Designing Good MapReduce Algorithms

An introduction to designing algorithms for the MapReduce framework for parallel processing of big data.

By Jeffrey D. Ullman

DOI: 10.1145/2331042.2331053

If you are familiar with "big data," you are probably familiar with the MapReduce approach to implementing parallelism on computing clusters [1]. A cluster consists of many compute nodes, which are processors with their associated memory and disks. The compute nodes are connected by Ethernet or switches so they can pass data from node to node.

Like any other programming model, MapReduce needs an algorithm-design theory. The theory is not just the theory of parallel algorithms—MapReduce requires we coordinate parallel processes in a very specific way. A MapReduce job consists of two functions written by the programmer, plus some magic that happens in the middle:

1. The *Map function* turns each input element into zero or more key-value pairs. A "key" in this sense is not unique, and it is in fact important that many pairs with a given key are generated as the Map function is applied to all the input elements.

2. The system sorts the key-value pairs by key, and for each key creates a pair consisting of the key itself and a list of all the values associated with that key.

3. The *Reduce function* is applied, for each key, to its associated list of values. The result of that application is a pair consisting of the key and whatever is produced by the Reduce function applied to the list of values. The output of the entire MapReduce job is what results from the application of the Reduce function to each key and its list.

When we execute a MapReduce job on a system like Hadoop [2], some number of Map tasks and some number of Reduce tasks are created. Each Map task is responsible for applying the Map function to some subset of the input elements, and each Reduce task is responsible for applying the Reduce function to some number of keys and their associated lists of values. The arrangement of tasks and the key-value pairs that communicate between them is suggested in Figure. 1. Since the Map tasks can be executed in parallel and the Reduce tasks can be executed in parallel, we can obtain an almost unlimited degree of parallelism—provided there are many compute nodes for executing the tasks, there are many keys, and no one key has an unusually long list of values

A very important feature of the MapReduce form of parallelism is that tasks have the blocking property [3]; that is, no Map or Reduce task delivers any output until it has finished all its work. As a result, if a hardware or software failure occurs in the middle of a MapReduce job, the system has only to restart the Map or Reduce tasks that were located at the failed compute node. The blocking property of tasks is essential to avoid restart of a job whenever there is a failure of any kind. Since MapReduce is often used for jobs that require hours on thousands of compute nodes, the probability of at least one failure is high, and without the blocking property large jobs would never finish.

There is much more to the technology of MapReduce. You may wish to consult, a free online text that covers MapReduce and a number of its applications [4].

EFFICIENT MAPREDUCE ALGORITHMS

A given problem often can be solved by many different MapReduce algorithms. We shall start with a real example of what can go wrong and then examine a model that lets us talk about the important tradeoff between the communication (from Map to Re-

