Big Data, NoSQL, MongoDB

Big Data architectures

Department of Information Engineering
Distributed Systems and Internet Technologies Lab
Via S. Marta 3 - 50139 Firenze, Italy

NoSQL

- Introdotto da Carlo Strozzi nel 1998 per indicare un database relazione che non utilizzava un'interfaccia SQL
- "non SQL", "non relational" o "not only SQL"
- Il termine è stato ripreso nel 2008 da Eric Evans (RackSpace) per indicare database non relazionali e distribuiti che non forniscono ACID (Atomicity, Consistency, Isolation, Durability)
- Top NoSQL: Aerospike, Amazon DynamoDB (AWS), Basho Technologies, Couchbase, DataStax, Google, IBM, MarkLogic, MapR Technologies, MongoDB, Microsoft Azure Cosmos DB, Neo Technology, Oracle, OrientDB, and Redis Labs

NoSQL vs SQL

I database tradizionali (**generalmente**) sono inadatti a gestire i Big Data

	SQL	NoSQL
Data Storage Model	Record salvati come righe in tabelle, come in un foglio di calcolo. Query complesse fra tabelle attraverso le JOIN	Variabile in base al tipo di database. Per esempio key-value, informazioni complesse come BLOB (value)
Schema	Struttura e tipo di dati sono fissi	Schemaless (dinamico). Si possono aggiungere nuovi campi *on the fly*, dati di tipo diverso possono coesistere
Scalabilità	Verticale (si incrementa la capacità dei singoli componenti del server), ma ci sono soluzioni clustered	Orizzontale, per aggiungere capacità di storage e calcolo basta aggiungere server, i dati sono distribuiti in modo automatico
Transazioni	Sì	Sì, in alcuni casi
Data manipulation	Select, Insert, and Update statements, e.g. SELECT fields FROM table WHERE…	Object-Oriented APIs
Consistency	Strong	Strong o eventual

Consistency Models

Consistency	
Strong	Tutti gli accessi sono visti da tutti i processi paralleli nello stesso ordine sequenziale
Eventual	Garantisce che, se non vengono fatti nuovi aggiornamenti a un dato, tutti gli accessi ad esso ritornano lo stesso valore
Weak	Tutti gli accessi alle variabili di sincronizzazione sono visti da tutti i processi nello stesso ordine sequenziale. Tutti gli altri accessi possono essere visti in ordine diverso. La sequenza di letture/scritture fra diverse operazioni di sincronizzazione è la stessa in ciascun processo
…	…

Perché NoSQL

- Elastic Scaling
  - RDBMS, scalabilità verticale (scale-up)
  - NoSQL, scalabilità orizzontale (scale-out)
- DBA Admins
  - RDBMS, la gestione e il monitoraggio dei database richiede personale esperto
  - NoSQL, è più semplice e facile da gestire, supporto automatico al ripristino dei dati
- Big Data
  - RDBMS, sono nati quando i Big Data non esistevano
  - NoSQL, progettati per i Big Data

Perché NoSQL

- Data Model flessibile
  - A differenza dei RDBMS, dove i cambiamenti allo schema sono operazioni tavolta onerose e complesse, hanno una struttura dati più semplice che permette cambiamenti senza riconfigurazioni
- Economicità
  - RDBMS, spesso necessitano di costosi server proprietari per la gestione dei dati
  - NoSQL, si può costruire un cluster di commodity server, con un costo finale per GB/s o transazione/s di molto inferiore

RDBMS (ACID) vs NoSQL (BASE)

Atomicity	**B**asically
Consistency	**A**vailable (CP)
Isolation	**S**oft-State (lo stato può cambiare nel tempo)
Durability	**E**ventually Consistent (propagazione asincrona)

ACID vs BASE

- **ACID (A. Reuter e T. Härder, 1983)**
  - *Atomicity*: la transazione è indivisibile, o avviene oppure no;
  - *Consistency*: ogni transazione porta il database in uno stato valido;
  - *Isolation*: ogni transazione è isolata dalle altre, si assicura la concorrenza;
  - *Durability*: una volta applicata una modifica al database, questa deve essere durevole (non-volatile memory);
- **BASE (E. A. Brewer, 2000)**
  - *Basically Available*, il database funziona, ma non necessariamente tutti i suoi elementi sono disponibili;
  - *Soft-State*, lo stato del sistema può cambiare nel tempo, senza intervento umano (a causa della consistenza eventuale);
  - *Eventually Consistent*, la consistenza è eventualmente assicurata;

Mercato NoSQL

Azienda	Prodotto	Versione
Aerospike	Aerospike	N/A
Amazon Web Services	Amazon DynamoDB and Amazon ElastiCache	N/A
Basho Technologies	Riak KV — NoSQL key-value database Riak TS — NoSQL time series database Riak S2 — large object store	Riak KV — 2.1.4 Riak TS — 1.3 Riak S2 — 2.1.2
Couchbase	Couchbase	4.0
DataStax	DataStax Enterprise	4.8
Google	Cloud Datastore	N/A
IBM	IBM Cloudant	N/A
MapR Technologies	MapR-DB (MapR Converged Data Platform)	5.1
MarkLogic	MarkLogic	8
Microsoft	Azure DocumentDB	N/A
MongoDB	MongoDB	3.2
Neo Technology	Neo4j	3.0
Oracle	Oracle NoSQL Database	4.0
OrientDB	OrientDB	2.2.0
Redis Labs	Redis Labs Enterprise Cluster	4.2

CHALLENGERS
LEADERS
Microsoft
Oracle
Amazon Web Services
SAP
IBM
MarkLogic
InterSystems
EnterpriseDB
DataStax
MapR
Google
NICHE PLAYERS
VISIONARIES
ABILITY TO EXECUTE
COMPLETENESS OF VISION
As of November 2017
© Gartner, Inc

Fonte: Gartner 2017

Cosa è cambiato?

- “Previously, we have used the market term "NoSQL" to imply nonrelational; NoSQL formerly implied alternative data types and scaling strategies from relational DBMSs. However, relational DBMSs have added, or are adding, features from NoSQL, while NoSQL DBMSs have added, or are in the process of adding, features from relational DBMSs. Therefore, the term "NoSQL" is no longer useful as a product distinction”.
- “While SQL is frequently associated with relational DBMS products, the availability of SQL does not define a relational DBMS. SQL is a data access language and can be used to access data in any DBMS, whether relational or nonrelational. Further, many of the nonrelational DBMSs have an SQL or SQL-like language (for example, MarkLogic and DataStax)”.

Fonte: Gartner 2017

NoSQL Leader

- **MongoDB**: è uscito dal Magic Quadrant ("did not meet the revenue requirements for inclusion"), ma è il più popolare database NoSQL. Open source, document-oriented. Oltre 2000 clienti (più della metà delle aziende Top 100 Fortune), facile da usare.
- **Amazon DynamoDB**: è il più popolare database NoSQL per il cloud. Storage su SSD, supporto ad applicazioni high-performace scale-driven, autosharding in base al workload. Facile da gestire e monitorare (API e AWS Management Console), integrato con Amazon EMR (possibilità di eseguire query indirizzate a datasource multipli), supporta sia modelli key-value che document, geospatial indexing.

NoSQL Leader

- **DataStax**, supporta la versione enterprise commerciale di Apache Cassandra (open source), geodistribuito fra i datacenter, wide-column store, distributed key-value database basato su Google Bigtable, fault tolerance, scale-out architecture, low-latency data access, analytics, search, monitoring, in-memory, security to support mission-critical applications.
- **MarkLogic**, è presente sul mercato con NoSQL da oltre un decennio, MarkLogic Server software commerciale, gira su AWS, Azure, e Google Cloud Platform, utilizzato in ambito business, mobile, Big Data, healthcare, real-time analytics, fraud detection, information discovery, content delivery, digital supply chain management

NoSQL Leader

- **IBM Cloudant**, IBM ha acquisito Cloudant (marzo 2014) per espandersi nel settore database-as-a-service (DBaaS), open source NoSQL document cloud database basato su Apache CouchDB. Forniscono hosting, administrative tools, analytics, e support per Cloudant, applicazioni in ambito industrial, financial services, gaming, produttori di mobile device, online learning, retail, healthcare
- **Couchbase**, Couchbase Server, open source NoSQL key-value e document database con built-in cache, utilizzato per appliaca zioni social e mobile, eCommerce, online gaming. Supporto per documenti, flexible data model, indexing, full-text search, MapReduce per real-time analytics. Clienti: AT&T, eBay, LinkedIn, McGraw Hill Education, Orbitz, Tesco

NoSQL Leader

- **Oracle’s NoSQL**, ACID transactions, geodistributed data, application security con autenticazione e session-level SSL encryption, integrazione con Oracle Database, Oracle Wallet, Hadoop. Key-value database con buone performance, scalabilità, sicurezza, high availability

NoSQL Leader

- **Google Cloud Datastore**, schemaless database con support a automatic sharding, high availability, ACID transactions, strong consistency, SQL-like queries, indici, durability for various types of workloads, MapReduce jobs. Ha una interfaccia RESTful. Cloud Bigtable generally disponibile da Giugno 2016, utilizzato per molti servizi Google (Search, Analytics, Maps, and Gmail). Gestisce massive workload con bassa latenza e alto throughput. Applicazioni in ambito IoT, user analytics, adtech, financial data analysis
- **Microsoft**, hanno presentato DocumentDB nel 2015. Bassa latenza, 99.99% availability service-level agreement, elastical scale throughput. Automatic schemaless indexing o indici secondary, supporto a query SQL e JavaScript, multidocument ACID transactions. È diventato Azure Cosmos DB (Maggio 2017).

