High-Dimensional Indexing by Distributed Aggregation

Yufei Tao

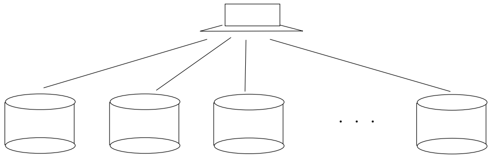
ITEE
University of Queensland

In this lecture, we will learn a new approach for indexing high-dimensional points. The approach borrows ideas from distributed aggregation: a class of problems in distributed computing that appears rather different from the geometric setup of high-dimensional indexing. We will also learn a new way to analyze the performance of an algorithm. This method—called competitive analysis—aims to prove that an algorithm is efficient on every input of the problem.

Distributed Computation

We will now temporarily forget about high-dimensional data, and focus instead on the paradigm of distributed computing.

In this paradigm, there are m servers which do not communicate with each other (consider that they are geographically apart). We want to perform certain computation on the data stored on these servers, but from a computer that is connected to the m servers in a network. For this purose, we need to acquire data from the servers over the network. The goal is to minimize the amount of communication required.



Many interesting problems can be studied in the paradigm described in the previous slide. We will look at one of them in this lecture. The problem is called distributed sum as defined in the next few slides.

Distributed Sum

Let S be a set of n objects. Each object $o \in S$ has m attributes, denoted as $A_1(o), A_2(o), ..., A_m(o)$, respectively. For each $i \in [1, m]$, the A_i-values of all objects are stored on server i. Specifically, the server sorts the set $\{A_i(o) \mid o \in S\}$ in non-descending order: denote the sorted list as L_i. Each object o has a score defined as $score(o) = \sum_{i=1}^{m} A_i(o)$. We want to identify the object with the smallest score (if multiple objects have this score, all should be reported).

Example

List L_1 (attr. A_1)	List L_2 (attr. A_2)	List L_3 (attr. A_3)	List L_4 (attr. A_4)
$(o_3, 1)$	$(o_6, 2)$	$(o_1, 1)$	$(o_1, 2)$
$(o_1, 8)$	$(o_1, 3)$	$(o_5, 2)$	$(o_3, 3)$
$(o_4, 9)$	$(o_2, 4)$	$(o_3, 3)$	$(o_2, 10)$
$(o_5, 10)$	$(o_4, 5)$	$(o_6, 8)$	$(o_5, 13)$
$(o_2, 12)$	$(o_3, 6)$	$(o_4, 10)$	$(o_6, 20)$
$(o_6, 18)$	$(o_5, 7)$	$(o_2, 11)$	$(o_4, 27)$

The answer is o_3, which has the smallest score $1 + 6 + 3 + 3 = 13$.

Distributed Sum

Each server $i \in [1, m]$ provides two operations:

- getnext(): returns the next object in L_i (or NULL if L_i has been exhausted).
- probe(o): returns the A_i value of the requested object o directly.

Example

List L_1 (attr. A_1)	List L_2 (attr. A_2)	List L_3 (attr. A_3)	List L_4 (attr. A_4)
$(o_3, 1)$	$(o_6, 2)$	$(o_1, 1)$	$(o_1, 2)$
$(o_1, 8)$	$(o_1, 3)$	$(o_5, 2)$	$(o_3, 3)$
$(o_4, 9)$	$(o_2, 4)$	$(o_3, 3)$	$(o_2, 10)$
$(o_5, 10)$	$(o_4, 5)$	$(o_6, 8)$	$(o_5, 13)$
$(o_2, 12)$	$(o_3, 6)$	$(o_4, 10)$	$(o_6, 20)$
$(o_6, 18)$	$(o_5, 7)$	$(o_2, 11)$	$(o_4, 27)$

The first getnext() on L_2 returns $(o_6, 2)$, while the second getnext on the same list returns $(o_1, 3)$. Performing probe(o_5) on L_3 returns $(o_5, 2)$.

Distributed Sum

Our objective is to retrieve the object(s) with the smallest score using only the aforementioned operations. The cost of an algorithm is the number of operations performed.