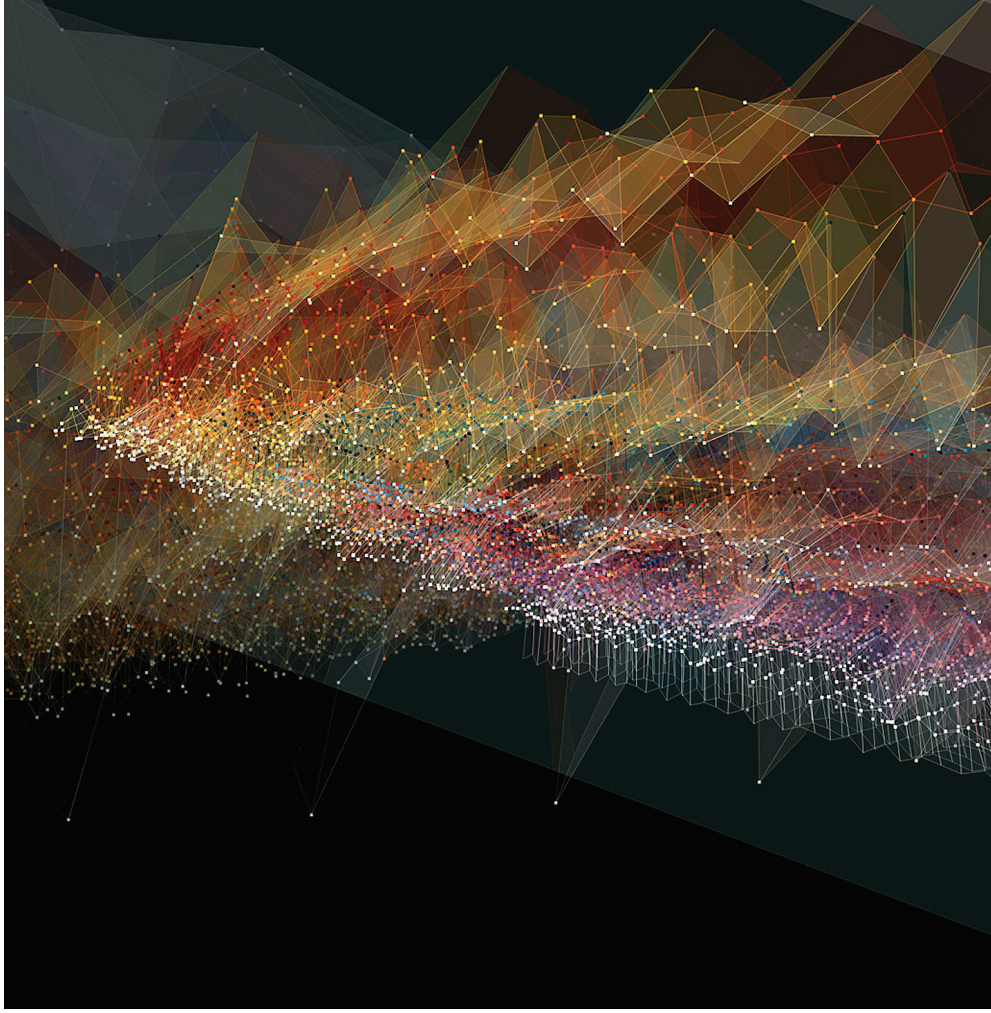
duce tasks) and computation (at the Reduce tasks).

Reducers. It is convenient to have a term to refer to the application of the Reduce function to a single key and its list. We call this application a reducer. The input size for a reducer is the length of the list. Notice that reducers are not exactly the same as Reduce tasks. Typically a Reduce task is given many keys and their lists, and thus executes the work of many "reducers." However, there could be one Reduce task per reducer, and in fact, there could even be one compute node per reducer if we wanted to squeeze the absolute maximum degree of parallelism out of an algorithm.

Analogously, we can think of a mapper as the application of the Map function to a single input element. Normally, mappers are grouped into Map tasks, and each Map task is responsible for many mappers. It is more common for us to be able to gain efficiency by redesigning the nature of the reducers than by redesigning the mappers, so we shall be concentrating on the reducers in this article.

Communication and computation costs. There are three principal sources of cost when you run a MapReduce job:

1. There is a map cost of executing the mappers. Normally, the input is a file distributed over many compute nodes, and each Map task executes at the same compute node that holds the input elements to which it is applied. This cost is essentially fixed, and consists of the computation cost of executing each mapper.
2. Each key-value pair must be transmitted to the location of the Reduce task that will execute the reducer for that key. While by coincidence this Reduce task may be located at the same compute node as the Map task that generated that key-value pair, we shall assume for convenience each key-value pair is shipped across the network that connects the compute nodes. The communication cost, or cost of moving the data from Map tasks to Reduce tasks, is thus proportional to the total number of key-value pairs generated by the mappers.
3. Each reducer must execute at the compute node to which its key is assigned. The computation cost for an algorithm is the sum of the time needed to execute each reducer.



