Item 58
git_comments:

1. * * This is a utility class that sorts cores in such a way that * it does not cause other nodes to wait for some replicas to be up * This checks with other nodes to see what they are waiting for
2. If all else is same. prioritize fewer replicas I have because that will complete the quorum for shard faster
3. * Licensed to the Apache Software Foundation (ASF) under one or more * contributor license agreements. See the NOTICE file distributed with * this work for additional information regarding copyright ownership. * The ASF licenses this file to You under the Apache License, Version 2.0 * (the "License"); you may not use this file except in compliance with * the License. You may obtain a copy of the License at * * http://www.apache.org/licenses/LICENSE-2.0 * * Unless required by applicable law or agreed to in writing, software * distributed under the License is distributed on an "AS IS" BASIS, * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. * See the License for the specific language governing permissions and * limitations under the License.
4. Prioritize replicas where most no:of other nodes are waiting for
5. means nobody else is waiting for this , so no need to priotitize
6. * Licensed to the Apache Software Foundation (ASF) under one or more * contributor license agreements. See the NOTICE file distributed with * this work for additional information regarding copyright ownership. * The ASF licenses this file to You under the Apache License, Version 2.0 * (the "License"); you may not use this file except in compliance with * the License. You may obtain a copy of the License at * * http://www.apache.org/licenses/LICENSE-2.0 * * Unless required by applicable law or agreed to in writing, software * distributed under the License is distributed on an "AS IS" BASIS, * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. * See the License for the specific language governing permissions and * limitations under the License.
7. replicaname vs. nodename
8. DN LIV MY
9. 70% chance that the node is up;
10. make a copy

git_commits:

1. **summary:** SOLR-7280: Load cores in sorted order & limit threads to improve cluster stability
   message: SOLR-7280: Load cores in sorted order & limit threads to improve cluster stability

github_issues:

github_issues_comments:

github_pulls:

github_pulls_comments:

github_pulls_reviews:

jira_issues:

1. **summary:** Load cores in sorted order and tweak coreLoadThread counts to improve cluster stability on restarts
   description: In SOLR-7191, Damien mentioned that by loading solr cores in a sorted order and tweaking some of the coreLoadThread counts, he was able to improve the stability of a cluster with thousands of collections. We should explore some of these changes and fold them into Solr.

jira_issues_comments:

1. moved patch over from SOLR-7191
2. [~erickerickson] planning to commit this soon
3. Great!
4. bq. // This assumes replicaCount is less than coreLoadThreadCount? Why is that a question? If you don't have enough threads to load all the cores in a shard, you have to wait for the leader vote timeout which is

long. That doesn't seem look good default behavior. How does this address that? I'm only mildly caught up on the related issue.

