Performing complex analyses on large-scale data is becoming one of the core challenges of many application domains. As the volume of data grows, and the speed with which new data is generated increases, applications need to perform their computations on highly-parallel and shared-nothing distributed systems. My thesis focuses on new theories, algorithms, and systems that expand the capabilities of distributed large-scale data processing, with a particular emphasis on two problems: (1) performing large-scale joins of record-oriented data; and (2) processing large graphs. For join processing, my research started by formalizing a theoretical model for answering two questions within the context of the MapReduce system: (i) how difficult is it to parallelize different problems; and (ii) how optimal are existing algorithms for a given problem? Driven by the insights from studying the difficulty of parallelizing different fuzzy and equi-join problems and analyzing existing algorithms, I worked on new algorithms that have provably guarantees. For graph processing, my research has ranged from building an open-source platform for scalable graph processing, to algorithms for distributing graphs across machines, to developing extensions to existing graph processing systems with the goal of making them easier to program. I first describe my research activities and results to date (Sections 1 and 2), then plans for future work (Section 3).

1 Large-scale Join Processing

Google’s MapReduce and its open source implementation Hadoop have become widely-used platforms for large-scale data processing and analysis. Since their emergence, many MapReduce algorithms have been designed for performing relational joins in large-scale record-oriented data. All of these algorithms perform their computation in one MapReduce round, and little had been done to study rigorously the optimality of these algorithms in terms of: (1) communication between mappers and reducers; (2) maximum memory required by a worker; (3) latency in each of the map and reduce phases. My research to date on large-scale join processing has been along two lines:

1.1 Theoretical Framework for One-Round Computations

My coauthors and I developed a new theoretical framework to study the optimality of one-round MapReduce computations [3]. Using our framework, we identified a fundamental tradeoff between communication and memory. For relational joins of multiple tables, and fuzzy-joining the records of a table under certain metrics, we were able to answer the following question: If a one-round MapReduce algorithm uses at most $M$ memory in the reduce phase, what is the minimum communication it requires between mappers and reducers? We then generalized our framework to answer the same question for several other problems, such as dense matrix multiplication and counting triangles in a graph. We have also shown that for certain values of memory $M$, some existing and some new algorithms for these problems are optimal in their communication.

1.2 One-Round and Multi-Round Join Algorithms

The optimality proofs for one-round join algorithms showed that designing multi-round algorithms is the only way to perform these operations more efficiently. I developed three new multi-round algorithms that are more efficient than the one-round algorithm [1]. DYM is a distributed version of Yannakakis’s famous sequential algorithm for a subclass of join queries called $\alpha$-acyclic queries. For $\alpha$-acyclic queries, DYM’s communication cost is provably optimal (up to a constant factor) among any algorithm for any instance, the strongest optimality guarantee any algorithm can offer. GYM is a generalization of DYM to queries with bounded cyclicity, incurring a preprocessing cost that is a function of the degree of cyclicity of the query. For queries whose outputs are large enough, GYM is also optimal, but GYM can be suboptimal if the outputs are small. Finally, for general join queries, I developed a distributed version of another serial algorithm, whose communication
cost is worst-case optimal, i.e., it is less than any other algorithm's worst-case communication cost, but not necessarily less on every instance. In addition to joins, my coauthors and I have developed a two-round dense matrix multiplication algorithm that has better communication cost than any one-round algorithm [3].

Our framework also revealed gaps between existing one-round algorithms and what I could prove to be optimal for some problems. For example, for the problem of fuzzy-joining the records of a table, I tightened this gap by developing a new one-round fuzzy join algorithm [2] that has lower communication cost than any previous algorithm. However, my algorithm does not meet the lower-bound communication cost we were able to prove, so both better algorithms and tighter lower-bound proofs may be found to close the gap.