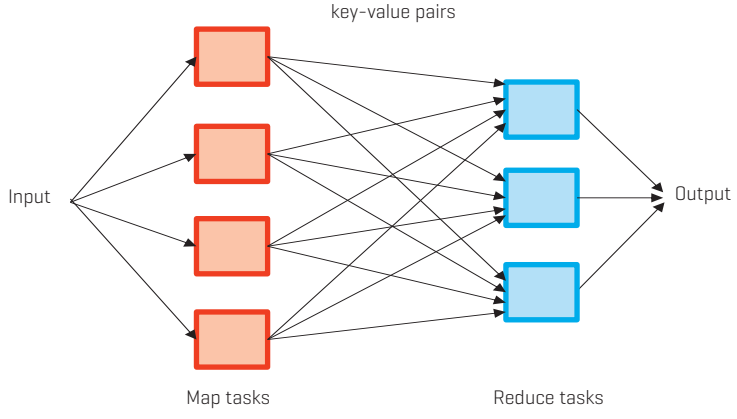
Generative artwork by Gwen Vanhee. Created in HTML5 and Javascript.

This distinction between communication cost and computation cost appears to ignore the computation needed to execute the mappers. However, commonly, this cost is proportional to the number of key-value pairs generated, and thus can be included in the communication cost. We shall therefore not discuss the cost of executing the mappers further.

It may not be obvious, but communication cost often dominates the computation cost. Typically, compute nodes are connected by gigabit Ethernet. That seems fast if you are downloading a song, but when you have to move a terabyte, it will take at least three hours across a gigabit Ethernet connection.

Skew and wall-clock time. We focus on communication and computation cost because in a public cloud, like

Figure 1: The structure of a MapReduce job.



Amazon's EC2, that is what you pay for [5]. You pay by the gigabyte for moving data across the network, and you rent compute nodes by the hour. However, in addition to wanting to minimize what we pay, we also want our job to finish soon. Thus, the total elapsed time before finishing the MapReduce job is also important.

As long as no mapper or reducer has too large an input size, we can divide them among as many compute nodes as we have access to, and thus have a wall-clock finishing time that is roughly the total time of the computation and communication, divided by the number of compute nodes. However, if we are not careful, or the data has a bad form, then we are limited in how fast we can finish by the phenomenon of skew.

The most common form of skew is when the data causes one key K to be produced a significant fraction of the time. If, say, half the key-value pairs generated by the mappers have key K, then the reducer for key K has half of all the data communicated. The computation time of the reducer for K will be at least half of the total computation time; it could be more if the running time of the Reduce function grows faster than linearly in the size of the list. In such a situation, the wall-clock time for finishing cannot be less than half the total computation cost, no matter how many compute nodes we use. From this point onward, we shall assume that skew is not a problem, although there is much evidence that skew does affect the wall-clock time significantly in many cases; see Kwon et al. for example [6].

The grand compromise. For many problems, there is a tradeoff between the input size for the reducers and the replication rate, or number of key-value pairs generated per input element. The smaller the input size, the more parallelism we can have, which leads to a lower wall-clock time. But for problems that are not "embarrassingly parallel," lowering the input size for the reducers means increasing the replication rate and therefore increasing the communication. The more communication, the slower the job runs and the more it costs. Thus, we must find just the right input size to compromise between our desire for low cost and low wall-clock time.

The study of optimal MapReduce algorithms can thus be viewed as the study of the function that gives the least possible replication rate for a given reducer input size. We need to do two things: Prove lower bounds on the replication rate as a function of input size; and discover algorithms whose replication rate matches the lower bound.

AN EXAMPLE OF THE TRADEOFF

To see how the grand compromise works in practice, I am going to tell a story about a real project. At Stanford, I coached several teams in the data-mining project course. One of the teams was looking at medical records for about a million patients, and was trying to discover unknown drug interactions. They were indeed successful not only in verifying known interactions, but in discovering several very suspicious, heretofore unknown, combinations of drugs that have significant side effects.

