column names like first_name, last_name, phone, email, and so on. This is a somewhat richer structure to work with.

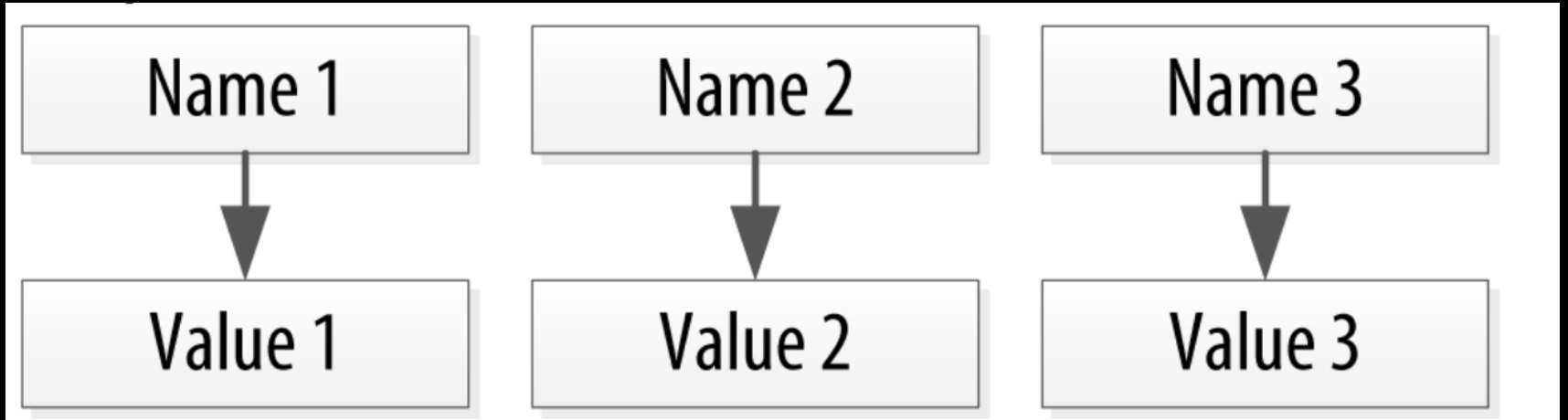


Figure 4-2. A map of name/value pairs

Enough?

- But the structure you've built so far works only if you have one instance of a given entity, such as a single person, user, hotel, or tweet. It doesn't give you much if you want to store multiple entities with the same structure, which is certainly what you want to do.
- So you need something that will group some of the column values together in a distinctly addressable group.
- You need a key to reference a group of columns that should be treated together as a set.
- You need rows.

The contents of a simple row

A primary key, which is itself one or more columns, and additional columns.

