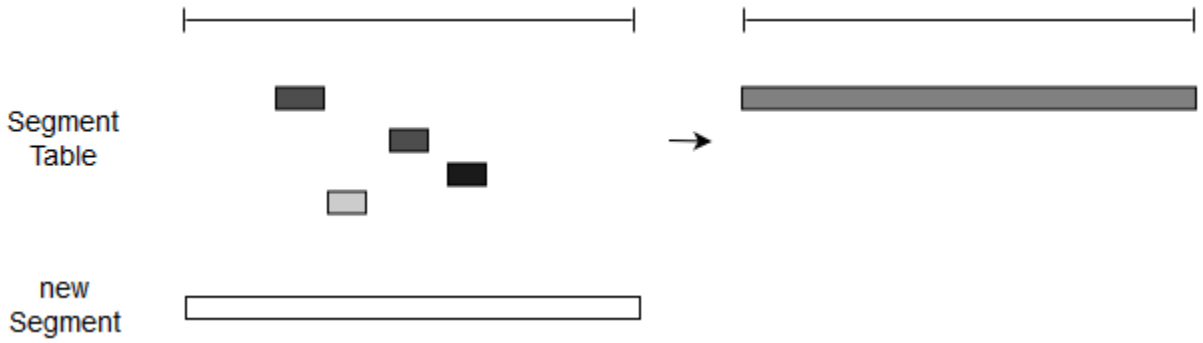
Long Segment Optimization

When inserting new segment, current segment table will try to coalesce new segment with existing segment. This will cause problem for long segment coalesce. Consider the following situation: for a new long segment, because it overlaps with all the segments in the table, the result will be a long hot segment in the table even if the latter part is not frequently range queried. This will move lots of data to LSM tree and affect random lookup performance.



The solution for this problem is to differentiate between cold segments and hot segments. In a sense, cold segments in segment table can be used to manage recency and hot segments in segment table can be used to manage frequency. When a new segment which includes existing segments is inserted and table size is larger than target, it will coalesce with existing segments only if the existing segments is below freezing point. The final heat is the larger heat of two segments not the sum. If it can't find a segment below freezing point, it will add a new segment.

For example, if boiling point is 10, freezing point is 5, there are already hot segment [10,20] with heat 10 and cold segment [30,40] with heat 5. Adding a new segment [1,100] with heat 2 will only coalesce with cold segment [30,40] and the final heat will be the larger one 5. This could avoid coalesce a long segment with a hot segment, and thus avoid large data move.

In order to better manage recency and frequency, we also ensure that: hot segments in the segment tables never overlap, and cold segments in the segment tables can overlap.

To maintain this principle, when a new segment is above boiling point, we must make sure that other hot segments don't overlap with it. One way to do so is to iterate through the segment table to look for overlapping segments. If there are existing overlapping segments, we divide the new segments into different pieces and sum the heat of overlapping part if the table size is smaller than the target number. For example, assume that the boiling point is 10 and there exists hot segment [10,20] with heat 10. If we add a new segment [5,15] with heat 10, there will be three segments at the end: [5,10] with heat 10, [10,15] with heat 20 and [15,20] with heat 10.

However, this way may cause fragmentation of the frequently range queried segment. The other way is to sum up the heats of two segments and expand the existing segment. For example, if [10,40] with heat 10 is above boiling point and there has already been a hot segment [30,50] with heat 15, we will expand it to [10,50] with heat 25. One problem of this solution is that, if users range queried a large range at first and a small range included by the large range later, the table can't tell these two ranges. For

example, if users range queried [10,100] frequently in the past but range query [90,100] currently, data in [10,80] will still keep in LSM tree because we treat [10,100] as a whole.

My current ***solution**** is that, when a new segment is above boiling point, we will check how many existing hot segments overlap with this segment. If there's only one segment and the table size is smaller than target size, divide them. Otherwise, expand these two segments and sum up their heats.

Because cold segments in the segment tables can overlap, when inserting new segment, the new segment may overlap with multiple existing segments, and we need to decide which segments to coalesce. The current design is to use "small segment first" ***solution%*****:** we iterate through all the cold segments in the segment table and pick the segment which, after coalescing, the result segment has the minimum length. For example, if there are cold segments [10,20] and [30,40]. We insert a new segment [15,38]. If we coalesce it with [10,20], we get [10,38] and the length is 28. If we coalesce it with [30,40], we get [15,40] and the length is 25. Because 25 < 28, we coalesce it with [30,40].

Finally, for a long segment which includes existing cold segments, the final length of segment is always the length of itself. In this case, we use ***solution#***: choose the longest segment included by the long segment and coalesce them together. This is to ensure that other segments are fine-grained. For example, if there are cold segments [15,20] and [30,40]. We insert a new segment [10,50]. We choose [30,40] since it is longer than [15,20].

