2. Bases de datos NoSQL

2.1 Contexto y un poco de historia

1. Este tipo de BDs surge en el contexto del procesamiento masivo de datos en la Web. Adicionalmente, nuevas fuentes de datos se han considerado incluyendo las de sensores, GPSs o sistemas de monitoreo que generan cantidades masivas de datos.
2. Para este tipo de procesamiento los RDBMS mostraron problemas relacionados con la eficiencia en el procesamiento, en la paralelización, en la escalabilidad y en los costos.
3. Google fue el pionero en la solución a estos problemas construyendo una infraestructura escalable para el procesamiento en paralelo de grandes cantidades de datos.
4. Para resolver esto, Google creó un ambiente consistente en un sistema distribuido de archivos, un almacenamiento de datos orientado a columnas, un sistema de coordinación distribuido y un ambiente de ejecución de algoritmos paralelos basados en MapReduce.
5. Posteriormente, entre los creadores de Lucene, una primera versión libre (open source) de la infraestructura de Google, y personal de Yahoo crearon Hadoop, un sistema que brinda todas las características del sistema de Google.
6. Después, en 2007, Amazon presenta sus ideas sobre un sistema de almacenamiento de datos, Dynamo, distribuido, altamente disponible y eventualmente consistente.
7. A partir de ahí, varios gigantes de la computación -Facebook, Yahoo, IBM, etc.- han desarrollado productos y presentado casos de éxito relacionados con el procesamiento masivo de datos (Big Data).

2.2 Tipos de bases de datos NoSQL

1. Orientadas a columnas ordenadas
   a. Basa su idea en asignar una clave a cada entidad de datos. Después crea familias de columnas (buckets) donde guarda pares: clave/valor (key/value), para cada valor de la entidad, identificándolos previamente con su clave asignada originalmente.
   b. Principales productos:
      - HBase: de la infraestructura de Hadoop. Implementada en Java. Varios métodos de acceso, incluyendo Thrift[1]. Consultas con Hive usando una interfaz SQL-like.
      - Hypertable: Implementada en C++. Soporta Thrift. Consultas con HQL, una interfaz SQL-like.
      - Cloudata: Implementada en Java. Soporta Thrift. Consultas con CQL, una interfaz SQL-like.

[1] Thrift es un marco de trabajo y un lenguaje de definición de interfaces que permite el desarrollo de servicios y APIs entre varios lenguajes. Los servicios generados en Thrift trabajan eficientemente entre C++, Java, C#, Python, PHP, Ruby, Perl, Haskelly SmallTalk.

2. Almacenamientos clave/valor (key/value)
   a. Un HashMap, o un arreglo asociativo, es la estructura de datos más simple que puede mantener un conjunto de pares clave/valor. Se tienen algoritmos O(1) para acceder a los datos. La clave de un par clave/valor es un valor único en el conjunto y puede ser fácilmente buscado para acceder a los datos.
   b. Un ejemplo de este tipo de productos es: Oracle's Berkeley DB. Otros productos están basados en cache como: EHCache y Memcached. Algunos otros dan un conjunto de APIs como Redis.
   c. Principales productos (proporcionan una consistencia fuerte en el almacenamiento):
      - Membase (propuesta para ser mezclada con Couchbase): construida sobre Memcached. Implementada en C y C++. Acceso a través de APIs.
      - Kyoto Cabinet: Implementada en C++. Acceso a través de APIs para varios lenguajes (C, C++, Java, C#, Python, Ruby, Perl, Erlang y OCaml[2]).
      - Redis: Implementada en C. Acceso a través de un conjunto abundante de métodos y operaciones. Tiene bibliotecas para varios lenguajes.

3. Almacenamientos clave/valor con consistencia eventual
   a. Estos productos están basados en las ideas de Amazon's Dynamo, la cual da una disponibilidad y escalabilidad altas a costa de no tener una consistencia fuerte, sino eventual, lo cual significa que hay pequeños intervalos en los cuales hay inconsistencia entre nodos duplicados.
   b. Principales productos:
      - Cassandra: de la fundación Apache. Implementada en Java. Acceso a través de Thrift. Tiene clientes para múltiples lenguajes. Consultas con un lenguaje SQL-like.
      - Voldemort: implementada en Java. Proporciona módulos de almacenamiento usando Berkeley DB o MySQL. Acceso con Thrift y Avro. Puede ser usada con Hadoop.
      - Riak: implementada en Erlang. Acceso con interfaces para JSON y con bibliotecas para varios lenguajes.

