NYU
Britt Ewen, Rob Mancuso

ENTERPRISE DATABASE PERFORMANCE AND TUNING

Agenda

- "Real World" Performance Tuning
  - Steps to get to the root cause of a problem
- Platform Overview
  - Relational Databases
  - HBase
  - MongoDB
- Case Studies
  - HBase (Part 1)
  - HBase (Part 2)
  - Relational Database 1
  - Relational Database 2
  - MongoDB

Database Performance Tuning

- Optimizer is the most complex part of almost any application stack
- Addressing a sudden drop-off in performance
  - Immediate impact requiring time sensitive response
  - Could follow a system change – known or unknown
- Improving a system
  - Known issue which can be addressed proactively
  - Increase in volume, number of users, …
  - System could be degrading slowly over time
- A reproducible problem is a solvable problem!
- Sometimes – often – the answers are simple
  - But… getting there is the ~~hard~~ fun part

Non Relational Databases

- Not a magic bullet to address all performance problems
  - Many similarities to the relational space
- Different platforms take different approaches
  - HBase != Cassandra != MongoDB != PostgreSQL != …
- What they take away more than what they add
  - No Joins = No data colocation concerns
  - No Transactions = No cross host coordination
  - Loosened Consistency = Easier to Distribute Data Geographically

Real World Performance Tuning

- "Something is Slow – Help!"
  - May or may not be database performance related
- This is akin to puzzle solving – but no one yet knows the answer

Real World Performance Tuning

- Try to start each new investigation with an open mind and clean slate
- What behavior is being identified v. what people think might of caused it
  - We often see a tendency to take the last problem solved and assume the problem has recurred
  - E.g.-"Reboot!", "Expand the Log!", "Increase Locks!", "Storage Problems!", …
- Correlation versus causation

Diagnosing a Performance Drop Off

- Isolating "what's changed"
  - Code migration
  - Database version change
  - Maintenance Job Schedules
    - E.g. – statistics, reorgs, backups
  - New data patterns
    - E.g. – "date roll"
  - Environment changes
    - E.g. – hardware, network, datacenter, …
  - "Nothing"
    - Hint: It's never "nothing"

