Item 63
git_comments:

1. Created for being used, so COMPUTING right away
2. Session created successfully
3. wake up single thread waiting for a session return (ok if not woken up, wait is short) Important to wake up a single one, otherwise of multiple waiting threads, all but one will immediately create new sessions
4. * * Number of commands currently using the session in {@link Status#EXECUTING}. There is one additional command * using the session and updating it if {@link #status} is {@link Status#COMPUTING}
5. * * Lock protecting access to {@link #sessionWrapperSet} and to {@link #creationsInProgress}
6. Second best case scenario: an available session
7. Best case scenario: an available session
8. Wait for a while before deciding what to do if waiting could help...
9. Either an existing session might be returned and become usable while we wait, or a session in the process of being created might finish creation, be used then returned and become usable. So we wait. wait 1 to 10 secs. Random to help spread wakeups.
10. We've waited, now we can either reuse immediately an available session, or immediately create a new one
11. * * Number of sessions currently being created but not yet present in {@link #sessionWrapperSet}. * * Access should only be done under the protection of {@link #lockObj}
12. * * Method returning an available session that can be used for {@link Status#COMPUTING}, either from the * {@link #sessionWrapperSet} cache or by creating a new one. The status of the returned session is set to {@link Status#COMPUTING}. * * Some waiting is done in two cases: * * A candidate session is present in {@link #sessionWrapperSet} but is still {@link Status#COMPUTING}, a random wait * is observed to see if the session gets freed to save a session creation and allow session reuse, * It is necessary to create a new session but there are already sessions in the process of being created, a * random wait is observed (if no waiting already occurred waiting for a session to become free) before creation * takes place, just in case one of the created sessions got used then {@link #returnSession(SessionWrapper)} in the meantime. * * * The random wait prevents the "thundering herd" effect when all threads needing a session at the same time create a new * one even though some differentiated waits could have led to better reuse and less session creations. * * @param allowWait usually <code>true</code> except in tests that know there's no point in waiting because nothing * will happen...
13. * * Returns an available session from the cache (the best one once cache strategies are defined), or null if no session * from the cache is available (i.e. all are still COMPUTING, are too old, wrong zk version or the cache is empty). * This method must be called while holding the monitor on {@link #lockObj}. * The method updates the session status to computing.
14. * * Nanoseconds (since/to some arbitrary time) when the session got created. Also used in logs (only in logs!) to identify the session.
15. * * Returns true if there's a session in the cache that could be returned (if it was free). This is required to * know if there's any point in waiting or if a new session should better be created right away.
16. We're going to create a new Session OUTSIDE of the critical section because session creation can take quite some time
17. * * A command is actively using and modifying the session to compute placements
18. Logging
19. JMM multithreaded access issue on lastUpdateTime.
20. **comment:** * * A command is not done yet processing its changes but no longer updates or even uses the session
    label: requirement
21. logOk
22. * * Sessions currently in use in {@link Status#COMPUTING} or {@link Status#EXECUTING} states. As soon as all * uses of a session are over, that session is removed from this set. Sessions not actively in use are NOT kept around. * * Access should only be done under the protection of {@link #lockObj}
23. used only by tests
24. Second session indeed reused when a new session is requested
25. This thread requests a session, computes using it for 50ms then returns is, executes for 1000ms more, releases the sessions and finishes.

26. The value asserted below is somewhat arbitrary. Running locally max seen is 10, so hopefully 30 is safe. Idea is to verify we do not allocate a high number of sessions even if many concurrent session requests arrive at the same time. The session computing time is short in purpose. If it were long, it would be expected for more sessions to be allocated.
27. **comment:** Done COMPUTING with second session, it can be reused
    label: code-design
28. **comment:** Must skip the wait time otherwise test takes a few seconds to run (and s1 is not returned now anyway so no point waiting).
    label: test
29. **comment:** Done COMPUTING with first session, it can be reused
    label: code-design
30. First session not yet released so is still in the cache
31. Got two sessions, they are different
32. First session indeed reused when a new session is requested
33. * * Verify number of sessions allocated when parallel session requests arrive is reasonable. * Test takes about 3 seconds to run.

git_commits:

1. **summary:** SOLR-14462: cache more than one autoscaling session (#1504)
   message: SOLR-14462: cache more than one autoscaling session (#1504) SOLR-14462: cache more than one autoscaling session

github_issues:

github_issues_comments:

github_pulls:

1. **title:** SOLR-14462: cache more than one autoscaling session
   body: # Description Allow caching and reusing multiple Sessions for computing replica placement. With a single cached session and fixed timeout, under high collection creation load multiple new sessions are created at once against the same cluster state and placement decisions do not take into account most other placement decisions. Moreover, existing code could not cache more than one session, so most sessions were used for a single placement decision. # Solution Multiple sessions can be cached and reused. Although not optimal (parallel sessions do not see the changes done by each other) this is still an improvement over existing implementation because more context is used for placement (sessions are reused). Also, less sessions are created because all created sessions are candidates for reuse (not only a single one). # Tests Tests in class TestPolicy were run and adapted to some method signature changes. A new test testMultiSessionsCache explicitly verifying multiple sessions can be cached was added. # Checklist Please review the following and check all that apply: - [X] I have reviewed the guidelines for [How to Contribute](https://wiki.apache.org/solr/HowToContribute) and my code conforms to the standards described there to the best of my ability. - [X] I have created a Jira issue and added the issue ID to my pull request title. - [X] I have given Solr maintainers [access] (https://help.github.com/en/articles/allowing-changes-to-a-pull-request-branch-created-from-a-fork) to contribute to my PR branch. (optional but recommended) - [X] I have developed this patch against the `master` branch. - [X] I have run `ant precommit` and the appropriate test suite. - [X] I have added tests for my changes. - [ ] I have added documentation for the [Ref Guide](https://github.com/apache/lucene-solr/tree/master/solr/solr-ref-guide) (for Solr changes only).
   label: code-design
2. **title:** SOLR-14462: cache more than one autoscaling session
   body: # Description Allow caching and reusing multiple Sessions for computing replica placement. With a single cached session and fixed timeout, under high collection creation load multiple new sessions are created at once against the same cluster state and placement decisions do not take into account most other placement decisions. Moreover, existing code could not cache more than one session, so most sessions were used for a single placement decision. # Solution Multiple sessions can be cached and reused. Although not optimal (parallel sessions do not see the changes done by each other) this is still an improvement over existing implementation because more context is used for placement (sessions are reused). Also, less sessions are created because all created sessions are candidates for reuse (not only a

single one). # Tests Tests in class TestPolicy were run and adapted to some method signature changes. A new test testMultiSessionsCache explicitly verifying multiple sessions can be cached was added. # Checklist Please review the following and check all that apply: - [X] I have reviewed the guidelines for [How to Contribute](https://wiki.apache.org/solr/HowToContribute) and my code conforms to the standards described there to the best of my ability. - [X] I have created a Jira issue and added the issue ID to my pull request title. - [X] I have given Solr maintainers [access] (https://help.github.com/en/articles/allowing-changes-to-a-pull-request-branch-created-from-a-fork) to contribute to my PR branch. (optional but recommended) - [X] I have developed this patch against the `master` branch. - [X] I have run `ant precommit` and the appropriate test suite. - [X] I have added tests for my changes. - [ ] I have added documentation for the [Ref Guide](https://github.com/apache/lucene-solr/tree/master/solr/solr-ref-guide) (for Solr changes only).

3. **title:** SOLR-14462: cache more than one autoscaling session
   body: # Description Allow caching and reusing multiple Sessions for computing replica placement. With a single cached session and fixed timeout, under high collection creation load multiple new sessions are created at once against the same cluster state and placement decisions do not take into account most other placement decisions. Moreover, existing code could not cache more than one session, so most sessions were used for a single placement decision. # Solution Multiple sessions can be cached and reused. Although not optimal (parallel sessions do not see the changes done by each other) this is still an improvement over existing implementation because more context is used for placement (sessions are reused). Also, less sessions are created because all created sessions are candidates for reuse (not only a single one). # Tests Tests in class TestPolicy were run and adapted to some method signature changes. A new test testMultiSessionsCache explicitly verifying multiple sessions can be cached was added. # Checklist Please review the following and check all that apply: - [X] I have reviewed the guidelines for [How to Contribute](https://wiki.apache.org/solr/HowToContribute) and my code conforms to the standards described there to the best of my ability. - [X] I have created a Jira issue and added the issue ID to my pull request title. - [X] I have given Solr maintainers [access] (https://help.github.com/en/articles/allowing-changes-to-a-pull-request-branch-created-from-a-fork) to contribute to my PR branch. (optional but recommended) - [X] I have developed this patch against the `master` branch. - [X] I have run `ant precommit` and the appropriate test suite. - [X] I have added tests for my changes. - [ ] I have added documentation for the [Ref Guide](https://github.com/apache/lucene-solr/tree/master/solr/solr-ref-guide) (for Solr changes only).
4. **title:** SOLR-14462: cache more than one autoscaling session
   body: # Description Allow caching and reusing multiple Sessions for computing replica placement. With a single cached session and fixed timeout, under high collection creation load multiple new sessions are created at once against the same cluster state and placement decisions do not take into account most other placement decisions. Moreover, existing code could not cache more than one session, so most sessions were used for a single placement decision. # Solution Multiple sessions can be cached and reused. Although not optimal (parallel sessions do not see the changes done by each other) this is still an improvement over existing implementation because more context is used for placement (sessions are reused). Also, less sessions are created because all created sessions are candidates for reuse (not only a single one). # Tests Tests in class TestPolicy were run and adapted to some method signature changes. A new test testMultiSessionsCache explicitly verifying multiple sessions can be cached was added. # Checklist Please review the following and check all that apply: - [X] I have reviewed the guidelines for [How to Contribute](https://wiki.apache.org/solr/HowToContribute) and my code conforms to the standards described there to the best of my ability. - [X] I have created a Jira issue and added the issue ID to my pull request title. - [X] I have given Solr maintainers [access] (https://help.github.com/en/articles/allowing-changes-to-a-pull-request-branch-created-from-a-fork) to contribute to my PR branch. (optional but recommended) - [X] I have developed this patch against the `master` branch. - [X] I have run `ant precommit` and the appropriate test suite. - [X] I have added tests for my changes. - [ ] I have added documentation for the [Ref Guide](https://github.com/apache/lucene-solr/tree/master/solr/solr-ref-guide) (for Solr changes only).
5. **title:** SOLR-14462: cache more than one autoscaling session
   body: # Description Allow caching and reusing multiple Sessions for computing replica placement. With a single cached session and fixed timeout, under high collection creation load multiple new sessions are created at once against the same cluster state and placement decisions do not take into account most other placement decisions. Moreover, existing code could not cache more than one session, so most sessions were used for a single placement decision. # Solution Multiple sessions can be cached and reused. Although not optimal (parallel sessions do not see the changes done by each other) this is still an improvement over existing implementation because more context is used for placement (sessions are

reused). Also, less sessions are created because all created sessions are candidates for reuse (not only a single one). # Tests Tests in class TestPolicy were run and adapted to some method signature changes. A new test testMultiSessionsCache explicitly verifying multiple sessions can be cached was added. # Checklist Please review the following and check all that apply: - [X] I have reviewed the guidelines for [How to Contribute](https://wiki.apache.org/solr/HowToContribute) and my code conforms to the standards described there to the best of my ability. - [X] I have created a Jira issue and added the issue ID to my pull request title. - [X] I have given Solr maintainers [access] (https://help.github.com/en/articles/allowing-changes-to-a-pull-request-branch-created-from-a-fork) to contribute to my PR branch. (optional but recommended) - [X] I have developed this patch against the `master` branch. - [X] I have run `ant precommit` and the appropriate test suite. - [X] I have added tests for my changes. - [ ] I have added documentation for the [Ref Guide](https://github.com/apache/lucene-solr/tree/master/solr/solr-ref-guide) (for Solr changes only).

6. **title:** SOLR-14462: cache more than one autoscaling session
   body: # Description Allow caching and reusing multiple Sessions for computing replica placement. With a single cached session and fixed timeout, under high collection creation load multiple new sessions are created at once against the same cluster state and placement decisions do not take into account most other placement decisions. Moreover, existing code could not cache more than one session, so most sessions were used for a single placement decision. # Solution Multiple sessions can be cached and reused. Although not optimal (parallel sessions do not see the changes done by each other) this is still an improvement over existing implementation because more context is used for placement (sessions are reused). Also, less sessions are created because all created sessions are candidates for reuse (not only a single one). # Tests Tests in class TestPolicy were run and adapted to some method signature changes. A new test testMultiSessionsCache explicitly verifying multiple sessions can be cached was added. # Checklist Please review the following and check all that apply: - [X] I have reviewed the guidelines for [How to Contribute](https://wiki.apache.org/solr/HowToContribute) and my code conforms to the standards described there to the best of my ability. - [X] I have created a Jira issue and added the issue ID to my pull request title. - [X] I have given Solr maintainers [access] (https://help.github.com/en/articles/allowing-changes-to-a-pull-request-branch-created-from-a-fork) to contribute to my PR branch. (optional but recommended) - [X] I have developed this patch against the `master` branch. - [X] I have run `ant precommit` and the appropriate test suite. - [X] I have added tests for my changes. - [ ] I have added documentation for the [Ref Guide](https://github.com/apache/lucene-solr/tree/master/solr/solr-ref-guide) (for Solr changes only).
7. **title:** SOLR-14462: cache more than one autoscaling session
   body: # Description Allow caching and reusing multiple Sessions for computing replica placement. With a single cached session and fixed timeout, under high collection creation load multiple new sessions are created at once against the same cluster state and placement decisions do not take into account most other placement decisions. Moreover, existing code could not cache more than one session, so most sessions were used for a single placement decision. # Solution Multiple sessions can be cached and reused. Although not optimal (parallel sessions do not see the changes done by each other) this is still an improvement over existing implementation because more context is used for placement (sessions are reused). Also, less sessions are created because all created sessions are candidates for reuse (not only a single one). # Tests Tests in class TestPolicy were run and adapted to some method signature changes. A new test testMultiSessionsCache explicitly verifying multiple sessions can be cached was added. # Checklist Please review the following and check all that apply: - [X] I have reviewed the guidelines for [How to Contribute](https://wiki.apache.org/solr/HowToContribute) and my code conforms to the standards described there to the best of my ability. - [X] I have created a Jira issue and added the issue ID to my pull request title. - [X] I have given Solr maintainers [access] (https://help.github.com/en/articles/allowing-changes-to-a-pull-request-branch-created-from-a-fork) to contribute to my PR branch. (optional but recommended) - [X] I have developed this patch against the `master` branch. - [X] I have run `ant precommit` and the appropriate test suite. - [X] I have added tests for my changes. - [ ] I have added documentation for the [Ref Guide](https://github.com/apache/lucene-solr/tree/master/solr/solr-ref-guide) (for Solr changes only).
   label: code-design
8. **title:** SOLR-14462: cache more than one autoscaling session
   body: # Description Allow caching and reusing multiple Sessions for computing replica placement. With a single cached session and fixed timeout, under high collection creation load multiple new sessions are created at once against the same cluster state and placement decisions do not take into account most other placement decisions. Moreover, existing code could not cache more than one session, so most sessions were used for a single placement decision. # Solution Multiple sessions can be cached and reused.

Although not optimal (parallel sessions do not see the changes done by each other) this is still an improvement over existing implementation because more context is used for placement (sessions are reused). Also, less sessions are created because all created sessions are candidates for reuse (not only a single one). # Tests Tests in class TestPolicy were run and adapted to some method signature changes. A new test testMultiSessionsCache explicitly verifying multiple sessions can be cached was added. # Checklist Please review the following and check all that apply: - [X] I have reviewed the guidelines for [How to Contribute](https://wiki.apache.org/solr/HowToContribute) and my code conforms to the standards described there to the best of my ability. - [X] I have created a Jira issue and added the issue ID to my pull request title. - [X] I have given Solr maintainers [access] (https://help.github.com/en/articles/allowing-changes-to-a-pull-request-branch-created-from-a-fork) to contribute to my PR branch. (optional but recommended) - [X] I have developed this patch against the `master` branch. - [X] I have run `ant precommit` and the appropriate test suite. - [X] I have added tests for my changes. - [ ] I have added documentation for the [Ref Guide](https://github.com/apache/lucene-solr/tree/master/solr/solr-ref-guide) (for Solr changes only).

