[Timeline Service Nextgen] Flow/User/Queue Level Aggregations

Introduction

Per previous discussions in design documents for YARN-2928, the basic scenario is the query for stats can happen on:

- Application level
    - return an application with aggregated stats
- Flow level:
    - expect return: aggregated stats for a flow_run, flow_version and flow
- User level:
    - expect return: aggregated stats for applications submitted by user
- Queue level:
    - expect return: aggregated stats for applications within the Queue

Application states is the basic building block for all other level aggregations. We can provide Flow/User/Queue level aggregated statistics info based on application states (a

dedicated table for application states is needed which is missing from previous design documents like HBase/Phoenix schema design).

Properties of various aggregations

- Storage Regions touched by queries: Single Storage server, multiple servers etc
    - App level aggregation should mostly happen on a single region (by properly define split strategy).
    - User and Flow level aggregation could involve small number of regions aggregation because of our current design for primary keys (putting User ID first).
    - Queue level aggregation could involve massive number of regions because it could include applications from many different users.
    - This means: the cost of aggregation is: Queue level > Flow/User level > Application level.

- Expectation on aggregation freshness
    - We should have cheaper aggregations happen more often while keeping expensive things happen less.
    - Low cost: 15 sec - 1min
    - Mid cost: 5 ~ 15 mins.
    - High cost: 30 ~ 60 mins

Key Assumptions

- Data source for v2 TimelineService is from: AM/NM/RM so far, most info get delivered to App-level collector first (except info from RM) and stream into backend storage via HBase/Phoenix API.

- Data type include: event/metrics/configuration/relationship, metrics info is our focus for aggregation. The query for other info can simply happen on the fly.

- Metrics data include time series (assume clock get synchronized on each machine), and aggregation on each level should apply on specific time range.

- Interval for gathering metrics data and storing aggregation data are configurable: the former one is at YARN cluster level and the later one is at Timeline service level.

Several options to do aggregation

1. Dedicated aggregators to do aggregation

For Flow/User/Queue level aggregation, this means application timeline collectors need to forward related metrics info to dedicated Flow/User/Queue aggregators where info get aggregated and push to backend which has following concerns:
1. Duplicated traffic for metrics info
2. Flow/User/Queue aggregator failed over case could be complicated which make aggregated states get delayed and even unpredictable

For app level aggregation, the 2nd concern above is still valid. However, just as metrics data could also be lost when NM failure happen during container monitoring interval. Comparing with outstanding pressure to backend storage for aggregation on all metrics records update, the cost of losing per app metrics during small interval due to app collector failure could still be acceptable. Thus, we can have app-level aggregator to cache metrics locally and do aggregation in a configurable time interval at application level.

2. Aggregation in backend storage - HBase coprocessor

One other idea is to leverage feature provided by backend storage, like: HBase coprocessor, to aggregate info against HBase columns or column families directly across regions or even tables.

Pros:
No/less additional traffic, can be no delay (region observer model), serving query fast, simple in failure cases.
Cons:
Additional pressure to backend storage (as we cannot leverage resources outside of region servers, like BigTable does).
The solution limited to HBase backend storage only.

Also, there are two options here for using coprocessor:

a. Observer model

Like trigger in relational database, we can deploy customized code (observers) running in server side, that get triggered automatically when operations (get, put, scan, etc.) on Region/WAL/Master happens.
Cons:

- Mostly for simple operation (permission check, filter, etc.).
- More complex for operations that need to cross regions (our table is big enough to have multiple regions) - two stage aggregation logic could be required.
- Not very flexible with a customized trigger/aggregation time interval.

Not quite fit for our case as we don't expect the aggregation could happen per record update.

b. Endpoints model

Like stored procedure in database, client can remotely invoke customized code (endpoints) in server side. The endpoints executed on target regions which can return region-level aggregated results. Then, client can simply do aggregation on returned results for all regions and have final aggregated states. The whole process is similar to mapreduce.

Pros: Straightforward for developing, flexible time interval for doing aggregation in different levels.
Cons: extra traffic could be involved when getting results from numerous regions. However, this shouldn't be a big problem given return results are aggregated info.

3. Aggregation happen on the read path

ATS reader may aggregate these metrics on the fly and serve them.
Pros: flexible, as we don't need aggregate info schema prepared ahead
Cons: could be slow in query as it do aggregations on the fly. Also, duplicated aggregations could be done during similar queries (assume no/less effective query result cache).
Obviously, not quite fit for metrics data.

Proposed solutions for App/Flow/User/Queue level aggregation cases

Application level aggregation

We need application level aggregation for two reasons:

- We need to present end user aggregated states for each application, include information like: resource (CPU, Memory) consumption, number of containers get launched/completed/failed, framework metrics, etc. We need this for apps while they are running as well as when they are done.
- Other level (Flow/User/Queue) aggregation can be more efficient to be based on Application-level aggregations rather than raw entity-level data as much less raws need to scan (with filter out non-aggregated entities, like: events, configurations, etc.).

The ApplicationState table stores time series data for application metrics as well as final states. It can be split into two tables by aggregated from RMTimelineCollector or AppLevelTimelineCollector.

