On the Elasticity of NoSQL Databases over Cloud Management Platforms

Dimitrios Tsoumakos
Department of Informatics, Ionian University

CS Colloquium Series, Department of Computer Science, University of Cyprus
Motivation – the story(1)

- ‘Big-data’ processing era
  - (Web) analytics, science, business
  - Store + analyze everything
- Distributed, high-performance processing
  - From P2P to Grid computing
  - And now to the clouds…

- Traditional databases not up to the task
Motivation – the story (2)

- **NoSQL**
  - Non-relational
  - Horizontal scalable
  - Distributed
  - Open source

- And often:
  - schema-free, easily replicated, simple API, eventually consistent / (not ACID), big-data-friendly, etc

- Many, many, implementations…
NoSQLs and elasticity

- Column family
  - Hbase, Cassandra, ...
- Document store
  - CouchDB, mongoDB, ...
- Key-Value store
  - Riak, Dynamo, Voldemort, ...
- Many offer **elasticity+sharding:**
  - Expand/contract resources according to demand
  - Pay-as-you-go, robustness, performance
  - Shared-nothing architecture allows that
  - Important! See Apr 2011 Amazon outage (foursquare, reddit,...)
thus...(end of the story)

- PaaS and NoSQLs are (or should be) inherently elastic
- How efficiently do they implement elasticity?
  - NoSQLs over an IaaS platform
    - EC2, Eucalyptus, OpenStack,…
  - Study that registers qualitative + quantitative results

- Related
  - Report NoSQL performance (not elasticity)
  - Cloud platform elasticity (no NoSQL)
  - Domain-specific
  - Initial cluster size (not dynamic)
Contributions

- VM-based framework for NoSQL cluster monitoring
- For a cluster resize, identify and measure
  - Cost, gains
  - In terms of:
    - Time, effort, increase in throughput, latency, …?
- Ultimate goal: Provide a generic platform
  - any NoSQL engine
  - User-defined policies
  - Automatic resource provisioning
- Attempt towards this goal
  - Tiramola system
Contribution side-effects

- Coding + infrastructure
  - Open source python code (GFOSS + google code)
  - [http://tiramola.googlecode.com](http://tiramola.googlecode.com)
- Using cloud-based client tools, platform-agnostic
  - EucaTools guarantee execution in numerous cloud platforms
- YCSB clients
- Cassandra, Hbase implementation
  - almost Voldemort, Riak
- How-to, best practices, glitches, erroneous assumptions,
  ...
Framework architecture

TIRAMOLA

Decision Making

Get fresh metrics
NoSQL Cluster resize
Hardware resize

Monitoring
Cluster Coordinator
Cloud Management

Collect Performance Metrics
Manage NoSQL nodes

Cloud Provider
Add/delete VMs
Adjust resources

User policies

Clients
Load

Virtual NoSQL Cluster
Architectural considerations

- Robustness
  - Daemon process that checkpoints and can be restarted
  - State is provided from the IaaS Cloud and the Monitoring module.
  - Applicable timeouts (not realtime systems!)

- Modularity
  - Different interchangeable components
  - APIs that utilize primitives (NoSQL and Policies)

- Expandability
  - Speed (irrelevant in most cases)
  - Written in Python
Platform Setup

- **16 physical nodes**
  - 2xQuadCore E5520 Intel Xeon® Hyperthreading (@2.27GHz)
  - 48GB RAM, 2 SAS – RAID 1

- **VMs**
  - Similar to an Amazon EC2 large instance
  - 4-core processor, 8GB RAM, 50GB disk space
  - QCOW image: 1.6GB compressed, 4.3GB uncompressed
  - VM root fs instead of EBS (Reddit outage)

- **Cluster**
  - Eucalyptus
    - → OpenStack
Clients, Data and Workloads

- Hbase (v. 0.20.6), Cassandra (v. 0.7.0 beta)
  - Hadoop 0.2.20
  - Replication factor: 3
  - 8 initial nodes (VMs)
- Ganglia 3.1.2
- YCSB tool
  - Database: 20M objects – 20GB raw (Cass ~60GB, Hbase ~90GB)
  - Loads: UNI_R, UNI_U, UNI_RMW, ZIPF_R
    - Default: uniform read, 10%-50% range
    - λ parameter (tricky)
    - Both client (YCSB) and cluster (Ganglia) metrics reported
Methodology

- Step 1: Identify which DB metrics are affected under various loads
  - Consider both server-side and client-side metrics

- Step 2: Identify costs + gain for a cluster resize
  - Cost of adding/removing nodes
  - Gains of increasing cluster size (how many nodes?)

