



PERCONA
Performance Consulting Experts

Choosing the Right Tools for the Job, SQL & NOSQL

MySQL User Conference 2010

Matt & Yves

Percona Inc

MySQLPerformanceBlog.com

About Us...

- <http://MysqlPerformanceBlog.com>
- <http://www.bigdbahead.com>

Matt Yonkovit

Principal Architect

Yves Trudeau

Principal Architect

NOSQL?

NoSQL is a movement promoting a loosely defined class of non-relational data stores that break with a long history of relational databases and ACID guarantees. These data stores may not require fixed table schemas, usually avoid join operations and typically scale horizontally. Academics and papers typically refer to these databases as structured storage. - Wiki

A few Solutions out there

- Tokyo Tyrant
- Cassandra
- Redis
- Memcached
- MongoDB
- CouchDB
- HBase
- Voldemort
- SimpleDB
- BigTable
- Many More

Two Main Camps

- Old School
- New Hotness

They get along Like Republicans and Democrats



Old School

- The database is the defacto for all data storage
- Companies assume that the database is the best choice
- Value the processes and structure that an RDBMS provides.
- Typically slower to change or to look at alternatives
- Require more thought and planning

New Hotness

- Some Think SQL is dead and for Legacy applications
- Some Think traditional RDBMS's do not scale and can not keep up with the data explosion that is occurring with web 2/3.x+
- Value the un-structured, ability to move quickly.
- Much less risk adverse, willing to try new things very quickly
- Do not have the ability, or want the ability to dedicate resources to database administration.

Middle Ground

Yes there are those who fall into the middle, they just do not talk as much :)

Matt's Editorial

Underlying Themes for the NOSQL Movement

- Truth is always somewhere in the middle, there is room for both the RDBMS and NOSQL Solutions
- This latest movement is made Largely of a desire for simplicity
- Its a way for developers to gain more control of the entire application stack
- Explosive Data Growth
- Be leary: The perception is growing that it is the savior for all the ill's of the data world ***Buzz Word Anyone***

Simplicity?

Simplicity in Scaling (Sharding is hard)

- Simplicity in Hardware (smaller commodity boxes, vs bigger ones)
- Simplified api's : SQL can be complex, Most NOSQL API's have at most a dozen methods for data access
- Simplicity in resources, no need for the wasteful DBA/Sysadmin Type Guys
- Simplicity In writing and reading data... schema? What schema?
- Simplicity in Design and Design Changes... how long does it take to add a new column to a 1B row table?

Control?

- DB's are voodoo black magic for many developers
- Having a DBA means some of the application is outside their control
- SQL while an established standard is not as agile or fun as something like Ruby
- Some people like to exercise complete control of their code
- It takes a lot of time and effort to understand all the pieces and internals of a database, if you can at all (closed source)
- Database designs tend to be rigid, developers sometimes can not do what they want

Simplicity and more control sound great!

- There is a price for them
- Some “features” of a relational database are reduced or removed entirely
 - Durability in some cases
 - Replication in some cases
- Things you may take for granted in a relational database, you have to write on your own, like:
 - Joins
 - Aggregates
 - Complex where clauses
- Example?

Example:

Hash Table

- This is a bit of an extreme example but
 - Let's say you wanted to have two sets of data, 1 user, 1 messages table.
 - This is a 1 to M relationship
 - in SQL its pretty straight forward:

Table User id, Username, Name, address, misc
Table Messages id, to_userid, message, message_time

SQL Methods

Insert new Message:

Insert into messages (to_userid, message, message_time) values (1, "hello", now());

Get a list of messages for a user:

*select * from messages where to_userid = 1 order by message_time...*

Hash Table Example

- Note there maybe other ways to do this, but its typically all in code

Insert new message:

```
mykey = uuid.generate
memcached.set mykey.to_s , row
listkey= "list_#{key.to_s}"
begin
  memcached.append listkey.to_s , "::{#{mykey.to_s}"
rescue Memcached::NotStored
  memcached.set listkey.to_s , mykey.to_s
end
```

More Hash Table Examples

Then to get the list of messages:

```
listkey= "list_#{key.to_s}"  
row = memcached.get listkey.to_s  
myrow=row.split(':::')  
# Passing array will call multi-get  
return memcached.get myrow
```

- This has serious issues like what to do when you have to delete a message
- Not Easy

But you said Simplicity!