5. Thanks [~markrmiller@gmail.com] for looking into this. The {{coreLoadThreads}} is something that's ignored in cloud mode today. Should the user not have a way to limit the no:of threads used to load cores in cloud mode. If there are thousands of cores, the cluster ends up unstable because of this. Normally, users will have a few cores per node and it is no big deal to keep the thread pool to be unbounded. The power users must have a way to tune this.
6. **body:** The symptom is that OOM errors "unable to create native thread" happens, resulting in replicas that never come up, sometimes never ending recovery cycles etc. etc. etc. Decreasing Xss or increasing Xmx doesn't help (maybe some other settings?). In my testing with 400 replicas in a JVM, something on the order of 1K temporary threads were spun up when I tried to start the JVM. More correctly that many threads were running, the rest (number unknown) never started at all. What the default should be is certainly debatable. The curious thing was that at no time did the JVM memory appear to be stressed (using jconsole to spot-check, not rigorous at all).... I've assumed that by ordering the replicas to come up based on what collection they belong to, they won't get stuck waiting for a leader election just because the ordering on instance1 happened to try to bring up collection1 then collection2 whereas instance2 tried to bring them up in reverse order. More like skip-lists in terms of waiting... Sure, with weird enough topology instance1 could have a long queue to get through before getting to collectionX whereas instance N could start with collectionX and have to go through leader vote wait timeouts, but that's way better than having a cluster that won't start at all. And when I tested starting 3 of my 4 JVMs, it was indeed painfully slow waiting for leader vote wait timeouts. But the cluster came up. Using 3 coreLoadThreads and starting all my JVMs at once took just a few minutes. And the painfully trappy behavior without this patch is that I can create all my collections just fine, but then I can't restart the cluster successfully.
   label: code-design
7. **body:** bq. Sure, with weird enough topology I don't think it takes a weird topology - just more replicas than thread to load them in a shard. The current behavior is trappy if you want to try and load tons and tons of cores. The change in behavior is trappy in general. I think we always anticipated a better strategy to deal with this, but I don't think this does it very well.
   label: code-design
8. One strategy might be to only require a certain number of replicas participate in an election to elect a leader and then tie that to the number of threads used to load replicas and then start replicas in some intelligent, per shard manner.
9. **body:** bq: I don't think it takes a weird topology - just more replicas than thread to load them in a shard. OK, I think I see what you're saying. You're talking about a "deep" topology, i.e. one with many replicas on a particular shard on a particular instance and I was looking at a "wide" topology, many collections per instance but each shard had only a few replicas. I've seen both in the field as I'm sure you have.... How much of both situations would be handled by creating an ordered list of all replicas that were leaders and loading those first then loading an ordered list of all replicas that weren't labeled as leader? There's still the case of a zillion leaders on a single instance, so some heuristic like you suggest seems to be in order. I'll emphasize though that the current code (without this patch) can prevent a cluster from coming up at _all_. With this patch the cluster at least comes up, albeit slowly if the leaderVoteWait comes into play. Bumping the number of threads to > the max replicas for a shard can handle the case you mentioned while keeping it "reasonable" can deal with the one I'm seeing. That said, I think the default should be quite high in the cloud case so we don't change the current behavior and let situations like I'm seeing deal with configuring this. I think it defaults to 8 currently, perhaps 100 (or unlimited) instead in cloud mode? How much of all of the above makes this patch "good enough for now" with perhaps follow-ons on more sophisticated approaches?
   label: code-design
10. bq. I'll emphasize though that the current code (without this patch) can prevent a cluster from coming up at all. The current code is a fix to a bug that existed when we did something like this before :) I don't want to reintroduce the bug. We didn't, and as far as I know still don't, support tons of cores very well. As we move to do that, we don't want to reintroduce bugs though.
11. bq. That said, I think the default should be quite high in the cloud case so we don't change the current behavior and let situations like I'm seeing deal with configuring this. I think it defaults to 8 currently, perhaps 100 (or unlimited) instead in cloud mode? I think it's trappy, whatever the default is. Unless we don't let users create shards with that many replicas or something.
12. bq. Unless we don't let users create shards with that many replicas or something. Although even that is probably not enough unless we only load one shard at a time.

