Cheaper by the Dozen: Batched Algorithms

Ben Gum [*], Richard Lipton
gum@cs.princeton.edu, rjl@cs.princeton.edu
Department of Computer Science
Princeton University

November 13, 2000

Abstract

In this paper we develop the idea of *batching*, processing several queries at a time, for more efficient algorithms. The prominence of massive data sets in computation has necessitated new ideas in algorithms. Consider a webpage that answers queries on a large data set. Instead of answering these queries one at a time, which can result in a substantial bottleneck, we wait for several queries to accumulate, and then apply a batched algorithm that can answer them significantly faster. We give an example of the power batched algorithms by showing an $O(nbd^{\cdot 3})$ time batched algorithm which answers b $d-$dimensional Hamming nearest neighbor queries on a dataset of size n, given that $b \geq d$.

To better understand the power of batching we give batched algorithms for the classical problems of: string matching, range searching, selection, 1D and 2D nearest neighbor, convex hull membership, and halfplane intersection. Our batched algorithms answer b queries on a dataset of size $n >> b$ in $0(nlogb)$ time, using $o(n)$ storage and at most $2n/B$ I/O's (where B is the block size). We use two techniques, query data structures and sampling, in the design of our batched algorithms. The advantages of these batched algorithms algorithms, over putting the massive dataset into a classical data structure, are threefold: improved asymptotic performance, significantly smaller data structures, and a number of I/O's which is linear in the size of the massive dataset.

1 Introduction

While computing power and memory size have been steadily increasing as predicted by Moore's Law, they are still dwarfed by the size of massive data sets resultant from a number of applications. Many problems arising from astrophysics, computational biology, telecommunications, and the Internet often have an amount of accompanying data in the terabyte range. The analysis of this

[*]Ben Gum's work was supported in part by an NSF Graduate Fellowship

data by classical algorithms is often prohibitively expensive. Thus new ideas are necessary to create algorithms to deal with these massive data sets.

In this paper we develop the idea of *batching*, processing several queries at a time, for more efficient algorithms for several query problems. The advantages of our algorithms, over the classical approach of putting the massive dataset into a data structure, are threefold: improved asymptotic performance, significantly smaller data structures, and a number of I/O's which is linear in the size of the massive dataset. We use two techniques, query data structures and sampling, in the design of our batched algorithms.

To illustrate the idea of batched algorithms, we consider the dictionary problem. Suppose we begin with n unsorted items. If we have only one query, it does not make sense to place the n items in a data structure; the best we can do is the brute force method of comparing the query with all n items. Now suppose we have b queries. If b is large enough and we have enough space, it makes sense to build a data structure such as a binary tree or perfect hash table.

However, if $b << n$, we can do better. We simply sort the list of the b queries and for each dictionary item, perform binary search to see see if the dictionary item matches a query. Thus we can check the b queries in $O(nlogb + blogb)$ time and using only $O(b)$ storage space. Compare this to $O(bn)$ time for the case of no preprocessing, and $O(nlogn + blogn)$ time and $O(n)$ space, for the case of a binary search tree. For this problem, and a number of others we will show, batched algorithms 'fill in the blank' between the brute force approach for one query, and the classical data structure approach for $\Omega(n)$ queries.

In the sections that follow, we show how batched algorithms can be applied to other problems. In section 2, we compare the idea of batching to other ideas in algorithms, including external memory algorithms and stream computing. We give an $O(nbd^{.3})$ time algorithm for the problem of $d-$dimensional Hamming nearest neighbor in section 3. In section 4, we show how query data structures can give us batched algorithms which answer b queries in $O(nlogb)$ time for 1D Nearest Neighbor, String Matching, 2D Range Searching, and 2D Nearest Neighbor. In section 5, by adding the idea of sampling to the query data structure, we give $O(nlogb)$ time batched algorithms for Selection, 2D Inhull, and Halfplane Intersection. Without the sampling phase, the algorithms would be I/O intensive and require at least $O(n)$ extra space. Finally, in section 6, we discuss some directions for future work.

2 Related Work

We get the name *batching* from cryptography. There are several works which apply the idea of batching to problems in cryptography, including signature verification [2], exponentiation [10], and RSA [4]. In these works, batched algorithms are proposed to solve b related problems with less that b times the calculation necessary for solving one. A similar idea, called *Mass Production*, is mentioned in [12] which gives a more efficient algorithm for evaluating circuits.