- It is if you use it for the right thing, for instance in an a true key-value setup the hash table is extremely simple and fast
- If you do things that the tools did not intend to do, then you are back to writing your own code for it.
 - Believe it or not I have talked to many people who view this as an advantage.
- By the way Benchmark Alert.... The complex “message” list example slowed down the # of rows I could read in memcached from ~50K to about 6K

Not all Created Equal

- Using my simple message example, you may tend to think that NoSQL is horrible.
 - For our simple message app a hash table is not a good fit
 - but there are other implementations
- Cassandra for instance allows you add a Super Column, which is at its very basic core a value that points to a list (multi-dimensional array)
 - This simplifies the code incredibly
- Tokyo has a table database and B-tree database which are much better fits
- Redis supports lists as well

Choose the right solution

- If you deploy the wrong solution you may not know it until its too late
- Each solution has pros and cons, its imperative you discover them before making a decision
- Do not be afraid to use multiple solutions.

Survey of a few data storage solutions

MySQL

- What is it... if you have to ask you're at the wrong conference
- Relational, popular, known

MySQL Pros/Cons

Pros

- Well Known
- Great Documentation
- ACID!!! + Predictable Crash Recovery
- Has SQL access
- Replication
- Transactional
- Much more mature, Battletested
- Large Community

Cons

- Moderate Administrative costs
- Complex scaling to multiple nodes
- Performance with very large databases can be tricky
- Many operations seem like a black box
-

NDB/MySQL Cluster

- What is it?

NDB Cluster is a high availability sharding framework.

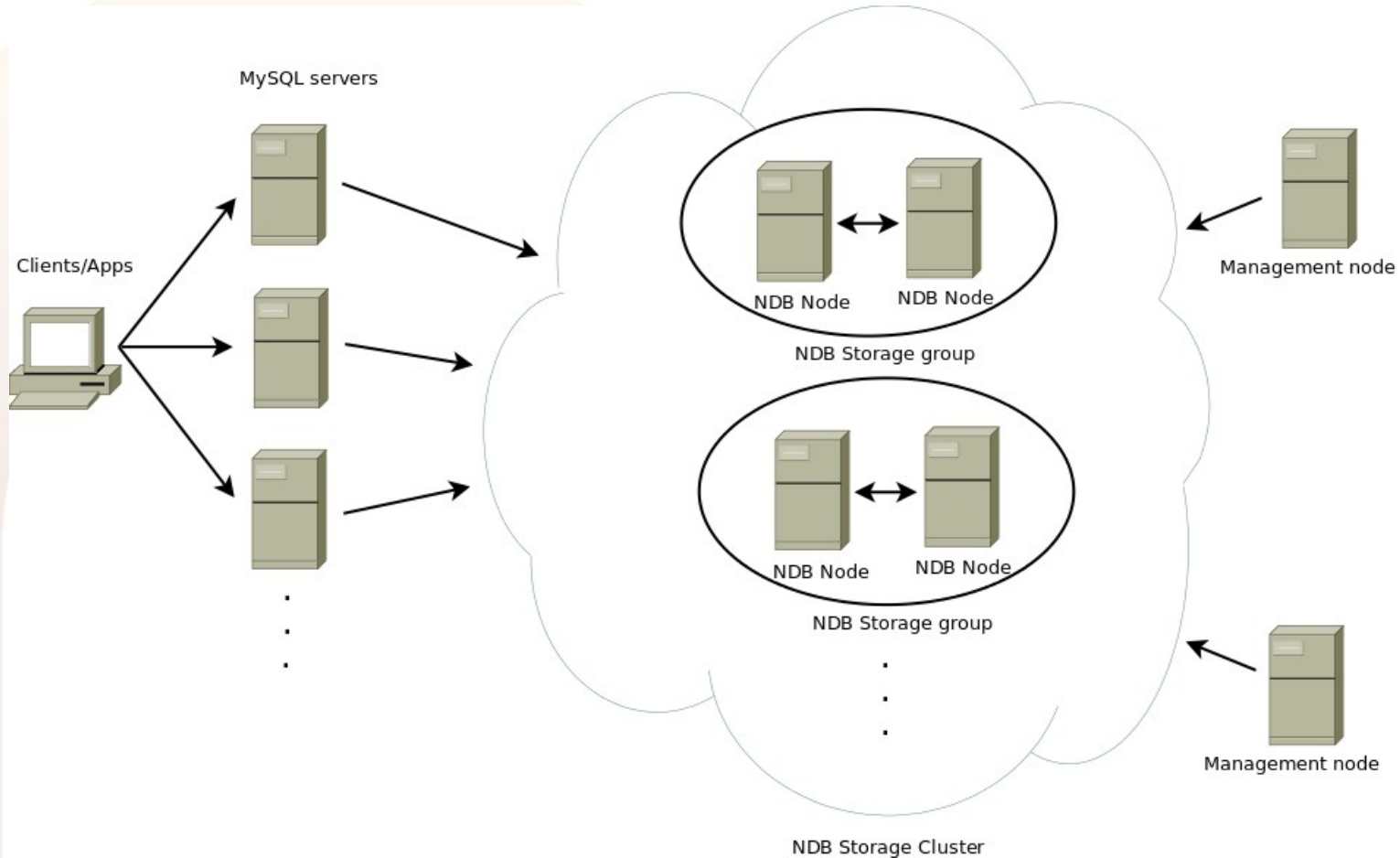
- What can it do?