Perché NoSQL?

- **Relational**
  - ACID (Atomicity, Consistency, Isolation, Durability) e JOIN (RDBMS)
- Per ottenere la scalabilità orizzontale bisogna rinunciare a qualcosa: i sistemi NoSQL rinunciano alle capacità relazionali
- **NoSQL**
  - **Key-value**, put, get, delete su chiave primaria
  - **Column-oriented**, tabelle ma senza JOIN, i dati sono immagazzinati in colonne, facilita le aggregazioni
  - **Document-oriented**, immagazzina i documenti come strutture JSON o XML, facili da gestire

- I database tradizionali (**generalmente**) sono inadatti a gestire i Big Data

Allora perché Facebook e Google non utilizzano esclusivamente NoSQL?

- Quando hanno iniziato il business nei Big Data, gli strumenti NoSQL non erano al grado di maturazione che presentano oggi
- Facebook (2009): 360M di utenti; (2018): 2.23B
- Hanno dovuto trovare soluzioni ad-hoc, investendo sul miglioramento delle soluzioni esistenti e implementandone di nuove

- Hanno sviluppato un distributed file system proprietario: Google File System (GFS)
- Continuano ad utilizzare SQL, MariaDB (fork di MySQL)
- Utilizzano BigTable (wide column store, 2d key-value store) come data store, ottimizzato per GFS
  - È il progetto da cui è derivato HBase
  - Utilizzato per Search, Analytics, Maps, Gmail
  - Scala fino a centinaia di PB automaticamente
  - È integrato con Hadoop e Google Cloud Platform
  - Supporta HBase API
  - Single-digit millisecond latency

facebook

Perché Facebook utilizza ancora SQL?

- Riprogettare un'infrastruttura esistente non è conveniente
- Hanno costruito architetture distribuite con supporto a SQL
- SQL e NoSQL da soli non risolvono tutti i problemi
- L'impegno nel settore NoSQL è comunque forte e in continua crescita

facebook

- Usano MySQL, ma con pesanti modifiche e personalizzazioni
  - Fork di MySQL (include MyRocks, uno storage engine RocksDB con MySQL)
    - https://github.com/facebook/mysql-5.6
    - https://github.com/facebook/mysql-8.0
- Usano RocksDB (key-value store)
  - Fork di LevelDB
    - http://rocksdb.org
    - https://github.com/facebook/rocksdb
- Usano TAO, distributed data store per il Social Graph
- Usano Hadoop per l'analisi dei dati, hanno sviluppato un tool
  - Presto (Distributed SQL Query Engine for Big Data) https://prestodb.io
  - Permette di eseguire query su Hive, Cassandra, RDBMS e data store, combinando i dati provenienti da datasource multipli
  - Eseguono 30,000 query/giorno sulla loro data warehouse (300 PB), 1 PB = 10^{15} byte
- Usano HBase (CP) su un cluster Hadoop

facebook

	Total Size	Technology	Bottlenecks
Facebook Graph	Single digit PB	MySQL, TAO	Random read IOPS
Facebook Messages, Time Series Data	Tens of PB	HBase, HDFS	Write IOPS, storage capacity
Facebook Photos	Hundreds of PB	Haystack (object-storage system)	Storage capacity
Data Warehouse	Hundreds of PB	Hive, HDFS, Hadoop	Storage capacity

Fonte: Facebook

- Utilizzato per
  - Small messages, message metadata (thread/message indices)
  - Search index
  - Large attachments in Haystack (photo store)
- 6 B messaggi/giorno
- 74 B operazioni/giorno
- Picco di 1.5 M operazioni/s
- 55% letture, 45% scritture
- Mediamente con una scrittura inseriscono 16 record
- Dati compressi con lzo
- Crescita di 8 TB/giorno

facebook

- Stonebraker/DeWitt from the DBMS community:
  - “Hadoop is a major step backwards”
- Benchmark mostrano che Hadoop/Hive è meno performante di un DBMS
  - http://database.cs.brown.edu/projects/mapreduce-vs-dbms/
  - Una query su Hive è 50 volte più lenta di una su DBMS

Perché usano Hadoop/Hive invece di DBMS in parallelo?
Perché non rimpiazzare il cluster Facebook (4000 nodi, 100 PB) con uno DBMS da 20 nodi?

- Hadoop/Hive è lento ma supporta un numero enorme di client concorrenti (100k)
  - Stanno lavorando per ridurre la latenza delle query
- Le prestazioni di un sistema Big Data non si misurano con la latenza delle query
  - Quanti dati possiamo immagazzinare (store) e reperire (query)? È il **Big** di Big Data
  - Quanti dati possiamo reperire in parallelo?

Teorema CAP

- Eric Brewer (2000, 2002): se non puoi limitare il numero di fallimenti e pretendi di servire tutte le richieste che sono dirette a qualunque server, allora non puoi essere consistente
- Si deve sempre rinunciare a qualcosa: consistenza, disponibilità, tolleranza ai guasti e riconfigurazione
  - **C**onsistency
  - **A**vailability
  - **P**artition Tolerance

Teorema CAP

- **C**onsistency
  - Tutte le repliche contengono gli stessi dati, che sono visti allo stesso modo da tutti i client
- **A**vailability
  - Ogni richiesta riceve una risposta, senza garanzia che il dato fornito sia quello più recente. Si può sempre leggere e scrivere
- **P**artition Tolerance
  - Il sistema rimane operativo in caso di malfunzionamento (e.g., partizionamento, problemi di rete)

SODDISFARE TUTTE QUESTE RICHIESTE È IMPOSSIBILE

Consistent – Available (CA)

Hanno problemi con il partizionamento, possono usare la replication

- **RDBMSs** => PostgreSQL, MySQL, ecc. (relational)
- **Vertica** (column-oriented)
- **Teradata Aster** (relational)
- **Greenplum** (relational)

Consistent – Partition Tolerant (CP)

Hanno problemi con l'availability nel gestire la consistenza fra nodi partizionati

- **BigTable** (column-oriented/tabular)
- **Hypertable** (column-oriented/tabular)
- **HBase** (column-oriented/tabular)
- **MongoDB** (document-oriented)
- **Terrastore** (document-oriented)
- **Redis** (key-value)
- **Scalaris** (key-value)
- **MemcacheDB** (key-value)
- **Berkeley DB** (key-value)

Available – Partition Tolerant (AP)

Raggiungono una *eventual consistency* attraverso la replication

- **Dynamo** (key-value)
- **Voldemort** (key-value)
- **Tokyo Cabinet** (key-value)
- **KAI** (key-value)
- **Cassandra** (column-oriented/tabular)
- **CouchDB** (document-oriented)
- **SimpleDB** (document-oriented)
- **Riak** (document-oriented)

- Sviluppato da 10gen nel 2007 => prima release open source nel 2009
- Database NoSQL, document-oriented
- Hash-based, schemaless database
- Ogni documento è provvisto di un identificativo (_id)
- Usa il formato BSON per immagazzinare i dati (basato su JSON, Binary-encoded serialization di un documento JSON)
- Scritto in C++
- Supporto per APIs (driver) per molti linguaggi (e.g., C, C++, C#, Java, Node.js, Perl, PHP, Python, ecc.)

- Secondary indexes
- Query language tramite API
- Atomic writes e fully-consistent reads
- Master-slave replication con failover automatico (replica set)
- Scalabilità orizzontale tramite range-based partitioning dei dati (sharding)
- No JOINs, no transactions
- Query relativamente semplici
- Focalizzato su Consistency e Partition Tolerance (CP)
- Auto-sharding, replication, high-availability, supporta Map/Reduce
- Supporta dati geospaziali
  - Spherical (latitude, longitude), flat (punti bidimensionali sul piano)
  - Indici geospaziali

È il document-store più popolare

Rank			DBMS	Database Model	Score		
Sep 2020	Aug 2020	Sep 2019			Sep 2020	Aug 2020	Sep 2019
1.	1.	1.	MongoDB	Document, Multi-model	446.48	+2.92	+36.42
2.	2.	2.	Amazon DynamoDB	Multi-model	66.18	+1.43	+8.36
3.	3.	↑ 4.	Microsoft Azure Cosmos DB	Multi-model	31.67	+0.94	+0.80
4.	4.	↓ 3.	Couchbase	Document, Multi-model	30.60	+1.54	-0.69
5.	5.	5.	CouchDB	Document	17.25	-0.14	-1.15

Fonte: db-engines.com

È il quinto database più popolare, e il NoSQL più popolare

Rank Sep 2020	Rank Aug 2020	Rank Sep 2019	DBMS	Database Model	Score Sep 2020	Score Aug 2020	Score Sep 2019
1.	1.	1.	Oracle	Relational, Multi-model	1369.36	+14.21	+22.71
2.	2.	2.	MySQL	Relational, Multi-model	1264.25	+2.67	-14.83
3.	3.	3.	Microsoft SQL Server	Relational, Multi-model	1062.76	-13.12	-22.30
4.	4.	4.	PostgreSQL	Relational, Multi-model	542.29	+5.52	+60.04
5.	5.	5.	MongoDB	Document, Multi-model	446.48	+2.92	+36.42

Fonte: db-engines.com

Chi lo usa?