9. **title:** SOLR-14462: cache more than one autoscaling session
   body: # Description Allow caching and reusing multiple Sessions for computing replica placement. With a single cached session and fixed timeout, under high collection creation load multiple new sessions are created at once against the same cluster state and placement decisions do not take into account most other placement decisions. Moreover, existing code could not cache more than one session, so most sessions were used for a single placement decision. # Solution Multiple sessions can be cached and reused. Although not optimal (parallel sessions do not see the changes done by each other) this is still an improvement over existing implementation because more context is used for placement (sessions are reused). Also, less sessions are created because all created sessions are candidates for reuse (not only a single one). # Tests Tests in class TestPolicy were run and adapted to some method signature changes. A new test testMultiSessionsCache explicitly verifying multiple sessions can be cached was added. # Checklist Please review the following and check all that apply: - [X] I have reviewed the guidelines for [How to Contribute](https://wiki.apache.org/solr/HowToContribute) and my code conforms to the standards described there to the best of my ability. - [X] I have created a Jira issue and added the issue ID to my pull request title. - [X] I have given Solr maintainers [access] (https://help.github.com/en/articles/allowing-changes-to-a-pull-request-branch-created-from-a-fork) to contribute to my PR branch. (optional but recommended) - [X] I have developed this patch against the `master` branch. - [X] I have run `ant precommit` and the appropriate test suite. - [X] I have added tests for my changes. - [ ] I have added documentation for the [Ref Guide](https://github.com/apache/lucene-solr/tree/master/solr/solr-ref-guide) (for Solr changes only).
10. **title:** SOLR-14462: cache more than one autoscaling session
    body: # Description Allow caching and reusing multiple Sessions for computing replica placement. With a single cached session and fixed timeout, under high collection creation load multiple new sessions are created at once against the same cluster state and placement decisions do not take into account most other placement decisions. Moreover, existing code could not cache more than one session, so most sessions were used for a single placement decision. # Solution Multiple sessions can be cached and reused. Although not optimal (parallel sessions do not see the changes done by each other) this is still an improvement over existing implementation because more context is used for placement (sessions are reused). Also, less sessions are created because all created sessions are candidates for reuse (not only a single one). # Tests Tests in class TestPolicy were run and adapted to some method signature changes. A new test testMultiSessionsCache explicitly verifying multiple sessions can be cached was added. # Checklist Please review the following and check all that apply: - [X] I have reviewed the guidelines for [How to Contribute](https://wiki.apache.org/solr/HowToContribute) and my code conforms to the standards described there to the best of my ability. - [X] I have created a Jira issue and added the issue ID to my pull request title. - [X] I have given Solr maintainers [access] (https://help.github.com/en/articles/allowing-changes-to-a-pull-request-branch-created-from-a-fork) to contribute to my PR branch. (optional but recommended) - [X] I have developed this patch against the `master` branch. - [X] I have run `ant precommit` and the appropriate test suite. - [X] I have added tests for my changes. - [ ] I have added documentation for the [Ref Guide](https://github.com/apache/lucene-solr/tree/master/solr/solr-ref-guide) (for Solr changes only).
    label: code-design
11. **title:** SOLR-14462: cache more than one autoscaling session
    body: # Description Allow caching and reusing multiple Sessions for computing replica placement. With a single cached session and fixed timeout, under high collection creation load multiple new sessions are created at once against the same cluster state and placement decisions do not take into account most other

placement decisions. Moreover, existing code could not cache more than one session, so most sessions were used for a single placement decision. # Solution Multiple sessions can be cached and reused. Although not optimal (parallel sessions do not see the changes done by each other) this is still an improvement over existing implementation because more context is used for placement (sessions are reused). Also, less sessions are created because all created sessions are candidates for reuse (not only a single one). # Tests Tests in class TestPolicy were run and adapted to some method signature changes. A new test testMultiSessionsCache explicitly verifying multiple sessions can be cached was added. # Checklist Please review the following and check all that apply: - [X] I have reviewed the guidelines for [How to Contribute](https://wiki.apache.org/solr/HowToContribute) and my code conforms to the standards described there to the best of my ability. - [X] I have created a Jira issue and added the issue ID to my pull request title. - [X] I have given Solr maintainers [access] (https://help.github.com/en/articles/allowing-changes-to-a-pull-request-branch-created-from-a-fork) to contribute to my PR branch. (optional but recommended) - [X] I have developed this patch against the `master` branch. - [X] I have run `ant precommit` and the appropriate test suite. - [X] I have added tests for my changes. - [ ] I have added documentation for the [Ref Guide](https://github.com/apache/lucene-solr/tree/master/solr/solr-ref-guide) (for Solr changes only).

12. **title:** SOLR-14462: cache more than one autoscaling session
    body: # Description Allow caching and reusing multiple Sessions for computing replica placement. With a single cached session and fixed timeout, under high collection creation load multiple new sessions are created at once against the same cluster state and placement decisions do not take into account most other placement decisions. Moreover, existing code could not cache more than one session, so most sessions were used for a single placement decision. # Solution Multiple sessions can be cached and reused. Although not optimal (parallel sessions do not see the changes done by each other) this is still an improvement over existing implementation because more context is used for placement (sessions are reused). Also, less sessions are created because all created sessions are candidates for reuse (not only a single one). # Tests Tests in class TestPolicy were run and adapted to some method signature changes. A new test testMultiSessionsCache explicitly verifying multiple sessions can be cached was added. # Checklist Please review the following and check all that apply: - [X] I have reviewed the guidelines for [How to Contribute](https://wiki.apache.org/solr/HowToContribute) and my code conforms to the standards described there to the best of my ability. - [X] I have created a Jira issue and added the issue ID to my pull request title. - [X] I have given Solr maintainers [access] (https://help.github.com/en/articles/allowing-changes-to-a-pull-request-branch-created-from-a-fork) to contribute to my PR branch. (optional but recommended) - [X] I have developed this patch against the `master` branch. - [X] I have run `ant precommit` and the appropriate test suite. - [X] I have added tests for my changes. - [ ] I have added documentation for the [Ref Guide](https://github.com/apache/lucene-solr/tree/master/solr/solr-ref-guide) (for Solr changes only).
    label: code-design
13. **title:** SOLR-14462: cache more than one autoscaling session
    body: # Description Allow caching and reusing multiple Sessions for computing replica placement. With a single cached session and fixed timeout, under high collection creation load multiple new sessions are created at once against the same cluster state and placement decisions do not take into account most other placement decisions. Moreover, existing code could not cache more than one session, so most sessions were used for a single placement decision. # Solution Multiple sessions can be cached and reused. Although not optimal (parallel sessions do not see the changes done by each other) this is still an improvement over existing implementation because more context is used for placement (sessions are reused). Also, less sessions are created because all created sessions are candidates for reuse (not only a single one). # Tests Tests in class TestPolicy were run and adapted to some method signature changes. A new test testMultiSessionsCache explicitly verifying multiple sessions can be cached was added. # Checklist Please review the following and check all that apply: - [X] I have reviewed the guidelines for [How to Contribute](https://wiki.apache.org/solr/HowToContribute) and my code conforms to the standards described there to the best of my ability. - [X] I have created a Jira issue and added the issue ID to my pull request title. - [X] I have given Solr maintainers [access] (https://help.github.com/en/articles/allowing-changes-to-a-pull-request-branch-created-from-a-fork) to contribute to my PR branch. (optional but recommended) - [X] I have developed this patch against the `master` branch. - [X] I have run `ant precommit` and the appropriate test suite. - [X] I have added tests for my changes. - [ ] I have added documentation for the [Ref Guide](https://github.com/apache/lucene-solr/tree/master/solr/solr-ref-guide) (for Solr changes only).
14. **title:** SOLR-14462: cache more than one autoscaling session

body: # Description Allow caching and reusing multiple Sessions for computing replica placement. With a single cached session and fixed timeout, under high collection creation load multiple new sessions are created at once against the same cluster state and placement decisions do not take into account most other placement decisions. Moreover, existing code could not cache more than one session, so most sessions were used for a single placement decision. # Solution Multiple sessions can be cached and reused. Although not optimal (parallel sessions do not see the changes done by each other) this is still an improvement over existing implementation because more context is used for placement (sessions are reused). Also, less sessions are created because all created sessions are candidates for reuse (not only a single one). # Tests Tests in class TestPolicy were run and adapted to some method signature changes. A new test testMultiSessionsCache explicitly verifying multiple sessions can be cached was added. # Checklist Please review the following and check all that apply: - [X] I have reviewed the guidelines for [How to Contribute](https://wiki.apache.org/solr/HowToContribute) and my code conforms to the standards described there to the best of my ability. - [X] I have created a Jira issue and added the issue ID to my pull request title. - [X] I have given Solr maintainers [access] (https://help.github.com/en/articles/allowing-changes-to-a-pull-request-branch-created-from-a-fork) to contribute to my PR branch. (optional but recommended) - [X] I have developed this patch against the `master` branch. - [X] I have run `ant precommit` and the appropriate test suite. - [X] I have added tests for my changes. - [ ] I have added documentation for the [Ref Guide](https://github.com/apache/lucene-solr/tree/master/solr/solr-ref-guide) (for Solr changes only).
label: code-design

15. **title:** SOLR-14462: cache more than one autoscaling session
body: # Description Allow caching and reusing multiple Sessions for computing replica placement. With a single cached session and fixed timeout, under high collection creation load multiple new sessions are created at once against the same cluster state and placement decisions do not take into account most other placement decisions. Moreover, existing code could not cache more than one session, so most sessions were used for a single placement decision. # Solution Multiple sessions can be cached and reused. Although not optimal (parallel sessions do not see the changes done by each other) this is still an improvement over existing implementation because more context is used for placement (sessions are reused). Also, less sessions are created because all created sessions are candidates for reuse (not only a single one). # Tests Tests in class TestPolicy were run and adapted to some method signature changes. A new test testMultiSessionsCache explicitly verifying multiple sessions can be cached was added. # Checklist Please review the following and check all that apply: - [X] I have reviewed the guidelines for [How to Contribute](https://wiki.apache.org/solr/HowToContribute) and my code conforms to the standards described there to the best of my ability. - [X] I have created a Jira issue and added the issue ID to my pull request title. - [X] I have given Solr maintainers [access] (https://help.github.com/en/articles/allowing-changes-to-a-pull-request-branch-created-from-a-fork) to contribute to my PR branch. (optional but recommended) - [X] I have developed this patch against the `master` branch. - [X] I have run `ant precommit` and the appropriate test suite. - [X] I have added tests for my changes. - [ ] I have added documentation for the [Ref Guide](https://github.com/apache/lucene-solr/tree/master/solr/solr-ref-guide) (for Solr changes only).
16. **title:** SOLR-14462: cache more than one autoscaling session
body: # Description Allow caching and reusing multiple Sessions for computing replica placement. With a single cached session and fixed timeout, under high collection creation load multiple new sessions are created at once against the same cluster state and placement decisions do not take into account most other placement decisions. Moreover, existing code could not cache more than one session, so most sessions were used for a single placement decision. # Solution Multiple sessions can be cached and reused. Although not optimal (parallel sessions do not see the changes done by each other) this is still an improvement over existing implementation because more context is used for placement (sessions are reused). Also, less sessions are created because all created sessions are candidates for reuse (not only a single one). # Tests Tests in class TestPolicy were run and adapted to some method signature changes. A new test testMultiSessionsCache explicitly verifying multiple sessions can be cached was added. # Checklist Please review the following and check all that apply: - [X] I have reviewed the guidelines for [How to Contribute](https://wiki.apache.org/solr/HowToContribute) and my code conforms to the standards described there to the best of my ability. - [X] I have created a Jira issue and added the issue ID to my pull request title. - [X] I have given Solr maintainers [access] (https://help.github.com/en/articles/allowing-changes-to-a-pull-request-branch-created-from-a-fork) to contribute to my PR branch. (optional but recommended) - [X] I have developed this patch against the `master` branch. - [X] I have run `ant precommit` and the appropriate test suite. - [X] I have added tests

for my changes. - [ ] I have added documentation for the [Ref Guide](https://github.com/apache/lucene-solr/tree/master/solr/solr-ref-guide) (for Solr changes only).

17. **title:** SOLR-14462: cache more than one autoscaling session
    body: # Description Allow caching and reusing multiple Sessions for computing replica placement. With a single cached session and fixed timeout, under high collection creation load multiple new sessions are created at once against the same cluster state and placement decisions do not take into account most other placement decisions. Moreover, existing code could not cache more than one session, so most sessions were used for a single placement decision. # Solution Multiple sessions can be cached and reused. Although not optimal (parallel sessions do not see the changes done by each other) this is still an improvement over existing implementation because more context is used for placement (sessions are reused). Also, less sessions are created because all created sessions are candidates for reuse (not only a single one). # Tests Tests in class TestPolicy were run and adapted to some method signature changes. A new test testMultiSessionsCache explicitly verifying multiple sessions can be cached was added. # Checklist Please review the following and check all that apply: - [X] I have reviewed the guidelines for [How to Contribute](https://wiki.apache.org/solr/HowToContribute) and my code conforms to the standards described there to the best of my ability. - [X] I have created a Jira issue and added the issue ID to my pull request title. - [X] I have given Solr maintainers [access] (https://help.github.com/en/articles/allowing-changes-to-a-pull-request-branch-created-from-a-fork) to contribute to my PR branch. (optional but recommended) - [X] I have developed this patch against the `master` branch. - [X] I have run `ant precommit` and the appropriate test suite. - [X] I have added tests for my changes. - [ ] I have added documentation for the [Ref Guide](https://github.com/apache/lucene-solr/tree/master/solr/solr-ref-guide) (for Solr changes only).
    label: code-design
18. **title:** SOLR-14462: cache more than one autoscaling session
    body: # Description Allow caching and reusing multiple Sessions for computing replica placement. With a single cached session and fixed timeout, under high collection creation load multiple new sessions are created at once against the same cluster state and placement decisions do not take into account most other placement decisions. Moreover, existing code could not cache more than one session, so most sessions were used for a single placement decision. # Solution Multiple sessions can be cached and reused. Although not optimal (parallel sessions do not see the changes done by each other) this is still an improvement over existing implementation because more context is used for placement (sessions are reused). Also, less sessions are created because all created sessions are candidates for reuse (not only a single one). # Tests Tests in class TestPolicy were run and adapted to some method signature changes. A new test testMultiSessionsCache explicitly verifying multiple sessions can be cached was added. # Checklist Please review the following and check all that apply: - [X] I have reviewed the guidelines for [How to Contribute](https://wiki.apache.org/solr/HowToContribute) and my code conforms to the standards described there to the best of my ability. - [X] I have created a Jira issue and added the issue ID to my pull request title. - [X] I have given Solr maintainers [access] (https://help.github.com/en/articles/allowing-changes-to-a-pull-request-branch-created-from-a-fork) to contribute to my PR branch. (optional but recommended) - [X] I have developed this patch against the `master` branch. - [X] I have run `ant precommit` and the appropriate test suite. - [X] I have added tests for my changes. - [ ] I have added documentation for the [Ref Guide](https://github.com/apache/lucene-solr/tree/master/solr/solr-ref-guide) (for Solr changes only).
19. **title:** SOLR-14462: cache more than one autoscaling session
    body: # Description Allow caching and reusing multiple Sessions for computing replica placement. With a single cached session and fixed timeout, under high collection creation load multiple new sessions are created at once against the same cluster state and placement decisions do not take into account most other placement decisions. Moreover, existing code could not cache more than one session, so most sessions were used for a single placement decision. # Solution Multiple sessions can be cached and reused. Although not optimal (parallel sessions do not see the changes done by each other) this is still an improvement over existing implementation because more context is used for placement (sessions are reused). Also, less sessions are created because all created sessions are candidates for reuse (not only a single one). # Tests Tests in class TestPolicy were run and adapted to some method signature changes. A new test testMultiSessionsCache explicitly verifying multiple sessions can be cached was added. # Checklist Please review the following and check all that apply: - [X] I have reviewed the guidelines for [How to Contribute](https://wiki.apache.org/solr/HowToContribute) and my code conforms to the standards described there to the best of my ability. - [X] I have created a Jira issue and added the issue ID to my pull request title. - [X] I have given Solr maintainers [access] (https://help.github.com/en/articles/allowing-changes-to-a-pull-request-branch-created-from-a-fork) to

contribute to my PR branch. (optional but recommended) - [X] I have developed this patch against the `master` branch. - [X] I have run `ant precommit` and the appropriate test suite. - [X] I have added tests for my changes. - [ ] I have added documentation for the [Ref Guide](https://github.com/apache/lucene-solr/tree/master/solr/solr-ref-guide) (for Solr changes only).
label: code-design

20. **title:** SOLR-14462: cache more than one autoscaling session
body: # Description Allow caching and reusing multiple Sessions for computing replica placement. With a single cached session and fixed timeout, under high collection creation load multiple new sessions are created at once against the same cluster state and placement decisions do not take into account most other placement decisions. Moreover, existing code could not cache more than one session, so most sessions were used for a single placement decision. # Solution Multiple sessions can be cached and reused. Although not optimal (parallel sessions do not see the changes done by each other) this is still an improvement over existing implementation because more context is used for placement (sessions are reused). Also, less sessions are created because all created sessions are candidates for reuse (not only a single one). # Tests Tests in class TestPolicy were run and adapted to some method signature changes. A new test testMultiSessionsCache explicitly verifying multiple sessions can be cached was added. # Checklist Please review the following and check all that apply: - [X] I have reviewed the guidelines for [How to Contribute](https://wiki.apache.org/solr/HowToContribute) and my code conforms to the standards described there to the best of my ability. - [X] I have created a Jira issue and added the issue ID to my pull request title. - [X] I have given Solr maintainers [access] (https://help.github.com/en/articles/allowing-changes-to-a-pull-request-branch-created-from-a-fork) to contribute to my PR branch. (optional but recommended) - [X] I have developed this patch against the `master` branch. - [X] I have run `ant precommit` and the appropriate test suite. - [X] I have added tests for my changes. - [ ] I have added documentation for the [Ref Guide](https://github.com/apache/lucene-solr/tree/master/solr/solr-ref-guide) (for Solr changes only).
21. **title:** SOLR-14462: cache more than one autoscaling session
body: # Description Allow caching and reusing multiple Sessions for computing replica placement. With a single cached session and fixed timeout, under high collection creation load multiple new sessions are created at once against the same cluster state and placement decisions do not take into account most other placement decisions. Moreover, existing code could not cache more than one session, so most sessions were used for a single placement decision. # Solution Multiple sessions can be cached and reused. Although not optimal (parallel sessions do not see the changes done by each other) this is still an improvement over existing implementation because more context is used for placement (sessions are reused). Also, less sessions are created because all created sessions are candidates for reuse (not only a single one). # Tests Tests in class TestPolicy were run and adapted to some method signature changes. A new test testMultiSessionsCache explicitly verifying multiple sessions can be cached was added. # Checklist Please review the following and check all that apply: - [X] I have reviewed the guidelines for [How to Contribute](https://wiki.apache.org/solr/HowToContribute) and my code conforms to the standards described there to the best of my ability. - [X] I have created a Jira issue and added the issue ID to my pull request title. - [X] I have given Solr maintainers [access] (https://help.github.com/en/articles/allowing-changes-to-a-pull-request-branch-created-from-a-fork) to contribute to my PR branch. (optional but recommended) - [X] I have developed this patch against the `master` branch. - [X] I have run `ant precommit` and the appropriate test suite. - [X] I have added tests for my changes. - [ ] I have added documentation for the [Ref Guide](https://github.com/apache/lucene-solr/tree/master/solr/solr-ref-guide) (for Solr changes only).
22. **title:** SOLR-14462: cache more than one autoscaling session
body: # Description Allow caching and reusing multiple Sessions for computing replica placement. With a single cached session and fixed timeout, under high collection creation load multiple new sessions are created at once against the same cluster state and placement decisions do not take into account most other placement decisions. Moreover, existing code could not cache more than one session, so most sessions were used for a single placement decision. # Solution Multiple sessions can be cached and reused. Although not optimal (parallel sessions do not see the changes done by each other) this is still an improvement over existing implementation because more context is used for placement (sessions are reused). Also, less sessions are created because all created sessions are candidates for reuse (not only a single one). # Tests Tests in class TestPolicy were run and adapted to some method signature changes. A new test testMultiSessionsCache explicitly verifying multiple sessions can be cached was added. # Checklist Please review the following and check all that apply: - [X] I have reviewed the guidelines for [How to Contribute](https://wiki.apache.org/solr/HowToContribute) and my code conforms to the standards described there to the best of my ability. - [X] I have created a Jira issue and added the issue ID

to my pull request title. - [X] I have given Solr maintainers [access](https://help.github.com/en/articles/allowing-changes-to-a-pull-request-branch-created-from-a-fork) to contribute to my PR branch. (optional but recommended) - [X] I have developed this patch against the `master` branch. - [X] I have run `ant precommit` and the appropriate test suite. - [X] I have added tests for my changes. - [ ] I have added documentation for the [Ref Guide](https://github.com/apache/lucene-solr/tree/master/solr/solr-ref-guide) (for Solr changes only).

