Top-k Dominating Queries, in Parallel, in Memory

Sean Chester, Orestis Gkorgkas, Kjetil Nørvåg
Norwegian University of Science and Technology (NTNU), Trondheim, Norway
{sean.chester,orestis,noervaag}@idi.ntnu.no

ABSTRACT

Top-k dominating queries return the k points that are better than the largest number of other points. Current methods for answering them focus on indexed data and sequential algorithms. To exploit modern-day parallelism and obtain order-of-magnitude improvements in execution time, we introduce three algorithms, the respective strengths and potential of which are revealed experimentally.

1. INTRODUCTION

Top-k dominating queries [6] elegantly fuse top-k queries [4] with skyline queries [2] to produce ranked data without user intervention. Intuitively, a point is ranked highly if it is unequivocally better than many other points. Unsurprisingly, however, it is expensive to evaluate which k points are better than the most others.

Given the recent emergence of manycore architectures, terabyte-sized RAM, and low-latency, non-volatile memory, it is natural to ask whether top-k dominating queries can be answered more efficiently in a shared, main-memory parallel context. Until now, research has focused on sequential computation in high-latency, disk-based settings [5, 8] and on parallel computation in large, distributed systems [1], or on applications such as web-service ranking [7] and network analysis [5]. This paper investigates how significant efficiency improves can be had on just a single machine.

More formally, a data point *dominates* another, distinct data point if it has equal or better values on every attribute. The *dominance score* of a point p is simply the number of other points that it dominates. The responses to a top-k dominating query are the k points with the highest dominance scores. Given an unindexed, memory-resident dataset, we wish to find those k points as fast as possible.

This paper presents preliminary work towards that goal. In the absence of previous ones, we define three quite distinct algorithms for answering top-k dominating queries on multicore architectures and explore their relative strengths experimentally. We find that each shows promise for maturation, while already executing quickly.

2. THREE PROPOSED ALGORITHMS

For multicore top-k dominating queries, we introduce an adaptation of the sequential state-of-the-art and two novel algorithms.

FILTER As a baseline, we adapt the sequential, filter-and-refine algorithm for non-indexed data [8]. The algorithm consists of three passes: 1) build a static grid over the data and calculate score bounds for every grid cell; 2) for every cell that cannot be immediately pruned, iterate the points in the cell and filter out those that clearly cannot be in the solution; and 3) refine the solution by computing the exact score for the remaining candidates. The counting and refining steps parallelise readily, since processing is local to each point, but the filtering step would incur a lot of write contention between threads: processing for each point p will update scores for other points in addition to p. Therefore, we select the faster but $\approx 2\times$ less effective filter option (Algorithm 5) of [8] to adapt: within reason, it is better to process excess points in the high-throughput refinement phase than filter them sequentially.

SORTED Our next algorithm maximises throughput, but with heuristics to improve efficiency. We first sort the data by Manhattan Norm. Then, for each point in parallel, we iterate the sorted list. For a point p at index i, we conduct one-sided dominance tests with points at index $< i$ to count how often p *is dominated*. Clearly, if p is dominated by at least k other points, it cannot be a top-k solution; so, we break if the count reaches k. Points that reach their own index are candidates for the solution. Over the remaining indexes $> i$, we flip the dominance test and instead count how many points *are dominated by* p as a score. Finally, we re-sort the data based on the scores; the solution consists of the first k sorted points.

PIVOTED Multicore skyline algorithms benefit from pivot-based partitioning to identify incomparability in addition to dominance [3]; this algorithm attempts it for top-k dominating queries. We will select a series of pivot points (those with the largest dominance area) one-by-one, and use them to globally, dynamically split the data space into a non-uniform grid. We maintain the set of points independently of the grid. Each pivot point newly partitions the entire set of points and the number that end up in the resultant north-east quadrant is exactly the dominance score of that pivot.[1] Meanwhile, the grid is further sub-divided and upper bounds of scores for points in each sub-partition can be updated. Once no cell has an upper bound score better than that of the k'th best pivot that we have seen, we terminate. The intuition of the algorithm is that partitioning is easy to parallelise and that the continual fracturing of the data space quickly introduces more knowledge of incomparability and thus very quickly drives down all the upper bound estimates.

3. EXPERIMENTS

This section empirically compares our three parallel proposals.

Setup We implement the three algorithms in `C++` using `OpenMP`

---

[1]We describe this in two dimensions for simplicity.

	CORRELATED				INDEPENDENT				ANTICORRELATED			
	1	14	28	56	1	14	28	56	1	14	28	56
FILTER	1929	669	535	568	13412	2672	1935	1861	86369	21056	17762	16745
SORTED	376	313	323	326	5225	624	617	667	54603	4836	2535	1780
PIVOTED	812	715	732	789	964	810	842	833	48008	4870	3115	2579

Table 1: Execution time of each algorithm when run on a single core ($t = 1$), single socket ($t = 14$), all physical cores ($t = 28$), and all virtual cores ($t = 56$). Data distribution varies, but $n = 10^6$, $d = 3$, and $k = 16$ are fixed to match [8]. Times are reported in milliseconds.
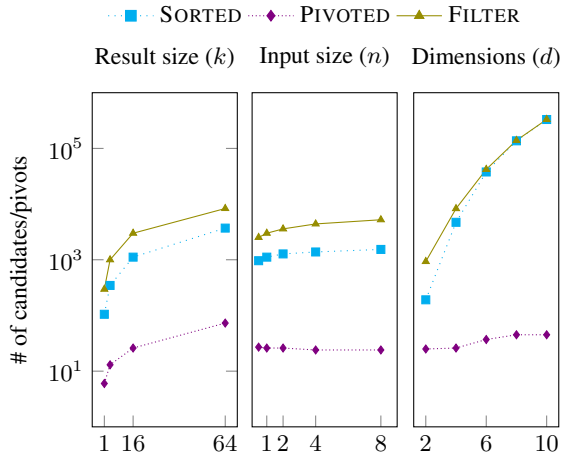