Therefore, the final algorithm should be:

Case1: the size of table is larger than target:

> *if has an existing identical segment: sum up the heats of two segments and use solution* if above boiling point after summing*
> *else if it is included by an existing segment: coalesce two segments use solution* only if both the new segment and the existing segment are above boiling point*
> *else if it overlaps an existing segment: coalesce two segments use solution* only if both the new segment and the existing segment are above boiling point*
> *else add the new segment*

Case2: the size of table is smaller than target:

> *if has an existing identical segment: sum up the heats of two segments*
> *else if it is included by an existing segment: sum up the heats of two segments only if the existing segment is above boiling point (may need to use solution* to coalesce hot segments)*
> *else if it includes an existing segment: use solution# if both the new segment and the existing segment is below freezing point, use solution* only if both the new segment and the existing segment are above boiling point*
> *else if it overlaps an existing segment: use solution% if both the new segment and the existing segment is below freezing point, use solution* only if both the new segment and the existing segment are above boiling point*
> *else add the new segment*

Tests

I have done some single tests for the new segment table. The first test is for segments coalesce. This test is to show that, for small range queries inside a long range, segment table can coalesce these small ranges into a large range. Pseudo codes for 10% long range queries are:

do 1000 times:
    in a large range [1, 90], randomly pick N start point
    for every start point x
        range query [x, x+10]
    check how many data has been moved to LSM tree
return the average percent of data moved to LSM tree during 1000 tests

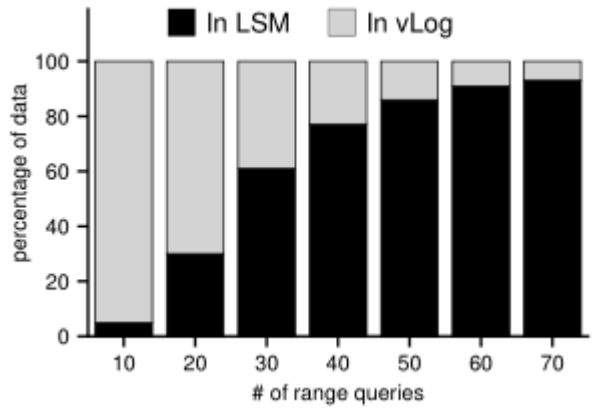


Figure 1: **Segments Coalesce Result for 10% long range queries**. This figure shows the average percentage of data migrated to LSM after a number of random range queries for 10% data.

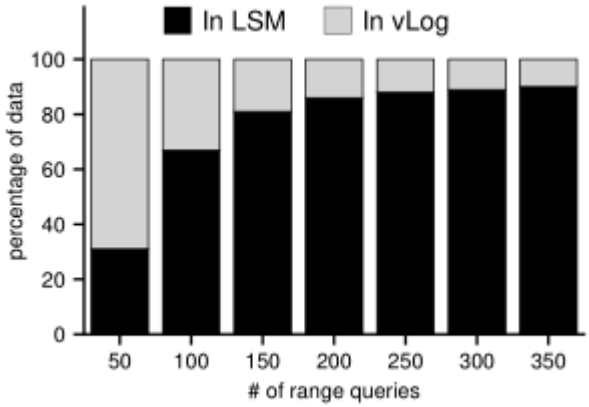


Figure 2: **Segments Coalesce Result for 2% long range queries.** This figure shows the average percentage of data migrated to LSM after a number of random range queries for 2% data.

The second test is for long range avoidance. This test is to show that, for an infrequent long-range query in the middle, segment table can avoid it. Pseudo codes for 10% long range queries are:

do 1000 times:
in a large range [1, 90], randomly pick N start point
choose random number M < N
for first M start point x
range query [x, x+10]
range query [1,200]
for last N – M start point x
range query [x, x+10]
check how many data in [101,200] has been moved to LSM tree
return the average percent of data moved to LSM tree during 1000 tests

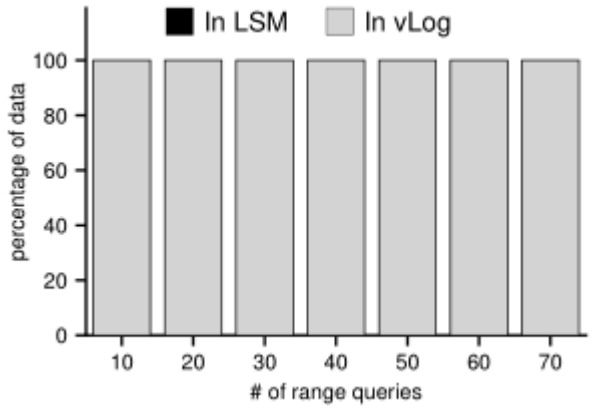


Figure 3: **Long Segment Avoidance Result for 10% long range queries**. This figure shows the average percentage of data migrated to LSM from the unexpected long range.

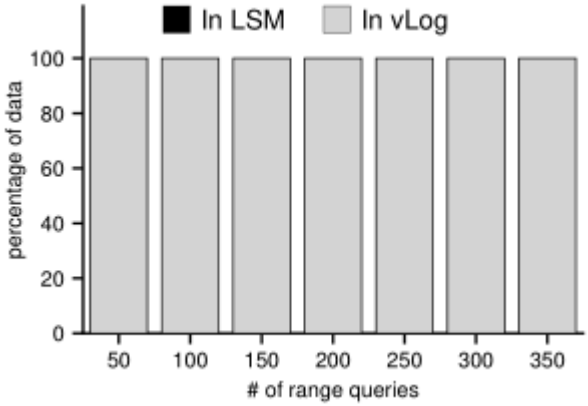


Figure 4: **Long Segment Avoidance Result for 2% long range queries**. This figure shows the average percentage of data migrated to LSM from the unexpected long range.