4. Bases de datos de documentos
   a. Este término hace referencia a conjuntos débilmente estructurados de pares clave/valor en documentos, típicamente JSON (JavaScript Object Notation), y no a *documentos* u *hojas de cálculo*. Estas bases de datos tratan a un documento como un todo, evitando dividirlo en sus pares constitutivos clave/valor.
   b. Principales productos:
      - MongoDb: Implementada en C++. Proporciona controladores para múltiples lenguajes. Consultas con un lenguaje SQL-like.
      - CouchDB: de la fundación Apache. Implementada en Erlang. Acceso con REST; también usa herramientas y clientes Web estándares para acceder a las bases de datos.

[2] En donde se dice que hay interfaces o bibliotecas para varios lenguajes normalmente es para la mayoría de los que aquí se citan.

2.3 Cassandra

Apache Cassandra es un manejador de base de datos NoSQL, masivamente escalable y libre (open source). Permite manejar grandes volúmenes de datos estructurados, semi-estructurados y no estructurados, a través de múltiples centros de datos y en la nube. Entre las características de Cassandra se encuentran la disponibilidad, la escalabilidad lineal, la consistencia eventual y la simplicidad operacional.

Una base de datos de Cassandra consiste de familias de columnas, donde cada familia es un conjunto de pares clave-valor. Cada familia de columnas tiene una clave y consiste de columnas y filas. Se puede pensar en cada familia de columnas como una tabla y cada par clave-valor como un registro de la tabla. De hecho, a partir de CQL 3 las *familias de columnas* son llamadas *tablas*.

Es importante destacar que Cassandra no maneja el concepto de clave externa (foreign key) por lo que no se pueden hacer juntas entre tablas[3].

Cassandra, como todos los manejadores NoSQL, distribuye y procesa la información de una base de datos a través de múltiples nodos y es aquí cuando se obtiene su mayor ventaja. En estos apuntes, y en el ejemplo, lo mostrado es considerando únicamente instalaciones de un solo nodo.

2.3.1 CQL

Cassandra brinda varios tipos de interfaces para interactuar con sus bases de datos. Una de ellas es CQL, siglas de Cassandra Query Language, que es un lenguaje SQL-like, el cual consiste de una serie de instrucciones que permiten la manipulación de bases de datos NoSQL en este manejador. En esta sección se describen sus principales instrucciones y en la siguiente se brinda un ejemplo de su uso. Las instrucciones se muestran en su sintaxis más simple. Las instrucciones se pueden dar desde un programa o interactivamente por medio de un shell de cql (cqlsh).

1. Creación del keyspace

Un *keyspace* es el equivalente a una base de datos en SQL. El keyspace en Cassandra es un nombre que define cómo los datos van a ser duplicados en los nodos. La instrucción es:

```
create keyspace nombreEspacio
with replication = { 'class' : 'SimpleStrategy',
   'replication_factor' : 1};
```

Donde la opción `SimpleStrategy` se usaría en caso de pocos nodos, y la opción `NetworkTopologyStrategy` se usaría para una configuración con varios clusters (conjuntos de nodos). El 2º parámetro indica cuántas veces se van a repetir los datos.

[3] Se usará el término *tabla*, tal como se hace en la literatura de Cassandra, para hacer referencia a familias de columnas.

2. Usar el keyspace

Una vez creado el keyspace, hay que usarlo para que las instrucciones subsecuentes se ejecuten sobre este espacio. La instrucción es:

```
use nombreEspacio;
```

3. Crear una tabla

Una tabla, como se mencionó previamente, es una familia de columnas. Todos los datos deberán ser introducidos en tablas declaradas dentro de un keyspace. Una tabla puede tener claves primarias simples o compuestas (formadas por varias columnas). Las columnas de las tablas pueden ser de tipos simple de datos o pueden contener colecciones de datos. La instrucción es:

```
create table nombreTabla (
  columna1  tipoDatos,
  [columna2 tipoDatos, …,]
  primary key (columna1 [,columna2, …]));
```

Los tipos de datos más comunes son:

```
ascii, bigint, blob, counter (valores enteros automáticos), decimal,
double, float, int, list, map, set, text, timestamp, uuid
(identificadores únicos), timeuuid (ídem, pero usando el tiempo),
varchar
```

4. Insertar datos en una tabla

La inserción de datos se hace de manera similar a la de una base de datos SQL. La instrucción es:

```
insert into [nombreEspacio.]nombreTabla (columna1 [, columna2, …])
values (valor1[, valor2, …]);
```

5. Recuperar datos de una tabla

La recuperación de datos se hace de manera similar a la de una base de datos SQL. La instrucción es:

```
select [distinct | count(*) | *] listaColumnas
  from [nombreEspacio.]nombreTabla
  [where expresión [and expression …]]
  [order by columna [asc | desc] [, …]]
  [limit n]
  [allow filtering];
```

`expresión` puede ser una expresión de comparación o `in (valorClave1[, valorClave2, …])`

`allow filtering` se debe usar cuando la consulta puede ser muy costosa como en el caso de una consulta que involucra a una columna de una clave compuesta.

