Item 342
**git_comments:**

1. * * Provide an immutable snapshot view of migrating tablets. Objects contained in the set may still be mutable.

**git_commits:**

1. **summary:** ACCUMULO-4143 Make protective copy of migrations
   **message:** ACCUMULO-4143 Make protective copy of migrations Makes a protective copy of the master's migrations, when requested, to prevent ConcurrentModificationExceptions when the requestor iterates over the set at the same time the master is updating it. This closes #68

**github_issues:**

**github_issues_comments:**

**github_pulls:**

1. **title:** ACCUMULO-4143 Make protective copy of migrations
   **body:** Makes a protective copy of the master's migrations, when requested, to prevent ConcurrentModificationExceptions when the requestor iterates over the set at the same time the master is updating it.

**github_pulls_comments:**

1. looks good to me too. I don't have a strong opinion about the immutable set. It was a thought i had while looking at the code.
2. > I don't have a strong opinion about the immutable set. It was a thought i had while looking at the code. Likewise for me. I thought I remembered seeing other immutable wrappers in the Master similar to your change, but there's nothing wrong with this as-is.

**github_pulls_reviews:**

1. could return an immutable set
2. **body:** Seems to put an unnecessary restriction on how other tools might use it, though. Didn't think that was necessary.
   **label:** code-design
3. My thinking is that we don't expect the caller to modify it. Now that a copy is made, its ok if caller modifies. But if in the future the implementation uses a concurrent map and does not copy, would not want caller of method to modify. Also the previous implementation didn't expect caller to modify, just read.
4. +1
5. **body:** I was thinking a bigger risk would be that a user would accidentally introduce a RuntimeException (UnsupportedOperationException) because they tried to do something which worked against some implementations of the interface, but not others, and because wrapped immutability is not something which we can check at compile-time. It just seems an unnecessary restriction to me. It's internal-only code, so we can change it any way we like in the future... and I think the risk you're highlighting is lower than the risk of somebody accidentally calling an unsupported method... and it's also much easier to protect against at the time the implementation changes. If it changes from a protective copy to another implementation, I would think it'd be obvious for the implementor to check to see if that breaks usage at that time. For now, I think it makes sense to treat the interface as "snapshot of migrations", and the protective copy (vs. another implementation) matches up to that interface semantics. We could change the interface method name to "migrationsSnapshot()" to be clear, though, and to avoid careless future impl changes which change that semantics.
   **label:** code-design
6. **body:** > We could change the interface method name to "migrationsSnapshot()" to be clear, though, and to avoid careless future impl changes which change that semantics. :+1: I like this general approach. I think migrations are just one place that we follow this pattern and could benefit from more descriptive method names.
   **label:** code-design
7. > I was thinking a bigger risk would be that a user would accidentally introduce a RuntimeException My hope was that the runtime exception would possibly expose a bug (in addition to causing a bug). However, there are certainly cases where it would only cause problems and not expose any bugs. > wrapped immutability is not something which we can check at compile-time That would be nice to have.
8. **body:** I'll go ahead and do the immutability, as well as rename the method... but I'll add a Javadoc to clearly state that it's immutible. It looks like the other (test) implementations are already immutable anyway.
   **label:** documentation

