Rabit bootstrap operations support

Chen Qin

Summary

Rabit is a lightweight interface / implementation of a *subset of MPI interfaces* featuring *peer to peer snapshot/restore* & *cluster zero restart recovery* capability.

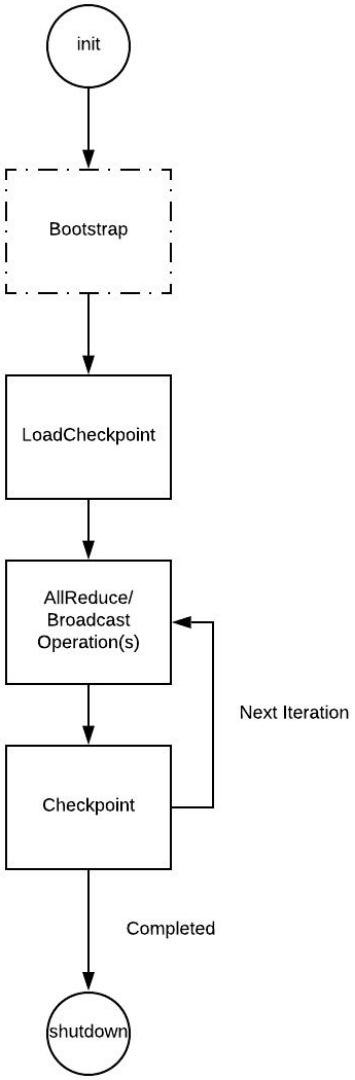
This framework has been the backbone of distributed XGB. It is widely used in both big data frameworks XGB Spark/Flink and K8S operators. As community observing various xgb worker reliability issues in distributed environment, it makes sense to document and share how Rabit fault tolerance is implemented with a bit more detail.

As we investigate various distributed XGB reliability issues in production, we identified a missing feature (bootstrap operations) yet to be supported by the current Rabit fault recovery mechanism. In the later part of this documentation, we would like to propose a design for this feature.

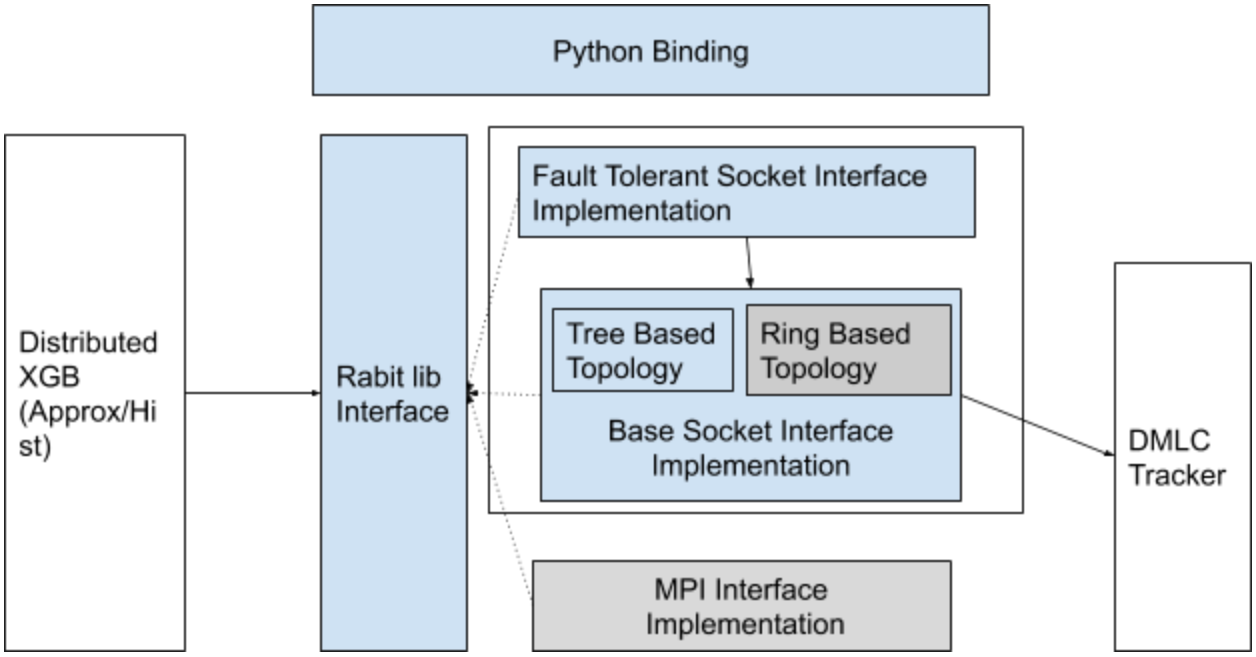
Rabit Overview

Rabit was designed and implemented as a *lightweight fault tolerant allreduce library* by Tianqi Chen, Ignacio Cano, Tianyi Zhou (origin paper link). It took a novel approach to address iterative machine learning synchronization problems with improved fault tolerant guarantee.

Following diagram shows overall state transition of a rabit program.



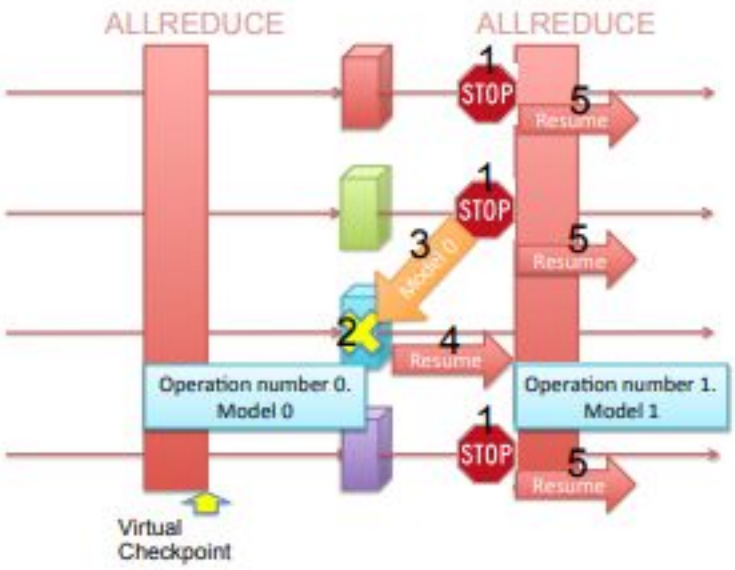
Rabit was implemented in C++ with multiple platforms support. It also provide Python/C binding for user to interact with Python code or JVM JNI code. With dmlc tracker (Python) to coordinate registration and binary tree topology setup, Rabit cluster form tree shaped network topology to facilitate collective allreduce/broadcast/loadcheckpoint/checkpoint etc operations.



Rabit is featured with simplicity in design and implement based on C++ socket. It's base socket interface implementation provides *non fault tolerant, best effort, pull based* socket implementation of *init, shutdown, allreduce, broadcast, getworld, getrank* etc MPI interface implementations.
User can run tree based topology MPI allreduce/broadcast in multiple platforms without tedious MPI environment setup.