6. Creación de índices

Los índices se deben crear cuando se requiere recuperar valores de una columna que no forma parte de la clave de la tabla. La instrucción es:

```
create index nombreÍndice on [nombreEspacio.]nombreTabla (columna);
```

7. Agregar campos a tablas

Una vez que se ha creado una tabla se pueden agregar campos a la misma de manera similar a como se hace en SQL. La instrucción es:

```
alter table [nombreEspacio.]nombreTabla
{add columna tipoDatos | drop columna | alter columna type
tipoDatos};
```

8. Colecciones

Cassandra da la posibilidad de usar **colecciones** de datos por medio de tres estructuras que brinda para tal fin: **conjuntos, listas y mapas**. Los conjuntos son colecciones de valores únicos no ordenados, aunque cuando se consultan se entregan de manera ordenada. Las listas son colecciones en las que hay un orden en los elementos dado por su posición dentro de la lista, pudiendo haber valores repetidos. Los mapas (maps) son pares clave/valor, donde dada una clave se puede obtener el valor asociado. Se puede usar información asociada al tiempo como clave para asignarle un valor correspondiente.

a. Conjuntos- la declaración de un conjunto se hace como:

```
set <tipoDatos>
```

Se pueden usar los operadores + y – para agregar o quitar valores en un conjunto.

b. Lista- la declaración de una lista se hace como:

```
list <tipoDatos>
```

Se pueden usar los operadores + y – para agregar o quitar valores en una lista. Dependiendo del orden de los operandos el valor se agregará al inicio o al final de la lista. También se puede usar columna[índice] para agregar un valor en una posición determinada de la lista.

c. Mapas- la declaración de un mapa se hace como:

```
map <tipoDatos1, tipoDatos2>
```

`tipoDatos1` es el tipo de datos de la clave del mapa y `tipoDatos2` es el tipo de los valores. También se pueden usar los operadores + y – para agregar o quitar valores en un mapa.
En particular, se puede hacer una declaración: `map<timestamp, text>`, para asociar una marca de tiempo a un texto.

9. Cambiar datos en una tabla

Se utiliza la instrucción `update` para modificar valores en una tabla. La instrucción es:

```
update [nombreEspacio.]nombreTabla set columna1= valor [, columna2=
   valor, …]
where condición;
```

La `condición` sólo puede contener columnas que forman parte de la clave primaria.

10. Borrar datos en una tabla

Se utiliza la instrucción `delete` para eliminar filas en una tabla. También se pueden borrar campos de filas, sin necesidad de eliminar las filas completas. La instrucción es:

```
delete [columna1 [, columna2, …]]
from [nombreEspacio.]nombreTabla
where condición;
```

La `condición` sólo puede contener columnas que forman parte de la clave primaria.

11. Borrar una tabla

Se utiliza la instrucción `drop table` para eliminar una tabla completa. La instrucción es:

```
drop table nomTabla;
```

2.3.2 Ejemplo de uso de CQL

En esta sección se brinda un ejemplo de la creación, manipulación y consulta de una base de datos NoSQL empleando CQL. Se creará una tabla de empleados y se irán mostrando las distintas variantes de las instrucciones que se pueden usar para manipular la tabla.

1. Ejecutar el cqlsh: <Iniciar>, entrada DataStax Community Edition, entrada Cassandra SQL Shell, <Enter>. Alternativamenet se puede usar el IDE dado por DevCenter.
2. Crear el keyspace (equivalente al create database de SQL):

```
create keyspace ejemplo
with replication = { 'class' : 'SimpleStrategy',
   'replication_factor' : 1};
```

3. Usar el keyspace:

```
use ejemplo;
```

4. Crear una tabla:

```
create table emp (
  depId   int,
  empId   int,
  nombre  varchar,
  apellido varchar,
  primary key (depId, empId));
```

