THE CUSTOMER SUCCESS PLATFORM

# HBase+Phoenix for OLTP

**Andrew Purtell**
Architect, Cloud Storage @ Salesforce
Apache HBase VP @ Apache Software Foundation
apurtell@salesforce.com
apurtell@apache.org
@akpurtell

v4

## whoami

Architect, Cloud Storage at Salesforce.com

Open Source Contributor, since 2007

- **Committer, PMC, and Project Chair, Apache HBase**
- **Committer and PMC, Apache Phoenix**
- Committer, PMC, and Project Chair, Apache Bigtop
- Member, Apache Software Foundation

Distributed Systems Nerd, since 1997

## Agenda

http://riverlink.org/wp-content/uploads/2014/01/grab-bag11.jpg

## HBase+Phoenix for OLTP

Common use case characteristics

- Live operational information
- Entity-relationship, one row per instance, attributes mapped to columns
- Point queries or short range scans
- Emphasis on update

Top concerns given these characteristics

- Low per-operation latencies
- Update throughput
- Fast fail
- Predictable performance

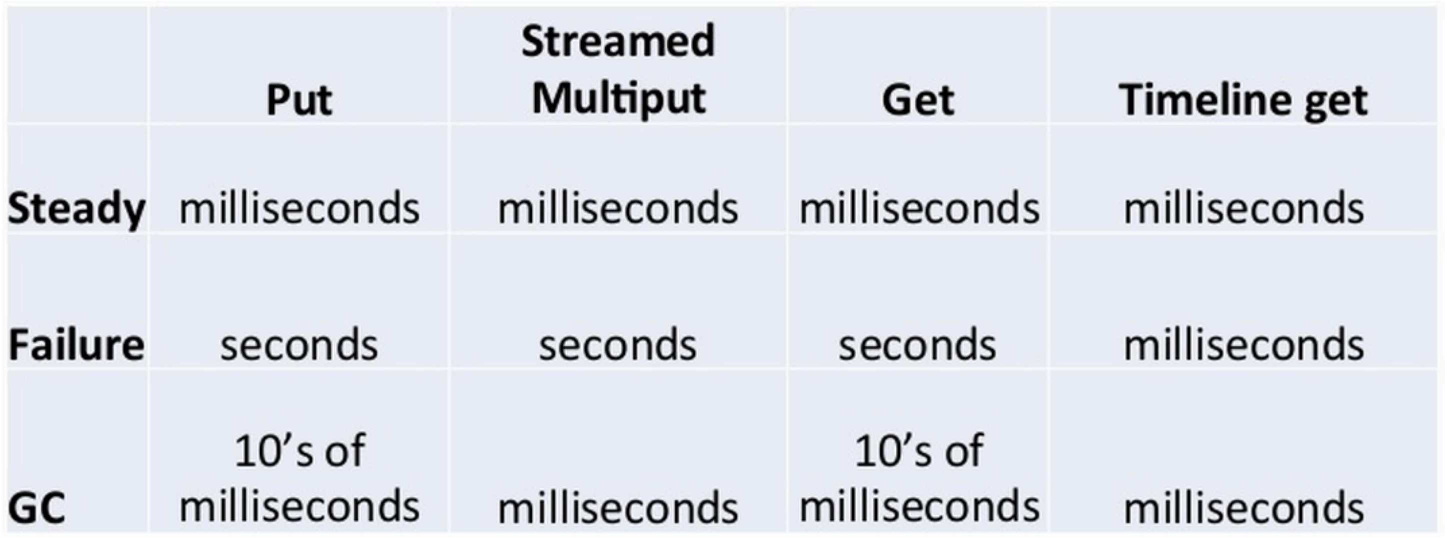
http://www.cn-vehicle.com/prodpic/2011-3-21-16-23-37.JPG

# Low Per-Operation Latencies

Major latency contributors

- Excessive work needed per query
- Request queuing
- JVM garbage collection
- Network
- Server outages
- OS pagecache / VMM / IO

Typical HBase 99%-ile latencies by operation



| | Put | Streamed Multiput | Get | Timeline get |
|---|---|---|---|---|
| **Steady** | milliseconds | milliseconds | milliseconds | milliseconds |
| **Failure** | seconds | seconds | seconds | milliseconds |
| **GC** | 10's of milliseconds | milliseconds | 10's of milliseconds | milliseconds |

NOTE: Phoenix supports HBase's timeline consistent gets as of version 4.4.0

# Limit The Work Needed Per Query

Avoid joins, unless one side is small, especially on frequent queries

Limit the number of indexes on frequently updated tables

Use covered indexes to convert table scans into efficient point lookups or range scans over the index table instead of the primary table

