BIG DATA ANALYTICS: METHODS AND APPLICATIONS

by

Erik Steven Paulson

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2018

Date of final oral examination: 5/9/2018

The dissertation is approved by the following members of the Final Oral Committee:

AnHai Doan, Professor, Computer Sciences, UW-Madison

Suman Banerjee, Professor, Computer Sciences, UW-Madison

Paraschos Koutris, Assistant Professor, Computer Sciences, UW-Madison

Theodoros Rekatsinas, Assistant Professor, Computer Sciences, UW-Madison

Jonathan Eckhardt, Associate Professor and Robert Pricer Chair in Enterprise Development,
Weinert Center for Entrepreneurship, UW-Madison

© Copyright by Erik Steven Paulson 2018

All Rights Reserved

ACKNOWLEDGMENTS

This dissertation would not have been possible without the support of a great many people. What follows is a very incomplete list of some of those to whom I owe a great deal of gratitude.

First, I want to thank my partner Ben. Tradition is that you thank your family at the very end, but as I met Ben on the very first day of graduate school, I think it appropriate to mention him first. We've been through a couple of moves, a dog, a house, a marriage, and a few more years of grad school than either of us thought possible, and throughout it all he's been my rock. There have been times that I regretted going to graduate school, but every time I think that I remind myself that if not for graduate school, I would never had met him. He is, without a doubt, the best thing that I got from graduate school.

My advisor AnHai Doan deserves a tremendous amount of thanks, more than I can probably ever give. It is fair to say that without him, I would never have finished this degree. It has been a joy working with AnHai, even if I have grumbled to Ben a few times that AnHai is insisting I do things a certain way. (Inevitably, AnHai's approach was correct.) I have learned more from watching him approach a blank whiteboard and work out a story than I have from reading hundreds of papers. At the end, his whiteboard will lay out – in amazing clarity – how to tell that story and what evidence, claims, or questions to answer and ask are necessary to fully support it. I do not always demonstrate to him that I can do the same, but I have a model to strive for.

I must also thank Jeff Naughton, who was my advisor for much of my time at the UW. Jeff is an amazing teacher. I thought I had no interest in databases, but having the opportunity to take 564 and 764 from him in my first year of graduate school completely changed my mind. I owe him further thanks for taking me on to the CondorDB project and giving me the opportunity to work with the database group. It is fair to say that it changed my life, and I learned so much about research, teaching, and being an excellent person from my time with Jeff.

I also want to thank professors Remzi Arpaci-Dusseau and Bill Cronon for their advice, especially early on in my graduate studies when I was working through some rough patches.

As a database student at Wisconsin, you never really have just one advisor, but you're a member of the entire group, so I wish to also thank David DeWitt, Raghu Ramakrishnan, Jignesh Patel, and Chris Ré for the time they spent with me. I am so very fortunate to have had opportunities to learn from all of them. Special thanks must go to David, who has opened many doors for me, and without whom I would not have been able to be a part of the work that makes up a large portion of this dissertation. I was there when he was drafting the blog posts that kicked it all off, and I'm glad he didn't listen to me when I suggested that maybe it was a little harsh. Overall, I regret that I have not had much chance to spend time with Paris Koutris and Theo Rekatsinas – they both do such cool work that I half wish that I could spend another few years in graduate school to work with

them, but I take some comfort in knowing that I will be joining the larger UW Madison Database Alumni community, and that there is a tight bond between our department and our alumni.

Speaking of alumni, I am so very lucky to have had such excellent collaborators to work with while at the UW. From the CondorDB days with Jiansheng Huang, Ameet Kini, Christine Reilly, Eric Robinson, Srinath Shankar, and Lakshmikant Shrinivas, to more recent work with Yash Govind, Pradap Konda, and Paul Suganthan, I could not have asked for better collaborators and friends. Yash in particular has been my collaborator for the work in entity matching, often taking on far more of the coding burden than was probably fair. He has been a great friend to work with. Others who worked with us on the development of CloudMatcher include Ali Hitawala, Mukilan Ashok, and Palani Nagarajan, and I am indebted to each of them. I am also very fortunate to have had excellent collaborators outside of the UW in Mike Stonebraker, Sam Madden, Dan Abadi, Andy Pavlo, and Alex Rasin. To have had the opportunity to work with just one of them would have been extraordinary, to have worked with all of them, well, it's beyond extraordinary.

I want to thank my oral defense committee for taking the time to help me take the last step and finish the degree. I want to especially thank Jon Eckhardt. I took his class early on in graduate school, and I learned a ton about how the business world works. It has given me a new frame to understand some of the motivations behind decisions, and it was one of the very best classes I ever took at the UW.