Gerarchia degli oggetti

- Un'**istanza** MongoDB può avere zero o più **database**
- Un ***database*** può avere zero o più ***collection***
- Una **collection** può avere zero o più **document**
- Un **document** può avere uno o più **field**
- Gli indici di MongoDB funzionano in modo analogo a quelli dei tradizionali sistemi RDBMS

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

- **Mongod**, è l’istanza del database
- **Mongos**, i processi di sharding
  - funziona come un database router: processa le richieste e decide quanti e quali istanze devono ricevere la query
  - mette insieme i risultati e li invia al client
- **Mongo**, una shell interattiva
- Struttura molto flessibile: si può avere un solo shard con più istanze di MongoDB, oppure uno shard locale per ciascun client (minimizza la latenza di rete)

- JSON, i dati sono organizzati come coppie chiave => valore

```
{
  '_id' : 1,
  'name' : { 'first' : 'Java', 'last' : 'C' },
  'contribs' : ['John', 'Sara', 'Paul', 'Lindsey'
],
  'awards' : [
    {
      'award' : 'W.W. McDowell Award',
      'year' : 1967,
      'by' : 'IEEE Computer Society'
    }, {
      'award' : 'Draper Prize',
      'year' : 1993,
      'by' : 'National Academy of
Engineering'
    }
  ]
}
```

CRUD

Create
- db.collection.insert(<document>) ← inserisce
- db.collection.save(<document>) ← aggiorna o inserisce
- db.collection.update(<query>, <update>, { upsert: true })

Read
- db.collection.find(<query>, <projection>)
- db.collection.findOne(<query>, <projection>)

Update
- db.collection.update(<query>, <update>, <options>)

Delete
- db.collection.remove(<query>, <justOne>)

Create

- **db.collection_name.insert(<document>)**
  - Non si specifica il campo _id, MongoDB genera una chiave unica
  - Es. *db.collection.insert({{type: “computer”, quantity: 10 })*
  - *db. collection.insert({_id: 10, type: “computer”, quantity: 1 })*
- **db.collection_name.update(<query>, <update>, { upsert: true })**
  - Aggiorna 1 o più record in una collection che soddisfa la query
- **db.collection_name.save(<document>)**
  - Aggiorna o crea un record

Read

- **db.collection.find(<query>, <projection>)**
  - Simile a una SELECT
    - *<query> where condition , <projection> fields in result set*
  - Esempio: *db.collection.find({parts: “computer”}).limit(5)*
  - Utilizza un cursore per gestire il result set
  - Si possono impostare limit, skip, sort
- **db.collection.findOne(<query>, <projection>)**

Update

- **db.collection_name.insert(<document>)**
  - Non si specifica il campo _id, MongoDB genera una chiave unica
  - Esempio *db. collection.insert({{type: “computer”, quantity: 10 })*
  - *db. collection.insert({_id: 10, type: “computer”, quantity: 1 })*
- **db.collection_name.save(<document>)**
  - Aggiorna o crea un record
- **db.collection_name.update(<query>, <update>, { upsert: true })**
  - Aggiorna 1 o più record in una collection che soddisfa la query
- **db.collection_name.findAndModify(<query>, <sort>, <update>,<new>, <fields>,<upsert>)**
  - Modifica record esistenti

Delete

- **db.collection_name.remove(<query>, <justone>)**
  - Cancella tutti i record che soddisfano il criterio
- *<justone>*, specifica di cancellare soltanto 1 record che soddisfa il criterio
- Esempio: *db.collection.remove(type: /^h/ }),* rimuove tutte le parti che iniziano con h
- *db. collection.remove(),* cancella tutti i documenti nella collection

Operatori

Nome	
$eq	Matches value that are equal to a specified value
$gt, $gte	Matches values that are greater than (or equal to a specified value
$lt, $lte	Matches values less than or (equal to) a specified value
$ne	Matches values that are not equal to a specified value
$in	Matches any of the values specified in an array
$nin	Matches none of the values specified in an array
$or	Joins query clauses with a logical OR returns all
$and	Join query clauses with a logical AND
$not	Inverts the effect of a query expression
$nor	Join query clauses with a logical NOR
$exists	Matches documents that have a specified field

Aggregation

Le operazioni di aggregazione processano i dati e restituiscono risultati, raggruppano valori a partire dai documenti e svolgono operazioni restituendo un risultato

MongoDB fornisce tre modi per eseguire un’aggregazione:

- **aggregation pipeline**
- **map-reduce function**
- **single purpose aggregation operations (count, distinct, group)**

Aggregation Pipeline

- Si basa sul concetto di data processing pipeline. I documenti passano attraverso una pipeline multi-stage che li trasforma in risultati aggregati
- Un pipeline stage di base fornisce dei filtri che operano come query e trasformazioni di documenti che modificano la forma del risultato
- Altri operatori di pipeline forniscono strumenti per il raggruppamento e l'ordinamento dei documenti per campo, e strumenti per aggregare il contenuto di array
- Un pipeline stage può usare operatori, per esempio per calcolare la media o eseguire la concatenazione di stringhe
- È il metodo più efficiente per l'aggregazione dei dati
- Funziona su sharded collection
- Può utilizzare indici per migliorare le prestazioni durante i vari stage, ed utilizza fasi di ottimizzazione interne

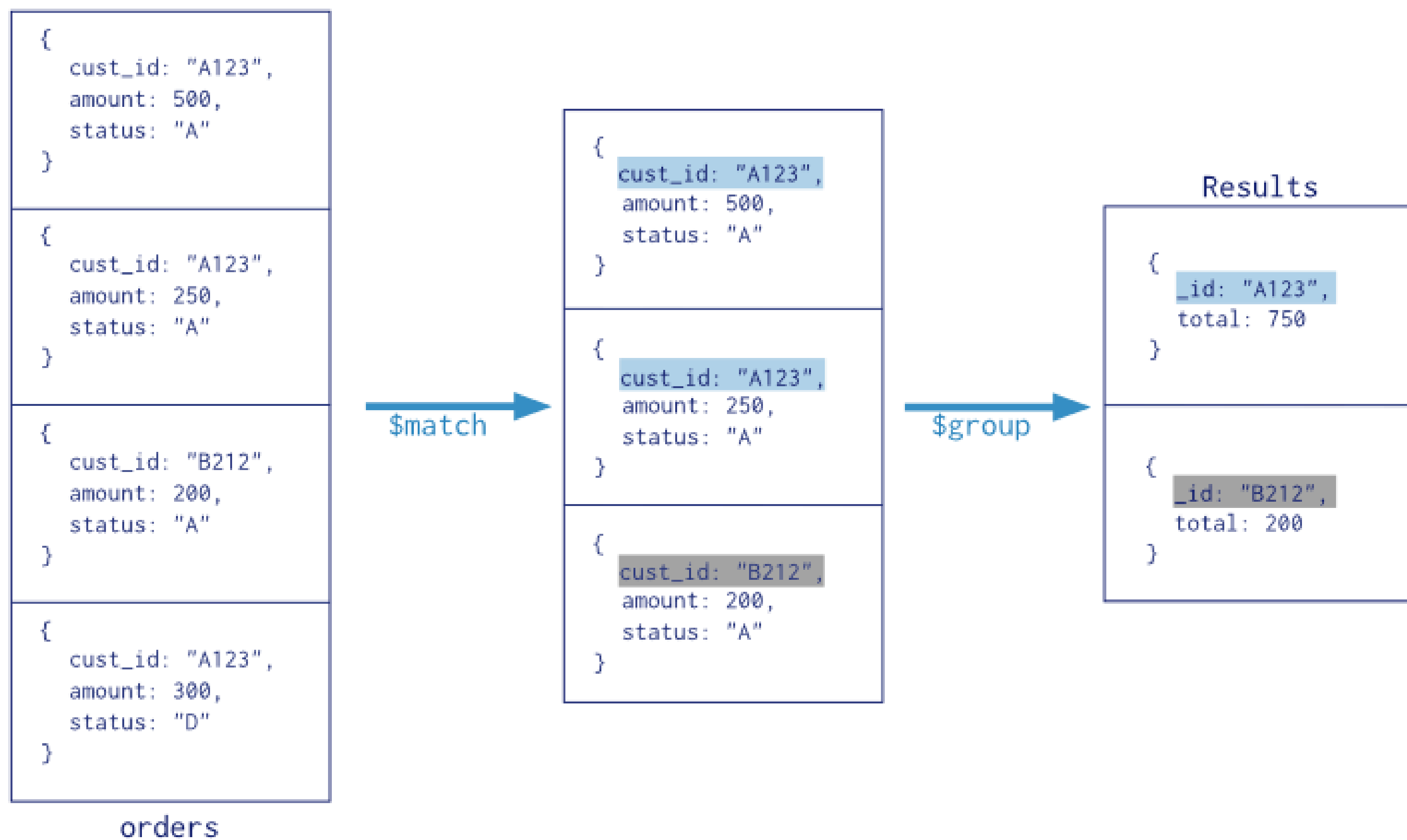
Aggregation Pipeline (esempio)

db.orders.aggregate([
{ $match: { status: “A” } },
{ $group: { _id: “$cust_id”, total: { $sum: “$amount” } } }
])