**jira_issues:**

1. **summary:** DeleteTableDuringSplitIT.test() fails occasionally
   **description:** Saw this twice during 1.6.5-rc1 failing: {code}
   test(org.apache.accumulo.test.functional.DeleteTableDuringSplitIT) Time elapsed: 182.218 sec <<< ERROR!
   java.util.concurrent.ExecutionException: java.lang.RuntimeException:
   org.apache.accumulo.core.client.AccumuloException: Internal error processing waitForFateOperation at
   java.util.concurrent.FutureTask.report(FutureTask.java:122) at
   java.util.concurrent.FutureTask.get(FutureTask.java:188) at
   org.apache.accumulo.test.functional.DeleteTableDuringSplitIT.test(DeleteTableDuringSplitIT.java:94) Caused by:
   java.lang.RuntimeException: org.apache.accumulo.core.client.AccumuloException: Internal error processing
   waitForFateOperation at
   org.apache.accumulo.test.functional.DeleteTableDuringSplitIT$2.run(DeleteTableDuringSplitIT.java:80) at
   java.util.concurrent.Executors$RunnableAdapter.call(Executors.java:471) at
   java.util.concurrent.FutureTask.run(FutureTask.java:262) at
   java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1145) at
   java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:615) at
   org.apache.accumulo.trace.instrument.TraceRunnable.run(TraceRunnable.java:47) at
   org.apache.accumulo.core.util.LoggingRunnable.run(LoggingRunnable.java:34) at
   java.lang.Thread.run(Thread.java:745) Caused by: org.apache.accumulo.core.client.AccumuloException: Internal
   error processing waitForFateOperation at
   org.apache.accumulo.core.client.impl.TableOperationsImpl.doFateOperation(TableOperationsImpl.java:336) at
   org.apache.accumulo.core.client.impl.TableOperationsImpl.doFateOperation(TableOperationsImpl.java:294) at
   org.apache.accumulo.core.client.impl.TableOperationsImpl.doTableFateOperation(TableOperationsImpl.java:1592)
   at org.apache.accumulo.core.client.impl.TableOperationsImpl.delete(TableOperationsImpl.java:679) at
   org.apache.accumulo.test.functional.DeleteTableDuringSplitIT$2.run(DeleteTableDuringSplitIT.java:78) at
   java.util.concurrent.Executors$RunnableAdapter.call(Executors.java:471) at
   java.util.concurrent.FutureTask.run(FutureTask.java:262) at
   java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1145) at
   java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:615) at
   org.apache.accumulo.trace.instrument.TraceRunnable.run(TraceRunnable.java:47) at
   org.apache.accumulo.core.util.LoggingRunnable.run(LoggingRunnable.java:34) at
   java.lang.Thread.run(Thread.java:745) Caused by: org.apache.thrift.TApplicationException: Internal error
   processing waitForFateOperation at org.apache.thrift.TApplicationException.read(TApplicationException.java:111)
   at org.apache.thrift.TServiceClient.receiveBase(TServiceClient.java:71) at
   org.apache.accumulo.core.master.thrift.FateService$Client.recv_waitForFateOperation(FateService.java:174) at
   org.apache.accumulo.core.master.thrift.FateService$Client.waitForFateOperation(FateService.java:159) at
   org.apache.accumulo.core.client.impl.TableOperationsImpl.waitForFateOperation(TableOperationsImpl.java:266) at
   org.apache.accumulo.core.client.impl.TableOperationsImpl.doFateOperation(TableOperationsImpl.java:308) at
   org.apache.accumulo.core.client.impl.TableOperationsImpl.doFateOperation(TableOperationsImpl.java:294) at
   org.apache.accumulo.core.client.impl.TableOperationsImpl.doTableFateOperation(TableOperationsImpl.java:1592)
   at org.apache.accumulo.core.client.impl.TableOperationsImpl.delete(TableOperationsImpl.java:679) at
   org.apache.accumulo.test.functional.DeleteTableDuringSplitIT$2.run(DeleteTableDuringSplitIT.java:78) at
   java.util.concurrent.Executors$RunnableAdapter.call(Executors.java:471) at
   java.util.concurrent.FutureTask.run(FutureTask.java:262) at
   java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1145) at
   java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:615) at
   org.apache.accumulo.trace.instrument.TraceRunnable.run(TraceRunnable.java:47) at
   org.apache.accumulo.core.util.LoggingRunnable.run(LoggingRunnable.java:34) at
   java.lang.Thread.run(Thread.java:745) {code}

**jira_issues_comments:**

1. Looked at the master logs, and it looks like the fate operation threw up with: {code} java.lang.RuntimeException:
   java.util.ConcurrentModificationException at
   org.apache.accumulo.server.master.state.TabletStateChangeIterator.setMigrations(TabletStateChangeIterator.java:237)
   at
   org.apache.accumulo.server.master.state.MetaDataTableScanner.configureScanner(MetaDataTableScanner.java:89)
   at org.apache.accumulo.master.tableOps.CleanUp.isReady(DeleteTable.java:101) at
   org.apache.accumulo.master.tableOps.CleanUp.isReady(DeleteTable.java:63) at
   org.apache.accumulo.master.tableOps.TraceRepo.isReady(TraceRepo.java:44) at
   org.apache.accumulo.fate.Fate$TransactionRunner.run(Fate.java:66) at
   org.apache.accumulo.fate.util.LoggingRunnable.run(LoggingRunnable.java:34) at
   java.lang.Thread.run(Thread.java:745) Caused by: java.util.ConcurrentModificationException at

java.util.TreeMap$PrivateEntryIterator.nextEntry(TreeMap.java:1115) at java.util.TreeMap$KeyIterator.next(TreeMap.java:1169) at org.apache.accumulo.server.master.state.TabletStateChangeIterator.setMigrations(TabletStateChangeIterator.java:233) ... 7 more {code} Problem appears to be caused by the master's tracking of migrations being modified while the table is being deleted.