Our batched algorithms on classical problems bear similarity to those of

Karp, Motwani and Raghavan in [8]. In the cases of inhull and halfplane, we derived our algorithms from their "Deferred Data Structure" algorithms. However, while their objective is that queries can appear online, we allow ourselves to see and manipulate the set of queries offline, but constrain ourselves to $o(n)$ extra storage (where n is the size of the massive data set) and at most a constant number of passes over the massive data set.

Algorithms on large datasets are discussed in detail in the literature on External Memory Algorithms [11]. However most of the External Memory Algorithms are based on either previously sorted data, or on creating a data structure for the external data. We go in a different direction in this paper. We consider the case when placing our massive dataset in a data structure is either impossible due to its sheer size, or uneconomical because our query set is significantly smaller.

Stream computing [6] is another algorithms idea related to our work. Stream computing algorithms try to compute functions of large datasets with just one pass through the data. In our work we also try to minimize the number of I/O's, but we allow ourselves an additional *sampling pass* in some cases. In addition, we focus on algorithms for answering queries, b at a time, on the massive datasets.

It is interesting to note that batched algorithms can be thought of as the opposite of parallel algorithms. A batched algorithm answers several queries on one machine, while a parallel algorithm solves one problem on several machines. A batched algorithm for b queries will take at most b times as much computation as an algorithm for a single query, whereas a parallel algorithm run on m machines can achieve at most a factor of m savings. In fact, in the batched algorithms that we describe in future section, we will be able to answer b queries in only $logb$ times the computation required for a single query.

3 Hamming Nearest Neighbor

We consider the problem of d-dimensional nearest neighbor on an unorganized data set of size n. In many cases the best known approach for answering a query is simply the brute force method, which takes $O(nd)$ time.

Notice however that the Hamming distance between two vectors $\vec{a}$ and $\vec{b}$ is $\vec{a} \cdot \vec{a} + \vec{b} \cdot \vec{b} - \vec{a} \cdot \vec{b}$. Using this fact and fast matrix multiplication we can create a batched algorithm, which multiplies two dxd matrices in $O(d^{.3})$ time, to answer b $d-$dimensional Hamming nearest neighbor queries in $O(nbd^{.3})$, where $b \geq d$. To see this, partition the queries and the data items into sets of size d.

This is a relatively simple algorithm but it illustrates the power of batched algorithms for difficult problems on massive data sets. We believe similar ideas can be used to create algorithms for different nearest neighbor problems. To further explore the power of batching, in the next sections we give batched algorithms for a number of classical problems.

4 Query Data Structures

In this section we present batched algorithms for several problems which rely on the technique of creating a query data structure with the b query items and using it to process the n data items. These batched algorithms typically take $poly(b)$ time to build the query data structure and $O(nlog(b))$ time to process the data. We chose b small enough so that the entire runtime is $O(nlog(b))$. Another advantage is that these algorithms require just $poly(b)$ space, usually $O(b)$ or $O(b^2)$.

4.1 1D Nearest Neighbor

For this problem the query data structure is a small modification of the sorted list of queries. First we sort the queries to get the list $q_1 \leq ... \leq q_b$. We also maintain the lists of variables $l_1 \leq ... \leq l_b$, which are initially set to ∞, and $r_1 \leq ... \leq r_b$, which are initially set to $-\infty$.

We then process the n unsorted data items as follows. For each data item d

1. Do a binary search on the query set to find i such that $q_i \leq d \leq q_{i+1}$.
2. If $d < r_i$ then set $r_i = d$
3. If $d > l_{i+1}$ then set $l_{i+1} = d$

After all the n data items have been processed, we can examine the final l's and r's, in $O(b)$ time, and find the nearest neighbor of each query item. This algorithm takes $O(nlogb)$ time, since $b << n$. Also just $O(b)$ space is required and each of the n unsorted items only needs to be examined once.