13. Had a chat with [~shalin] and came up with the following design. h4. Objectives * Move away from the current design of infinite number of threads for core loads which leads to OOM or other issues * Avoid the leaderVoitWait problem which leads to shards with no leader for a long time or even (down shards) Blindly sorting cores based on replica names is not foolproof. It can lead to deadlocks depending on how the replicas are distributed. The sorting logic could be as follows. h5. Core Sorting logic When a node comes up, it reads the list of live nodes and the states of each collection it hosts. Construct a List of shards {{collectionName+shardName}} it hosts sorted by the (no:of replicas for that shard in other started nodes + no:of replicas present in the current node for that replica) . Break the tie by sorting the name in alphabetic {{collectionName+shardName}} order. This ensures that no other node is waiting for some replica in this node to be up. h5. Thread count The default no:of {{coreLoadThreads}} should be much higher for SolrCloud (Maybe 50 ?). The user should be able to override the value by explicitly configuring it.
14. Or, ensure that the coreLoadThreads is >= max(collection's replicas on a single node) ?
15. A patch with tests implementing the new design (with some change). [~shalin] please take a look
16. The sorting of shards to be loaded in the following order # The shards with least no:of replicas in *down* nodes and there is at least one live node waiting for replicas of this shard. If these nodes are in *down* nodes, it is no use bringing up a replica because, until those *down* nodes come up, that shard cannot be up # The shards with max no:of replica in *live* nodes. Because more replicas in other nodes are waiting for this replica to be up # least no:of replicas in my node. This helps us finish a shard by starting the least no:of cores # finally, use the replica name to break the tie .this helps in having a predictable order across nodes The thread count is set to 24 as the default in cloud. However, users can override that
17. there was a bug in the previous patch
18. **body:** What was the motivation behind changing the sorting logic? Can you please explain what use-cases does the new logic cover better? bq. The shards with least no:of replicas in down nodes and there is at least one live node waiting for replicas of this shard. If these nodes are in down nodes, it is no use bringing up a replica because, until those down nodes come up, that shard cannot be up This is flawed because with these rules, a shard which has exactly 1 replica in total and that too on the current node will always be chosen last.
    label: code-design
19. Hmm, actually the logic we had discussed earlier also fails on this corner case. Let me think more on this.
20. Shalin, that is the expected behavior. If there is only one replica for a shard and that replica is in this node, then nobody else is waiting for that replica to come up.that means nobody else will wait time out because of that replica.
21. Ah, right, I missed that. I'm reviewing the rest of the patch.
22. Commit 6c1b75b06bf2fe53be776923097e54b8c560826d in lucene-solr's branch refs/heads/master from [~noble.paul] [https://git-wip-us.apache.org/repos/asf?p=lucene-solr.git;h=6c1b75b] SOLR-7280: In cloud-mode sort the cores smartly before loading & limit threads to improve cluster stability
23. Commit 74633594d891d3f6eff61ad39310f6410dbfc313 in lucene-solr's branch refs/heads/master from [~noble.paul] [https://git-wip-us.apache.org/repos/asf?p=lucene-solr.git;h=7463359] SOLR-7280: precommit errors
24. Commit 6902eddd8df66b6120730eccf179797b6a585848 in lucene-solr's branch refs/heads/branch_6x from [~noble.paul] [https://git-wip-us.apache.org/repos/asf?p=lucene-solr.git;h=6902edd] SOLR-7280: In cloud-mode sort the cores smartly before loading & limit threads to improve cluster stability
25. Commit 5932f52588a2773b2fb87feee8a3501eb8c5cb5c in lucene-solr's branch refs/heads/branch_6x from [~noble.paul] [https://git-wip-us.apache.org/repos/asf?p=lucene-solr.git;h=5932f52] SOLR-7280: In cloud-mode sort the cores smartly before loading & limit threads to improve cluster stability
26. **body:** minor nit: when compiling I see {{\[javac] /home/mdrob/workspace/lucene-solr/solr/core/src/java/org/apache/solr/core/CoreSorter.java:24: warning: documentation comment not expected here}}. The javadoc should be after the package declaration.
    label: code-design
27. **body:** Work in progress patch for back-porting to 5x, just re-doing the lambdas. WIP because I just noticed that the new test class _also_ has lambdas that have not been changed in this patch. Even so, I could still compile the server and run my test suite on it. Results: This is _vastly_ better than the old 5x code, I don't get OOM errors and the like. Occasionally I'll get 1 or two replicas (out of 1,600) that stay in the "down" state, but do come back up when I restart the Solr instance hosting them. At Ishan's suggestion I applied my 5x patch to a clean 6x code base and don't see any replicas staying down there (so far) so it looks like some other changes between 5x and 6x are making the startup process more robust as well.
    label: code-design