Rabit Fault Recovery

Rabit designed novel approach towards fault tolerance. Compare to big data frameworks such as Spark/Flink, Rabit is specifically catered towards iterative collective blocking global synchronization MPI use cases. It's heavily rely on *allreduce primitive* to reach global consensus *what is next transaction* and *how to facilitate next transaction on each node.*



In case of fault recovery phase, it runs 5 steps process to recover failed node(s)

1. Pause all alive nodes at next allre duce/broadcast/checkpoint calls
2. Failed worker load last checkpoint from all peers using message passing and routing protocol after it re-register in tracker; all nodes reset links to peers in topology
3. Failed worker rerun from last checkpoint until it hits first allreduce/broadcast operation.
4. Entire cluster reach consensus and pass back previous succeed allreduce/broadcasts result for failed worker to catch up.
5. Once every node runs to same allreduce/broadcast operation sequence. Entire cluster reach consensus and call/buffer base socket allreduce/broadcast implementation.

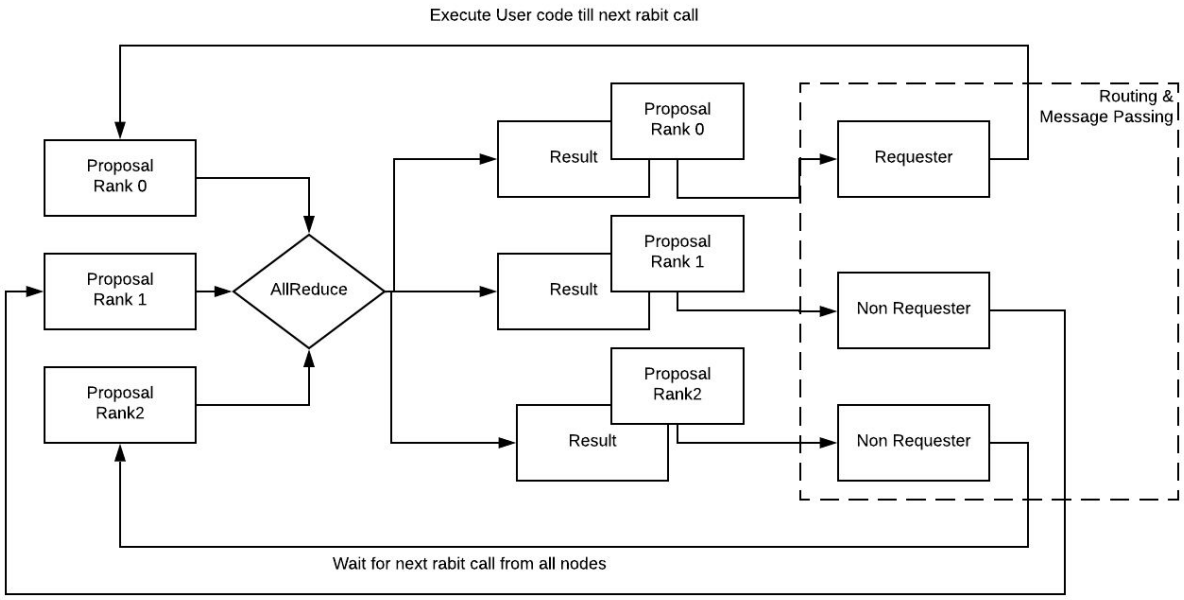
Rabit Consensus Protocol

> Global consensus is reached via running a allreduce on entire cluster using special control message called *ActionSummary* from each node.

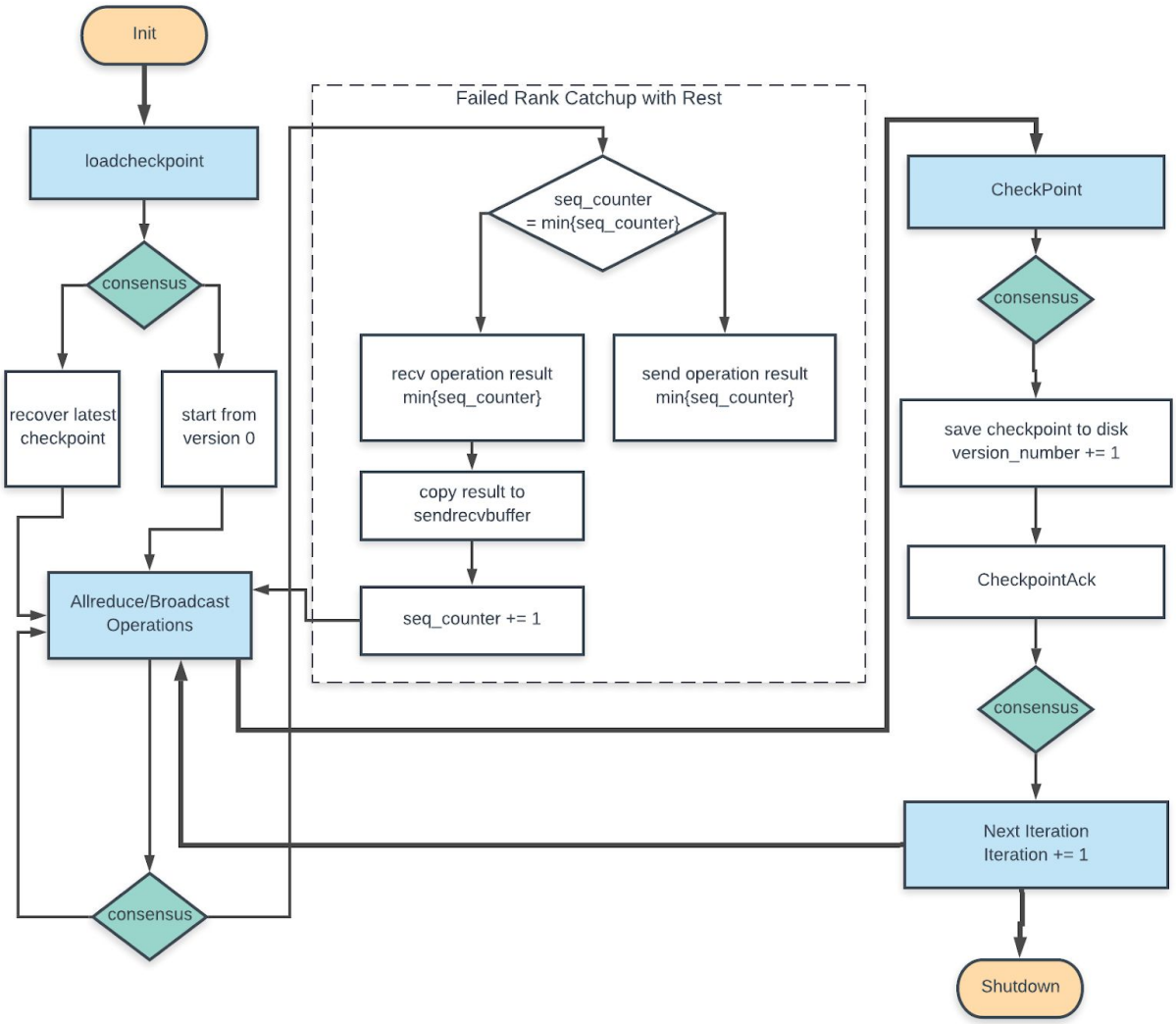
Rabit leverage allreduce primitive on control message ActionSummary to reach global consensus on what to do in the next transaction. It roughly runs follow steps until the program exits.

