Item 216
git_comments:

1. parent subset will clear its cache in propagateCostImprovements0 method itself
2. ignore
3. Propagate cost improvement since this potentially would change the subset's best cost

git_commits:

1. **summary:** [CALCITE-2018] Complete work of calcite-2018 (Xiening Dai)
   message: [CALCITE-2018] Complete work of calcite-2018 (Xiening Dai) The change completes the fix of CALCITE-2018. Below are a few key additions - 1. When a Rel is renamed and it is subset's best RelNode, we need to propagate change improvement since the subset's best cost could be changed. 2. Add codes to handle CyclicMetadataException. Right now we just ignore it. We will revisit once we have figured out a better way to handle CyclicMetadataException. 3. Enable isValid() check under assert. Also check validation after RelSet merge.

github_issues:

github_issues_comments:

github_pulls:

github_pulls_comments:

github_pulls_reviews:

jira_issues:

1. **summary:** Queries failed with AssertionError: rel has lower cost than best cost of subset
   description: *Problem description* When rootLogger level is DEBUG, unit tests * MaterializationTest.testMaterializationSubstitution2 * MaterializationTest.testJoinMaterializationUKFK8 * MaterializationTest.testJoinMaterializationUKFK6 * JdbcTest.testWhereNot unit tests are failed with error AssertionError: rel has lower cost than best cost of subset. Full stack trace for test {{MaterializationTest.testMaterializationSubstitution2}}: {noformat} java.lang.AssertionError: rel [rel#245:EnumerableUnion.ENUMERABLE.[](input#0=rel#246:Subset#5.ENUMERABLE.[],input#1=rel#239:Subset#6.ENUMERABLE.[0],all=true)] has lower cost {14.0 rows, 19.0 cpu, 0.0 io} than best cost {15.0 rows, 20.0 cpu, 0.0 io} of subset [rel#243:Subset#7.ENUMERABLE.[]] at org.apache.calcite.plan.volcano.VolcanoPlanner.validate(VolcanoPlanner.java:906) at org.apache.calcite.plan.volcano.VolcanoPlanner.register(VolcanoPlanner.java:866) at org.apache.calcite.plan.volcano.VolcanoPlanner.ensureRegistered(VolcanoPlanner.java:883) at org.apache.calcite.plan.volcano.VolcanoPlanner.ensureRegistered(VolcanoPlanner.java:101) at org.apache.calcite.rel.AbstractRelNode.onRegister(AbstractRelNode.java:336) at org.apache.calcite.plan.volcano.VolcanoPlanner.registerImpl(VolcanoPlanner.java:1495) at org.apache.calcite.plan.volcano.VolcanoPlanner.register(VolcanoPlanner.java:863) at org.apache.calcite.plan.volcano.VolcanoPlanner.ensureRegistered(VolcanoPlanner.java:883) at org.apache.calcite.plan.volcano.VolcanoPlanner.ensureRegistered(VolcanoPlanner.java:1766) at org.apache.calcite.plan.volcano.VolcanoRuleCall.transformTo(VolcanoRuleCall.java:135) at org.apache.calcite.plan.RelOptRuleCall.transformTo(RelOptRuleCall.java:234) at org.apache.calcite.rel.rules.FilterProjectTransposeRule.onMatch(FilterProjectTransposeRule.java:143) at org.apache.calcite.plan.volcano.VolcanoRuleCall.onMatch(VolcanoRuleCall.java:212) at org.apache.calcite.plan.volcano.VolcanoPlanner.findBestExp(VolcanoPlanner.java:650) at org.apache.calcite.tools.Programs$5.run(Programs.java:326) at org.apache.calcite.tools.Programs$SequenceProgram.run(Programs.java:387) at org.apache.calcite.prepare.Prepare.optimize(Prepare.java:187) at org.apache.calcite.prepare.Prepare.prepareSql(Prepare.java:318) at org.apache.calcite.prepare.Prepare.prepareSql(Prepare.java:229) at org.apache.calcite.prepare.CalcitePrepareImpl.prepare2_(CalcitePrepareImpl.java:786) at org.apache.calcite.prepare.CalcitePrepareImpl.prepare_(CalcitePrepareImpl.java:640) at org.apache.calcite.prepare.CalcitePrepareImpl.prepareSql(CalcitePrepareImpl.java:610) at org.apache.calcite.schema.Schemas.prepare(Schemas.java:346) at org.apache.calcite.materialize.MaterializationService$DefaultTableFactory.createTable(MaterializationService.java:374) at org.apache.calcite.materialize.MaterializationService.defineMaterialization(MaterializationService.java:137) at org.apache.calcite.materialize.MaterializationService.defineMaterialization(MaterializationService.java:99) at org.apache.calcite.schema.impl.MaterializedViewTable$MaterializedViewTableMacro.