Match stage:
cerco i documenti con status pari ad “A”

Group stage:
Raggruppo i risultati per $cust_id e sommo l’amount di ciascuno al relativo totale

Aggregation Pipeline (esempio)



Map-Reduce

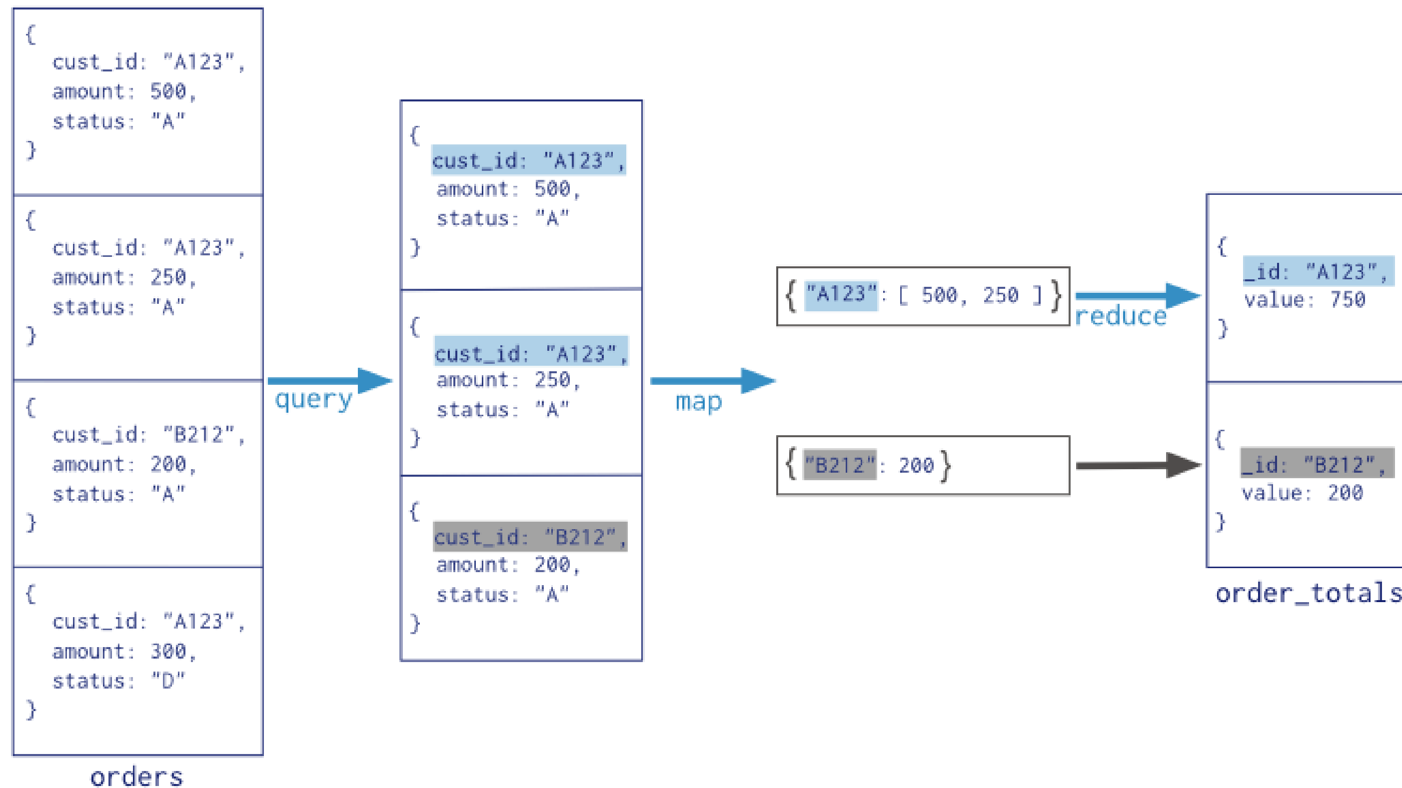
- Le operazioni di aggregazione tramite map-reduce consistono di due fasi
  - *map stage*, processa ogni documento e restituisce uno o più oggetti per ogni documento in ingresso
  - *reduce phase*, combina le uscite dell'operazione map
- Può includere un *finalize stage* per applicare altre modifiche ai risultati.
- Si può specificare una query per filtrare i documenti in ingresso come pure il sort o il limit dei risultati
- Utilizza funzioni JavaScript (eseguite dal processo mongod) per eseguire le operazioni e per la finalizzazione (opzionale) finale
- È meno efficiente e molto più complesso dell'aggregation pipeline

Map-Reduce (esempio)

Map → function() { emit(this.cust_id, this.amount); },
Reduce → function(key, values) { return Array.sum(values) },
Query → query: { status: "A" },
Output → out: "order_totals"

```
db.orders.mapReduce(
    function() { emit( this.cust_id, this.amount ); },
    function(key, values) { return Array.sum( values ) },
    {
     query: { status: “A” },
     out: “order_totals”
}
)
```

Map-Reduce (esempio)



Single Purpose Aggregation Operations

- **db.collection.count()**
- **db.collection.group()**
- **db.collection.distinct()**
  - Queste operazioni aggregano i documenti da una singola collection, ma non dispongono della flessibilità e capacità di aggregazione dell’aggregation pipeline e del map-reduce

Indici

- B+ tree indexes
- Un indice è creato automaticamente per il campo _id (primary index)
- Servono per incrementare le performance delle query e per garantire unicità
- Supporto di indici semplici o *compound*, equivalenti ai *composite index* di SQL, (l’ordine conta)
  - Indici creati su campi di tipo *array* comportano la creazione di indici separati per ogni elemento dell’array
- Un indice contiene soltanto riferimenti ai documenti che hanno il campo indicizzato (sparse property)
- Diventano necessari quando la dimensione totale dei documenti supera la disponibilità di memoria RAM
- L’ottimizzatore delle query può utilizzare un solo indice durante il reperimento dei dati

Replication

- Assicura ridondanza, backup, automatic failover
- Avviene fra gruppi di server chiamati *replica set*
  - *Primary set*, insieme di server ai quali i client inviano update diretti
  - *Secondary set*, insieme di server usati per la duplicazione dei dati
    - (secondary only, hidden delayed, arbiters, non-voting)
  - Si possono avere al più 12 replica set
  - Se il primary set fallisce il secondary set vota per eleggere il nuovo primary set

Consistency

- Le operazioni eseguite sul primary set di una replica sono consistenti con l’ultima operazione di scrittura
- Le letture sul primary set hanno *strict consistency*, ovvero riflettono gli ultimi cambiamenti sui dati
- Le letture sul secondary set hanno *eventual consistency*, ovvero gli aggiornamenti si propagano gradualmente
- Se i client permettono letture da un secondary set allora il client può leggere uno stato precedente del database
- I fallimenti avvengono prima dell’aggiornamento dei nodi secondari
  - Il sistema identifica quando è necessario un rollback
  - Gli utenti sono responsabili di applicare manualmente i rollback

Normalization vs Denormalization

- *Normalization,* significa dividere i dati in collection multiple con reference fra le collection
- Ciascun dato sta in una collection, ma collection multiple possono fare riferimento ad esso
- Per cambiare i dati basta aggiornare un solo document
- MongoDB non consente di fare le JOIN, e quindi prendere i document da collection multiple richiede più di una query

Normalization vs Denormalization

- *Denormalization,* è l'opposto della normalization: embedding dei dati in un singolo document
- Invece di document che contengono reference a una copia dei dati, i document hanno ciascuno una copia dei dati
- Document multipli hanno bisogno di essere aggiornati se il dato cambia, ma tutti i dati relativi possono essere ottenuti con una singola query
- Tipicamente la normalization rende le scritture più veloci e la denormalization rende le letture più veloci. Bisogna trovare il trade-off migliore