To find pairs of drugs that had particular side effects, they created a record for each of the 6,500 drugs in the study. That record contained information about the medical history of patients who had taken the drug; these records averaged about a megabyte in length. The records for each pair of drugs needed to be compared in order to determine whether a particular side effect was more common among patients using both drugs than those using only one or neither.

Their initial plan was to use MapReduce with one reducer for each pair of drugs. That is, keys would be ordered lists of two drugs $[i, j]$ with $i < j$, and the associated values would be the records for the two drugs. The Map function would take a drug i with record R and turn it into many key-value pairs. Each of these had a value (i, R), meaning that R was the record for drug i. But the keys were all the lists consisting of i and any other drug j. For each of the 6,500 drugs they therefore created 6,499 key-value pairs—each about a megabyte in size—for a total communication cost of about 20 terabytes. It was no surprise that they were unable to do this MapReduce job, even given the generous allocation of free EC2 service that Amazon had provided for the class to use.

So they needed to make a compromise between their desire to run as many reducers as possible in parallel and their need to keep the communication within bounds. They grouped the drugs into 65 groups, numbered 1 to 65, of 100 drugs each. Instead of keys being sets of two drugs, they used keys sets of two group numbers. The mapper for drug i and record R created 64 key-value pairs. In each, the value was (i, R), as before. The keys were all pairs of two groups, one of which is the group of drug i and the other of which is any other group.

A reducer in the new scheme received a key that is a set of two groups, and a list of 200 elements (i, R), where i is a drug in one of the two groups and R is the patient record for that drug. The reducer compared each element (i_1, R_1) and (i_2, R_2) on its list, provided i_1 and i_2 were drugs in different groups. A small trick that I won't go into was necessary to make sure that drugs in the same group were also compared by exactly one of the reducers.

As a result, every pair of drugs had their records compared exactly once, just as in the original scheme, so the computation cost was essentially the same as before. The input size to a reducer grew by a factor of 100, so the minimum wall-clock time was much greater under the new scheme. However, the replication rate shrunk by a factor of over 100, so the communication was around 200 gigabytes instead of 20 terabytes. Using the new scheme, the various costs balanced well, and the job was able to complete easily.

SOME CONCRETE TRADEOFFS

Now, we are going to see some facts about particular problems and the way reducer input size and replication rate are related for these problems. We shall look at the problem of finding bit strings at Hamming distance 1, and then at the problem of finding triangles in a graph. However, we begin by looking at the tradeoff implied by the previous discussion.

Tradeoff for the medical example. We can generalize the two different strategies we considered as follows. Suppose there are d drugs, and we want to group them into g groups. The record for each drug is then replicated

$g-1$ times, which we'll approximate as g times to simplify the formulas. The input size for each reducer is $2d/g$ records. Conventionally, we use q for the maximum allowable input size for a reducer and r for the replication rate. In this case, we have $r = g$ and $q = 2d/g$, so r as a function of q is

$$r = 2d/q$$

That is, the replication rate is proportional to the number of drugs and inversely proportional to the reducer input size.

As long as g divides d evenly, we can choose any g we like and have an algorithm that solves the problem. We discussed only two cases: $d = g = 6{,}500$ (the original attempt) and $d = 6{,}500$, $g = 65$, which worked. However, if the communication were still too costly at $g = 65$, we could have lowered it further to decrease the replication rate yet again. At some point, the communication cost would cease to be the dominant cost, and we could extract what parallelism remains to keep the wall-clock time as low as possible.

Strings at Hamming distance 1. We are now going to take up a problem that was analyzed in a recent paper on understanding the limits of map-reduce computation [7]. Two bit strings of the same length b are at Hamming distance d if they differ in exactly d corresponding bits. For example, 0011 and 1011 are at Hamming distance 1 because they differ only in the first bit.

