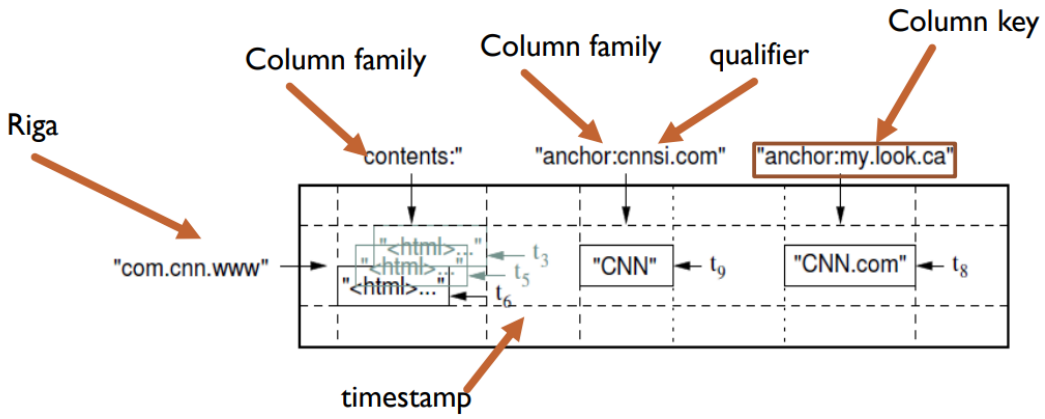
Lezione 5 27/03/2024

Per ora abbiamo visto aspetti modellistici, siamo partiti dal key value, modello utilizzato per sistemi che hanno bisogno di cashare i risultati in maniera condivisa (shopping cart di un sito di ecommerce).

Wide column store

I sistemi colonnari si possono considerare un'evoluzione dei key value.

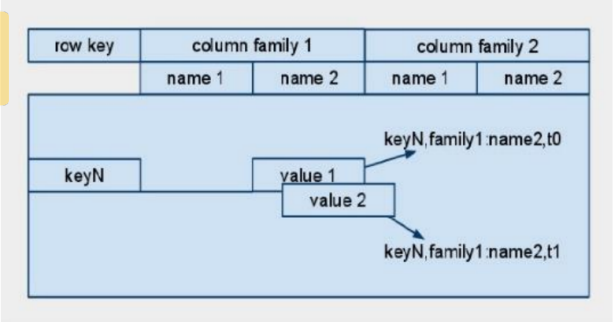
I **modelli colonnari** arricchiscono il concetto di valore, e al posto di mettere un solo valore, mettono un insieme di attributi. A differenza della tabella relazionale (key value) l'idea è che si acceda a questi dati per riga. Esiste un concetto di column family, ovvero un'insieme di attributi che hanno della caratteristiche di correlazione. Esiste poi la chiave e il timestamp (data di ultima modifica della riga, in questo modo posso anche tornare indietro nel tempo).

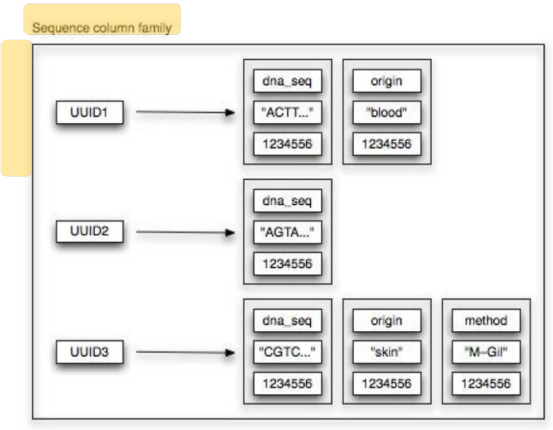


Una singola cella è raggiungibile tramite la chiave, la column family, il time stamp.

Esempio **schema free**:

- Table insieme di righe

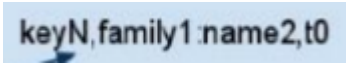




Possiamo vedere come nella seconda riga non c'è origin, mentre nella terza c'è anche il method. Metto solo quello che conosco, quello che non conosco non lo metto. Il vantaggio è la maggiore flessibilità, se devo aggiungere dei dati diversi dopo, quelli già presenti non vanno modificati.