4.2 String Matching

We consider the standard string matching problem. We are given a $0,1$ string of length n, and b $0,1$ query strings, each of length m, where $mb << n$. We use a modification of the Rabin-Karp fingerprinting algorithm [7]. Let our text be $X = x_1x_2...x_n$ where $x_1, x_2, ..., x_n$ are bits. Let $Y_1, Y_2, ..., Y_b$ be the b query strings, each of length m. Let each length-m substring of X be referred to as $X(j) = x_{j+1}x_{j+2}...x_{j+m}$ for all $0 \leq j \leq n-m$.

The *fingerprinting* consists of looking at the residues of length-m bit strings mod a random prime. Although fingerprinting introduces a small probability of a false match, it greatly reduces the time necessary to compare to check for matches. From now on we think of length-m bit strings as m-bit integers. We randomly select a prime p from all the primes less than T (we will choose T later to make the probability of a mismatch small). Let the fingerprint of Z, $F_p(Z)$, be $Z(modp)$. Notice that $F_p(X(j+1)) = [2F_p(X(j)) - 2^m x_{j+1} + x_{j+m+1}]modp$. So once $F_p(X(j))$ is calculated, $F_p(X(j+1))$ can be found with $O(1)$ operations.

Suppose we have two m-bit integers $Z_1 \neq Z_2$. The probability that $F_p(Z_1) = F_p(Z_2)$ is the same as the probability that p is a factor of $Z_1 - Z_2$ which is at most $m/\pi(T)$, where $\pi(T)$ is the number of primes less that T. (The m in the numerator is because the number of distinct prime factors of any positive integer k is at most $log_2 k$). Thus the probability that $F_p(X(j)) = F_p(Y_i)$, given $X(j) \neq Y_i$, for at least one $0 \leq j \leq n-m$, $1 \leq i \leq b$ is at most $nbm/\pi(T) = O(1/nb)$ if we set $T = n^2 b^2 m log(n^2 bm)$.

In this batched algorithm, our query data structure is simply a sorted list of the b fingerprints of the query strings. Thus for each $X(j)$, we can check in $O(logb)$ whether the fingerprint of $X(j)$ is equal to the fingerprint of a query string. In total this takes a preprocessing time of $O(mb + blogb)$ to calculate the fingerprints of the query strings and sort those fingerprints. Since $bm << n$, the total runtime of the algorithm is $O(nlogb)$ and the total storage used is $O(b)$. (Note that this is a Monte Carlo algorithm; it can be converted to a Las Vegas algorithm by verifying each match, and if there is a mismatch, then the rest is checked by the $O(nbm)$ brute force algorithm).

4.3 2D Range Searching

Consider the standard 2D range query problem. We have an unorganized set of n points, S, in R^2 and an axis parallel rectangle, defined by $x1 < x2$ and $y1 < y2$. We would like to find out which points in S lie in the rectangle defined by $(x1, x2, y1, y2)$. For one query rectangle we need $O(n)$ comparisons. We will give a batched algorithm for b query rectangles which runs in $O(nlogb + k)$, where k is the number of points reported, takes $O(b^2 + k)$ space, and examines each of the n data points only once. For convenience, we assume that none of the data points lie on a line which defines the query rectangles. (If this is not the case, we can adapt our algorithm, but it is slightly cumbersome).

We place all the $x1$'s and $x2$'s in a ($O(logb)$-depth) binary search tree. At each leaf of this tree we place a ($O(logb)$-depth) binary search tree made of the $y1$'s and $y2$'s. This gives us a size $O(b^2)$ binary tree of depth $O(logb)$ such that each leaf corresponds to a rectangle (possible unbounded on up to 2 sides) in R^2. Each of the query rectangles can be partitioned into rectangles represented by the leaf nodes of this tree. For each query rectangle, we maintain a list (initially empty) of all the data points contained in it. All this takes $O(b^2)$ time.

We process each data point in by finding the leaf rectangle that it lies in and adding it to the list of each query rectangle that the leaf rectangle lies in. It takes $O(nlogb)$ time for all of the n data points to find the leaf rectangles that they lie in and k time to update the query rectangle lists. Thus for small enough b we see that the total runtime is $O(nlogb + k)$.