<init>(MaterializedViewTable.java:110) at org.apache.calcite.schema.impl.MaterializedViewTable$MaterializedViewTableMacro.<init>(MaterializedViewTable.java:100) at org.apache.calcite.schema.impl.MaterializedViewTable.create(MaterializedViewTable.java:81) at org.apache.calcite.model.ModelHandler.visit(ModelHandler.java:364) at org.apache.calcite.model.JsonMaterialization.accept(JsonMaterialization.java:42) at org.apache.calcite.model.JsonSchema.visitChildren(JsonSchema.java:98) at org.apache.calcite.model.JsonMapSchema.visitChildren(JsonMapSchema.java:48) at org.apache.calcite.model.ModelHandler.populateSchema(ModelHandler.java:257) at org.apache.calcite.model.ModelHandler.visit(ModelHandler.java:273) at org.apache.calcite.model.JsonCustomSchema.accept(JsonCustomSchema.java:45) at org.apache.calcite.model.ModelHandler.visit(ModelHandler.java:196) at org.apache.calcite.model.ModelHandler.<init>(ModelHandler.java:88) at org.apache.calcite.jdbc.Driver$1.onConnectionInit(Driver.java:104) at org.apache.calcite.avatica.UnregisteredDriver.connect(UnregisteredDriver.java:139) at java.sql.DriverManager.getConnection(DriverManager.java:571) at java.sql.DriverManager.getConnection(DriverManager.java:187) at org.apache.calcite.test.CalciteAssert$MapConnectionFactory.createConnection(CalciteAssert.java:1227) at org.apache.calcite.test.CalciteAssert$AssertQuery.createConnection(CalciteAssert.java:1266) at org.apache.calcite.test.CalciteAssert$AssertQuery.returns(CalciteAssert.java:1337) at org.apache.calcite.test.CalciteAssert$AssertQuery.returns(CalciteAssert.java:1320) at org.apache.calcite.test.CalciteAssert$AssertQuery.sameResultWithMaterializationsDisabled(CalciteAssert.java:1548) at org.apache.calcite.test.MaterializationTest.testMaterializationSubstitution2(MaterializationTest.java:2062) at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method) at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:57) at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43) at java.lang.reflect.Method.invoke(Method.java:606) at org.junit.runners.model.FrameworkMethod$1.runReflectiveCall(FrameworkMethod.java:50) at org.junit.internal.runners.model.ReflectiveCallable.run(ReflectiveCallable.java:12) at org.junit.runners.model.FrameworkMethod.invokeExplosively(FrameworkMethod.java:47) at org.junit.internal.runners.statements.InvokeMethod.evaluate(InvokeMethod.java:17) at org.junit.runners.ParentRunner.runLeaf(ParentRunner.java:325) at org.junit.runners.BlockJUnit4ClassRunner.runChild(BlockJUnit4ClassRunner.java:78) at org.junit.runners.BlockJUnit4ClassRunner.runChild(BlockJUnit4ClassRunner.java:57) at org.junit.runners.ParentRunner$3.run(ParentRunner.java:290) at org.junit.runners.ParentRunner$1.schedule(ParentRunner.java:71) at org.junit.runners.ParentRunner.runChildren(ParentRunner.java:288) at org.junit.runners.ParentRunner.access$000(ParentRunner.java:58) at org.junit.runners.ParentRunner$2.evaluate(ParentRunner.java:268) at org.junit.runners.ParentRunner.run(ParentRunner.java:363) at org.junit.runner.JUnitCore.run(JUnitCore.java:137) at com.intellij.junit4.JUnit4IdeaTestRunner.startRunnerWithArgs(JUnit4IdeaTestRunner.java:68) at com.intellij.rt.execution.junit.IdeaTestRunner$Repeater.startRunnerWithArgs(IdeaTestRunner.java:47) at com.intellij.rt.execution.junit.JUnitStarter.prepareStreamsAndStart(JUnitStarter.java:242) at com.intellij.rt.execution.junit.JUnitStarter.main(JUnitStarter.java:70) {noformat} *Possible root cause* JaninoRelMetadataProvider caches metadata queries of {{RelSubset}} even when {{RelSubset.best}} value is not set. Actually not all RelNodes in {{RelSubset.set.rels}} have the same row count for some queries. When {{RelMetadataQuery.getRowCount(RelNode rel)}} method is called with {{RelSubset}} instance without best value, row count of {{RelSubset.set.rel}} is cached. After assigning best relNode, the {{RelMetadataQuery.getRowCount(RelNode rel)}} method returns old cached value if {{RelMetadataQuery}} instance still the same. This error appears when