For $d = 1$ there is an interesting lower bound on replication rate as a function of q, the maximum number of strings that can be sent to any reducer. For an algorithm to find all pairs of strings at Hamming distance 1 in some input set of bit strings of length b, every pair of bit strings at distance 1 must be covered by some reducer; in the sense that if they exist in the input set, then both strings will be sent to that reducer (perhaps among other reducers). The number of possible inputs is 2^b, and the number of possible outputs—pairs at distance 1— is $(b/2)2^b$. To see why the latter count is correct, notice that each of the 2^b bit strings of length b is at distance 1 from b other strings; those are the strings constructed by flipping exactly one of the b bits. So we would expect $b2^b$ pairs, but that counts each pair twice, once from the point of view of each of the two strings. Thus the correct count of possible outputs is $(b/2)2^b$.

There is a theorem that says among any collection of q bit strings, there are at most $(q/2)\log_2 q$ pairs at distance 1 [7]. We're not going to prove it here, but we'll use it to get an exact lower bound on the replication rate r as a function of q. First, suppose we use p reducers, and the i^{th} reducer has $q_i \leq q$ bit strings that it will receive if they are present in the input. Since all the $(b/2)2^b$ pairs of strings at distance 1 must be covered by some reducer, we know that

$$\sum_{i=1}^{p} (q_i/2)\log_2 q_i \geq (b/2)2^b$$

That is, the sum of the maximum number of outputs that each reducer can cover must be at least the number of outputs.

We are going to replace $\log_2 q_i$ by $\log_2 q$ in the above inequality. Since q is an upper bound on q_i, the inequality must continue to hold; that is

$$\sum_{i=1}^{p} (q_i/2)\log_2 q \geq (b/2)2^b$$

Notice we chose not to replace the factor q_i by q.

The replication rate r is the sum of the number of inputs q_i to each reducer divided by the total number of possible inputs, 2^b, that is, $\Sigma_{(i=1)}^{p} q_i/2^b$. We can manipulate the inequality above so that exactly $\Sigma_{(i=1)}^{p} q_i/2^b$ appears on the left, and everything else is on the right. That gives us

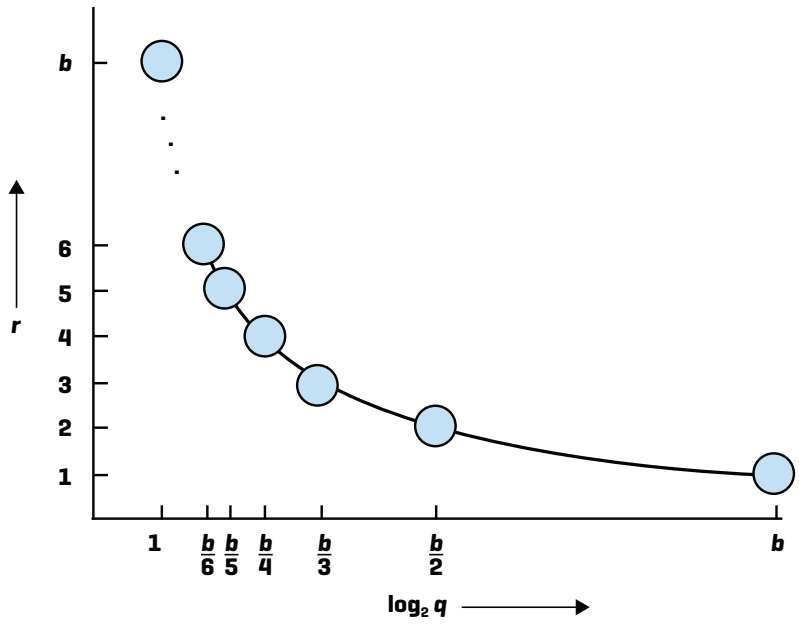
$$r = \sum_{i=1}^{p} q_i/2^b \quad \text{b}/\log_2 q$$

This inequality says for the problem of finding strings at Hamming distance 1, the replication rate is proportional to b, the string length, and inversely proportional to the logarithm of the maximum number of inputs that can be assigned to one reducer. Figure 2 shows the form of the lower bound on r and also shows points where we have algorithms that match the lower bound.