Il problema è che lo schema dei dati non è conosciuto, queste 3 righe non mi garantiscono che la quarta non sia diversa.

Questo è come si accede ad un dato.

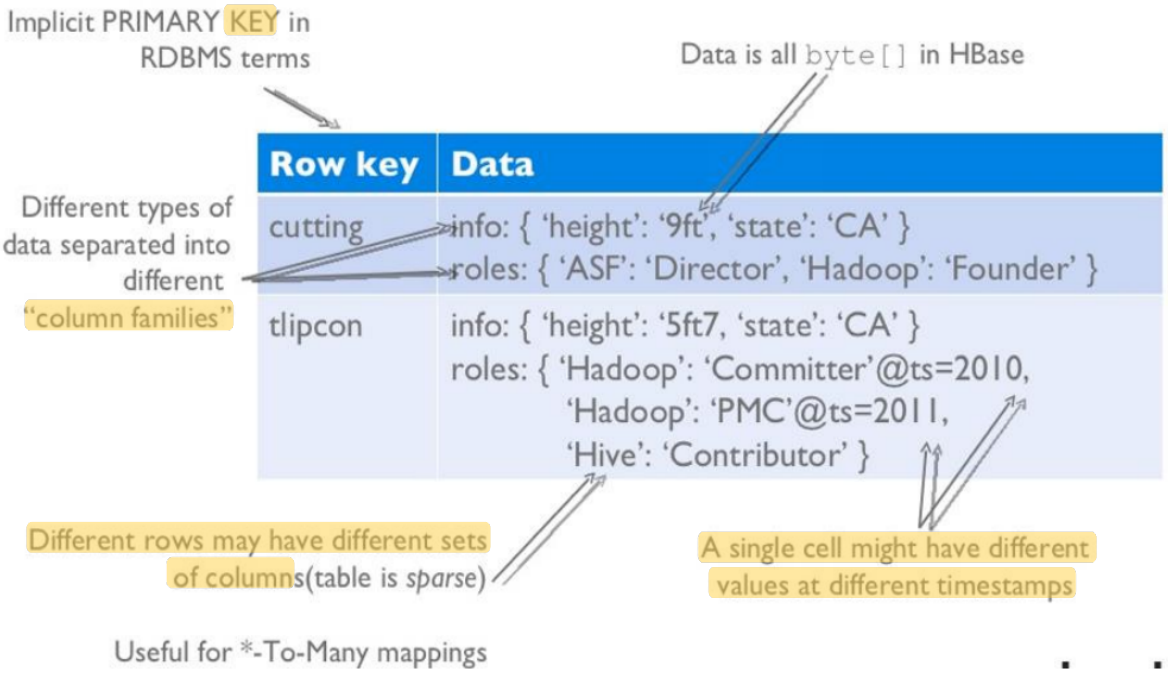


L'utilizzo dei timestamp permette anche di fare in modo che la transazioni con timestamp minore vengano eseguite prima di quelle che sono state create dopo, questo viene fatto con un protocollo.