2 Large-Scale Graph Analysis

I have also worked extensively on the problem of how to efficiently process very large data sets that take the form of a graph. My approach has been to study existing systems, and the graph algorithms executed on these systems, understand their limitations, and offer alternative solutions. The overall goal is to provide systems and algorithms for large-scale graph processing that are easy to program and debug, while remaining efficient. My research in this area has consisted of five main thrusts:

2.1 The GPS System

Early in my graduate work I became interested in Google’s Pregel system for processing large-scale graphs. In Pregel, the vertices of the input graph are distributed across compute nodes and send each other messages in iterations to perform a computation. Similar to MapReduce, Pregel's programming interface is based around implementing a single function, `vertex.compute()`, which is executed on each vertex in parallel in each iteration. As a platform for my own research, I built an open-source Pregel clone, called GPS [6]. Around the same time we released GPS, an Apache open-source Pregel clone called Giraph was released by Yahoo! (currently developed at Facebook). Since its release, Giraph has adopted a number of GPS features, including an API extension called `master.compute()` and several storage and network layer optimizations. My subsequent work on large-scale graph-processing built on both GPS and Giraph, as well as on GraphX, a graph-processing system built on top of the Apache Spark data-processing system.

2.2 Graph Partitioning

In GPS and other distributed graph-processing systems, how the vertices of the graph are partitioned across compute nodes is an important decision: the partitioning of the graph determines the number of messages sent across the network. In Pregel, the default partitioning is random—vertices are allocated round-robin among the computing nodes—and the partitioning doesn’t change throughout the computation. My work on graph partitioning [6] studied the effect on overall performance of more sophisticated partitioning schemes, including static partitioning algorithms before the computation begins, and dynamic repartitioning based on observing computation behavior. The most sophisticated static partitioning schemes incurred significant preprocessing costs, but could decrease network I/O by an order of magnitude and run-time by 2.5x, while dynamic repartitioning achieved modest improvements without any preprocessing.

2.3 Distributed Graph Algorithms

Little work had been done on developing efficient algorithms in Pregel-like systems for fundamental graph problems. Some existing parallel graph algorithms are designed for the PRAM computation model and do not fit easily into Pregel's vertex-centric model. Others can be inefficient in Pregel's distributed environment due to high communication generated between vertices, or a very large number of iterations on some graphs. I developed and carefully optimized vertex-centric versions of a large suite of fundamental graph algorithms [8], including many existing algorithms, as well as new algorithms for strongly connected components, maximal matching, coloring, and minimum spanning tree. The algorithms were developed on our GPS system.
2.4 Higher-Level Languages and APIs

Large-scale graph processing systems typically expose a small set of functions, such as the `vertex.compute()` function of Pregel and GPS, or the `gather()`, `apply()`, and `scatter()` functions of the PowerGraph system. Some graph algorithms, such as PageRank, shortest paths, and connected components, can be specified naturally inside these vertex-centric functions, but when an algorithm requires a chain of vertex-centric and possibly global computations, long and convoluted programs result. Similar issues with the MapReduce framework have led to widely-used languages such as Pig Latin and Hive, which introduce higher-level data-processing primitives. My work on higher-level graph-processing primitives took an analogous approach. We studied the suite of distributed graph algorithms we had implemented on GPS and identified a set of primitives, called HeiP [7], that abstract the most commonly appearing distributed graph operations. We implemented HeiP primitives on top of GraphX and re-implemented our previous suite of algorithms along with many additional algorithms to verify our sense of the usefulness of HeiP primitives. We chose to implement HeiP on GraphX because it was the easiest platform to prototype higher-level primitives among the existing systems. Our experience has been that implementing algorithms using our primitives is more intuitive and much faster than using the APIs of existing distributed systems.

In another direction toward using higher-level languages for distributed graph computations, a student of Prof. Olukotun and I extended the compiler of Green-Marl, a domain-specific language for graph algorithms designed by Prof. Olukotun's group, to compile algorithms written in Green-Marl into GPS code [4]. Unlike HeiP, Green-Marl programs are written as if the graph is stored in a single machine, and the Green-Marl compiler automatically translates the user's program into `vertex.compute()` functions. One of our challenges was to identify the subset of Green-Marl, which we called Pregel-canonical programs, that can be compiled to a vertex-centric API.