4.4 2D Nearest Neighbor

For each of b query points $q_1, ...q_b$, we would like to find the closest of n unorganized data points to it. The classical algorithm, builds a Voronoi diagram with the n points in $O(nlogn)$ and then finds which region each of the b query points lies in. We would like to find a batched algorithm to answer b queries in

$O(nlogb)$, without the cost and space requirements of creating a Voronoi Diagram with the n data points. However, it is not clear what a good query data structure would be.

A reasonable starting point is to put the b query points in a Voronoi Diagram. For each query point q_i, we find c_i, the closest of the of the n data points to q_i that lies in q_i's Voronoi Region. It takes $O(nlogb)$ to compute the c_i's. Unfortunately, for any q_i there is no guarantee that c_i is q_i's nearest neighbor, or that c_i even exists. However, for each q_i some c_j is a $3-$approximate nearest neighbor. To see this let nn_i be the nearest neighbor of q_i. If nn_i is equal to c_i or some c_j, such that $i \neq j$, then we are done. Otherwise let nn_i lie in the Voronoi region of q_j. Since nn_i lies in the Voronoi region of q_j, $dist(q_i, nn_i) \geq dist(q_j, nn_i)$. Also since $c_j \neq nn_i$, $dist(q_j, nn_i) \geq dist(q_j, c_j)$. So $dist(q_i, c_j) \leq dist(q_i, nn_i) + dist(nn_i, q_j) + dist(q_j, c_j) \leq 3 * dist(nn_i, c_j)$.

Thus in O(nlogb) we can compute the 3-approximate 2D-nearest neighbor of b query points. It is easy to find an example that shows this algorithm is no better than 3-approximate. We can improve this to 2-approximate if we partition each Voronoi region into six 60 deg wedges around the q_i in that region and then save 6 points (the closest point to q_i in each of the six 60 deg wedges in its Voronoi region). There is a simple example that shows this improved algorithm is no better than 2-approximate. However, we believe these approximate nearest neighbor algorithms are promising starting points for exact nearest neighbor algorithms.

5 Query Data Structures Plus Sampling

In this section we present batched algorithms for Selection, Inhull, and Halfplane Intersection. For these problems it is possible to use straightforward query data structures to answer b queries in $O(nlogb)$ time. However these algorithms have two major disadvantages in that they uses $O(n)$ storage and require that each data item be examine $logb$ times. To remedy these drawbacks, we give sampling-based algorithms which, with high probability, solve the same problem in $O(nlogb)$ time, but only require $o(n)$ space and only look at each of the n unsorted items at most twice.

5.1 Batched Randomized Selection

In this section we give two algorithms for batched selection. The first algorithm uses a query data structure in a straightforward manner. The second algorithm combines sampling with a similar query data structure and answers the queries much more efficiently.

Recall the classical selection problem: given an unsorted dataset S, of size n, and an integer k, such that $1 \leq k \leq n$, return the kth smallest element of S. Blum et al [3] gave an $O(n)$ deterministic algorithm for selection in the early 1970's and a number of people since then have given algorithms with a better coefficient. We define the batched version of selection as follows. Let $S_{(i)}$ be the ith smallest elements in a dataset S. We are give queries: $k_1, ..., k_b$ and a dataset S and asked to find $S_{(k_1)}, ..., S_{(k_b)}$. For notational convenience, assume $k_1 < \cdots < k_b$. In fact, a special case of this problem, finding b quantiles of an unsorted dataset [1], has many applications.

We can use the algorithm by Blum et al in a divide and conquer fashion to find an $O(nlogb)$ algorithm for batched selection. We select $k_{\lceil b/2 \rceil}$ in $O(n)$ time, use it to partition S in two parts, then recurse on each half. This algorithm corresponds to creating a partial binary search tree of S, where the b interior nodes are $S_{(k_1)}, ..., S_{(k_b)}$. This takes time $T(n,b) \leq O(n) + T(k_{\lceil b/2 \rceil}, b/2) + T(n - k_{\lceil b/2 \rceil}, b/2)$, where $T(n,1) = O(n)$. This recurrence is satisfied by $T(n,b) = O(nlogb)$.

However this algorithm has some serious drawbacks. It requires storage space for the size n data structure, which may be impossible, or prohibitively expensive. Even if there is sufficient space in external memory, each element of S must be moved around as many as $O(logb)$ times, which may be very expensive. Finally even if S fits in main memory, the algorithm either has a large leading coefficient of its runtime, or is very complicated to implement (depending on whether an old deterministic or a new new deterministic select algorithm is used).