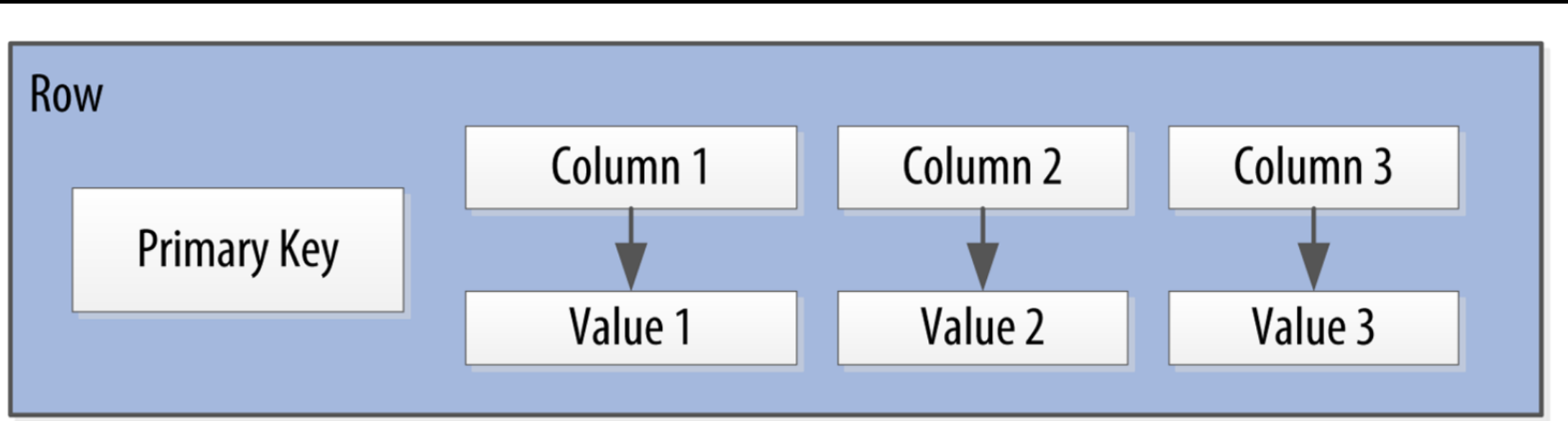


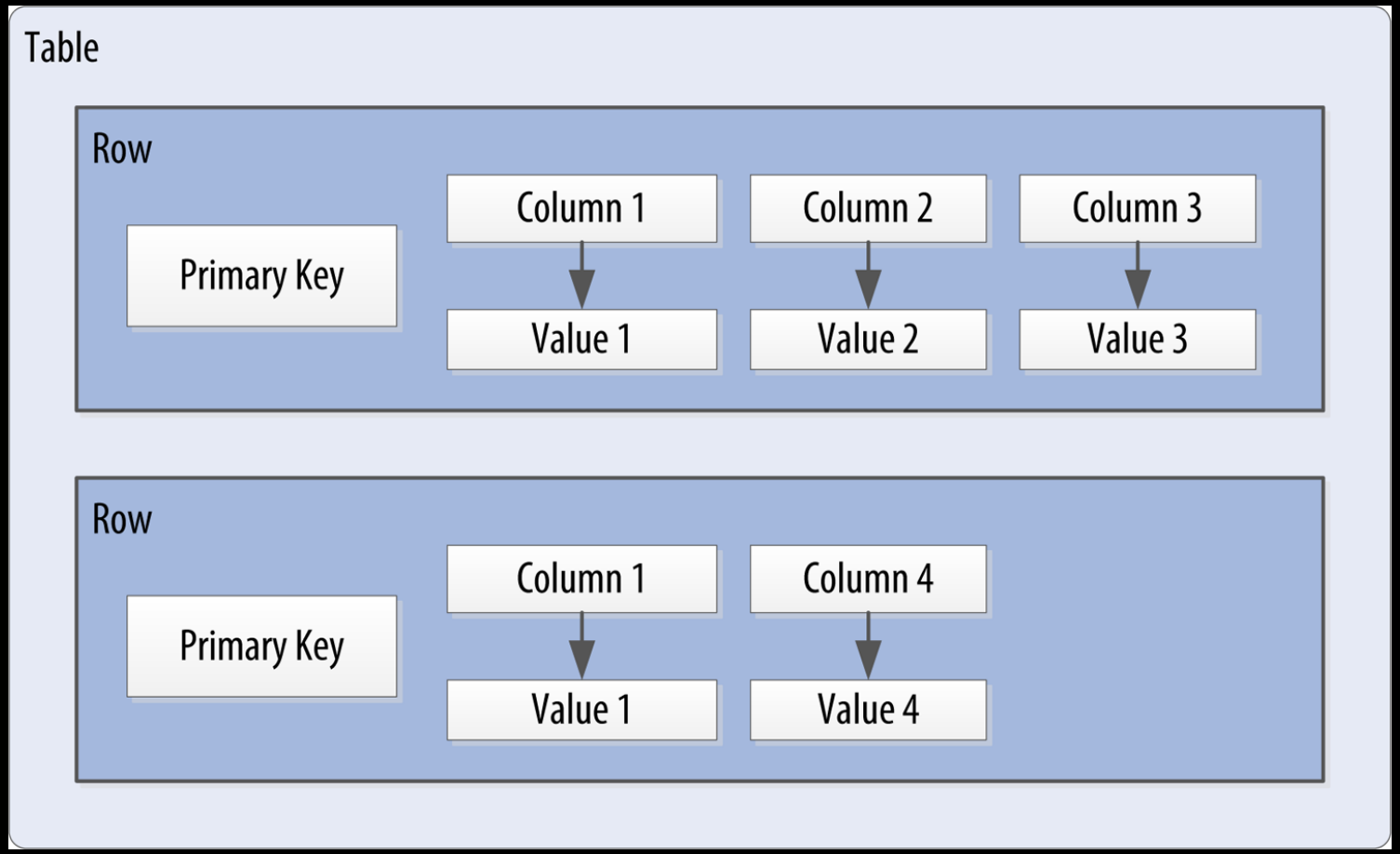
Figure 4-3. A Cassandra row

Cassandra tables

- Cassandra defines a table to be a logical division that associates similar data.
- For example, you might have a user table, a hotel table, an address book table, and so on.
- In this way, a Cassandra table is analogous to a table in the relational world.
- You don’t need to store a value for every column every time you store a new entity.

Some people have a second phone number and some don’t

- That’s OK.
- Instead of storing null for those values you don’t know, which would waste space, you just don’t store that column at all for that row. So now you have a sparse (escasa en español), multidimensional array structure.