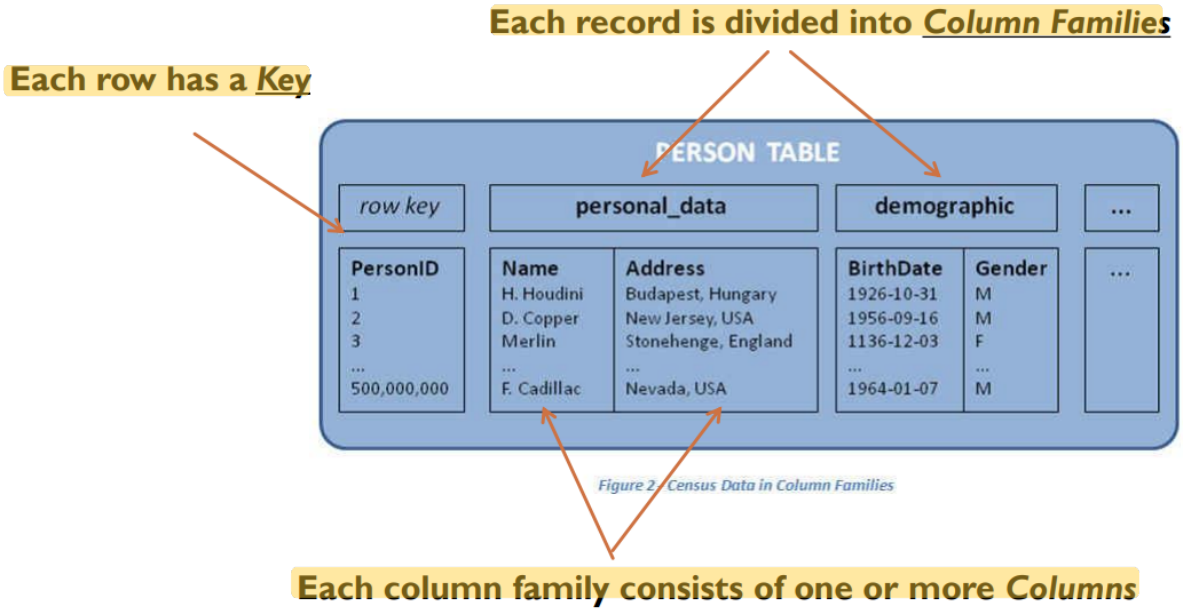
Hbase

è una versione pubblica di bigtable di google.

I dati sono divisi in tables, ogni table è composta di colonne a loro volta raggruppate in column_families. Una column_family può avere una o più column, e c'è il multi version (timestamp).



Altro esempio:

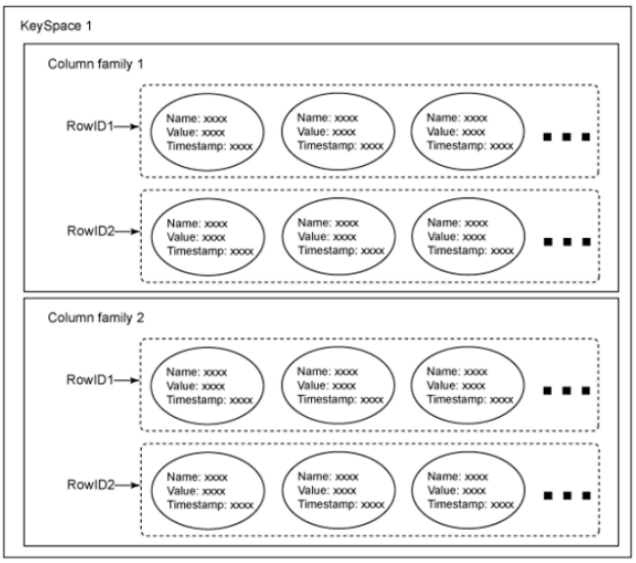


Non sono obbligato a mettere tutti i dati, ma alcune column family sono obbligatorie.

Cassandra

Questo è un modello simile, ha una nomenclatura diversa (usato da facebook).

L'idea è di avere una chiave, esiste uno spazio delle chiavi che rappresenta la "tabella relazionale", ci sono delle column_family che hanno delle chiavi, quindi per la stessa chiave ho più column family.

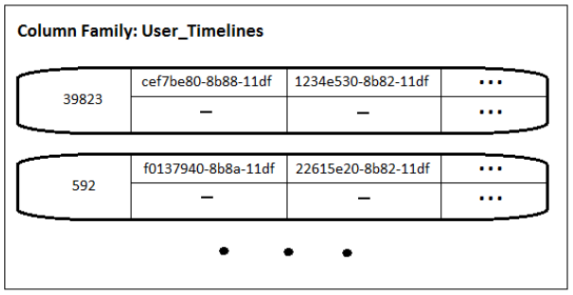


Una column family equivale ad una tabella. La stessa chiave della riga può essere presente in più column family.

Se prima ogni riga ha le column family, qua ogni column family ha le righe, è al contrario.