- Can easily scale with no application change
- Can provide very high availability levels, up to 5 9s, telco grade HA
- Can mix NoSQL and SQL, it is a MySQL SE
- Can handle a very high write load
- Can reach 1 Million transactions/s



NDB/MySQL Cluster

- Quick Architecture diagram



NDB Pros/Cons

Pros

- Built-in redundancy and failover
- Built-in sharding
- Built-in distributed backup
- ACID!!!
- Has a SQL access
- NDB API very fast
- Replication (with MySQL)
- Transactional
- Much more mature

Cons

- High Administration costs
- Complex
- Resource intensive for non testing, NoOfReplica = 2 is a must
- Joins and complex queries can be slow
- NDB API is... let's say, not for beginners

Tokyo Cabinet/Tyrant

- Tokyo Cabinet is a database management library that allows for data storage in multiple formats
 - Formats include hash, table, fixed length rows
- Tokyo Tyrant is the network Daemon that sits on top of a Tokyo Cabinet database
- Written for use on the mixi website

Tokyo Cabinet/Tyrant

Pros

- Easy to implement
- Hash table behaves like memcached, only persistent
- Very Fast
- Lots of flexibility
- Ability to use memcached API allows for you to use with existing code
- Replication
- Embeddable

Cons

- Not Durable
- Single write thread
- Minimal documentation
- Small Community
- Scalability Uncertainty

Cassandra

- What is it...

“The Apache Cassandra Project develops a highly scalable second-generation distributed database, bringing together Dynamo's fully distributed design and Bigtable's ColumnFamily-based data model.”
- Developed and release by facebook, seemingly left hanging in the wind until the last 12-18 months.
- A lot of “Buzz” and name websites starting to make heavy use of it
- Performance by distribution

Cassandra

Pros

- Larger Community than most other NoSQL Solutions
- Very easy to add nodes, scale as needed
- More than a key-value
- Configurable write settings
- Replication/HA
- Java... good or bad thing?
- Changing schema can be easy

Cons

- Inconsistent Experience (Thrift/Api related)
- Eventually Consistent
- My Tests show mixed performance results, Benefit from lots of nodes
- Not as polished as other solutions
- Durability concerns
- Limited config
- Higher ramp up time than other solutions

API Oddities

- Challenging on certain version of Ubuntu to get the ruby api installed, others worked perfectly
- Perl API was a little hard to setup and use
 - find very little documentation
 - Posts mention it works on one version but not another
- Ruby api or maybe thrift in general seem rather slow.
 - 16 threads pegs client, but server is not touched was only getting 400 rows loaded per second, while the same config for mysql loaded near 1800 rows per second

API benchmarks

Empty DB

- **MySQL**

- @ 16threads I get 1300-1400 trans per second
- @ 24threads I get 1400-1500 trans per second
- @ 32threads I get 1800-1900 trans per second
- @ 48threads I get 2100-2200 trans per second
- @ 64threads I get 2200-2300 trans per second
- @ 128threads I get 2400-2500 trans per second