5. Insertar valores:

```
insert into emp (depId, empId, nombre, apellido)
  values (1, 1, 'Ana', 'Suarez');
insert into emp (depId, empId, nombre, apellido)
  values (1, 2, 'Roberto', 'Aldama');
insert into emp (depId, empId, nombre, apellido)
  values (2, 3, 'Cecilia', 'Campos');
insert into emp (depId, empId, nombre, apellido)
  values (2, 4, 'David', 'Flores');
insert into emp (depId, empId, nombre, apellido)
  values (3, 5, 'Ana', 'Macedo');
```

6. Recuperar valores:

```
select * from emp;
select * from emp where depid=1;
select * from emp where empid=1 allow filtering;
```

7. Crear un índice (para recuperar valores de un atributo que no es parte de la clave primaria):

```
create index on emp(nombre);
```

8. Recuperación valores usando el campo *nombre*:

```
select * from emp where nombre='Ana';
```

9. Agregar un campo tipo conjunto a la tabla:

```
alter table emp add tels set<text>;
```

10. Insertar datos en un conjunto:

```
update emp set tels={'11','12'} where depid=1 and empid=1;
update emp set tels= tels+{'13'} where depid=1 and empid=1;
```

11. Agregar un campo tipo lista a la tabla:

```
alter table emp add hijos list<text>;
```

12. Insertar datos en una lista:

```
update emp set hijos=['Samuel','Diana'] where depid=1 and empid=1;
```

13. Agregar un campo tipo mapa a la tabla:

```
alter table emp add cargos map<int,text>;
```

14. Insertar datos en un mapa:

```
update emp set cargos={1:'Asistente'} where depid=1 and empid=1;
update emp set cargos=cargos+{2:'Tecnico'} where depid=1 and
empid=1;
```

15. Otro ejemplo con un mapa:

```
alter table emp add sueldos map<timestamp,text>;
update emp set sueldos={'2017-01-02':'5000'} where depid=2 and
  empid=3;
```

16. Borrar datos de un campo de una tabla:

```
delete apellido from emp where depid=2 and empid=4;
```

2.3.3 Algunos aspectos adicionales

En esta sección se destacan algunos aspectos adicionales importantes tanto de Cassandra como de cql.

1. La clave primaria de una tabla puede estar constituida por más de una columna. Cuando esto sucede, la primera columna de la clave se conoce como **clave de partición (partition key)**. Esta clave se usa para distribuir las filas de la tabla entre los diversos nodos que puedan constituir un cluster[4] de la base de datos. Normalmente se asignaría cada valor de la clave de partición en un nodo distinto.

   Cuando hay más de una columna en la clave primaria, las demás columnas se conocen como de **agrupación (clustering columns)**. Los valores de estas columnas se almacenan de manera contigua junto con su clave de partición; esto hace que la recuperación de valores sea altamente eficiente, cuando la recuperación se hace por medio de la clave primaria.

   Adicionalmente, la clave de partición puede ser **compuesta (composite partition key)** lo cual significa que, a su vez, puede estar constituida por varias columnas. Esto significa que cada valor de la clave de partición compuesta (esto es, una fila de la tabla) puede estar almacenada en nodos distintos.

2. Una colección se recupera completamente cuando se hace una consulta, esto es, se recuperan todos los valores que forman a la colección. En consecuencia, las colecciones deberían usarse sólo cuando contienen pocos valores. Si una colección va a tener muchos valores, hay que pensar en otro tipo de modelado para que no haya problemas con cantidades grandes de datos. Cabe mencionar que los valores de un conjunto se recuperan en orden ascendente y los de una lista, por la posición que ocupan en la misma. En un mapa, el valor de un par clave/valor también se puede cambiar con update usando la notación: set columna[clave]= nuevoValor.

[4] En general, se usa el término **cluster** para hacer referencia a un conjunto de nodos.
En particular, en Cassandra se usa el término **Data Center** para hacer referencia a un conjunto de nodos relacionados, entendiendo por esto que están dedicados a algún tipo de proceso específico, por ejemplo: búsquedas, procesos analíticos o procesos transaccionales. En un cluster puede haber varios data centers.

3. Índices. Un índice sobre columnas que no forman parte de la clave de partición, permite recuperar datos de una manera eficiente ya que hace búsquedas más rápidas de los datos que están en las columnas indizadas.

   Dado que no se trata de una base de datos relacional, se recomienda que los índices se construyan sobre columnas que tienen valores para los cuales se pueden recuperar muchas filas que contienen los valores indizados, como podría ser el caso de autores de libros.

   Los casos principales en los que no se recomienda construir índices son aquellos en los cuales las columnas tienen un alta cardinalidad (muchos valores distintos), como podría ser el título de libros, ya que en cada consulta se regresarían pocos resultados; o en columnas que tienen muchas actualizaciones, como podría ser el caso del precio de las acciones que cotizan en bolsa, debido al elevado gasto que hay que hacer en el mantenimiento del índice.