23. **title:** SOLR-14462: cache more than one autoscaling session
body: # Description Allow caching and reusing multiple Sessions for computing replica placement. With a single cached session and fixed timeout, under high collection creation load multiple new sessions are created at once against the same cluster state and placement decisions do not take into account most other placement decisions. Moreover, existing code could not cache more than one session, so most sessions were used for a single placement decision. # Solution Multiple sessions can be cached and reused. Although not optimal (parallel sessions do not see the changes done by each other) this is still an improvement over existing implementation because more context is used for placement (sessions are reused). Also, less sessions are created because all created sessions are candidates for reuse (not only a single one). # Tests Tests in class TestPolicy were run and adapted to some method signature changes. A new test testMultiSessionsCache explicitly verifying multiple sessions can be cached was added. # Checklist Please review the following and check all that apply: - [X] I have reviewed the guidelines for [How to Contribute](https://wiki.apache.org/solr/HowToContribute) and my code conforms to the standards described there to the best of my ability. - [X] I have created a Jira issue and added the issue ID to my pull request title. - [X] I have given Solr maintainers [access](https://help.github.com/en/articles/allowing-changes-to-a-pull-request-branch-created-from-a-fork) to contribute to my PR branch. (optional but recommended) - [X] I have developed this patch against the `master` branch. - [X] I have run `ant precommit` and the appropriate test suite. - [X] I have added tests for my changes. - [ ] I have added documentation for the [Ref Guide](https://github.com/apache/lucene-solr/tree/master/solr/solr-ref-guide) (for Solr changes only).
24. **title:** SOLR-14462: cache more than one autoscaling session
body: # Description Allow caching and reusing multiple Sessions for computing replica placement. With a single cached session and fixed timeout, under high collection creation load multiple new sessions are created at once against the same cluster state and placement decisions do not take into account most other placement decisions. Moreover, existing code could not cache more than one session, so most sessions were used for a single placement decision. # Solution Multiple sessions can be cached and reused. Although not optimal (parallel sessions do not see the changes done by each other) this is still an improvement over existing implementation because more context is used for placement (sessions are reused). Also, less sessions are created because all created sessions are candidates for reuse (not only a single one). # Tests Tests in class TestPolicy were run and adapted to some method signature changes. A new test testMultiSessionsCache explicitly verifying multiple sessions can be cached was added. # Checklist Please review the following and check all that apply: - [X] I have reviewed the guidelines for [How to Contribute](https://wiki.apache.org/solr/HowToContribute) and my code conforms to the standards described there to the best of my ability. - [X] I have created a Jira issue and added the issue ID to my pull request title. - [X] I have given Solr maintainers [access](https://help.github.com/en/articles/allowing-changes-to-a-pull-request-branch-created-from-a-fork) to contribute to my PR branch. (optional but recommended) - [X] I have developed this patch against the `master` branch. - [X] I have run `ant precommit` and the appropriate test suite. - [X] I have added tests for my changes. - [ ] I have added documentation for the [Ref Guide](https://github.com/apache/lucene-solr/tree/master/solr/solr-ref-guide) (for Solr changes only).
25. **title:** SOLR-14462: cache more than one autoscaling session
body: # Description Allow caching and reusing multiple Sessions for computing replica placement. With a single cached session and fixed timeout, under high collection creation load multiple new sessions are created at once against the same cluster state and placement decisions do not take into account most other placement decisions. Moreover, existing code could not cache more than one session, so most sessions were used for a single placement decision. # Solution Multiple sessions can be cached and reused. Although not optimal (parallel sessions do not see the changes done by each other) this is still an improvement over existing implementation because more context is used for placement (sessions are reused). Also, less sessions are created because all created sessions are candidates for reuse (not only a single one). # Tests Tests in class TestPolicy were run and adapted to some method signature changes. A new test testMultiSessionsCache explicitly verifying multiple sessions can be cached was added. # Checklist Please review the following and check all that apply: - [X] I have reviewed the guidelines for [How to Contribute](https://wiki.apache.org/solr/HowToContribute) and my code conforms to the

standards described there to the best of my ability. - [X] I have created a Jira issue and added the issue ID to my pull request title. - [X] I have given Solr maintainers [access] (https://help.github.com/en/articles/allowing-changes-to-a-pull-request-branch-created-from-a-fork) to contribute to my PR branch. (optional but recommended) - [X] I have developed this patch against the `master` branch. - [X] I have run `ant precommit` and the appropriate test suite. - [X] I have added tests for my changes. - [ ] I have added documentation for the [Ref Guide](https://github.com/apache/lucene-solr/tree/master/solr/solr-ref-guide) (for Solr changes only).

26. **title:** SOLR-14462: cache more than one autoscaling session
    body: # Description Allow caching and reusing multiple Sessions for computing replica placement. With a single cached session and fixed timeout, under high collection creation load multiple new sessions are created at once against the same cluster state and placement decisions do not take into account most other placement decisions. Moreover, existing code could not cache more than one session, so most sessions were used for a single placement decision. # Solution Multiple sessions can be cached and reused. Although not optimal (parallel sessions do not see the changes done by each other) this is still an improvement over existing implementation because more context is used for placement (sessions are reused). Also, less sessions are created because all created sessions are candidates for reuse (not only a single one). # Tests Tests in class TestPolicy were run and adapted to some method signature changes. A new test testMultiSessionsCache explicitly verifying multiple sessions can be cached was added. # Checklist Please review the following and check all that apply: - [X] I have reviewed the guidelines for [How to Contribute](https://wiki.apache.org/solr/HowToContribute) and my code conforms to the standards described there to the best of my ability. - [X] I have created a Jira issue and added the issue ID to my pull request title. - [X] I have given Solr maintainers [access] (https://help.github.com/en/articles/allowing-changes-to-a-pull-request-branch-created-from-a-fork) to contribute to my PR branch. (optional but recommended) - [X] I have developed this patch against the `master` branch. - [X] I have run `ant precommit` and the appropriate test suite. - [X] I have added tests for my changes. - [ ] I have added documentation for the [Ref Guide](https://github.com/apache/lucene-solr/tree/master/solr/solr-ref-guide) (for Solr changes only).
27. **title:** SOLR-14462: cache more than one autoscaling session
    body: # Description Allow caching and reusing multiple Sessions for computing replica placement. With a single cached session and fixed timeout, under high collection creation load multiple new sessions are created at once against the same cluster state and placement decisions do not take into account most other placement decisions. Moreover, existing code could not cache more than one session, so most sessions were used for a single placement decision. # Solution Multiple sessions can be cached and reused. Although not optimal (parallel sessions do not see the changes done by each other) this is still an improvement over existing implementation because more context is used for placement (sessions are reused). Also, less sessions are created because all created sessions are candidates for reuse (not only a single one). # Tests Tests in class TestPolicy were run and adapted to some method signature changes. A new test testMultiSessionsCache explicitly verifying multiple sessions can be cached was added. # Checklist Please review the following and check all that apply: - [X] I have reviewed the guidelines for [How to Contribute](https://wiki.apache.org/solr/HowToContribute) and my code conforms to the standards described there to the best of my ability. - [X] I have created a Jira issue and added the issue ID to my pull request title. - [X] I have given Solr maintainers [access] (https://help.github.com/en/articles/allowing-changes-to-a-pull-request-branch-created-from-a-fork) to contribute to my PR branch. (optional but recommended) - [X] I have developed this patch against the `master` branch. - [X] I have run `ant precommit` and the appropriate test suite. - [X] I have added tests for my changes. - [ ] I have added documentation for the [Ref Guide](https://github.com/apache/lucene-solr/tree/master/solr/solr-ref-guide) (for Solr changes only).

github_pulls_comments:

1. **body:** Gradle precommit (that I did not run, maybe should be added to the PR checklist?) reports 2 violations: > cause: 'hasParens true prevLineNotIf true pernicketyLevel true' Suspicious logging call, Parameterize and possibly surround with 'if (log.is*Enabled) {..}'. Help at: 'gradlew helpValidateLogCalls' > /home/runner/work/lucene-solr/lucene-solr/solr/solrj/src/java/org/apache/solr/client/solrj/cloud/autoscaling/PolicyHelper.java:433, This one is wrongly detected, there is a surrounding "if (log.isDebugEnabled())" around the log, but the if bloc also gets the TimeSource. > cause: 'hasPlus: true' Suspicious logging call, Parameterize and possibly surround with 'if (log.is*Enabled) {..}'. Help at: 'gradlew helpValidateLogCalls' > /home/runner/work/lucene-solr/lucene-solr/solr/solrj/src/java/org/apache/solr/client/solrj/cloud/autoscaling/PolicyHelper.java:555).

Likely because the log is written as: `log.debug("New session created, " + sessionWrapper.getCreateTime());` and not: `log.debug("New session created, {}", sessionWrapper.getCreateTime());` The direct concatenation is more efficient in Java. If Solr coding standards impose the "format" style for logs, I'll change it (format style make sense when there's no surrounding "if" for debug level, as they're cheaper to not execute than the concatenation that would have to be executed even if the log is not output).
label: code-design
2. @noblepaul will you be able to have a look at this PR?
3. @murblanc if you are done with the changes you planned to do , I shall do a review and merge this soon
4. > @murblanc if you are done with the changes you planned to do , I shall do a review and merge this soon I am done @noblepaul, so please go ahead.
5. @noblepaul this PR seems to have fallen through the cracks... I'm looking at other aspects of Autoscaling issues and this being merged would make my life easier.
6. @noblepaul if no objections, I'll merge that soon (after rebasing, precommitting and checking all is ok).

github_pulls_reviews:

1. Minor: "requirees" -> requires
2. Assuming the thread could be waiting on the lockObj for a while, it might make more sense to leave at the beginning of the critical section?
3. **body:** Nit: rename to "hasViableSession" or something since it also requires zkVersion check
label: code-design
4. Do we know how many sessions we might have at any given time? Could this be expensive? I suppose it guarantees savings of 1-10 seconds in cases where allowWait is true and there's nothing worth waiting for; just wondering what average case scenario is
5. **body:** Wrote up a *slightly* different implementation here: https://gist.github.com/megancarey/ae2bad53d320ef660ef45c8b003901e1 No more redundant code but I suppose worse in terms of memory, since it makes a recursive call
label: code-design
6. Thanks. I have the MacBook Pro butterfly keyboard, it's a catastrophe!
7. **body:** From a timing perspective you're right. I was trying to minimize non necessary activity done while holding the lock. I'll put the log back inside the synchronized block since the logging delay is negligible compared to the wait for a session to become free.
label: code-design
8. hasCandidateSession
9. Under low load, there should be 0 or 1 sessions. More than that implies a lot of concurrent commands and the iteration here is negligible. A Session is released as soon as it is possible to do so, when all commands that have been using it have completed.
10. **body:** Thanks. I feel it makes the flow a bit harder to read and the savings are not huge so I prefer to stick to the original structure of this method. (the memory impact is negligible IMO. There's also an additional call to hasNonExpiredSession in the proposal but again no big deal)
label: code-design
11. Minor: "yeet" ðŸ˜„
12. **body:** Minor: over-indented?
label: code-design
13. Minor: put allowWait at beginning of check to short-circuit if allowWait is false
14. I think we want to put lines 543-552 in the if statement, since we probably don't want to check for an available session twice in immediate succession if allowWait is false/there are no candidate sessions to wait for.
15. Might want to add a test for creationsInProgress? But would probably require refactoring/exposing creationsInProgress
16. Put it in purpose at the end: it's only true in tests
17. Look ok to me
18. Think different
19. Yes. Not sure it's worth it. We'll see what others think.

jira_issues:

1. **summary:** Autoscaling placement wrong with concurrent collection creations

description: Under concurrent collection creation, wrong Autoscaling placement decisions can lead to severely unbalanced clusters. Sequential creation of the same collections is handled correctly and the cluster is balanced. *TL;DR;* under high load, the way sessions that cache future changes to Zookeeper are managed cause placement decisions of multiple concurrent Collection API calls to ignore each other, be based on identical â€œinitialâ€ cluster state, possibly leading to identical placement decisions and as a consequence cluster imbalance. *Some context first* for those less familiar with how Autoscaling deals with cluster state change: a PolicyHelper.Session is created with a snapshot of the Zookeeper cluster state and is used to track already decided but not yet persisted to Zookeeper cluster state changes so that Collection API commands can make the right placement decisions. A Collection API command either uses an existing cached Session (that includes changes computed by previous command(s)) or creates a new Session initialized from the Zookeeper cluster state (i.e. with only state changes already persisted). When a Collection API command requires a Session - and one is needed for any cluster state update computation - if one exists but is currently in use, the command can wait up to 10 seconds. If the session becomes available, it is reused. Otherwise, a new one is created. The Session lifecycle is as follows: it is created in COMPUTING state by a Collection API command and is initialized with a snapshot of cluster state from Zookeeper (does not require a Zookeeper read, this is running on Overseer that maintains a cache of cluster state). The command has exclusive access to the Session and can change the state of the Session. When the command is done changing the Session, the Session is â€œreturnedâ€ and its state changes to EXECUTING while the command continues to run to persist the state to Zookeeper and interact with the nodes, but no longer interacts with the Session. Another command can then grab a Session in EXECUTING state, change its state to COMPUTING to compute new changes taking into account previous changes. When all commands having used the session have completed their work, the session is â€œreleasedâ€ and destroyed (at this stage, Zookeeper contains all the state changes that were computed using that Session). The issue arises when multiple Collection API commands are executed at once. A first Session is created and commands start using it one by one. In a simple 1 shard 1 replica collection creation test run with 100 parallel Collection API requests (see debug logs from PolicyHelper in file policy.logs), this Session update phase (Session in COMPUTING status in SessionWrapper) takes about 250-300ms (MacBook Pro). This means that about 40 commands can run by using in turn the same Session (45 in the sample run). The commands that have been waiting for too long time out after 10 seconds, more or less all at the same time (at the rate at which they have been received by the OverseerCollectionMessageHandler, approx one per 100ms in the sample run) and most/all independently decide to create a new Session. These new Sessions are based on Zookeeper state, they might or might not include some of the changes from the first 40 commands (depending on if these commands got their changes written to Zookeeper by the time of the 10 seconds timeout, a few might have made it, see below). These new Sessions (54 sessions in addition to the initial one) are based on more or less the same state, so all remaining commands are making placement decisions that do not take into account each other (and likely not much of the first 44 placement decisions either). The sample run whose relevant logs are attached led for the 100 single shard single replica collection creations to 82 collections on the Overseer node, and 5 and 13 collections on the two other nodes of a 3 nodes cluster. Given that the initial session was used 45 times (once initially then reused 44 times), one would have expected at least the first 45 collections to be evenly distributed, i.e. 15 replicas on each node. This was not the case, possibly a sign of other issues (other runs even ended up placing 0 replicas out of the 100 on one of the nodes). From the client perspective, http admin collection CREATE requests averaged 19.5 seconds each and lasted between 7 and 28 seconds (100 parallel threads). This is likely an indication that the last 55 collection creations didnâ€™t see much of the state updates done by the first 45 creations (client delay is longer though than actual Overseer command execution time by http time + Collections API Zookeeper queue time) . *A possible fix* is to not observe any delay before creating a new Session when the currently cached session is busy (i.e. COMPUTING). It will be somewhat less optimal in low load cases (this is likely not an issue, future creations will compensate for slight unbalance and under optimal placement) but will speed up Collection API calls (no waiting) and will prevent multiple waiting commands from all creating new Sessions based on an identical Zookeeper state in cases such as the one described here. For long (minutes and more) autoscaling computations it will likely not make a big difference. If we had more than a single Session being cached (and reused), then less ongoing updates would be lost. Maybe, rather than caching the new updated cluster state after each change, the changes themselves (the deltas) should be tracked. This might allow to propagate changes between sessions or to reconcile cluster state read from Zookeeper with the stream of changes stored in a Session by identifying which deltas made it to Zookeeper, which ones are new from Zookeeper (originating from an update in another session) and which are still pending.