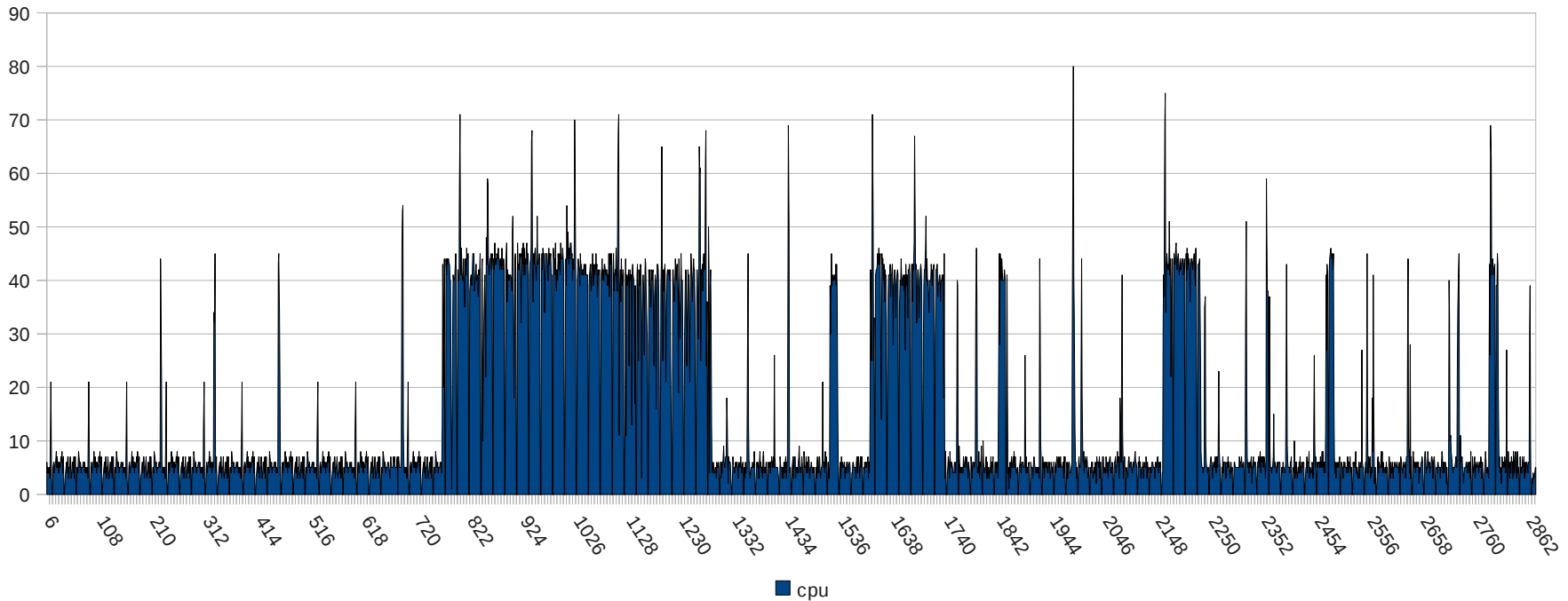
- **Cassandra**

- @ 16threads I get 550-650 trans per second
- @ 24threads I get 650-750 trans per second
- @ 32threads I get 700-800 trans per second
- @ 48threads I get 700-800 trans per second
- @ 64threads I get 700-800 trans per second
- @ 128threads I get 600-700 trans per second

Use Case: Heavy Write Load?

- Cassandra Has great write speeds, until you need to start sync data (Garbage Collection Issues)

Cassandra CPU During Write Load



Cassandra Use Cases

- Most publicized use case is the twitter example with time series data.
- When you have lots and lots of data
- You cannot shard your DB
- Your application does not require the complexities of a SQL Database

Redis

- What is it...

Redis is an open-source, networked, in-memory, persistent, journaled, key-value data store similar to memcached.

- Development funded by VMWare
- Currently GA, version 1.2.0
- BSD License
- A kind of hybrid between Memcached and Tokyo Cabinet
- In memory data with persistence

Redis

- Not only a key/value store
 - counters
 - Lists ($O(1)$ pushpop simultaneous op)
 - Sets
 - Ordered sets
- Adjustable persistence
- Replication, easy slave provisioning with the “sync” command
- Text based protocol (Telnet!)
- Written in C

Redis

Pros

- Simple (less than 1min compile time)
- Persistence
- Built-in replication
- Good language support
- Client based sharding (aka Memcached)
- Async persistence
- Can be fully durable

Cons

- Single-threaded!!! (but no lock contention)
- Data must fit in memory

Redis Use Cases

- Memcached replacement, adding cache persistence and replication
- Lists can be used for a simple queue manager
- Lists can be used for a simple log ring buffer
- With lists, sets and ordered sets, much more general applications can be built.

