HBASE-10994: HBase Multi-Tenancy

Problem Overview

Currently HBase treats all tables, users, and workloads in the same way.
This is ok, until multiple users and workloads are applied on the same cluster/table. Some workloads/users must be prioritized over others, and some other workloads must not impact others.

We can separate the problem into three components.

- Isolation/Partitioning (Physically split on different machines)
- Scheduling (Prioritize small/interactive workloads vs long/batch workloads)
- Quotas (Limit a user/table requests/sec or size)

Examples:

- Isolation/Partitioning example:
  If on the same cluster you have a test table and a production table, long queries on the test table may impact performance on the production table.
  *This can easily be solved by adding a set of **isolation** rules: The test table should use machine A, B, C the Production Table should use Machine D, E, F.*
- Scheduling example:
  If a Map-Reduce Job is running on a live-serving table, the small interactive requests from the users may be slowed down by the long requests coming from the MR job.
  *This can be solved by applying some **scheduling** rules: e.g. small queries should be prioritized over long queries.*
- Quotas example:
  Prevent a user to use too much space in the cluster, or use too much bandwidth.
  This can be solved by applying some scheduling rules and storing user/table stats.

Usage Assumptions

- Get and Small Scans are typical from a user request or real-time application; in general they should have higher priority than other requests.
- Large/Full-Table Scans are typical of Map-Reduce Jobs; in general they should have lower priority than other requests and they may be “throttled”.
- Inserts/Updates on a specific table should have uniform size.
- The key distribution (update, scan, get) distribution should be uniform.
- Large Insert/Updates (e.g. images) should be isolated on its own table/family.
- Some tables may be “hot” and some may be not.

Terms

- **Users**: (HBase Shell, Client App, MR job)
- **Tables**: Is the smallest unit of scalability in this document. Unless we are talking about "Global Stats (Multiserver aggregation)" when there is a reference to "Table" will be the set of regions for that table on a specific Region Server. *(The user should only know about the table not the division done internally by HBase to scale out)*
- **Namespaces**: Will not be mentioned much in this document, since we are considering the smallest unity of scalability to make the examples, and namespaces are just a group of tables, so everytime you see "Table" you can replace it with "Namespace". A rule applied to a namespace will be applied to all the tables in it.

Isolation/Partitioning

Isolation and partitioning is useful, in situation where the "admin" knows exactly what is the workload of each table and knows how to manually separate them, to reserve enough resources to run everything "smoothly". *(The current way to achieve this today, is by setting up one cluster per use case)*

The admin should be able to:

- Isolate "Table A" on a specify set of Region Servers *(See HBASE-6721 RegionServer Group based Assignment)*
- Allow only N Regions/size of "Table A" to be assigned on a specify set of Region Servers

NOTE that we also have to force HDFS to follow these isolations rules.

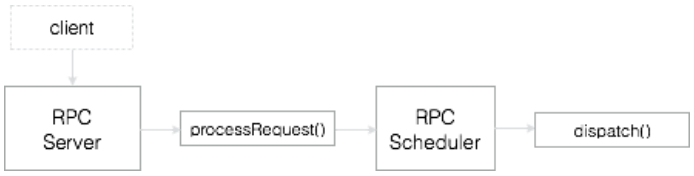
From an implementation point of view the isolation/partitioning will be implemented as a set of Assignment-Manager/Load-Balancer rules.