The algorithms at the endpoints are easy to see. If $\log_2 q = b$, then $q = 2^b$, which means that one reducer can get all the possible inputs. In that case, there is no need for any replication; that is, if $\log_2 q = b$, then $r = 1$ suffices.

At the other extreme, if $\log_2 q = 1$, that is, $q = 2$, then we need one reducer for each possible pair of strings at distance 1. Each string s must be sent to the b different reducers that correspond to pairs $\{s, t\}$ where t is one of the b strings at Hamming distance 1 from s. In terms of key-value pairs, the keys are pairs of strings at distance 1. The Map function generates from an input

Figure 2: Known algorithms matching the lower bound on replication rate.



string s the b key-value pairs with value s and key $\{s, t\}$, where t is one of the bit strings at distance 1 from s. Then the reducer for key $\{s, t\}$ looks at the list of values associated with this key, and if both s and t are present outputs that pair. Otherwise, it outputs nothing. (In fact, unless at least one of s and t is present on the input, this reducer will not even exist.)

The other points shown in Figure 1 represent variants of the "splitting" algorithm [8]. For any integer $k \geq 2$ that divides b, we can split the positions of b-bit strings into k equal parts. Let a reducer correspond to one of these k segments and a particular bit string of length $2^{(k-1)b/k}$ that can appear in all but that segment. A bit string s is sent to k different reducers. Start by deleting the first of the k segments from s and send s to the reducer corresponding to segment number 1 and the bits of s in all but segment 1. Then, starting from s again, drop the second segment and send s to the reducer corresponding to segment 2 and the bits of s that remain. Continue in this way for each of the k segments. For example, if $b = 6$, $k = 3$, and $s = 011100$, then send s to the three reducers:

1. Segment = 1 and string = 1100.
2. Segment = 2 and string = 0100.
3. Segment = 3 and string = 0111.

The replication rate is clearly $r = k$, and the number of bit strings that can be assigned to any reducer is the number of possible strings in any one segment, that is, $q = 2^{b/k}$. If we take logarithms, we get $\log_2 q = b/k$. Since $r = k$, we find $r = b/\log_2 q$ is an upper bound as well as a lower bound.

Triangle Finding. Another problem for which we can obtain closely matching upper and lower bounds on the replication rate as a function of the maximum input size for a reducer is finding the number of triangles in a large graph, such as the graph of a social network. We shall not go into the applications of triangle-finding, but intuitively, we expect that closely knit communities of friends would have many triangles. That is, whenever A is friends with both B and C, we would expect it is likely that B and C are also friends with each other. The most efficient MapReduce algorithm for finding triangles is from a technical report published last year [9]. On a graph with m edges, it uses total computation time $O(m^{(3/2)})$, which is the best possible according to Alon [10]. This MapReduce algorithm makes use of a serial algorithm for finding all triangles in time $O(m^{3/2})$, due to Schank's Ph.D. work [11], and the conversion of that algorithm to a MapReduce algorithm using the same total computation is from Suri and Vassilvitskii [12].

Suppose the m edges of a graph on n nodes are chosen so that each possible edge is equally likely to be chosen. If we run the algorithm using enough reducers so that the expected number of edges at any reducer is q, then the replication rate is $O(\sqrt{m/q})$. That is, each edge will be sent as the value of a key-value pair to that number of different reducers. We shall not give the argument here, but it is shown that $\Omega(\sqrt{m/q})$ is also a lower bound on the replication rate [7]; i.e., the algorithm mentioned gives, to within a constant factor, the lowest possible replication rate.

