[Types of aggregation]
There are 3 types of aggregated values we want to see:

(1) current snapshot sum
(2) time series sum along with the current average and max for gauges (see below for different definitions of the "average")
(3) time-based aggregation (based on time windows like daily, weekly)

At the app level, the flow-run level, the flow level, and the user and the queue level, there is a desire to see if it is possible to have all 3 types of information. It might not be possible/applicable in some cases, however. For example, for app-level aggregation, time-based aggregation does not apply. Time-based aggregation makes sense only for higher-level aggregation (e.g. flow/user/queue).

We reviewed the draft proposal of the more complete native HBase schema design (attached to the JIRA).

[App level aggregation]
This is largely unchanged from what we agreed on previously. Framework-specific metrics will be sent to the per-app collector aggregated by the AM itself. YARN-system metrics for containers will be sent to the per-app collector by individual node managers, and will be aggregated in memory by the per-app collector.

Ultimately the per-app collector is responsible for writing aggregated metrics to the storage on its own interval.

[Flow-run table]
The flow-run table forms the core of the "real time" aggregation. Although in a way this could be technically considered an optimization (in the sense that the aggregated values can be computed from the individual time series of metrics in the apps), this is an essential part that makes it possible to serve up aggregated metric values fast.

You can envision a new RM UI of the active or the most recent flow runs based on this flow run table for example.

We discussed the details of the flow-run table, based on our schema design doc. See that document (above) for more details.

[Application table]
There was a quick consensus that we want the application table apart from the raw entities table for the performance reason, and that its schema should be almost identical to the entities table.

[Different types of metrics]

There are 2 different types of metrics (at least): a counter and a gauge. Different aggregation may be needed for each type. It was also pointed out that the metrics API needs to be updated to reflect the counter vs. gauge type. Also, the type info should be persisted in the storage.

[1] counters
The counter is quite straightforward. The counter is basically a cumulative (ever-increasing) metric, such as “HDFS bytes written” or “# of input records”. For them, it is sufficient to do a simple of sum of individual metric values.

[2] gauges
The gauge is a metric that can increase or decrease and is not cumulative in nature. Examples include “CPU usage” or “used memory in bytes”. For them, in addition to a simple sum (which is still useful), one may want to apply more statistical formulae (such as averages, min, max, etc.).

[Aggregation of gauges]
Both for counters and gauges, the simple sum is still understood as the result of the aggregation.

For gauges, we may want to consider storing more derived numbers. Although it is still possible to derive these numbers on the read path using the raw data, it would be cheaper for the per-app collector to compute them in memory and write to the storage.

[Different definitions of the “average”]
There can be different definitions of the average when it comes to a gauge-type metric because there are 2 dimensions involved here: (1) per-container, and (2) time. It would be useful to look at a concrete example to differentiate them more clearly.

Consider the following case of CPU utilization of an app (in the units of cores):

	t1	t2	t3	t4	t5	t6	t7	t8	t9	t10
c1			1	1	1	0.5				
c2	0.5	1	1	1	0.5					
c3	0.5	1	1	1	1	1	0.5	0.5		
c4		0.5	1	1	1	1	1	0.5		
c5									1	1
app	1	2.5	4	4	3.5	2.5	1.5	1	1	1

[1] time average & max

This looks at the total sum of the metric, but computes the average **over time**. Assuming the times are spaced evenly, the running average and the maximum look like the following for the above data:

	t1	t2	t3	t4	t5	t6	t7	t8	t9	t10
avg	1	1.75	2.5	2.875	3	2.92	2.71	2.5	2.33	**2.2**
max	1	2.5	4	4	4	4	4	4	4	**4**

In essence, this average is the integral of the sum graph (the area under the graph) at the given time. In other words, the average multiplied by the elapsed time of the application represents the total resource usage over time.

This would be useful in determining the total resource consumption coming from the application.

The consensus in the room was to retain this average and max for a gauge in addition to the basic aggregation. The average and the max do not need to be time series themselves; i.e. it is sufficient to keep track of the latest average and max.

This means storing a couple of additional values (average and max) for each gauge (in addition to the main time series for the metric) in the application table.

[2] per-container average
This is defined as the sum of container CPU utilization divided by the number of **active** containers at that time. Note that the number of active containers is a variable of time, and thus this may not be as trivial as dividing it by the number of total containers. The time running average of the per-container CPU utilization for the above data is as follows:

	t1	t2	t3	t4	t5	t6	t7	t8	t9	t10
avg	0.5	0.83	1	1	0.875	0.83	0.75	0.5	1	1

This might be useful in determining how **each** container might be using resources; e.g. calibrating the resource ask to the RM. But the consensus in the room was this is not as important as the first type, and thus we will not look into this at this time.

[Flow-level collector, etc.]
It was suggested that we might want to look into having a flow-level collector, not unlike the app-level collector for individual apps, and having it do the aggregation. One can even imagine a chain of collectors (app collector chained with a flow collector, a user collector, and a queue collector).

This could potentially simplify things by moving the logic of aggregation that needs to be done in the storage layer to the flow/user/queue collector. This is an alternative to the currently discussed design proposal.

The lifecycle of such flow/user/queue collectors is potentially an issue in the absence of an explicit YARN flow API. For example,

- When and where should the flow collector be started?
- How does the flow collector get terminated?
- If we use an inactivity as an indicator for terminating a flow collector, what if there is a pause between app activities? If we bring the flow collector up again, can it bootstrap itself successfully?
- Where and when should the user and queue collectors be started and stopped?

Also, discovery of these collectors would be needed and implemented.

This is probably a major feature, and we were not sure if we want to go that route without an explicit YARN flow API. Introducing a YARN flow API is a major undertaking in and of itself (requested by multiple features). Creating a dependency on a yet-to-be-started major feature didn't seem feasible at this point.

[Running time-based aggregation]
I believe the consensus at the end was to do only time-based aggregation (e.g. daily, weekly, etc.) for flows, users and queues. Correct me if I'm wrong.

The base proposal was to do it using something like a mapreduce job. Vinod pointed out that the dependency is backwards (a YARN feature depending on mapreduce). Vinod and Li mentioned there is a way to do this with co-processor endpoints. Instead of a mapreduce batch job for daily aggregations, the thinking is along the lines of having a co-processor endpoint fetch the data that is to be aggregated for the daily time interval. The standalone process that invokes the endpoint will then perform aggregation on the data fetched along the flow, user, queue tracks and write to the relevant phoenix tables. This then becomes similar to having a mapreduce job with some mappers and 1 reducer.

[Other topics]
Li pointed out that there is a "null delimiter" (?) with Phoenix and that Phoenix can query native HBase tables easily if we use null delimiters in the native HBase tables. This might help with the native HBase schema as well if it makes us avoid all the escaping/encoding business. Li will look into that some more.

Also, it was pointed out that we want to store the user id in the app-flow lookup table.

[Open/unresolved questions]

- How do we handle the averages and maxes of gauges for the flow-run aggregation?
- How would we handle framework-specific gauges? Note that the AM would perform the aggregation.
- How would we handle long-running apps for time-based aggregation? The current ideas revolve around (e.g. RM) dumping metrics for active apps at regular intervals.