1. Every node has to be alive and connected, it sends proposal when it calls allreduce/broadcast/checkpoint/loadcheckpoint rabit API
2. Allreduce were executed to run reduce function on those proposals, share same ack to every node with possible different proposals

3. Based on priority, certain proposal(s) were accepted based on information on ack as well as compare ack with proposal itself on each node
4. After possible routing and message passing,
    a. node finished and returned were able to move forward and execute till making another proposal
    b. nodes with not selected proposals wait till those returned node send another batch of proposal(s) at next allreduce/broadcast/checkpoint/loadcheckpoint synchronization point
5. *Eventually*, every node runs on same phase where all proposals were accepted at the same time and executed collectively

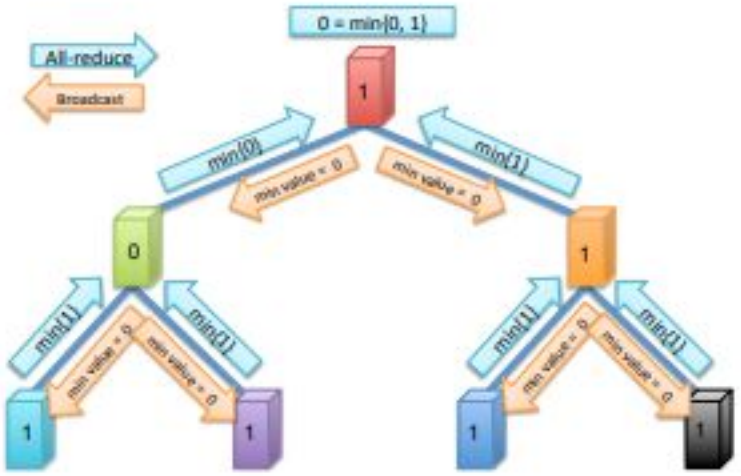


Following diagram shows a simplified version of how rabit consensus protocol helps failed node catch up with the rest in a rabit program lifecycle.



Every node has to connect with its root and children in topology. The possible node proposal payload could be one of the following types `ActionSummary req(flag, seqno);`

- checkpoint with flag set to kCheckpoint and seqno to kSpecialOp
- loadcheckpoint with flag set to kLoadCheck and seqno to kSpecialOp
- checkpoint/loadcheckpoint ack with flag set to kCheckAck and seqno to kSpecialOp
- allreduce/broadcast with seqno to current sequential allreduce/broadcast index and flags unset 0.



By the time every node propose it's next intended action and run allreduce all the way to rank 0, actionsummary reducer try to find if any node set those special flags

- kCheckpoint: checkpoint current model
- kLoadCheck: load latest checkpoint
- kCheckpointAck: either checkpoint or loadcheckpoint finished

Based on the priority of those flags, *kCheckAct > kCheckpoint > kLoadCheck > no flags (allreduce/broadcast)* nodes proposing those operations will play a leader role (a.k.a requester) and force rest fleet finish it's operation until the entire cluster finishes what requester asked. Once requester finish, it finish and move forward to next allreduce/broadcast operation.

In no flags priority, which means every node are calling allreduce or broadcast. Rabit check minimal seqno proposed by all nodes and compare if there is sequence number difference. If not, it indicates every node are progressing at the same phase and hit the same code path. Rabit will call base class to run collective allreduce/broadcast and store results in buffer (ResultBuf) increase seqno by 1; Otherwise, it tried to figure out which node is left behind (minimal seqno) and assign a leader role for that node(s) to recover results from rest of nodes. Detail here link

When proposed checkpoint succeed in a node, that node goes into critical section

- Increase version_number (aka iteration index)
- Persist checkpoint locally to disk
- reset seqno back to 0

- clear buffer (aka ResultBuf)

After it succeeds, it runs checkpoint acknowledge to unblock other operations.

Routing & Message Passing

“ Routing: Once the model version to recover is agreed among the nodes, the routing protocol executes. It finds the shortest path a failed node needs to use in order to retrieve the model. It uses two rounds of message passing. The first round computes the distance from a failed node to the nearest node that has the model. The second round sends requests through the shortest path until it reaches the node that owns the model, and retrieves it. “ - RABIT: A Reliable Allreduce and Broadcast Interface

In the first round stated above, all rabit nodes enter this routine at the same time. Depending on it’s role decided by global consensus phase, every node figure out what to do in the next data transfer phase. Those with shortest distance in tree topology will be picked to send data back to node request data recovery.

In actual data transfer(aka recover data routine) workers either write to specific link or read from specific links until specific byte sent or received.

```
 /*!
  * \brief try to finish the data recovery request,
  *       this function is used together with TryDecideRouting
  * \param role the current role of the node
  * \param sendrecvbuf_ the buffer to store the data to be sent/recived
  *         - if the role is kHaveData, this stores the data to be sent
  *         - if the role is kRequestData, this is the buffer to store the result
  *         - if the role is kPassData, this will not be used, and can be NULL
  * \param size the size of the data, obtained from TryDecideRouting
  * \param recv_link the link index to receive data, if necessary, obtained from TryDecideRouting
  * \param req_in the request of each link to send data, obtained from TryDecideRouting
  *
  * \return this function can return kSuccess/kSockError/kGetExcept, see ReturnType for details
  * \sa ReturnType, TryDecideRouting
  */
 ReturnType TryRecoverData(RecoverType role,
                           void *sendrecvbuf_,
```

```
size_t size,
int recv_link,
const std::vector<bool> &req_in);
```

Strategy of “Fail-All”

Distributed XGB can run both in XGB-Spark / Flink settings. As we speak right now, Rabit failure recovery has not been working as we expected. Checkpoint and recovery has been rely on canonical checkpoint recovery implemented in Spark layer using strategy “fail-all” and load checkpoint

1. upload learner binary to remote file system (such as HDFS)
2. shutdown all nodes if any node fail
3. download binary from HDFS, start all nodes from last succeed checkpoint

This checkpointing strategy has been widely adopted by tensorflow, dataflow etc. It’s sufficient to run on compute with reasonable reliability SLA. End user have intuitive and simple recovery model in mind. Uber internal job failures show this strategy may be vulnerable to frequent of failures from software stack or hardware in preemptive scheduling environment.

