Item 150
git_comments:

git_commits:

1. **summary:** [SPARK-9876][SQL][FOLLOWUP] Enable string and binary tests for Parquet predicate pushdown and replace deprecated fromByteArray.
   message: [SPARK-9876][SQL][FOLLOWUP] Enable string and binary tests for Parquet predicate pushdown and replace deprecated fromByteArray. ## What changes were proposed in this pull request? It seems Parquet has been upgraded to 1.8.1 by https://github.com/apache/spark/pull/13280. So, this PR enables string and binary predicate push down which was disabled due to [SPARK-11153](https://issues.apache.org/jira/browse/SPARK-11153) and [PARQUET-251](https://issues.apache.org/jira/browse/PARQUET-251) and cleans up some comments unremoved (I think by mistake). This PR also replace the API, `fromByteArray()` deprecated in [PARQUET-251](https://issues.apache.org/jira/browse/PARQUET-251). ## How was this patch tested? Unit tests in `ParquetFilters` Author: hyukjinkwon <gurwls223@gmail.com> Closes #13389 from HyukjinKwon/parquet-1.8-followup.

github_issues:

github_issues_comments:

github_pulls:

1. **title:** [SPARK-9876][SQL][FOLLOWUP] Enable string and binary tests for Parquet predicate pushdown and replace deprecated fromByteArray.
   body: ## What changes were proposed in this pull request? It seems Parquet has been upgraded to 1.8.1 by https://github.com/apache/spark/pull/13280. So, this PR enables string and binary predicate push down which was disabled due to [SPARK-11153](https://issues.apache.org/jira/browse/SPARK-11153) and [PARQUET-251](https://issues.apache.org/jira/browse/PARQUET-251) and cleans up some comments unremoved (I think by mistake). This PR also replace the API, `fromByteArray()` deprecated in [PARQUET-251](https://issues.apache.org/jira/browse/PARQUET-251). ## How was this patch tested? Unit tests in `ParquetFilters`

github_pulls_comments:

1. **[Test build #59580 has finished](https://amplab.cs.berkeley.edu/jenkins/job/SparkPullRequestBuilder/59580/consoleFull)** for PR 13389 at commit [`04f24e3`](https://github.com/apache/spark/commit/04f24e31b8ec536ac57eec789292def7a3960633). - This patch passes all tests. - This patch merges cleanly. - This patch adds no public classes.
2. Hi @rdblue @liancheng , Could you please take a look?
3. +1 overall, good catch on those tests.
4. @rdblue Thank you for the review! I just noticed `fromByteArray` was deprecated so I replaced them to the new ones. Do you mind if I ask a quick look for this again? I saw this was reviewed by you in `parquet-mr`.
5. **[Test build #60124 has finished](https://amplab.cs.berkeley.edu/jenkins/job/SparkPullRequestBuilder/60124/consoleFull)** for PR 13389 at commit [`5611978`](https://github.com/apache/spark/commit/5611978e03f0d1a3f5e2481c07d28a645e2751d2). - This patch passes all tests. - This patch merges cleanly. - This patch adds no public classes.
6. @rdblue @liancheng Could you please take another look?
7. Looks fine other than one comment.
8. LGTM, merging to master. Thanks!

github_pulls_reviews:

1. Nit: Lots of unnecessary white-space changes. I don't generally like to commit these, but it's fine if this is more in line with the project's published style guidelines.
2. If you're converting from a `String`, you can use `Binary.fromString`, which sets the reuse flag correctly. It looks like this should be using the "constant" variant, which signals to Parquet that the underlying bytes won't be changed by the program after they have been passed to Parquet.
3. Thank you for your review! (Actually it is `UTF8String`. So, it has to be converted into `String` to use `Binary.fromString`).. though.. I am a bit worried that it might possibly be reused in the future (although I think it is not reused for now). This can write corrupt statistics if this is reused.. Is my understanding correct?
4. `UTF8String` itself is immutable, but the underlying buffer it points to can be mutable. I'd vote for using `Binary.fromReusedByteArray` here.

jira_issues:

1. **summary:** Binary column statistics error when reuse byte[] among rows
   description: I think it is a common practice when inserting table data as parquet file, one would always reuse the same object among rows, and if a column is byte[] of fixed length, the byte[] would also be reused. If I use ByteArrayBackedBinary for my byte[], the bug occurs: All of the row groups created by a single task would have the same max & min binary value, just as the last row's binary content. The reason is BinaryStatistic just keep max & min as parquet.io.api.Binary references, since I use ByteArrayBackedBinary for byte[], the real content of max & min would always point to the reused byte[], therefore the latest row's content. Does parquet declare somewhere that the user shouldn't reuse byte[] for Binary type? If it doesn't, I think it's a bug and can be reproduced by [Spark SQL's RowWriteSupport |https://github.com/apache/spark/blob/master/sql/core/src/main/scala/org/apache/spark/sql/parquet/ParquetTableSupport.scala#L353-354] The related Spark JIRA ticket: [SPARK-6859|https://issues.apache.org/jira/browse/SPARK-6859]

jira_issues_comments:

1. It sounds like we need to add a defensive copy to make sure the internal min and max byte arrays aren't changed, and work out a way to ignore min/max stats for all existing data.
2. We have a fix for this already in our pull request (49) for PARQUET-77
3. [~jnadeau], does that include the fix to ignore min/max byte arrays for data written with 1.6.0 and earlier?
4. [~jnadeau], I checked the PR, It seems you just change {code} public void updateStats(Binary min_value, Binary max_value) { if (min.compareTo(min_value) > 0) { min = min_value; } if (max.compareTo(max_value) < 0) { max = max_value; } } {code} to {code} public void updateStats(Binary min_value, Binary max_value) { if (min.compareTo(min_value) > 0) { min = Binary.fromByteArray(min_value.getBytes()); } if (max.compareTo(max_value) < 0) { max = Binary.fromByteArray(max_value.getBytes()); } } {code} As I mentioned in the description, the bug occurs when I reuse byte[], using ByteArrayBackedBinary as Binary, fromByteArray & getBytes are not really doing the "defensive copy" job mentioned by [~rdblue], it's still passing the original byte[]. Correct me if I misunderstand your PR :)
5. [~yijieshen] It looks like the issue was only solved in the PR for the case where the new ByteBuffer backed implementation returned by the fromByteBuffer() method in the Binary.java class is used (or others that make a copy themselves). For most implementations of Binary it looks to be making a copy to return a byte[] from the getBytes() method. It looks like a more general approach will be needed if the Binary object may return an array it plans on re-using. That being said, as there is no way to resize an array, it would seem like this getBytes() method might be best to be defined to be required to return a copy, unless the Binary object is immutable or you are only sharing the byte[] where the values are of equal length. This seems to be the entire purpose for having the interface in the first place, which allows a buffer of a large length to stay allocated for all values and have the length set appropriately with each new value. If this was enforced at the Binary interface level than I think the PR would solve the functional issue. It does however create the unfortunate overhead of making a ton of copies if a column is full of increasing or decreasing values, it might be worth processing a small list of values to see if it finds a new min/max and make one defensive copy for that group.
6. I think a Binary is supposed to just be a unifying API over byte[] and various other byte[]-like types (slices of byte arrays, byte buffers, and so on). So I think that should come with the same caveats as sharing a byte[] -- if someone gives you a byte[] you might need to make a defensive copy, as in this case for the statistics. But in cases where you just use the Binary to write to a stream, no need for a copy. I guess what I'm saying is, it's not Binary that should be responsible for handling shared state, but rather the user of it, right?
