CS61 Lecture Notes [1] - 15 - NoSQL

Admin

Slides: 15 - NoSQL.ppt.pdf
SQL: FriendsOfFriends.sql

NoSQL Motivation

SLIDE 15–1

In NoSQL, “SQL” refers to traditional RDBMS’s. It arose due to the fact that for some systems we might want to use some other system. Thus, “Not Only SQL”.

RDBMS values

“Efficient, reliable, convenient, and safe multi-user storage of and access to massive amounts of persistent data”
– Widom

ACID

● Persistent data access
  ○ We want our data when we want it!
    ■ Archives, warehousing, or even this morning.
    ■ Analytics and BI
  ○ main memory is fast, but finite

● Concurrency
  ○ Never let the customer wait
  ○ internal as well as external

● Integration with business processes
  ○ large # of processes and applications need to share data.
  ○ resulting in a shared database environment
Some of these capabilities may be more than we need,
Some of these capabilities may prevent us from doing what we want!

**Impedance Mismatch**

Developers think in data structures that reside in memory.
These in-memory data structures are typically more flexible and more representative of the real world.

Example: The data making up a product order appears as an entity on the UI, but it really represents several entries in various RDBMS tables, such as “customers”, “orders”, “order items”, “creditcardinfo”, etc., with all the foreign keys and such to ties them together.

In memory it’s likely a single or a few data structures.

The issue is the requisite simplicity of RDBMS entries.

- the values in an RDBMS tuple must be simple - no vectors or structure or nested data structures.
- So, if it makes sense to use a richer in-memory data structure, you have to struggle sometimes to fit it into an RDBMS.
- Thus, the term impedance mismatch : two different representations requiring translation back and forth.

**The rise of Web Services**

There are other mismatches
* in the RDBMS world common schemas must match.
- If the customers table columns change then some queries and applications may fail.
- Indexes needed by one application might get in the way of another

Place a single application in front of the RDBMS and have it channel, convert, map, etc., requests to it. Only this one application has to know about the DB structure.

This moves interoperability requirements to the interfaces, and thus to the network.

**The deluge**

Next came the problem of the three V’s, leading to scalability challenges.

**Volume of data**

large data sets become difficult to use when stored in RDBMS
queries begin to take too long, joins get HUGE (“join pain”)

due to the underlying data model which tends to build a set of all possible answers to a query before filtering to get the right one.

**Velocity of data**
rate that data changes over time

rarely static - bursty

data store must be able to handle regular high-speed writes as well as bursts

the values of specific properties can change, sometimes quite rapidly

The structure of the data can vary over time as well, sometimes quite rapidly

causes

business needs can change quickly

the data being collected can also change quickly

**Variety of data**

data may sometimes be well structured, and unstructured other times

dense, sparse, connected or not

**Emergence**

This Led to the NoSQL (Not Only SQL) meaning a non-relational data store

NoSQL consistency models are more lenient

SO companies had to choose between bigger and bigger centralized computers, but at the time that only worked so far. Concerns about resilience arose, and the very real drive toward widely distributed, always-on, applications encouraged clusters of commodity hardware instead.

The RDBMS’s of the day were NOT designed to run on clusters … Some systems could handle writing to shared disk subsystems, but that wasn't always possible in widely distributed systems AND it didn't help resilience.

The web has led to applications that must easily store and process data which is bigger in volume, changes more rapidly, is often sparse across the domains, and is more structurally varied than traditional RDBMS can handle.

---

**CAP Theorem (Brewer 2000)**

Proven by Lynch and Gilbert in 2003

It is impossible for a web service to provide the following three guarantees
• Consistency
  The same definition we’ve been using. (like the ATM example)

• Availability
  Brewer means “if you can talk to a node in a cluster, the node can read and write data.”

