Trends in NoSQL Technologies

Database Systems, CS386D
Instructor: Don Batory

Ankit, Prateek and Dheeraj
Agenda

Fundamental Concepts
Why NoSQL?
What is NoSQL?
NoSQL Taxonomy
Case Studies
Project Summary
References
Why NoSQL?

New Trends
Growth in data

"Big Data" + "Unstructured data"

Source: http://www.couchbase.com
Connectedness
"Social networks"

Source: http://http://tjm.org/
Architecture

"Concurrency"

Source: http://www.couchbase.com
Source: http://www.slideshare.net/thobe/nosql-for-dummies
What is the biggest data management problem driving your use of NoSQL in the coming year?

1. Flexibility (49%)
2. Scalability (35%)
3. Performance (29%)

Source: Couchbase NoSQL Survey, December 2011, n=1351
Extending the scope of RDBMs

Data Partitioning ("sharding")
+ Enables scalability
- Difficult process
- No-cross shard joins
- Schema management on every shard

Denormalizing
+ Increases speed
+ Provides flexibility to sharding
- Eliminates relational query benefits

Distributed Caching
+ Accelerated reads
+ Scale out: ability to serve larger number of requests
- Another tier to manage
RDMBS

Not a "One Shoe fits all" solution

**Oracle** has tried it

Need something different
Fundamental Concepts

ACID and CAP (A Quick Review)
**ACID**

**Atomicity** * **Consistency** * **Isolation** * **Durability**

Set of properties that guarantee that database transactions are processed reliably

Example:

Transfer of funds from one bank account to another (lower the FROM account and raise the TO account)
It is impossible in a for a distributed computer system to simultaneously provide all three of the following guarantees:

Cluster: A distributed network of nodes which acts as a gateway to the user

- **Consistency**
  - Data is consistent across all the nodes of the cluster.

- **Availability**
  - Ability to access cluster even if nodes in cluster go down.

- **Partition Tolerance**
  - Cluster continues to function even if there is a “partition” between two nodes.
What is different?

What is NoSQL database technology
Design Features

Data Model

No schema enforced by database - "Schemaless"

Four major categories

  Key/Value stores
  Document Stores
  Columnar stores
  Graph Databases
At the core all NoSQL databases are key/value systems, the difference is whether the database understands the value or not.
Key Value Stores - examples

- In memory / on disk
- Key/Value pair storage
- Transactional support
- Purpose: Lightweight DB

- Persistent In-memory
- Key/Value pair storage
- Provides special data structures
- pub/sub capabilities.
- Purpose: Caching and beyond

- In memory / on disk
- Key/Value pair storage
- Purpose: Caching

- Disk-based with built in memcache
- Cache refill on restart
- Highly Available (replication)
- Add/Remove live cluster
- Purpose: Caching
• DB understands values
• Data store in JSON/XML/BSON objects
• Secondary Indexes possible
• Schemaless
• Query on attributes inside values possible

Document Stores
MongoDB, Couchbase
Document Stores - examples

- memcached + couchDB
- Data stored as JSON objects
- Autosharding (replication)*
- Highly Available
- Create indexes, views.
- Query against indexes.
- Native support for map-reduce

- Data stored as BSON
- Very easy to get started
- Disk based with in-memory caching
- Auto-sharding*
- Supports Ad-hoc queries
- Native support for map-reduce.

* Auto-sharding - As system load changes, assignment of data to shards is rebalanced automatically
- DB understands values
- You don’t need to model all the columns required by your application upfront.
- Technically it’s a partitioned row store, where rows are organized into tables with a required primary key.

source: http://stackoverflow.com
Column oriented store - examples

- Open source clone to Google's BigTable
- Runs only on top of HDFS
- CP based system

- Modeled after Google's BigTable
- Clustered like Dynamo
- Good cross datacenter support
- Supports efficient queries on columns
- Eventually consistent
- AP based system
Graph Databases

Neo4J, GraphDB, Pregel

- Apply Graph Theory to the storage of information about the relationship between entries
- Used for recommendation engines.
Graph DB - example

- Disk-based system
- External caching required
- Nodes, relationships and paths
- Properties on nodes
- Complex query on relations

Wait for more in the Graph DB presentation!
In-house solutions

- No schema required before inserting data
- No schema change required to change data format
- Auto-sharding without application participation
- Distributed queries
- Integrated main memory caching
- Data Synchronization (multi-datacenter)
Case Studies

Amazon's Dynamo and Google's Bigtable
Google's BigTable
BigTable

Designed to scale.
   And the scale we are talking is of Petabytes!
A distributed store for managing *structured* data.
Three dimensional Table structure.
Uninterpreted bytes storage.