Example

$(o_3, 1)$	$(o_6, 2)$	$(o_1, 1)$	$(o_1, 2)$
$(o_1, 8)$	$(o_1, 3)$	$(o_5, 2)$	$(o_3, 3)$
$(o_4, 9)$	$(o_2, 4)$	$(o_3, 3)$	$(o_2, 10)$
$(o_5, 10)$	$(o_4, 5)$	$(o_6, 8)$	$(o_5, 13)$
$(o_2, 12)$	$(o_3, 6)$	$(o_4, 10)$	$(o_6, 20)$
$(o_6, 18)$	$(o_5, 7)$	$(o_2, 11)$	$(o_4, 27)$
List L_1 (attr. A_1)	List L_2 (attr. A_2)	List L_3 (attr. A_3)	List L_4 (attr. A_4)

We can easily solve the problem by retrieving everything from all servers, but this incurs a cost of nm. The challenge is to design an algorithm to reduce this cost as much as possible.

Applications

This problem underlies a high-dimensional indexing approach that we will explain later. Moreover, it directly models a class of applications that aim to find the top-1 object by aggregating scores from multiple sites:

- Finding the best hotel (in a region, e.g., Brisbane) by aggregating user ratings from several booking sites.
- Finding the most popular movie by aggregating ratings from several movie recommendation sites.
- ...

Algorithm

We now describe an algorithm—named the threshold algorithm (TA)—for solving the problem. Let us gain some ideas about the algorithm using our running example.

$(o_3, 1)$	$(o_6, 2)$	$(o_1, 1)$	$(o_1, 2)$
$(o_1, 8)$	$(o_1, 3)$	$(o_5, 2)$	$(o_3, 3)$
$(o_4, 9)$	$(o_2, 4)$	$(o_3, 3)$	$(o_2, 10)$
$(o_5, 10)$	$(o_4, 5)$	$(o_6, 8)$	$(o_5, 13)$
$(o_2, 12)$	$(o_3, 6)$	$(o_4, 10)$	$(o_6, 20)$
$(o_6, 18)$	$(o_5, 7)$	$(o_2, 11)$	$(o_4, 27)$
List L_1 (attr. A_1)	List L_2 (attr. A_2)	List L_3 (attr. A_3)	List L_4 (attr. A_4)

First, perform a getnext on every server. In this way, we obtain $(o_3, 1)$ from L_1, $(o_6, 2)$ from L_2, $(o_1, 1)$ from L_3, and $(o_1, 2)$ from L_4.

For L_1, set threshold $\tau_1 = 1$, indicating that all the unseen objects on L_1 must have A_1 value at least 1. Similarly, set $\tau_2 = 2, \tau_3 = 1$, and $\tau_4 = 2$.

Algorithm

Throughout the algorithm, whenever a new object o is returned by a sever, we retrieve the remaining $m-1$ attributes of o from the other servers using the probe operation. This allows us to calculate $score(o)$ immediately.

List L_1 (attr. A_1)	List L_2 (attr. A_2)	List L_3 (attr. A_3)	List L_4 (attr. A_4)
$(o_3, 1)$	$(o_6, 2)$	$(o_1, 1)$	$(o_1, 2)$
$(o_1, 8)$	$(o_1, 3)$	$(o_5, 2)$	$(o_3, 3)$
$(o_4, 9)$	$(o_2, 4)$	$(o_3, 3)$	$(o_2, 10)$
$(o_5, 10)$	$(o_4, 5)$	$(o_6, 8)$	$(o_5, 13)$
$(o_2, 12)$	$(o_3, 6)$	$(o_4, 10)$	$(o_6, 20)$
$(o_6, 18)$	$(o_5, 7)$	$(o_2, 11)$	$(o_4, 27)$

As mentioned, the getnext on L_1 retrieved o_3; thus, we perform probe(o_3) on L_2, L_3, and L_4. This gives $score(o_3) = 13$. Similarly, we obtain $score(o_6) = 48$ and $score(o_1) = 16$.