- CREATE INDEX index ON table ( … ) *INCLUDE ( … )*

Leading columns in the primary key constraint should be filtered in the WHERE clause

- Especially the first leading column
- IN or OR in WHERE enables skip scan optimizations
- Equality or <, > in WHERE enables range scan optimizations

Let Phoenix optimize query parallelism using statistics

- Automatic benefit if using Phoenix 4.2 or greater in production

## Tune HBase RegionServer RPC Handling

hbase.regionserver.handler.count (hbase-site)

Set to cores x spindles for concurrency

Optionally, split the call queues into separate read and write queues for differentiated service

hbase.ipc.server.callqueue.handler.factor

- Factor to determine the number of call queues: 0 means a single shared queue, 1 means one queue for each handler

hbase.ipc.server.callqueue.read.ratio (hbase.ipc.server.callqueue.read.share in 0.98)

- Split the call queues into read and write queues: 0.5 means there will be the same number of read and write queues, < 0.5 for more read than write, > 0.5 for more write than read

hbase.ipc.server.callqueue.scan.ratio (HBase 1.0+)

- Split read call queues into small-read and long-read queues: 0.5 means that there will be the same number of short-read and long-read queues; < 0.5 for more short-read, > 0.5 for more long-read

## Tune JVM GC For Low Collection Latencies

Use the CMS collector

-XX:+UseConcMarkSweepGC

Keep eden space as small as possible to minimize average collection time. Optimize for low collection latency rather than throughput.

-XX:+UseParNewGC – Collect eden in parallel

-Xmn512m – Small eden space

-XX:CMSInitiatingOccupancyFraction=70 – Avoid collection under pressure

-XX:+UseCMSInitiatingOccupancyOnly – Turn off some unhelpful ergonomics

Limit per request scanner result sizing so everything fits into survivor space but doesn't tenure

hbase.client.scanner.max.result.size (in hbase-site.xml)

- Survivor space is 1/8th of eden space (with -Xmn512m this is ~51MB )
- max.result.size x handler.count < survivor space

## Disable Nagle for RPC

Disable Nagle's algorithm

TCP delayed acks can add up to ~200ms to RPC round trip time

In Hadoop's core-site and HBase's hbase-site

- ipc.server.tcpnodelay = true
- ipc.client.tcpnodelay = true

In HBase's hbase-site

- hbase.ipc.client.tcpnodelay = true
- hbase.ipc.server.tcpnodelay = true

Why are these not default? Good question

## Limit Impact Of Server Failures

Detect regionserver failure as fast as reasonable (hbase-site)

- zookeeper.session.timeout <= 30 seconds – Bound failure detection within 30 seconds (20-30 is good)

Detect and avoid unhealthy or failed HDFS DataNodes (hdfs-site, hbase-site)

- dfs.namenode.avoid.read.stale.datanode = true
- dfs.namenode.avoid.write.stale.datanode = true

## Server Side Configuration Optimization for Low Latency

Skip the network if block is local (hbase-site)

- dfs.client.read.shortcircuit = true
- dfs.client.read.shortcircuit.buffer.size = 131072 – important to avoid OOME

Ensure data locality (hbase-site)

- hbase.hstore.min.locality.to.skip.major.compact = 0.7  (0.7 <= n <= 1)

Make sure DataNodes have enough handlers for block transfers (hdfs-site)

- dfs.datanode.max.xcievers >= 8192
- dfs.datanode.handler.count – match number of spindles

## Schema Considerations

### Block Encoding

Use FAST_DIFF block encoding

```
    CREATE TABLE … (
        …
    ) DATA_BLOCK_ENCODING=‘FAST_DIFF’
```

FAST_DIFF encoding is automatically enabled on all Phoenix tables by default

Almost always improves overall read latencies and throughput by allowing more data to fit into blockcache

Note: Can increase garbage produced during request processing

## Schema Considerations

### Salting

Use salting to avoid hotspotting

```
CREATE TABLE … (
    …
) SALT_BUCKETS = N
```

Do not salt automatically. Use only when experiencing hotspotting

Once you need it, for optimal performance the number of salt buckets should approximately equal the number of regionservers

## Schema Considerations

### Primary key (row key) design

Primary key design is the single most important design criteria that drives performance

Make sure that what ever you're filtering on in your most common queries drives your primary key constraint design

Filter against leading columns in the primary key constraint in the WHERE clause, especially the first

Further advice is use case specific, suggest writing user@phoenix.apache.org with questions

## Optimize Writes For Throughput

Optimize UPSERT for throughput

UPSERT VALUES

- Batch by calling it multiple times before commit()
- Use PreparedStatement for cases where you're calling UPSERT VALUES again and again