row count for best relNode is different from set.rel relNode. In the test {{MaterializationTest.testMaterializationSubstitution2}} row count of {{rel#237:LogicalFilter}} was cached and returned its value when relSubset had best {{rel#347:EnumerableFilter}}.

jira_issues_comments:

1. Can you clarify what you mean by "debug is enabled"?
2. To fix this issue, I think, we should avoid JaninoRelMetadataProvider queries caching for RelSubset instances when best RelNode is not set and update cached values every time when best value is updated.
3. When set log4j.rootLogger=DEBUG in log4j.properties
4. Could you please confirm if the approach that I am proposed is acceptable, and then I would like to start to prepare the fix.
5. I have created pull request for this issue: https://github.com/apache/calcite/pull/552. [~julianhyde], could you please take a look?
6. Adding two fields to RelNode -- and mutable ones at that -- is likely to add both cost and complexity to the volcano planner algorithm. I am nervous of going there. Let's back up a bit. Why are costs changing during a RelMetadataQuery instance? The assumption is that RelMetadataQuery is so short-lived that no cost changes occur while it is alive. How is that assumption being broken?
7. Let's use {{MaterializationTest.testMaterializationSubstitution2()}} test as an example. One of the RenNodes whose old value was gotten from the cache is {{rel#246:Subset}} A problem with old cached value appears for the first time (I can be wrong when saying that it is for the first time, but in this case, it also happens) when RelNode {{rel#246}} has best {{rel#279:EnumerableFilter}}. When RelNode with smaller cost than bestCost (in our case it is {{rel#347:EnumerableFilter}}) is passed into the {{RelSubset.propagateCostImprovements0()}} method, the current cost is calculated for the first time. At this point old value is cached. In [this line|https://github.com/apache/calcite/blob/56d5261abb893dd92b0a2cc4893292876a2b7880/core/src/main/java/org/apache/calcite/plan/volcano/RelSubset.java#L34 method {{propagateCostImprovements()}} is called on the parent subset {{rel#243:Subset}}. Again in the method {{propagateCostImprovements0()}} but called on the parent subset {{rel#243}}, is called {{planner.getCost(rel, mq)}}, where rel is parent RelNode for our {{rel#246}}: {{rel#245:EnumerableUnion}}. But when calculating {{planner.getCost(rel, mq)}} cached value for {{rel#246}} is gotten and used for futher calculations. All these things happen when the same {{RelMetadataQuery}} instance is used.
8. [~julianhyde], does my explanation make sense? Or maybe you can suggest some other approach to fix this issue?
9. Sorry for the delay - I'm traveling. I will review and give feedback when I return.
10. I reviewed it again. I am still really scared of the impact of adding mutable state to RelNode. Especially since we have been making a concerted effort over several years to make it immutable (e.g. getting rid of calls to RelNode.setInput). I don't think we should move forward with this fix. I confess I still haven't had time to grok your test case and try to devise another solution. Sorry. Can some other folks please chime in?
11. I have just faced up with the more serious consequence of this issue: in the query, where 5 tables are joining, {{JoinToMultiJoinRule}} could not be applied correctly, since instead of one of the joins, project with that join was specified. This project actually also had greater cost in its subset than join rel node, but because of this bug, project rel node was chosen as the best. In fact, RelNode is not immutable, especially when it has RelSubset as an input. We could not cache RelNode metadata blindly expecting that it does not change. It's a matter of time before more serious consequences would be discovered.
