

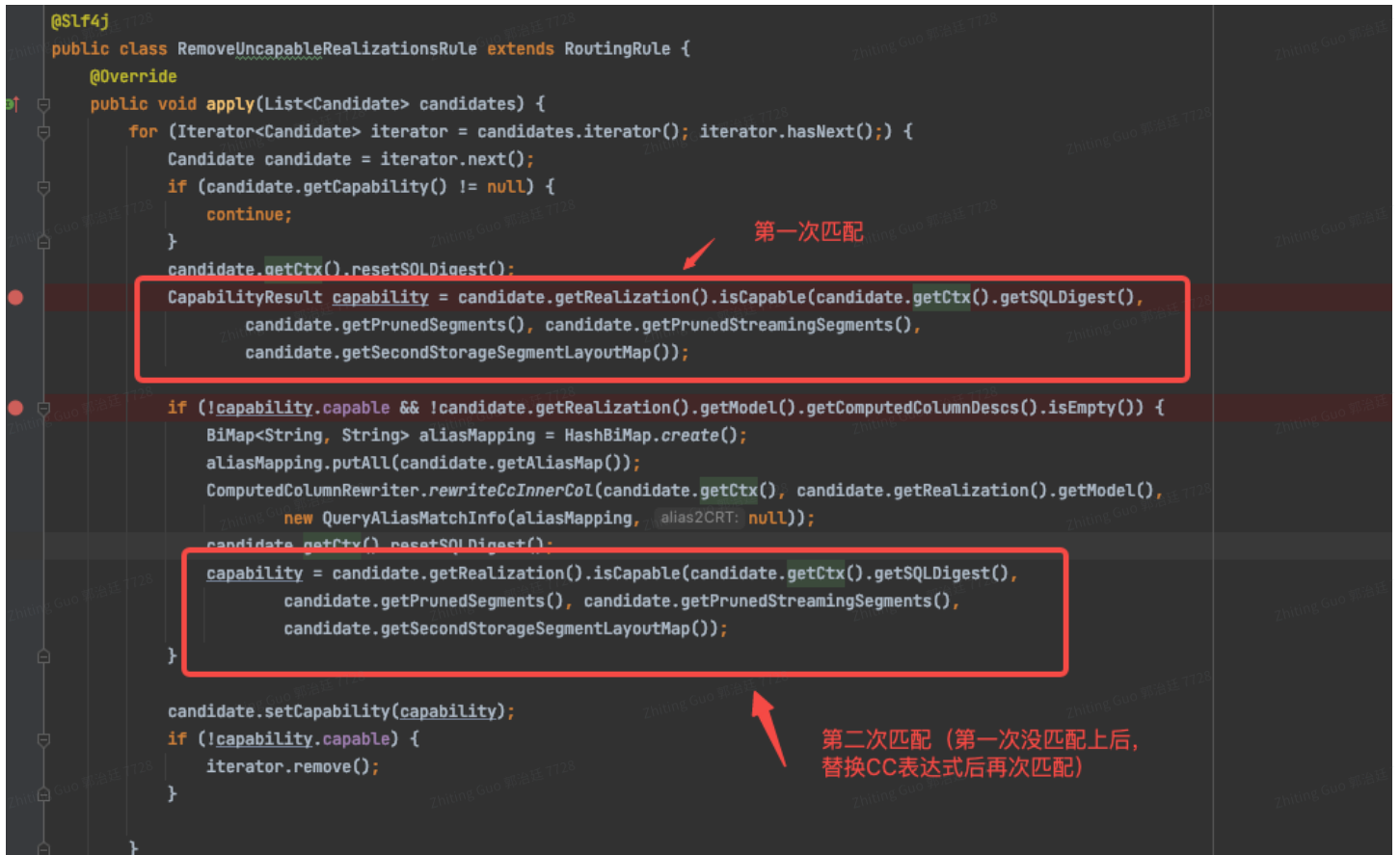
# When query contains computed columns, fail to guarantee the priority of using the aggregate index to answer the aggregate query

## BackGround

After enabling the "kylin.query.use-tableindex-answer-non-raw-query = true" & "kylin.query.layout.prefer-aggrindex = true" parameter, the aggregate query can match the aggregate index and the basic detail index, but the final hit is the basic detail index. What is puzzling is why the aggregate query is answered using the basic detail index?

## Root Cause

In the code shown in the figure below, the logic of index matching here is: first match the index for the first time, if it does not match, then replace the CC expression and match again. The problem lies in this. When the CC expression is not replaced for the first time, the aggregate index cannot answer the query at this time, only the detail index can answer, and then the detail index is directly selected here, resulting in the failure of the "kylin.query.layout.prefer-aggrindex = true" configuration.



## Fix Design

Modify the code logic mentioned in RC (org.apache.kylin.query.route.rules.

RemoveUncapableRealizationsRule #apply),

During the first match, a deep copy of `olapContext.getSQLDigest ()` will be attempted, and the copied object will be replaced with a CC expression, and then index matching will be performed. If the match is successful, the original `SQLDigest` object will be replaced by CC and the result of successful matching will be returned; if the match fails, a second match will be performed. The second match uses the original `SQLDigest` object, and if the match is successful, the result will be returned normally; if the match fails, the candidate will be removed.

The point is that "deep copy of `olapContext.getSQLDigest ()` and CC expression replacement of the copied object" needs to ensure that changes to the copy object do not affect the native object.