I was lucky to have spent several years as a member of the Condor team and with Miron Livny, and even today I still consider myself a member of the team, whose input is appreciated and valued. From Miron, I learned much about how to build systems and software for the long haul, and above all else, the importance of understanding the failure cases. I am certain that I annoy the heck out of coworkers when I ask "what happens if this goes wrong" or "what happens if this can't connect to that", but they are crucial questions, and no one asks them like Miron.

I've been an undergraduate, a member of the research staff, and a graduate student in the Computer Sciences department, so a full list of friends I've made over the years from the department and university would be far too long to put into this section, so what follows necessarily has to leave many good friends out – to them, my apologies. But, special thanks to the following for their friendship: Brad Beckmann, Thien Tran, Tim Denehy, Zach Miller, Jen Beckmann, Allison Holloway, Alan Halverson, Ameet Soni, Todd Tannenbaum, Greg Thain, Nathan Burnett, Philip Brenner, David Parter, Jeff Wright, Suchita Shah, Jason Smathers, Joey Labuz, Tyler Junger, and Kurt Gosselin.

I appreciate the support and flexibility my company has provided me during while I finished my graduate work. In particular, I am indebted to Youngchoon Park for saying "yes" when I said I need to take a leave for a month to wrap up this work, and for his support in keeping my job interesting and varied.

I do want to thank the rest of my family for all of the love and support they've given me over the years, including knowing not to ask me how it's going. I know that they will be my biggest cheering section when I tell them "it's done".

Finally, I want to thank two teachers that aren't here to share this moment. AIDS took Al, and a reckless semi-truck driver took Doug. I hope I have made you both proud.

TABLE OF CONTENTS

Page

LIST OF TABLES v

LIST OF FIGURES vi

ABSTRACT viii

1 Introduction 1

1.1 Contributions and Outline of This Dissertation 2

2 MapReduce and Parallel Databases:
A Comparison of Two Approaches to Data Analysis 5

2.1 Introduction 5
2.2 Two Approaches to Large-Scale Data Analysis 7
2.2.1 MapReduce 7
2.2.2 Parallel DBMSs 8
2.3 Architectural Elements 9
2.3.1 Schema Support 10
2.3.2 Indexing 11
2.3.3 Programming Model 12
2.3.4 Data Distribution 13
2.3.5 Execution Strategy 14
2.3.6 Flexibility 14
2.3.7 Fault Tolerance 15
2.4 Performance Benchmarks 15
2.4.1 Benchmark Environment 16
2.4.2 The Original MR Benchmark 19
2.4.3 Analytical Tasks 26
2.5 Discussion 38
2.5.1 System Aspects 38
2.5.2 User-level Discussion 44
2.6 Conclusions 46

Page

3 CloudMatcher: A Cloud/Crowd Service for Entity Matching 50

3.1 Introduction 50
3.2 Preliminaries 53
3.3 The Corleone & Falcon Systems 55
3.3.1 The Corleone System 55
3.3.2 The Falcon System 57
3.4 The CloudMatcher Service 60
3.4.1 Motivations and Goals 60
3.4.2 Limitations of Current Solutions 63
3.4.3 Key Ideas of the CloudMatcher Solution 64
3.4.4 The Entity Matching Workflow of CloudMatcher 65
3.4.5 Partitioning the Entity Matching Workflow 67
3.4.6 Executing the Workflow Fragments 68
3.4.7 The User Interaction/Crowd Engines 68
3.5 Workflows as Composites of Services 70
3.6 Deployment and Architecture 71
3.7 Experience with CloudMatcher 74
3.7.1 Entity Matching on Real-World Datasets 74
3.7.2 Scaling to Support a Campus-wide Entity Matching Service 77
3.7.3 Using Basic Services in Novel Ways 83
3.8 Lessons Learned 87
3.9 Conclusions 90

4 Related Work 92

5 Conclusion 97

Bibliography 101

LIST OF TABLES

Table Page

3.1 The results of applying CloudMatcher to several representative datasets. 75

3.2 Total Time, Queue Time, and Run Times of the “Bursty” experiments. The time is in hours:minutes:seconds 80

3.3 Min and Max Total Times, Queue Times, and Run Times of the “Bursty” experiments. The time is in hours:minutes:seconds 81

LIST OF FIGURES

Figure Page

2.1 Load Times – Grep Task Data Set (535MB/node) 21

2.2 Load Times – Grep Task Data Set (1TB/cluster) 22

2.3 Load Times – UserVisits Data Set (20GB/node) 23

2.4 Grep Task Results – 535MB/node Data Set 24

2.5 Grep Task Results – 1TB/cluster Data Set 25

2.6 Selection Task Results 28

2.7 Aggregation Task Results (2.5 million Groups) 30

2.8 Aggregation Task Results (2000 groups) 30

2.9 Join Task Results 32

2.10 UDF Aggregation Task Results 36

3.1 Examples of EM across two tables. 51

3.2 Most current EM solutions consist of a blocking step and a matching step. 53

3.3 The EM workflow of Corleone. 55

3.4 (a) A decision tree learned by Corleone and (b) blocking rules extracted from the tree. 58