7. For a design standpoint, I completely agree that it would be clearest to have the Binary objects maintain zero shared state. I think my concern is that the interface that is trying to unify byte[] with other types might be ill fit to have a method that returns a bare byte[] without an explicit expectation that it be an immutable copy of the data being currently represented. An implementation that re-uses a buffer will not be able to return its byte[] alone if the buffer has a chance of being allocated to the wrong size for the current value, so it would need to make a copy anyway. As far as writing to a stream is concerned, this is actually already covered by methods on Binary itself, which can write the current value to a stream with whatever method they need, specific to how the bytes are stored. I think adding interfaces for any other use cases is the cleanest way to do it. For example how the statistics want to use the data, it might make the most sense for the binary to include a clone method explicitly, that will create a copy of the binary in the most efficient manner possible. For Drill it would be useful if we could have the flexibility to return a new Binary backed by a direct memory buffer that we can track with our own allocator, for another type of binary, maybe one that internally has some amount of compression on individual values, would return a more efficient copy of itself than a copy that could be made of the byte[] returned from this method.
8. Yes, my point was that Binary is just a wrapper, and the decision to copy / not copy seems like it should be handled outside the wrapper. So maybe all that is needed is to add a helper method to Binary called copyToByteArray()? Though toByteArray() usually make a copy, so maybe we should just make that the contract? Or just some consistency around ownership? If you pass a byte[] to a Binary, should we say the contract is that ownership has passed to the Binary and you shouldn't mutate it any more?
9. Hey guys, I am planning to take a stab on this. After following the detailed discussion so far, I think (feel free to correct me) following needs to be done. 1. We need a clone() or copyToByteArray() method in Binary. Having the method should make it obvious to the user that the byte[] should not be mutated and if it plans on doing it clone() or copyToByteArray() should be used. However, this should be explicitly mentioned in the docs as well. 2. Update BinaryStatistics to use clone() or copyToByteArray() on the Binary value passed to it. 3. Ignore min/max byte arrays for data written with 1.6.0 and earlier. If you guys agree with it, I should be able to submit a PR soon.
10. [~alexlevenson], [~jaltekruse] and [~rdblue], if time permits can you guys can take a look at the suggestion above. I will wait for at least one of you guys to confirm that my suggestions are inline with what you guys have in mind.
11. I think this makes sense to me, I would try to wait for Ryan or Alex's input. I am trying to put some more time into parquet, but I am not a committer on the project, so they would be the best for a final answer on what confirms to the patterns in the current code.
12. The only thing I want to check is whether we've got this backwards or not. Binary already has a getBytes() method and I think that is exactly what we need, I don't think we need a clone() method do we? All the getBytes() methods do a copy except for ByteArrayBackedBinary. Maybe it should do a copy. Or maybe a copy should happen when ByteArrayBackedBinary is constructed. The real question is, who "owns" the byte array passed to ByteArrayBackedBinary? It seems a little odd to "give" this array to ByteArrayBackedBinary and then continue to mutate it right? But again, one way to think of Binary is as a *view* to some other byte source. So the real bug is in the statistics collection code path which should be making a defensive copy. But do we need Binary to manage this or not? Another approach would be to make getBytes() always do a copy (which it currently does for everything except ByteArrayBackedBinary). Most code paths should be using the .writeTo methods anyway right?
13. I am still learning parquet code base so please take my comments with a grain of salt. Doing a copy in BinaryStatistics is probably better as it potentially reduces number of copies (and garbage). Though in the worst case when binary field is sorted it will still create a copy for each update. I think on the write path Binary.getBytes() almost always needs to return a copy. E.g. PlainBinaryDictionaryValuesWriter already does copy() before storing a Binary in the internal dictionary. However I think that this is not needed on the read path. Which suggests that two methods are required. One more possible cleanup could be to remove public methods from Statistics classes that are not used or used in tests only (e.g. genericGetMin/Max, getMax, getMin, etc). This will break backward compatibility so I am not sure if this can be done though.
14. Thanks for the inputs guys. The reason I was suggesting to add a clone() method is that we do not change the existing behavior. However, on giving another thought, I think getBytes(), generally in java, returns a copy. Maybe we should be consistent with what