UPSERT SELECT

- Use connection.setAutoCommit(true), pipelines scan results out as writes without unnecessary buffering
- If your rows are small, consider increasing phoenix.mutate.batchSize
    - Number of rows batched together and automatically committed during UPSERT SELECT, default 1000

## Fast Fail

For applications where failing quickly is better than waiting

Phoenix query timeout (hbase-site, client side)

- phoenix.query.timeoutMs – max tolerable wait time

HBase level client retry count and wait (hbase-site, client side)

- hbase.client.pause = 1000
- hbase.client.retries.number = 3
- If you want to ride over splits and region moves, increase hbase.client.retries.number substantially (>= 20)

RecoverableZookeeper retry count and retry wait (hbase-site, client side)

- zookeeper.recovery.retry = 1 (no retry)

ZK session timeout for detecting server failures (hbase-site, server side)

- zookeeper.session.timeout <= 30 seconds (20-30 is good)

## Timeline Consistent Reads

For applications that can tolerate slightly out of date information

HBase timeline consistency (HBASE-10070)

- With read replicas enabled, read-only copies of regions (replicas) are distributed over the cluster
- One RegionServer services the default or primary replica, which is the only replica that can service writes
- Other RegionServers serve the secondaries replicas, follow the primary RegionServer and only see committed updates. The secondary replicas are read-only, but can serve reads immediately while the primary is failing over, cutting read availability blips from ~seconds to ~milliseconds

Phoenix supports timeline consistency as of 4.4.0

1. Deploy HBase 1.0.0 or later
2. Enable timeline consistent replicas on the server side
3. ALTER SESSION SET CONSISTENCY = 'TIMELINE'
    - or set the connection property 'Consistency' to "timeline" in the JDBC connect string
    - or set 'phoenix.connection.consistency' = "timeline" in client hbase-site for all connections

## OS Level Tuning For Predictable Performance

Turn transparent huge pages (THP) off

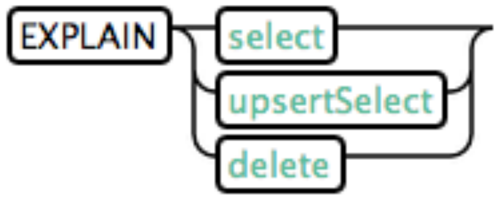
```
echo never > /sys/kernel/mm/transparent_hugepage/enabled
echo never > /sys/kernel/mm/transparent_hugepage/defrag
```

Set vm.swappiness = 0

Set vm.min_free_kbytes to at least 1GB (8GB on larger memory systems)

Disable NUMA zone reclaim with vm.zone_reclaim_mode = 0

# EXPLAINing Predictable Performance



Check the computed physical plan using EXPLAIN

Consider rewriting queries when:

Prefer operations on SERVER, not CLIENT

- SERVER ops are distributed over the servers and execute in parallel
- CLIENT ops execute within the single client JDBC driver
- Consider tweaking your query to increase the use of SERVER side operations

Scanning strategy is TABLE SCAN, prefer RANGE SCAN or SKIP SCAN

- Filter against leading columns in the primary key constraint, PK may need redesign
- Possibly means you need to introduce a global index that covers your query or local index
- If you have an index but the optimizer is missing it, try hinting the query (SELECT /*+ INDEX(<table> <index>) */ ...)

If using JOINs, please read https://phoenix.apache.org/joins.html

## What's New?

Exciting new or upcoming features for OLTP use cases

Functional Indexes

Spark integration

- Make RDDs or data frames out of fast Phoenix queries for Spark streaming and batch workflows

Transactions (WIP)

- Using Tephra (http://tephra.io/), supports REPEATABLE_READ isolation level
- Need to manage the Tephra Transaction Manager as a new system component
- Work in progress, but close to release, try out the 'txn' branch

Calcite Integration for federated query (WIP)

- Interoperate and federate queries over other Apache Calcite adopters (Drill, Hive, Samza, Kylin) and any data source with Calcite driver support (Postgres, MySQL, etc.)
- See the 'calcite' branch

# What’s New?

## Exciting new or upcoming features for OLTP use cases

Query Server

- Builds on Calcite’s Avatica framework for thin JDBC drivers
- Offloads query planning and execution to different server(s) – The “fat” parts of the Phoenix JDBC driver become hosted on a middle tier, enablig scaling independent from clients and HBase
- Shipping now in 4.5+
- Still evolving, so no backward compatibility guarantees yet
- A more efficient wire protocol based on protobufs is WIP

salesforce

## Thank you

Q&A

apurtell@salesforce.com

apurtell@apache.org

@akpurtell