- Il valore di una riga è una serie di coppie key-value pairs cioè columns
- key = column name
- Ogni riga contiene almeno una colonna

- key: User ID
- column names memorizzano i valori dei tweet id
- Se i valori di tutte le column names sono uguali a "-" allora non sono usati



Cassandra ha un query language che è identico al modello relazionale.

Se so che accedo ai dati riga per riga, e difficilmente vorrò fare join tra righe, allora ogni riga è indipendente, e quindi posso frammentare i dati geograficamente come voglio. Questo è il vantaggio.

L'errore spesso fatto dagli studenti è il non dividere il modello dall'implementazione. Qui, a differenza del modello relazionale, il modello è spesso associato al prodotto. Ci sono alcuni aspetti che cambiano in base all'implementazione del prodotto (velocità), altri no.

Vediamo due famiglie di modelli, il modello documentare e il modello a grafo.

MongoDB

Il documento è come un json, è un albero che ha un nodo radice identificato da una chiave (_id) che punta ad un insieme di coppie nome valore, o un array di sottodocumenti (stessa cosa).

```
Contacts

{
     "_id": 2,
     "name": "Steven Jobs",
     "title": "VP, New Product
Development",
     "company": "Apple Computer",
     "phone": "408-996-1010",
     "address_id": 1
}
```

```
Addresses

{
    "_id": 1,
    "street": "10260 Bandley
Dr",
    "city": "Cupertino",
    "state": "CA",
    "zip_code": "95014",
    "country": "USA"
}
```

In questo esempio nel modello relazionale faremmo due tabelle, evitando anche di ripetere gli indirizzi. Quindi per sapere l'indirizzo di un contatto si fa una join, perché abbiamo l'id dell'indirizzo. Questo è **referencing**.

C'è un altro modo chiamato **embedding**, l'informazione non è più in due posti staccati ma è unita, mettendo un sottodocumento con l'indirizzo.

I modelli documentari usano questo metodo, gestiscono i dati come un'insieme di documenti.

Il vantaggio è che mettendo tutto insieme non sono necessari join, e c'è più divisione.

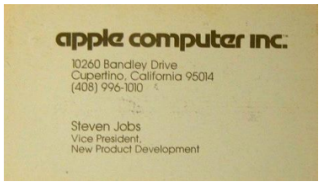
```
Contacts

{
     "_id": 2,
     "name": "Steven Jobs",
     "title": "VP, New Product
Development",
     "company": "Apple Computer",
     "address": {
     "street": "10260 Bandley Dr",
     "city": "Cupertino",
     "state": "CA",
     "zip_code": "95014",
     "country": "USA"
      },
      "phone": "408-996-1010"
}
```

Funziona bene quando accedo alle informazioni nei rami alti dell'albero spesso, perché per raggiungere i sottoalberi devo percorrere l'albero.

Infatti spesso per fare analisi dei dati, i dati vengono inseriti in un db diverso che funziona meglio.

```
{
    "name": "Steven Jobs",
    "title": "VP, New Product Development",
    "company": "Apple Computer",
    "address": {
    "street": "10260 Bandley Dr",
    "city": "Cupertino",
    "state": "CA",
    "zip_code": "95014"
    },
    "phone": "408-996-1010"
}
```




```
{
    "name": "Larry Page",
    "url": "http://google.com/",
    "title": "CEO",
    "company": "Google!",
    "email": "larry@google.com",
    "address": {
        "street": "555 Bryant, #106",
        "city": "Palo Alto",
        "state": "CA",
        "zip_code": "94301"
    }
    "phone": "650-618-1499",
    "fax": "650-330-0100"
}
```

Se poi il (in questo caso) biglietto da visita cambia, con questo modello posso tenere documenti che hanno dati diversi, ho flessibilità nello schema. Bisogna però avere delle metodologie di sviluppo di codice condiviso, per evitare per esempio di chiamare gli attributi in modo diverso, altrimenti la flessibilità causa un problema.

I tipi non esistono, la tipizzazione non è un problema del dbms.