label: code-design

2. **summary:** Autoscaling placement wrong with concurrent collection creations
description: Under concurrent collection creation, wrong Autoscaling placement decisions can lead to severely unbalanced clusters. Sequential creation of the same collections is handled correctly and the cluster is balanced. *TL;DR;* under high load, the way sessions that cache future changes to Zookeeper are managed cause placement decisions of multiple concurrent Collection API calls to ignore each other, be based on identical â€œinitialâ€ cluster state, possibly leading to identical placement decisions and as a consequence cluster imbalance. *Some context first* for those less familiar with how Autoscaling deals with cluster state change: a PolicyHelper.Session is created with a snapshot of the Zookeeper cluster state and is used to track already decided but not yet persisted to Zookeeper cluster state changes so that Collection API commands can make the right placement decisions. A Collection API command either uses an existing cached Session (that includes changes computed by previous command(s)) or creates a new Session initialized from the Zookeeper cluster state (i.e. with only state changes already persisted). When a Collection API command requires a Session - and one is needed for any cluster state update computation - if one exists but is currently in use, the command can wait up to 10 seconds. If the session becomes available, it is reused. Otherwise, a new one is created. The Session lifecycle is as follows: it is created in COMPUTING state by a Collection API command and is initialized with a snapshot of cluster state from Zookeeper (does not require a Zookeeper read, this is running on Overseer that maintains a cache of cluster state). The command has exclusive access to the Session and can change the state of the Session. When the command is done changing the Session, the Session is â€œreturnedâ€ and its state changes to EXECUTING while the command continues to run to persist the state to Zookeeper and interact with the nodes, but no longer interacts with the Session. Another command can then grab a Session in EXECUTING state, change its state to COMPUTING to compute new changes taking into account previous changes. When all commands having used the session have completed their work, the session is â€œreleasedâ€ and destroyed (at this stage, Zookeeper contains all the state changes that were computed using that Session). The issue arises when multiple Collection API commands are executed at once. A first Session is created and commands start using it one by one. In a simple 1 shard 1 replica collection creation test run with 100 parallel Collection API requests (see debug logs from PolicyHelper in file policy.logs), this Session update phase (Session in COMPUTING status in SessionWrapper) takes about 250-300ms (MacBook Pro). This means that about 40 commands can run by using in turn the same Session (45 in the sample run). The commands that have been waiting for too long time out after 10 seconds, more or less all at the same time (at the rate at which they have been received by the OverseerCollectionMessageHandler, approx one per 100ms in the sample run) and most/all independently decide to create a new Session. These new Sessions are based on Zookeeper state, they might or might not include some of the changes from the first 40 commands (depending on if these commands got their changes written to Zookeeper by the time of the 10 seconds timeout, a few might have made it, see below). These new Sessions (54 sessions in addition to the initial one) are based on more or less the same state, so all remaining commands are making placement decisions that do not take into account each other (and likely not much of the first 44 placement decisions either). The sample run whose relevant logs are attached led for the 100 single shard single replica collection creations to 82 collections on the Overseer node, and 5 and 13 collections on the two other nodes of a 3 nodes cluster. Given that the initial session was used 45 times (once initially then reused 44 times), one would have expected at least the first 45 collections to be evenly distributed, i.e. 15 replicas on each node. This was not the case, possibly a sign of other issues (other runs even ended up placing 0 replicas out of the 100 on one of the nodes). From the client perspective, http admin collection CREATE requests averaged 19.5 seconds each and lasted between 7 and 28 seconds (100 parallel threads). This is likely an indication that the last 55 collection creations didnâ€™t see much of the state updates done by the first 45 creations (client delay is longer though than actual Overseer command execution time by http time + Collections API Zookeeper queue time) . *A possible fix* is to not observe any delay before creating a new Session when the currently cached session is busy (i.e. COMPUTING). It will be somewhat less optimal in low load cases (this is likely not an issue, future creations will compensate for slight unbalance and under optimal placement) but will speed up Collection API calls (no waiting) and will prevent multiple waiting commands from all creating new Sessions based on an identical Zookeeper state in cases such as the one described here. For long (minutes and more) autoscaling computations it will likely not make a big difference. If we had more than a single Session being cached (and reused), then less ongoing updates would be lost. Maybe, rather than caching the new updated cluster state after each change, the changes themselves (the deltas) should be tracked. This might allow to propagate changes between sessions or to reconcile cluster state read from Zookeeper with the stream of changes stored in a Session by identifying

which deltas made it to Zookeeper, which ones are new from Zookeeper (originating from an update in another session) and which are still pending.
label: code-design

3. **summary:** Autoscaling placement wrong with concurrent collection creations
description: Under concurrent collection creation, wrong Autoscaling placement decisions can lead to severely unbalanced clusters. Sequential creation of the same collections is handled correctly and the cluster is balanced. *TL;DR;* under high load, the way sessions that cache future changes to Zookeeper are managed cause placement decisions of multiple concurrent Collection API calls to ignore each other, be based on identical â€œinitialâ€ cluster state, possibly leading to identical placement decisions and as a consequence cluster imbalance. *Some context first* for those less familiar with how Autoscaling deals with cluster state change: a PolicyHelper.Session is created with a snapshot of the Zookeeper cluster state and is used to track already decided but not yet persisted to Zookeeper cluster state changes so that Collection API commands can make the right placement decisions. A Collection API command either uses an existing cached Session (that includes changes computed by previous command(s)) or creates a new Session initialized from the Zookeeper cluster state (i.e. with only state changes already persisted). When a Collection API command requires a Session - and one is needed for any cluster state update computation - if one exists but is currently in use, the command can wait up to 10 seconds. If the session becomes available, it is reused. Otherwise, a new one is created. The Session lifecycle is as follows: it is created in COMPUTING state by a Collection API command and is initialized with a snapshot of cluster state from Zookeeper (does not require a Zookeeper read, this is running on Overseer that maintains a cache of cluster state). The command has exclusive access to the Session and can change the state of the Session. When the command is done changing the Session, the Session is â€œreturnedâ€ and its state changes to EXECUTING while the command continues to run to persist the state to Zookeeper and interact with the nodes, but no longer interacts with the Session. Another command can then grab a Session in EXECUTING state, change its state to COMPUTING to compute new changes taking into account previous changes. When all commands having used the session have completed their work, the session is â€œreleasedâ€ and destroyed (at this stage, Zookeeper contains all the state changes that were computed using that Session). The issue arises when multiple Collection API commands are executed at once. A first Session is created and commands start using it one by one. In a simple 1 shard 1 replica collection creation test run with 100 parallel Collection API requests (see debug logs from PolicyHelper in file policy.logs), this Session update phase (Session in COMPUTING status in SessionWrapper) takes about 250-300ms (MacBook Pro). This means that about 40 commands can run by using in turn the same Session (45 in the sample run). The commands that have been waiting for too long time out after 10 seconds, more or less all at the same time (at the rate at which they have been received by the OverseerCollectionMessageHandler, approx one per 100ms in the sample run) and most/all independently decide to create a new Session. These new Sessions are based on Zookeeper state, they might or might not include some of the changes from the first 40 commands (depending on if these commands got their changes written to Zookeeper by the time of the 10 seconds timeout, a few might have made it, see below). These new Sessions (54 sessions in addition to the initial one) are based on more or less the same state, so all remaining commands are making placement decisions that do not take into account each other (and likely not much of the first 44 placement decisions either). The sample run whose relevant logs are attached led for the 100 single shard single replica collection creations to 82 collections on the Overseer node, and 5 and 13 collections on the two other nodes of a 3 nodes cluster. Given that the initial session was used 45 times (once initially then reused 44 times), one would have expected at least the first 45 collections to be evenly distributed, i.e. 15 replicas on each node. This was not the case, possibly a sign of other issues (other runs even ended up placing 0 replicas out of the 100 on one of the nodes). From the client perspective, http admin collection CREATE requests averaged 19.5 seconds each and lasted between 7 and 28 seconds (100 parallel threads). This is likely an indication that the last 55 collection creations didnâ€™t see much of the state updates done by the first 45 creations (client delay is longer though than actual Overseer command execution time by http time + Collections API Zookeeper queue time) . *A possible fix* is to not observe any delay before creating a new Session when the currently cached session is busy (i.e. COMPUTING). It will be somewhat less optimal in low load cases (this is likely not an issue, future creations will compensate for slight unbalance and under optimal placement) but will speed up Collection API calls (no waiting) and will prevent multiple waiting commands from all creating new Sessions based on an identical Zookeeper state in cases such as the one described here. For long (minutes and more) autoscaling computations it will likely not make a big difference. If we had more than a single Session being cached (and reused), then less ongoing updates would be lost. Maybe, rather than caching the new updated cluster state after each change, the changes

themselves (the deltas) should be tracked. This might allow to propagate changes between sessions or to reconcile cluster state read from Zookeeper with the stream of changes stored in a Session by identifying which deltas made it to Zookeeper, which ones are new from Zookeeper (originating from an update in another session) and which are still pending.

4. **summary:** Autoscaling placement wrong with concurrent collection creations
   description: Under concurrent collection creation, wrong Autoscaling placement decisions can lead to severely unbalanced clusters. Sequential creation of the same collections is handled correctly and the cluster is balanced. *TL;DR;* under high load, the way sessions that cache future changes to Zookeeper are managed cause placement decisions of multiple concurrent Collection API calls to ignore each other, be based on identical â€œinitialâ€ cluster state, possibly leading to identical placement decisions and as a consequence cluster imbalance. *Some context first* for those less familiar with how Autoscaling deals with cluster state change: a PolicyHelper.Session is created with a snapshot of the Zookeeper cluster state and is used to track already decided but not yet persisted to Zookeeper cluster state changes so that Collection API commands can make the right placement decisions. A Collection API command either uses an existing cached Session (that includes changes computed by previous command(s)) or creates a new Session initialized from the Zookeeper cluster state (i.e. with only state changes already persisted). When a Collection API command requires a Session - and one is needed for any cluster state update computation - if one exists but is currently in use, the command can wait up to 10 seconds. If the session becomes available, it is reused. Otherwise, a new one is created. The Session lifecycle is as follows: it is created in COMPUTING state by a Collection API command and is initialized with a snapshot of cluster state from Zookeeper (does not require a Zookeeper read, this is running on Overseer that maintains a cache of cluster state). The command has exclusive access to the Session and can change the state of the Session. When the command is done changing the Session, the Session is â€œreturnedâ€ and its state changes to EXECUTING while the command continues to run to persist the state to Zookeeper and interact with the nodes, but no longer interacts with the Session. Another command can then grab a Session in EXECUTING state, change its state to COMPUTING to compute new changes taking into account previous changes. When all commands having used the session have completed their work, the session is â€œreleasedâ€ and destroyed (at this stage, Zookeeper contains all the state changes that were computed using that Session). The issue arises when multiple Collection API commands are executed at once. A first Session is created and commands start using it one by one. In a simple 1 shard 1 replica collection creation test run with 100 parallel Collection API requests (see debug logs from PolicyHelper in file policy.logs), this Session update phase (Session in COMPUTING status in SessionWrapper) takes about 250-300ms (MacBook Pro). This means that about 40 commands can run by using in turn the same Session (45 in the sample run). The commands that have been waiting for too long time out after 10 seconds, more or less all at the same time (at the rate at which they have been received by the OverseerCollectionMessageHandler, approx one per 100ms in the sample run) and most/all independently decide to create a new Session. These new Sessions are based on Zookeeper state, they might or might not include some of the changes from the first 40 commands (depending on if these commands got their changes written to Zookeeper by the time of the 10 seconds timeout, a few might have made it, see below). These new Sessions (54 sessions in addition to the initial one) are based on more or less the same state, so all remaining commands are making placement decisions that do not take into account each other (and likely not much of the first 44 placement decisions either). The sample run whose relevant logs are attached led for the 100 single shard single replica collection creations to 82 collections on the Overseer node, and 5 and 13 collections on the two other nodes of a 3 nodes cluster. Given that the initial session was used 45 times (once initially then reused 44 times), one would have expected at least the first 45 collections to be evenly distributed, i.e. 15 replicas on each node. This was not the case, possibly a sign of other issues (other runs even ended up placing 0 replicas out of the 100 on one of the nodes). From the client perspective, http admin collection CREATE requests averaged 19.5 seconds each and lasted between 7 and 28 seconds (100 parallel threads). This is likely an indication that the last 55 collection creations didnâ€™t see much of the state updates done by the first 45 creations (client delay is longer though than actual Overseer command execution time by http time + Collections API Zookeeper queue time) . *A possible fix* is to not observe any delay before creating a new Session when the currently cached session is busy (i.e. COMPUTING). It will be somewhat less optimal in low load cases (this is likely not an issue, future creations will compensate for slight unbalance and under optimal placement) but will speed up Collection API calls (no waiting) and will prevent multiple waiting commands from all creating new Sessions based on an identical Zookeeper state in cases such as the one described here. For long (minutes and more) autoscaling computations it will likely not make a big difference. If we had more than a single Session being cached (and reused), then less ongoing updates

would be lost. Maybe, rather than caching the new updated cluster state after each change, the changes themselves (the deltas) should be tracked. This might allow to propagate changes between sessions or to reconcile cluster state read from Zookeeper with the stream of changes stored in a Session by identifying which deltas made it to Zookeeper, which ones are new from Zookeeper (originating from an update in another session) and which are still pending.
label: code-design

5. **summary:** Autoscaling placement wrong with concurrent collection creations
description: Under concurrent collection creation, wrong Autoscaling placement decisions can lead to severely unbalanced clusters. Sequential creation of the same collections is handled correctly and the cluster is balanced. *TL;DR;* under high load, the way sessions that cache future changes to Zookeeper are managed cause placement decisions of multiple concurrent Collection API calls to ignore each other, be based on identical â€œinitialâ€ cluster state, possibly leading to identical placement decisions and as a consequence cluster imbalance. *Some context first* for those less familiar with how Autoscaling deals with cluster state change: a PolicyHelper.Session is created with a snapshot of the Zookeeper cluster state and is used to track already decided but not yet persisted to Zookeeper cluster state changes so that Collection API commands can make the right placement decisions. A Collection API command either uses an existing cached Session (that includes changes computed by previous command(s)) or creates a new Session initialized from the Zookeeper cluster state (i.e. with only state changes already persisted). When a Collection API command requires a Session - and one is needed for any cluster state update computation - if one exists but is currently in use, the command can wait up to 10 seconds. If the session becomes available, it is reused. Otherwise, a new one is created. The Session lifecycle is as follows: it is created in COMPUTING state by a Collection API command and is initialized with a snapshot of cluster state from Zookeeper (does not require a Zookeeper read, this is running on Overseer that maintains a cache of cluster state). The command has exclusive access to the Session and can change the state of the Session. When the command is done changing the Session, the Session is â€œreturnedâ€ and its state changes to EXECUTING while the command continues to run to persist the state to Zookeeper and interact with the nodes, but no longer interacts with the Session. Another command can then grab a Session in EXECUTING state, change its state to COMPUTING to compute new changes taking into account previous changes. When all commands having used the session have completed their work, the session is â€œreleasedâ€ and destroyed (at this stage, Zookeeper contains all the state changes that were computed using that Session). The issue arises when multiple Collection API commands are executed at once. A first Session is created and commands start using it one by one. In a simple 1 shard 1 replica collection creation test run with 100 parallel Collection API requests (see debug logs from PolicyHelper in file policy.logs), this Session update phase (Session in COMPUTING status in SessionWrapper) takes about 250-300ms (MacBook Pro). This means that about 40 commands can run by using in turn the same Session (45 in the sample run). The commands that have been waiting for too long time out after 10 seconds, more or less all at the same time (at the rate at which they have been received by the OverseerCollectionMessageHandler, approx one per 100ms in the sample run) and most/all independently decide to create a new Session. These new Sessions are based on Zookeeper state, they might or might not include some of the changes from the first 40 commands (depending on if these commands got their changes written to Zookeeper by the time of the 10 seconds timeout, a few might have made it, see below). These new Sessions (54 sessions in addition to the initial one) are based on more or less the same state, so all remaining commands are making placement decisions that do not take into account each other (and likely not much of the first 44 placement decisions either). The sample run whose relevant logs are attached led for the 100 single shard single replica collection creations to 82 collections on the Overseer node, and 5 and 13 collections on the two other nodes of a 3 nodes cluster. Given that the initial session was used 45 times (once initially then reused 44 times), one would have expected at least the first 45 collections to be evenly distributed, i.e. 15 replicas on each node. This was not the case, possibly a sign of other issues (other runs even ended up placing 0 replicas out of the 100 on one of the nodes). From the client perspective, http admin collection CREATE requests averaged 19.5 seconds each and lasted between 7 and 28 seconds (100 parallel threads). This is likely an indication that the last 55 collection creations didnâ€™t see much of the state updates done by the first 45 creations (client delay is longer though than actual Overseer command execution time by http time + Collections API Zookeeper queue time) . *A possible fix* is to not observe any delay before creating a new Session when the currently cached session is busy (i.e. COMPUTING). It will be somewhat less optimal in low load cases (this is likely not an issue, future creations will compensate for slight unbalance and under optimal placement) but will speed up Collection API calls (no waiting) and will prevent multiple waiting commands from all creating new Sessions based on an identical Zookeeper state in cases such as the one

described here. For long (minutes and more) autoscaling computations it will likely not make a big difference. If we had more than a single Session being cached (and reused), then less ongoing updates would be lost. Maybe, rather than caching the new updated cluster state after each change, the changes themselves (the deltas) should be tracked. This might allow to propagate changes between sessions or to reconcile cluster state read from Zookeeper with the stream of changes stored in a Session by identifying which deltas made it to Zookeeper, which ones are new from Zookeeper (originating from an update in another session) and which are still pending.