ApplicationState Table (aggregated from RMTimelineCollector*)*

PKs: User_ID + Cluster_ID + Flow_ID + FlowRun_ID + Application ID	**Number of Attempts**	**Container Aggregate Metrics**	**Resource Metrics**	**Time: start, end, last_modification**
… + application_1431113219200_0001	1	allocated: 0 preempted:0 failed: 0 reuse: 0	mem-allocated: 0 cpu-allocated: 0 resource-preempted: (0, 0)	start: 1431113213456 end: last_modification: 1431113213456

ApplicationState Table (aggregated from AppLevelTimelineCollector*)*

PKs: User_ID + Cluster_ID + Flow_ID + FlowRun_ID + Application ID	Container Aggregate metrics	Resource Metrics	Framework Metrics	Time: start, end, last_modification
… + application_1431113219200_0001	allocated: 0 preempted:0 failed: 0 reuse: 0 (Note: belows are currently not available yet) TotalContainerRequest: 0 ContainerRequestByLocalityNode: 0 ContainerRequestByLocalityRack: 0 ContainerRequestByPriority: 0	pMem-consumption: 0 vMem-consumption: 0 CPU-consumption: 0 Resource-consumption: 0	MR: *HDFS_BYTES_READ: 107374182*, *HDFS_BYTES_WRITTEN*: 268435456	start: 1431113219200, end: last_modification: 1431113219200

Note:
1. UUID for application ID is deferred to some other JIRA to discuss (YARN-3588), we could add redundant info here like user_ID or flow_ID for aggregation efficient purpose.
2. About store time series data, we will put timestamp as version of cells for aggregation tables

Application level TimelineCollector could cache metrics data (source from NM and AM) locally on each metric entities it received for a given time interval (let's say, 15 secs) which is configurable by users. Also, it can do the aggregation on these delta data to ApplicationState table in the same interval before application get finished.
One challenge here is some metrics data is from RM (resource allocated, etc.) which could bypass app-level TimelineCollector. Two option is: RM do aggregation on these

metrics or app-level TimelineCollector achieve it by accessing data from backend storage or calling backend storage features like: HBase endpoints coprocessor.

Flow level aggregation

Flow level aggregation includes three levels aggregation actually:

- The first level is Flow_run level which represents one execution of a flow and shows exactly aggregated data for a run of flow.
- The 2nd level is Flow_version level which represents summary info of a version of flow.
- The last level is Flow level which represents summary info of a specific flow.

The example of Flow_Run_States table, Flow_Version_Summary Flow_Summary table could be something like below:

Flow_Run_States Table

PKs: User, Cluster, Flow, Flow_Run ID	**Number of applications**	**Container Counters**	**Resource Metrics**	**Framework Metrics**	**Time (start, last modified)**
… + Flow_daily_analysis_0001_v1_1431113219200	applications: 3	allocated: 0 preempted: 0 failed: 0 reuse: 0	mem-allocated: 0 cpu-allocated: 0 mem-consumption: 0 cpu-consumption: 0 Resource-consumption: 0	*HDFS_BYTES_READ:* 14952678 *HDFS_BYTES_WRITTEN:* 14952678	start: 1431113229567 last modified:1431113229567

Flow_Version_Summary Table

PKs: User + Cluster + Flow + Flow_Version	Number of Flow_runs/ applications	Container counters (accumulated/mean, etc.)	Resource Metrics (accumulated/mean, etc.)	Framework Metrics (accumulated/mean, etc.)	Time (start, last modified)
… + Flow_daily_analysis_0001_v1	Flow_runs: 2 Finished_Flow_runs: 1 Running_Flow_runs: 1 TotalApplications: 6 ApplicationsPerRun(avg): 3	allocated-acc: 0 preempted-acc:0 failed-acc: 0 reuse-acc: 0 allocated-avg: 0 preempted-avg:0 failed-avg: 0 reuse-avg: 0	mem-allocated-acc: 0 cpu-allocated-acc: 0 mem-consumption-acc: 0 cpu-consumption-acc: 0 Resource-consumption-acc: 0 mem-allocated-avg: 0 cpu-allocated-avg: 0 mem-consumption-avg: 0 cpu-consumption-avg: 0 Resource-consumption-avg: 0	*HDFS_BYTES_READ-mean:* 14952678 *HDFS_BYTES_WRITTEN-acc:* 14952678	start: 1431113229567 last modified:14311132295 67