developers/ users are probably already used to. The onus of cautiously using getBytes() for performance/ space considerations is on the clients. Does the modified approach below makes sense? 1. Modify getBytes in ByteArrayBackedBinary to return a copy of backing byte array. 2. Update BinaryStatistics to use getBytes() on the Binary value passed to it only when it is updating its min or max. 3. Ignore min/max byte arrays for data written with 1.6.0 and earlier. If you guys agree with it, I should be able to submit a PR soon.

15. [~singhashish] I think that sounds good, but we will now be doing more copies than we used to, and only some of them will be necessary -- I think we need to watch out for whether this is a big performance regression or not. Another approach is to split Binary into two -- ImmutableBinary and MutableBinary, and use these two types to signal who has ownership of the backing byte arrays (eg, ImmutableBinary is guaranteed to never be mutated underneath you, but MutableBinary require a copy to safely be held on to).
16. [~alexlevenson] I agree with you that we will have to watch out for performance regression. However, having ImmutableBinary and MutableBinary still requires clients to use the right version of it, which will be true even if we just have {{getBytes()}} return the copy. In current state, {{getBytes()}} acts differently for different implementations. For some implementations it returns a copy and for some it returns the backing array. I believe it will be best to fix this and have a standard definition of {{getBytes}}. Once we know for sure that {{getBytes}} will always return a copy, we should to fix usages of the {{getBytes()}}. For example, {{AvroIndexedRecordCoverter.FieldBytesConverter}} call {{getBytes}} and then create a new buffer from those bytes, so there is a redundant copy happening for sure. However, I think it would be scope creep if we plan on doing it as part of this JIRA. We should create a separate JIRA with a clear intent. Would you agree?
17. Yeah sounds good to me -- I think the PR should try to also find as many (now un-needed) defensive copies and remove them. That should help, and if it turns into a performance problem we can address it. We should also look for any code that hangs on to a Binary like the statistics does.
18. I agree with the idea to make {{getBytes}} return a copy. As was noted, this is also used in {{PlainBinaryDictionaryValuesWriter}}, where we had to fix basically the same bug with a defensive copy. If we forget that getBytes makes a copy, things run a bit slower, but that is better than choosing to make callers copy the data themselves and corrupting data when we make a mistake. We should make sure that the uses are currently correct by running through all the uses of getBytes, I think.
19. Submitted PR, https://github.com/apache/parquet-mr/pull/197. There are a few places which do not need a copy of the backing byte array, as they are not modifying it. I am wondering if it makes sense to add {{getBackingBytes()}} to Binary. Having the method will help in avoiding copies, usages of {{getBytes()}}, done at places where there is no need. However, the method only makes sense for {{ByteArrayBackedBinary}}. If you guys think there is value in adding the method, then I can update the PR.
20. +1 for {{getBackingBytes}}. I was just working on PARQUET-286, which would ideally not use a copy and currently calls {{getBytes}}. Thought maybe we can come up with a name that doesn't necessarily imply no copy is made? The problem is that sometimes you must make a copy. For example, when a {{ByteBuffer}} isn't backed by a {{byte[]}}.
21. [~rdblue] I have updated the PR with a new, unsafe, version of {{getBytes}}, {{getBytesUnsafe}}.
22. The main goal of the {{Binary}} class is to abstract out over the bytes representation, whether if it's a {{byte[]}} a subsection of an existing array or something else. The idea is to avoid copying bytes as much as possible when those needs to passed several layers in and possibly logically split or concatenated before the final reading or writing. Depending on what we do with the bytes and where they are coming from, we want to either copy them or get the existing array. We certainly want to avoid unnecessary copies. this depends on two things: - whether the producer of the {{Binary}} object intends to reuse the underlying bytes (if it is backed by a buffer) or they are actually never mutated anymore (only the Binary object as a reference to it) - whether the consumer of the {{Binary}} object intends to hold onto it (dictionaries, stats) or not (it just writes it to a buffer). We can capture this at the producer level by differentiating between mutable and immutable bytes. {{Binary.fromMutable(...)}} or {{Binary.fromImmutable(...)}} Then on the consumer side we use {{getBackingBytes()}} if we don't intend to hold on it outside the current scope and {{copy()}} if we do. Now {{ImmutableBinary}} can implement {{copy()}} as just {{return this}}. I would prefer that we don't call the new method {{getBytesUnsafe}} as it is not unsafe if used in the right condition (not using the resulting bytes outside the current scope). I prefer {{getBackingBytes}} that reflects better the intent. +1 on making {{getBytes()}} a synonym of {{copy()}}
23. I agree with your breakdown of the dependencies, but I don't think Mutable/Immutable is the right distinction here. This violates the contract of [{{Immutable}}|https://jsr-305.googlecode.com/svn/trunk/javadoc/javax/annotation/concurrent/Immutable.html]: Immutable means ". . . that methods do not publish references to any internal state which is mutable by implementation even if not by design". We certainly intend for those bytes to not be changed, but there is no way to control that this is the case. With the proposed {{ImmutableBinary}} API, it would be easy for someone to accidentally mutate the backing bytes with entirely reasonable operations: {{copy}} appears to return a copy of the bytes, so it seems safe to reuse and update. I think a {{copy}} method should reliably copy or return an object that is immutable. The API only needs to reflect the consumer's intent because it doesn't necessarily know where the binary came from. I agree it makes sense to have a flag or separate class to signal the producer's intent when it creates the Binary. I think what Ashish has currently works. We have to assume that the caller isn't thinking about the bytes changing in the background, so {{getBytes}} making a copy is definitely a requirement. I like {{getBytesUnsafe}} because it makes the caller either avoid it or be careful. And I think that is in line with the typical use of "unsafe": be careful because you can do something dumb. If it were completely unsafe, we wouldn't expose the method.
24. I have updated the PR with logic to skip using binary stats of files written with versions older than 1.6.1.
25. I made some comments on the PR
26. Issue resolved by pull request 197 [https://github.com/apache/parquet-mr/pull/197]