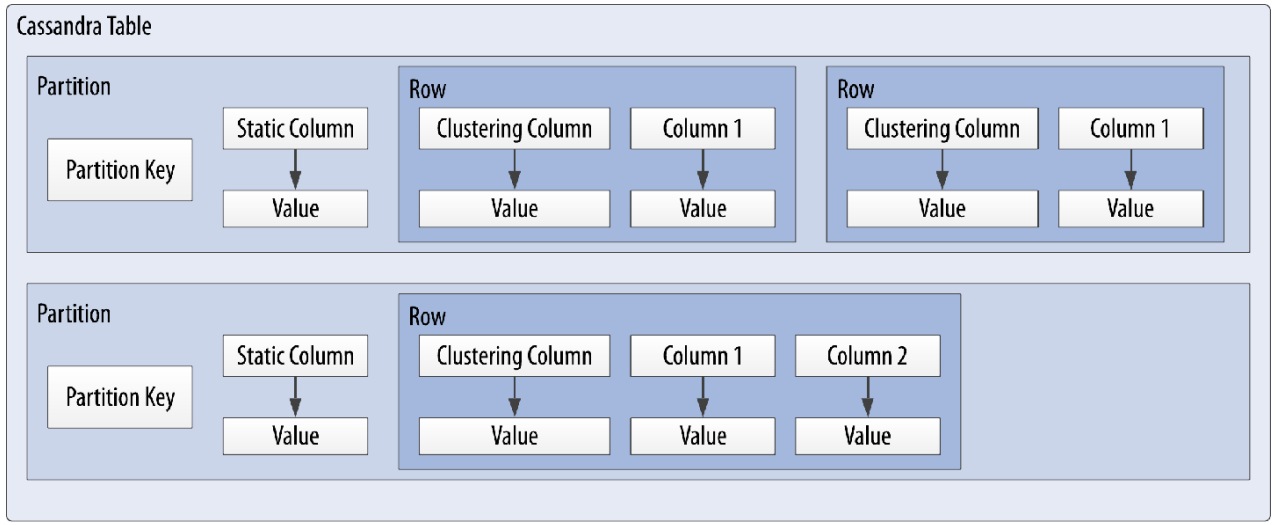


What about primary keys in Cassandra

- Cassandra uses a special type of primary key called **a composite key** (or compound key) to represent groups of related rows, also called **partitions**.
- The composite key consists of a partition key, plus an optional set of **clustering columns**.
- The partition key is used to determine the nodes on which rows are stored and can itself consist of multiple columns.
- The clustering columns are used to control how data is sorted for storage within a partition.
- Cassandra also supports an additional construct called **a static column**, which is for storing data that is not part of the primary key but is shared by every row in a partition.

Partitions and partition keys

- Each partition is uniquely identified by a partition key, and the clustering keys are used to uniquely identify the rows within a partition.
- In case where no clustering columns are provided, each partition consists of a single row.