12. I agree that this is a serious problem. It's not easy to solve because, as you say, the statistics and RelNode graph are constantly changing. The current strategy is to use clock ticks (one tick each time a planner rule fires and produces a result), to flush the cache on the clock tick, and assume that it is constant for the duration of the tick. I believe that if we tried to explicitly propagate improvements up through the graph we would hit performance problems, correctness problems, and cycles. Of course there is some mutable state in the RelNode graph. (Otherwise we'd have to copy virtually the entire graph each time a rule fired.) But that doesn't mean that adding more mutable state is the solution.
13. [~volodymyr], is this change going to impact all metadata providers? Some rules extract information from the underlying query plan via metadata providers. If this change clears the cache every time we the planner infers an equivalent expression, we might have some performance degradation. It seems to me that at least we should have two different routes depending on the nature of the metadata: for instance, while cost can change for two equivalent expressions, other metadata information such as data size or cardinality should not.
14. [~jcamachorodriguez], considering rels in RelSubset, I observed cases where such metadata as row count is different for few rels from the same RelSet. So I don't think that we can be sure that some kind of metadata left unchanged. Currently, I am reworking this fix. Main points that already done: 0) Left the same metadata cache structure as it was in my original PR. It allows finding and deleting all cached values for the same RelNode easier. 1) Made changes to use the same {{RelOptCluster}} instance for RelNodes, so it makes possible to use single {{RelMetadataQuery}} instance between the first call of {{RelOptCluster.getMetadataQuery()}} and {{RelOptCluster.invalidateMetadataQuery()}} call. 2) Cached metadata values are removed for current {{RelSubset}} and all parent {{RelNode}} s when {{RelSubset.propagateCostImprovements()}} is called and best rel node was changed. 3) Cached metadata values are removed for current {{RelNode}} when its input was changed. 4) Made changes in {{RelSet.mergeWith()}} method to call {{RelSubset.propagateCostImprovements()}} when resulting best is known instead of just assigning {{RelSubset.best}}. It is needed for the cases when {{getOrCreateSubset()}} method returns existing {{RelSubset}}, so we need to inform its parent rels that its best rel node was changed. (Currently, only otherSubset parents are informed.)
15. [~julianhyde], I have reworked my pull request: https://github.com/apache/calcite/pull/552. Could you please take a look again?
16. [~volodymyr], I see that the new pull request is quite significantly different from your previous one. I am pleased that you dropped the idea of RelNode.version; I think that would have caused problems, not to mention increasing memory use. In your new PR, attaching RelMetadataQuery to RelOptCluster is a radical change, because now we need to keep RelMetadataQuery consistent over a longer life-cycle; I am trying to decide whether I like it. I can well believe that you found bugs in {{RelSet.mergeWith()}} and {{RelSubset.propagateCostImprovements()}}. I will add more comments when I have had more chance to review/experiment/think.
17. [~julianhyde], thanks for looking into this. In order to preserve actual data in the cache, I used versioning for rel nodes in the previous version of my pull request. It allowed handling the case of using several RelMetadataQuery instances for the caching at the same time. But the code in the actual pull request assumes that the same RelMetadataQuery instance is used at the same life-cycle. Therefore, I had to make changes to pass existing RelOptCluster and attach RelMetadataQuery to RelOptCluster. If a problem connected with longer life-cycle of RelMetadataQuery will appear, it may be fixed by calling {{RelOptCluster.invalidateMetadataQuery()}} at one more place. Since previously was used several RelMetadataQuery instances, there also was another problem with longer life-cycle of RelMetadataQuery.