2.5 The Graft Debugger For Apache Giraph

Algorithms running on small graphs can be debugged on a single machine, using a traditional debugger to set breakpoints, step forward and backward in the code, and visualize how the values of the vertices and edges are changing. However, debugging graph algorithms that run on large-scale distributed systems is very challenging, as the underlying engine, such as GPS or Giraph, executes many tasks across different machines in parallel. Together with two masters students I supervised, we interviewed existing users of GPS and Giraph and learned how they manually debug their programs. Based on our observations, we built a new debugger from scratch, called Graft [5], tailored specifically for the needs of GPS and Giraph users. By the time we had started our research on debugging, Giraph already had a strong user community, so we chose to develop Graft for Giraph, although it can easily be adopted to GPS and other Pregel-like systems. Graft's main technical contributions are: (1) a rich API for specifying which sets of vertices should be "captured" during execution; (2) a graph and iteration based GUI to visualize the captured vertices from iteration to iteration; and (3) tools for replicating "post-facto" the exact lines of `vertex.compute()` that executed for a particular vertex and iteration. Graft has been adopted by Apache Giraph's main code base.

3 Future Research

My research style so far has spanned purely theoretical (e.g., my work on the theoretical MapReduce framework [3] and one-round fuzzy joins algorithms [2]), to heavily systems-oriented (e.g., GPS [6], HeiP [7], and Graft [5]), to a combination of the two (e.g., my work on distributed graph algorithms [8] and multi-round join algorithms [1]). I enjoy all three styles and plan to continue in all of them to the extent possible. I believe the following set of topics will give more than enough to keep my research group busy in the first few years of my faculty career.

Join Optimization Based on New Algorithms: The recent algorithmic progress in distributed joins and the developments in large-scale data processing systems have not yet met. Current systems typically perform pairwise join plans, which are known to be suboptimal. It would be a significant advance to implement and experiment with the optimal one-round algorithm and our new multi-round algorithms, comparing their performance against each other and the join plans of existing systems. There are many dimensions to explore in the performance comparisons. Should we prefer different algorithms for different degrees of query cyclicity? Our current results suggest that the optimal one-round algorithm is only worse than our multi-round
algorithms at high levels of parallelism. Should we use one algorithm when using thousands of processors, and another one when using only dozens of processors? How should we handle skew? And finally, how do different execution environments, e.g., Hadoop-like disk-based systems, or Spark-like systems with caching features, affect our choice of algorithms?

Incremental and Streaming Join Algorithms: Many applications process data that arrives continuously in small batches or updates. On the theory side, I’m interested in developing distributed join processing algorithms that are suitable for a streaming or incremental environment. An important application of these algorithms is in distributed materialized views, a feature that is not yet supported in existing widely-used systems such as Hive, SparkSQL, or Impala. A natural next step after designing promising algorithms would be to try them out on existing systems with streaming capabilities, such as Microsoft’s Naiad and Spark’s Streaming engine.

Extreme Parallelism: Many of the existing algorithms and systems in today’s data-processing ecosystem have natural limits on how much parallelism they can exploit. For example, all of the existing join algorithms can exploit maximum parallelism levels that are functions of the domain sizes of the attributes of the join query. Similarly, graph processing systems can utilize as many processors as there are vertices, which may be orders of magnitude smaller than the number of edges. What would happen if we had millions of processors? I am interested in developing algorithms and systems for join and graph processing that can exploit what we call extreme parallelism.

Memory-Constrained Graph Processing: Current graph-processing systems assume that there is sufficient memory at each worker. Our observation from talking to programmers of existing systems and reading programmer postings is that this assumption fails quite often. We believe storage layers of distributed graph-processing systems should be more robust to working under limited memory, and for now the solution is to utilize disk. Three main operations may need to be handled carefully in a disk-based distributed graph engine: (1) the computation on the vertices, (2) a mix of memory and disk I/O for accessing the graph, vertex values and messages, and (3) the network I/O for sending and receiving messages across machines. One primary challenge to making the system efficient would be to effectively overlap each operation as much as possible to hide the latencies of the others.

Network Optimizations For Synchronous Data-Processing: Most of today’s distributed data and graph-processing systems, e.g., Spark, Hadoop, and Pregel, are synchronous engines and have a very simple communication topology: every machine writes and reads from every other machine. Batching of messages is typically the only optimization in the current implementation of these systems. I am interested in exploring other optimizations, for example: When a machine has multiple batches ready to send, does the order in which the messages are sent make a difference? When one machine has more workload than others, could we speed it up with a network layer optimization? Should the messaging be aware of the rack allocations of the machines? I believe answering these questions could yield simple yet effective optimizations for existing systems.

References