2. bq. Problem appears to be caused by the master's tracking of migrations being modified while the table is being deleted. Yup, Saw the same exact thing over here.
3. Further regression from ACCUMULO-3601?
4. Possibly. A solution seems to be to make the master just provide a protective copy of the keySet.
5. GitHub user ctubbsii opened a pull request: https://github.com/apache/accumulo/pull/68 ACCUMULO-4143 Make protective copy of migrations Makes a protective copy of the master's migrations, when requested, to prevent ConcurrentModificationExceptions when the requestor iterates over the set at the same time the master is updating it. You can merge this pull request into a Git repository by running: $ git pull https://github.com/ctubbsii/accumulo ACCUMULO-4143 Alternatively you can review and apply these changes as the patch at: https://github.com/apache/accumulo/pull/68.patch To close this pull request, make a commit to your master/trunk branch with (at least) the following in the commit message: This closes #68 ---- commit 7c5ab89ff70f5d25f764981e4f7c70da52bc8fef Author: Christopher Tubbs <ctubbsii@apache.org> Date: 2016-02-11T20:55:16Z ACCUMULO-4143 Make protective copy of migrations Makes a protective copy of the master's migrations, when requested, to prevent ConcurrentModificationExceptions when the requestor iterates over the set at the same time the master is updating it. ----
6. Github user keith-turner commented on a diff in the pull request: https://github.com/apache/accumulo/pull/68#discussion_r52667397 --- Diff: server/master/src/main/java/org/apache/accumulo/master/Master.java --- @@ -1336,7 +1332,11 @@ public void assignedTablet(KeyExtent extent) { } @Override - public Collection<KeyExtent> migrations() { - return migrations.keySet(); + public Set<KeyExtent> migrations() { + Set<KeyExtent> migrationsCopy = new HashSet<KeyExtent>(); + synchronized (migrations) { + migrationsCopy.addAll(migrations.keySet()); + } + return migrationsCopy; --- End diff -- could return an immutable set
7. Github user ctubbsii commented on a diff in the pull request: https://github.com/apache/accumulo/pull/68#discussion_r52668001 --- Diff: server/master/src/main/java/org/apache/accumulo/master/Master.java --- @@ -1336,7 +1332,11 @@ public void assignedTablet(KeyExtent extent) { } @Override - public Collection<KeyExtent> migrations() { - return migrations.keySet(); + public Set<KeyExtent> migrations() { + Set<KeyExtent> migrationsCopy = new HashSet<KeyExtent>(); + synchronized (migrations) { + migrationsCopy.addAll(migrations.keySet()); + } + return migrationsCopy; --- End diff -- Seems to put an unnecessary restriction on how other tools might use it, though. Didn't think that was necessary.
8. Github user keith-turner commented on a diff in the pull request: https://github.com/apache/accumulo/pull/68#discussion_r52669078 --- Diff: server/master/src/main/java/org/apache/accumulo/master/Master.java --- @@ -1336,7 +1332,11 @@ public void assignedTablet(KeyExtent extent) { } @Override - public Collection<KeyExtent> migrations() { - return migrations.keySet(); + public Set<KeyExtent> migrations() { + Set<KeyExtent> migrationsCopy = new HashSet<KeyExtent>(); + synchronized (migrations) { + migrationsCopy.addAll(migrations.keySet()); + } + return migrationsCopy; --- End diff -- My thinking is that we don't expect the caller to modify it. Now that a copy is made, its ok if caller modifies. But if in the future the implementation uses a concurrent map and does not copy, would not want caller of method to modify. Also the previous implementation didn't expect caller to modify, just read.
9. Github user joshelser commented on a diff in the pull request: https://github.com/apache/accumulo/pull/68#discussion_r52669338 --- Diff: server/master/src/main/java/org/apache/accumulo/master/Master.java --- @@ -1336,7 +1332,11 @@ public void assignedTablet(KeyExtent extent) { } @Override - public Collection<KeyExtent> migrations() { - return migrations.keySet(); + public Set<KeyExtent> migrations() { + Set<KeyExtent> migrationsCopy = new HashSet<KeyExtent>(); + synchronized (migrations) { + migrationsCopy.addAll(migrations.keySet()); + } + return migrationsCopy; --- End diff -- +1
10. Github user ctubbsii commented on a diff in the pull request: https://github.com/apache/accumulo/pull/68#discussion_r52676712 --- Diff: server/master/src/main/java/org/apache/accumulo/master/Master.java --- @@ -1336,7 +1332,11 @@ public void assignedTablet(KeyExtent extent) { } @Override - public Collection<KeyExtent> migrations() { - return migrations.keySet(); + public Set<KeyExtent> migrations() { + Set<KeyExtent> migrationsCopy = new HashSet<KeyExtent>(); + synchronized (migrations) { + migrationsCopy.addAll(migrations.keySet()); + } + return migrationsCopy; --- End diff -- I was thinking a bigger risk would be that a user would accidentally introduce a RuntimeException (UnsupportedOperationException) because they tried to do something which worked against some implementations of the interface, but not others, and because wrapped immutability is not something which we can check at compile-time. It just seems an unnecessary restriction to me. It's internal-only code, so we can change it any way we like in the future... and I think the risk you're highlighting is lower than the risk of somebody accidentally calling an unsupported method... and it's also much easier to protect against at the time the implementation changes. If it changes from a protective copy to another implementation, I would think it'd be