18. [~julianhyde], did you have a chance to take a look? Are there any other questions/changes I should address to merge this change?
19. I am still nervous about this change, because you have increased the lifetime of {{RelMetadataQuery}}, and therefore there is more opportunity for the metadata to go stale. I have thought of something that would help: can you add some code that walks over the metadata and checks that it is valid? (E.g. check that the cheapest rel in the subset really has lowest cost, and that the cached selectivity of an expression is that same that would be calculated if we did it again.) This method does not need be efficient - it would only be enabled in debug mode - but it will give us confidence that the change propagation is working correctly. If you do this I will commit.
20. [~julianhyde], I tried to add these checks for cached and uncached metadata values, and all tests in Calcite are passed. But when I ran Drill unit tests with this check, some of them failed. This failure appeared because *the cumulative cost of best rel node increased*. For example when we have such piece of the plan: {noformat} SingleMergeExchangePrel(sort0=[1 DESC]): rowcount = 60.175, cumulative cost = {60.175 rows, 481.4 cpu, 0.0 io, 492953.6 network, 0.0 memory}, id = 2232 SortPrel(subset=[rel#2228:Subset#201.PHYSICAL.HASH_DISTRIBUTED([[$1]]).[1 DESC]], sort0=[$1], dir0=[DESC]): rowcount = 60.175, cumulative cost = {60.175 rows, 1422.7999242130231 cpu, 0.0 io, 0.0 network, 962.8 memory}, id = 2896 HashToRandomExchangePrel(subset= [rel#2885:Subset#201.PHYSICAL.HASH_DISTRIBUTED([[$1]]).[]], dist0=[[$1]]): rowcount = 60.175, cumulative cost = {60.175 rows, 962.8 cpu, 0.0 io, 492953.6 network, 0.0 memory}, id = 2918 StreamAggPrel(subset=[rel#2900:Subset#201.PHYSICAL.HASH_DISTRIBUTED([[$0]]).[]], group=[{0}], revenue= [SUM($1)]): rowcount = 60.175, cumulative cost = {601.75 rows, 9628.0 cpu, 0.0 io, 0.0 network, 0.0 memory}, id = 2215 HashToMergeExchangePrel(subset= [rel#2214:Subset#204.PHYSICAL.HASH_DISTRIBUTED([[$0]]).[0]]): rowcount = 601.75, cumulative cost = {601.75 rows, 9628.0 cpu, 0.0 io, 4929536.0 network, 0.0 memory}, id = 2213 StreamAggPrel(subset=[rel#2205:Subset#201.PHYSICAL.HASH_DISTRIBUTED([[$0]]).[0]], group=[{0}], revenue= [SUM($1)]): rowcount = 601.75, cumulative cost = {6017.5 rows, 96280.0 cpu, 0.0 io, 0.0 network, 0.0 memory}, id = 2472 {noformat} rel node {{SingleMergeExchangePrel}} has a smaller cumulative cost than the cumulative cost of the same rel node after applying some rules: {noformat} SingleMergeExchangePrel(sort0=[1 DESC]): rowcount = 601.75, cumulative cost = {601.75 rows, 4814.0 cpu, 0.0 io, 4929536.0 network, 0.0 memory}, id = 2232 SortPrel(subset=[rel#2228:Subset#201.PHYSICAL.HASH_DISTRIBUTED([[$1]]).[1 DESC]], sort0=[$1], dir0=[DESC]): rowcount = 601.75, cumulative cost = {601.75 rows, 22223.88016652411 cpu, 0.0 io, 0.0 network, 9628.0 memory}, id = 2896 HashToRandomExchangePrel(subset= [rel#2885:Subset#201.PHYSICAL.HASH_DISTRIBUTED([[$1]]).[]], dist0=[[$1]]): rowcount = 601.75, cumulative cost = {601.75 rows, 9628.0 cpu, 0.0 io, 4929536.0 network, 0.0 memory}, id = 2884 StreamAggPrel(subset=[rel#2205:Subset#201.PHYSICAL.HASH_DISTRIBUTED([[$0]]).[0]], group=[{0}], revenue=[SUM($1)]): rowcount = 601.75, cumulative cost = {6017.5 rows, 96280.0 cpu, 0.0 io, 0.0 network, 0.0 memory}, id = 2472 {noformat} But its cost was changed because the cumulative cost of {{SortPrel}} from the second plan is less than the cost of the same rel node from the first plan. Perhaps we should change