28. regarding the lambdas in the test, is there a functional difference between what is there and something like: {code} expect(mockCC.isZooKeeperAware()).andReturn(Boolean.TRUE).anyTimes(); expect(mockCC.getZkController()).andReturn(mockZKC).anyTimes(); expect(mockClusterState.getLiveNodes()).andReturn(liveNodes).anyTimes(); expect(mockZKC.getClusterState()).andReturn(mockClusterState).anyTimes(); {code} Looks like a lot of complexity added from the lambdas, but I'm not sure if there is a more subtle nuance.
29. **body:** {code} - public int getCoreLoadThreadCount() { - return coreLoadThreads; + public int getCoreLoadThreadCount(int def) { + return coreLoadThreads == null ? def : coreLoadThreads; } {code} It might make sense for this to look like this instead: {code} public int getCoreLoadThreadCount(boolean zkAware) { return coreLoadThreads == null ? (zkAware ? DEFAULT_CORE_LOAD_THREADS_IN_CLOUD : DEFAULT_CORE_LOAD_THREADS) : coreLoadThreads; } {code} That was the standalone v cloud logic doesn't have to leak out as much.
    label: code-design
30. 👍
31. Commit 2d1496c83d83bb6582af39af6cf272828d83c9e3 in lucene-solr's branch refs/heads/master from [~noble.paul] [https://git-wip-us.apache.org/repos/asf?p=lucene-solr.git;h=2d1496c] SOLR-7280: refactored to incorporate Mike's suggestions. Default thread count for cloud is limited to 8 now. In our internal teting 8 has given us the best stability during restarts
32. Commit fcd2cc57d595440de032cd3a58aabd72e69c3299 in lucene-solr's branch refs/heads/branch_6x from [~noble.paul] [https://git-wip-us.apache.org/repos/asf?p=lucene-solr.git;h=fcd2cc5] SOLR-7280: refactored to incorporate Mike's suggestions. Default thread count for cloud is limited to 8 now. In our internal teting 8 has given us the best stability during restarts
33. Commit 89a1fe661e7b73082d019543a83a7f511e74c9ca in lucene-solr's branch refs/heads/branch_6x from [~noble.paul] [https://git-wip-us.apache.org/repos/asf?p=lucene-solr.git;h=89a1fe6] SOLR-7280: refactored to incorporate Mike's suggestions. Default thread count for cloud is limited to 8 now. In our internal teting 8 has given us the best stability during restarts
34. Success! The problem I was having before was that I was _still_ getting OOM errors in my setup with the default 24 coreLoadTheads. Reducing it to 8 cured the problem, I ran my test setup all last night and there were zero problems. I've attached the patch for 5x. I had to re-implement the lambda expressions in the original, I _think_ I did the right thing in the new CoreSorterTest, but any checks welcome. This patch also sets the default coreLoadThreads to 8 as Noble discussed. [~mdrob] thanks for the pointer on the junit stuff BTW. I didn't incorporate your other suggestion, but having it in 6x and 7x suffices I think. This passes precommit and test as well as my stress test. So, the question becomes should this be merged into the 5x code line so it'll be picked up by any (hypothetical) 5x releases or just left here and we'll deal with whether it should be included in any new 5x release when the time comes? Any firm opinions? This topic has come up on more than one occasion, but even checking it into 5x still means people would have to build it themselves.
35. {{+ List<CountsForEachShard> l = new ArrayList<>();}} Could init this list to copy.size() {{+ List<CloudDescriptor> ret = new ArrayList<>();}} Same idea here, cc.getCores().size() Other than that, your unwrapped lambdas look fine to me.
36. backported the changes I committed to 6x today
37. Commit 9fbb2fe752b5baa224c33e0fd441cfb9082dd102 in lucene-solr's branch refs/heads/branch_5_5 from [~noble.paul] [https://git-wip-us.apache.org/repos/asf?p=lucene-solr.git;h=9fbb2fe] SOLR-7280: Load cores in sorted order & limit threads to improve cluster stability
38. Commit bcaf4999d83a5a4cbc7f02ecc8d1f00e4a4a7212 in lucene-solr's branch refs/heads/branch_5x from [~noble.paul] [https://git-wip-us.apache.org/repos/asf?p=lucene-solr.git;h=bcaf499] SOLR-7280: Load cores in sorted order & limit threads to improve cluster stability (cherry picked from commit 9fbb2fe)
39. Did this just reintroduce the old bug if you do > 8 replicas or it tried to address it somehow?
40. **body:** Yeah, looks like I have the same issue as I mentioned. I've got to veto this commit unless that issue is addressed nicely or (the ugly fix) you just make it fail if someone tries to create a shard with more replica than load threads.
    label: code-design
41. Not in my testing, but I'll give it another whirl tonight just to be sure. I'm thinking of, say, 3 collections spread over 4 JVMS each with 3 shards each with 150 replicas and 3 coreLoadThreads.... Plus I'll vary the order of bring the JVMs up and loop all night that way. I've already go the infrastructure in place....
42. **body:** I have no problem if it works. Perhaps some other unrelated changes have influenced how the load threads interact with leader election. I just don't see the problem explained or addressed and there is a previous JIRA issue with this exact problem. The only way I can see it would still not be a problem is if

waiting for a leader doesn't hold up core loading. I don't remember seeing that change though. You may not be able to test this simply with tests either - many tests lower the leaderVoteWait to very low to speed up tests. In production it should be like 3 minutes by default.
label: test