Figure 1: The number of candidates generated by each algorithm as a function of the input data and result size. Independent data.

by extending the openly available `SkyBench` suite [3] and compile with GNU `gcc-4.9.3`. We run experiments on a dual socket, 28-core Intel Xeon E5-2683 v3 at 2 GHz that is running Ubuntu 14.04 and has hyperthreading enabled. Time is measured for each algorithm after loading the input from files into a flat array. Ties in dominance score are broken by the order in which candidate points are processed so as not to bias the preferred order of any algorithm.

For comparability with Section 8.3 of [8], we use standard synthetic datasets with defaults of $k = 16$, $n = 10^6$, and $d = 3$.

Performance of algorithms Table 1 shows the execution time of the three algorithms relative to the data distribution and number of threads. In general, we observe the typical pattern that all algorithms perform best on correlated data and worst on anticorrelated data. This is unsurprising because: a) the dominance scores for the top-k points is higher on correlated data, increasing the bounds used for pruning; and b) more points are dominated by at least k other points and so can be discarded from processing earlier. With additional computational work on anticorrelated data more parallelism can be exposed, thereby achieving more parallel scalability.

The PIVOTED algorithm typically performs best at low thread counts, whereas the SORTED algorithm exhibits very good parallel scalability, even across NUMA nodes, and is consistently the fastest when using all threads. The FILTER method is limited by Amdahl's Law because of the sequential second phase, so experiences diminishing returns with increasing thread counts. The PIVOTED method struggles to utilise all threads on such low-dimensional correlated and independent data, because there are not enough partitions to iterate and the partitions are quite imbalanced.

The performance of PIVOTED on anticorrelated data is disappointing, given the success that other pivot-based methods (e.g., [3]) have had for standard skyline queries. We will in the future investigate whether better choices of pivot points and the discarding of

some unpromising partitions can improve performance here. Similarly, somehow parallelising the second phase of FILTER could yield sizeable improvements for that algorithm.

On filters and pivots Figure 1 internally inspects the performance of the algorithms by counting "important" points. For PIVOTED, these are the *pivots*. For the other methods, these are the *candidate* points that cannot be pruned. The pivots and candidates are exactly those points for which the dominance score is expensively, explicitly computed. We study how variations in k, n (measured in millions), and d (all on independent data) affect this value.

While it is clear from Figure 1 that PIVOTED computes by far the fewest exact scores, each pivot additionally further partitions the grid, creating much more overhead than evaluating candidates in the other algorithms; so, it is dangerous to compare these values directly across algorithms. Nonetheless, the low number of explicit counts required does suggest that PIVOTED could emerge as the clear best algorithm if the partitioning overhead can be reduced.

The filtering of the other methods is very sensitive to d, filtering less than 65% of the input at $d = 10$, but for $d \leq 5$, both prune $\geq 99\%$. This indicates a reasonable trade-off for FILTER, because sophisticated pruning replaces parallel work with sequential work.

Summary and future work SORTED outperforms the other methods on account of good throughput, but we plan to further develop FILTER (perhaps by trading off more filtering granularity for parallelism) and PIVOTED (trying to bridle the exponential growth of explicitly managed partitions) to see if this result can be overturned. The strong single-threaded performance of PIVOTED in particular, especially on less extreme workloads, suggests that improving its parallel scalability could lead to an extremely fast algorithm.

4. REFERENCES

[1] D. Amagata, Y. Sasaki, T. Hara, and S. Nishio. Efficient processing of top-k dominating queries in distibuted environments. In *Proc. of WWW*, 2015.

[2] S. Börzsönyi, D. Kossmann, and K. Stocker. The skyline operator. In *Proc. of ICDE*, pages 421–430, 2001.

[3] S. Chester, D. Šidlauskas, I. Assent, and K. S. Bøgh. Scalable parallelization of skyline computation for multi-core processors. In *Proc. ICDE*, pages 1083–1094, 2015.

[4] I. F. Ilyas, G. Beskales, and M. A. Soliman. A survey of top-k query processing techniques in relational database systems. *ACM Comput. Surv.*, 40(4), 2008.

[5] M. Kontaki, A. N. Papadopoulos, and Y. Manolopoulos. Continuous top-k dominating queries. *IEEE Trans. Knowl. Data Eng.*, 24(5):840–853, 2012.

[6] D. Papadias, Y. Tao, G. Fu, and B. Seeger. Progressive skyline computation in database systems. *TODS*, 30(1):41–82, 2005.

[7] D. Skoutas, D. Sacharidis, A. Simitsis, V. Kantere, and T. K. Sellis. Top-k dominant web services under multi-criteria matching. In *Proc. of EDBT*, pages 898–909, 2009.

[8] M. L. Yiu and N. Mamoulis. Multi-dimensional top-k dominating queries. *VLDB J.*, 18(3):695–718, 2009.