Algorithm

We then repeat the above. Namely, for each $i \in [1, m]$, perform another getnext on each L_i. Suppose that it returns $(o, A_i(o))$; we update τ_i to $A_i(o)$. If o has not been seen before, obtain its values on the other servers with probe(o).

$(o_3, 1)$	$(o_6, 2)$	$(o_1, 1)$	$(o_1, 2)$
$(o_1, 8)$	$(o_1, 3)$	$(o_5, 2)$	$(o_3, 3)$
$(o_4, 9)$	$(o_2, 4)$	$(o_3, 3)$	$(o_2, 10)$
$(o_5, 10)$	$(o_4, 5)$	$(o_6, 8)$	$(o_5, 13)$
$(o_2, 12)$	$(o_3, 6)$	$(o_4, 10)$	$(o_6, 20)$
$(o_6, 18)$	$(o_5, 7)$	$(o_2, 11)$	$(o_4, 27)$
List L_1 (attr. A_1)	List L_2 (attr. A_2)	List L_3 (attr. A_3)	List L_4 (attr. A_4)

Continuing our example, we retrieve the 2nd pair from each list: $(o_1, 8), (o_1, 3), (o_5, 2), (o_3, 3)$. Since o_5 is seen for the first time, we retrieve $A_1(o_5) = 10, A_2(o_5) = 7, A_4(o_5) = 13$, and calculate $score(o_5) = 32$. Now $\tau_1 = 8, \tau_2 = 3, \tau_3 = 2$, and $\tau_4 = 3$.

Algorithm

$(o_3, 1)$	$(o_6, 2)$	$(o_1, 1)$	$(o_1, 2)$
$(o_1, 8)$	$(o_1, 3)$	$(o_5, 2)$	$(o_3, 3)$
$(o_4, 9)$	$(o_2, 4)$	$(o_3, 3)$	$(o_2, 10)$
$(o_5, 10)$	$(o_4, 5)$	$(o_6, 8)$	$(o_5, 13)$
$(o_2, 12)$	$(o_3, 6)$	$(o_4, 10)$	$(o_6, 20)$
$(o_6, 18)$	$(o_5, 7)$	$(o_2, 11)$	$(o_4, 27)$
List L_1 (attr. A_1)	List L_2 (attr. A_2)	List L_3 (attr. A_3)	List L_4 (attr. A_4)

Recall that our current best object o_3 has score 13. Since $\sum_{i=1}^{4} \tau_i = 8 + 3 + 2 + 3 = 16 > 13$, the algorithm terminates here with o_3 as the final result (think: why?).

Pseudocode of the TA Algorithm

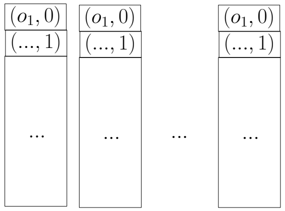
algorithm TA

```
1. $\tau_1 = \tau_2 = ... = \tau_m = -\infty$
2. while $\sum_{j=1}^{m} \tau_j \le score(o_{best})$ where $o_{best}$ is the
   object with the lowest score among the seen objects
3.     for $i = 1$ to $m$
4.         $(o, A_i(o))$ = getnext() on $L_i$
5.         $\tau_i = A_i(o)$
6.         if $o$ seen for the first time then
7.             obtain $score(o)$ by probing the values of $o$ on the
               other $m - 1$ servers
```

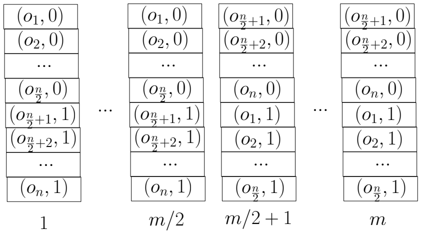
We will refer to Lines 3-7 collectively as a round. In our previous example, the algorithm terminated after 2 rounds.

Analysis

The cost of the TA algorithm heavily depends on the input. On the easiest input, it may finish with a cost of $O(m)$, while on the hardest input it may incur a cost of $\Omega(nm)$.