6. **summary:** Autoscaling placement wrong with concurrent collection creations
description: Under concurrent collection creation, wrong Autoscaling placement decisions can lead to severely unbalanced clusters. Sequential creation of the same collections is handled correctly and the cluster is balanced. *TL;DR;* under high load, the way sessions that cache future changes to Zookeeper are managed cause placement decisions of multiple concurrent Collection API calls to ignore each other, be based on identical â€œinitialâ€ cluster state, possibly leading to identical placement decisions and as a consequence cluster imbalance. *Some context first* for those less familiar with how Autoscaling deals with cluster state change: a PolicyHelper.Session is created with a snapshot of the Zookeeper cluster state and is used to track already decided but not yet persisted to Zookeeper cluster state changes so that Collection API commands can make the right placement decisions. A Collection API command either uses an existing cached Session (that includes changes computed by previous command(s)) or creates a new Session initialized from the Zookeeper cluster state (i.e. with only state changes already persisted). When a Collection API command requires a Session - and one is needed for any cluster state update computation - if one exists but is currently in use, the command can wait up to 10 seconds. If the session becomes available, it is reused. Otherwise, a new one is created. The Session lifecycle is as follows: it is created in COMPUTING state by a Collection API command and is initialized with a snapshot of cluster state from Zookeeper (does not require a Zookeeper read, this is running on Overseer that maintains a cache of cluster state). The command has exclusive access to the Session and can change the state of the Session. When the command is done changing the Session, the Session is â€œreturnedâ€ and its state changes to EXECUTING while the command continues to run to persist the state to Zookeeper and interact with the nodes, but no longer interacts with the Session. Another command can then grab a Session in EXECUTING state, change its state to COMPUTING to compute new changes taking into account previous changes. When all commands having used the session have completed their work, the session is â€œreleasedâ€ and destroyed (at this stage, Zookeeper contains all the state changes that were computed using that Session). The issue arises when multiple Collection API commands are executed at once. A first Session is created and commands start using it one by one. In a simple 1 shard 1 replica collection creation test run with 100 parallel Collection API requests (see debug logs from PolicyHelper in file policy.logs), this Session update phase (Session in COMPUTING status in SessionWrapper) takes about 250-300ms (MacBook Pro). This means that about 40 commands can run by using in turn the same Session (45 in the sample run). The commands that have been waiting for too long time out after 10 seconds, more or less all at the same time (at the rate at which they have been received by the OverseerCollectionMessageHandler, approx one per 100ms in the sample run) and most/all independently decide to create a new Session. These new Sessions are based on Zookeeper state, they might or might not include some of the changes from the first 40 commands (depending on if these commands got their changes written to Zookeeper by the time of the 10 seconds timeout, a few might have made it, see below). These new Sessions (54 sessions in addition to the initial one) are based on more or less the same state, so all remaining commands are making placement decisions that do not take into account each other (and likely not much of the first 44 placement decisions either). The sample run whose relevant logs are attached led for the 100 single shard single replica collection creations to 82 collections on the Overseer node, and 5 and 13 collections on the two other nodes of a 3 nodes cluster. Given that the initial session was used 45 times (once initially then reused 44 times), one would have expected at least the first 45 collections to be evenly distributed, i.e. 15 replicas on each node. This was not the case, possibly a sign of other issues (other runs even ended up placing 0 replicas out of the 100 on one of the nodes). From the client perspective, http admin collection CREATE requests averaged 19.5 seconds each and lasted between 7 and 28 seconds (100 parallel threads). This is likely an indication that the last 55 collection creations didnâ€™t see much of the state updates done by the first 45 creations (client delay is longer though than actual Overseer command execution time by http time + Collections API Zookeeper queue time) . *A possible fix* is to not observe any delay before creating a new Session when the currently cached session is busy (i.e. COMPUTING). It will be somewhat less optimal in low load cases (this is likely not an issue, future creations will compensate for slight unbalance and under optimal placement) but will speed up Collection API calls (no waiting) and will prevent multiple waiting

commands from all creating new Sessions based on an identical Zookeeper state in cases such as the one described here. For long (minutes and more) autoscaling computations it will likely not make a big difference. If we had more than a single Session being cached (and reused), then less ongoing updates would be lost. Maybe, rather than caching the new updated cluster state after each change, the changes themselves (the deltas) should be tracked. This might allow to propagate changes between sessions or to reconcile cluster state read from Zookeeper with the stream of changes stored in a Session by identifying which deltas made it to Zookeeper, which ones are new from Zookeeper (originating from an update in another session) and which are still pending.

7. **summary:** Autoscaling placement wrong with concurrent collection creations
description: Under concurrent collection creation, wrong Autoscaling placement decisions can lead to severely unbalanced clusters. Sequential creation of the same collections is handled correctly and the cluster is balanced. *TL;DR;* under high load, the way sessions that cache future changes to Zookeeper are managed cause placement decisions of multiple concurrent Collection API calls to ignore each other, be based on identical â€œinitialâ€ cluster state, possibly leading to identical placement decisions and as a consequence cluster imbalance. *Some context first* for those less familiar with how Autoscaling deals with cluster state change: a PolicyHelper.Session is created with a snapshot of the Zookeeper cluster state and is used to track already decided but not yet persisted to Zookeeper cluster state changes so that Collection API commands can make the right placement decisions. A Collection API command either uses an existing cached Session (that includes changes computed by previous command(s)) or creates a new Session initialized from the Zookeeper cluster state (i.e. with only state changes already persisted). When a Collection API command requires a Session - and one is needed for any cluster state update computation - if one exists but is currently in use, the command can wait up to 10 seconds. If the session becomes available, it is reused. Otherwise, a new one is created. The Session lifecycle is as follows: it is created in COMPUTING state by a Collection API command and is initialized with a snapshot of cluster state from Zookeeper (does not require a Zookeeper read, this is running on Overseer that maintains a cache of cluster state). The command has exclusive access to the Session and can change the state of the Session. When the command is done changing the Session, the Session is â€œreturnedâ€ and its state changes to EXECUTING while the command continues to run to persist the state to Zookeeper and interact with the nodes, but no longer interacts with the Session. Another command can then grab a Session in EXECUTING state, change its state to COMPUTING to compute new changes taking into account previous changes. When all commands having used the session have completed their work, the session is â€œreleasedâ€ and destroyed (at this stage, Zookeeper contains all the state changes that were computed using that Session). The issue arises when multiple Collection API commands are executed at once. A first Session is created and commands start using it one by one. In a simple 1 shard 1 replica collection creation test run with 100 parallel Collection API requests (see debug logs from PolicyHelper in file policy.logs), this Session update phase (Session in COMPUTING status in SessionWrapper) takes about 250-300ms (MacBook Pro). This means that about 40 commands can run by using in turn the same Session (45 in the sample run). The commands that have been waiting for too long time out after 10 seconds, more or less all at the same time (at the rate at which they have been received by the OverseerCollectionMessageHandler, approx one per 100ms in the sample run) and most/all independently decide to create a new Session. These new Sessions are based on Zookeeper state, they might or might not include some of the changes from the first 40 commands (depending on if these commands got their changes written to Zookeeper by the time of the 10 seconds timeout, a few might have made it, see below). These new Sessions (54 sessions in addition to the initial one) are based on more or less the same state, so all remaining commands are making placement decisions that do not take into account each other (and likely not much of the first 44 placement decisions either). The sample run whose relevant logs are attached led for the 100 single shard single replica collection creations to 82 collections on the Overseer node, and 5 and 13 collections on the two other nodes of a 3 nodes cluster. Given that the initial session was used 45 times (once initially then reused 44 times), one would have expected at least the first 45 collections to be evenly distributed, i.e. 15 replicas on each node. This was not the case, possibly a sign of other issues (other runs even ended up placing 0 replicas out of the 100 on one of the nodes). From the client perspective, http admin collection CREATE requests averaged 19.5 seconds each and lasted between 7 and 28 seconds (100 parallel threads). This is likely an indication that the last 55 collection creations didnâ€™t see much of the state updates done by the first 45 creations (client delay is longer though than actual Overseer command execution time by http time + Collections API Zookeeper queue time) . *A possible fix* is to not observe any delay before creating a new Session when the currently cached session is busy (i.e. COMPUTING). It will be somewhat less optimal in low load cases (this is likely not an issue, future creations will compensate for slight unbalance and under

optimal placement) but will speed up Collection API calls (no waiting) and will prevent multiple waiting commands from all creating new Sessions based on an identical Zookeeper state in cases such as the one described here. For long (minutes and more) autoscaling computations it will likely not make a big difference. If we had more than a single Session being cached (and reused), then less ongoing updates would be lost. Maybe, rather than caching the new updated cluster state after each change, the changes themselves (the deltas) should be tracked. This might allow to propagate changes between sessions or to reconcile cluster state read from Zookeeper with the stream of changes stored in a Session by identifying which deltas made it to Zookeeper, which ones are new from Zookeeper (originating from an update in another session) and which are still pending.

8. **summary:** Autoscaling placement wrong with concurrent collection creations
description: Under concurrent collection creation, wrong Autoscaling placement decisions can lead to severely unbalanced clusters. Sequential creation of the same collections is handled correctly and the cluster is balanced. *TL;DR;* under high load, the way sessions that cache future changes to Zookeeper are managed cause placement decisions of multiple concurrent Collection API calls to ignore each other, be based on identical â€œinitialâ€ cluster state, possibly leading to identical placement decisions and as a consequence cluster imbalance. *Some context first* for those less familiar with how Autoscaling deals with cluster state change: a PolicyHelper.Session is created with a snapshot of the Zookeeper cluster state and is used to track already decided but not yet persisted to Zookeeper cluster state changes so that Collection API commands can make the right placement decisions. A Collection API command either uses an existing cached Session (that includes changes computed by previous command(s)) or creates a new Session initialized from the Zookeeper cluster state (i.e. with only state changes already persisted). When a Collection API command requires a Session - and one is needed for any cluster state update computation - if one exists but is currently in use, the command can wait up to 10 seconds. If the session becomes available, it is reused. Otherwise, a new one is created. The Session lifecycle is as follows: it is created in COMPUTING state by a Collection API command and is initialized with a snapshot of cluster state from Zookeeper (does not require a Zookeeper read, this is running on Overseer that maintains a cache of cluster state). The command has exclusive access to the Session and can change the state of the Session. When the command is done changing the Session, the Session is â€œreturnedâ€ and its state changes to EXECUTING while the command continues to run to persist the state to Zookeeper and interact with the nodes, but no longer interacts with the Session. Another command can then grab a Session in EXECUTING state, change its state to COMPUTING to compute new changes taking into account previous changes. When all commands having used the session have completed their work, the session is â€œreleasedâ€ and destroyed (at this stage, Zookeeper contains all the state changes that were computed using that Session). The issue arises when multiple Collection API commands are executed at once. A first Session is created and commands start using it one by one. In a simple 1 shard 1 replica collection creation test run with 100 parallel Collection API requests (see debug logs from PolicyHelper in file policy.logs), this Session update phase (Session in COMPUTING status in SessionWrapper) takes about 250-300ms (MacBook Pro). This means that about 40 commands can run by using in turn the same Session (45 in the sample run). The commands that have been waiting for too long time out after 10 seconds, more or less all at the same time (at the rate at which they have been received by the OverseerCollectionMessageHandler, approx one per 100ms in the sample run) and most/all independently decide to create a new Session. These new Sessions are based on Zookeeper state, they might or might not include some of the changes from the first 40 commands (depending on if these commands got their changes written to Zookeeper by the time of the 10 seconds timeout, a few might have made it, see below). These new Sessions (54 sessions in addition to the initial one) are based on more or less the same state, so all remaining commands are making placement decisions that do not take into account each other (and likely not much of the first 44 placement decisions either). The sample run whose relevant logs are attached led for the 100 single shard single replica collection creations to 82 collections on the Overseer node, and 5 and 13 collections on the two other nodes of a 3 nodes cluster. Given that the initial session was used 45 times (once initially then reused 44 times), one would have expected at least the first 45 collections to be evenly distributed, i.e. 15 replicas on each node. This was not the case, possibly a sign of other issues (other runs even ended up placing 0 replicas out of the 100 on one of the nodes). From the client perspective, http admin collection CREATE requests averaged 19.5 seconds each and lasted between 7 and 28 seconds (100 parallel threads). This is likely an indication that the last 55 collection creations didnâ€™t see much of the state updates done by the first 45 creations (client delay is longer though than actual Overseer command execution time by http time + Collections API Zookeeper queue time) . *A possible fix* is to not observe any delay before creating a new Session when the currently cached session is busy (i.e. COMPUTING). It will be somewhat less optimal in low

load cases (this is likely not an issue, future creations will compensate for slight unbalance and under optimal placement) but will speed up Collection API calls (no waiting) and will prevent multiple waiting commands from all creating new Sessions based on an identical Zookeeper state in cases such as the one described here. For long (minutes and more) autoscaling computations it will likely not make a big difference. If we had more than a single Session being cached (and reused), then less ongoing updates would be lost. Maybe, rather than caching the new updated cluster state after each change, the changes themselves (the deltas) should be tracked. This might allow to propagate changes between sessions or to reconcile cluster state read from Zookeeper with the stream of changes stored in a Session by identifying which deltas made it to Zookeeper, which ones are new from Zookeeper (originating from an update in another session) and which are still pending.

9. **summary:** Autoscaling placement wrong with concurrent collection creations
description: Under concurrent collection creation, wrong Autoscaling placement decisions can lead to severely unbalanced clusters. Sequential creation of the same collections is handled correctly and the cluster is balanced. *TL;DR;* under high load, the way sessions that cache future changes to Zookeeper are managed cause placement decisions of multiple concurrent Collection API calls to ignore each other, be based on identical â€œinitialâ€ cluster state, possibly leading to identical placement decisions and as a consequence cluster imbalance. *Some context first* for those less familiar with how Autoscaling deals with cluster state change: a PolicyHelper.Session is created with a snapshot of the Zookeeper cluster state and is used to track already decided but not yet persisted to Zookeeper cluster state changes so that Collection API commands can make the right placement decisions. A Collection API command either uses an existing cached Session (that includes changes computed by previous command(s)) or creates a new Session initialized from the Zookeeper cluster state (i.e. with only state changes already persisted). When a Collection API command requires a Session - and one is needed for any cluster state update computation - if one exists but is currently in use, the command can wait up to 10 seconds. If the session becomes available, it is reused. Otherwise, a new one is created. The Session lifecycle is as follows: it is created in COMPUTING state by a Collection API command and is initialized with a snapshot of cluster state from Zookeeper (does not require a Zookeeper read, this is running on Overseer that maintains a cache of cluster state). The command has exclusive access to the Session and can change the state of the Session. When the command is done changing the Session, the Session is â€œreturnedâ€ and its state changes to EXECUTING while the command continues to run to persist the state to Zookeeper and interact with the nodes, but no longer interacts with the Session. Another command can then grab a Session in EXECUTING state, change its state to COMPUTING to compute new changes taking into account previous changes. When all commands having used the session have completed their work, the session is â€œreleasedâ€ and destroyed (at this stage, Zookeeper contains all the state changes that were computed using that Session). The issue arises when multiple Collection API commands are executed at once. A first Session is created and commands start using it one by one. In a simple 1 shard 1 replica collection creation test run with 100 parallel Collection API requests (see debug logs from PolicyHelper in file policy.logs), this Session update phase (Session in COMPUTING status in SessionWrapper) takes about 250-300ms (MacBook Pro). This means that about 40 commands can run by using in turn the same Session (45 in the sample run). The commands that have been waiting for too long time out after 10 seconds, more or less all at the same time (at the rate at which they have been received by the OverseerCollectionMessageHandler, approx one per 100ms in the sample run) and most/all independently decide to create a new Session. These new Sessions are based on Zookeeper state, they might or might not include some of the changes from the first 40 commands (depending on if these commands got their changes written to Zookeeper by the time of the 10 seconds timeout, a few might have made it, see below). These new Sessions (54 sessions in addition to the initial one) are based on more or less the same state, so all remaining commands are making placement decisions that do not take into account each other (and likely not much of the first 44 placement decisions either). The sample run whose relevant logs are attached led for the 100 single shard single replica collection creations to 82 collections on the Overseer node, and 5 and 13 collections on the two other nodes of a 3 nodes cluster. Given that the initial session was used 45 times (once initially then reused 44 times), one would have expected at least the first 45 collections to be evenly distributed, i.e. 15 replicas on each node. This was not the case, possibly a sign of other issues (other runs even ended up placing 0 replicas out of the 100 on one of the nodes). From the client perspective, http admin collection CREATE requests averaged 19.5 seconds each and lasted between 7 and 28 seconds (100 parallel threads). This is likely an indication that the last 55 collection creations didnâ€™t see much of the state updates done by the first 45 creations (client delay is longer though than actual Overseer command execution time by http time + Collections API Zookeeper queue time) . *A possible fix* is to not observe any delay before creating a new Session

when the currently cached session is busy (i.e. COMPUTING). It will be somewhat less optimal in low load cases (this is likely not an issue, future creations will compensate for slight unbalance and under optimal placement) but will speed up Collection API calls (no waiting) and will prevent multiple waiting commands from all creating new Sessions based on an identical Zookeeper state in cases such as the one described here. For long (minutes and more) autoscaling computations it will likely not make a big difference. If we had more than a single Session being cached (and reused), then less ongoing updates would be lost. Maybe, rather than caching the new updated cluster state after each change, the changes themselves (the deltas) should be tracked. This might allow to propagate changes between sessions or to reconcile cluster state read from Zookeeper with the stream of changes stored in a Session by identifying which deltas made it to Zookeeper, which ones are new from Zookeeper (originating from an update in another session) and which are still pending.