• Partition-tolerance
  the cluster can survive comm link breaks that separate the cluster into multiple parts

  Slide 15–2

Example

Alice in London, Bob in Mumbai, both want to reserve a room at Hotel Zed in Paris

Reservation system has two nodes, one in London and the other in Mumbai, with a comm-link

There’s only one room left

Problems can arise

If there was only one server

  • It's consistent and available (if it's up)
  • It can’t be partitioned

This is the typical DB today

The two nodes (systems) have to agree on serializing their requests

* This fails if the link goes down

  • One node can be Primary, the other a Secondary
  
  • Requests from the Secondary must go thru the Primary
  
  • If the link goes down, we have a failure of Availability in Brewer’s terms since Alice can talk to London but can’t make a reservation
  
  • Both nodes can be allowed to continue to take reservations, but overbooking can result

Broken consistency

  • This is might be acceptable since there are usually “no-shows” to remedy any overbooking

(Optional) Other implications of relaxing ACID
Relaxing durability

- In-memory DB would have great performance, but all is lost after a shutdown
- Capturing telemetry at high speed might be more important than if you miss some data is the server crashes
- Replicated data might be lost if the Secondary sends it up to the Primary but the Primary crashes before replicating it back out to the others.

**ACID is nice, but it comes with a price**

**ACID vs. BASE (from Brewer)**

- Basic Availability: works most of the time. Trade-off is really latency rather than availability.
- Soft-state: Stores don’t have to be write-consistent, nor do different replicas have to be mutually consistent at all times.
- Eventually consistent: Stores exhibit consistency at some later point
  - Amazon example of ALWAYS being able to write to your shopping cart, even if you end up with several. On checkout just collect them all and let the customer decide.
  - e.g., lazily at read time).

Of course, this more lenient data store simply transfers to the program the responsibility of figuring out any data problems.

For example, what about durability? Surely this is inviolate?

* Consider in-memory DB’s that support extremely high-throughput apps. Perhaps a lazy replication of the data to disk store is sufficient to cover the transactions and if there’s a crash only a few transactions may be lost.
* strict durability systems depend on transaction logs. These can be slow without hardware assist, but using hardware allows for the possibility that the system will fail due to a power loss.

What kinds of applications are like/tolerate this?

* data acquisition systems - can afford to miss a data point
* ...

in high performance scanarios you may use hardware buffering to allow faster writes to

**NoSQL emergence led to another advancement: Polyglot Persistence**

Instead of always using an RDBMS, we choose a data storage mechanism based on the nature of the data being stored and how we want to use it.
As a result, an organization may employ a variety of persistent data stores. whichever one is best for the data and the circumstances.

### Primary reasons to consider NoSQL

1. ability to handle data access with sizes and performance that necessitate a cluster of machines.
2. improvement of productivity of app development by using a more convenient or intuitive data interaction style.
3. 3V's data, and including sparse data
4. desire for highly distributed systems running on commodity hardware

### Shift to aggregates

RDBMS tables cannot hold nested data. The table entries must be simple types and consistent (no variable types).

Nice, consistent, and enables RDBMS efficient operation, but limiting.

Aggregate orientation assumes that you sometimes want to operate on data with more complex structures than simple scalars of base types.

Aggregates also more closely match the in-memory data structures used by developers.

**on board example**: think of the Amazon order system

RDBMS would use several tables:
* customer,
* billing address,
* shipping address (variable, so must select from a table with a ShippingAddressID),
* Cart containing Items (each of which is a product with attributes),
* Each item has shipping preferences (speed, giftwrapping), and payment info
* payment info is ccard/giftcard/etc.

Show with multiplicities

An aggregate system might group them this way (JSON):

```json
// customer info collection
{
  "id":142,
  "fname":"charles",
  "mi":"c",
  "lname":"palmer",
  "billingaddress":{
    "streetaddr1":"2103 Xenon Way",
    "city":"Santa Fe",
    "St":"NM"
  }
}
// carts
```
Survey of NoSQL Databases

SLIDE 15–3

Key-Value and Document Store Databases

Primarily thought of as a collection of aggregates

Key-Value DB

SLIDE 15–4

the aggregate is opaque to the database - a blob

Store whatever you want - no structure.

Only access the blob via a key.

You can only search via a key - you cannot search within the blob. It's "opaque".
Very different from RDBMS, where ad hoc queries are allowed.