Documents

- Al centro di MongoDB vi è il document: un insieme ordinato di chiavi con i valori associati
- MongoDB è type-sensitive e case-sensitive
- Per esempio, questi document sono distinti
  - {"foo" : 3} {"foo" : "3"}
- Così come questi
  - {"foo" : 3} {"Foo" : 3}
- Un document MongoDB non può contenere duplicate keys

Documents

- Key/value pairs nei document sono ordinate
  - {"x" : 1, "y" : 2} non è la stessa cosa di {"y" : 2, "x" : 1}
- L'ordine dei campi non importa, non è necessario progettare lo schema tenendone conto (MongoDB li può riordinare)

Collections

- Una collection è un insieme di document. Se un document è l'analogo MongoDB di una riga in un database relazionale, una collection può essere pensata come l'analogo di una tabella

Dynamic Schemas

- Le Collection hanno dynamic schema. Questo significa che i document in una singola collection possono avere varie “shapes”
- Per esempio, ciascuno di questi document può essere memorizzato in una collection
  - {"greeting" : "Hello, world!"} {"foo" : 5}

Multiple collections

- Dal momento che ogni document può essere messo in una collection, perché servono collection separate?
  - Mantenere diversi tipi di document nella stessa collection può essere scomodo per gli sviluppatori e gli amminstratori
  - È più veloce prendere una lista di collection che estrarre una lista dei tipi in una collection
  - Grouping dei document dello stesso tipo insieme nella stessa collection consente la data locality (si hanno meno disk seek)
  - Si impone una sorta di struttura nei document quando si creano gli indici. (Questo è in particular modo vero nel caso di unique indexes).
  - Questi indici sono definiti per collection. Inserendo solo document di un solo tipo nella stessa collection, possiamo indicizzare le nostre collection più efficientemente

Naming

- Una collection è identificata dal suo nome. I nomi delle collection possono essere qualunque stringa UTF-8, con alcune restrizioni
- La stringa vuota ("") non è un nome valido di collection
- I nomi delle collection non possono contenere il carattere \0 (null character) perché esso delimita la fine del nome di una collection
- Non si può creare una collection con il nome che comincia con “system”, un prefisso riservato per le internal collection. Per esempio, system.users collection contiene gli utenti del database, e system.namespaces collection contiene le informazioni su tutte le collection del database
- Le collection create dall’utente non devono contenere il carattere riservato $ nel nome. I vari driver disponibili per il database non supportano l’utilizzo del $ nel nome della collection, perché alcune collection di sistema lo contengono. Non si deve usare il $ nel nome della collection, a meno che non sia necessario accedere a una di queste collection di sistema

Subcollections

- Una convenzione per organizzare le collection è di usare le namespaced subcollections, separate dal carattere "."
- Per esempio, un'applicazione contenente un blog potrebbe avere una collection chiamata blog.posts e un'altra collection chiamata blog.authors
- Questo è soltanto per un motivo organizzativo, non c'è alcuna relazione fra la blog collection (non è nemmeno necessario che esista) e la sua "figlia"

Databases

- In aggiunta al grouping dei document per collection, MongoDB raggruppa le collection in database
- Una singola istanza di MongoDB può ospitare diversi database, ciascuno consistente di zero o più collection
- Un database ha i suoi permessi, e ciascun database è memorizzato in file separati sul disco
- Una buona regola è di memorizzare tutti i dati di una singola applicazione nello stesso database
- Database separati sono utili per memorizzare dati di diverse applicazioni o utenti sullo stesso server MongoDB

Databases

- I database sono identificati da nomi. I nomi possono essere qualsiasi stringa UTF-8, con le seguenti restrizioni
  - La stringa vuota ("") non è un nome di database valido
  - Un database non può contenere alcuno di questi caratteri: /, \, ., ", *, , , :, |, ?, $, (uno spazio singolo), o \0 (null character). In pratica, vanno utilizzati i caratteri alfanumerici ASCII.
  - I nomi dei database sono case-sensitive, anche su filesystem non-case-sensitive. Per semplificare, utilizzare caratteri minuscoli
  - I nomi dei database hanno un limite di 64 byte

Databases

- Ci sono molti altri nomi di database riservati, che si possono leggere ma che hanno una semantica particolare
- admin
  - Questo è il “root” database, in termini di autenticazione. Se un utente viene aggiunto al database admin, l’utente automaticamente eredita i permessi per tutti i database. Ci sono alcuni server-wide commands che possono essere eseguiti soltanto dall’admin database, come per esempio fare il listing di tutti i database o lo shut down del server

Databases

- local
  - Questo database non verrà mai replicato e può essere utilizzato per memorizzare qualsiasi collection che deve essere locale per un singolo server
- config
  - Quando MongoDB è utilizzato in una configurazione sharded, utilizza il config database per memorizzare le informazioni sugli shard

Namespaces

- Concatenando il nome di un database con quello di una collection si ottiene una fully qualified collection name, cioè un namespace
- Per esempio, se si utilizza la collection blog.posts nel database cms, il namespace di quella collection sarà cms.blog.posts
- I namespace hanno un limite di 121 byte in lunghezza e, in pratica, devono essere inferiori a 100 byte

Basic Data Types

- I document in MongoDB possono essere pensati come "JSON-like", dal momento che sono contettualmente simili a oggetti Javascript
- JSON non ha un tipo date, che rende difficoltoso lavorare con le date
- C'è un tipo number, ma soltanto uno—non c'è modo di differenziare fra float e integer, senza contare la distinzione fra numeri a 32-bit e 64-bit
- Non c'è modo di rappresentare altri tipi di uso commune, come espressioni regolari o funzioni

Basic Data Types

- MongoDB aggiunge il supporto per un numero di data type aggiuntivi, conservando la struttura essenzialmente key/value di JSON
- null
  - Può essere utilizzato per rappresentare sia un valore nullo che un campo non presente: {"x" : null}
- boolean
  - C’è un tipo boolean,che può essere utilizzato per valori true e false: {"x" : true}
- number
  - La shell di default utilizza numeri 64-bit floating point

Basic Data Types

- Per gli interi, si utilizzano le classi NumberInt o NumberLong, che rappresentano signed integer di 4-byte o 8-byte signed integers, rispettivamente
  - {"x" : NumberInt("3")} {"x" : NumberLong("3")}
- string
  - Ogni stringa di caratteri UTF-8 può essere rappresentata utilizzando il tipo stringa: {"x" : "foobar"}
- date
  - Le date sono memorizzate in millisecond dall'Unix epoch. Il time zone non è memorizzato: {"x" : new Date()}

Basic Data Types

- regular expression
  - Le query possono utilizzare espressioni regolari con la sintassi JavaScript: {"x" : /foobar/i}
- array
  - Set o list di valori possono essere rappresentati con array: {"x" : ["a", "b", "c"]}
- embedded document
  - I document possono contenere interi document embedded come valori in un document padre: {"x" : {"foo" : "bar"}}

Basic Data Types

- object id
  - Un object id è un 12-byte ID per i document
    - {"x" : ObjectId()}
- Ci sono altri tipi meno comuni:
  - binary data
    - Binary data è una stringa arbitraria di byte. Non può essere modificata dalla shell. Binary data è l’unico modo di salvare stringhe non-UTF-8 nel database
  - code
    - Le query e i document possono pure contenere codice JavaScript: {"x" : function() { /* ... */ }}

Dates

- In JavaScript, la classe Date è utilizzata come MongoDB date type. Quando si crea un oggetto Date, bisogna invocare new Date(...), non Date(...).
- Chiamando il costruttore come una funzione (cioè, non includendo new) ritorna una string representation della data, non un oggetto Date
- Non è una scelta di MongoDB; è il modo di funzionamento di JavaScript
- Le date nella shell sono visualizzate utilizzando il time zone locale. Sono memorizzate come millisecondi dall'Unix epoch, e non hanno informazioni sul time zone associate (il time zone può essere memorizzato in un'altra key)

Arrays

- Gli array sono valori che possono essere utilizzati per operazioni ordered (lists, stacks, queues) e unordered (set). Nel seguente document la chiave "things" ha per valore un array
  - {"things" : ["pie", 3.14]}
- Gli array possono contenere tipi di dati differenti come valori (in questo caso una stringa e un numero floating point)
- MongoDB "capisce" la loro struttura e sa eseguire operazioni all'interno di essi. Questo ci consente di fare query sugli array e costruire indici sul loro contenuto
- MongoDB ci consente pure ti fare atomic update che modificano il contenuto degli array

Embedded Documents

- I document possono essere utilizzati come valore per una key
- Si chiamano embedded document
- Gli embedded document possono essere utilizzati per organizzare i dati in un modo più naturale, rispetto a una struttura piatta di coppie key/value
- Per esempio, se abbiamo un document che rappresenta una persona e vogliamo memorizzare il suo indirizzo, possiamo annidare questa informazione in un document embedded "address"
  - { "name" : "John Doe", "address" : { "street" : "123 Park Street", "city" : "Anytown", "state" : "NY" } }
- Come per gli array, MongoDB "capisce" la struttura degli embedded document ed è capace di utilizzarli per costruire indici, fare query, o aggiornamenti