10. **summary:** Autoscaling placement wrong with concurrent collection creations
description: Under concurrent collection creation, wrong Autoscaling placement decisions can lead to severely unbalanced clusters. Sequential creation of the same collections is handled correctly and the cluster is balanced. *TL;DR;* under high load, the way sessions that cache future changes to Zookeeper are managed cause placement decisions of multiple concurrent Collection API calls to ignore each other, be based on identical â€œinitialâ€ cluster state, possibly leading to identical placement decisions and as a consequence cluster imbalance. *Some context first* for those less familiar with how Autoscaling deals with cluster state change: a PolicyHelper.Session is created with a snapshot of the Zookeeper cluster state and is used to track already decided but not yet persisted to Zookeeper cluster state changes so that Collection API commands can make the right placement decisions. A Collection API command either uses an existing cached Session (that includes changes computed by previous command(s)) or creates a new Session initialized from the Zookeeper cluster state (i.e. with only state changes already persisted). When a Collection API command requires a Session - and one is needed for any cluster state update computation - if one exists but is currently in use, the command can wait up to 10 seconds. If the session becomes available, it is reused. Otherwise, a new one is created. The Session lifecycle is as follows: it is created in COMPUTING state by a Collection API command and is initialized with a snapshot of cluster state from Zookeeper (does not require a Zookeeper read, this is running on Overseer that maintains a cache of cluster state). The command has exclusive access to the Session and can change the state of the Session. When the command is done changing the Session, the Session is â€œreturnedâ€ and its state changes to EXECUTING while the command continues to run to persist the state to Zookeeper and interact with the nodes, but no longer interacts with the Session. Another command can then grab a Session in EXECUTING state, change its state to COMPUTING to compute new changes taking into account previous changes. When all commands having used the session have completed their work, the session is â€œreleasedâ€ and destroyed (at this stage, Zookeeper contains all the state changes that were computed using that Session). The issue arises when multiple Collection API commands are executed at once. A first Session is created and commands start using it one by one. In a simple 1 shard 1 replica collection creation test run with 100 parallel Collection API requests (see debug logs from PolicyHelper in file policy.logs), this Session update phase (Session in COMPUTING status in SessionWrapper) takes about 250-300ms (MacBook Pro). This means that about 40 commands can run by using in turn the same Session (45 in the sample run). The commands that have been waiting for too long time out after 10 seconds, more or less all at the same time (at the rate at which they have been received by the OverseerCollectionMessageHandler, approx one per 100ms in the sample run) and most/all independently decide to create a new Session. These new Sessions are based on Zookeeper state, they might or might not include some of the changes from the first 40 commands (depending on if these commands got their changes written to Zookeeper by the time of the 10 seconds timeout, a few might have made it, see below). These new Sessions (54 sessions in addition to the initial one) are based on more or less the same state, so all remaining commands are making placement decisions that do not take into account each other (and likely not much of the first 44 placement decisions either). The sample run whose relevant logs are attached led for the 100 single shard single replica collection creations to 82 collections on the Overseer node, and 5 and 13 collections on the two other nodes of a 3 nodes cluster. Given that the initial session was used 45 times (once initially then reused 44 times), one would have expected at least the first 45 collections to be evenly distributed, i.e. 15 replicas on each node. This was not the case, possibly a sign of other issues (other runs even ended up placing 0 replicas out of the 100 on one of the nodes). From the client perspective, http admin collection CREATE requests averaged 19.5 seconds each and lasted between 7 and 28 seconds (100 parallel threads). This is likely an indication that the last 55 collection creations didnâ€™t see much of the state updates done by the first 45 creations (client delay is longer though than actual Overseer command execution time by http time + Collections

API Zookeeper queue time) . *A possible fix* is to not observe any delay before creating a new Session when the currently cached session is busy (i.e. COMPUTING). It will be somewhat less optimal in low load cases (this is likely not an issue, future creations will compensate for slight unbalance and under optimal placement) but will speed up Collection API calls (no waiting) and will prevent multiple waiting commands from all creating new Sessions based on an identical Zookeeper state in cases such as the one described here. For long (minutes and more) autoscaling computations it will likely not make a big difference. If we had more than a single Session being cached (and reused), then less ongoing updates would be lost. Maybe, rather than caching the new updated cluster state after each change, the changes themselves (the deltas) should be tracked. This might allow to propagate changes between sessions or to reconcile cluster state read from Zookeeper with the stream of changes stored in a Session by identifying which deltas made it to Zookeeper, which ones are new from Zookeeper (originating from an update in another session) and which are still pending.

11. **summary:** Autoscaling placement wrong with concurrent collection creations
description: Under concurrent collection creation, wrong Autoscaling placement decisions can lead to severely unbalanced clusters. Sequential creation of the same collections is handled correctly and the cluster is balanced. *TL;DR;* under high load, the way sessions that cache future changes to Zookeeper are managed cause placement decisions of multiple concurrent Collection API calls to ignore each other, be based on identical â€œinitialâ€ cluster state, possibly leading to identical placement decisions and as a consequence cluster imbalance. *Some context first* for those less familiar with how Autoscaling deals with cluster state change: a PolicyHelper.Session is created with a snapshot of the Zookeeper cluster state and is used to track already decided but not yet persisted to Zookeeper cluster state changes so that Collection API commands can make the right placement decisions. A Collection API command either uses an existing cached Session (that includes changes computed by previous command(s)) or creates a new Session initialized from the Zookeeper cluster state (i.e. with only state changes already persisted). When a Collection API command requires a Session - and one is needed for any cluster state update computation - if one exists but is currently in use, the command can wait up to 10 seconds. If the session becomes available, it is reused. Otherwise, a new one is created. The Session lifecycle is as follows: it is created in COMPUTING state by a Collection API command and is initialized with a snapshot of cluster state from Zookeeper (does not require a Zookeeper read, this is running on Overseer that maintains a cache of cluster state). The command has exclusive access to the Session and can change the state of the Session. When the command is done changing the Session, the Session is â€œreturnedâ€ and its state changes to EXECUTING while the command continues to run to persist the state to Zookeeper and interact with the nodes, but no longer interacts with the Session. Another command can then grab a Session in EXECUTING state, change its state to COMPUTING to compute new changes taking into account previous changes. When all commands having used the session have completed their work, the session is â€œreleasedâ€ and destroyed (at this stage, Zookeeper contains all the state changes that were computed using that Session). The issue arises when multiple Collection API commands are executed at once. A first Session is created and commands start using it one by one. In a simple 1 shard 1 replica collection creation test run with 100 parallel Collection API requests (see debug logs from PolicyHelper in file policy.logs), this Session update phase (Session in COMPUTING status in SessionWrapper) takes about 250-300ms (MacBook Pro). This means that about 40 commands can run by using in turn the same Session (45 in the sample run). The commands that have been waiting for too long time out after 10 seconds, more or less all at the same time (at the rate at which they have been received by the OverseerCollectionMessageHandler, approx one per 100ms in the sample run) and most/all independently decide to create a new Session. These new Sessions are based on Zookeeper state, they might or might not include some of the changes from the first 40 commands (depending on if these commands got their changes written to Zookeeper by the time of the 10 seconds timeout, a few might have made it, see below). These new Sessions (54 sessions in addition to the initial one) are based on more or less the same state, so all remaining commands are making placement decisions that do not take into account each other (and likely not much of the first 44 placement decisions either). The sample run whose relevant logs are attached led for the 100 single shard single replica collection creations to 82 collections on the Overseer node, and 5 and 13 collections on the two other nodes of a 3 nodes cluster. Given that the initial session was used 45 times (once initially then reused 44 times), one would have expected at least the first 45 collections to be evenly distributed, i.e. 15 replicas on each node. This was not the case, possibly a sign of other issues (other runs even ended up placing 0 replicas out of the 100 on one of the nodes). From the client perspective, http admin collection CREATE requests averaged 19.5 seconds each and lasted between 7 and 28 seconds (100 parallel threads). This is likely an indication that the last 55 collection creations didnâ€™t see much of the state updates done by the first 45 creations

(client delay is longer though than actual Overseer command execution time by http time + Collections API Zookeeper queue time) . *A possible fix* is to not observe any delay before creating a new Session when the currently cached session is busy (i.e. COMPUTING). It will be somewhat less optimal in low load cases (this is likely not an issue, future creations will compensate for slight unbalance and under optimal placement) but will speed up Collection API calls (no waiting) and will prevent multiple waiting commands from all creating new Sessions based on an identical Zookeeper state in cases such as the one described here. For long (minutes and more) autoscaling computations it will likely not make a big difference. If we had more than a single Session being cached (and reused), then less ongoing updates would be lost. Maybe, rather than caching the new updated cluster state after each change, the changes themselves (the deltas) should be tracked. This might allow to propagate changes between sessions or to reconcile cluster state read from Zookeeper with the stream of changes stored in a Session by identifying which deltas made it to Zookeeper, which ones are new from Zookeeper (originating from an update in another session) and which are still pending.

12. **summary:** Autoscaling placement wrong with concurrent collection creations
description: Under concurrent collection creation, wrong Autoscaling placement decisions can lead to severely unbalanced clusters. Sequential creation of the same collections is handled correctly and the cluster is balanced. *TL;DR;* under high load, the way sessions that cache future changes to Zookeeper are managed cause placement decisions of multiple concurrent Collection API calls to ignore each other, be based on identical â€œinitialâ€ cluster state, possibly leading to identical placement decisions and as a consequence cluster imbalance. *Some context first* for those less familiar with how Autoscaling deals with cluster state change: a PolicyHelper.Session is created with a snapshot of the Zookeeper cluster state and is used to track already decided but not yet persisted to Zookeeper cluster state changes so that Collection API commands can make the right placement decisions. A Collection API command either uses an existing cached Session (that includes changes computed by previous command(s)) or creates a new Session initialized from the Zookeeper cluster state (i.e. with only state changes already persisted). When a Collection API command requires a Session - and one is needed for any cluster state update computation - if one exists but is currently in use, the command can wait up to 10 seconds. If the session becomes available, it is reused. Otherwise, a new one is created. The Session lifecycle is as follows: it is created in COMPUTING state by a Collection API command and is initialized with a snapshot of cluster state from Zookeeper (does not require a Zookeeper read, this is running on Overseer that maintains a cache of cluster state). The command has exclusive access to the Session and can change the state of the Session. When the command is done changing the Session, the Session is â€œreturnedâ€ and its state changes to EXECUTING while the command continues to run to persist the state to Zookeeper and interact with the nodes, but no longer interacts with the Session. Another command can then grab a Session in EXECUTING state, change its state to COMPUTING to compute new changes taking into account previous changes. When all commands having used the session have completed their work, the session is â€œreleasedâ€ and destroyed (at this stage, Zookeeper contains all the state changes that were computed using that Session). The issue arises when multiple Collection API commands are executed at once. A first Session is created and commands start using it one by one. In a simple 1 shard 1 replica collection creation test run with 100 parallel Collection API requests (see debug logs from PolicyHelper in file policy.logs), this Session update phase (Session in COMPUTING status in SessionWrapper) takes about 250-300ms (MacBook Pro). This means that about 40 commands can run by using in turn the same Session (45 in the sample run). The commands that have been waiting for too long time out after 10 seconds, more or less all at the same time (at the rate at which they have been received by the OverseerCollectionMessageHandler, approx one per 100ms in the sample run) and most/all independently decide to create a new Session. These new Sessions are based on Zookeeper state, they might or might not include some of the changes from the first 40 commands (depending on if these commands got their changes written to Zookeeper by the time of the 10 seconds timeout, a few might have made it, see below). These new Sessions (54 sessions in addition to the initial one) are based on more or less the same state, so all remaining commands are making placement decisions that do not take into account each other (and likely not much of the first 44 placement decisions either). The sample run whose relevant logs are attached led for the 100 single shard single replica collection creations to 82 collections on the Overseer node, and 5 and 13 collections on the two other nodes of a 3 nodes cluster. Given that the initial session was used 45 times (once initially then reused 44 times), one would have expected at least the first 45 collections to be evenly distributed, i.e. 15 replicas on each node. This was not the case, possibly a sign of other issues (other runs even ended up placing 0 replicas out of the 100 on one of the nodes). From the client perspective, http admin collection CREATE requests averaged 19.5 seconds each and lasted between 7 and 28 seconds (100 parallel threads). This is likely an indication that

the last 55 collection creations didnâ€™t see much of the state updates done by the first 45 creations (client delay is longer though than actual Overseer command execution time by http time + Collections API Zookeeper queue time) . *A possible fix* is to not observe any delay before creating a new Session when the currently cached session is busy (i.e. COMPUTING). It will be somewhat less optimal in low load cases (this is likely not an issue, future creations will compensate for slight unbalance and under optimal placement) but will speed up Collection API calls (no waiting) and will prevent multiple waiting commands from all creating new Sessions based on an identical Zookeeper state in cases such as the one described here. For long (minutes and more) autoscaling computations it will likely not make a big difference. If we had more than a single Session being cached (and reused), then less ongoing updates would be lost. Maybe, rather than caching the new updated cluster state after each change, the changes themselves (the deltas) should be tracked. This might allow to propagate changes between sessions or to reconcile cluster state read from Zookeeper with the stream of changes stored in a Session by identifying which deltas made it to Zookeeper, which ones are new from Zookeeper (originating from an update in another session) and which are still pending.

13. **summary:** Autoscaling placement wrong with concurrent collection creations
description: Under concurrent collection creation, wrong Autoscaling placement decisions can lead to severely unbalanced clusters. Sequential creation of the same collections is handled correctly and the cluster is balanced. *TL;DR;* under high load, the way sessions that cache future changes to Zookeeper are managed cause placement decisions of multiple concurrent Collection API calls to ignore each other, be based on identical â€œinitialâ€ cluster state, possibly leading to identical placement decisions and as a consequence cluster imbalance. *Some context first* for those less familiar with how Autoscaling deals with cluster state change: a PolicyHelper.Session is created with a snapshot of the Zookeeper cluster state and is used to track already decided but not yet persisted to Zookeeper cluster state changes so that Collection API commands can make the right placement decisions. A Collection API command either uses an existing cached Session (that includes changes computed by previous command(s)) or creates a new Session initialized from the Zookeeper cluster state (i.e. with only state changes already persisted). When a Collection API command requires a Session - and one is needed for any cluster state update computation - if one exists but is currently in use, the command can wait up to 10 seconds. If the session becomes available, it is reused. Otherwise, a new one is created. The Session lifecycle is as follows: it is created in COMPUTING state by a Collection API command and is initialized with a snapshot of cluster state from Zookeeper (does not require a Zookeeper read, this is running on Overseer that maintains a cache of cluster state). The command has exclusive access to the Session and can change the state of the Session. When the command is done changing the Session, the Session is â€œreturnedâ€ and its state changes to EXECUTING while the command continues to run to persist the state to Zookeeper and interact with the nodes, but no longer interacts with the Session. Another command can then grab a Session in EXECUTING state, change its state to COMPUTING to compute new changes taking into account previous changes. When all commands having used the session have completed their work, the session is â€œreleasedâ€ and destroyed (at this stage, Zookeeper contains all the state changes that were computed using that Session). The issue arises when multiple Collection API commands are executed at once. A first Session is created and commands start using it one by one. In a simple 1 shard 1 replica collection creation test run with 100 parallel Collection API requests (see debug logs from PolicyHelper in file policy.logs), this Session update phase (Session in COMPUTING status in SessionWrapper) takes about 250-300ms (MacBook Pro). This means that about 40 commands can run by using in turn the same Session (45 in the sample run). The commands that have been waiting for too long time out after 10 seconds, more or less all at the same time (at the rate at which they have been received by the OverseerCollectionMessageHandler, approx one per 100ms in the sample run) and most/all independently decide to create a new Session. These new Sessions are based on Zookeeper state, they might or might not include some of the changes from the first 40 commands (depending on if these commands got their changes written to Zookeeper by the time of the 10 seconds timeout, a few might have made it, see below). These new Sessions (54 sessions in addition to the initial one) are based on more or less the same state, so all remaining commands are making placement decisions that do not take into account each other (and likely not much of the first 44 placement decisions either). The sample run whose relevant logs are attached led for the 100 single shard single replica collection creations to 82 collections on the Overseer node, and 5 and 13 collections on the two other nodes of a 3 nodes cluster. Given that the initial session was used 45 times (once initially then reused 44 times), one would have expected at least the first 45 collections to be evenly distributed, i.e. 15 replicas on each node. This was not the case, possibly a sign of other issues (other runs even ended up placing 0 replicas out of the 100 on one of the nodes). From the client perspective, http admin collection CREATE requests averaged 19.5

seconds each and lasted between 7 and 28 seconds (100 parallel threads). This is likely an indication that the last 55 collection creations didnâ€™t see much of the state updates done by the first 45 creations (client delay is longer though than actual Overseer command execution time by http time + Collections API Zookeeper queue time) . *A possible fix* is to not observe any delay before creating a new Session when the currently cached session is busy (i.e. COMPUTING). It will be somewhat less optimal in low load cases (this is likely not an issue, future creations will compensate for slight unbalance and under optimal placement) but will speed up Collection API calls (no waiting) and will prevent multiple waiting commands from all creating new Sessions based on an identical Zookeeper state in cases such as the one described here. For long (minutes and more) autoscaling computations it will likely not make a big difference. If we had more than a single Session being cached (and reused), then less ongoing updates would be lost. Maybe, rather than caching the new updated cluster state after each change, the changes themselves (the deltas) should be tracked. This might allow to propagate changes between sessions or to reconcile cluster state read from Zookeeper with the stream of changes stored in a Session by identifying which deltas made it to Zookeeper, which ones are new from Zookeeper (originating from an update in another session) and which are still pending.