- Step 3: Showcase the automated use-case
  - Tiramola System

- Step 4: Useful conclusions – best practices
1. Metrics affected 1/2

- **Max throughput**
  - HBase $\mu_{max} = 80\text{Kreqs/sec}$ @ $\lambda=80\text{Kreqs/sec}$
  - Cassandra $\mu_{max} = 13\text{Kreqs/sec}$ @ $\lambda=20\text{Kreqs/sec}$

- **Max total cluster CPU usage**
  - HBase $\text{CPU}_{max}=55\%$
  - Cassandra $\text{CPU}_{max}=76\%$

- **Further $\lambda$ increase has no effect**
  - Systems are fully utilized and requests are queued

- Stress systems by increasing client request $\lambda$
- UNI_R workload
- Identify critical operation points
- 8-node cluster, no resize
1. Metrics affected 2/2

- **Constant HBase Memory usage of 17GB**
  - Reads take up constant memory

- **Cassandra Memory usage increases**
  - Caching result during P2P routing – cluster restart for cleanup!

- **Linear latency increase in both systems**
  - Pending requests increase, as throughput remains constant after critical operation point
Methodology

- Step 1: Identify which DB metrics are affected under various loads
  - Consider both server-side and client-side metrics

- **Step 2: Identify costs + gain for a cluster resize**
  - Cost of adding/removing nodes
  - Gains of increasing cluster size (how many nodes?)

- Step 3: Showcase the automated use-case
  - Tiramola System

- Step 4: Useful conclusions – best practices
The setup

- $\lambda_{\text{start}} = 180\text{reqs/sec (way over critical point)}$
- At time $t=370\text{sec}$:
  - Double the cluster size (add extra 8 nodes)
  - Triple the cluster size (add extra 16 nodes)
  - 4 different experiments for each database
    - READ+8, READ+16, UPDATE+8 and UPDATE+16
- Measure client, cluster-side metrics
  - query latency, throughput and total cluster usage
  - Vs time
2. HBase cluster resize 1/2

- Transient negative effect in read workloads
  - HMaster reassigns regions
  - Clients experience cache misses
- Effect is more apparent in READ+8
  - A larger cluster absorbs load more easily
- Updates experience oscillations
- Compaction effect
  - Incoming data is cached in memory (low latency)
  - When memory is full, data is flushed to disk (high latency)
2. HBase cluster resize 2/2

- For READ, throughput is (roughly) doubled and tripled
  - More servers handle more requests
  - Data is not transferred, but it is cached

- Update is not affected
  - I/O bound operation
  - Updates will be handled by the initial 8 nodes
  - Only new regions due to compaction will be served by new nodes
2. Cassandra cluster resize 1/2

- **READ workloads**
  - Initial latency goes to 1/2 and 1/3 of 22 secs
  - More servers join the ring and take requests
  - No transient effect because of decentralized p2p nature

- **UPDATE workloads**
  - The same effect as in the read case
  - Writes are faster than reads due to weak consistency model
2. Cassandra cluster resize 2/2

- **READ workloads**
  - Similar trend as in the latency case

- **UPDATE workloads**
  - Same almost linear effect
  - Weak consistency model

- **READ workloads**
  - CPU decreases to 60% and 50%
  - Load is enough for every node to contribute

- **UPDATE same as READ**
What about the time cost?