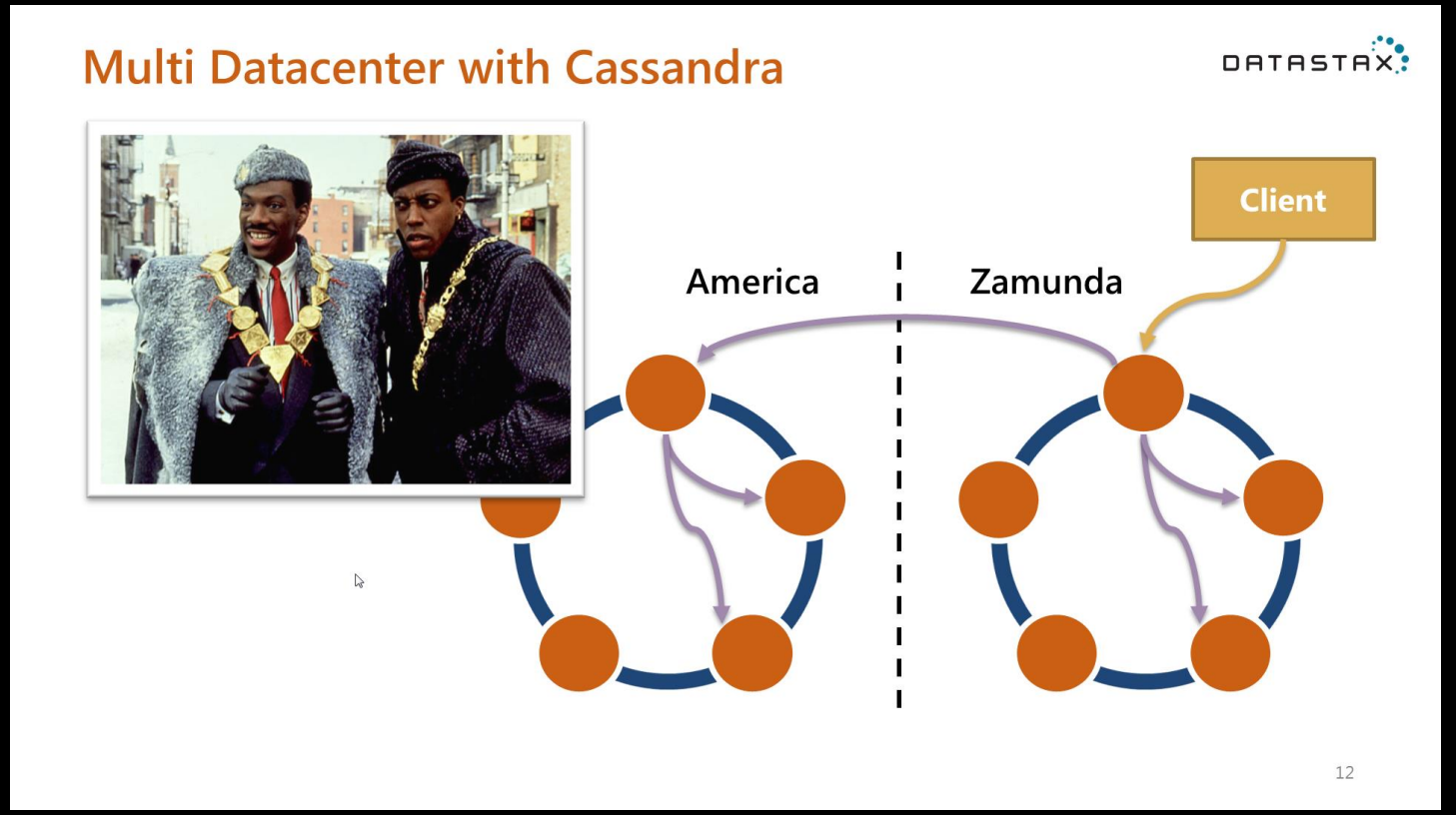


Putting these concepts all together, we have the basic Cassandra data structures:

- **The column**, which is a name/value pair
- **The row**, which is a container for columns referenced by a primary key.
- **The partition**, which is a group of related rows that are stored together on the same nodes.
- **The table**, which is a container for rows organized by partitions
- **The keyspace**, which is a container for tables.
- **The cluster**, which is a container for keyspaces that spans one or more nodes.

Clusters

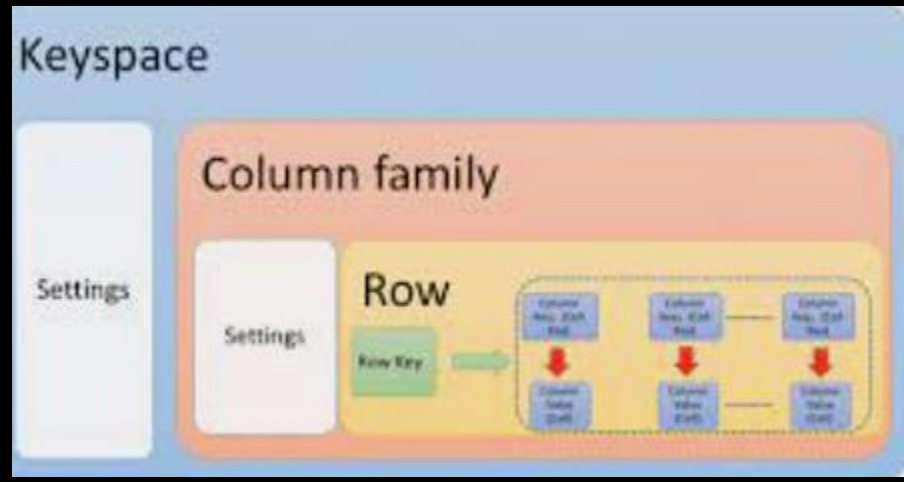
Cassandra database is specifically designed to be distributed over several machines operating together that appear as a single instance to the end user. **So the outermost structure in Cassandra is the cluster**, sometimes called the ring, because Cassandra assigns data to nodes in the cluster by arranging them in a ring.



Keyspaces

```
cassandra@cqlsh> CREATE KEYSPACE “Mykeyspace” with
replication = {‘class’ : ‘SimpleStrategy’,
‘replication_factor’ : ‘3’};
```

- A cluster is a container for **keyspaces**.
- A keyspace is the outermost container for data in Cassandra, corresponding closely to a database in the relational model.
- In the same way that a database is a container for tables in the relational model, a **keyspace is a container for tables in the Cassandra** data model.
- Like a relational database, a keyspace has a name and a set of attributes that define keyspace-wide behavior such as replication.
- Amazon Keyspaces (for Apache Cassandra) - Amazon Web Services



Tables

A table is a container for an ordered collection of rows, each of which is itself an ordered collection of columns. Rows are organized in partitions and assigned to nodes in a Cassandra cluster according to the column(s) designated as the partition key. The ordering of data within a partition is determined by the clustering columns.

When you write data to a table in Cassandra, you specify values for one or more columns. That collection of values is called a row. You must specify a value for each of the columns contained in the primary key as those columns taken together will uniquely identify the row.

```
cqlsh:my_keyspace> CREATE TABLE user ( first_name text ,
  last_name text, title text, PRIMARY KEY (last_name, first_name)) ;
```

Columns

- A column is the most basic unit of data structure in the Cassandra data model.
- A column contains a name and a value. You constrain each of the values to be of a particular type when you define the column.
- You'll want to dig into the various types that are available for each column, but first let's take a look into some other attributes of a column that we haven't discussed yet: **timestamps and time to live**.
- These attributes are key to understanding how Cassandra uses time to keep data current.