4. Instrucción insert. Las filas que se insertan en una tabla pueden tener un tiempo de vida en las mismas, si al insertarlas así se indica. La variante a usar es la siguiente:

   ```
   insert into [nombreEspacio.]nombreTabla (columna1 [, columna2, …])
   values (valor1[, valor2, …])
   using TTL segundos;
   ```

   El tiempo de vida en la tabla de las filas insertadas con esta opción es el indicado en TTL. Después de ese tiempo, la fila ya no existiría en la tabla.

2.3.4 Instrucción select

A continuación se listan algunas cuestiones adicionales sobre la instrucción select.

1. La única función de agregados que acepta es la de contar:

   ```
   select count(*) from emp;
   ```

2. Se puede limitar la cantidad de filas que entrega una consulta:

   ```
   select * from emp limit 3;
   ```

3. Se pueden especificar rangos en una consulta:

   ```
   select * from emp
   where empid>=2 and empid<=4 allow filtering;
   ```

2.4 Modelo de datos agregados

- Es un modelo de datos que comparten tres de las cuatro categorías de las BD NoSQL: clave-valor, documentos y familia de columnas. La que tiene otro tipo de modelo es la de grafos.
- Las BD relacionales manejan conjuntos de tuplas, donde cada tupla es una lista de valores ordenados atómicos. Los datos agregados serían más como los objetos: permiten manejar estructuras de datos más complejas, las cuales pueden contener tuplas, listas, etc.
- Un dato agregado es una colección de objetos relacionados que se tratan como una unidad. En particular, es una unidad para el manejo de los datos y la administración de la consistencia.
- Se quiere que la actualización de los agregados se haga con operaciones atómicas y que la comunicación con el almacenamiento se haga en términos de agregados.
- Este modelado facilita el manejo de los datos en clusters, dado que los agregados forman una unidad natural para la duplicación y la partición horizontal (sharding) de los datos.
- Ventajas y desventajas
  - Si el acceso a los datos agregados se hace de la misma manera en que fueron almacenados, el acceso es muy rápido. En cambio, si el acceso se hace de otra manera (p. ej., acceder por productos), el acceso puede ser muy lento y puede requerirse de índices para acelerar las búsquedas.
  - Una gran ventaja de los datos agregados es que se pueden guardar "completos" en un nodo, de un cluster, lo cual facilita las tareas de acceso y de duplicación de los datos.

- Ejemplo
  - Se emplea un sistema que sirve para guardar órdenes de compras de productos.
  - Modelo relacional

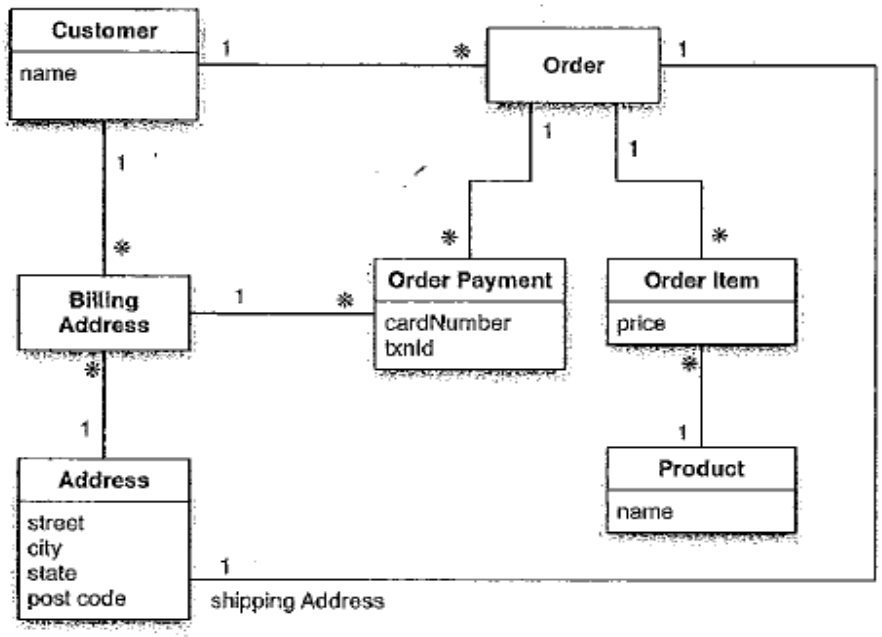

Figure 2.1 *Data model oriented around a relational database (using UML notation [Fowler UML])*