14. **summary:** Autoscaling placement wrong with concurrent collection creations
description: Under concurrent collection creation, wrong Autoscaling placement decisions can lead to severely unbalanced clusters. Sequential creation of the same collections is handled correctly and the cluster is balanced. *TL;DR;* under high load, the way sessions that cache future changes to Zookeeper are managed cause placement decisions of multiple concurrent Collection API calls to ignore each other, be based on identical â€œinitialâ€ cluster state, possibly leading to identical placement decisions and as a consequence cluster imbalance. *Some context first* for those less familiar with how Autoscaling deals with cluster state change: a PolicyHelper.Session is created with a snapshot of the Zookeeper cluster state and is used to track already decided but not yet persisted to Zookeeper cluster state changes so that Collection API commands can make the right placement decisions. A Collection API command either uses an existing cached Session (that includes changes computed by previous command(s)) or creates a new Session initialized from the Zookeeper cluster state (i.e. with only state changes already persisted). When a Collection API command requires a Session - and one is needed for any cluster state update computation - if one exists but is currently in use, the command can wait up to 10 seconds. If the session becomes available, it is reused. Otherwise, a new one is created. The Session lifecycle is as follows: it is created in COMPUTING state by a Collection API command and is initialized with a snapshot of cluster state from Zookeeper (does not require a Zookeeper read, this is running on Overseer that maintains a cache of cluster state). The command has exclusive access to the Session and can change the state of the Session. When the command is done changing the Session, the Session is â€œreturnedâ€ and its state changes to EXECUTING while the command continues to run to persist the state to Zookeeper and interact with the nodes, but no longer interacts with the Session. Another command can then grab a Session in EXECUTING state, change its state to COMPUTING to compute new changes taking into account previous changes. When all commands having used the session have completed their work, the session is â€œreleasedâ€ and destroyed (at this stage, Zookeeper contains all the state changes that were computed using that Session). The issue arises when multiple Collection API commands are executed at once. A first Session is created and commands start using it one by one. In a simple 1 shard 1 replica collection creation test run with 100 parallel Collection API requests (see debug logs from PolicyHelper in file policy.logs), this Session update phase (Session in COMPUTING status in SessionWrapper) takes about 250-300ms (MacBook Pro). This means that about 40 commands can run by using in turn the same Session (45 in the sample run). The commands that have been waiting for too long time out after 10 seconds, more or less all at the same time (at the rate at which they have been received by the OverseerCollectionMessageHandler, approx one per 100ms in the sample run) and most/all independently decide to create a new Session. These new Sessions are based on Zookeeper state, they might or might not include some of the changes from the first 40 commands (depending on if these commands got their changes written to Zookeeper by the time of the 10 seconds timeout, a few might have made it, see below). These new Sessions (54 sessions in addition to the initial one) are based on more or less the same state, so all remaining commands are making placement decisions that do not take into account each other (and likely not much of the first 44 placement decisions either). The sample run whose relevant logs are attached led for the 100 single shard single replica collection creations to 82 collections on the Overseer node, and 5 and 13 collections on the two other nodes of a 3 nodes cluster. Given that the initial session was used 45 times (once initially then reused 44 times), one would have expected at least the first 45 collections to be evenly distributed, i.e. 15 replicas on each node. This was not the case, possibly a sign of other issues (other runs even ended up placing 0 replicas out of the 100 on

one of the nodes). From the client perspective, http admin collection CREATE requests averaged 19.5 seconds each and lasted between 7 and 28 seconds (100 parallel threads). This is likely an indication that the last 55 collection creations didnâ€™t see much of the state updates done by the first 45 creations (client delay is longer though than actual Overseer command execution time by http time + Collections API Zookeeper queue time) . *A possible fix* is to not observe any delay before creating a new Session when the currently cached session is busy (i.e. COMPUTING). It will be somewhat less optimal in low load cases (this is likely not an issue, future creations will compensate for slight unbalance and under optimal placement) but will speed up Collection API calls (no waiting) and will prevent multiple waiting commands from all creating new Sessions based on an identical Zookeeper state in cases such as the one described here. For long (minutes and more) autoscaling computations it will likely not make a big difference. If we had more than a single Session being cached (and reused), then less ongoing updates would be lost. Maybe, rather than caching the new updated cluster state after each change, the changes themselves (the deltas) should be tracked. This might allow to propagate changes between sessions or to reconcile cluster state read from Zookeeper with the stream of changes stored in a Session by identifying which deltas made it to Zookeeper, which ones are new from Zookeeper (originating from an update in another session) and which are still pending.

15. **summary:** Autoscaling placement wrong with concurrent collection creations
description: Under concurrent collection creation, wrong Autoscaling placement decisions can lead to severely unbalanced clusters. Sequential creation of the same collections is handled correctly and the cluster is balanced. *TL;DR;* under high load, the way sessions that cache future changes to Zookeeper are managed cause placement decisions of multiple concurrent Collection API calls to ignore each other, be based on identical â€œinitialâ€ cluster state, possibly leading to identical placement decisions and as a consequence cluster imbalance. *Some context first* for those less familiar with how Autoscaling deals with cluster state change: a PolicyHelper.Session is created with a snapshot of the Zookeeper cluster state and is used to track already decided but not yet persisted to Zookeeper cluster state changes so that Collection API commands can make the right placement decisions. A Collection API command either uses an existing cached Session (that includes changes computed by previous command(s)) or creates a new Session initialized from the Zookeeper cluster state (i.e. with only state changes already persisted). When a Collection API command requires a Session - and one is needed for any cluster state update computation - if one exists but is currently in use, the command can wait up to 10 seconds. If the session becomes available, it is reused. Otherwise, a new one is created. The Session lifecycle is as follows: it is created in COMPUTING state by a Collection API command and is initialized with a snapshot of cluster state from Zookeeper (does not require a Zookeeper read, this is running on Overseer that maintains a cache of cluster state). The command has exclusive access to the Session and can change the state of the Session. When the command is done changing the Session, the Session is â€œreturnedâ€ and its state changes to EXECUTING while the command continues to run to persist the state to Zookeeper and interact with the nodes, but no longer interacts with the Session. Another command can then grab a Session in EXECUTING state, change its state to COMPUTING to compute new changes taking into account previous changes. When all commands having used the session have completed their work, the session is â€œreleasedâ€ and destroyed (at this stage, Zookeeper contains all the state changes that were computed using that Session). The issue arises when multiple Collection API commands are executed at once. A first Session is created and commands start using it one by one. In a simple 1 shard 1 replica collection creation test run with 100 parallel Collection API requests (see debug logs from PolicyHelper in file policy.logs), this Session update phase (Session in COMPUTING status in SessionWrapper) takes about 250-300ms (MacBook Pro). This means that about 40 commands can run by using in turn the same Session (45 in the sample run). The commands that have been waiting for too long time out after 10 seconds, more or less all at the same time (at the rate at which they have been received by the OverseerCollectionMessageHandler, approx one per 100ms in the sample run) and most/all independently decide to create a new Session. These new Sessions are based on Zookeeper state, they might or might not include some of the changes from the first 40 commands (depending on if these commands got their changes written to Zookeeper by the time of the 10 seconds timeout, a few might have made it, see below). These new Sessions (54 sessions in addition to the initial one) are based on more or less the same state, so all remaining commands are making placement decisions that do not take into account each other (and likely not much of the first 44 placement decisions either). The sample run whose relevant logs are attached led for the 100 single shard single replica collection creations to 82 collections on the Overseer node, and 5 and 13 collections on the two other nodes of a 3 nodes cluster. Given that the initial session was used 45 times (once initially then reused 44 times), one would have expected at least the first 45 collections to be evenly distributed, i.e. 15 replicas on each node. This was

not the case, possibly a sign of other issues (other runs even ended up placing 0 replicas out of the 100 on one of the nodes). From the client perspective, http admin collection CREATE requests averaged 19.5 seconds each and lasted between 7 and 28 seconds (100 parallel threads). This is likely an indication that the last 55 collection creations didnâ€™t see much of the state updates done by the first 45 creations (client delay is longer though than actual Overseer command execution time by http time + Collections API Zookeeper queue time) . *A possible fix* is to not observe any delay before creating a new Session when the currently cached session is busy (i.e. COMPUTING). It will be somewhat less optimal in low load cases (this is likely not an issue, future creations will compensate for slight unbalance and under optimal placement) but will speed up Collection API calls (no waiting) and will prevent multiple waiting commands from all creating new Sessions based on an identical Zookeeper state in cases such as the one described here. For long (minutes and more) autoscaling computations it will likely not make a big difference. If we had more than a single Session being cached (and reused), then less ongoing updates would be lost. Maybe, rather than caching the new updated cluster state after each change, the changes themselves (the deltas) should be tracked. This might allow to propagate changes between sessions or to reconcile cluster state read from Zookeeper with the stream of changes stored in a Session by identifying which deltas made it to Zookeeper, which ones are new from Zookeeper (originating from an update in another session) and which are still pending.

16. **summary:** Autoscaling placement wrong with concurrent collection creations
description: Under concurrent collection creation, wrong Autoscaling placement decisions can lead to severely unbalanced clusters. Sequential creation of the same collections is handled correctly and the cluster is balanced. *TL;DR;* under high load, the way sessions that cache future changes to Zookeeper are managed cause placement decisions of multiple concurrent Collection API calls to ignore each other, be based on identical â€œinitialâ€ cluster state, possibly leading to identical placement decisions and as a consequence cluster imbalance. *Some context first* for those less familiar with how Autoscaling deals with cluster state change: a PolicyHelper.Session is created with a snapshot of the Zookeeper cluster state and is used to track already decided but not yet persisted to Zookeeper cluster state changes so that Collection API commands can make the right placement decisions. A Collection API command either uses an existing cached Session (that includes changes computed by previous command(s)) or creates a new Session initialized from the Zookeeper cluster state (i.e. with only state changes already persisted). When a Collection API command requires a Session - and one is needed for any cluster state update computation - if one exists but is currently in use, the command can wait up to 10 seconds. If the session becomes available, it is reused. Otherwise, a new one is created. The Session lifecycle is as follows: it is created in COMPUTING state by a Collection API command and is initialized with a snapshot of cluster state from Zookeeper (does not require a Zookeeper read, this is running on Overseer that maintains a cache of cluster state). The command has exclusive access to the Session and can change the state of the Session. When the command is done changing the Session, the Session is â€œreturnedâ€ and its state changes to EXECUTING while the command continues to run to persist the state to Zookeeper and interact with the nodes, but no longer interacts with the Session. Another command can then grab a Session in EXECUTING state, change its state to COMPUTING to compute new changes taking into account previous changes. When all commands having used the session have completed their work, the session is â€œreleasedâ€ and destroyed (at this stage, Zookeeper contains all the state changes that were computed using that Session). The issue arises when multiple Collection API commands are executed at once. A first Session is created and commands start using it one by one. In a simple 1 shard 1 replica collection creation test run with 100 parallel Collection API requests (see debug logs from PolicyHelper in file policy.logs), this Session update phase (Session in COMPUTING status in SessionWrapper) takes about 250-300ms (MacBook Pro). This means that about 40 commands can run by using in turn the same Session (45 in the sample run). The commands that have been waiting for too long time out after 10 seconds, more or less all at the same time (at the rate at which they have been received by the OverseerCollectionMessageHandler, approx one per 100ms in the sample run) and most/all independently decide to create a new Session. These new Sessions are based on Zookeeper state, they might or might not include some of the changes from the first 40 commands (depending on if these commands got their changes written to Zookeeper by the time of the 10 seconds timeout, a few might have made it, see below). These new Sessions (54 sessions in addition to the initial one) are based on more or less the same state, so all remaining commands are making placement decisions that do not take into account each other (and likely not much of the first 44 placement decisions either). The sample run whose relevant logs are attached led for the 100 single shard single replica collection creations to 82 collections on the Overseer node, and 5 and 13 collections on the two other nodes of a 3 nodes cluster. Given that the initial session was used 45 times (once initially then reused 44 times), one would have

expected at least the first 45 collections to be evenly distributed, i.e. 15 replicas on each node. This was not the case, possibly a sign of other issues (other runs even ended up placing 0 replicas out of the 100 on one of the nodes). From the client perspective, http admin collection CREATE requests averaged 19.5 seconds each and lasted between 7 and 28 seconds (100 parallel threads). This is likely an indication that the last 55 collection creations didnâ€™t see much of the state updates done by the first 45 creations (client delay is longer though than actual Overseer command execution time by http time + Collections API Zookeeper queue time) . *A possible fix* is to not observe any delay before creating a new Session when the currently cached session is busy (i.e. COMPUTING). It will be somewhat less optimal in low load cases (this is likely not an issue, future creations will compensate for slight unbalance and under optimal placement) but will speed up Collection API calls (no waiting) and will prevent multiple waiting commands from all creating new Sessions based on an identical Zookeeper state in cases such as the one described here. For long (minutes and more) autoscaling computations it will likely not make a big difference. If we had more than a single Session being cached (and reused), then less ongoing updates would be lost. Maybe, rather than caching the new updated cluster state after each change, the changes themselves (the deltas) should be tracked. This might allow to propagate changes between sessions or to reconcile cluster state read from Zookeeper with the stream of changes stored in a Session by identifying which deltas made it to Zookeeper, which ones are new from Zookeeper (originating from an update in another session) and which are still pending.

17. **summary:** Autoscaling placement wrong with concurrent collection creations
description: Under concurrent collection creation, wrong Autoscaling placement decisions can lead to severely unbalanced clusters. Sequential creation of the same collections is handled correctly and the cluster is balanced. *TL;DR;* under high load, the way sessions that cache future changes to Zookeeper are managed cause placement decisions of multiple concurrent Collection API calls to ignore each other, be based on identical â€œinitialâ€ cluster state, possibly leading to identical placement decisions and as a consequence cluster imbalance. *Some context first* for those less familiar with how Autoscaling deals with cluster state change: a PolicyHelper.Session is created with a snapshot of the Zookeeper cluster state and is used to track already decided but not yet persisted to Zookeeper cluster state changes so that Collection API commands can make the right placement decisions. A Collection API command either uses an existing cached Session (that includes changes computed by previous command(s)) or creates a new Session initialized from the Zookeeper cluster state (i.e. with only state changes already persisted). When a Collection API command requires a Session - and one is needed for any cluster state update computation - if one exists but is currently in use, the command can wait up to 10 seconds. If the session becomes available, it is reused. Otherwise, a new one is created. The Session lifecycle is as follows: it is created in COMPUTING state by a Collection API command and is initialized with a snapshot of cluster state from Zookeeper (does not require a Zookeeper read, this is running on Overseer that maintains a cache of cluster state). The command has exclusive access to the Session and can change the state of the Session. When the command is done changing the Session, the Session is â€œreturnedâ€ and its state changes to EXECUTING while the command continues to run to persist the state to Zookeeper and interact with the nodes, but no longer interacts with the Session. Another command can then grab a Session in EXECUTING state, change its state to COMPUTING to compute new changes taking into account previous changes. When all commands having used the session have completed their work, the session is â€œreleasedâ€ and destroyed (at this stage, Zookeeper contains all the state changes that were computed using that Session). The issue arises when multiple Collection API commands are executed at once. A first Session is created and commands start using it one by one. In a simple 1 shard 1 replica collection creation test run with 100 parallel Collection API requests (see debug logs from PolicyHelper in file policy.logs), this Session update phase (Session in COMPUTING status in SessionWrapper) takes about 250-300ms (MacBook Pro). This means that about 40 commands can run by using in turn the same Session (45 in the sample run). The commands that have been waiting for too long time out after 10 seconds, more or less all at the same time (at the rate at which they have been received by the OverseerCollectionMessageHandler, approx one per 100ms in the sample run) and most/all independently decide to create a new Session. These new Sessions are based on Zookeeper state, they might or might not include some of the changes from the first 40 commands (depending on if these commands got their changes written to Zookeeper by the time of the 10 seconds timeout, a few might have made it, see below). These new Sessions (54 sessions in addition to the initial one) are based on more or less the same state, so all remaining commands are making placement decisions that do not take into account each other (and likely not much of the first 44 placement decisions either). The sample run whose relevant logs are attached led for the 100 single shard single replica collection creations to 82 collections on the Overseer node, and 5 and 13 collections on the two other nodes of a 3 nodes cluster.

Given that the initial session was used 45 times (once initially then reused 44 times), one would have expected at least the first 45 collections to be evenly distributed, i.e. 15 replicas on each node. This was not the case, possibly a sign of other issues (other runs even ended up placing 0 replicas out of the 100 on one of the nodes). From the client perspective, http admin collection CREATE requests averaged 19.5 seconds each and lasted between 7 and 28 seconds (100 parallel threads). This is likely an indication that the last 55 collection creations didnâ€™t see much of the state updates done by the first 45 creations (client delay is longer though than actual Overseer command execution time by http time + Collections API Zookeeper queue time) . *A possible fix* is to not observe any delay before creating a new Session when the currently cached session is busy (i.e. COMPUTING). It will be somewhat less optimal in low load cases (this is likely not an issue, future creations will compensate for slight unbalance and under optimal placement) but will speed up Collection API calls (no waiting) and will prevent multiple waiting commands from all creating new Sessions based on an identical Zookeeper state in cases such as the one described here. For long (minutes and more) autoscaling computations it will likely not make a big difference. If we had more than a single Session being cached (and reused), then less ongoing updates would be lost. Maybe, rather than caching the new updated cluster state after each change, the changes themselves (the deltas) should be tracked. This might allow to propagate changes between sessions or to reconcile cluster state read from Zookeeper with the stream of changes stored in a Session by identifying which deltas made it to Zookeeper, which ones are new from Zookeeper (originating from an update in another session) and which are still pending.