_id and ObjectIds

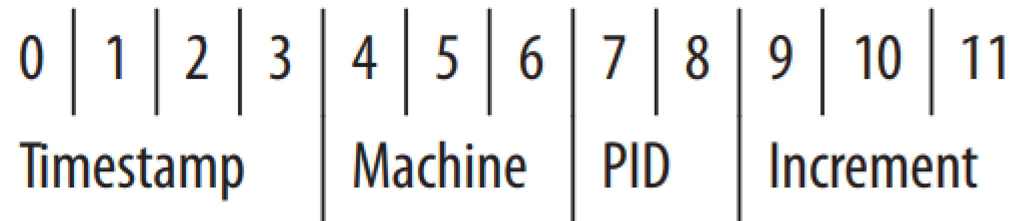
- Ogni document in MongoDB deve avere una "_id" key
- Il valore della "_id" key può essere di qualunque tipo, ma il default è ObjectId
- In una singola collection, ogni document deve avere un unico valore "_id", che assicura che ogni document nella collection può essere univocamente identificato
- Se si hanno due collection, ciascuna può avere un document con lo stesso valore per "_id"
- Nessuna collection può contenere più di un document con lo stesso "_id"

ObjectIds

- ObjectId è il default type per "_id“
- La classe ObjectId è progettata per essere leggera, e consente di generare id in modo univoco su tutte le machine
- La natura distribuita di MongoDB è la ragione principale per la quale si utilizzano ObjectId invece di qualcosa di più tradizionale, come chiavi primarie autoincrementing
  - È difficile e time-consuming sincronizzare autoincrementing primary key fra diversi server
- Poiché MongoDB è concepito come un database distribuito, è importante poter generare identificativi univoci in un ambiente sharded

ObjectIds

- I 12 byte di un ObjectId sono generate così



- I primi 4 byte di un ObjectId sono il timestamp in secondi dall'Unix epoch
- Il timestamp, quando combinato con i successivi 5 byte, fornisce univocità e granularità dell'ordine di un secondo

ObjectIds

- Poiché il current time è utilizzato negli ObjectId, si può pensare che i server debbano essere sincronizzati con NTP
- Sebbe sia una buona pratica, il timestamp attuale non è fondamentale per gli ObjectId; è importante invece che sia aggiornato ogni secondo e incrementato
- I successivi 3 byte di un ObjectId sono un identificativo univoco della macchina sulla quale viene generato. È solitamente un hash dell'hostname. Utilizzando questi byte, si è certi che macchine differenti non generano ObjectId che collidono

ObjectIds

- Per fornire univocità fra i diversi processi concorrenti che generano gli ObjectId su una singola macchina, i successivi due byte sono presi dal process identifier (PID) del processo che genera l'ObjectId
- I primi nove byte di un ObjectId garantiscono la sua univocità fra le macchine e i processi nell'ordine di un secondo
- Gli ultimi 3 byte sono l'incremento di un contatore che è responsabile per l'univocità entro un secondo di un processo. Questo consente di avere fino a 256^3 (16,777,216) ObjectId unici per processo in un secondo

Autogeneration of _id

- Se non c’è “_id" in un document al momento dell’inserimento, ne viene aggiunto automaticamente uno
- È gestito dal server MongoDB ma generalmente si fa dal client
- La decisione di eseguire la generazione dal client riflette la filosofia di MongoDB: i task devono essere eseguiti fuori dal server quando possibile
- Con database scalabili come MongoDB, è più facile fare scale out all’application layer piuttosto che al database layer. Spostare il lavoro sul client aiuta il database a scalare meglio

Running Scripts with the Shell

- Passare gli script da riga di commando:
  - $ mongo script1.js script2.js script3.js
    MongoDB shell version: 2.4.0
    connecting to: test
    I am script1.js
    I am script2.js
    I am script3.js
    $
- La shell mongo shell esegue ciascun script e poi esce

Running Scripts with the Shell

- Se si vuole eseguire uno script utilizzando come connessione un host non locale
  - $ mongo --quiet server-1:30000/foo script1.js script2.js script3.js
- Si possono eseguire script dalla shell interattiva utilizzando la funzione load()
  - > load("script1.js")

    I am script1.js

    >

Inserting and Saving Documents

- Le Insert sono il metodo base per aggiungere dati in MongoDB
- Per inserire un document in una collection, si utilizza il metodo insert
  - > db.foo.insert({"bar" : "baz"})
- Questo aggiunge una "_id" key al document (se non ce n’è già una) e la memorizza in MongoDB

Batch Insert

- Se si vogliono fare inserimenti multipli di documenti in una collection in modo veloce, si possono utilizzare i batch insert
- Batch insert permettono di passare un array al database
  -
```
> db.foo.batchInsert([{"_id" : 0}, {"_id" : 1}, {"_id" : 2}])
> db.foo.find()
{ "_id" : 0 }
{ "_id" : 1 }
{ "_id" : 2 }
```
- Se si devono inviare centinaia o migliaia di document i tempi si riducono molto

Batch Insert

- La versione corrente di MongoDB non accetta messaggi più lunghi di 48 MB, vi è un limite a quanto si può inserire con una batch insert
- Se si cerca di inserire più di 48 MB, molti driver dividono la batch insert in multipli di 48 MB
- Se si stanno importando dati con una batch insert e si ha un fallimento a metà della procedura, i document seguenti l’errore non vengono inseriti:
- > db.foo.batchInsert([{"_id" : 0}, {"_id" : 1}, {"_id" : 1}, {"_id" : 2}])
- Soltanto i primi due document vengono inseriti: non si possono inserire document con lo stesso "_id".

Batch Insert

- Se si vogliono ignorare gli errori e fare in modo che la batch insert cerchi di inserire il resto del batch, si può utilizzare l'opzione continueOnError
- Inserisce il primo, secondo, e quarto document dell'esempio precedente
- La shell non supporta questa opzione, ma i driver sì

Insert Validation

- MongoDB fa controlli minimi sui dati inseriti: controlla la struttura base del document e aggiunge il campo "_id" se non specificato
- Uno dei check è sulla dimensione: i document devono essere più piccoli di 16 MB
- È un limite arbitrario per prevenire bad schema design e assicurare performance adeguate

Removing Documents

- > db.foo.remove()
- Rimuove i document dalla collection
- Non rimuove la collection, i metadati di essa permangono
- Per rimuovere tutto dalla collection mailing.list collection dove il valore "optout" è true
  - > db.mailing.list.remove({"opt-out" : true})
- Una volta rimossi i dati, essi lo sono in modo definitivo. Non c'è modo di fare undo di document cancellati

Remove Speed

- Rimuovere i document è usualmente un'operazione veloce, ma se si vuole cancellare un'intera collection, è più veloce farne il drop (e poi ricreare eventualmente gli indici della collection vuota)
- Per esempio, supponiamo di inserire un milione di elementi dummy con la seguente
  - > for (var i = 0; i < 1000000; i++) { ... db.tester.insert({"foo": "bar", "baz": i, "z": 10 - i}) ... }

Remove Speed

- Rimuoviamo gli elementi inseriti misurando i tempi
- Un semplice remove
  - > var timeRemoves = function() { ... var start = (new Date()).getTime(); ... ... db.tester.remove(); ... db.findOne(); // makes sure the remove finishes before continuing ... ... var timeDiff = (new Date()).getTime() - start; ... print("Remove took: "+timeDiff+"ms"); ... }

    > timeRemoves()
- Su un MacBook Air, si ottiene “Remove took: 9676ms”

Remove Speed

- Se rimuoviamo utilizzando db.tester.drop(), il tempo si reduce a un millisecondo!
- Si tratta ovviamente di un miglioramento sensibile, ma non è possible specificare alcun criterio
- Si cancella l’intera collection, compresi tutti i suoi metadati

Document Replacement

- Il tipo più semplice di aggiornamento rimpiazza un document con un altro
- Utile per schema migration
- Per esempio, supponendo di fare major changes a un document
  - { "_id" : ObjectId("4b2b9f67a1f631733d917a7a"), "name" : "joe", "friends" : 32, "enemies" : 2 }

Document Replacement

- Si vuole spostare "friends" e "enemies" in un subdocument "relationships"
- Si cambia la struttura del document con un update
- > var joe = db.users.findOne({"name" : "joe"});
- > joe.relationships = {"friends" : joe.friends, "enemies" : joe.enemies}; { "friends" : 32, "enemies" : 2 }
- > joe.username = joe.name; "joe"
- > delete joe.friends; true
- > delete joe.enemies; true
- > delete joe.name; true
- > db.users.update({"name" : "joe"}, joe);

Using Modifiers

- Tipicamente soltanto certe porzioni di un document necessitano di aggiornamenti
- Si possono aggiornare i campi di un document con atomic update modifier
- Gli update modifier sono chiavi speciali che possono essere utilizzate per specificare operazioni complesse di update, come modifica, aggiunta, o rimozione di key, o modifica di array o embedded document