Customer

Id	Name
1	Martin

Order

Id	CustomerId	ShippingAddressId
99	1	77

Product

Id	Name
27	NoSQL Distilled

BillingAddress

Id	CustomerId	AddressId
55	1	77

OrderItem

Id	OrderId	ProductId	Price
100	99	27	32.45

Address

Id	City
77	Chicago

OrderPayment

Id	OrderId	CardNumber	BillingAddressId	txnId
33	99	1000-1000	55	abelif879rft

Figure 2.2 *Typical data using RDBMS data model*

- Modelo de datos agregados

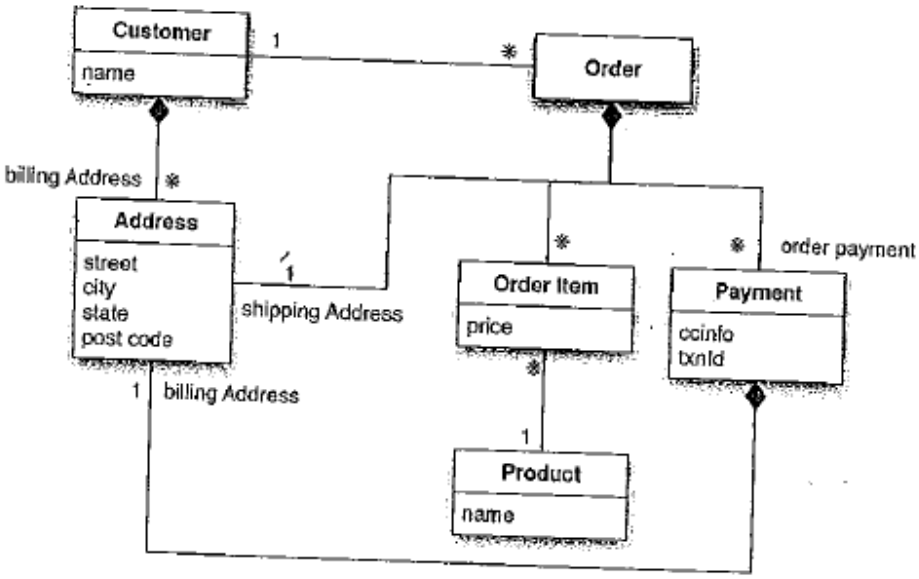

Figure 2.3 *An aggregate data model*

Again, we have some sample data, which we'll show in JSON format as that's a common representation for data in NoSQL land.

```
// in customers
{
"id":1,
"name":"Martin",
"billingAddress":[{"city":"Chicago"}]
}

// in orders
{
"id":99,
"customerId":1,
"orderItems":[
  {
  "productId":27,
  "price": 32.45,
  "productName": "NoSQL Distilled"
   }
  ],
"shippingAddress":[{"city":"Chicago"}]
"orderPayment":[
  {
    "ccinfo":"1000-1000-1000-1000",
    "txnId":"abelif879rft",
    "billingAddress": {"city": "Chicago"}
  }
  ],
}
```