To remedy these problems, we give an improved algorithm which uses sampling to create a much smaller, faster data structure. Our sampling-based algorithm, which uses ideas from the Floyd, Rivest randomized algorithm for selection, first presented in [5] and simplified in [9], can be used to select b items using $n \log_2 \lceil 2b + 1 \rceil + o(n)$ comparisons, when $b = O(n^{1/6})$.

Algorithm:

1. Sample $n^{5/6}$ elements from S with replacement, call the sample R. Sort R. (Sampling with replacement is easier to do and does not hurt our analysis).

2. For each k_i, let $y_i = \frac{k_i}{n} n^{5/6} = k_i n^{-1/6}$ (y_i is the expected number of items in R which are less that $S_{(k_i)}$).

3. Let $x_i = R_{(y_i - \sqrt{n})}$, if $y_i > \sqrt{n}$ otherwise let $x_i = S_{(1)}$. Let $z_i = R_{(y_i + \sqrt{n})}$, if $y_i < n - \sqrt{n}$ otherwise let $z_i = S_{(n)}$. (We will show that with high probability, for all i, $x_i \leq S_{(k_i)} \leq z_i$).

4. Sort the set containing all the x_i's and z_i's. The x_i's and z_i's partition the rest of the set S into at most $2b + 1$ parts (according to which two an item of S lies between). Using binary search, find the place of each of the n elements of S among the x_i's and z_i's. (This binary search part is the most costly phase of the algorithm).

5. Let $I_i = \{s \in S | x_i \leq s \leq z_i\}$. Sort the elements in each interval I_i . (We will show that with high probability each interval I_i contains at most $4n^{2/3}$ elements).

6. Now to find each $S_{(k_i)}$, if $S_{(k_i)}$ lies in I_i then from step 4 we know how many elements of S that are less than $S_{(k_i)}$ and from step 5 we have sorted I_i so we can quickly find $S_{(k_i)}$.

Through a straightforward use of Chernoff-Hoeffding bounds, we can show that with high probability the following conditions hold:

1. for all i, $S_{(k_i)}$ lies in I_i
2. for all i, I_i contains at most $4n^{2/3}$ elements

We need the first condition so we can perform step 6 of our algorithm and find all the $S_(k_i)$'s. The second condition implies that step 5 of our algorithms takes $o(n)$ time. The binary search phase of step 4 requires $n \log_2 \lceil 2b+1 \rceil$ comparisons. Since the other steps take $o(n)$ comparisons, the total number of comparisons used by the algorithms is $n \log_2 \lceil 2b+1 \rceil + o(n)$. Thus with high probability our algorithm correctly answers b select queries on an unsorted dataset of size n in $O(nlogb)$ time, using $o(n)$ extra storage, and examining each data item at most twice.

5.2 2D Inhull

Using ideas similar to [8] we give batched algorithms for the deciding whether or not each of b query points lie inside the convex hull of a set P of n data points. As in the case of select, we first give a straightforward I/O-intensive algorithm and then refine it using a sampling phase. For notational convenience, let n be a multiple of b, let all the x coordinates of the points be distinct.

In $O(nlogb)$ time we partition the n data points by x coordinate into b vertical strips each containing n/b points. Let V_i be the ith strip from the left, P_i be the set of the n/b data points which lie in V_i, and let L_i be the vertical line separating V_i from V_{i+1}. Next, in $O(nlogb)$ time, we find the set T of at most $2b-2$ segments of the convex hull of P which intersect the L_i's. We see that if a query point q lies in the convex hull of P, then it lies in the convex hull of $T \cup P_i$, where V_i is the vertical strip containing q. Once we have each P_i, and know T this algorithm takes us $O(n/b+b)$ time. (This is because given a query point, and a set of k points, the query point lies inside the convex hull of the k points if and only if the range of angles of the vectors from the query point to the k points is at least 180 degrees. We can test this in $O(k)$ time using $O(1)$ space). Thus the total time to answer all b queries is $O(nlogb)$.