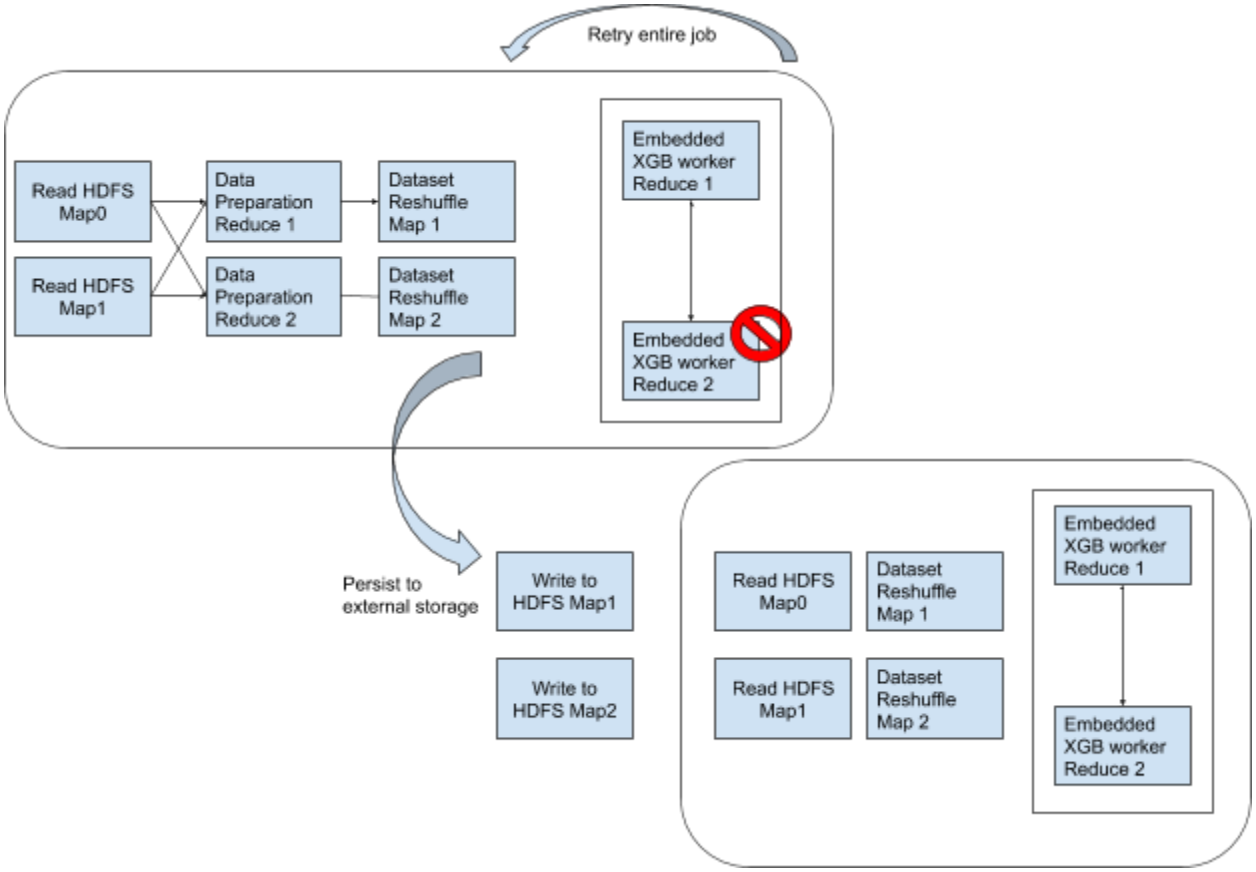
Here is an example root causing results from one of seasoned eng following a series of stability issues.

> *“For the most recent run below are the kernel log snippet which shows hardware issue on that machine.*
>
> *I will suggest to somehow build some fault tolerance in the job/framework as you can not guarantee all the machines in the cluster is healthy … broadly we need to have fault tolerance built into the job.”*

Here is an input from one of our customers we worked with on large scale training job:

> *“the reason I used 375 workers in XGBoost, wheres 400 cores in Spark was that I wanted to leave an extra 25 cores available for the case of node failures.”*

Despite of various advantages stated above, on a broader picture, production ML jobs often runs data preparation and feature generation using big data frameworks with multiple stage workflow “map-reduce” diagram (Jeff D, Sanjay G paper link). Single task retryable were one of the key requirements popular map-reduce framework employed in order to fit into end to end workflow. When we employ strategy of fail all, single task failure will force all reducer/mapper on application fail can cause massive data reshuffle network IO operation.

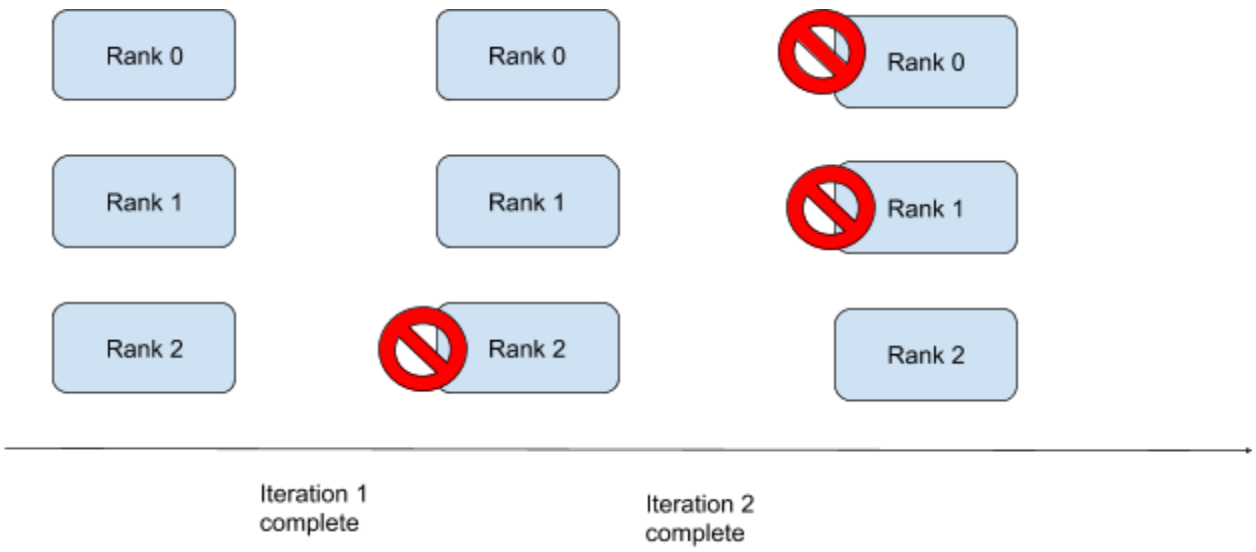


We have seen large production jobs manually split workflow into two jobs, adding external storage output after dataset reshuffle to avoid those overheads. The problem of this approach is it adds extra work to manage those mini jobs as well as adding external storage data IO overhead.
In the next section, we plan to explain a different approach that allows embedded xgb workers run natively on map-reduce frameworks.