MongoDB

RDBMS		MongoDB
Database	⇨	Database
Table, View	⇨	Collection
Row	⇨	Document (BSON)
Column	⇨	Field
Index	⇨	Index
Join	⇨	Embedded Document
Foreign Key	⇨	Reference
Partition	⇨	Shard

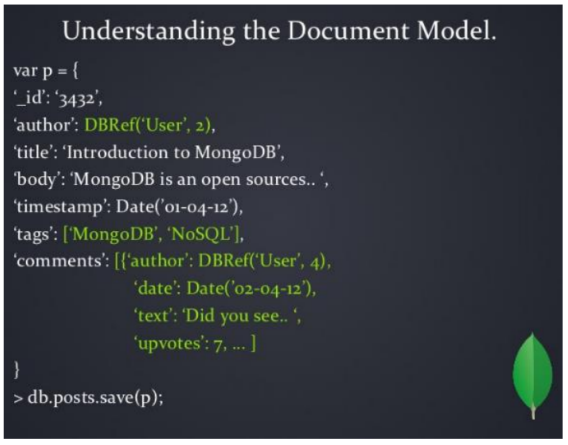
é un modello document based, i dati sono salvati in binario, gli accessi ai dati sono fatti tramite index.

MongoDB non è fatto per fare join, il produttore stesso chiede di non usarli perchè non sono efficienti.

Creazione e manipolazione dei dati

Usa la filosofia di non utilizzo di SQL.

```
db.CollectionName.insertOne({
    name:" Andrea" ,
    age: 26,
    Teach [«datawarehouse», «architecture»]
})
```



Il query language è simile ad un linguaggio di programmazione, simile a java.

Al posto di scrivere la query SQL e il codice che la esegue, si fa tutto insieme nella programmazione.

Se vanno caricati molti dati json al posto di uno solo, è possibile suggerire a mongodb di non restituire una risposta e di non salvare nei log, per salvare tempo. Questo è pericoloso ma veloce.

Questa sintassi è equivalente all'update dell'SQL. Al posto di dire where name = "Andrea", devo scrivere quello, perchè anche le query sono scritte in json, quindi $eq significa equal, perchè l'equal va trasformato in json, deve essere visto come un'istruzione.

upsert:true significa che se non trova nessuno che si chiama andrea, lo crea. è un "update or insert"

```
db.collectionName.updateAll(
{
    name:{$eq: «Andrea»},
    {$set: {Role: «PA»}},
    {$upsert:true}
}
)
```

Un delete cancella riga per riga, un drop cancella tutto.

- Delete one/all document that matchs a given query

```
db.CollectionName.deleteOne(
    { status=«D»}
)
db.CollectionName.deleteMany({})
Delete all documents not collection!
db.CollectionName.drop()
```

Query data

- db.CollectionName.find(jsonQuery,JsonSelectList).Function()
- Eg. Find all users that have more that 18 years old and show me the name and the address of the first 5

```
db.user.find({age:{$gt:18}},{name:1,address:1}).limits(5)
```

In sql

```
SELECT name, address
FROM user
WHERE age>18
LIMIT 5
```

gt sta per greater, >18. è l'unico parametro obbligatorio, in questo caso sono specificati anche gli attributi da mostrare, il :1 significa mostra solo quelli, tutti gli altri sono implicitamente a 0. Il limits(5) restituisce i top 5.

Operation	Syntax	Example	RDBMS Equivalent
Equality	{<key>:<value>}	db.mycol.find({"by":"tutorials point"}).pretty()	where by = 'tutorials point'
Less Than	{<key>:{$lt:<value>}}	db.mycol.find({"likes":{$lt:50}}).pretty()	where likes < 50
Less Than Equals	{<key>:{$lte:<value>}}	db.mycol.find({"likes":{$lte:50}}).pretty()	where likes <= 50
Greater Than	{<key>:{$gt:<value>}}	db.mycol.find({"likes":{$gt:50}}).pretty()	where likes > 50
Greater Than Equals	{<key>:{$gte:<value>}}	db.mycol.find({"likes":{$gte:50}}).pretty()	where likes >= 50
Not Equals	{<key>:{$ne:<value>}}	db.mycol.find({"likes":{$ne:50}}).pretty()	where likes != 50