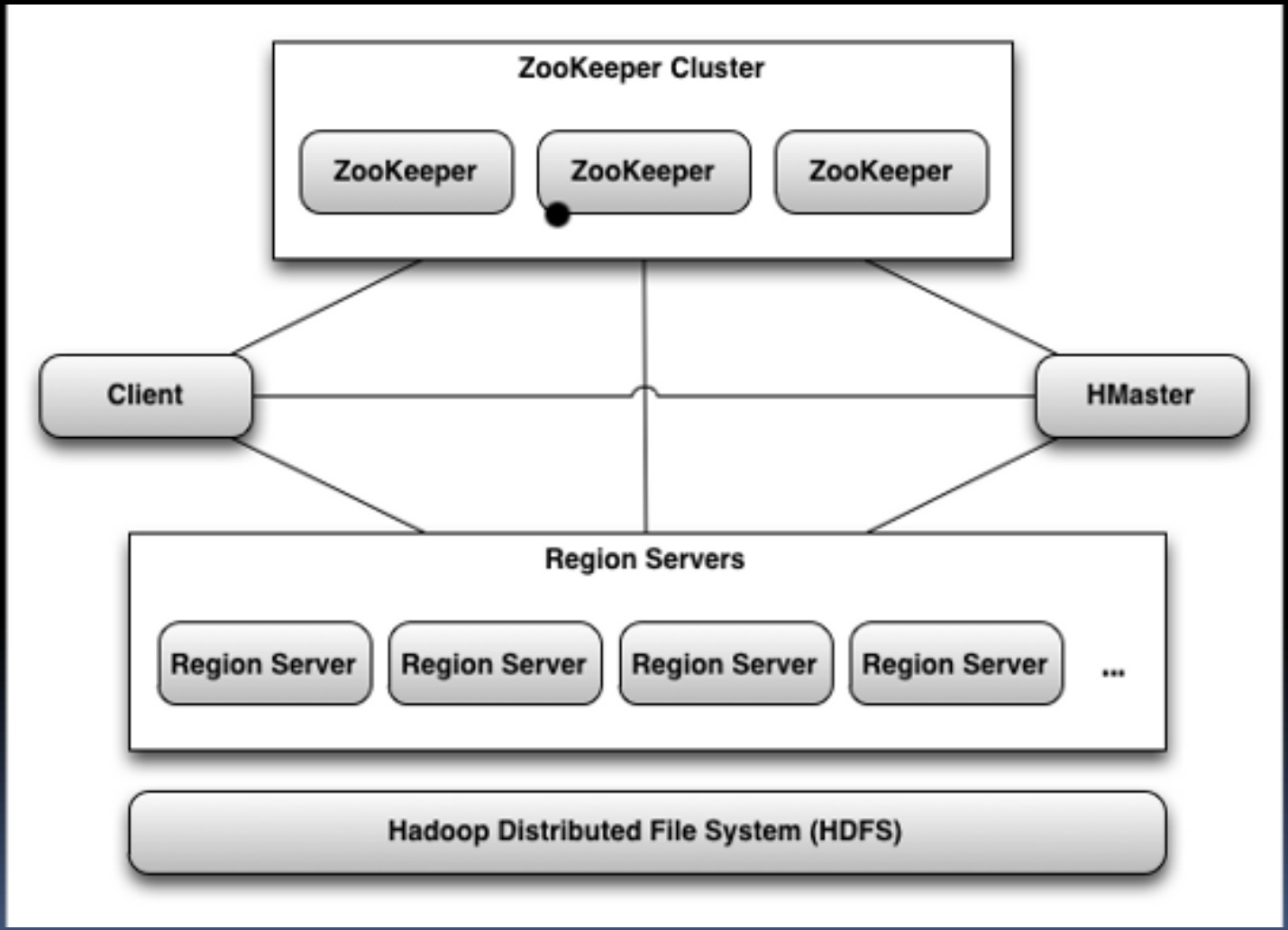
Sample “P&T” Checklist

- What’s the level of impact of the performance issue?
- When did the issue start and is it still ongoing?
- When was performance last acceptable?
- **What’s changed?**
  - Upgrades
  - New workload
  - Behavior – increased volume, usage
  - Code or application deployments
  - Object definitions – structure, indexes
  - Host or storage migrations
  - Failover
  - …
- What have you looked at already?
  - Have you identified a specific issue?

