Optimizations for small range queries

The statistics of WiscKey as a whole may look like a ghost cache: WiscKey does the migration of data lazily; so the statistics in WiscKey records both physically moved and unmoved data. For the cache algorithm, the restriction is the size of physical cache. It is known that cache is expensive and should be kept small. Therefore, we need to find an algorithm to let the cache always keep the most valuable data. However, in WiscKey, we don't have the same restriction. We could make our statistics table as large as we want. It makes totally sense to store all the data directly into LSM tree if users are only doing long range query. The only concern is that it may occupy too much memory because of the large key size. However, we assume that this problem is not serious because WiscKey is used mainly for keys with small size.

There are other concerns for the statistics in WiscKey because it is dealing with segments not pages. What we want is a "segment replacement algorithm" which takes the properties of segments into consideration. There are some things to notice when dealing with segments:

- Segments could overlap with each other. Therefore, when inserting new segments, there may be some segments already in the statistics that overlaps with this segment, and we need to consider how to insert new segments more smoothly.
- There are also different definitions of segment hit. We could call it a segment hit whenever the new segment is totally identical to some existing segments. Or, a segment hit happens when the new segment is included or overlapped by an existing segment.
- Similarly, we could have different definitions of segment miss. We could call it a segment miss whenever the new segment is not identical to some existing segments. Or, a segment hit happens when the new segment is not included or overlapped by an existing segment.

In the ARC, it introduced a target parameter to help cache learning. We could also restrict the size of the table by providing the target table size. The target size could serve only as a reference. Statistics will try harder to make the size of table around the target size, but it will not be bounded by this size.

We define three relationship between segments:

1. two segments are identical if their starting and ending points are the same.
2. one segment is inclusive of another if it has smaller starting point and larger ending point.
3. two segments are overlapped if one segment has smaller starting point, but its ending point is larger than the other's starting point and smaller than the other's ending point.

When adding new segment, we use the following algorithm:

Case1: the size of table is larger than target:

> *if has an existing identical segment: sum up the heats of two segments*
> *if it is included by an existing segment: sum up the heats of two segments only if the existing segment is above boiling point*
> *if it includes an existing segment: add the new segment and do nothing to the existing segment*
> *if it overlaps an existing segment: sum up the heats of two segments and expand the existing segment only if both the new segment and the existing segment are above boiling point*

Case2: the size of table is smaller than target:

if has an existing identical segment: sum up the heats of two segments
if it is included by an existing segment: sum up the heats of two segments
if it includes an existing segment: sum up the heats of two segments and expand the existing segment
if it overlaps an existing segment: sum up the heats of two segments and expand the existing segment

Test Result:

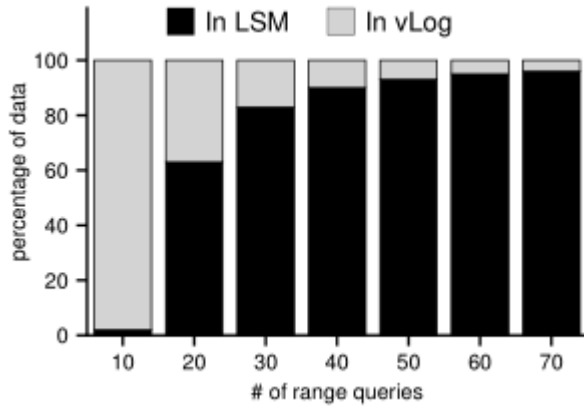

Figure 1: **Segments Coalesce Result for 10% long range queries**. This figure shows the average percentage of data migrated to LSM after a number of random range queries for 10% data.

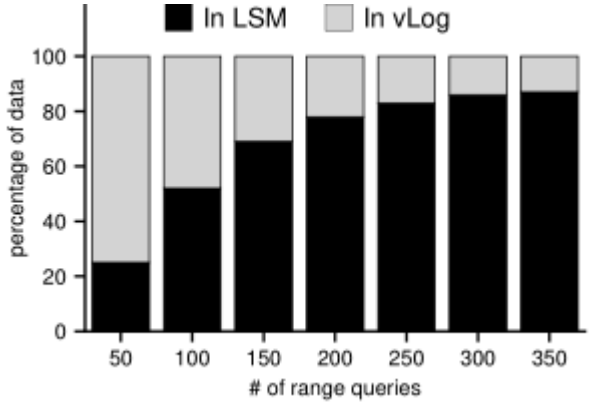

Figure 2: **Segments Coalesce Result for 1% long range queries.** This figure shows the average percentage of data migrated to LSM after a number of random range queries for 1% data.

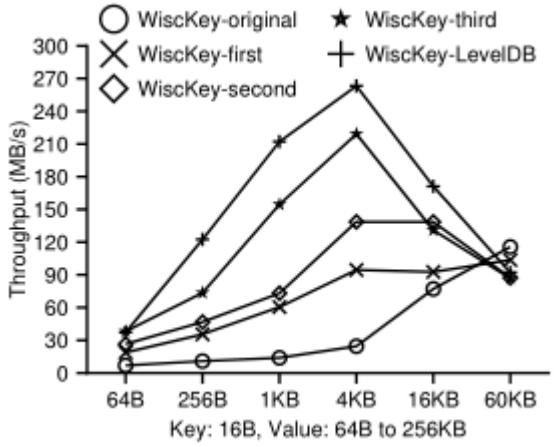

Figure 3: **Range Query Performance after 10% long Range Queries**. This figure shows range query performance after Small Range Queries for 10% data. 256 MB of data is queried from a 1-GB database that is randomly loaded.