Using Modifiers

- Esempio
  - { "_id" : ObjectId("4b253b067525f35f94b60a31"), "url" : "www.example.com", "pageviews" : 52 }
- Ogni volta che si visita una pagina il modifier "$inc" incrementa la "pageviews" key
  - > db.analytics.update({"url" : "www.example.com"}, ... {"$inc" : {"pageviews" : 1}})
- Rifacendo la query pageviews è stato incrementato
  - > db.analytics.find() { "_id" : ObjectId("4b253b067525f35f94b60a31"), "url" : "www.example.com", "pageviews" : 53 }

Using Modifiers

- Il “$set” modifier
- "$set" imposta il valore del campo. Se il valore non esiste, viene creato
- Può essere utile per aggiornare lo schema o aggiungere chiavi
- Per esempio
  - > db.users.findOne() { "_id" : ObjectId("4b253b067525f35f94b60a31"), "name" : "joe", "age" : 30, "sex" : "male", "location" : "Wisconsin" }

Using Modifiers

- Utilizzando "$set“ per aggiungere un favorite book
  - > db.users.update({"_id" : ObjectId("4b253b067525f35f94b60a31")}, ... {"$set" : {"favorite book" : "War and Peace"}})
- Ora il document ha una “favorite book” key
  - > db.users.findOne() { "_id" : ObjectId("4b253b067525f35f94b60a31"), "name" : "joe", "age" : 30, "sex" : "male", "location" : "Wisconsin", "favorite book" : "War and Peace" }
- Si può fare l’"$unset“
  - > db.users.update({"name" : "joe"}, ... {"$unset" : {"favorite book" : 1}})

Array modifiers ($push)

- "$push" aggiunge elementialla fine dell’array se esiste, o ne crea uno nuovo se non esiste
  - > db.blog.posts.findOne() { "_id" : ObjectId("4b2d75476cc613d5ee930164"), "title" : "A blog post", "content" : "..." }
  - > db.blog.posts.update({"title" : "A blog post"}, ... {"$push" : {"comments" : ... {"name" : "joe", "email" : "joe@example.com", ... "content" : "nice post."}}})
  - > db.blog.posts.findOne() { "_id" : ObjectId("4b2d75476cc613d5ee930164"), "title" : "A blog post", "content" : "...", "comments" : [{ "name" : "joe", "email" : "joe@example.com", "content" : "nice post." }] }

Array modifiers ($pop)

- Ci sono molti modi per rimuovere elementi da un array. Se si tratta un array come una queue o uno stack, si può utilizzare "$pop", che rimuove elementi dalla cima o dalla coda
- {"$pop" : {"key" : 1}} rimuove un elemento dalla fine dell’array
- {"$pop" : {"key" : -1}} rimuove un elemento dall’inizio dell’array

Upserts

- Un upsert è un tipo speciale di update
- Se non si trovano document che fanno match con il criterio di update, viene creato un nuovo document combinando il criterio e i document aggiornati
- Se si trova un matching document, viene aggiornato normalmente
- Gli upsert sono utili perché eliminano il bisogno di fare il "seed" della collection: si utilizza lo stesso codice per creare e aggiornare i document
    - db.analytics.update({"url" : "/blog"}, {"$inc" : {"pageviews" : 1}}, true)

Updating Multiple Documents

- Gli update, di default, aggiornano soltanto il primo document trovato che fa match con il criterio
- Se vi sono più matching document, non vengono aggiornati
- Per modificare tutti i document, si passa true come quarto parametro di update
  - > db.users.update({"birthday" : "10/13/1978"}, ... {"$set" : {"gift" : "Happy Birthday!"}}, false, true)

Write Concern

- Write concern è un’impostazione del client utilizzata per descrivere come scrivere in sicurezza
- Di default, inserimenti, cancellazioni, e aggiornamenti aspettano una risposta dal database—la scrittura ha avuto successo o no?—prima di continuare
- Generalmente, i client lanciano un'eccezione
- Ci sono molte opzioni disponibili per calibrare con precisione il comportamento desiderato dell’applicazione

Write Concern

- I due write concern sono acknowledged e unacknowledged write
  - Acknowledged write sono il default: si ha una risposta che ci dice se il database ha processato con successo la scrittura
  - Unacknowledged write non ritornano alcuna risposta, non si sa se la scrittura ha avuto successo oppure no
- In generale, le applicazioni dovrebbero usare le acknowledged write
- Per dati di minore importanza (e.g., log o bulk data loading), si possono utilizzare unacknowledged write

Write Concern

- Un tipo di errore che è facile non intercettare quando si utilizza l’unacknowledged write è l’inserimento di dati non validi
- Per esempio, inserendone due con lo stesso "_id", la shell restituirà un’eccezione
  - \> db.foo.insert({"_id" : 1})
    \> db.foo.insert({"_id" : 1})
    E11000 duplicate key error index: test.foo.$_id_ dup key: { : 1.0 }
- Mentre con una “unacknowledged” write concern non si ha nessuna eccezione

Write Concern

- { w: <value>, j: <boolean>, wtimeout: <number> }
- l'opzione w serve a richiedere la conferma che l'operazione di scrittura si è propagata a un numero specificato di istanze mongod o a istanze mongod con tag specificati
- l'opzione j serve a richiedere la conferma che l'operazione di scrittura è stata scritta sul journal
- l'opzione wtimeout serve a specificare un limite di tempo per evitare che le operazioni di scrittura si blocchino a tempo indeterminato

Write Concern

- w: 1
  - Richiede il riconoscimento che l'operazione di scrittura si è propagata allo *standalone* mongod o al primario in un replica set. *w: 1* è il default write concern per MongoDB
- w: 0
  - Non richiede alcuna conferma dell'operazione di scrittura. Tuttavia, *w: 0* può restituire all'applicazione informazioni sulle eccezioni di socket e sugli errori di rete
  - Se si specifica *w: 0* ma si include *j: true*, il *j: true* prevale per richiedere il riconoscimento da parte del mongod standalone o il primario di un replica set

Specifying Which Keys to Return

- Per esempio, se interessano soltanto le chiavi "username" e "email"
  - > db.users.find({}, {"username" : 1, "email" : 1})

    { "_id" : ObjectId("4ba0f0dfd22aa494fd523620"), "username" : "joe", "email" : "joe@example.com" }

Query Conditionals

- "$lt", "$lte", "$gt", e "$gte" sono operatori di comparazione, corrispondenti a <,<=, >, and >=, rispettivamente. Possono essere combinati per cercare range di valori
- Per esempio, per trovare gli utenti fra 18 e 30 anni
  - `> db.users.find({"age" : {"$gte" : 18, "$lte" : 30}})`

Query Conditionals

- Questi tipi di range query sono utilizzate spesso con le date
- Per esempio, per trovare le persone che si sono registrate prima del 1 Gennaio 2020
- > start = **new** Date("01/01/2020")
- > db.users.find({"registered" : {"$lt" : start}})

Query Conditionals

- Se si vogliono trovare tutti gli utenti che non hanno l'username "joe"
  > db.users.find({"username" : {"$ne" : "joe"}})
- "$ne" può essere utilizzato con qualunque tipo di dato

OR Queries

- Ci sono due modi per fare una query OR in MongoDB. "$in" è utilizzato per una varietà di valori in una singola key. "$or" è più generale; si utilizza per fare query su qualunque valore fra key multiple
  - > db.raffle.find({"ticket_no" : {"$in" : [725, 542, 390]}})
  - > db.users.find({"user_id" : {"$in" : [12345, "joe"]}})
- L’opposto di "$in" è "$nin", che ritorna document che non danno match su nessun criterio nell’array
  - > db.raffle.find({"ticket_no" : {"$nin" : [725, 542, 390]}})

OR Queries

- Per trovare i document con "ticket_number" pari a 725 o "winner" pari a true si usa il "$or" conditional ("$or" utilizza un array di possibili criteri)
  - > db.raffle.find({"$or" : [{"ticket_no" : 725}, {"winner" : **true**}]})
- Se, per esempio, si vuole cercare uno qualunque fra tre "ticket_number" o la chiave "winner”
  - > db.raffle.find({"$or" : [{"ticket_number" : {"$in" : [725, 542, 390]}},{"winner" : **true**}]})

$not

- "$not" è un metaconditional: può essere applicato prima di ogni altro criterio. Per esempio, considerando l’operatore modulo, "$mod" ricerca le chiavi i cui valori, quando divisi per il primo valore dato, hanno un resto pari al secondo valore
  - > db.users.find({"id_num" : {"$mod" : [5, 1]}})
- La query precedente restituisce gli utenti con "id_num" 1, 6, 11, 16 ecc.
- Si può utilizzare "$not"
  - > db.users.find({"id_num" : {"$not" : {"$mod" : [5, 1]}}})