- Otro tipo de modelado

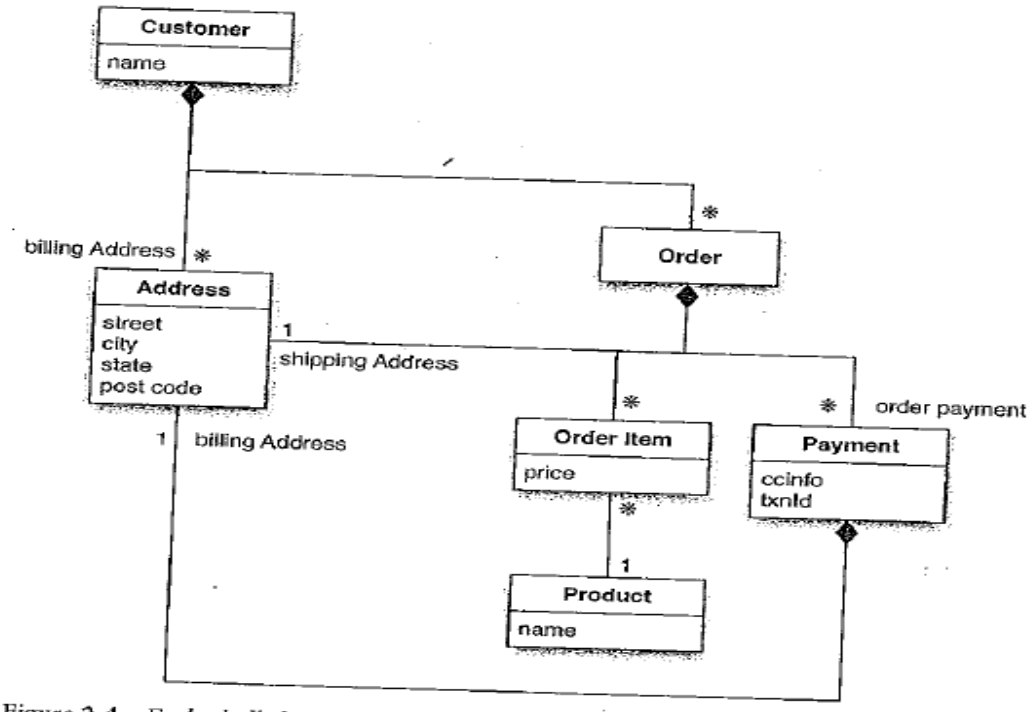

Figure 2.4 *Embed all the objects for customer and the customer's orders*

```
// in customers
{
"customer": {
"id": 1,
"name": "Martin",
"billingAddress": [{"city": "Chicago"}],
"orders": [
  {
    "id":99,
    "customerId":1,
    "orderItems":[
    {
    "productId":27,
    "price": 32.45,
    "productName": "NoSQL Distilled"
    }
  ],
  "shippingAddress":[{"city":"Chicago"}]

  "orderPayment":[
    {
    "ccinfo":"1000-1000-1000-1000",
    "txnId":"abelif879rft",
    "billingAddress": {"city": "Chicago"}
    }],
  }]
}
}
```

Bases de datos de documentos

- Entran dentro de la categoría de bases de datos NoSQL de clave-valor.
- Entre sus características principales están:
  - Almacenan datos con el concepto de datos agregados.
  - Ven a los datos como si tuvieran estructura.
  - Imponen límites en lo que puede ser guardado en ellas, definiendo estructuras y tipos aceptables.
  - Se puede hacer consultas basadas en los campos de los agregados.
  - También se pueden crear índices.

2.5 MongoDB

A continuación se muestra con un ejemplo el uso de las principales instrucciones para la creación y manejo de una base de datos en MongoDB.

El ejemplo es similar al de la sección 2.3.2 anterior, en la cual se ilustran las instrucciones equivalentes para Cassandra.

Cabe aclarar que las bases de datos en MongoDB son "sin esquema", esto es, no es necesario especificar explícitamente instrucciones del tipo *create table* para indicar la estructura de los datos que se almacenarán en las bases de datos; MongoDB se encarga de la creación y manejo interno de las mismas.

Antes de entrar al ejemplo, se dan en la siguiente tabla los términos equivalentes que se usan en SQL (BD's relacionales) y en MongoDB.

SQL	**MongoDB**
Base de datos	Base de datos
Tabla	Colección
Fila (o tupla)	Documento
Columna	Campo
Operación de junta	$lookup, docs. incrustados
rowid (clave única por el sistema)	_id
Índice	Índice

2.5.1 Ejemplo de uso de MongoDB

En esta sección se brinda un ejemplo de la creación, manipulación y consulta de una base de datos NoSQL empleando MongoDB. Se creará una colección de empleados y se irán mostrando las distintas variantes de las instrucciones que se pueden usar para manipularla.

1. Abrir una terminal del sistema y ejecutar el motor de la base de datos (después de haber instalado el software) dando `mongod`. Abrir otra terminal del sistema y dar (es como una interfaz de comandos): `mongo`.
2. Crear/usar la base de datos. Si la base no existe, se crea; si existe, se entra a su espacio:

```
use ejemplo;
```

3. Crear una colección. En MongoDB no tiene que crearse un esquema de "tabla" explícito; simplemente se da el nombre de la colección y MongoDB se encarga de manejar internamente la estructura:

```
db.createCollection("emp")
```

4. Insertar documentos (datos): simplemente se dan los datos en notación JSON (Java Script Object Notation). Se pueden usar los métodos *insertMany(…)* o *insertOne(…)*, de manera alterna, para insertar varios o un solo documento, respectivamente; en estos dos casos, ya no sería necesario usar *save*. Si se utiliza el primer método, hay que guardar los documentos especificándolos dentro de un arreglo:

```
a= {depId: 1,
    nomDep: "Compu",
    emps: [
      { empId: 1, nomEmp: "Ana", apEmp: "Suarez" },
      { empId: 2, nomEmp: "Roberto", apEmp: "Campos" }
    ]
   };
db.emp.save(a);

a= {depId: 2,
    nomDep: "Conta",
    emps: [
      { empId: 3, nomEmp: "Cecilia", apEmp: "Campos" },
      { empId: 4, nomEmp: "David", apEmp: "Flores" }
    ]
   };
db.emp.save(a);

a= {depId: 3,
    nomDep: "Admin",
    emps: [
      { empId: 5, nomEmp: "Ana", apEmp: "Macedo" }
     ]
   };
db.emp.save(a);
```