**CP** - Choses Consistency over Availability in the case of network partitioning (CAP theorem)

Basically, it is just a sparse, distributed, persistent sorted map store.
Sparse
  Most of the columns are empty

Persistent
  Data gets stored permanently in the disk

Sorted
  Data kept in hierarchical fashion
  Spatial Locality

Consistent

Features
  • sparse
  • distributed
  • scalable
  • persistent
  • sorted
  • consistent
  • map store
Tablet: BigTable's basic unit of storage

1. Rows
2. Column Families
3. Timestamps
Storage Hierarchy

Tablet

Metadata tablet

Root tablet

Chubby file
Optimizations

- Bloom Filters
- Caching

**Bloom Filter**: Drastically reduces the number of disk seeks required for read!
Optimizations

- Bloom Filters
- Caching

Higher Level Cache
- For scenarios where same data is read repeatedly

Lower Level Cache
- For spatial locality

Tablet Server

HLC

LLC

GFS

Google File System
Motivation:

“Customers should be able to view and add items to their shopping cart even if network routes are broken or data centers are being destroyed by tornadoes.”

**AP:** It chooses availability over consistency in the case of network partitioning.
Highly Available key-value
High performance (low latency)
Highly scalable (hundreds of nodes)
"Always on" available (esp. for writes)
Partition/Fault-tolerant

*Eventually* consistent
Key Techniques

Consistent Hashing
   For data partitioning, replicating and load balancing

Sloppy Quorums
   Boosts availability in present of failures
A Simple Example

Imagine that our consistent hash is mapped to a continuum of values.

All values are mapped to the continuum using some hash algorithm like MD5. This results in unpredictable assignments which can cause very imbalanced distribution of “key space”.

Adding a Node

!Important!
Adding a node does not cause the entire key-space to rebalance. This is very important to the implementation; adding nodes should not change all the answers, it should only "claim" key space from a single node.

Consistent Hashing
Dynamically add nodes

Consistent Hashing

Dynamically remove nodes

Important!
Removing a Node should cause the hash to rebalance such that only the now vacant key space is reclaimed. As with adding, removing a node should not remap the entire key space.

Improving Distribution

Virtual Keys
By calculating virtual keys we can decrease the standard deviation in key space for each node. The important factor here is making sure that the approach generating the virtual keys is not random and is reproducible.

Key 1’s Virtual Keys
Key 2’s Virtual Keys
Key 3’s Virtual Keys

Replication

N = 3

<table>
<thead>
<tr>
<th>Hash</th>
<th>Node</th>
<th>Replicas</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>A</td>
<td>B, C</td>
</tr>
<tr>
<td>12</td>
<td>B</td>
<td>C, D</td>
</tr>
<tr>
<td>19</td>
<td>B</td>
<td>C, D</td>
</tr>
<tr>
<td>20</td>
<td>C</td>
<td>D, E</td>
</tr>
<tr>
<td>37</td>
<td>D</td>
<td>E, F</td>
</tr>
<tr>
<td>40</td>
<td>E</td>
<td>F, A</td>
</tr>
<tr>
<td>54</td>
<td>F</td>
<td>A, B</td>
</tr>
</tbody>
</table>
- R number of nodes that need to participate in read
- W number of nodes that need to participate in write
- $R + W > N$ (a quorum system)

**Dynamo:**

$W = 1$ (Always available for write)

Yields $R = N$ (reads pay penalty)

Typical: $R=2$, $W=2$, $N=4$
Dynamo Summary

An eventually consistent highly available key/value store
AP in CAP space
Focuses on low latency, SLAs
Very low latency writes, reconciliation in reads

Key techniques used in many other distributed systems
  Consistent hashing, (sloppy) quorum-based replication, vector clocks, gossip-based membership, merkel tree synchronization
Project Summary

To be SQL or Not to be SQL
Bright future of NoSQL

Companies using NoSQL
- Google
- Facebook
- Amazon
- Twitter
- Linkedin
... many many more.
Conclusion

Even NoSQL - Not a "One Size fits all" kinda shoe.