Strategy of “Retry-One”

“Retry-one” strategy benefits large scale cluster with compromised SLA.(a.k.a GCP preemptive instances) Compute with compromised SLA often face much higher frequency of series of interruptions and failures.

Here is an example of training job facing series of interruptions. The cause of such interruptions could be a software bug, network issue or preemptive notification from scheduler. Rank 2 suffers interruption and restarts after 1st iteration complete, Rank 0,1 suffer interruption and restart after 2nd iteration complete.



Note: *We assume that the failed node was not significant compared to the entire cluster size and that the failed node's training data was not cached in the failed node*. Basically if the task failed and restarted and the original input data has gone, it still needs to redo the upstream stage unless we use an external shuffle service. (And even if we use an external shuffle service, we may not be able to avoid re-doing the upstream: the node saving shuffle pieces may be preempted and the shuffle data will also be gone.) It also assume distributed XGB can request compute resources after training job started. It's not supported by Spark which may require overprovision during job starts at the moment.

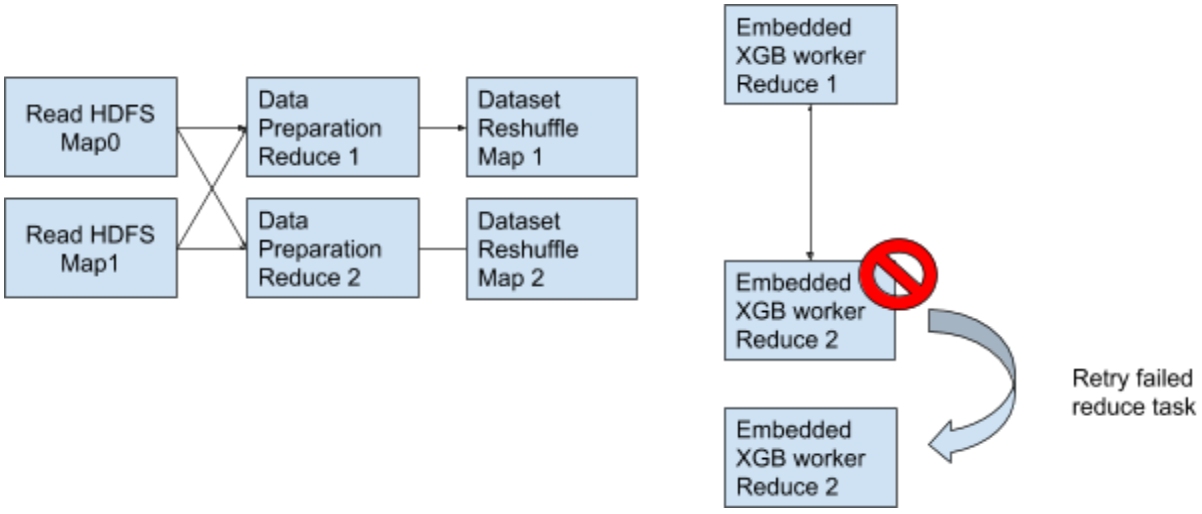
With the assumption above, compare to "fail-all" strategy,

- "Retry-one" strategy does less frequent resource (compute) requests (and wait) to cloud providers
- "Retry-one" strategy reduces data reshuffle overhead of getting partitioned training dataset to cluster

As cluster grows larger and training dataset getting bigger, "retry-one" gives a better result on running on compromised SLA cluster.

	Total container resource requests	data reshuffle overhead
"fail-all"	3+3+3(total partitions)	3 (total partitions) x 3 (job execution times)
"retry-one"	1 + 2 (failed partitions)	3 + 1+ 2 (failed partitions)

Compare to “fall-all” strategy, retry one stregety can get better support from map-reduce frameworks. Failed task only needs to pull it’s data from cached output hence reduce data shuffle pressure. Notice we can also cache it to external storage which can help restart job only deal with training part as the diagram shows.



“Retry-One” --bootstrap operations issue

In distributed XGB, we observed issues from stuck nodes to node fail to recover due to living XGB/Rabit worker hang and/or failed restart nodes showing allreduce size assertion failure. It took us a bit of time to root cause those issues. Along the way, we identified common pattern of such failure pointing to similar bootstrap operations Rabit is yet to support.

> Bootstrap allreduce(s)/broadcast(s) is defined as out of order operation(s) executed only once before node first call checkpoint.

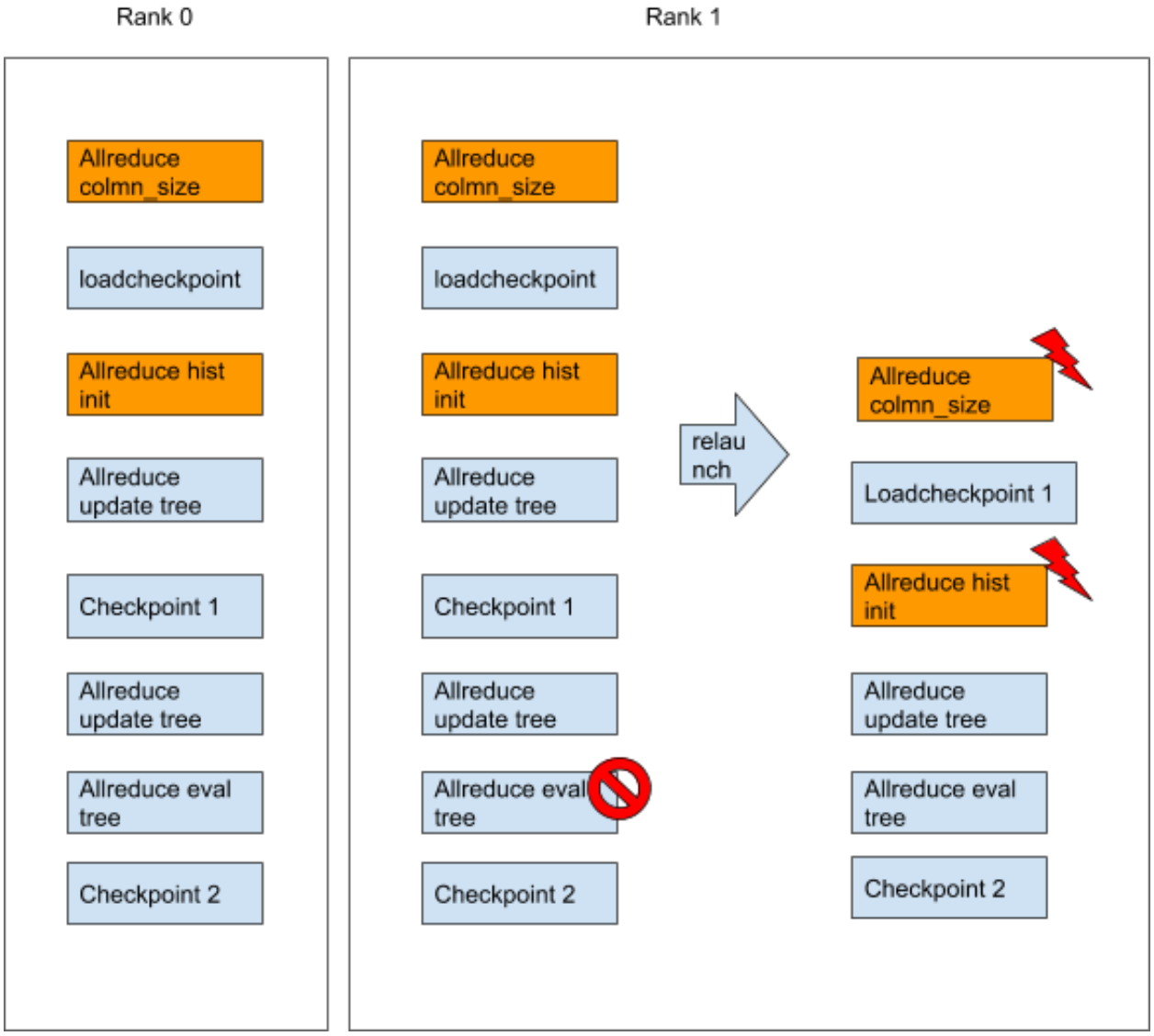
The example of such bootstrap allreduces range from