3.5 The EM workflow of Falcon as a DAG of basic operators. 59

3.6 CloudMatcher as a cloud/crowd EM service. 61

3.7 The CloudMatcher architecture 64

Figure Page

3.8 A refinement of the Falcon DAG, to create the workflow for CloudMatcher. The resulting workflow consists of three parts, as shown in (a)-(c). 66

3.9 A partitioning of the first part of the CloudMatcher workflow into interactive and batch fragments. 68

3.10 The services of CloudMatcher. 70

3.11 CloudMatcher Head Node Diagram. Components in blue are software written for CloudMatcher 72

3.12 CloudMatcher Worker Node Diagram 72

3.13 Uniform arrival rate, jobs arriving for a minimum of 24 hours 78

3.14 “Bursty” run, 80 submitted jobs 81

3.15 “Bursty” run, 120 submitted jobs 82

3.16 “Bursty” run, 160 submitted jobs 82

3.17 “Bursty” run, 200 submitted jobs 83

3.18 Defining custom blocking rules in CloudMatcher 84

3.19 Crowdsourcing tuple labels on Amazon MTurk 85

3.20 Basic services to be invoked for the cross validation workflow 86

3.21 Setting parameters for the cross validation service in CloudMatcher 86

3.22 Example cross validation service results in CloudMatcher 87

ABSTRACT

Big Data is now pervasive. This has driven a critical need to develop novel methods to store and process data at large scale, as well as to develop new applications to use and make sense of this data. In this dissertation, we make two contributions toward addressing this need. First, we study methods for large-scale data analysis. In particular, we compare the popular MapReduce model to parallel relational database management systems, and empirically analyze their strengths and weaknesses. The MapReduce (MR) paradigm for large-scale data analysis has received significant attention [86]. Although the basic control flow of this framework has existed in parallel relational database management systems (DBMSs) for over 20 years, some have called MR a dramatically new computing model [40, 86]. We evaluate both kinds of systems in terms of performance and development complexity. To this end, we define a collection of benchmarks that we have run on an open-source version of MR as well as on two parallel DBMSs. For each benchmark, we measure each systems performance for various degrees of parallelism on a cluster of 100 shared-nothing nodes. Our results reveal some interesting trade-offs. We speculate about the causes of the dramatic performance difference and consider implementation concepts that future systems should take from both kinds of architectures.

In the second contribution, we examine how Big Data scaling methods, such as MapReduce, can be used to build a scalable and flexible cloud-based entity matching applications, and what lessons can be learned from this effort for the future development of similar applications. Entity matching (EM) finds disparate data instances that refer to the same real-world entity. EM has been long studied and is crucial to many fields, and will become even more so in the age of Big

Data and data science. However, it is still very difficult for domain scientists to use such EM systems, especially at scale. In response, we have developed CloudMatcher, a cloud/crowd service for EM. CloudMatcher aims to be a fast, easy- to-use, scalable, and highly available EM service on the Web. As far as we can tell, no such application has been developed for EM in the data management research community. We describe CloudMatcher's development in the past two years and its deployment in the past six months, providing a detailed analysis of its performance over several representative datasets and in several scale-up experiments, and discussing lessons learned. Taken together, our contributions in this dissertation advance the topic of Big Data analytics, for both aspects of methods and applications.

Chapter 1

Introduction

Big Data is now pervasive. From astronomy to zoology, from agriculture to manufacturing, and from government to entertainment to healthcare, the ability to store and process large amounts of data has touched virtually all fields. Advances in sensor technologies and other data collection methods, the continued plummeting cost of compute and storage technologies, and the connection of several billion more people to the Internet will only increase the importance and expectations of data.

In order to make sense of this data and to build applications to use this data, there have been questions as to what systems we should use to store and process it. Do we use existing systems with long histories, such as parallel relational databases, or do we use newer systems, such as MapReduce, Spark, NoSQL storage systems, or DataFlow systems? If we do use or build new systems, what from existing systems can inform the new systems, and what improvements in the new systems can flow back to existing systems? And then, once equipped with effective tools to store and process data, how can application developers take advantage of these new tools?