Unfortunately, the above algorithm creates a data structure of size $O(n)$, which may be prohibitively expensive. Using a sampling phase we give an algorithm which uses only $O(b^3)$ storage (which is advantageous provided $b = o(n^{1/3})$). First we, randomly with replacement, take a sample S, of b^3 points, from P. Then we find the vertical lines $L'_1, ...L'_{b-1}$ which partition the plane into b vertical strips, each containing b^2 points of S. Using Chernoff-Hoeffding bounds we can show that with high probability, at most $3n/b$ point of P lie in each vertical strip. With one $O(nlogb)$-time pass through P we can compute T'

and then with a second $O(nlogb)$-time pass we can check to see if each query point lies within the convex hull of the union of T' and P'_i, where V'_i is vertical strip containing the query point.

5.3 Halfplane Intersection

Finally we consider the question of whether a query point lies in the intersection of n unsorted halfplanes. If the intersection is empty, the problem is trivial. Otherwise we can transpose the halfplanes in $O(n)$ time so that they contain the origin. Then this problem dualizes to the problem of whether a query line does not intersect a convex hull. Observe that for any vertical line L, if we know the segments of the convex hull that intersect L then we can say one of four things about a query line q:

1. q does not intersect the convex hull
2. q does intersect the convex hull
3. if q intersects the convex hull, it does so to the left of L
4. if q intersects the convex hull, it does so to the right of L

Using this observation we can come up with an $O(nlogb)$ batched algorithm for answering b queries, similar to that in the previous section, which using the same sampling scheme can be done using a size $O(b^3)$ data structure.

6 Future Work

In the previous sections, we have explored the idea of batching and the power of its applications to algorithms for query problems. We gave a batched algorithm for answering b queries of d-dimensional Hamming nearest neighbor on a corpus of size n in $O(nbd^{\cdot 3})$ time, for $b \geq d$. In addition, using two techniques, query data structures and sampling, we have given batched algorithms for the classical problems of: string matching, range searching, selection, (1D and approximate 2D) nearest neighbor, inhull, and halfplane intersection. These batched algorithms answer b queries on a dataset of size $n >> b$ in $0(nlogb)$ time using $o(n)$ storage, and at most $2n$ I/O's.

There are numerous problems which would benefit from efficient batched algorithms. High dimensional nearest neighbor and computational biology are two very promising directions for work in batched algorithms. However we believe that nearly all query problems are have the potential for efficient batched algorithms. We hope this paper will inspire future work in this area.

References

[1] Alsabti, K., Ranka, S., Singh, V. *A One-Pass Algorithm for Accurately Estimating Quantiles for Disk-Resident Data*, 23rd VLDB Conference. (1997).

[2] Bellare, M., Garay, J., Rabin, T. *Fast Batch Verification for Modular Exponentiation and Digital Signatures*, Eurocrypt. (1998).

[3] Blum, M., Floyd, R., Pratt, V., Rivest, R., Tarjan, R. *Time Bounds for Selection*, Journal of Computer and System Sciences, 7 (1973), 448–461.

[4] Fiat, A. *Batch RSA* Journal of Cryptography, Vol 10, No 2 (1997), 75–88.

[5] Floyd, R., Rivest, R. *Expected Time Bounds for Selection*, Communications of the ACM, 18 (1975), 165–177.

[6] Henzinger, M., Raghavan, P., Rajagopalan, S. *Computing on Data Streams* SRC Technical Note, 1998-011. (1998).

[7] Karp, R., Rabin, M. *Efficient randomized pattern-matching algorithms*, IBM Journal of Research and Development, 31 (1987), 249–260.

[8] Richard M. Karp, Rajeev Motwani, and Prabhakar Raghavan. *Deferred data structuring*. SIAM Journal on Computing, 17(5):883-902, October 1988.

[9] Motwani, R., Raghavan, P. *Randomized Algorithms*, Cambridge University Press. (1995).

[10] M'Raïhi, D., Naccache, D. *Batch exponentiation - A fast DLP based signature generation strategy* 3rd ACM Conference on Computer and Communications

[11] Vitter, J. *External Memory Algorithms and Data Structures: Dealing with MASSIVE DATA* ACM Computing Surveys. (2000).

[12] Wegener, I. *The Complexity of Boolean Functions* John Wiley and Sons. (1987).