Timestamps

- Each time you write data into Cassandra, a timestamp, in microseconds, is generated for each column value that is inserted or updated.
- Internally, Cassandra uses these timestamps for resolving any conflicting changes that are made to the same value, in what is frequently referred to as a last write wins approach.

Time to live (TTL)

- One very powerful feature that Cassandra provides is the ability to expire data that is no longer needed.
- This expiration is very flexible and works at the level of individual column values. The time to live (or TTL) is a value that Cassandra stores for each column value to indicate how long to keep the value.
- The TTL value defaults to null, meaning that data that is written will not expire.

CQL Types

CQL Types

- Numeric Data Types
- Textual Data Types
- Time and Identity Data Types
- Other Simple Data Types
- Collections
- Tuples
- User-Defined Types

Numeric Data Types

int **A 32-bit signed integer (as in Java)** bigint **A 64-bit signed long integer (equivalent to a Java long)** smallint **A 16-bit signed integer (equivalent to a Java short)** tinyint **An 8-bit signed integer (as in Java)**	varint **A variable precision signed integer (equivalent to java.math.BigInteger)** float **A 32-bit IEEE-754 floating point (as in Java)** double **A 64-bit IEEE-754 floating point (as in Java)** **decimal** **A variable precision decimal (equivalent to java.math.BigDecimal)**

Textual Data Types

text, varchar

Synonyms for a UTF-8 character string

ascii

An ASCII character string

Time Data Types

timestamp

The time can be encoded as a 64-bit signed integer, but it is typically much more useful to input a timestamp using one of several supported ISO 8601 date formats.

date, time

The 2.2 release introduced date and time types that allowed these to be represented independently; that is, a date without a time, and a time of day without reference to a specific date. As with timestamp, these types support ISO 8601 formats.

Identity Data Types

uuid

A universally unique identifier (UUID) is a 128-bit value in which the bits conform to one of several types, of which the most commonly used are known as Type 1 and Type 4. The CQL uuid type is a Type 4 UUID, which is based entirely on random numbers. UUIDs are typically represented as dash-separated sequences of hex digits.

timeuuid

This is a Type 1 UUID, which is based on the MAC address of the computer, the system time, and a sequence number used to prevent duplicates. This type is frequently used as a conflict-free timestamp.

Other Simple Data Types

- **boolean**
  - This is a simple true/false value
- **blob**
  - The CQL blob type is useful for storing media or other binary file types. Cassandra does not validate or examine the bytes in a blob.
- **inet**
  - This type represents IPv4 or IPv6 internet addresses. cqlsh accepts any legal format for defining IPv4 addresses, including dotted or nondotted representations containing decimal, octal, or hexadecimal values.
- **counter**
  - The counter data type provides a 64-bit signed integer, whose value cannot be set directly, but only incremented or decremented. Cassandra is one of the few databases that provides race-free increments across data centers.

Collections

- CQL provides three collection types to help you with these situations: sets, lists, and maps.
- **set**
  - The set data type stores a collection of elements. The elements are unordered when stored, but are returned in sorted order.
- **list**
  - The list data type contains an ordered list of elements. By default, the values are stored in order of insertion.
- **map**
  - The map data type contains a collection of key-value pairs. The keys and the values can be of any type except counter.

Tuples

- Cassandra provides two different ways to manage more complex data structures: **tuples** and **user-defined types**.
- Tuples provides a way to have a fixed-length set of values of various types.

```
cqlsh:my_keyspace> ALTER TABLE user ADD
  address tuple<text, text, text, int>;
```

```
cqlsh:my_keyspace> UPDATE user SET address =
  ('7712 E. Broadway', 'Tucson', 'AZ', 85715 )
  WHERE first_name = 'Mary' AND last_name = 'Rodriguez';
```

But

- There is also no way to update individual fields of a tuple; the entire tuple must be updated. For these reasons, tuples are infrequently used in practice, because Cassandra offers an alternative that provides a way to name and access each value, with user defined types.

User Defined Types

Cassandra gives you a way to define your own types to extend its data model.

These user-defined types (UDTs) are easier to use than tuples since you can specify the values by name rather than position.