Scheduling

Scheduling is used to prioritize a set of requests. Example:

- Prioritize requests for "Table T" of "Type R" (get, small-scan, ...)
- Prioritize requests from "User U" for "Table A" of "Type R"…

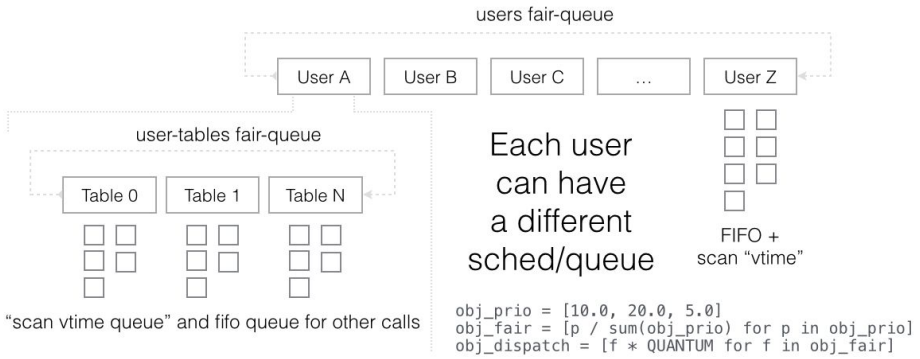
From an implementation point of view the scheduling is just a pluggable RPC Scheduler. HBase currently has already a pluggable RPC scheduler. Each request is parsed and passed to the RPC scheduler. The current implementations are the FifoRpcScheduler, which is a single FIFO queue scheduler and a SimpleRpcScheduler which keeps 3 FIFO queues ("priorityq", "replicationq", "callq").



A first set of “hardcoded” rules can be enforced to mitigate the MR vs user-query problem:

- Add a “vtime” field to the scanner, so the longer a scan request lives the less priority it gets.
- Scans can be “truncated” to prevent blocking others. (e.g. if a user set max-data-size 1GB we can force to return few MB)
- Enforce a Request-Type priority (e.g. Put, Small Scan, Large Scan) on a write heavy table.

A more advanced way to schedule requests is to have allow each user to have its own “scheduling policy” to allows each user to define priorities for each table, and for each table define request-types priorities.



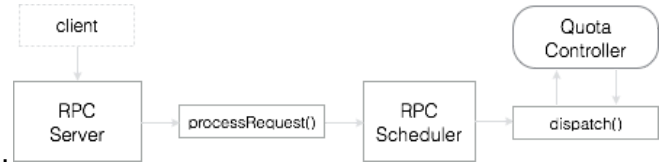
We can store the “base” priorities in a table along with the user quotas *(see next section)* and add a new “priority” flag to each request to replace the default value. (e.g I want a one time get to be executed as first thing on a low priority table).

Quotas

Quotas can be used to limit the number of table/regions or requests in the system:

- Global quotas
  - limit: number of table/regions per machine *(HBASE-8410 Basic quota support for namespaces)*
- Table level quotas
  - throttling: req/sec, data/sec
  - limit: table size, number of regions
- User level quotas
  - throttling: req/sec, data/sec
  - limit: max number of table/regions (total)

From an implementation point of view quotas will be part of scheduling, since we have to block/limit/throttle incoming requests based on the usage.



The simplest way to implement quotas is by using a “token bucket” algorithm (but we will have an interface to make it pluggable).

- For Put requests we know the exact amount of data we are inserting.
- For Scan requests we can estimate and enforce the limit on the server side.

Global stats vs RS stats:
The main problem is sending “usage stats” update across the cluster.

Let say that we want to limit a user to 10 query/sec. Assuming that the requests are distributed across multiple machines each machine must know the exact usage of the user. Each request can trigger an update in ZooKeeper, but this will end up to be a huge bottleneck.

We can workaround this problem by assuming a uniform use on each machine, and divide the total quota for the user for the total number of the machines. Each request will update the “usage stats” only on the local machine. We get the performance back, but we end up with a lower-cap if we have a “large” scan that hits a single RS.

Another way to solve the problem is to be less strict on the limit. Instead of updating the global stats every request, we keep and enforce the limits on the RS and delay the stats update. This means that we may be able to serve more than 10 query/sec, so instead of having a constant query/sec we have spikes with dead period to catch up with our quota.

What can we do with the current Code?

