Item 2
git_comments:

git_commits:

1. **summary:** Parallelize streaming of different keyspaces for bootstrap and rebuild
   message: Parallelize streaming of different keyspaces for bootstrap and rebuild patch by Corentin Chary; reviewed by jasobrown for CASSANDRA-4663

github_issues:

github_issues_comments:

github_pulls:

github_pulls_comments:

github_pulls_reviews:

jira_issues:

1. **summary:** Parallelize streaming of different keyspaces for bootstrap and rebuild
   description: This is not fast enough when someone is using SSD and may be 10G link. We should try to create multiple connections and send multiple files in parallel. Current approach under utilize the link(even 1G). This change will improve the bootstrapping time of a node.

jira_issues_comments:

1. CASSANDRA-3668 let stream session and sstableloader handle multiple files.
2. So we can close this as Duplicate, right?
3. The related JIRA is for stable loader whereas I opened it for streaming during bootstrap. This came up in NGCC today cc [~tjake] ...Close it if we already have a JIRA for this
4. **body:** I made a change to RangeStreamer to created multiple StreamSessions per host (Split token ranges into chunks equal to the number of sockets). I saw a performance improvement (time-wise) of ~33%. Since the same code is used for bootstrap and nodetool rebuild, it will help in both cases. The one side-effect that operators need to be aware of is the number of SS Tables created on destination (since they will blow up corresponding to number of splits). I suggest we could add a -par option for nodetool rebuild command and let operators provide number of connections. For bootstrap, we can provide yaml setting and default to 1. (If we do decide to add yaml setting, do I need to worry about any version breaking stuff?) If that makes sense, I will create a patch for trunk.
   label: code-design
5. [~anubhavk] Can you explain what you mean by "since they will blow up corresponding to number of splits" I would think the easiest thing would be to send many files contained in the {{StreamSession}} at once vs one at a time. Since for each host this already contains many files.
6. **body:** bq. I would think the easiest thing would be to send many files contained in the StreamSession at once vs one at a time. this would definitely be the best thing to do, but would probably require a non-trivial change on the {{StreamSession}} to create/manage many connection handlers and split init/prepare phase from file sending and probably require some additional headers/messages. the easiest approach with the current protocol is to create multiple stream session objects, which can in fact increase the number of sstables created after streaming. with that said, I think we should go with the best (and not the easiest) approach, since we already suffer from creating too many sstables due to vnodes and that could be worsened with that approach.
   label: code-design
7. This seems very dangerous until CASSANDRA-11303 is merged (seems to be patch available).
8. **body:** Agree with Paulo. I don't like SS Tables blowing up. I will spend some time on sending multiple files at a time, and see what it offers.
   label: code-design
9. **body:** FWIW, the work I'm doing in #8457 (move internode messaging to netty) will need to pull along changing the streaming subsystem to netty, as well. I'm currently working on that (it can and will be a separate ticket from #8457), and I've been thinking about the entire streaming workflow, including sending multiple files in parallel. In my estimation, there are assumptions baked into the existing streaming workflow that makes sending files in parallel a non-trivial task; however, that does not mean it's impossible nor potentially beneficial, as my work is targeting 4.0.
   label: code-design
10. **body:** I ran some more tests on the original code and change with multiple sockets, and confirmed that the end-to-end time we see during streaming is a direct function of how long it takes for the sender to send bytes through (meaning sender is the only "slow" entity which makes the problem somewhat tangible). Then, I tested sending multiple files in parallel through some hacks, but as I was expecting it does not yield much improvements mainly because {{WritableByteChannel}} is a blocking channel across threads. From docs, "Only one write operation upon a writable