Create your own address type:

```
cqlsh:my_keyspace> CREATE TYPE address (
  street text,
  city text,
  state text,
  zip_code int);
```

Let’s write some code

Create and describe keyspace

```
cqlsh> CREATE KEYSPACE unir WITH replication = {'class':  'SimpleStrategy', 'replication_facto
r': '1'} AND  durable_writes = true;
cqlsh>
cqlsh> DESCRIBE KEYSPACE unir;

CREATE KEYSPACE unir WITH replication = {'class': 'SimpleStrategy', 'replication_factor': '1'}
  AND durable_writes = true;

cqlsh>
```

Create a table in your keyspace

```
cqlsh> USE unir;
cqlsh:unir> CREATE TABLE user ( first_name text ,
        ...   last_name text, title text, PRIMARY KEY (last_name, first_name)) ;
cqlsh:unir> DESCRIBE TABLE user;
```

```
CREATE TABLE unir.user (
    last_name text,
    first_name text,
    title text,
    PRIMARY KEY (last_name, first_name)
) WITH CLUSTERING ORDER BY (first_name ASC)
    AND bloom_filter_fp_chance = 0.01
    AND caching = {'keys': 'ALL', 'rows_per_partition': 'NONE'}
    AND comment = ''
    AND compaction = {'class': 'org.apache.cassandra.db.compaction.SizeTieredCompactionStrateg
y', 'max_threshold': '32', 'min_threshold': '4'}
    AND compression = {'chunk_length_in_kb': '64', 'class': 'org.apache.cassandra.io.compress.
LZ4Compressor'}
    AND crc_check_chance = 1.0
    AND dclocal_read_repair_chance = 0.1
    AND default_time_to_live = 0
    AND gc_grace_seconds = 864000
    AND max_index_interval = 2048
    AND memtable_flush_period_in_ms = 0
    AND min_index_interval = 128
    AND read_repair_chance = 0.0
    AND speculative_retry = '99PERCENTILE';
```

Writing and Reading Data in cqlsh

```
cqlsh:unir> INSERT INTO user (first_name, last_name, title)
        ...   VALUES ('Bill', 'Nguyen', 'Mr.');
cqlsh:unir> SELECT * FROM user WHERE first_name='Bill' AND
        ...   last_name='Nguyen';

 last_name | first_name | title
-----------+------------+-------
    Nguyen |       Bill |   Mr.

(1 rows)
cqlsh:unir>
```

What happens when you only specify one of the values?

```
cqlsh:unir> SELECT * FROM user where last_name = 'Nguyen';

 last_name | first_name | title
-----------+------------+-------
    Nguyen |       Bill |   Mr.

(1 rows)
cqlsh:unir>
```

What happens when you only specify one of the values?

```
cqlsh:unir> SELECT * FROM user where first_name = 'Bill';
InvalidRequest: Error from server: code=2200 [Invalid query] message="Cannot execute this quer
y as it might involve data filtering and thus may have unpredictable performance. If you want
to execute this query despite the performance unpredictability, use ALLOW FILTERING"
cqlsh:unir>
```

This behavior might not seem intuitive at first, but it has to do with the composition of the primary key you used for this table. This is your first clue that there might be something a bit different about accessing data in Cassandra as compared to what you might be used to in SQL.

Let’s add another user with the same last_name and then repeat the SELECT command

```
cqlsh:unir> INSERT INTO user (first_name, last_name, title)
        ...   VALUES ('Wanda', 'Nguyen', 'Mrs.');
cqlsh:unir> SELECT * FROM user WHERE last_name='Nguyen';

 last_name | first_name | title
-----------+------------+-------
    Nguyen |       Bill |   Mr.
    Nguyen |      Wanda |  Mrs.

(2 rows)
```

When you add a new row to the table, you are not required to provide values for nonprimary key columns.

```
cqlsh:unir> INSERT INTO user (first_name, last_name)
        ...  VALUES ('Mary', 'Rodriguez');
```

```
cqlsh:unir> SELECT * FROM user;

 last_name | first_name | title
-----------+------------+-------
 Rodriguez |       Mary |  null
    Nguyen |       Bill |   Mr.
    Nguyen |      Wanda |  Mrs.

(3 rows)
```

Using the ALTER TABLE command

```
cqlsh:unir> ALTER TABLE user ADD middle_initial text;
cqlsh:unir> DESCRIBE TABLE user;

CREATE TABLE unir.user (
    last_name text,
    first_name text,
    middle_initial text,
    title text,
    PRIMARY KEY (last_name, first_name)
) WITH CLUSTERING ORDER BY (first_name ASC)
    AND bloom_filter_fp_chance = 0.01
    AND caching = {'keys': 'ALL', 'rows_per_partition': 'NONE'}
    AND comment = ''
    AND compaction = {'class': 'org.apache.cassandra.db.compaction.SizeTieredCompactionStrat
y', 'max_threshold': '32', 'min_threshold': '4'}
    AND compression = {'chunk_length_in_kb': '64', 'class': 'org.apache.cassandra.io.compres
LZ4Compressor'}
    AND crc_check_chance = 1.0
    AND dclocal_read_repair_chance = 0.1
    AND default_time_to_live = 0
    AND gc_grace_seconds = 864000
    AND max_index_interval = 2048
    AND memtable_flush_period_in_ms = 0
```

Let’s write some additional rows, populate different columns for each, and read the results:

```
cqlsh:unir> INSERT INTO user (first_name, middle_initial, last_name,
        ...   title)
        ...   VALUES ('Bill', 'S', 'Nguyen', 'Mr.');
cqlsh:unir> INSERT INTO user (first_name, middle_initial, last_name,
        ...   title)
        ...   VALUES ('Bill', 'R', 'Nguyen', 'Mr.');
cqlsh:unir> SELECT * FROM user WHERE first_name='Bill' AND
        ...   last_name='Nguyen';

 last_name | first_name | middle_initial | title
-----------+------------+----------------+-------
    Nguyen |       Bill |              R |   Mr.

(1 rows)
```

Was this the result that you expected?