- db.CollectionNam.find(

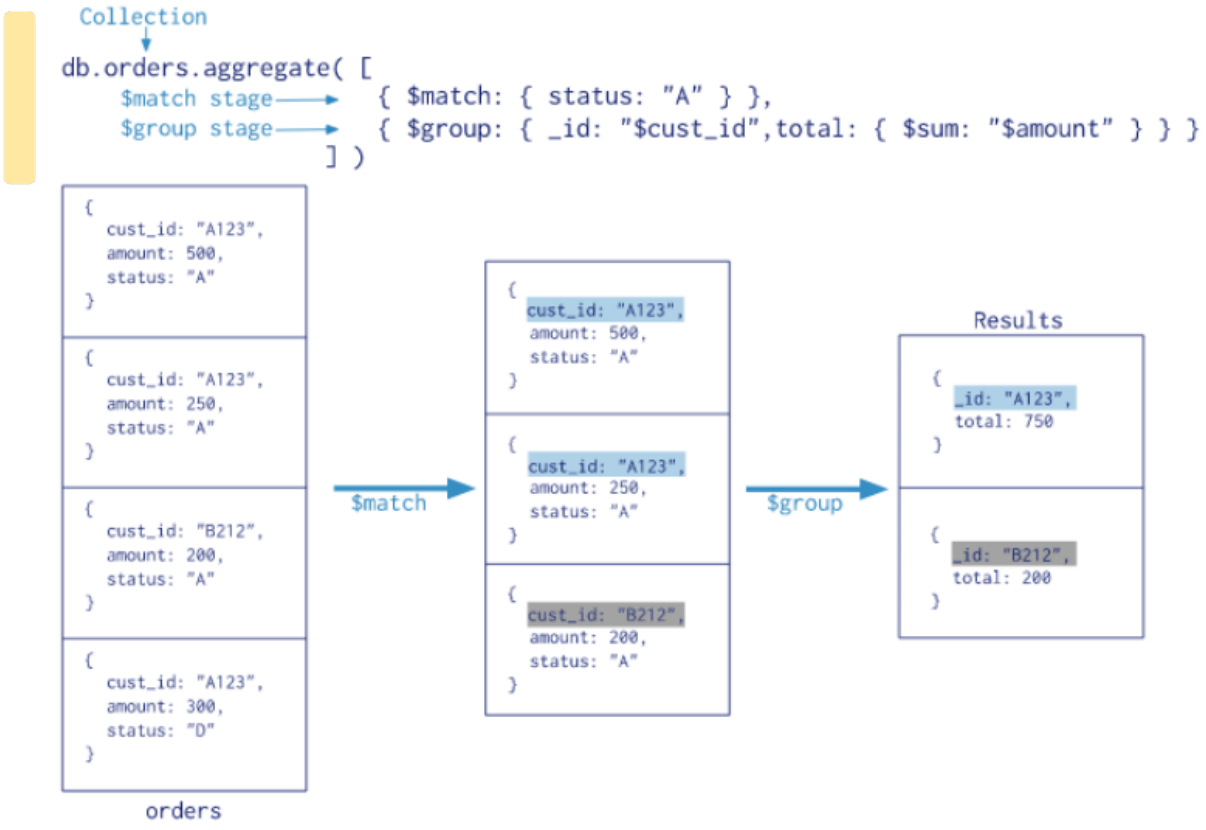
```
{ "likes": {$gt:10},
   $or: [{"by": "XXX"},
         {"title": "MongoDB books"}
         ]
}).pretty()
```

La virgola è un "and", bisogna specificare subito dopo che è un or invece.

Aggregation

Sono le query tipo le count, usate per l'analisi dei dati.

Una pipeline è una sequenza di operazioni eseguite una dopo l'altra, che permettono la creazione di analytics.



Per esempio qui prima si trovano tutti i documenti con lo status “A”, e poi vengono raggruppati per customer id e viene calcolato il totale (somma).

$lookup permette di fare un join con un’altra collezione, ma non è performante.

$unwind permette di srotolare un array in oggetti singoli

$out prende il risultato di una collezione e la scrive sul db senza restituirla

altri...