- Allreduce training dataset column size, dataset column size testing dataset
- Broadcast column sampler seed value before first iteration in fast hist three method
- Allreduce in Histogram init method within fast hist three implementation

Bootstrap allreduces may runs *out of order* before first checkpoint call. Once cluster pass first iteration of checkpoint, those allreduce results were removed from every node. When a worker recover and rerun those allreduces, rest of nodes will not be able to recover those data.

Following is an example of bootstrap allreduces, rank 0 is healthy while rank 1 fail and recover. The yellow rectangles are bootstrap allreduce operations run once before first checkpoint call.

- Rank 1 failed while running eval tree allreduce before checkpoint 2
- Rank 1 relaunch and hit bootstrap allreduce asking for column size of training dataset
- Rank 0 tried to recover with allreduce update tree binary based on seqno (failed payload size check)
- Assume rank 1 can move forward and call bootstrap allreduce hist init, rank 0 use update tree payload and try to recovery rank1(failed payload size check again hopefully)
- Assume rank 1 can move forward to allreduce update tree, rank 0 would be able to align seqno and backfill data correctly.



These bootstrap allreduces are *unrecoverable* as well as *out of ordered* .

- Unrecoverable: Rabit needs to have extra way to storage those results outside of iteration based cache
- Out of order: Rabit required to develop kind of method call signature to random access those cache outside of sequential based lookup

Bootstrap Cache Based Solution

The rationale of this approach is to be consistent on how rabit do fault recovery with seqno matching. By keeping those bootstrap allreduces in another buffer, rabit backfill bootstrap allreduces if needed.

Rabit API Change

We propose to introduce a is_bootstrap flag in both allreduce and broadcast method. If user set it to true, those allreduce and broadcast results will be cached in entire job lifecycle. Failed node would be able to recover via those cache results.

*Allreduce(void *sendrecvbuf_,* *size_t type_nbytes,* *size_t count,* *ReduceFunction reducer,* *PreprocFunction prepare_fun = NULL,* *void *prepare_arg = NULL,* ***bool is_bootstrap = false,*** *const char* _file = _FILE,* *const int _line = _LINE,* *const char* _caller = _CALLER* *) = 0;*	virtual void Broadcast(void *sendrecvbuf_, size_t size, int root, ***bool is_bootstrap = false,*** const char* _file = _FILE, const int _line = _LINE, const char* _caller = _CALLER) = 0;

Due to the fact failed node recovery may run allreduce / broadcast in different sequence. Current rabit sequence number matching based recovery may not work properly. In last three parameters of new API, we tried to capture caller signature and generate unique key to lookup for bootstrap operations.

```
// keeps rabit api caller signature
#ifndef RABIT_API_CALLER_SIGNATURE
#define RABIT_API_CALLER_SIGNATURE
#ifdef __linux__
#define _FILE  __builtin_FILE()
#define _LINE  __builtin_LINE()
#define _CALLER  __builtin_FUNCTION()
#else
#define _FILE  "N/A"
#define _LINE  -1
#define _CALLER  "N/A"
#endif  // __linux__
#endif  // RABIT_API_CALLER_SIGNATURE
```

Following is an example of how rabit_api_caller_signatures looks like running one iteration of distributed XGB. Not all calls are cached, as an example to show signature format. We plan to upstream this logging changes as well.

```
[0] allreduce (src/data/data.cc::229::Load#8x1) finished version 0, seq 0, take 0.000856 seconds
[0] allreduce (src/learner.cc::734::LazyInitModel#4x1) finished version 0, seq 0, take 0.000394 seconds
[0] allreduce (src/tree/./updater_basemaker-inl.h::63::SyncInfo#4x254) finished version 0, seq 1, take 0.000464 seconds
[0] broadcast (src/tree/./updater_basemaker-inl.h::98::SampleCol#8@0) root 0 finished version 0, seq 2, take 0.000251 seconds
[0] broadcast (src/tree/./updater_basemaker-inl.h::98::SampleCol#476@0) root 0 finished version 0, seq 3, take 0.041262 seconds
[0] allreduce (src/tree/updater_histmaker.cc::727::CreateHist#16x118) finished version 0, seq 4, take 0.044093 seconds
[0] allreduce (src/tree/updater_histmaker.cc::727::CreateHist#16x236) finished version 0, seq 5, take 0.000722 seconds
[0] allreduce (src/tree/updater_histmaker.cc::727::CreateHist#16x472) finished version 0, seq 6, take 0.000805 seconds
[0] allreduce (src/tree/updater_histmaker.cc::727::CreateHist#16x708) finished version 0, seq 7, take 0.000755 seconds
[0] allreduce (src/tree/updater_histmaker.cc::727::CreateHist#16x708) finished version 0, seq 8, take 0.000720 seconds
[0] broadcast (src/tree/updater_sync.cc::38::Update#8@0) root 0 finished version 0, seq 9, take 0.000239 seconds
[0] broadcast (src/tree/updater_sync.cc::38::Update#976@0) root 0 finished version 0, seq 10, take 0.043363 seconds
[0] checkpoint size 1376 finished version 1, seq 11, take 0.043974 seconds
[0] checkpoint ack finished version 1, take 0.000413 seconds
[0] allreduce (src/metric/elementwise_metric.cu::338::Eval#8x2) finished version 1, seq 0, take 0.000561 seconds
[0] checkpoint size 1376 finished version 2, seq 1, take 0.000051 seconds
```

Bootstrap operations are cached in another ResultBuffer similar as how operations are stored. On another side, rabit_api caller_signatures were also stored to match seqno in cache.