The application can store its data in a schema-less manner. It can be stored as a datatype of a programming language, such as an object. There is no fixed data model.

Fields may be added at will and may be nonuniform or even nested.

Fields may not be updated, however, only replaced.

They act like large, distributed hash maps where these usually opaque values are stored and retrieved by a key.

The Key space of the hash map is spread across multiple buckets on systems in the network.

Keys like Usernames, Email addresses, Lat-Long, SSN's, zip codes, etc. are all typical

for Fault tolerance you replicate the buckets on multiple systems.

The various machines are not full replicas of each other, to aid in load-balancing

Some systems do consistent hashing

If the target system is down for a WRITE, the hash is temporarily remapped and the data stored. When the system comes back up, the data is moved to where it should be.

If you don’t have the key, most of these DB’s won’t provide you with a listing

So you need to ensure you can get the key (or generate it), or it's based on a timestamp or other external info, or that users will always remember it ;-)

Typical uses

Storing session information

User preferences

Shopping Cart Data

Document Store DB

Example
First example is a row in a regular RDBMS

Second won’t fit into the same RDBMS since it has different attribute names

This isn’t a problem in a Document Store DB
* the database can see the structure in the aggregate.

A document is essentially a list of keys and values. Each field can have 0, 1, or many values

**Limits what you can put in it, but you gain flexibility in accessing it**

queries can be based on the fields within the document aggregate

there are no strict schemas of the document content - you can if you need to

You can search on any of the fields, and retrieval can be parts instead of the whole aggregate

indices can be built on those fields

**Typical Uses**

Event logging

lots of different formats, depending on what subsystem is making a log entry

Blogging platforms

Web analytics

it’s easy to update certain fields of a document as the information being monitored changes

page hit counts

network traffic analyses

**Column-Family Stores**

From Google’s “BigTable” 2006 paper

[static.googleusercontent.com—bigtable-osdi06.pdf](static.googleusercontent.com—bigtable-osdi06.pdf)

**Essentially a two-level aggregate store.**

A key is used to select a “row” which holds a map of links to the various aggregates
These second-level aggregates are referred to as “Columns”

Queries can pick out specific columns, such as GET (‘1234’, ’name’)

The Columns are gathered into “column families” with other columns that are often referenced together.

**Typical uses**

**Blogs**

Store blog entries with tags, categories, links, etc., in separate columns.

Comments can be in the same row, or place in a separate keyspace.

**Usage/license expiration**

A column can be set to “expire” in some of these systems, and when that happens it is removed from the DB.

**Shards**

Since these NoSQL data stores are focused on performance, it can be useful to distribute different parts of the data on different servers – sharding

Something in the key identifies which node will have the data

Each of those servers handles the reads and writes for that particular set of shards

Simple sharing can bring trouble if the server for a set of keys goes down

**The aggregate model offers a convenient way to chop up the data**

e.g., if certain aggregates are more likely to be accessed from Boston, place them in shards nearby

e.g., to balance the load on aggregates equally likely to be accessed, distribute them evenly across the servers.

**Replication may also be employed for performance and resilience**

Example: Lotus Notes
The Secure Distributed Storage idea arises here

Graph Databases

SLIDE 15–9
While the others were motivated by the need to run on clusters, Graph DB’s were motivated by a need for smaller records with complex and dynamic interconnections (relations).

While you can do similar things with RDBMS’s, as the relationships get increasingly complex, the joins required to find what you’re looking for leads to poor performance.

When we store a graph-like structure in a RDBMS, a single relationship isn’t too hard: “Manages”

As we need to add more relationships, it begins to get complicated

In the RDBMS, we essentially have to think of the traversals we will want, and then to design the RDBMS accordingly

If you want to add/remove relationships dynamically, well you can’t really.

Graph DB’s shift the bulk of the work of navigating the graph to INSERTs, leaving QUERYs as fast as possible.

QUERY is really just a very fast traversal of the graph from some starting point.

The relationship between nodes isn’t calculated during a QUERY, the relationship is always already there.