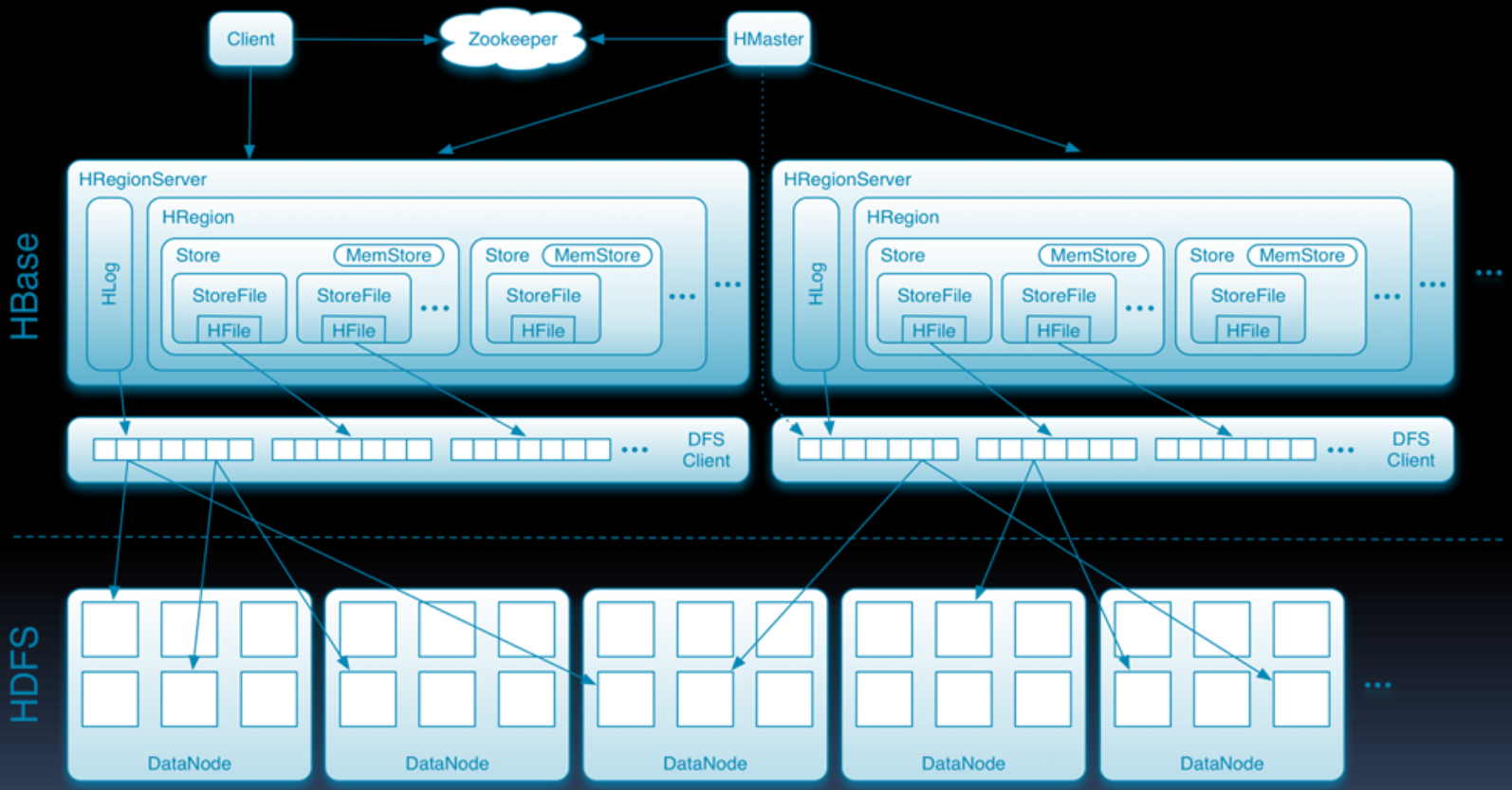
HBase Overview

- Distributed database for large to massive data sets
- Real-time access for key-based look-ups
- Tables partitioned by region
  - Regions are stripes of data
  - Each region owns a chunk/range of the table's data
  - Regions spread out over regionservers
  - Each region has its own in-memory write cache

HBase Architecture - Cluster



Inside a Regionserver



HBase Table Schema

Sorted map of key/value pairs

- Key={row key, column family, column qualifier, timestamp}

Row key (rk)

- Primary key for the row (eg – employee number)

Column family (cf)

- Grouping of column qualifiers (eg – Info)
- Data stored by column family

Column qualifier (cq)

- Column name, defined at insert time (eg – fname, lname, position)
- Cf:cq collectively known as a column

Timestamp (ts)

- Stored as a long - milliseconds since epoch (1/1/1970 UTC)
- Automatically add at insert time (eg – 1347981724589)
- Can be overridden with your own value (eg – 1, 2, …)

Relational

create table Product
(productID int, instrType varchar, currency varchar, issueDate date, …)

productID	instrType	marketCode	currency	issueDate
10000	stock	US	USD	1/1/2012

HBase

create table Product, 'f'