a little bit cost estimation in Drill, but I am not sure that it will help to prevent other similar cases. Or we should try to modify Volcano planner to be able to revert the change of best rel nodes when the cumulative cost of neither of top rel nodes wasn't decreased. Or we may reflect the increase in the cumulative cost of best rel node in bestCost. [~julianhyde], [~amansinha100], could you please advise me the better way to fix this issue?

21. I have added a check for stored {{RelSubset.bestCost}} into {{VolcanoPlanner.validate()}} method (line 907): {code:java} if (subset.best != null) { RelOptCost bestCost = getCost(subset.best, RelMetadataQuery.instance()); if (!subset.bestCost.equals(bestCost)) { throw new AssertionError("relSubset [" + subset.getDescription() + "] has wrong best cost " + subset.bestCost + ". Correct cost is " + bestCost); } } {code} and Calcite unit tests failed: {noformat} Failed tests: MaterializationTest.testJoinMaterializationUKFK9:1823->checkMaterialize:198->checkMaterialize:205->checkThatMaterialize:233 relSubset [rel#226287:Subset#8.ENUMERABLE.[]] has wrong best cost {221.5 rows, 128.25 cpu, 0.0 io}. Correct cost is {233.0 rows, 178.0 cpu, 0.0 io} ScannableTableTest.testPFPushDownProjectFilterAggregateNested:279 relSubset [rel#12950:Subset#5.ENUMERABLE.[]] has wrong best cost {63.8 rows, 62.308 cpu, 0.0 io}. Correct cost is {70.4 rows, 60.404 cpu, 0.0 io} ScannableTableTest.testPFTableRefusesFilterCooperative:221 relSubset [rel#13382:Subset#2.ENUMERABLE.[]] has wrong best cost {81.0 rows, 181.01 cpu, 0.0 io}. Correct cost is {150.5 rows, 250.505 cpu, 0.0 io} ScannableTableTest.testProjectableFilterableCooperative:148 relSubset [rel#13611:Subset#2.ENUMERABLE.[]] has wrong best cost {81.0 rows, 181.01 cpu, 0.0 io}. Correct cost is {150.5 rows, 250.505 cpu, 0.0 io} ScannableTableTest.testProjectableFilterableNonCooperative:165 relSubset [rel#13754:Subset#2.ENUMERABLE.[]] has wrong best cost {81.0 rows, 181.01 cpu, 0.0 io}. Correct cost is {150.5 rows, 250.505 cpu, 0.0 io} FrameworksTest.testUpdate:336->executeQuery:367 relSubset [rel#22533:Subset#2.ENUMERABLE.any] has wrong best cost {19.5 rows, 37.75 cpu, 0.0 io}. Correct cost is {22.575 rows, 52.58 cpu, 0.0 io} {noformat} The root cause is the same as for the test failure in the comment above - the cumulative cost of best rel node increased. To ensure that this failure was not caused by incorrect metadata caching, I made changes in {{RelOptCluster.getMetadataQuery()}} method so it always returns {{RelMetadataQuery.instance()}}, therefore cached metadata isn't used. [~julianhyde], since this issue does not connect with metadata caching, may we merge existing pull request without those additional checks, and create a separate Jira to fix the issue described in these two last comments? Regarding this fix, we have used it in Calcite fork which is used by Drill and it allowed executing of several queries that failed with {{CannotPlanException}} before. Also, all unit and functional Drill tests are passed with enabled check {{VolcanoPlanner.validate()}} (without adding these additional checks that we mentioned).
22. Logging another JIRA sounds like a good idea, and you should do it. However I'm not sure yet whether I would commit this fix before we have a fix for the other issue. Because to do so would be to move from a situation where metadata is broken to a situation where metadata is still broken but in a different way. I can't tell whether we are making things better or worse.
23. [~volodymyr] [~julianhyde] Any progress on this issue? I think it is a serious bug and we need to fix this as quickly as possible, if no one take this issue, i will take and rework it.
24. [~danny0405], thanks for moving this issue forward. This issue partially solves the problem with propagating best cost including the fixes for preserving only actual `RelMetadataQuery`. But it is blocked by a more serious problem - CALCITE-2166 where was shown that the cumulative cost of best rel node actually is not always the best cost, the problem there with the way how the best cost is calculated. PS. I have rebased my PR onto the master and resolved all the conflicts. Also, I don't mind fixing this Jira either with the fix I proposed or any alternative fix, the main thing is to resolve not fewer problematic cases compared to the original PR.