Shoe horning your database is just bad, bad, bad!

Use when
- Data schema keeps on varying often
- Scalability really becomes an issue

Not to use when
- The data is inherently relational
- Lots of complex queries to write
- You need good helping resources
  eg. debugger, performance tools
References - 1

Dynamo: amazon's highly available key-value store. In Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles (SOSP '07). ACM, New York, NY, USA, 205-220


http://docs.mongodb.org/manual/core/sharding-introduction
http://mongodb.com/learn/nosql
http://www.mongodb.com/learn/nosql
http://www.cs.rutgers.edu/~pxk/417/notes/content/bigtable.html
http://en.wikipedia.org/wiki/ACID
References - 2

http://www.slideshare.net/mongodb/mongodb-autosharding-at-mongo-seattle

http://www.slideshare.net/danglbl/schemaless-databases

http://infoq.com/presentations/NoSQL-Survey-Comparison

http://info.mongodb.com/rs/mongodb/images/10gen_Top_5_NoSQL_Considerations.pdf


http://technosophos.com/2014/04/11/nosql-no-more.html
Questions
Backup Slides

Basic Concepts
Any storage model other than tabular relations.

- Trees
- Graphs
- Key-value
- XML
- etc.
Auto-sharding

TODO:
Impedence Mismatch

You break structured data into pieces and spread it across different tables.
leads to object relational mapping

lots of traffic => buy bigger boxes. Lot of small boxes. SQL was designed to run on single box.
Consistent Hashing
For data partitioning, replicating and load balancing

Sloppy Quorums
Boosts availability in present of failures

Vector Clocks
For tracking casual dependencies among different versions of the same key (data)

Gossip-based group membership protocol
For maintaining information about live nodes

Anti-entropy protocol using hash/merkle trees
Background synchronization of divergent replicas
Consistent Hashing

Availability
Replication and partitioning
Vector Clocks

Tracking causal dependencies

Figure 3: Version evolution of an object over time.
Merkel Trees
Each node keeps a merkel tree for each of its key ranges
Compare the root of the tree with replicas
  if equal => replicas in synch
  Traverse the tree and synch those keys that differ

Membership:
Node contacts a random node every 1s.
Gossip used for exchanging and partitioning/placement metadata
Merkel Trees

Diagram showing the structure of a Merkle Tree with leaves V1, V2, V3, and V4.

- **rootHash**: hash(H0||H1)
- **H0**: hash(H00||H01)
  - **H00**: hash(V1)
  - **H01**: hash(V2)
- **H1**: hash(H10||H11)
  - **H10**: hash(V3)
  - **H11**: hash(V4)

The top node is the root hash, derived from the hashes of two child nodes, which are in turn derived from the hashes of their respective child nodes and data elements (V1, V2, V3, V4).
Atomicity requires that each transaction is "all or nothing"
The consistency property ensures that any transaction will bring the database from one valid state to another valid state.

**Success**

\[
A: a + x \\
B: b - x
\]

**Failure**

\[
A + B = a + b \\
A + B = a + b - 10
\]
The isolation property ensures that the concurrent execution of transactions results in a system state that would be obtained if transactions were executed serially, i.e. one after the other.

<table>
<thead>
<tr>
<th>Success</th>
<th>Failure</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1: a - x</td>
<td>T1: a - x</td>
</tr>
<tr>
<td>T1: b + x</td>
<td>T2 : b - x</td>
</tr>
<tr>
<td>T2: b - x</td>
<td>T2 : a + x</td>
</tr>
<tr>
<td>T2: a + x</td>
<td>T1: b + x</td>
</tr>
</tbody>
</table>

failure
Durability means that once a transaction has been committed, it will remain so, even in the event of power loss, crashes, or errors.

### Success

- T1: a - x
- T1: b + x
- T2: b - x
- T2: a + x

### Failure

the changes are lost
Persistent
   Data gets stored permanently in the disk

Sorted
   Data kept in hierarchical fashion
   Spatial Locality

Map Store
   Just a collection of (key, value) pairs

Features
   • sparse
   • distributed
   • scalable
   • persistent
   • sorted
   • map store
BASE

An alternative to ACID

• Basically Available
  • Support partial failures without total system failure.
• Soft state
  • optimistic and accepts that consistency will be in state of flux.
• Eventual Consistency
  • Given a sufficiently long period of time over which no changes are sent, all updates can be expected to propagate eventually.