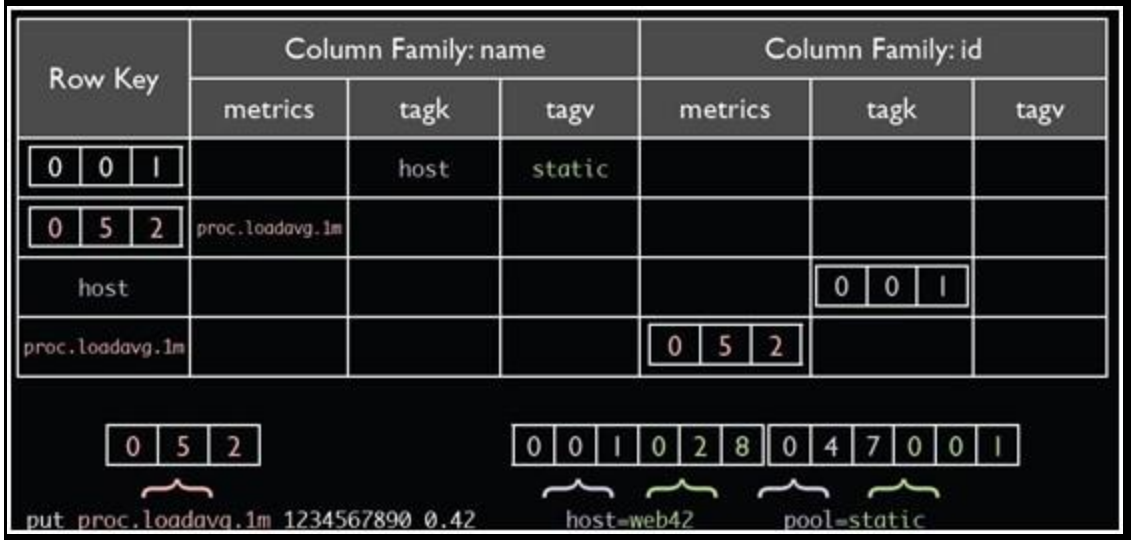
Rowkey	CF:CQ	TS	Value
10000	f:instrType	13424589..	stock
10000	f:marketCode	13424589..	US
10000	f:currency	13424589..	USD
10000	F:issueDate	13424589..	1/1/2012

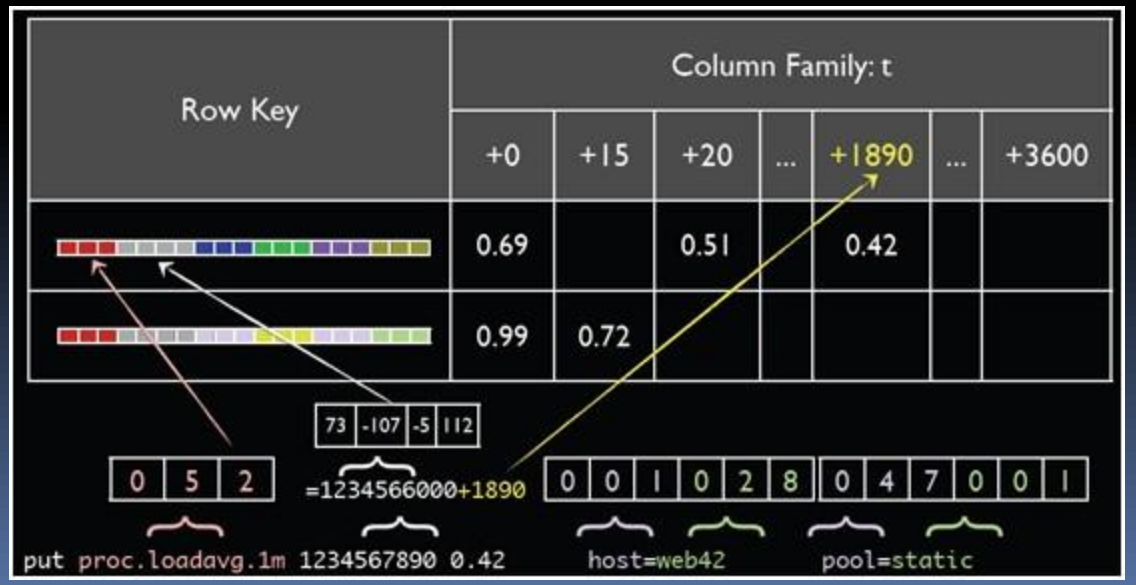
Case Studies

Case Study: HBase Key Selection

- Telemetry data stored by:
  - Entity = Object (e.g.-server, host) producing metrics
    - Stored as tags
  - Timestamp = Snapshot of value being collected
    - Stored by hour, with finer grain values on row
- Started with:
  - Rowkey = {Metric}{TimestampByHour}{Entity}{TagKey=TagValue},…
  - Column Qualifier = {TimeOffsetWithinHour}
- Example:
  - {PHYSICAL_IO}{1234566000}{ENTITY=DATASERVER1}{IOSIZE=16384}

Case Study: HBase Key Selection





Case Study: HBase Key Selection

- Model is good for lookup by Metric
  - Also supports pulling a range of values by time
- But… lookups are primarily by "Entity", not "Metric"
  - E.g.- "Show me the I/O for this server", vs. "Show me I/O across all of my servers"

Case Study: What do you think?