obvious for the implementor to check to see if that breaks usage at that time. For now, I think it makes sense to treat the interface as "snapshot of migrations", and the protective copy (vs. another implementation) matches up to that interface semantics. We could change the interface method name to "migrationsSnapshot()" to be clear, though, and to avoid careless future impl changes which change that semantics.

11. Github user joshelser commented on a diff in the pull request: https://github.com/apache/accumulo/pull/68#discussion_r52678587 --- Diff: server/master/src/main/java/org/apache/accumulo/master/Master.java --- @@ -1336,7 +1332,11 @@ public void assignedTablet(KeyExtent extent) { } @Override - public Collection<KeyExtent> migrations() { - return migrations.keySet(); + public Set<KeyExtent> migrations() { + Set<KeyExtent> migrationsCopy = new HashSet<KeyExtent>(); + synchronized (migrations) { + migrationsCopy.addAll(migrations.keySet()); + } + return migrationsCopy; --- End diff -- > We could change the interface method name to "migrationsSnapshot()" to be clear, though, and to avoid careless future impl changes which change that semantics. :+1: I like this general approach. I think migrations are just one place that we follow this pattern and could benefit from more descriptive method names.
12. Github user keith-turner commented on the pull request: https://github.com/apache/accumulo/pull/68#issuecomment-183087366 looks good to me too. I don't have a strong opinion about the immutable set. It was a thought i had while looking at the code.
13. Github user joshelser commented on the pull request: https://github.com/apache/accumulo/pull/68#issuecomment-183091912 > I don't have a strong opinion about the immutable set. It was a thought i had while looking at the code. Likewise for me. I thought I remember seeing other immutable wrappers in the Master similar to your change, but there's nothing wrong with this as-is.
14. Github user keith-turner commented on a diff in the pull request: https://github.com/apache/accumulo/pull/68#discussion_r52682644 --- Diff: server/master/src/main/java/org/apache/accumulo/master/Master.java --- @@ -1336,7 +1332,11 @@ public void assignedTablet(KeyExtent extent) { } @Override - public Collection<KeyExtent> migrations() { - return migrations.keySet(); + public Set<KeyExtent> migrations() { + Set<KeyExtent> migrationsCopy = new HashSet<KeyExtent>(); + synchronized (migrations) { + migrationsCopy.addAll(migrations.keySet()); + } + return migrationsCopy; --- End diff -- > I was thinking a bigger risk would be that a user would accidentally introduce a RuntimeException My hope was that the runtime exception would possibly expose a bug (in addition to causing a bug). However, there are certainly cases where it would only cause problems and not expose any bugs. > wrapped immutability is not something which we can check at compile-time That would be nice to have.
15. Github user ctubbsii commented on a diff in the pull request: https://github.com/apache/accumulo/pull/68#discussion_r52684501 --- Diff: server/master/src/main/java/org/apache/accumulo/master/Master.java --- @@ -1336,7 +1332,11 @@ public void assignedTablet(KeyExtent extent) { } @Override - public Collection<KeyExtent> migrations() { - return migrations.keySet(); + public Set<KeyExtent> migrations() { + Set<KeyExtent> migrationsCopy = new HashSet<KeyExtent>(); + synchronized (migrations) { + migrationsCopy.addAll(migrations.keySet()); + } + return migrationsCopy; --- End diff -- I'll go ahead and do the immutability, as well as rename the method... but I'll add a Javadoc to clearly state that it's immutible. It looks like the other (test) implementations are already immutable anyway.
16. Github user asfgit closed the pull request at: https://github.com/apache/accumulo/pull/68
17. While technically, this bug only existed in the 1.6 branch, because it had already been fixed in newer branches, the changes to the interface and some cleanup was applied to all branches.