Conditionals and Modifiers

- Le chiavi $-prefisse sono utilizzate in posizioni differenti
- "$lt" è nell'inner document; negli update, "$inc" è la chiave dell'outer document
- I conditional sono nella inner document key, e i modifier sono sempre una chiave nell'outer document

Conditionals Semantics

- Si possono avere condizioni multiple su una singola key
- Per esempio, per trovare tutti gli utenti fra 20 e 30 anni, si utilizzano sia "$gt" che "$lt" sulla chiave "age"
- `> db.users.find({"age" : {"$lt" : 30, "$gt" : 20}})`

null

- null si comporta in modo più strano
- Si cercano document dove "y" key è null
  - > db.c.find({"y" : **null**})
    { "_id" : ObjectId("4ba0f0dfd22aa494fd523621"), "y" : **null** }
- Non fa match solo sui document con chiave pari a null, ma pure su quelli che non hanno tale chiave

Querying Arrays

- Se è necessario modificare più elementi di un array, si può utilizzare "$all". Questo consente di fare match di una lista di elementi.
- Si possono trovare i document con "apple" e “orange" utilizzando "$all"
  - > db.food.find({fruit : {$all : ["apple", " orange "]}})
    {"_id" : 1, "fruit" : ["apple", " orange ", "peach"]}
    {"_id" : 3, "fruit" : ["cherry", " orange ", "apple"]}

Querying Arrays

- Per cercare un elemento specifico di un array, si può specificare un indice con la sintassi *key.index*
- > db.food.find({"fruit.2" : "peach"})
- Gli array sono sempre 0-indexed, si cerca il match del terzo elemento dell’array con la stringa "peach"

Array and range query interactions

- Gli scalari (non-array elements) devono fare match nei document con ciascun criterio della query
- Per esempio, facendo una query con {"x" : {"$gt" : 10, "$lt" : 20}}, "x" deve essere più grande di 10 e inferiore a 20
- Se il campo "x" di un document è un array, il document fa match se c'è un element di "x" che fa match con ogni parte del criterio, ma qualunque clause può fare match con un diverso elemento dell'array

Querying on Embedded Documents

- Ci sono due modi di fare una query di un embedded document: cercare l'intero document o le sue coppie key/value
- Query dell'intero embedded document
  - {
    "name" : {
    "first" : "Joe",
    "last" : "Schmoe“
    },
    "age" : 45
    }

Security

- Ci sono problemi di sicurezza con JavaScript, relative all'esecuzione di funzioni Javascript. Si può fare il drop di un database
- > func = "function() {db.dropDatabase();}"

Cursors

- > **for**(i=0; i<100; i++) {

  ... db.collection.insert({x : i});

  ... }

  > **var** cursor = db.collection.find();
- Per iterare i risultati, si utilizza il metodo next() del cursore. Si può usare hasNext() per controllare se c'è un altro risultato
- > **while** (cursor.hasNext()) {

  ... obj = cursor.next();

  ... *// do stuff*

  ... }
- cursor.hasNext() controlla che il prossimo risultato esista, e cursor.next() ne fa il fetch

Indexing

- Un indice è simile all'indice di un libro. Invece di cercare l'intero libro si cerca in una lista ordinata che punta al contenuto, e consente di eseguire le query molto più velocemente
- Una query che non usa un indice si dice *table scan* (un termine derivato dai database relazionali), che significa che il server deve sfogliare l'intero libro per trovare i risultati
- Creazione di un indice sul filed username
  - > db.users.ensureIndex({"username" : 1})

Compound Indexes

- > db.users.find().sort({"age" : 1, "username" : 1})
- Ordina per "age" e poi per "username. Per ottimizzare il sort, si può fare un indice con "age" e "username"
- > db.users.ensureIndex({"age" : 1, "username" : 1})
- Si chiama *compound index* ed è utile se la query ha sort multipli
- Un compound index è un indice di più di un campo

Unique Indexes

- Gli *unique index* garantiscono che ogni valore appaia al più una volta nell'indice
- Per esempio, per avere "username" univoco
  - > db.users.ensureIndex({"username" : 1}, {"unique" : **true**})
- Con un compound unique index
  - > db.users.ensureIndex{"username" : 1, "age" : 1}, {"unique" : **true**})

Geospatial Indexing

- MongoDB ha geospatial index. I più comuni sono 2dsphere, per superfici sulla Terra, e 2d, per flat map (e time series data)
- 2dsphere consente di specificare punti, linee, e poligoni in formato GeoJSON. Un punto è dato da un array di due elementi, che rappresentano [*longitude*, *latitude*]
  - ```
    {
    "name" : "New York City",
    "loc" : {
    "type" : "Point",
    "coordinates" : [50, 2]
    }
    }
    ```

    ```

# Geospatial Indexing

- Una linea è data da un array di punti:
  - {
    "name" : "Hudson River",
    "loc" : {
    "type" : "Line",
    "coordinates" : [[0,1], [0,2], [1,2]]
    }
    }

# Geospatial Indexing

- Un poligono è specificato nello stesso modo di una linea (un array di punti), ma con un type diverso
  - ```
    {
    "name" : "New England",
    "loc" : {
    "type" : "Polygon",
    "coordinates" : [[0,1], [0,2], [1,2]]
    }
    }
    ```

    ```

# Geospatial Indexing

- Il campo "loc" può essere qualunque, ma i nomi dei campi dentro il suo sotto oggetto sono specificati da GeoJSON e non possono essere modificati
- Si può creare un geospatial index utilizzando il tipo "2dsphere" con ensureIndex:
- > db.world.ensureIndex({"loc" : "2dsphere"})

# Geospatial Queries

- Ci sono diversi tipi di geospatial query: intersection, within, e nearness
- Per esempio, per cercare i document che intersecano una location con l'operatore "$geoIntersects"
  - > **var** eastVillage = {
    ... "type" : "Polygon",
    ... "coordinates" : [
    ... [-73.9917900, 40.7264100],
    ... [-73.9917900, 40.7321400],
    ... [-73.9829300, 40.7321400],
    ... [-73.9829300, 40.7264100]
    ... ]}

# Geospatial Queries

- Per cercare i punti, le linee, i poligoni
- > db.map.find({"loc" : {"$geoIntersects" : {"$geometry" : eastVillage}}})

# 2D Indexes

- Per non-spherical maps (video game maps, time series data, ecc.) si usa un "2d“ index, invece di "2dsphere":
- > db.hyrule.ensureIndex({"tile" : "2d"})
- I "2d" index assumono una superficie perfettamente piatta, invece di una sfera. I "2d" index non dovrebbero essere utilizzati con sfere a meno che non sia importante la distorsione attorno ai poli

# GridFS

- GridFS è un meccanismo per memorizzare file di grandi dimensioni in MongoDB
- GridFS consente la memorizzazione di grandi file nella stessa cartella
- MongoDB alloca data files in chunk da 2 GB
- Performance basse: accedere ai file da MongoDB è più lento che da filesystem
- Si possono modificare soltanto i document cancellandoli e riscrivendoli interamente
- MongoDB memorizza i file come document multipli, non può fare il lock su tutti i chunk di un file allo stesso tempo

# mongofiles

- L'operazione di *put* prende un file dal filesystem e lo inserisce in GridFS; *list* elenca i file che sono stati aggiunti a GridFS; e *get* fa l'opposto di *put*: prende un file da GridFS e lo scrive nel filesystem
- mongofiles supporta altre due operazioni: ricerca per filename in GridFS e cancellazione di un file da GridFS

UNIVERSITÀ DEGLI STUDI FIRENZE
DINFO
DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

- È stato utilizzato per gestire i dati provenienti dalla rete FIRENZE WiFi
  - Eventi di connessione/sconnessione agli apparati di rete da parte dei dispositivi mobile
  - dati provenienti dal comune di Firenze in streaming e in forma anonimizzata
- > 240M di documenti
- I dati sono analizzati per capire come si muovono le persone in città
  - Dove si collegano
  - Per quanto tempo si collegano
  - Ogni quanto si collegano
  - Quante sono nei vari momenti della giornata
  - Quanti utenti nuovi arrivano giornalmente

FIRENZE WiFi

Global
Radius: 0.0008
Max Opacity: 0.80
Scale Radius:
Use Local Extrema:
APs:
WiFi Heatmap:

Leaflet | Map data © OpenStreetMap contributors, CC-BY-SA, Imagery © Mapbox

FIRENZE WiFi

Time Range: 2016-10-07 07:38:05 - 2016-10-07 08:38:05
Radius: 0.0008
Max Opacity: 0.80
Heatmap (5140) Streaming APs
Time Range:
5 min 10 min 15 min 30 min 60 min
3 hours 6 hours 9 hours 12 hours
1 day 3 days 1 week 1 month
Distinct people Local Extrema
Clustered markers
To Date static
Reply Speed:
Realtime 10 s 30 s 1 min 5 min
15 min 30 min 1 h 2 h 3 h

Leaflet | Map data © OpenStreetMap contributors, CC-BY-SA, Imagery © Mapbox