Flow_Summary Table

User + Cluster + Flow_ID	**Number of Flow_versions/Flow_runs/applications**	**Container counters (accumulated/statistical mean)**	**Resource Metrics (accumulated/statistical mean)**	**Framework Metrics (accumulated/mean, etc.)**	**Time (start, last_modification, avg_execution)**
… + Flow_daily_analysis_0001	Flow_versions: 2 Flow_runs: 4 Finished_Flow_runs: 3 Running_Flow_runs: 1 TotalApplications: 12 ApplicationsPerRun(avg): 3	allocated-acc: 0 preempted-acc:0 failed-acc: 0 reuse-acc: 0 allocated-mean (per flow?): 0 preempted-mean:0 failed-mean: 0 reuse-mean: 0	mem-allocated-acc: 0 cpu-allocated-acc: 0 mem-consumption-acc: 0 cpu-consumption-acc: 0 Resource-consumption-acc: 0 mem-allocated-mean: 0 cpu-allocated-mean: 0 mem-consumption-mean: 0 cpu-consumption-mean: 0 Resource-consumption-mean: 0	*HDFS_BYTES_READ-mean:* 14952678 *HDFS_BYTES_WRITTEN-acc:* 14952678	start: last_modification: avg_execution:

According to [previous discussion], for flow level aggregation, we prefer to leverage feature provided by backend storage, like: coprocessor, to aggregate info against HBase columns or column families directly across regions or even tables.

With endpoints of HBase coprocessor, client (running some places which is discussed later) can trigger the endpoints periodically in a fine grained time interval **(1 mins or less)** as scan rows in ApplicationState table should be lower cost than other levels like user or queue.

The implementation level control flow will be described in “Detail Design for Flow/User Level Aggregation”.

User level aggregation

User level aggregation represents summary info of a specific user, it should include summary info of accumulated and statistic means (by two levels: application and flow), like: number of Flows, applications, resource consumption, resource means per app or flow, etc.
An example of user summary table is as below:
.

User_Summary Table

USER_ID	Number of Flows/Applications	Container counters	Resource Metrics	Cluster resource percentage (avg, peak)	Time: start, last_modification
Dr_Who_007	Flow Num: 10 Finished_Applications: 20 Running_Applications: 10 Failed_Applicaitons: 2	allocated: 0 preempted: 0 failed: 0 reuse: 0	mem-allocated: 0 cpu-allocated: 0 mem-consumption: 0 cpu-consumption: 0 Resource-consumption: 0	avg: 0.05 peak: 0.2	

Similar to flow level aggregation, we could still use endpoints model of HBase coprocessor to do aggregation, the time interval for trigger aggregation should be larger than flow level as it involves scan on more raws/regions, that could be **5 - 10 minutes**.

Queue level aggregation

Queue level aggregation represents summary info of a specific queue, include: number of applications, resource allocated/consumption details, etc.
An example of user summary table is as below:

Queue_Summary Table

Queue_ID	current time	Number of applications	Container counters	Resource Metrics	execution time per app (avg)
Queue_backup_0001	1431113229567	TotalApplications: 6 FinishedApplications: 4 RunningApplications: 1 FailedApplications: 1	allocated: 0 preempted: 0 failed: 0 reuse: 0	mem-allocated: 0 cpu-allocated: 0 mem-consumption: 0 cpu-consumption: 0 Resource-consumption: 0	3215342

The time interval for trigger queue level aggregation should be longer (30 - 60 minutes) than Flow/User as it involves the aggregation on application states across more regions (because application state get stored in sequence of per user_id but not queue_id).

Note: all time intervals of aggregation discussed above should be configurable.

Open Questions

How do we support multiple time interval aggregation across Flow/User/Queue level aggregation? As user may need to query one hour, one day or one month aggregated states in adhoc way. May be we just store the fine-grained time interval states, and do some additional aggregations for query on the fly? Or we have daily,weekly or even monthly table that could cache results for return?

HBase coprocessor (observer model or endpoints model) need to deploy related jars on HBase server side (make sure hbase-site.xml to point to related jars). Does this involve any complexity in deploying new timeline service (need to more touch on HBase cluster)?

It worth to carefully discuss where and when to trigger HBase endpoints coprocess for Flow/User/Queue level aggregations.
Some initative thinking is to have Timeline Reader daemon to take this role because the lifecycle of app collector is very short and RM is easy to be the bottleneck for calculating and traffic for all aggregations.

SQL support on some levels of aggregation?

About aggregation strategy. similar to Ambari, we can (carefully) limit the ttl for each level of aggregation so that on each aggregation we can read a small set of historical time series data items and aggregate them into the next stage.

Do we want to support sub-app aggregations? As an example, framework like Tez reuses containers. May be we just need to model the today's aggregations so that it is easy to add a new type of aggregation?

(proposed again) Do we need aggregations against application attempts? Even it is not today's priority, but should still be in TODO list.

What if some app in a flow-run misses a metric? Tez app misses MR metrics. May be treat the missing values as 0?

Appendix I: Aggregatable Metrics List

RM:
App attempts metrics, container metrics and resource allocation metrics:

- Number of Application Attempts
- allocated/preempted Containers
- allocated/preempted Resource

Mostly comes from RMAppMetrics and RMAppAttemptMetrics

NM:
Container resource consumption metrics:

- memoryMetric (currentTime, currentPmemUsage)
- cpuMetric (currentTime, milliVcoresUsed)

AM:
General metrics, like:

- number of allocated/preempted/completed/failed containers
- number of resource request

App specific metrics/counters, like:

- number of failed/finished maps/reduces
- HDFS_BYTES_READ, HDFS_BYTES_WRITTEN