If you're following closely, you may have noticed that both of the INSERT statements here specify a previous row uniquely identified by the primary key columns first_name and last_name.

As a result, Cassandra has faithfully updated the row you indicated, and your SELECT will only return the single row that matches that primary key. The two INSERT statements have only served to first set and then overwrite the middle_initial.

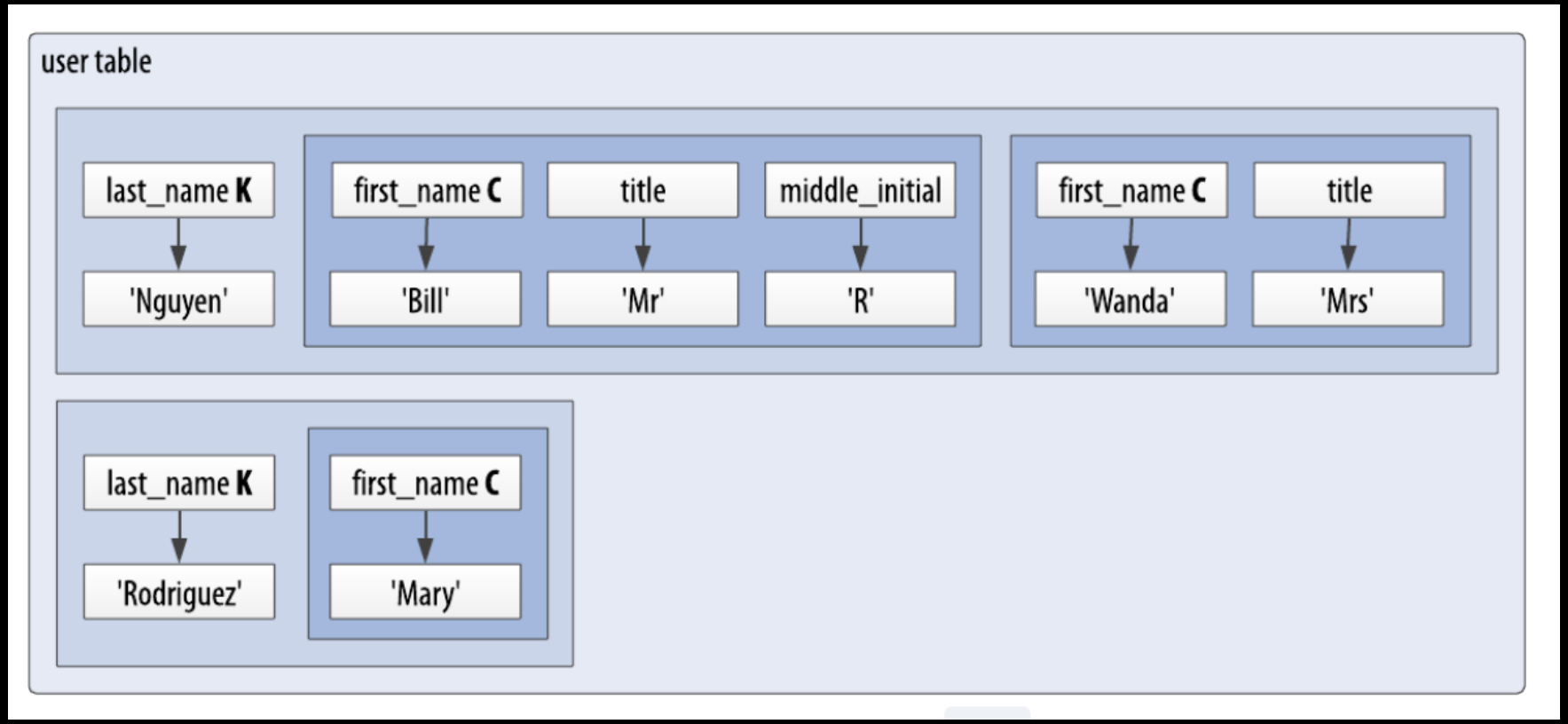
```
cqlsh:unir> INSERT INTO user (first_name, middle_initial, last_name,
       ...   title)
       ...   VALUES ('Bill', 'S', 'Nguyen', 'Mr.');
cqlsh:unir> INSERT INTO user (first_name, middle_initial, last_name,
       ...   title)
       ...   VALUES ('Bill', 'R', 'Nguyen', 'Mr.');
cqlsh:unir> SELECT * FROM user WHERE first_name='Bill' AND
       ...   last_name='Nguyen';

 last_name | first_name | middle_initial | title
-----------+------------+----------------+-------
    Nguyen |       Bill |              R |   Mr.

(1 rows)
```