- **VM initialization**
  - 3min for addition, negligible for removal (few secs)

- **Node configuration**
  - Config files and propagation (at most 30 sec cycle)

- **Region rebalance**
  - Actively participate in the NoSQL cluster
  - Cassandra more efficient, Hbase depends on data, #nodes,…

- **Data rebalance**
  - Optional
  - Hbase: data / cluster-size dependent (+2h)
  - Cassandra: individual loadbalance signals
Cassandra vs. HBase

- **Hbase**
  - has fast reads
  - In the READ+16 case
    - HBase handles ~160K reqs/sec with a latency of 2 sec and 40% CPU
    - Cassandra handles ~40K reqs/sec with a latency of 8 sec and 50% CPU

- **Cassandra**
  - has fast(er) writes
  - Stable throughput/latency without oscillations
  - No transient negative effect during node joins
Methodology

- Step 1: Identify which DB metrics are affected under various loads
  - Consider both server-side and client-side metrics

- Step 2: Identify costs + gain for a cluster resize
  - Cost of adding/removing nodes
  - Gains of increasing cluster size (how many nodes?)

- **Step 3: Showcase the automated use-case**
  - Tiramola System

- Step 4: Useful conclusions – best practices
Tiramola core: Decision-Making module

- Formulation as an MDP:
  - \(\{S, A, \{P_{i\alpha_j}\}, \gamma, R\}\)
  - Identify \(V(s) = E\left\{\sum_k \gamma^k r_{k+t+1} | s_t = s\right\}\)
  - State = \# running VMs
  - Actions = \{add-n, rem-n, no-op\}
  - Reward Function – sets the policy for the resize
    - Immediate gain for going to state \(s\)
    - \(R(s) = f(\text{gains, costs})\)
    - Thus: \(V^*(s) = R(s) + \max_{s \xrightarrow{a} s'} V^*(s')\) (Bellman’s equation)
  - System of equations, optimal policy greedy w.r.t. \(V^*\)
Estimate $R(s)$ for possible transitions

- How to know exact $R(s)$ for all $s$, without making the transition?
  - Assume “deterministic”, reliable cluster behavior
  - Similar input $\Rightarrow$ similar performance metrics

- Idea: cluster measurements around current load

- Add 2 extra dimensions: #VMs, load
  - Find, for all permissible transitions, what latency would be
Evaluating Tiramola – setup

- Initial state: $S_4$, 1 to 16 VMs cluster size allowed
- Sinusoid-like READ loads – vary peak, periodicity
  - Alter YCSB clients
- Reward functions
  - $r_1(s) = -C \cdot |\text{VMs}|$
  - $r_2(s) = B \cdot \text{thr}$
  - $r_3(s) = B \cdot \text{thr} - C \cdot |\text{VM}|^2$
  - $r_4(s) = B \cdot \text{thr} - C \cdot |\text{VMs}| - A \cdot \text{lat.}$
- Monitor every min, decide every 10 min, 5 min backoff
- Training Period for initial data points
Evaluating Tiramola – 1

- $r_1, r_2$

- $r_3, r_4$
Evaluating Tiramola – 2

- Different amplitude

- Different periodicity
Methodology

- **Step 1:** Identify which DB metrics are affected under various loads
  - Consider both server-side and client-side metrics
- **Step 2:** Identify costs + gain for a cluster resize
  - Cost of adding/removing nodes
  - Gains of increasing cluster size (how many nodes?)
- **Step 3:** Use steps 1, 2 to showcase an automated use-case
  - Using Hbase

- **Step 4:** Useful conclusions – best practices
4. Useful conclusions – best practices

- Choose the right DB for your application/workload (when in doubt, go with the one you’re familiar with)
- HBase is a better all-rounder; Cassandra is handicapped by slow read performance and absence of shared FS.
- TIRAMOLA is robust and in principle can incorporate any NoSQL or application by writing ~100 Python lines
- Building PaaSs is critical for the Cloud. Most users do not have the knowledge, inclination, time or money to do it themselves ⇒ need for adaptive tools (in our example, elastic NoSQL databases).
Questions?

- “TIRAMOLA: Elastic NoSQL Provisioning through A Cloud Management Platform” – SIGMOD 2012 (Demo Track)
- “On the Elasticity of NoSQL Databases over Cloud Management Platforms” – CIKM 2011
- “Elastic NoSQL databases over the Cloud” – ΕΛΛΑΚ 2011
- “Automatic, multi-grained elasticity-provisioning for the Cloud” – FP7 STReP (under review)
- http://tiramola.googlecode.com