There is no limit to the number and kind of relationships a node can have

Relationships can be dynamically created / deleted

Some graph DB systems like Neo4j allow you to specify Java objects as a relationship, while others have simpler relationship capabilities

Example queries

SLIDE 15–10
“FIND AN EMPLOYEE OF BigCO WITH A FRIEND WHO LIKES THE SAME BOOK IN THE DATABASE CATEGORY THAT WAS WRITTEN BY TWO FRIENDS.”

“FIND THE BOOKS IN THE DATABASE CATEGORY THAT ARE WRITTEN BY SOMEONE WHOM A FRIEND OF MINE LIKES” → no match

Example vs. RDBMS
See this simple SQL script (in MySQLWorkbench)

**Query for BOB's friends:**

```
SELECT PersonFriend.friend
```

**Query for the inverse: who are friends with Bob? (directional)**

```
SELECT Person.person
```

**NOW we ask “Who are Alice’s friends of friends?”**

```
SELECT pf1.person as PERSON, pf3.person as FRIEND
```

**Typical uses are apps requiring lots of relationships and very fast QUERYs**

Connected Data, such as Social Networks or (like Facebook or tracking down members of criminal groups)

**Routing, dispatch, and location-based services**

relationships between nodes can have a “distance” property

Using distance and location properties in a graph of points of interest can enable an application to make recommendations of nearby services

**Recommendation engines**

which products are usually bought together

which products when bought together should raise an alarm (fertilizer bomb components)

searching for patterns in relationships that might indicate fraudulent transactions

**MongoDB Example**

### Mongo Shell

1. `mongo`
2. `use test`
3. `db.restaurant.find() // finds everything`
4. `db.restaurants.find().sort( { "borough": 1, "address.zipcode": 1 } )`
5. `db.restaurants.find({"name":"Juni"}) // find one`
6. `db.restaurants.update(`
   `{ "name": "Juni" },`
   `{ $set: { "cuisine": "American (New)" } },`
7. **//aggregation**
   db.restaurants.aggregate(
   [{ $group: { "_id": "$borough", "count": { $sum: 1 } } }
   ]); 
8. **Filter and group aggregates**
   db.restaurants.aggregate(
   [{ $match: { "borough": "Queens", "cuisine": "Brazilian" } },
   { $group: { "id": "$address.zipcode", "count": { $sum: 1 } } }
   ]);
   *The result has id field representing the zipcode group-by spec.*

### pymongo

```python
from pymongo import MongoClient
...
client = MongoClient() //create a collection on localhost
client = MongoClient("mongodb://mongodb0.example.net:27019")  // or somewhere else
...
db = client.test // assign db to database named test
// or
db = client['test']
...
from datetime import datetime
result = db.restaurants.insert_one(
   {
      "address": {
         "street": "2 Avenue",
         "zipcode": "10075",
         "building": "1480",
         "coord": [-73.9557413, 40.7720266]
      },
      "borough": "Manhattan",
      "cuisine": "Italian",
      "grades": [
         { "date": datetime.strptime("2014-10-01", "%Y-%m-%d"),
           "grade": "A",
           "score": 11
         },
         { "date": datetime.strptime("2014-01-16", "%Y-%m-%d"),
           "grade": "B",
           "score": 17
         }
      ],
      "name": "Vella",
      "restaurant_id": "41704620"
   }
)
```

using Python
MapReduce Framework

While NoSQL has its advantages, it does lack some of the nice things that the more mature RDBMS systems have. This means more responsibility falls to the programmer.

Came from Google, and Hadoop is the open source implementation of it.
See original paper: MapReduce: Simplified Data Processing on Large Clusters

designed for data analysis applications

no data model - data stored in files

User provides five key functions and the MR system provides the glue to make them work, and it adds fault tolerance

map, reduce, reader, writer, combine

General Flow
Input records into map, map produces $k_0 \ldots k_n$ key, value pairs. Each moves into a reduce that is handling those keys, and then reduce produces output records based upon the stream of values associated with the key.

**Map**

**SLIDE 15–12**

Divide and conquer!

divide analysis problem into subproblems

User creates *ONE* Map function that takes an input record and produces zero or more <key, value> pairs.