43. If it indeed remains an issue, another thing we might try is loading cores with a limited number of threads, but registering with ZK with many more, instead of doing both in the same thread as we are now.
44. bq. Plus I'll vary the order of bring the JVMs up and loop all night that way. If you simply do that and walk away and come back in the morning, it will work. The bug is that starting your shard will go from as fast as possible to 3 minutes plus just because you created too many replicas. And not only will it take 3 minutes, but all the replicas will not participate in the election as is currently designed. That's a bug, and one we have already fixed and one I'm not willing to allow back :)
45. **body:** [~mdrob] made a nice little test for this and the initial results make it look something was fishy before this or after it. Need to dig into it a little more.
label: test
46. Okay, I think I got it. At some point after the bug I'm referring to was resolved, registering in ZK got spun off into it's own thread and no longer holds up the core load thread. You can see this in ZkContainer. That effectively creates the situation I suggest above: bq. If it indeed remains an issue, another thing we might try is loading cores with a limited number of threads, but registering with ZK with many more, instead of doing both in the same thread as we are now. So we are okay on this issue. [~mdrob], it would be great to get some form of that test committed.
47. This is the test that Mark was talking about. (attached)
48. If anybody wants to apply that, they should also add a call to {{resetFactory()}} in the {{@AfterClass}} that I missed.
49. Well, I'll still test it for fun, if _you're_worried I should be more se. But it sounds like you're getting more comfortable with the approach. Anyway, a couple of things: bq: If you simply do that and walk away and come back in the morning, it will work.... These aren't Junit tests, but scripts because of similar concerns I had. They: 1> bring up and down real solr JVMs via shell scripts, so the default 3 minute timeout is in place That said, making it longer (say 20 minutes?) would emphasize the issue....and... 2> I have a monitor process that records how long it took for all the replicas to come up so I can see any anomalies. 3> brings JVMs up and down in different orders to avoid happening to have all the leaders on the first node that comes up. I've been nervous that the way I'm testing certainly isn't foolproof. bq: registering in ZK got spun off into it's own thread Hmm, interesting since the symptom I'm seeing is being unable to spawn native threads and a large spike in threads during startup (steady-state 1,200 threads, startup started crapping out around 2,600)..... upping the Xmx or Xss (or both) doesn't matter and bumping the ulimit didn't either....
50. It should work even if there is only one thread and many replicas. The idea is to sort your cores first in such a way that you prioritize replicas others are waiting for and deprioritize cores which depend on other 'down' nodes. So, this node will should not timeout.
51. Tests ran find last night FWIW...
52. [~mdrob@cloudera.com] why do you want to test with {{createCollection(collection, "conf1", 1, NodeConfigBuilder.DEFAULT_CORE_LOAD_THREADS_IN_CLOUD * 3)}}. Why not keep it at the default {{8}} {{ // we only need 1 node because we're going to be shoving all of the replicas on there to promote deadlock}} How do you plan to get a deadlock if you are only going to have one node? The deadlock happens because a node waits for other nodes to come up and those nodes wait for this node (directly or through a circular dependency)
53. **body:** You only need one node because this issue is about loading cores on one node. You also of course need more replicas than load threads to stimulate the old issue. The one issue with the test is I don't think it would fail nicely if the problem was reintroduced as it was when I saw it? As I saw it, if the problem was reintroduced the test would just take longer. We should probably just make sure it doesn't take 180 seconds + (or whatever we set the leaderVoteWait to in the test) to startup and elect a leader, and if it does, fail with a nice message.
label: code-design
54. bq. But it sounds like you're getting more comfortable with the approach. It has nothing to do with being comfortable with the approach. There was an existing bug that was fixed that no one working on this issue seems to yet comprehend. Which had me worried the problem still existed. I found that another change from another JIRA has decoupled core loading and leader election. All as I mention above. Details matter.
55. the zk register threads are still unbounded and that is done asynchronously. We are just limiting the core load threads. So, after this fix, we will not have this reoccurring if there is only one node. It is more

important to test it in a multi-node setup
56. Do note that all of my (non jUnit) tests were with 4 nodes. Of course having those built into junit tests is A Good Thing, just FYI.
57. Didn't close this when the code was committed.
58. Bulk close resolved issues after 6.2.0 release.
59. I've just been testing this and found a couple of issues: All the core registrations are done in background threads. This can flood the overseer queue. See CoreContainer.load() calls zkSys.registerInZk(core, true, false); I've increased leaderConflictResolveWait to 30min but every 15s I can see: org.apache.solr.handler.admin.PrepRecoveryOp; After 15 seconds, core ip_1224_shard1_replica1 (shard1 of ip_1224) still does not have state: recovering; forcing ClusterState update from ZooKeeper Again, I think this can flood the overseer queue.
60. Hey [~dk], could you file a new JIRA issue with this info and link it to this one? Sounds like you are probably correct, but this issue has already been released so any further changes or improvements needs a fresh issue on top of notifying the original issue.
61. **body:** It would seem the next step is probably to make the ZkController#preRegister phase more efficient. We would want to try and avoid all the individual down publish calls. This is what initially populates those entries in ZK though, which is why this was not simply switched to a more efficient 'down' node Overseer action like some other places. We may need another bulk Overseer command or perhaps we can expand the 'down' node one.
label: code-design
62. This is an old issue but nonetheless I want to bring to attention that the CoreSorter introduced here isn't doing its job. It only does so in its unit test but not in real-world usage (actual use inside CoreContainer). See SOLR-14342.