SetCache

When user set is_bootstrap to true, at the end of allreduce operation, setcache will be called to insert bootstrap operation result into local cache instead of current Rabit seq buffer.
Similar to Rabit fault recovery design, setcache path doesn't require nodes communicate with each other.

```
int SetCache(const std::string &key, const void *buf,
  const size_t type_nbytes, const size_t count);
```

When user call setcache with same key on same node twice, Rabit will throw assertion failure. This is because our bootstrap cache implementation assumes those cached calls only runs once. It can help user *avoid abuse bootstrap* cache in normal allreduce/broadcast operations.

GetCache

When user set is_bootstrap to true, before calling current allreduce/bootstrap routines. Rabit will try to fetch cache and see if it can recover from cache, otherwise, it will fallback to the original routine.

```
int GetCache(const std::string &key, void *buf, const size_t type_nbytes,
  const size_t count, const bool byref = false);
```

Calling GetCache() will run one more consensus allreduce operation using ActionSummary. In order to avoid multiple synchronizations we introduced another 32bit integer as well another locad_cache flag into ActionSummary.

```
static const int kLoadCache = 16;
```

In order to avoid int overflow caused by introducing additional bit offset, we plan to change actionsummary integer from signed int32 to unsigned int32.

```
explicit ActionSummary(int action_flag, int role_diff_flag = 0,
    u_int32_t minseqno = kSpecialOp, u_int32_t maxseqno = kSpecialOp) {
    seqcode = (minseqno << 5) | action_flag;
    maxseqcode = (maxseqno << 5) | role_diff_flag;
  }
```

In addition, to optimize GetCache overhead, we implemented different reducer behavior on new 32bit integer.

- seqno() on **new integer** will return max of all seqno()
- load_cache() on new integer will return if all nodes set load_cache true
- diff_seq() on new integer will return 1 if any nodes pair have different cache entry size

Max seqno() can be used by getting largest of current cache entry seqno in entire cluster, recovered node could overwrite it's own cache entries once it has less cache entreis.

Load cache flag run bitOr operation to all nodes, if all nodes set load_cache to true, then it's return true to every node. This indicates all nodes are trying to load_cache because it's fresh start and hitting getcache code. Recovery will be skipped and every node move forward; Otherwise, it returns which indicate not everyone are requesting cache, it will try to use max seq no from ActionSummary results and run cache restore routine.

Once every node has the same cache seqno, kdiffSeq on new integer will return 0 hence no need to recover cache.

Those optimizations can help avoid additional round trip and cache copy in allreduce consensus, adding new int32 can also help isolate the impact of possible code bug from hitting Rabit recovery code.

KLoadCache in ActionSummary

As mentioned above calling new API run the same global consensus process, except a new flag load_cache is introduced in ActionSummary.

New priority of flags are *kCheckAct* > *kCheckpoint* > *kLoadCheck* > *kLoadCache* > *no flags*

When we check allreduce consensus result. At load cache check side, we put assertions to avoid load_cache runs into unexpected state.

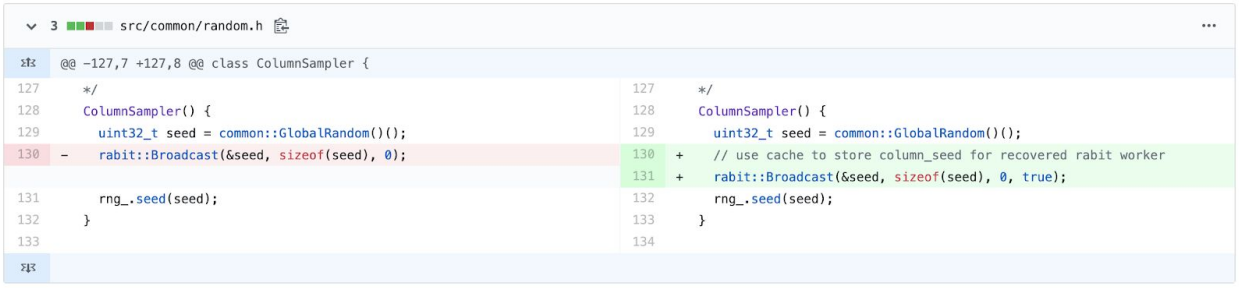
```
// run all nodes in an isolated cache restore logic
if (act.load_cache()) {
 // load cache should not running in parallel with other states
 utils::Assert(!act.load_check(),
  "load cache state expect no nodes doing load checkpoint");
 utils::Assert(!act.check_point() ,
  "load cache state expect no nodes doing checkpoint");
 utils::Assert(!act.check_ack(),
  "load cache state expect no nodes doing checkpoint ack");

 // if all nodes are requester in load cache, skip
 if (act.load_cache(SeqType::KAND)) return false;
 // if restore cache failed, retry from what's left
 if (TryRestoreCache(req.load_cache(), act.seqno(), act.seqno(SeqType::KAND))
  != kSuccess) continue;
 // if requested load cache, then mission complete
 if (req.load_cache()) return true;
 continue;
}
```

Proposed XGB Changes

```
156 -
157     Init(&sketchs, max_num_bins);
158     monitor_.Stop("Init");
159   }
@@ -174,7 +173,7 @@ void HistCutMatrix::Init
174     }
175     CHECK_EQ(summary_array.size(), in_sketchs->size());
176     size_t nbytes = WXQSketch::SummaryContainer::CalcMemCost(max_num_bins * kFactor);
177 -   sreducer.Allreduce(dmlc::BeginPtr(summary_array), nbytes, summary_array.size());
178     this->min_val.resize(sketchs.size());
179     row_ptr.push_back(0);
180     for (size_t fid = 0; fid < summary_array.size(); ++fid) {
```

```
156     Init(&sketchs, max_num_bins);
157     monitor_.Stop("Init");
158   }
173     }
174     CHECK_EQ(summary_array.size(), in_sketchs->size());
175     size_t nbytes = WXQSketch::SummaryContainer::CalcMemCost(max_num_bins * kFactor);
176 +   sreducer.Allreduce(dmlc::BeginPtr(summary_array), nbytes, summary_array.size(), NULL, NULL, true);
177     this->min_val.resize(sketchs.size());
178     row_ptr.push_back(0);
179     for (size_t fid = 0; fid < summary_array.size(); ++fid) {
```



3 src/common/random.h

```
@@ -127,7 +127,8 @@ class ColumnSampler {
127     */
128     ColumnSampler() {
129       uint32_t seed = common::GlobalRandom()();
130 -     rabit::Broadcast(&seed, sizeof(seed), 0);
131       rng_.seed(seed);
132     }
133
```

```
127     */
128     ColumnSampler() {
129       uint32_t seed = common::GlobalRandom()();
130 +     // use cache to store column_seed for recovered rabit worker
131 +     rabit::Broadcast(&seed, sizeof(seed), 0, true);
132       rng_.seed(seed);
133     }
134
```