25. [~volodymyr] Ok, i will try to fix CALCITE-2166 and then hope this PR can be merged.
26. I also run into this issue recently. I haven't looked into [~volodymyr]'s fix in details. But there's one simple thing we can do which is to add an invalidateMetadataQuery() call just before RuleQueue.dump() returns. RuleQueue.dump() calculates and caches RelNode cost while printing out the trace. Then the subsequent rule firing doesn't get chance to recompute RelNode cost since it's already in the cache.
27. I saw the issue just logged for this workaround. But it doesn’t make clear that it is just a workaround, nor state its relationship to this, the issue that describes the root cause. Can you make sure that the other issue is in proper context.
28. Hi [~julianhyde], [~volodymyr] also has some questions. We can continue the discussion under CALCITE-3257. Once we agree on the problem, I would update/clarify the CL. Thanks.
29. [~xndai] I see that you are proposing in the git conversation to turn on validation for the test suite. That would mean that the test suite is running a different path than the production code, and I am against the idea. The fact that you are running into cyclic metadata errors and choosing to ignore them is another worrying sign. By all means enable validation in a "long" test suite -- we've talked many times about a "long test suite" on the dev list, but I don't recall whether we implemented one -- but not the regular one.
30. [~julianhyde] I do agree that ignoring cyclic metadata exception is worrying. But why turning on validation will run a different path than the production code? I see if the assertion is enabled, it just adds additional check isValid(), just like any other existing assertion checking in Calcite.
31. The cyclic metadata exception is due to the change of memo that's caused by the fix of CALCITE-2018. Actually this is a separate issue and we saw it in different queries even before the fix. I don't think it is caused by the validation. In the validation step, I choose to create a new RelMetadataQuery object each time so it doesn't affect the cache of volcano planner in any way. The only side effect I can see is the test running time could be extended due to the extra validation steps (which is not cheap). But I think it worths it as it catches any correctness and consistency issues of memo right the way.
32. [~julianhyde][~volodymyr] Do you have any other feedback regarding the fix? If you feel strong against enabling validation in test run, I can remove that change. Can we also review the cost propogation part, and move forward with this issue?
33. With a dynamic programming algorithm, extra validation may cause extra things to be computed, and therefore can change behavior. So let's disable the extra validation. As I said, you can have an additional test (as part of the long suite) that runs with extra validation switched on. As long as we keep the test with validation switched off. Assertions enabled vs. disabled ({{-ea}} vs {{-da}}) is not the point. We expect the test suite to pass with either. Anyway. The big change here is that we get {{RelMetadataQuery}} from the cluster, rather than from the thread. Can you explain why that change was necessary? If we had been more disciplined to close the {{RelMetadataQuery}} at the end of preparation, would that have had the same effect? Is {{RelMetadataQuery.instance()}} now obsolete? If not, what are the circumstances when it is OK to call it?
34. Minor point: you have changed the parameters of {{CalciteMaterializer}} constructor from {{(CalcitePrepareImpl, CalcitePrepare.Context, CatalogReader, CalciteSchema, RelOptPlanner, SqlRexConvertletTable)}} to {{(CalcitePrepareImpl, CalcitePrepare.Context, CatalogReader, CalciteSchema, RelOptPlanner,SqlRexConvertletTable, RelOptCluster)}}. Can you swap the last 2 arguments so that {{cluster}} is now in the same place that {{planner}} used to be.
