Optimizer Integration

January 22, 2015

We model the expected response time for the algorithms in terms of their time complexity and additionally the writes incurred by each one of them during the course of their execution.

1 Sort

1.1 Quicksort

In case of native quicksort, the average case time-complexity is $N_R log_2(N_R)$. Additionally, as calculated in earlier section, the writes is of the order of $N_R L_R(0.5 log_2 \frac{N_R L_R}{D} + 1)$. This would incur additional time of $\lambda \times N_R L_R(0.5 log_2 \frac{N_R L_R}{D} + 1)$. Thus, the total time complexity is $N_R(log_2(N_R) + \lambda L_R(0.5 log_2(\frac{N_R L_R}{D}) + 1))$.

1.1.1 Level-wise latency

There are $log_2 N_R$ levels. The data will be fetched from PCM till level k in the binary tree for quicksort where $\frac{N_R L_R}{2^k} > D$ i.e $k < log_2 \frac{N_R L_R}{D}$. Similarly, DRAM latency is between levels $\frac{N_R L_R}{2^k} < D$ to $\frac{N_R L_R}{2^k} > L2$ giving k between $log_2 \frac{N_R L_R}{L2} - log_2 \frac{N_R L_R}{D}$. Identical calculation holds for other levels.

Also, since the latency is per line, we have to convert the data in terms of the lines also.

Hence, the total read latency will be - $N_R L_R(\frac{t_{pcmread}}{line_{PCM}}(log_2 \frac{N_R L_R}{D}) + \frac{t_{dram}}{line_{DRAM}}(log_2 \frac{N_R L_R}{L2} - log_2 \frac{N_R L_R}{size_D}) + \frac{t_{L2}}{line_{L2}}(log_2 \frac{N_R L_R}{size_{L1}} - log_2 \frac{N_R L_R}{size_{L2}}) + \frac{t_{L1}}{line_{L1}}(log_2 N_R - log_2 \frac{N_R L_R}{size_{L1}}))$. This of course assumes $log_2 N_R > log_2 \frac{N_R L_R}{size_{L1}}$ i.e $size_{L1} > L_R$

The above expression simplies to

$$N_R L_R log_2\left(\frac{(N_R L_R)^{\frac{t_{pcmread}}{line_{pcm}}}}{D^{\frac{t_{pcmread}}{line_{pcm}} - \frac{t_{DRAM}}{line_{DRAM}}} \times L_2^{\frac{t_{DRAM}}{line_{DRAM}} - \frac{t_{L2}}{line_{L2}}} \times L_1^{\frac{t_{L2}}{line_{L2}} - \frac{t_{L1}}{line_{L1}}} \times L_R^{\frac{t_{L1}}{line_{L1}}}}\right) \quad (1)$$

Note that there is no L_R in the last term of the expression since as mentioned before, the number of levels are $log_2 L_R$.

1.2 Multi-pivot quicksort

Coming to Multi-pivot quicksort, the time-complexity is again given by $N_R log_2(N_R)$. Similarly, the writes are given by $2N_R L_R$ if $N_R L_R > D$, otherwise $N_R L_R$. The total time complexity in this case is $N_R(log_2(N_R) + 2\lambda L_R)$

For level-wise read latency for Multi-pivot sort, for count pass, the data would be always retrieved from PCM. For swap pass - it may or may not be depending on the relative sizes, thus incurring $min(2, \lceil \frac{N_R L_R}{D} \rceil) \times \frac{N_R L_R}{line_{pcm}} \times t_{pcm}$.

For each of the $2N_R$ elements, we fetch $log_2 p$ pivots due to binary search among p pivots. Assuming each pivot starts from a separate PCM line, $\frac{L_R}{line_{Li}} \rceil \times t_{Li}$ time will be incurred for fetching each pivot. Hence, we pay a total additional latency of

$$2N_R \times log_2 p \times \lceil \frac{L_R}{line_{Li}} \rceil \times t_{Li} \tag{2}$$

where Li is the level of cache where all p pivots can fit.

Afterwards, assuming we get ideal pivots such that all the partitions fit are D sized, the simple quicksort on these partitions will incur writes which are given by the earlier equation replacing $N_R L_R$ by D giving latency as

$$Dlog_2(\frac{D^{\frac{t_{DRAM}}{line_{DRAM}}}}{L_2^{\frac{t_{DRAM}}{line_{DRAM}} - \frac{t_{L2}}{line_{L2}}} \times L_1^{\frac{t_{L2}}{line_{L2}} - \frac{t_{L1}}{line_{L1}}} \times L_R^{\frac{t_{L1}}{line_{L1}}}}) \tag{3}$$

VG: Check in all expressions where ceil and floor functions apply, especially the number of cache lines should be ceil"ed" before multiplying with latency per line

2 Join

2.1 Build Phase

Assume that the entire hash table can fit in some cache level L_i (this might even be DRAM for that matter). We assume that the access to buckets is uniformly random, i.e., each access causes last level cache miss.

The build phase involves jumping to the corresponding bucket to which the inner table tuple hashes - which will incur 1 cache line miss. This is common to both.

2.1.1 Conventional HJ

For each of the N_R tuples, apart from fetching *1 line* for the right bucket, additional $\lceil \frac{size_{entry}+P}{size_{Li}} \rceil$ lines will be fetched for making an entry. This gives a total latency of

$$t_{Li} \times N_R \times (\lceil \frac{size_{entry} + P}{size_{Li}} \rceil + 1) \tag{4}$$

3 Challenges in Optimizer Integration

1. The estimated writes and cycles for *each* operator has to be put in place, not just the optimized operators.

   a) Shall we restrict ourselves only to plans containing the optimized operators? If yes, how to do that?
   b) Making this automated will be a huge overhead because:
   - We would need exact (not estimated) inner and outer tuples at each level of the plan tree. To make this work, selectivity injection or some such mechanism has to be resorted to.
   - there is no longer any cost associated with fetching tuples from the disk
   - the actual structures allocated by Postgres are far complicated to estimate *writes per tuple* due to each structure. The best thing instead would be to get optimal plan from there using our own code values. Thereafter, execute my own code so that the estimates match.

2. If we cannot use present estimates of cycles for each operator, but have to come up with a slightly more exact figure rather than order notation, it is a whole new work. Instead, just focus on adding additional cycles due to writes for each estimate? For this, an absolute value has to be put in place instead of order notation which needs to be looked at.

3. How to give the Pareto set is another challenge for which EXPAND paper should act as a reference.