6 src/data/data.cc

```
@@ -223,10 +223,12 @@ DMatrix* DMatrix::Load(const std::string& uri,
223       LOG(CONSOLE) << dmat->Info().num_row_ << 'x' << dmat->Info().num_col_ << " matrix with "
224                    << dmat->Info().num_nonzero_ << " entries loaded from " << uri;
225     }
226     /* sync up number of features after matrix loaded.
227 -    * partitioned data will fail the train/val validation check
228      * since partitioned data not knowing the real number of features. */
229 -   rabit::Allreduce<rabit::op::Max>(&dmat->Info().num_col_, 1);
```

```
223       LOG(CONSOLE) << dmat->Info().num_row_ << 'x' << dmat->Info().num_col_ << " matrix with "
224                    << dmat->Info().num_nonzero_ << " entries loaded from " << uri;
225     }
226 +
227     /* sync up number of features after matrix loaded.
228 +    * partitioned data will fail the train/val validation check
229      * since partitioned data not knowing the real number of features. */
230 +   rabit::Allreduce<rabit::op::Max>(&dmat->Info().num_col_, 1, NULL, NULL, true, fname.c_str());
231 +
```

Verification

Integration Tests

Sample code create a 10 node cluster and simulate node 0,4,9 fail at the same time and node 1 failed immediately after those 3 nodes failed. As we can see all four nodes recovered via retry and finished training and evaluation without restart job.
test can be found at https://github.com/chenqin/xgboost/blob/test/tests/cli/runxgbtests.sh

Run *git clone and build.*sh on *Linux* system then runxgbtest.sh
https://github.com/chenqin/xgboost/commit/0bb492ccb148a5fd2bbdbd2e64e580ad3e27e577

```
[18:34:03] 161x127 matrix with 3542 entries loaded from ../../demo/data/agaricus.txt.test
[18:34:03] 162x127 matrix with 3564 entries loaded from ../../demo/data/agaricus.txt.test
[18:34:03] 161x127 matrix with 3542 entries loaded from ../../demo/data/agaricus.txt.test
2019-07-22 18:34:04,136 INFO [18:34:04] [0]     test-rmse:0.003235
2019-07-22 18:34:04,231 INFO [18:34:04] [1]     test-rmse:0.007249
2019-07-22 18:34:04,328 INFO [18:34:04] [2]     test-rmse:0.007253
2019-07-22 18:34:04,428 INFO [18:34:04] [3]     test-rmse:0.007253
2019-07-22 18:34:04,524 INFO [18:34:04] [4]     test-rmse:0.007253
[9]@@@Hit Mock Error:CheckPoint
[4]@@@Hit Mock Error:CheckPoint
[0]@@@Hit Mock Error:CheckPoint
2019-07-22 18:34:04,624 INFO [18:34:04] [5]     test-rmse:0.007253
[1]@@@Hit Mock Error:CheckPoint
[18:34:05] start cq-MS-7B84:4
[18:34:05] Load part of data 4 of 10 parts
[18:34:05] 651x127 matrix with 14322 entries loaded from ../../demo/data/agaricus.txt.train
[18:34:05] start cq-MS-7B84:9
[18:34:05] Load part of data 9 of 10 parts
[18:34:05] 650x127 matrix with 14300 entries loaded from ../../demo/data/agaricus.txt.train
[18:34:05] start cq-MS-7B84:0
[18:34:05] Load part of data 0 of 10 parts
[18:34:05] start cq-MS-7B84:1
[18:34:05] Load part of data 1 of 10 parts
[18:34:05] 650x127 matrix with 14300 entries loaded from ../../demo/data/agaricus.txt.train
[18:34:05] 652x127 matrix with 14344 entries loaded from ../../demo/data/agaricus.txt.train
[18:34:05] Load part of data 4 of 10 parts
[18:34:05] Load part of data 9 of 10 parts
[18:34:05] 161x127 matrix with 3542 entries loaded from ../../demo/data/agaricus.txt.test
[18:34:05] 160x127 matrix with 3520 entries loaded from ../../demo/data/agaricus.txt.test
[18:34:05] Load part of data 1 of 10 parts
[18:34:05] Load part of data 0 of 10 parts
[18:34:05] 161x127 matrix with 3542 entries loaded from ../../demo/data/agaricus.txt.test
[18:34:05] 161x127 matrix with 3542 entries loaded from ../../demo/data/agaricus.txt.test
[18:34:05] Tree method is specified to be 'hist' for distributed training.
[18:34:05] Tree method is specified to be 'hist' for distributed training.
[18:34:05] Tree method is specified to be 'hist' for distributed training.
[18:34:05] Tree method is specified to be 'hist' for distributed training.
2019-07-22 18:34:05,972 INFO [18:34:05] [6]     test-rmse:0.006189
2019-07-22 18:34:06,076 INFO [18:34:06] [7]     test-rmse:0.005330
2019-07-22 18:34:06,172 INFO [18:34:06] [8]     test-rmse:0.004574
2019-07-22 18:34:06,272 INFO [18:34:06] [9]     test-rmse:0.003929
2019-07-22 18:34:06,376 INFO [18:34:06] [10]    test-rmse:0.003416
2019-07-22 18:34:06,384 INFO @tracker All nodes finishes job
2019-07-22 18:34:06,385 INFO @tracker 2.52322793007 secs between node start and job finish
```

Scalability Tests

In order to measure benchmark it's scalability and reliability, we scheduled 200 preemptive peloton compute instances and run non sampled *12B* rows of ETA trip data at *max_depth of 18*.

"spark.executor.memoryOverhead": 0.9,
"spark.blacklist.stage.maxFailedTasksPerExecutor": 200,
"spark.task.maxFailures": 200,
"spark.driver.memoryOverhead": 0.65,
"spark.local.dir": "/mnt/01/spark_shuffle",

"spark.shuffle.service.enabled": "true"

The training job use around 400 vCPU and 13.6TB of memory with rabit checkpoint deal with failure recovery.(spark checkpoint disabled) As diagram shows at around 20:30 and 23:00 there are two failures (likely due to over subscribed queue resource get preempted) get self recovered by rabit within 3-5 mins.

Unhandled Failure Cases

Rabit can't handle all failure cases. In case of the following scenarios, Rabit needs to fallback and leverage existing checkpoint mechanism:

- Tracker / Master Failure
- Majority of nodes down at the same time

- More failures per task than `spark.task.maxFailures = 4(default)` `spark.stage.maxConsecutiveAttempts` if a xgboost node keeps failing despite retry (spark) for more than four times, entire spark job will fail.

Opt In Roll Out & Backward Compatibility

Updated Rabit API by default disabled this feature. Rabit user have to manually opt in, in XGB side, we plan to propose a configuration for user to opt-in with setting set to true.

In Rabit side, we introduce a flag gate *rabit_cache* set to 0 by default. User has to opt in by turning flag in XGB config file to *rabit_cache=1* to use bootstrap cache.

At the same time, we are working on our own shadow pipelines to test stability and prediction accuracy with/without this feature open. XGB-SPARK with airline dataset using EMR is one option we can consider from community testing aspect.

Call for Collaboration

Formal Verification / Tracker Hardening / Improvements

GPU NCCL Support

Rabit retryable fault recovery runs as an overlay on top of NCCL.