Case Study: HBase Key Selection

- Store data differently:

Rowkey =
{Entity}{TimestampByHour}{Metric}{TagKey=TagValue},
…

Case Study: HBase Access Patterns

- What if we need to store both representations?
- Lookups by Entity
  - "CPU for DATASERVER1 From 8AM to 9AM"
- Lookups by Metric
  - "Top Ten CPU Busy"
- Is it a real-time requirement or batch driven?

Case Study: HBase Access Patterns

- Need a "secondary index"
  - But no native HBase support
- Design and maintain it ourselves
  - Secondary table (like a clustered index)
    - Good for low cardinality columns
    - More storage, more load
  - Secondary index (like a nonclustered index)
    - Good for high cardinality columns
    - Less storage, less load

Case Study: HBase Access Patterns

- How do we maintain our "index"?
  - Client-side: dual posting
  - Server-side: coprocessors (like triggers)
- Coprocessors abstract complexity from client
- Used asynchronous client to scale
  - 100k writes/sec
- Scans against metrics are now fast and light on load

Main Table (by entity)

Rowkey	CF:CQ	Value
dataserver1+TS+metric=CPU	t:60	45
dataserver1+TS+metric=CPU	t:120	42

Secondary Table (by metric)

Rowkey	CF:CQ	Value
CPU+TS+entity=dataserver1	t:60	45
CPU+TS+entity=dataserver1	t:120	42

Case Study: Tuning Write Path

- Slow copy of large table from one cluster to another
  - Estimated @ 30 hours for 1 billion rows
- Symptoms:
  - Spikes in throughput followed by idle time
  - Error messages:
    - "Too many storefiles"
    - "Memstore reached max size"
- What we looked at:
  - System resources
  - HBase metrics

Case Study: What do you think?

Case Study: Tuning Write Path

- What should we do?
  - Increasing memstore or storefiles addresses **symptoms** but not the **problem**
- What we did:
  - Region Pre-Splits
  - HBase splits are expensive multi-stage online operations
  - Pre-splitting reduces and up-fronts work
- Result:
  - 1 billion rows loaded in 90 minutes

Case Study: Query Access Path

- Symptoms:
  - "The Database is Slow"
  - Jobs exceeding expected run time
  - High CPU, logical (cache read), and physical (disk read) I/O
  - All applications using database server are affected

- What we looked at:
  - Identified queries utilizing greatest resources
    - Specifically look for large I/O count relative to amount of data
  - Compared CPU to prior week's run
  - Followed-up with application team regarding any changes

Identifying the Problem

```
Table                          LIO        PIO
------------------------------ ---------- ---------
database1..testtbl 1           9751121          8
database1..testtbl2               2991          0

--------------------------------
select ...
from testtbl 1   t1,
        testtbl2    t2
where t1.account_num = t2.account_num
and t1.batchID=1234567
and t2.period_dt="11/18/2013"
--------------------------------
```

```
FROM TABLE
    testtbl2
Nested iteration.
Index : idx3
Positioning by key.
Keys are:
    period_dt  ASC

FROM TABLE
    testtbl1
Nested iteration.
Index : idx1
Positioning by key.
Keys are:
    account_num  ASC
```

Analyze The Query

Possible Access Paths

t1 (tablescan) ----> t2 (tablescan)

t1 (tablescan) ----> t2 (idx on account_num)

t1 (tablescan) ----> t2 (idx on period_dt)

t1 (idx on batchID) ----> t2 (tablescan)

t1 (idx on batchID) ----> t2 (idx on account_num)

t1 (idx on batchID) ----> t2 (idx on period_dt)

t2 (tablescan) ----> t1 (tablescan)

t2 (tablescan) ----> t1 (idx on account_num)

t2 (tablescan) ----> t1 (idx on batchID)

t2 (idx on period_dt) ----> t1 (tablescan)

t2 (idx on period_dt) ----> t1 (idx on account_num)

t2 (idx on period_dt) ----> t1 (idx on batchID)

testtbl1 (1.5 million rows)

- Nonclustered on account_num
- Nonclustered on batchID

testtbl2 (1.5 million rows)

- Nonclustered on account_num
- Nonclustered on period_dt

SARG/JOIN Selectivity

t1.account_num: `0.0008875 (1k)`

t1.batchID: `0.01014 (15k)`

t2.account_num: `0.0008875 (1k)`

t2.period_dt: `0.000000(0 rows?)`

Incorrect! Real selectivity is 0.10 (150k rows)

Case Study: What do you think?