Best case $O(m)$



Worst case $\Omega(nm)$

Analysis

Next, we will show that the TA algorithm in fact performs reasonably well on every input (even when it has cost $\Omega(nm)$!). More specifically, we will show that its cost can be higher than that of any algorithm by a factor of at most $O(m)$. Hence, when m is small, the TA algorithm can no longer be improved considerably.

Analysis

Suppose that we have fixed the input, namely, $L_1, L_2, ..., L_m$. Denote by $cost_{TA}$ the number of operations performed by the TA algorithm.

Consider any other algorithm α whose cost is $cost_\alpha$. We will show that

$$cost_{TA} \quad = \quad O(m^2) \cdot cost_\alpha$$

If TA has the above property, it is said to be $O(m^2)$-competitive.

Analysis

Lemma: $cost_\alpha \geq m$.

Proof: We will prove that α must probe at least one object from each of $L_1, L_2, ..., L_m$.

Suppose that this is not true, such that α terminates without probing anything from L_i for some $i \in [1, m]$. Let o^* be the object returned by α. As α knows nothing about $A_i(o^*)$, it cannot rule out the possibility where $A_i(o^*)$ is exceedingly large and prevents o^* from being the best object. $\square$

Analysis

We will now prove our main theorem:

Theorem: $cost_{TA} \le (m^2 + m) \cdot cost_\alpha$.

Proof: First, if $cost_\alpha \ge n$, then the theorem follows directly from the obvious fact that $cost_{TA} \le mn$.

Next, we will focus exclusively on the scenario where $cost_\alpha < n$. Notice that in this case, there must have been at least one object $o_\#$ that has not been seen by α—that is, α does not know any attribute of $o_\#$.

In the following, we will denote by o^* the best object eventually reported (if multiple objects are returned, let o^* be any of them).

Analysis

Proof (cont.): Let x be the number of rounds performed by TA. Recall that (i) each round incurs m getnext operations, and (ii) each getnext is followed by at most $m - 1$ probe operations. Therefore:

$$cost_{TA} \leq m \cdot x + m \cdot x \cdot (m-1) = m^2 \cdot x.$$

Next, we will prove:

$$cost_\alpha \geq x - 1. \tag{1}$$

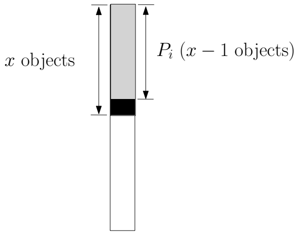
If we manage to do so, then we will have

$$\begin{aligned} cost_{TA} &\leq m^2(cost_\alpha + 1) = m^2 \cdot cost_\alpha + m^2 \\ &\leq m^2 \cdot cost_\alpha + m \cdot cost_\alpha \\ &= (m^2 + m) \cdot cost_\alpha \end{aligned}$$

which will complete the proof.

Analysis

Proof (cont.): Now it remains to prove (1). Let P_i be the sequence of the first $x - 1$ objects of L_i.



Next we will prove that α must access all the positions of at least one of $P_1, ..., P_m$. This will establish (1), and hence, complete the whole proof.

Analysis

Proof (cont.): Let τ'_i be the A_i-value of the last object in P'_i. It must hold that

$$\sum_{i=1}^{m} \tau'_i \leq score(o^*). \tag{2}$$

To see this, consider:

- **Case 1: o^* seen in the first $x-1$ rounds.** In this case, o^* must be the best object at the end of round $x-1$. By the fact that TA did not terminate at round $x-1$, (2) holds.
- **Case 2: o^* seen in the x-th round.** It thus follows that $A_i(o^*) \geq \tau'_i$ for every $i \in [1, m]$. In this case, (2) also holds.

Analysis

Proof (cont.): Assume for contradiction that, every P_i has at least one position that is not accessed by algorithm α.

Recall that there is an object $o_\#$ such that α has not seen any attribute of $o_\#$. As a result, α cannot rule out the possibility that, $o_\#$ is in P_i for every $i \in [1, m]$—namely, $o_\#$ sits at the position in P_i that is missed by α.

