### Objective

This document proposes describes a proposal to support unbounded limit in **direct runner** via SQL shell, and lists approaches to enable unbounded limit with tradeoffs.

### Background

The SQL shell users may want to run a "*SELECT col_a FROM pubsub_table LIMIT 10*" to sample data from pubsub. However, unbounded limit is not supported yet. As SQL Shell runs pipelines in global window plus default trigger, queries that are against unbounded data will never return.

### Design Detail

Here are some highlights of the design:

1. Users are allowed to run unbounded limit in direct runner. So far, attempts to run queries in non direct runner will be failed with an exception, so running unbounded limit in other runners is not a concern.
2. When an unbounded limit query runs through SQL shell, a daemon thread is spawned by BeamEnumerableConverter to monitor a shared variable, which saves a enum showing whether the converter has collected enough values to return. That shared variable is co-owned by a DoFn, the "collector", which is applied to PCollection returned by RelNode tree. So the workflow is, the "collector" DoFn collects values from ReNode tree and saves values into a buffer. If the size of that buffer is greater than the limit count, the "collector" will notify the daemon thread to terminate the pipeline by modifying that shared variable. Note that this should only work in direct runner because threads can access the same memory address(shared variable) within the same JVM. Finally, the daemon thread terminates the beam pipeline.
3. BeamSortRel has also to be modified to separate Sort and LIMIT logic. Currently Sort and LIMIT are combined together in BeamSortRel by a Top transform. However, in Beam, group by is not allowed on unbounded input data in global window. In order to make unbounded limit work, a DoFn is created to implement LIMIT only functionality in BeamSortRel.

```
0: jdbc:beam:>
SELECT pubsub_topic.payload.id, pubsub_topic.payload.name from pubsub_topic limit 1;
0: jdbc:beam:> SELECT pubsub_topic.payload.id, pubsub_topic.payload.name from pubsub_topic limit 1;
Jun 15, 2018 4:15:21 PM org.apache.beam.sdk.io.gcp.pubsub.PubsubUnboundedSource createRandomSubscription
WARNING: Created subscription projects/google.com:clouddfe/subscriptions/ruwang-test_beam_2852699076343852243
+-------------+-------+
|     id      | name  |
+-------------+-------+
| 13          | JASON |
+-------------+-------+
1 row selected (52.927 seconds)
```

(SQL shell terminates unbound limit query once enough values are collected.)

### Problems and Potential solutions

There is a problem that, dedicated logic is needed to allow BeamEnumerableConverter being aware of unbounded limit. There are three ways to achieve it:

a. Check Structure of RelNode Tree

The following are plans generated when running limit queries with and without GROUP BY:

*(With GROUP BY:)*

```
BeamSortRel(fetch=[10])
  BeamAggregationRel(group=[{0}])
    BeamProjectRel(f_int=[$0])
      BeamIOSourceRel(table=[[beam, PCOLLECTION]])
```

*(Without GROUP BY:)*

```
BeamSortRel(fetch=[10])
  BeamProjectRel(f_int=[$0])
    BeamIOSourceRel(table=[[beam, PCOLLECTION]])
```

So each time, BeamEnumerableConverter could check the structure of RelNode tree. Once it finds structures are as the same as above, the converter will execute unbounded limit code path.

Update:
Beam will optimize physical plan and change the order of ProjectRel and SortRel. Therefore, It is not enough to only check SortRel as the root. Three combinations of check are needed:

1. SortRel only
2. Project + Sort
3. Project + Aggregate + Sort

Update 2:
GROUP BY does not work on pubsub table with column type “ROW <id INTEGER,name VARCHAR>)”. Calcite parser will fail on query like “SELECT pubsub_topic.payload.id from pubsub_topic group by pubsub_topic.payload.id” without giving clearly reason. Therefore, right now **only LIMIT without grouping by can be supported.**

Update 3:
BeamProjectRel has been replaced by BeamCalRel.

**Pros:**
Users do not need extra steps to run unbounded limit queries.

**Cons:**

1. If only check if SortRel is at the root of the tree, and if users input a SELECT * query, type mismatching or type cannot casting exception will be thrown out, which will be very confusing to users.
2. If we want to throw more meaningful message, dedicated logical is required in BeamEnumerableConverter to check tree structures. BeamSortRel, BeamAggregationRel and BeamProjectRel also require modifications to support the check.

b. Pipeline Options (https://docs.google.com/document/d/1UTsSBuruJRfGnVOS9eXbQI6NauCD4WnSAPgA_Y0zjdk/edit?usp=sharing)

As SQL shell supports setting pipeline option manually, we can utilize it and allow user set an option argument to enable unbounded limit. Once BeamEnumerableConverter sees the option argument is on, it will go through unbound limit code path without checking RelNode tree structure.

**Pros:**
Avoid checking plans to reduce tree structure checking logic. Relnode implementations wouldn't require too much change.

**Cons:**
1. Users need to take extra steps to run unbound limit.
2. Once the option is on, the convert will try to run unbound limit logic without checking the tree path(aka. input query). So users will take the risk to run a non unbounded limit query with unpredictable result after setting the option argument.

c. Calcite trait indicating "known finite size"

Whenever a collection has a known finite size due to LIMIT, set this trait. Then BeamEnumerableConverter checks the trait and will cancel the pipeline when that many rows are received.

**Pros**:
Works even when the LIMIT is buried deep inside the plan.

**Cons**:
1. A much more extensive change than the other approaches.
2. Beam SQL could not support unbounded limit deeply inside the plan (e.g. nested query with a LIMIT deeply inside) because LIMIT converts unbounded data to bounded data in global window but Beam model does not have a good story about this kind of conversion.