INSERT, UPDATE, AND UPSERT

- Because Cassandra uses an append model, there is no fundamental difference between the insert and update operations.
- If you insert a row that has the same primary key as an existing row, the row is replaced.
- If you update a row and the primary key does not exist, Cassandra creates it.
- For this reason, it is often said that Cassandra supports *upsert*, meaning that inserts and updates are treated the same

Visualize the data you've inserted up to this point

last_name	first_name	middle_initial	title
Rodriguez	Mary	null	null
Nguyen	Bill	R	Mr.
Nguyen	Wanda	null	Mrs.



Viendo los timestamps

```
cqlsh:unir> SELECT first_name, last_name, title, writetime(title)
        ...   FROM user;

 first_name | last_name | title | writetime(title)
------------+-----------+-------+------------------
       Mary | Rodriguez |  null |             null
       Bill |    Nguyen |   Mr. | 1658197724775000
      Wanda |    Nguyen |  Mrs. | 1658196029504000

(3 rows)
```

As you might expect, there is no timestamp for a column that has not been set. You might expect that if you ask for the timestamp on first_name or last_name, you'd get a similar result to the values obtained for the title column.

You’re not allowed to ask for the timestamp on primary key columns:

```
cqlsh:unir>  SELECT WRITETIME(first_name) FROM user;
InvalidRequest: Error from server: code=2200 [Invalid query] message="Cannot use selection fun
ction writeTime on PRIMARY KEY part first_name"
cqlsh:unir>
```

Specify a timestamp when writing

- Cassandra also allows you to specify a timestamp you want to use when performing writes.
- To do this, you'll use **the CQL UPDATE** . Use the optional **USING TIMESTAMP** option to manually set a timestamp (note that the timestamp must be later than the one from your SELECT command, or the UPDATE will be ignored):

USING TIMESTAMP

```
cqlsh:unir> UPDATE user USING TIMESTAMP 1567886623298243
        ...   SET middle_initial = 'Q' WHERE first_name = 'Mary' AND last_name = 'Rodriguez';
cqlsh:unir> SELECT first_name, middle_initial, last_name,
        ...   WRITETIME(middle_initial) FROM user WHERE first_name = 'Mary' AND
        ...   last_name = 'Rodriguez';

 first_name | middle_initial | last_name | writetime(middle_initial)
------------+----------------+-----------+---------------------------
       Mary |              Q | Rodriguez |          1567886623298243

(1 rows)
```

Time to live (TTL)

- Let’s show this by adding the TTL() function to a SELECT command in cqlsh to see the TTL value for Mary’s title:

```
cqlsh:unir> SELECT first_name, last_name, TTL(title)
        ...   FROM user WHERE first_name = 'Mary' AND last_name = 'Rodriguez';

 first_name | last_name | ttl(title)
------------+-----------+------------
       Mary | Rodriguez |       null

(1 rows)
```

Let’s set the TTL on the middle_initial column to an hour (3,600 seconds) by adding the USING TTL option to your UPDATE command:

```
cqlsh:unir> UPDATE user USING TTL 3600 SET middle_initial =
        ...   'Z' WHERE first_name = 'Mary' AND last_name = 'Rodriguez';
```

```
cqlsh:unir> SELECT first_name, middle_initial,
        ...   last_name, TTL(middle_initial)
        ...   FROM user WHERE first_name = 'Mary' AND last_name = 'Rodriguez';

 first_name | middle_initial | last_name | ttl(middle_initial)
------------+----------------+-----------+---------------------
       Mary |              Z | Rodriguez |                3468

(1 rows)
```

Try inserting a row using TTL of 60 seconds and check that the row is initially there:

```
cqlsh:unir> INSERT INTO user (first_name, last_name)
        ...   VALUES ('Jeff', 'Carpenter') USING TTL 60;
cqlsh:unir> SELECT * FROM user WHERE first_name='Jeff' AND
        ...   last_name='Carpenter';

 last_name | first_name | middle_initial | title
-----------+------------+----------------+-------
 Carpenter |       Jeff |           null |  null

(1 rows)
cqlsh:unir> SELECT * FROM user WHERE first_name='Jeff' AND
        ...   last_name='Carpenter';

 last_name | first_name | middle_initial | title
-----------+------------+----------------+-------
 Carpenter |       Jeff |           null |  null

(1 rows)
cqlsh:unir>
cqlsh:unir> SELECT * FROM user WHERE first_name='Jeff' AND
        ...   last_name='Carpenter';

 last_name | first_name | middle_initial | title
-----------+------------+----------------+-------
```