Many more!

- There are many more solutions available
 - Wanted to include mongodb
 - Many more look interesting
- Additional cases will be posted via blog as time permits

Basic Deployment Guidelines

Pitfalls to Avoid

- Doing it because everyone else is
- Thinking a platform change is a magic bullet
- Assuming a Black box
 - Still have to backup, monitor, etc
- One size does not fit all
 - Mix and Match for a best of breed
- Not Planning or Testing

Important Questions to Ask

- What is your tolerance for data loss?
- How tolerant are you risk?
- What happens if you get inconsistent data returned?
- How big is your dataset?
- What is the goal of this app?
- What are the reporting requirements?
- What is the estimated read/write split?
- Realistically how big will the application get?

The Chicken or the Egg

If you rewrite your application to take advantage of a NOSQL solution, was the rewrite the major source of the speed up or was it NOSQL?

- Ruby On Rails App
- Conversion to NoSQL allowed for Substantial improvement in performance
- The improvement however was artificial:
 - The rewrite of portions of there code made the app much more efficient
 - Active Record in this case was not playing nice with mysql (2-3K queries on a page!)
- Moral : do not always blame the DB for an inefficient app

Recap

- Data requirements are making us rethink how data is stored
 - SSD's, Data optimizations, bigger boxes, appliances, nosql, etc are all trying to solve the “data” problem
- People demand something that is simple and gives them more control
- New Applications have new challenges that require outside the box thinking

Time Check?

More detailed Use Cases

Use Case: Session Data

- Game Session data
- Website session data
 - Often used to fail over app servers
- This data is limited in scope... often just as long as the user is logged in.
- Typically HEAVY write load

Use Case: Session Data

- For non-persistent data memcached is still a very solid choice.
- Heavy load environments (like big MMOPG) sometimes use NDB cluster in non-disk mode
- Persistent session data would also excel in something like Tokyo Tyrant.

Use Case: Key-Value

- In most modern rdbms's this is a lookup via PK.
- These apps operate very similar to the session data, they do not joins or aggregates to speak of.
- Similar to gaming sessions I have seen these in a lot of gaming companies that store things like lists of players items, missions completed, etc.

Use Case: Key-Value

- Memcached is not persistent, so it typically is ruled out fairly quickly.
- MySQL does an admirable job, but you need to manage sharding on your own.
- Cassandra is slow on a single or with few nodes but can add a lot of horsepower
- Merging Tokyo Tyrant and the memcached API is interesting as you can use the Hash algorithm and get persistence (not durable however)

Use Case: Data Warehouse

- A datawarehouse is a data store specifically designed to facilitate strategic and long term reporting
- These typically are very large, with lots of data
- Response time is not typically priority #1
- Many batch jobs
- Goal is Business Intelligence

Datawarehouse Confusion

- Reporting is king... lack of knowledge of proper data warehousing techniques has led to bloated evil looking databases
- Storing every record from every click in a “database” is not data warehousing.
 - This gets people into a lot of trouble, it does not scale, it destroys performance
- Your looking at aggregated data here, not the individual
- The traditional DW downfall is you have to plan. Its not as agile as storing raw records.

Use Case: Data Warehouse

- The Relational Database, specifically MySQL with an array of “pluggable” storage engines designed for datawarehousing still is the best fit.
- MongoDB seems to have some interesting features, but we ran out of time before we could put them through the paces
- Many NOSQL tools can scale to handle massive amount of data, but are not designed to or can compete with a full blown DW and BI suite.

Use Case: Log Processing

- You want to crunch and munge through massive amounts of data

The great debate

- What comes first the chicken or the egg?
Performance or New Architecture?
 - By simplifying your “database and data storage”, re-factoring your code, and removing legacy bloat did performance increases come from a switch to NOSQL or the Code Re-factoring?
 - Scary is many people are more willing to completely rip apart code in an effort to move to a new database or new NOSQL solution then they are to rebuild around the same architecture
 - Bad Code and Design are more of a detriment to performance then internal database issues

Hybrid

- Best approach takes advantage of both SQL and NOSQL
- Example:
 - User data in Cassandra
 - Transactional data in MySQL
 - Session Data in memcached

Questions?

- matt@percona.com
- yves@percona.com