35. I can disable the validation in test. But I would also love to know what are the extra things that are computed and affect the behavior. As I examine the code path, the only stateful object updated in the validation pass is the RelMetadataQuery.map. But since the RelMetadataQuery object used in validation is a complete new object, that would not affect planner's behavior IMO. The change with RelMetadataQuery is to make sure we have one single RelMetadataQuery instance during the rule firing, so we can avoid the overhead of instantiation of RelMetadataQuery and reuse the RelNode cost cache (RelMetadataQuery.map). For this fix, I think the bigger change is to make sure we clean RelNode cost cache entries when best rel is updated. In order to remove a single cache entry, the cache map structure is also updated to include RelNode as a key. Regarding the minor point, this part was authored by [~volodymyr] But I can made an update to swap the two arguments. Thanks.
36. I pushed an update to https://github.com/xndai/calcite/tree/2018 please take a look.
37. Hi [~volodymyr], are you still working on this issue? Are you ok if I open a new pull request so we can move forward? Thanks.
38. Hi [~xndai], I'll rebase my branch, add commit with your changes and take a look at the comments in Jira at the end of the week.
39. [~volodymyr] Any updates on this? The issue is affecting our production and we'd like to get it into the next release so we can pick it up. Also the fix of CALCITE-3446 might also has an effect on this, so we want to have them integrate and test earlier.
40. [~xndai], sorry for the delay, I have rebased my branch onto the master, added your changes and added one more minor change connected with swapping arguments. Also, it was surprising, that some changes from my PR were merged into master as CALCITE-3403 (original commit before rebase may be found in your branch: [https://github.com/apache/calcite/compare/master...xndai:2018] and merged into the master commit: [https://github.com/apache/calcite/commit/3cbbafa941128dc5097c2a26711f5751f764e12d]
41. Hi [~volodymyr] I don't know about the background of 3403. Reusing the RelMetadataQuery seems quite reasonable. That's probably someone else would come up with the same fix. Thanks.
42. Fixed in [e0d869a|https://github.com/apache/calcite/commit/e0d869ab8e540dec71b9278f5a3202df4a2dfb5c], [d3b1d85|https://github.com/apache/calcite/commit/d3b1d857380972b7ca7fa5b70f9903977054594c], [ad90589|https://github.com/apache/calcite/commit/ad9058999bdea1329dba2267604512d8e199919f], thanks for the great work [~volodymyr] and [~xndai] ! And thanks all the colleagues participate in pushing this forward !
43. This change breaks unit tests with DEBUG enabled. Here's a link to test failure: https://github.com/apache/calcite/commit/51309bfaff2f283bde94d439c3e2b8094417e5c0/checks?check_suite_id=308601679#step:4:2409 Here's stacktrace: {noformat} java.lang.NullPointerException at org.apache.calcite.plan.volcano.VolcanoPlanner.isValid(VolcanoPlanner.java:880) at org.apache.calcite.plan.volcano.VolcanoPlanner.ensureRegistered(VolcanoPlanner.java:870) at org.apache.calcite.plan.volcano.VolcanoPlanner.ensureRegistered(VolcanoPlanner.java:92) at

org.apache.calcite.rel.AbstractRelNode.onRegister(AbstractRelNode.java:321) at org.apache.calcite.plan.volcano.VolcanoPlanner.registerImpl(VolcanoPlanner.java:1701) at org.apache.calcite.plan.volcano.VolcanoPlanner.setRoot(VolcanoPlanner.java:298) at org.apache.calcite.tools.Programs.lambda$standard$3(Programs.java:269) {noformat} Explanation: VolcanoPlanner.setRoot is called. That is "root" is still null setRoot(VolcanoPlanner.java:298) is {code:java} this.root = registerImpl(rel, null); {code} ... then, at the end of ensureRegistered there's {code:java} if (LOGGER.isDebugEnabled()) { assert isValid(Litmus.THROW); // this gets executed before this.root was ever executed } {code} then it fails with NPE since planner.root is still {{null}}

44. [~vladimirsitnikov], this change was done in CALCITE-3487, before that, {{RelMetadataQuery metaQuery}} was {{RelMetadataQuery.instance()}}.
45. Ah, thanks for the analysis. In practice, it is "fun" how the bug was discovered only due to Gradle executing tests with "debug by default"
46. Resolved in release 1.22.0 (2020-03-05).