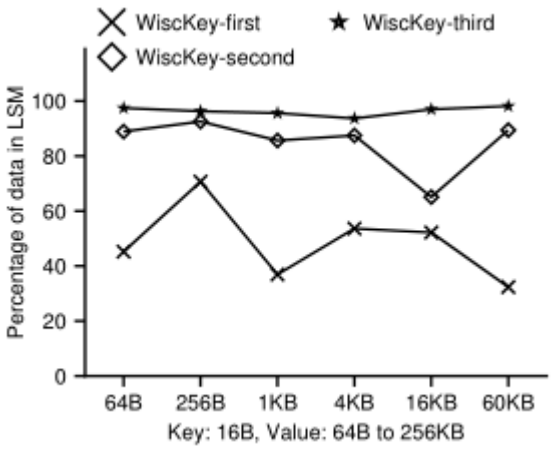

Figure 4: **Percentage of data read from LSM.** This figure shows the percentage of data stored in the LSM tree after Small Range Queries for 10% data in Figure 3.

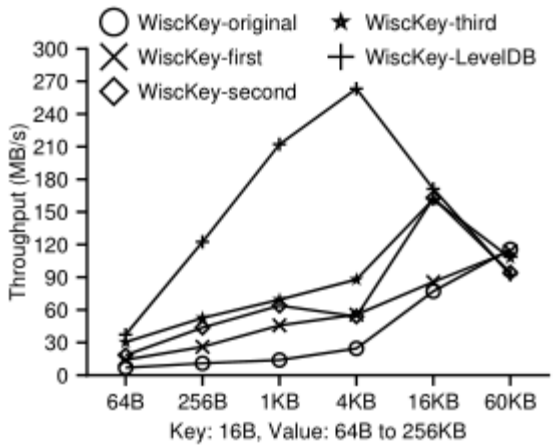

Figure 5: **Range Query Performance after 1% Range Queries**. This figure shows range query performance after Small Range Queries for 1% data. 256 MB of data is queried from a 1-GB database that is randomly loaded.

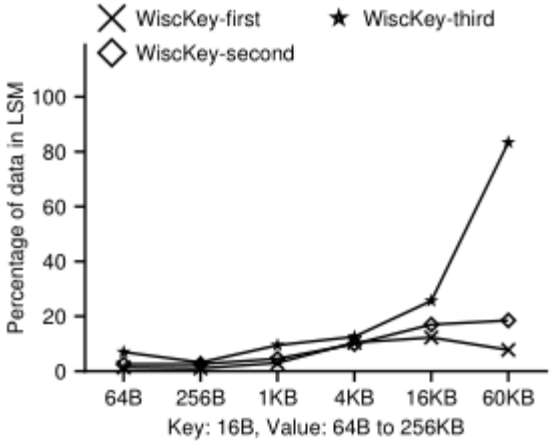

Figure 6: **Percentage of data read from LSM.** This figure shows the percentage of data stored in the LSM tree after Small Range Queries for 1% data in Figure 5.

In the test result, Figure 1 and Figure 2 show the distribution of data after range queries. In Figure 1, range query iterates through 10% of total data, while in Figure 2, it iterates through 1% of total data. Both range query starts at a random point, and the results in the Figure are the average results from 1000 experiments. We can see that for 10% long range query, it grows rapidly and converges to 100% data in LSM. As for 1% long range queries, it grows slowly and converges slowly. The different changes for different range queries are due to their possibilities to overlap and coalesce.

Figure 3 shows the Range Query Performance after 10% long Range Queries. Figure 4 shows the Percentage of data read from LSM in the range queries in Figure 3. In Figure 3, WiscKey-original stores all the data in vLog, while WiscKey-LevelDB stores all the data in LSM. These two performances show the lower and upper bound for range queries when value size is small. WiscKey-first shows the range queries performance after 20 10% long range queries, WiscKey-second shows the range queries performance after 40 10% long range queries and WiscKey-third shows the range queries performance after 60 10% long range queries. We can see that for 10% long range queries, the percentage of data migrated to LSM grows quickly and converge to 100%. This is similar to what we have seen in Figure 1. The range queries performance grows as more data are moved to LSM tree. The range queries bonus for data in LSM tree is not so obvious when value size is near 60 KB.

Figure 5 and Figure 6 shows the Range Query Performance after 1% long Range Queries and its corresponding data distribution. WiscKey-original stores all the data in vLog, while WiscKey-LevelDB stores all the data in LSM. WiscKey-first shows the range queries performance after 50 1% long range queries, WiscKey-second shows the range queries performance after 100 1% long range queries and WiscKey-third shows the range queries performance after 150 1% long range queries. Compared to 10 % long range queries, it grows and converges slowly. One reason is that for smaller range queries, it grows and converges smaller as shown in Figure 2. Also, the number of range queries is not enough to let it converge. We can see that there's a rapid grow when value size is 60KB, and this is just because of the randomness. Since we only conduct this experiment once, it is not statistically significant.