In that possibility,

$$score(o_\#) \leq \sum_{i=1}^{m} \tau'_i \leq score(o^*);$$

and hence, α is incorrect by not outputting $o_\#$. □

We now return to high-dimensional spaces, and explain how the TA algorithm can be utilized for index designing. For this purpose, we will again use nearest neighbor search (and its approximate version) as the representative problem. Recall:

Let P be a set of d-dimensional points in $\mathbb{R}^d$. The (Euclidean) nearest neighbor (NN) of a query point $q \in \mathbb{R}^d$ is the point $p \in P$ that has the smallest Euclidean distance to q.

We denote the Euclidean distance between p and q as $\|p, q\|$.

Index Creation

For every dimension $i \in [1, d]$, simply sort P by coordinate i. This creates d sorted lists: $P_1, P_2, ..., P_d$, one for each dimension.

For each P_i, we create a suitable structure (e.g., a hash table) that allows us to obtain, in $O(1)$ time, the i-th coordinate of any given point $p \in P$.

Answering a NN Query

We will do so by turning the query into a distributed sum problem.

Let $q = (q[1], q[2], ..., q[d])$ be the query point. Suppose that, somehow, we have obtained d sorted lists $L_1, L_2, ..., L_d$ of P where:

- In L_i ($i \in [1, d]$), the points $p \in P$ have been sorted in ascending order of $|p[i] - q[i]|^2$.

Then, the goal is essentially to minimize

$$score(p) = \sum_{i=0}^{d} |p[i] - q[i]|^2$$

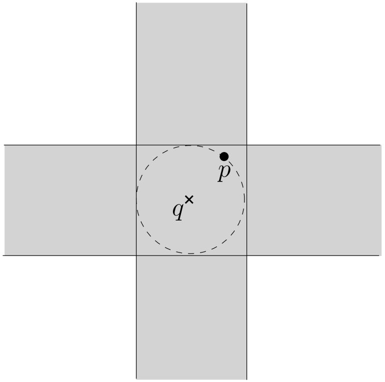
which is a distributed sum problem with $m = d$ lists.

Answering a NN Query

At first glance, it appears that we do not have $L_1, L_2, ..., L_d$ of P. In fact, we do. It will be left to you to think:

- How to implement the getnext operation in $O(1)$ time (except for the first getnext, which takes $O(\log n)$ time).
- How to implement the probe operation in $O(1)$ time.

2D Illustration



The algorithm accesses (at most) the points of P in the shaded area (p is the NN of q).

Recall that the TA algorithm is $(2m^2)$-competitive. In other words, the NN query algorithm is good only if $m = d$ is small. Unfortunately, this contradicts our goal of supporting high-dimensional queries.

Interestingly, this problem is significantly alleviated if it suffices to retrieve approximate NNs, where we only have to consider $m = O(\log n)$, no matter how large is d.

ϵ-Approximate Nearest Neighbor Search

Let P be a set of d-dimensional points in $\mathbb{R}^d$. Given a query point $q \in \mathbb{R}^d$, an ϵ-approximate nearest neighbor query returns a point $p \in P$ satisfying:

$$\|p, q\| \quad \leq \quad (1 + \epsilon) \cdot \|p^*, q\|$$

where p^* is the NN of q in P.

The Johnson-Lindenstrauss (JL) Lemma

We will not state the lemma formally, but we will explain how to use it for ϵ-approximate NN queries search. Our description will trade some rigor for simplicity, but should work quite well in practice.

First, pick $m = c \cdot \log_2 n$ random lines $\ell_1, \ell_2, ..., \ell_m$ passing the origin, where c is a sufficiently large constant (depending on m). Then, project all the points of P into the subspace defined by $\ell_1, \ell_2, ..., \ell_m$. Let P' be the resulting dataset. Note that P' is m-dimensional.

Given a query q in the original space, we obtain its projection q' in the subspace. Then, we find the (exact) NN p' of q' in the subspace (using the TA algorithm as explained before).

Return the original "image point" p of p'. The JL-lemma states that p is an $(1 + \epsilon)$-approximate NN of q with a high probability.