Assignment/Load Balancer Rules

We have the pluggable balancers but not a chain of balancer rules.
This means that if we are going to implement a balancer based on a “fixed partition scheme” we have to “reimplement” the internal balancing based on the RS load or Favored Nodes…

so the first step will be adding the concept of chained rules, to allow to internally balance each partition with one of the balancer that we already have. Then we have to define a way to describe partitions and hotness for each table.

IPC Scheduling

The HBase RPC engine has already a pluggable Interface, and each request has a *username* and a *table name* associated. *(NOTE that the username in case of auth “simple” is just the shell login)*.

We can start by adding the “vtime” to the scanner, and add a new “TypeBased RPCScheduler” which put each request in its own “type queue” (small requests, large requests, writes), which should improve a bit the situation of the current MR vs user requests problem.

Similar to what we should do with the balancer policies, we should add a chained-rpc scheduler policy if we want to quotas and table priorities.

Future/Out of scope (Aka Current “Problems”)

We can schedule requests to prioritize and/or throttle tables and users based on defined policies. But this will not improve predictability or give us any extra guarantee.

- We don’t know which requests can be served directly from cache
- We are not able to concurrently schedule requests for different disks
- We don’t have any idea on how busy is the HDFS node underneath
- We should only concurrently schedule requests on different “locks” (e.g. same table different row, or different tables) to avoid waiting on the object lock.

also we are relying on the user to play nice

- A Put with 10k data will take twice more than put with 5k data
    - We can’t split batch request since we guarantee “atomicity”
- We may be able to split scan requests in smaller chunk or just throttle them and add more threads to fake better concurrency.
- We always assume that each scan or write request has its own disk.. but it is not true
    - Just because you have a large file or you are writing sequentially doesn’t mean that you are doing seq I/O
    - If you are hitting a single disk and
        - you have concurrent request on N hfiles you are not doing 20M seq io
        - you have the fsync() for the wal you are probably not reading
    - (more in general we don’t control our I/O path)

Testing

All the testing depends on which “configuration” are you using. There is no single on/off switch here.

- User Priorities (IPC Scheduling)
    - Run a MR Job like CopyTable and a set of get/short scan
        - The time taken by get/short scan should be not too dissimilar from the one that takes without the MR job running
    - Run Long-Scan and Get/Short-Scan using same user and multiple thread
        - …
- Table Priorities (IPC Scheduling)

    - Two tables with the same content, run CopyTable. The one with the higher priority should end quickly.
    - …
- Table Partitioning (Assignment/Load Balancer)
    - scan hbase:meta to verify that the rules are respected
    - simple partitioning each table on a different machine, the cluster stats at the end of CopyTable should show the same results.
    - …
- Quota Limit
    - Verify that the ops/sec or the data/sec is respected on the client side (in theory we can monitor the “stats table”)
    - …

Roadmap

Here is a suggested list of steps, to get to the “full multi-tenancy” support with a set of “small” patches.

- HBASE-10993: Deprioritize Long-Running Scan, by adding a “vtime” field to the Scanner.
    - *This will improve the current “performance” situation when running user queries together with Map-Reduce jobs.*
- HBASE-11355/HBASE-11724: Add a TypeBasedRpcScheduler
    - *The first cut will be just to prioritize “reads or writes”*
- Refactor LoadBalancer to be chainable
    - *e.g. We want to be able to reuse the code of favored-nodes balancer when we will have the “Region Group Assignment”.*
- Refactor RPCScheduler to be chainable
    - *Instead of having a single class with all the code, we should split the various policies and allow each user to select a set of policies. e.g. long-running scan deprioritization + write prioritization + ...)*
- Add “user fairness” rpc scheduler
    - *Instead of having a single queue for all the request each user will get its own queue with the type based scheduler as default for each user.*
- HBASE-11598: Introduce “User Settings/Quotas” Table/Manager
    - (This requires a “cache/notification bus” HBASE-9864)
    - HBASE-11598: Add simple rate limiter to the rpc scheduler (as first use case)
        - Push back on the client if quota exceeded
    - Add pluggable rpc scheduler for each user
    - ...