18. **summary:** Autoscaling placement wrong with concurrent collection creations
description: Under concurrent collection creation, wrong Autoscaling placement decisions can lead to severely unbalanced clusters. Sequential creation of the same collections is handled correctly and the cluster is balanced. *TL;DR;* under high load, the way sessions that cache future changes to Zookeeper are managed cause placement decisions of multiple concurrent Collection API calls to ignore each other, be based on identical â€œinitialâ€ cluster state, possibly leading to identical placement decisions and as a consequence cluster imbalance. *Some context first* for those less familiar with how Autoscaling deals with cluster state change: a PolicyHelper.Session is created with a snapshot of the Zookeeper cluster state and is used to track already decided but not yet persisted to Zookeeper cluster state changes so that Collection API commands can make the right placement decisions. A Collection API command either uses an existing cached Session (that includes changes computed by previous command(s)) or creates a new Session initialized from the Zookeeper cluster state (i.e. with only state changes already persisted). When a Collection API command requires a Session - and one is needed for any cluster state update computation - if one exists but is currently in use, the command can wait up to 10 seconds. If the session becomes available, it is reused. Otherwise, a new one is created. The Session lifecycle is as follows: it is created in COMPUTING state by a Collection API command and is initialized with a snapshot of cluster state from Zookeeper (does not require a Zookeeper read, this is running on Overseer that maintains a cache of cluster state). The command has exclusive access to the Session and can change the state of the Session. When the command is done changing the Session, the Session is â€œreturnedâ€ and its state changes to EXECUTING while the command continues to run to persist the state to Zookeeper and interact with the nodes, but no longer interacts with the Session. Another command can then grab a Session in EXECUTING state, change its state to COMPUTING to compute new changes taking into account previous changes. When all commands having used the session have completed their work, the session is â€œreleasedâ€ and destroyed (at this stage, Zookeeper contains all the state changes that were computed using that Session). The issue arises when multiple Collection API commands are executed at once. A first Session is created and commands start using it one by one. In a simple 1 shard 1 replica collection creation test run with 100 parallel Collection API requests (see debug logs from PolicyHelper in file policy.logs), this Session update phase (Session in COMPUTING status in SessionWrapper) takes about 250-300ms (MacBook Pro). This means that about 40 commands can run by using in turn the same Session (45 in the sample run). The commands that have been waiting for too long time out after 10 seconds, more or less all at the same time (at the rate at which they have been received by the OverseerCollectionMessageHandler, approx one per 100ms in the sample run) and most/all independently decide to create a new Session. These new Sessions are based on Zookeeper state, they might or might not include some of the changes from the first 40 commands (depending on if these commands got their changes written to Zookeeper by the time of the 10 seconds timeout, a few might have made it, see below). These new Sessions (54 sessions in addition to the initial one) are based on more or less the same state, so all remaining commands are making placement decisions that do not take into account each other (and likely not much of the first 44 placement decisions either). The sample run whose relevant logs are attached led for the 100 single shard single replica collection creations to 82

collections on the Overseer node, and 5 and 13 collections on the two other nodes of a 3 nodes cluster. Given that the initial session was used 45 times (once initially then reused 44 times), one would have expected at least the first 45 collections to be evenly distributed, i.e. 15 replicas on each node. This was not the case, possibly a sign of other issues (other runs even ended up placing 0 replicas out of the 100 on one of the nodes). From the client perspective, http admin collection CREATE requests averaged 19.5 seconds each and lasted between 7 and 28 seconds (100 parallel threads). This is likely an indication that the last 55 collection creations didnâ€™t see much of the state updates done by the first 45 creations (client delay is longer though than actual Overseer command execution time by http time + Collections API Zookeeper queue time) . *A possible fix* is to not observe any delay before creating a new Session when the currently cached session is busy (i.e. COMPUTING). It will be somewhat less optimal in low load cases (this is likely not an issue, future creations will compensate for slight unbalance and under optimal placement) but will speed up Collection API calls (no waiting) and will prevent multiple waiting commands from all creating new Sessions based on an identical Zookeeper state in cases such as the one described here. For long (minutes and more) autoscaling computations it will likely not make a big difference. If we had more than a single Session being cached (and reused), then less ongoing updates would be lost. Maybe, rather than caching the new updated cluster state after each change, the changes themselves (the deltas) should be tracked. This might allow to propagate changes between sessions or to reconcile cluster state read from Zookeeper with the stream of changes stored in a Session by identifying which deltas made it to Zookeeper, which ones are new from Zookeeper (originating from an update in another session) and which are still pending.

19. **summary:** Autoscaling placement wrong with concurrent collection creations
description: Under concurrent collection creation, wrong Autoscaling placement decisions can lead to severely unbalanced clusters. Sequential creation of the same collections is handled correctly and the cluster is balanced. *TL;DR;* under high load, the way sessions that cache future changes to Zookeeper are managed cause placement decisions of multiple concurrent Collection API calls to ignore each other, be based on identical â€œinitialâ€ cluster state, possibly leading to identical placement decisions and as a consequence cluster imbalance. *Some context first* for those less familiar with how Autoscaling deals with cluster state change: a PolicyHelper.Session is created with a snapshot of the Zookeeper cluster state and is used to track already decided but not yet persisted to Zookeeper cluster state changes so that Collection API commands can make the right placement decisions. A Collection API command either uses an existing cached Session (that includes changes computed by previous command(s)) or creates a new Session initialized from the Zookeeper cluster state (i.e. with only state changes already persisted). When a Collection API command requires a Session - and one is needed for any cluster state update computation - if one exists but is currently in use, the command can wait up to 10 seconds. If the session becomes available, it is reused. Otherwise, a new one is created. The Session lifecycle is as follows: it is created in COMPUTING state by a Collection API command and is initialized with a snapshot of cluster state from Zookeeper (does not require a Zookeeper read, this is running on Overseer that maintains a cache of cluster state). The command has exclusive access to the Session and can change the state of the Session. When the command is done changing the Session, the Session is â€œreturnedâ€ and its state changes to EXECUTING while the command continues to run to persist the state to Zookeeper and interact with the nodes, but no longer interacts with the Session. Another command can then grab a Session in EXECUTING state, change its state to COMPUTING to compute new changes taking into account previous changes. When all commands having used the session have completed their work, the session is â€œreleasedâ€ and destroyed (at this stage, Zookeeper contains all the state changes that were computed using that Session). The issue arises when multiple Collection API commands are executed at once. A first Session is created and commands start using it one by one. In a simple 1 shard 1 replica collection creation test run with 100 parallel Collection API requests (see debug logs from PolicyHelper in file policy.logs), this Session update phase (Session in COMPUTING status in SessionWrapper) takes about 250-300ms (MacBook Pro). This means that about 40 commands can run by using in turn the same Session (45 in the sample run). The commands that have been waiting for too long time out after 10 seconds, more or less all at the same time (at the rate at which they have been received by the OverseerCollectionMessageHandler, approx one per 100ms in the sample run) and most/all independently decide to create a new Session. These new Sessions are based on Zookeeper state, they might or might not include some of the changes from the first 40 commands (depending on if these commands got their changes written to Zookeeper by the time of the 10 seconds timeout, a few might have made it, see below). These new Sessions (54 sessions in addition to the initial one) are based on more or less the same state, so all remaining commands are making placement decisions that do not take into account each other (and likely not much of the first 44 placement decisions either). The sample run

whose relevant logs are attached led for the 100 single shard single replica collection creations to 82 collections on the Overseer node, and 5 and 13 collections on the two other nodes of a 3 nodes cluster. Given that the initial session was used 45 times (once initially then reused 44 times), one would have expected at least the first 45 collections to be evenly distributed, i.e. 15 replicas on each node. This was not the case, possibly a sign of other issues (other runs even ended up placing 0 replicas out of the 100 on one of the nodes). From the client perspective, http admin collection CREATE requests averaged 19.5 seconds each and lasted between 7 and 28 seconds (100 parallel threads). This is likely an indication that the last 55 collection creations didnâ€™t see much of the state updates done by the first 45 creations (client delay is longer though than actual Overseer command execution time by http time + Collections API Zookeeper queue time) . *A possible fix* is to not observe any delay before creating a new Session when the currently cached session is busy (i.e. COMPUTING). It will be somewhat less optimal in low load cases (this is likely not an issue, future creations will compensate for slight unbalance and under optimal placement) but will speed up Collection API calls (no waiting) and will prevent multiple waiting commands from all creating new Sessions based on an identical Zookeeper state in cases such as the one described here. For long (minutes and more) autoscaling computations it will likely not make a big difference. If we had more than a single Session being cached (and reused), then less ongoing updates would be lost. Maybe, rather than caching the new updated cluster state after each change, the changes themselves (the deltas) should be tracked. This might allow to propagate changes between sessions or to reconcile cluster state read from Zookeeper with the stream of changes stored in a Session by identifying which deltas made it to Zookeeper, which ones are new from Zookeeper (originating from an update in another session) and which are still pending.
label: code-design

20. **summary:** Autoscaling placement wrong with concurrent collection creations
description: Under concurrent collection creation, wrong Autoscaling placement decisions can lead to severely unbalanced clusters. Sequential creation of the same collections is handled correctly and the cluster is balanced. *TL;DR;* under high load, the way sessions that cache future changes to Zookeeper are managed cause placement decisions of multiple concurrent Collection API calls to ignore each other, be based on identical â€œinitialâ€ cluster state, possibly leading to identical placement decisions and as a consequence cluster imbalance. *Some context first* for those less familiar with how Autoscaling deals with cluster state change: a PolicyHelper.Session is created with a snapshot of the Zookeeper cluster state and is used to track already decided but not yet persisted to Zookeeper cluster state changes so that Collection API commands can make the right placement decisions. A Collection API command either uses an existing cached Session (that includes changes computed by previous command(s)) or creates a new Session initialized from the Zookeeper cluster state (i.e. with only state changes already persisted). When a Collection API command requires a Session - and one is needed for any cluster state update computation - if one exists but is currently in use, the command can wait up to 10 seconds. If the session becomes available, it is reused. Otherwise, a new one is created. The Session lifecycle is as follows: it is created in COMPUTING state by a Collection API command and is initialized with a snapshot of cluster state from Zookeeper (does not require a Zookeeper read, this is running on Overseer that maintains a cache of cluster state). The command has exclusive access to the Session and can change the state of the Session. When the command is done changing the Session, the Session is â€œreturnedâ€ and its state changes to EXECUTING while the command continues to run to persist the state to Zookeeper and interact with the nodes, but no longer interacts with the Session. Another command can then grab a Session in EXECUTING state, change its state to COMPUTING to compute new changes taking into account previous changes. When all commands having used the session have completed their work, the session is â€œreleasedâ€ and destroyed (at this stage, Zookeeper contains all the state changes that were computed using that Session). The issue arises when multiple Collection API commands are executed at once. A first Session is created and commands start using it one by one. In a simple 1 shard 1 replica collection creation test run with 100 parallel Collection API requests (see debug logs from PolicyHelper in file policy.logs), this Session update phase (Session in COMPUTING status in SessionWrapper) takes about 250-300ms (MacBook Pro). This means that about 40 commands can run by using in turn the same Session (45 in the sample run). The commands that have been waiting for too long time out after 10 seconds, more or less all at the same time (at the rate at which they have been received by the OverseerCollectionMessageHandler, approx one per 100ms in the sample run) and most/all independently decide to create a new Session. These new Sessions are based on Zookeeper state, they might or might not include some of the changes from the first 40 commands (depending on if these commands got their changes written to Zookeeper by the time of the 10 seconds timeout, a few might have made it, see below). These new Sessions (54 sessions in addition to the initial one) are based on

more or less the same state, so all remaining commands are making placement decisions that do not take into account each other (and likely not much of the first 44 placement decisions either). The sample run whose relevant logs are attached led for the 100 single shard single replica collection creations to 82 collections on the Overseer node, and 5 and 13 collections on the two other nodes of a 3 nodes cluster. Given that the initial session was used 45 times (once initially then reused 44 times), one would have expected at least the first 45 collections to be evenly distributed, i.e. 15 replicas on each node. This was not the case, possibly a sign of other issues (other runs even ended up placing 0 replicas out of the 100 on one of the nodes). From the client perspective, http admin collection CREATE requests averaged 19.5 seconds each and lasted between 7 and 28 seconds (100 parallel threads). This is likely an indication that the last 55 collection creations didnâ€™t see much of the state updates done by the first 45 creations (client delay is longer though than actual Overseer command execution time by http time + Collections API Zookeeper queue time) . *A possible fix* is to not observe any delay before creating a new Session when the currently cached session is busy (i.e. COMPUTING). It will be somewhat less optimal in low load cases (this is likely not an issue, future creations will compensate for slight unbalance and under optimal placement) but will speed up Collection API calls (no waiting) and will prevent multiple waiting commands from all creating new Sessions based on an identical Zookeeper state in cases such as the one described here. For long (minutes and more) autoscaling computations it will likely not make a big difference. If we had more than a single Session being cached (and reused), then less ongoing updates would be lost. Maybe, rather than caching the new updated cluster state after each change, the changes themselves (the deltas) should be tracked. This might allow to propagate changes between sessions or to reconcile cluster state read from Zookeeper with the stream of changes stored in a Session by identifying which deltas made it to Zookeeper, which ones are new from Zookeeper (originating from an update in another session) and which are still pending.

jira_issues_comments:

1. **body:** Did a test with no wait before creating a new Session if current cached session is COMPUTING. Doesn't work (creates a new Session every time more or less, saw 99 sessions for 100 collection creations) because the way the cache can only hold a single session make them non reusable. Not waiting before creating a session implies being able to cache more than one. Note the run was faster with better throughput of creations per second. Max time was significantly lower, cluster imbalance was similar. Measures below are http request times seen from JMeter for creation of 1 shard 1 replica collections. Wait 10 seconds to see if session becomes available: Avg 17879ms, min 7794, max 26063, 3.8/sec, 81/16/3 collections per node, 57 Sessions created total. Do not wait, create new session if cached one not available: Avg 17721ms, min 7097, max 20951, 4.7/sec, 80/11/9 collections per node, 99 Sessions created total.
   label: code-design
2. **body:** [~ab] [~noble] I have created a PR for fixing the caching of a single Session and cache instead multiple sessions. Also randomized the wait time a bit (from 10 seconds to 1-10 seconds) so that in case of multiple concurrent requests not all wake up together. [https://github.com/apache/lucene-solr/pull/1504] Comparing the same parallel 100 simple collection (1 replica 1 shard, 3 nodes) creation run between before and after the change: Before: Avg 17879ms, min 7794, max 26063, 3.8/sec, 81/16/3 collections per node, 57 Sessions created total. After: Avg 17328ms, min 4743, max 25176, 3.9/sec, 95/5/0 collections per node, 32 Sessions created total. Better session reuse but still very high (even higher) imbalance, due as stated above to something else. Will keep investigating.
   label: code-design
3. Thanks [~murblanc] I shall review this soon
4. **body:** Gradle precommit (that I did not run) reports 2 violations: _cause: 'hasParens true prevLineNotIf true pernicketyLevel true' Suspicious logging call, Parameterize and possibly surround with 'if (log.is*Enabled) \{..}'. Help at: 'gradlew helpValidateLogCalls'_ _/home/runner/work/lucene-solr/lucene-solr/solr/solrj/src/java/org/apache/solr/client/solrj/cloud/autoscaling/PolicyHelper.java:433,_ This one is wrongly detected, there is a surrounding "if (log.isDebugEnabled())" around the log, but the if bloc also gets the TimeSource. _cause: 'hasPlus: true' Suspicious logging call, Parameterize and possibly surround with 'if (log.is*Enabled) \{..}'. Help at: 'gradlew helpValidateLogCalls'_ _/home/runner/work/lucene-solr/lucene-solr/solr/solrj/src/java/org/apache/solr/client/solrj/cloud/autoscaling/PolicyHelper.java:555)._ Likely because the log is written as: log.debug("New session created, " + sessionWrapper.getCreateTime()); and not: log.debug("New session created, {}", sessionWrapper.getCreateTime()); The direct concatenation is more efficient in Java. If Solr coding standards impose the "format" style for logs, I'll change it (format style make sense when there's not

surrounding if for debug level, as they're cheaper to not execute than the concatenation that would have to be executed even if the log is not output).
label: code-design

5. PolicyHelper logs for 100 collection creation with [PR 1504|https://github.com/apache/lucene-solr/pull/1504]. [^PolicyHelperNewLogs.txt]
6. In existing (9.0 master) code as well as in the PR, when a new Session is required, it is created in PolicyHelper.createSession() called from PolicyHelper.get(). The session is therefore created while holding the lockObj lock! When SolrCloud has a large number of collections/shards/replicas, session creation can take a few seconds. Parallel session creation is therefore significantly delayed. It would be better to not hold the lock while creating the session. That lock should only be used to protect changes to SessionRef (and should be acquired after a Session got created to register that session with the SessionRef).
7. I took into account the comments and updated the PR (+ rebase). [https://github.com/apache/lucene-solr/pull/1504] This includes now changes to the way new Sessions are created when needed, by +taking the actual creation outside of the critical section+. Performance tests run elsewhere showed that with a large number of collections in a cluster creating the session could take quite some time, and because it is serialized with existing implementation, these times add up when multiple commands are run (and running 100 commands concurrently is supported in Overseer). With the proposal here, creations can happen concurrently. There's a random wait delay used to wait for sessions to be returned if there already exists cached sessions OR if sessions are in the process of being created. This avoid the thundering herd effect of all waiting threads making the same decision at the same time and ending up creating a large number of sessions.
8. [~noble.paul] do you think you'll be able to have a look at this?
9. Commit 25428013fb0ed8f8fdbebdef3f1d65dea77129c2 in lucene-solr's branch refs/heads/master from Ilan Ginzburg [https://gitbox.apache.org/repos/asf?p=lucene-solr.git;h=2542801] SOLR-14462: cache more than one autoscaling session (#1504) SOLR-14462: cache more than one autoscaling session
10. Commit 25428013fb0ed8f8fdbebdef3f1d65dea77129c2 in lucene-solr's branch refs/heads/master from Ilan Ginzburg [https://gitbox.apache.org/repos/asf?p=lucene-solr.git;h=2542801] SOLR-14462: cache more than one autoscaling session (#1504) SOLR-14462: cache more than one autoscaling session
11. This should be back-ported to 8.6, it's an important bugfix.
12. Reopening to backport for 8.6. FYI [~bruno.roustant]
13. PR for backport to branch_8x for inclusion in 8.6 at [https://github.com/apache/lucene-solr/pull/1630] Don't know if I'll have more luck now for a review than with the original version in master, otherwise I'll merge that later today...
14. Commit 78152876fda92c61d1c6bcdf5e8953042a592b4f in lucene-solr's branch refs/heads/branch_8x from Ilan Ginzburg [https://gitbox.apache.org/repos/asf?p=lucene-solr.git;h=7815287] SOLR-14462: cache more than one autoscaling session (#1630) Cherry picked from 25428013fb0ed8f8fdbebdef3f1d65dea77129c2
15. Merged into branch_8x for inclusion in 8.6 release.
16. Commit 06b1f3e86694b35365fd569a0581b1f6fc2cadb3 in lucene-solr's branch refs/heads/master from Ilan Ginzburg [https://gitbox.apache.org/repos/asf?p=lucene-solr.git;h=06b1f3e] SOLR-14462: adjust test so less sessions are used even if test runs slowly. fix synchronization issue. (#1656)
17. Commit e65631e026c5bb5c8eeb6fd1351bf798c0c6985c in lucene-solr's branch refs/heads/branch_8x from Ilan Ginzburg [https://gitbox.apache.org/repos/asf?p=lucene-solr.git;h=e65631e] SOLR-14462: adjust test so less sessions are used even if test runs slowly. fix synchronization issue. (#1657) cherry picked from 06b1f3e86694b35365fd569a0581b1f6fc2cadb3
18. Closing after the 8.6.0 release