channel may be in progress at any given time. If one thread initiates a write operation upon a channel then any other thread that attempts to initiate another write operation will block until the first operation is complete." We would need to move to {{AsynchronousSocketChannel}} to get true parallelism (which obviously is a deeper change - not impossible though).
label: code-design
11. On the systems that I've looked at it was mainly the receiver that was CPU bound. I'm attaching a patch that provides a workaround for 3.X.
12. [~jasobrown] what do you think about the attached patch (until something better is done for 4.0)
13. Does it affect repair streaming sessions as well?
14. [~jlida]: [~iksaif]'s current patch only addresses bootstrap and rebuild (which can be seen as a variant of bootstrap)
15. [~iksaif] Unfortunately, I believe this patch isn't going to do what you want. In the existing code, {{connectionsPerHost}} is passed down to {{StreamCoordinator}}, where it is primarily going to be used on the side that is transferring files https://github.com/apache/cassandra/blob/trunk/src/java/org/apache/cassandra/streaming/StreamCoordinator.java#L189. In {{StreamCoordinator#sliceSSTableDetails()}}, we essentially round robin the sstable chunks across the number of connections we will use. It's also referenced from {{HostStreamingData#getOrCreateNextSession()}}, but that's not going to increase the inbound sessions (see next paragraph). Your current patch sets the {{connectionsPerHost}} at the receiver end of the stream (even though it would be the node that is initiating the stream session). As it's a bootstrapping node that is requesting the ranges from the peer, and given the structure and protocol of the current stream session implementation (session and protocol are essentially sequential, single threaded, and expect a single socket), I think you'd need to get down into session and protocol management code (and muck with a whole lot) in order to to parallelize from the (non-initiating) sending node. I'm happy to be wrong about my understanding here, so if you've tested it out and have seen good gains, please share :).
16. **body:** The thing is that as far as I understand StreamSession#getOrCreateNextSession() will create multiple sessions: {code} INFO [main] 2017-01-04 16:42:41,003 StorageService.java:1438 - JOINING: Starting to bootstrap... INFO [main] 2017-01-04 16:42:41,532 StreamResultFuture.java:90 - [Stream #cef1c0e0-d29c-11e6-b978-d922683f145b] Executing streaming plan for Bootstrap INFO [StreamConnectionEstablisher:1] 2017-01-04 16:42:41,538 StreamSession.java:266 - [Stream #cef1c0e0-d29c-11e6-b978-d922683f145b] Starting streaming to /10.50.4.115 INFO [StreamConnectionEstablisher:1] 2017-01-04 16:42:41,541 StreamCoordinator.java:264 - [Stream #cef1c0e0-d29c-11e6-b978-d922683f145b, ID#0] Beginning stream session with /10.50.4.115 INFO [STREAM-IN-/10.50.4.115:7000] 2017-01-04 16:42:41,562 StreamResultFuture.java:187 - [Stream #cef1c0e0-d29c-11e6-b978-d922683f145b] Session with /10.50.4.115 is complete INFO [StreamConnectionEstablisher:2] 2017-01-04 16:42:41,574 StreamSession.java:266 - [Stream #cef1c0e0-d29c-11e6-b978-d922683f145b] Starting streaming to /10.50.4.115 INFO [StreamConnectionEstablisher:2] 2017-01-04 16:42:41,576 StreamCoordinator.java:264 - [Stream #cef1c0e0-d29c-11e6-b978-d922683f145b, ID#1] Beginning stream session with /10.50.4.115 INFO [STREAM-IN-/10.50.4.115:7000] 2017-01-04 16:42:41,594 StreamResultFuture.java:187 - [Stream #cef1c0e0-d29c-11e6-b978-d922683f145b] Session with /10.50.4.115 is complete INFO [StreamConnectionEstablisher:3] 2017-01-04 16:42:41,601 StreamSession.java:266 - [Stream #cef1c0e0-d29c-11e6-b978-d922683f145b] Starting streaming to /10.50.4.115 