5. Recuperar valores. A continuación se muestran las diversas alternativas que se tienen para consultar la información en la base de datos (importan mayúsculas y minúsculas):

```
db.emp.find();                  //Obtener todos los empleados.
db.emp.find().pretty();         //Se ven más "legibles".

//Muestra campos explícitos. El 1 es true; 0 es false.
db.emp.find({},{emps:1});
db.emp.find({},{emps:1,_id:0});         //Ídem, sin el _id.

//Con una condición explícita.
db.emp.find({depId:3},{nomDep:1});

//Condición and.
db.emp.find({depId:1,nomDep:"Compu"},{emps:1});

//Condición or.
db.emp.find({$or:[{depId:1},{depId:2}]},{emps:1});

//Comparación: $gt, $gte, $lt, $lte, $ne
db.emp.find({depId:{$gt:2}},{emps:1});
db.emp.find({depId:{$gte:1,$lt:3}},{emps:1});

//Equivalente a like "%n%" (de sql). Se emplea la notación de punto
//para acceder a campos de un documento (hay que encerrar entre ").
db.emp.find({"emps.nomEmp":/n/});

//Equivalente a like "A%" (de sql).
db.emp.find({"emps.nomEmp":/^A/});

//Para ordenar ascendentemente (-1 para descendente).
db.emp.find({},{nomDep:1}).sort({nomDep:1});

//Otras.
db.emp.find().limit(1);
db.emp.find().count();
db.emp.find({depId:1}).count(); //Cant. de veces que aparece el
                                //depto. 1.
cemps= db.emp;          //Para usar la colección "emp" con "cemps".
cemps.find({"emps.empID": {$in: [3,4]}});    //Empls. Con id=3 o 4.
cemps.find({"emps.apEmp": "Campos"});

//Para hacer agrupaciones.
cemps.aggregate([{$group: {_id: "$nomDep",
                           total: {$count: $"emps"}}},
                 {$sort: {total: 1}}]);
//Opcionalmente se podría usar $match (antes de $group), de una
//manera parecida al where (de sql): {$match: {depID: 1}}
```

6. Crear un índice sobre un campo:

```
db.emp.createIndex({"emps.nomEmp":1});
```

7. Agregar un campo tipo fecha a la colección:

```
db.emp.updateMany({ },{$set:{fechaInicio: new Date()}});
```

8. Insertar datos en el nuevo campo:

```
db.emp.update({depId:1},{$set:{fechaInicio: "2018-01-01"}});
```

9. Agregar un campo tipo lista a la colección:

```
db.emp.updateMany({ },{$set:{lugares: ["A","B"]}});
```

10. Borrar datos de un campo de un documento de una colección:

```
db.emp.update({depId:1},{$set:{lugares:""}});
```

11. Borrar un campo de una colección:

```
db.emp.updateMany({},{$unset:{lugares:""}});
```

12. Borrar documentos de una colección:

```
db.emp.deleteMany({depId:1});   //También hay deleteOne.
```

2.5.2 Algunos operadores comunes de MongoDB

A continuación se listan algunos operadores comunes usados en mongo. La sintaxis completa debe buscarse en la documentación de mongo.

Actualización

Command	Description
$set	Sets the given field with the given value
$unset	Removes the field
$inc	Adds the given field by the given number
$pop	Removes the last (or first) element from an array
$push	Adds the value to an array
$pushAll	Adds all values to an array
$addToSet	Similar to push, but won't duplicate values
$pull	Removes matching value from an array
$pullAll	Removes all matching values from an array

Consultas

Command	Description
$regex	Match by any PCRE-compliant regular expression string (or just use the // delimiters as shown earlier)
$ne	Not equal to
$lt	Less than
$lte	Less than or equal to
$gt	Greater than
$gte	Greater than or equal to
$exists	Check for the existence of a field
$all	Match all elements in an array
$in	Match any elements in an array
$nin	Does not match any elements in an array
$elemMatch	Match all fields in an array of nested documents
$or	or
$nor	Not or
$size	Match array of given size
$mod	Modulus
$type	Match if field is a given datatype
$not	Negate the given operator check