Case Study: Query Access Path

- What we did:
  - Updated statistics
- What caused it?
  - New index added for different workload
  - Impacts all queries accessing table with SARGs for covered fields
- Long term fix
  - Several options
    - Drop index
    - Manage stats
    - Leverage optimizer ability to directly influence access plan ("force")
    - …

Case Study: Selective Slowdown

- Symptoms:
  - No known changes to an application's stack
  - No database changes
  - Performance of certain long running OLAP ("Online Analytical Processing") queries stalls
  - Short running OLTP ("Online Transaction Processing") queries largely unaffected

Case Study: What do you think?

Case Study: Selective Slowdown

- Resolution:
  - Storage Contention
  - Migrate to a different storage "silo"
- Shared disk Storage Area Network ("SAN") environment
- Multiple databases leverage same subsystem
- Unrelated workloads can impact one another

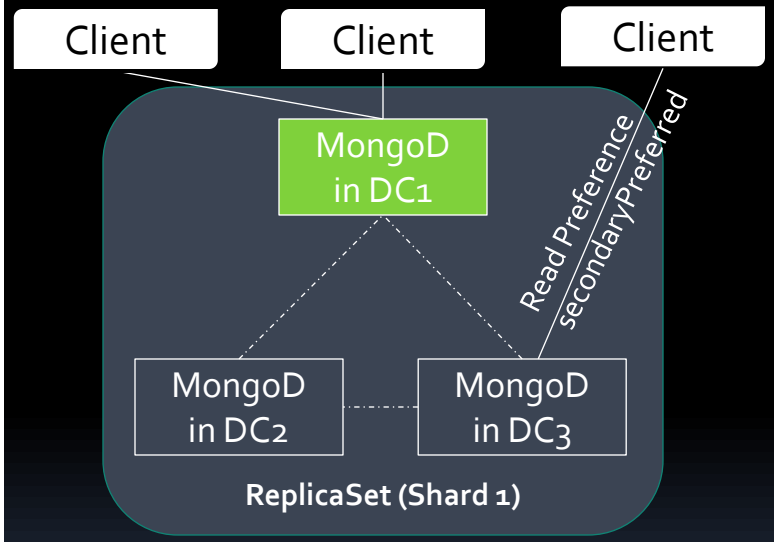
MongoDB Overview

- Non-relational "NoSQL" Database
- Document Store
  - Clusters → Databases → Collections → Fields
- Indexing
  - Query plans still exist
- Statement level read and write concerns
- Sharding for horizontal scale out

MongoDB Architecture

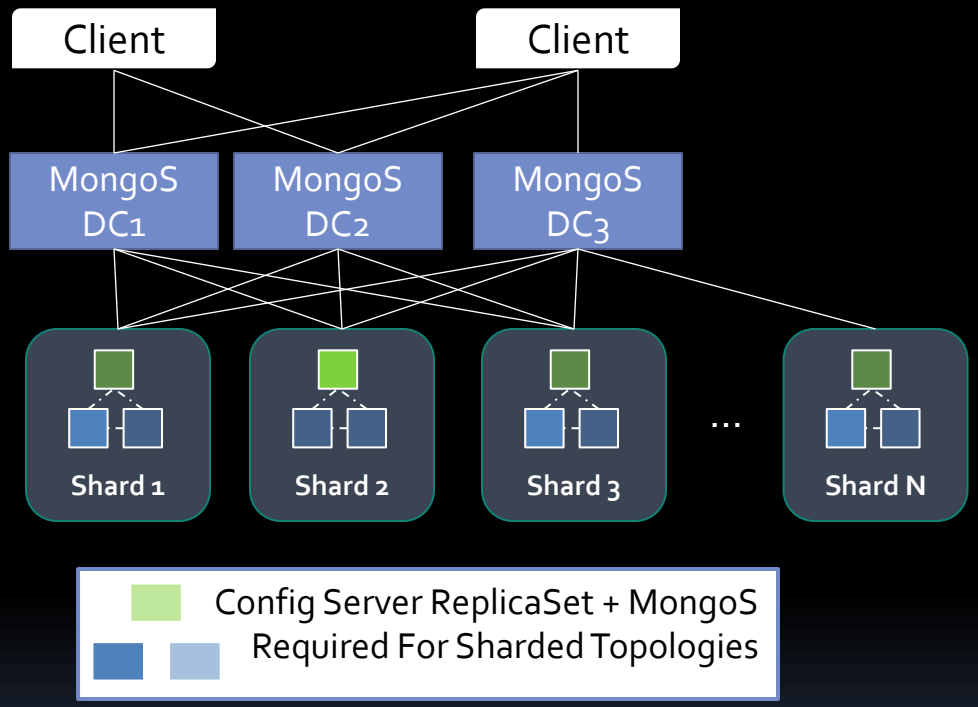
The "**MongoD**" process is a standalone server process that contains databases, collections, documents, accepts connections, queries, etc.