INFO [StreamConnectionEstablisher:3] 2017-01-04 16:42:41,602 StreamCoordinator.java:264 - [Stream #cef1c0e0-d29c-11e6-b978-d922683f145b, ID#2] Beginning stream session with /10.50.4.115 INFO [STREAM-IN-/10.50.4.115:7000] 2017-01-04 16:42:41,607 StreamResultFuture.java:187 - [Stream #cef1c0e0-d29c-11e6-b978-d922683f145b] Session with /10.50.4.115 is complete INFO [StreamConnectionEstablisher:4] 2017-01-04 16:42:41,609 StreamSession.java:266 - [Stream #cef1c0e0-d29c-11e6-b978-d922683f145b] Starting streaming to /10.50.4.126 INFO [StreamConnectionEstablisher:4] 2017-01-04 16:42:41,610 StreamCoordinator.java:264 - [Stream #cef1c0e0-d29c-11e6-b978-d922683f145b, ID#0] Beginning stream session with /10.50.4.126 INFO [STREAM-IN-/10.50.4.126:7000] 2017-01-04 16:42:41,676 StreamResultFuture.java:173 - [Stream #cef1c0e0-d29c-11e6-b978-d922683f145b ID#0] Prepare completed. Receiving 7 files(1.181GiB), sending 0 files(0.000KiB) INFO [StreamConnectionEstablisher:5] 2017-01-04 16:42:41,679 StreamSession.java:266 - [Stream #cef1c0e0-d29c-11e6-b978-d922683f145b] Starting streaming to /10.50.4.126 INFO [StreamConnectionEstablisher:5] 2017-01-04 16:42:41,682 StreamCoordinator.java:264 - [Stream #cef1c0e0-d29c-11e6-b978-d922683f145b, ID#1] Beginning stream session with /10.50.4.126 INFO [STREAM-IN-/10.50.4.126:7000] 2017-01-04 16:42:41,718 StreamResultFuture.java:173 - [Stream #cef1c0e0-d29c-11e6-b978-d922683f145b ID#1] Prepare completed. Receiving 8 files(772.249MiB), sending 0 files(0.000KiB) INFO [StreamConnectionEstablisher:6] 2017-01-04 16:42:41,719 StreamSession.java:266 - [Stream #cef1c0e0-d29c-11e6-b978-d922683f145b] Starting streaming to /10.50.4.126 INFO [StreamConnectionEstablisher:6] 2017-01-04 16:42:41,720 StreamCoordinator.java:264 - [Stream #cef1c0e0-d29c-11e6-b978-d922683f145b, ID#2] Beginning stream session with /10.50.4.126 INFO [STREAM-IN-/10.50.4.126:7000] 2017-01-04 16:42:41,726 StreamResultFuture.java:187 - [Stream #cef1c0e0-d29c-11e6-b978-d922683f145b] Session with /10.50.4.126 is complete {code} Because of fetchAsync() iterates on keyspaces to call requestRanges() this will spread the keyspaces across multiple sessions (it would be even better to spread the ranges). Since the bottleneck is the CPU bound STREAM-IN thread (IncomingMessageHandler) it will improve the performances by creating multiple of these threads. With three keyspaces and three connections I was able to double the throughput and hopefully that is not just placebo effect.
label: code-design
17. Yeah, I think you are correct here. I didn't look that far up the stack (into {{RangeStreamer}}), but that will surely parallelize transfers by keyspace. I've kicked off the tests here: ||trunk|| |

[branch|https://github.com/jasobrown/cassandra/tree/4663-trunk]| | [dtest|http://cassci.datastax.com/view/Dev/view/jasobrown/job/jasobrown-4663-trunk-dtest/]| | [testall|http://cassci.datastax.com/view/Dev/view/jasobrown/job/jasobrown-4663-trunk-testall/]| If everything looks good with the tests, I'll rename the ticket to something more appropriate and commit.

18. OK, test look reasonable (no new dtest failures that are not currently failing on trunk). For the record, [~iksaif]'s patch parallelizes the streaming of the keyspaces being sent. If you have multiple keyspaces, you will potentially get an nice win. However, if there is only one keyspace (or most data is in one), there is no parallelization gained. I'll update the title of this jira to reflect that. As part of CASSANDRA-12229, or an immediate follow up, I'll be addressing sending sending files in parallel, so that request won't fall off the radar. +1'ing the ticket and will commit briefly
19. OOF 1/20
20. committed as sha {{4d67639d38b3e3a6fd0a3487a99b9755abda469d}} to 4.0. Thanks!