Typically separates out subproblems based on the key (with limited processing)

The key is not the same kind of key as in RDBMS’s… this key may have many values associated with it. In fact, that’s the point of MapReduce.

The framework will run the map function in multiple processes on one or more processors and systems.

**Example from Garcia-Molina**

Suppose we want to build an inverted index for a collection of documents $D$ each having a document id $i$. Thus, a document is identified as $D_i$.

We build the Map function to take a $D_i$ and read it character by character to find each word $w_j$. As the words are identified, the Map function emits the <key, value> pair: $<w, i>$.

Thus, the output of Map for a single $D_i$ is a stream of word and document id pairs $(w, [i_1, i_2, \ldots, i_n])$.

This will likely include duplicate pairs, but that can be handled later.

**Reduce**

**SLIDE 15–13**

User supplies one Reduce function that takes a set of values associated with a given key and produces zero or more outputs.

The framework will run multiple Reduce functions on one or more processors and systems.

**Example from Garcia-Molina (continued)**

The intermediate results from the Map functions are the pairs $(w, [i_1, i_2, \ldots, i_n])$.

The Reduce function should take a list of document ID’s, removes duplicates, and sorts the new list of now unique document ID’s.
Note that the Map function works on a single document, and the reduce works on a single word.

**Reader and Writer**

User creates READER and WRITER for reading records from disk and writing records to disk, respectively.

Reader extracts records from files, feeds the maps, and the reduce feeds into the writer functions.

**Combine**

The `combine()` works with MAP and takes a set of records for a given key to send a combined version of those <key, value> pairs to feed the reducer more efficiently. Sort of a pre-processor for Reduce.

**What the framework provides**

**SLIDE 15–14**

The user provides just those 4–5 functions and the framework provides the infrastructure to support all of this activity.

The framework distributes the pieces to multiple machines - everything is parallelizable by its very nature.

It also supports scalability by enabling adaptation to the addition of new machines.

Fault tolerance can recover reducer failures, etc.

**Here’s graphical example of the simple word counter**

**SLIDE 15–15**

**Example 2 from Garcia-Molina**

Suppose we want to modify the previous example to produce a word count across all the documents. That is, for each word $w$ that occurs in at least one document in the collection, we want the final output to be $(w, c)$, where $w$ is the word and $c$ is the total count of its occurrences across all the documents.

The Map function works the same, only emitting the <key, value> pair: $<w, 1>$ this time.

The Reduce function will now receive the pair $(w, [1, 1, \ldots, 1])$, with a 1 for each occurrence of the word $w$. The Reduce function simply adds up all the 1’s.

**Other capabilities**

You can even implement selection, projection.

**EXAMPLE: Web Log (Widom)**
record: Userid, URL, timestamp, addlinfo

task: count # of accesses for each domain

map: takes a record, look inside, and extract the URL, and emit
<domain, NULL>

reduce: (domain, list of NULL values representing one access to that domain) - so it just has to count how many NULL values it gets for each domain, and then emits <domain,count>

map(record) → <domain(key), NULL>

reduce(domain(key), list of NULLs) → <domain, count>

combiner takes the domain and list of nulls, and produces the <domain,count → summary for EACH Map function.

Insert combine() after Map

combine takes the output from Map and preprocesses it to do the counting there, producing a list of <domain, count> pairs which are sent on to reduce.

More efficient to count in the Map node.

Example MapReduce in JavaScript and MongoDB

From Isuru Suriarachchi’s Blog
SLIDE 15–16,20

Add on’s

HIVE (Schemas, SQL like language) and PIG (more imperative, but with relational operators)

Both compile to a workflow of Hadoop (MapReduce) jobs.

Both very popular

Examples of the various kinds of NoSQL Data Stores

For a list of the NoSQL databases of each type see
nosql.mypopescu.com—nosql
1. Key references used for this NoSQL lecture: Fowler, et.al., “NoSQL Distilled”; Robinson, et.al., “Graph Databases (Early Release)”; McCreary & Kelly, “Making Sense of NoSQL”. Other references include texts by Coronel, Widom, Ullman, Jukic, and Silberschatz.