High Availability via **Replica Sets** – Each Replica Set has Identical Data via Async OpLog Replication



A majority of Replica Set (RS) nodes must be up for a primary to be elected
Write activity is always to primary
Client connects to 1+ "mongoD" nodes

Horizontal Scaling via **Sharding**
Collections Partitioned by Document Field **(Shard Key)** into new ReplicaSet (Shard) to Expand Cluster



Shards are ReplicaSets, each RS Node has Same Data
The MongoS (router) abstracts Sharding from Application
Collection Shard Keys need to be chosen carefully

Case Study: MongoDB

- Symptoms:
  - Significant drop off in write performance after migrating to a new cluster
- Old and new environments both online

Case Study: What do you think?

Case Study: MongoDB

- Resolution: journalCommitInterval
  - Original cluster had setting of 10ms
  - New cluster had 100ms
- Configuration settings are important!

MongoDB: Other Considerations

- Shard key selection
- Choosing write and read concerns
- Indexing for performance
- Geographic cluster topology

Other Real World Examples

- Memory Settings
  - Physical I/O uptick primarily affects OLAP queries
- Increase in Short Duration Transactions
  - Degrades performance across the board
- Cross Datacenter Access
  - Following failover or hardware migration
- Changes to Query Optimization Time
  - Version upgrade, additional indexes, …
  - Greater impact to short duration queries
- Output diagnostic messages
  - Can have a surprising overall impact

Appendix

SQL Relational Tuning Factors

- Query plan selection
  - Join strategy
  - Index selection
  - Static v. dynamic plans
    - Statement cache
    - Stored procedures
- Statistics
  - Keeping them in synch with actual data patterns
- Indexing
- Fragmentation / REORG
  - Often overlooked, but can change access plans or degrade performance of a query dramatically
- System Performance
  - CPU Utilization
  - I/O Throughput
  - Network latency
- Configuration
  - DB memory, system memory

Transaction Isolation and Concurrency Semantics

- Locking, blocking, logging
- MVCC v. Lock Based Isolation

HBase: Other Considerations

- HBase / HDFS DFSClient "Short Circuits"
  - Improves performance if affinity maintained
- Avoid running HDFS rebalancer
  - Will obviate HBase's own affinitization

HBase – Telemetry Data Use Case

OpenTSDB is an open source HBase application designed to store and serve time series metrics data

Tables

- tsdb-uid: metadata table that maps names to IDs
- tsdb: Stores the raw data, uses a composite row key:
  - { metricID, timestamp, tagkeyID, tagvalueID…tagN }

Example: Retrieve LogicalReads for dataserver from 10am to 11am…

Convert metric and entity into IDs

- LogicalReads = metricID 123
- Dataserver = tagkeyID 456
- ENTITY= tagvalueID 789
- 10am = 1360854000; 11am = 1360857600

Create and run a scanner

- startrow=123|1360854000 |%456789%
- stoprow=123|1360857600 |%456789%

We can overload the metricID field for performance reasons

- startrow=123|1360854000 (123= ENTITY/LogicalReads)
- stoprow=123|1360857600