SUMMARY

We have tried to motivate the need to study MapReduce algorithms from the point of view of how they trade parallelism for communication cost. We represent the degree of parallelism by the upper limit on the number of inputs that one reducer may receive; the smaller this limit, the more potential parallelism. We represent communication cost by the replication rate, that is, the number of key-value pairs produced for each input. Depending on your computational resources and your network, you may prefer one of many different points along the curve that represents this tradeoff. As a result, it is interesting to discover lower bounds on the replication rate as a function of the reducer input size.

For two problems, finding strings at Hamming distance 1 and finding triangles in a graph, we gave lower bounds on replication rate r as a function of input size q that are tight. That is, there are algorithms for a wide variety of q values whose replication rate is, to within a constant factor, that given by the lower bound.

However, there are problems in a variety of domains for which optimal MapReduce algorithms have not been studied. Analyzing these problems requires deriving new lower bounds, designing algorithms that attain them, and choosing parameters to balance the tradeoff between communication and computation costs on modern computer architectures. By understanding such tradeoffs, we can design MapReduce algorithms that are efficient both in terms of wall-clock time and in terms of data movement.

References

[1] Dean, J. and Ghemawat, S. MapReduce: Simplified data processing on large clusters. In *Proceedings of the Sixth Conference on Symposium on Opearting Systems Design & Implementation* (San Francisco, Dec. 6-8, 2004). 137–150.

[2] White, T. *Hadoop: The Definitive Guide. Storage and Analysis at Internet Scale, Second Edition.* O'Reilly Media, Sebastopol, CA, 2011.

[3] Afrati, F.N., Borkar, V. R., Carey, M. J., Polyzotis, N., and Ullman, J. D. MapReduce extensions and recursive queries. In *Proceedings of the 14th International Conference on Extending Database Technolog* (Uppsala, Sweden, March 21-24). ACM Press, New York, 2011, 1–8.

[4] Rajaraman, A., and Ullman, J. D. *Mining of Massive Datasets*. Cambridge University Press, Cambridge, UK, 2011. Also available on-line at http://infolab.stanford.edu/~ullman/mmds.html.

[5] Amazon Elastic Compute Cloud (Amazon EC2). Amazon Inc., 2008.

[6] Kwon, Y., Balazinska, M., Howe, B., and Rolia, J. A. Skewtune: mitigating skew in MapReduce applications. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (Scottsdale, May 20-24). ACM Press, New York, 2012, 25–36.

[7] Afrati, F. N., Das Sarma, A., Salihoglu, S., and Ullman, J. D. Vision paper: Towards an understanding of the limits of MapReduce computation. CoRR, abs/1204.1754, 2012.

[8] Afrati, F. N., Das Sarma, A., Menestrina, D., Parameswaran, A. and Ullman, J. D. Fuzzy joins using MapReduce. In *Proceedings of the International Conference on Data Engineering* (Washington, D.C., April 1-5, 2012).

[9] Afrati, F. N., Fotakis, D., and Ullman, J. D. Enumerating subgraph instances using MapReduce. Technical report. Stanford University, December 2011. http://ilpubs.stanford.edu:8090/1020/.

[10] Alon, N. On the number of subgraphs of prescribed type of graphs with a given number of edges. *Israel Journal of Mathematics* 38, 1-2 (1981), 116–130.

[11] Schank, T. Algorithmic Aspects of Triangle-Based Network Analysis. Ph.D. Thesis. Universitat Karlsruhe (TH), 2007.

[12] Suri, S. and Vassilvitskii, S. Counting triangles and the curse of the last eeducer. In *Proceedings of the 20th International Conference on World Wide Web* (Hyderabad, India, Mar. 28-April 1). ACM Press, New York, 2011, 607–614.

Biography

Jeff Ullman is a retired professor of computer science at Stanford University. He has written textbooks covering automata theory, compilers, data structures and algorithms, database systems, and data mining. During his teaching career at Princeton and Stanford, he graduated 53 Ph.D. students, many of whom have become leaders in academia and industrial startups.