In this dissertation, we address aspects of both of these questions to better understand big data analytics. Specifically, we focus on the following two research questions. The first question is about methods for Big Data. We focus in particular on **a comparison of MapReduce and parallel relational databases**. There has been considerable enthusiasm around the MapReduce (MR) paradigm for large-scale data analysis. At the same time, parallel relational databases (RDBMSs) have long been available, can be used for much of the same processing work as MR is being used for, and can run on identical hardware. So, how does one determine if they should use MapReduce or a parallel RDBMS? We evaluate both MR and parallel RDBMSs in terms of performance and

development complexity. To this end, we define a collection of benchmarks that we have run on an open-source version of MR as well as on two parallel DBMSs. For each benchmark, we measure each systems performance for various degrees of parallelism on a cluster of 100 shared-nothing nodes. We examine the performance differences and implementation concepts underlying both systems, and how each system can inform the other, as well as future systems.

The second research question is about applications of Big Data. We focus in particular on **developing an entity matching application in the cloud, over a machine cluster.** Given that we now have systems that can effectively process large amounts of data, what is involved in building an application that takes advantage of these systems? As an interesting example, we build and study a system to do entity matching in the cloud. Entity matching (EM) finds disparate data instances that refer to the same real-world entity, and is critical to many domains, from health care to e-commerce to knowledge based construction and virtually any domain that involves combining data from different sources. EM is well-studied, but until recently there were few if any "end-to-end" systems. Existing systems addressed part of the EM process, but assumed the user was a sophisticated developer well-versed in the EM field. For domain users who may have EM needs but lack the background, taking advantage of these systems is difficult. Recent work has addressed these limitations with systems like Corleone, which demonstrates a "hands-off", "end-to-end" workflow, and Falcon, which demonstrates how to scale up Corleone [56, 39]. Our system, CloudMatcher aims to be a fast, easy-to-use, scalable, and highly available EM service on the Web. We describe its development and deployment in the past two years, providing a detailed analysis of its performance over several representative datasets and in several scale-up experiments. Taken together, our contributions in this dissertation advance the topic of Big Data analytics, for both aspects of methods and applications.

1.1 Contributions and Outline of This Dissertation

To summarize, in this dissertation I make the following contributions:

- **A Big Data BenchMark:** I help develop a benchmark of Big Data analytics tasks that are common across many fields. This benchmark was among the first benchmarks being developed for Big Data, and since has proven influential. It has been incorporated into Big Data benchmarks developed by Intel as part of their HiBench suite [65] and the UC-Berkeley AMPLab Big Data Benchmark. [5]

- **A Large-Scale Deployment and An Extensive Set of Experiments:** I provide a large-scale deployment of a cluster of 100 share-nothing nodes, designed, and carried out extensive experiments on this cluster to compare MapReduce and a set of parallel RDBMSs.

- **Findings on the Strengths and Weaknesses of the Two Big Data Methods:** I provide a set of findings and detailed analysis on the strengths and weaknesses of MR and RDBMSs. Overall, we find that each system has things to learn from the other, and future big data processing systems should retain the lessons learned from parallel RDBMSs and make them available to their users.

- **An Architecture for a Cloud-Based Entity Matching Service:** I develop a novel solution architecture for CloudMatcher, to bring the Falcon and Corleone workflows into a multi-user, scalable cloud service.

- **A Solution for Giving Users Flexibility in Entity Matching Workflows:** Instead of implementing a single end-to-end EM workflow, I show that it is much better to break the workflow into a set of basic services, then implement those. This way the user can mix and match the services and compose them to flexibly build novel EM workflows.

- **Development of A Cloud-Based Entity Matching Service:** Over the past two years, I have contributed to developing an industrial-strength cloud-based entity matching service, using the above architecture. As far as I can tell, this is the first such service being developed in academia. It is also the first hands-off EM service on the cloud. It has been used at UW-Madison in domain sciences and is planned for deployment at American Family Insurance and Johnson Controls.

- **An Extensive Set of Experiments to Evaluate CloudMatcher:** I provide an extensive set of experiments that demonstrate that (a) CloudMatcher can effectively perform EM on a broad variety of datasets, (b) we can scale CloudMatcher to run hundreds of jobs per day using moderate resources, thereby making it a reasonable solution to deploy at an organization such as UW-Madison, and (c) users can use the basic services provided by CloudMatcher to deploy and execute a variety of EM workflows.

- **A Discussion of Limitations and Opportunities for Future Research:** I provide an extensive discussion of limitations and future research directions, based on our experiments with CloudMatcher. CloudMatcher lays down a foundation for future work. We have designed it to be easy to extend and add additional services, or to study new variations of existing services and workflows. CloudMatcher is not limited to just entity matching, but can be used as a system to study other parts of the data cleaning and integration field.

The work on CloudMatcher is carried out in collaboration with Yash Govind and several other colleagues. Parts of this dissertation have been published in conferences. The MR vs RDBMS study has been published in SIGMOD 2009 [87], which has received 1,230 citations as of April 30, 2018. The work on CloudMatcher has been published in the Big data analytics-as- a-Service: Architecture, Algorithms, and Applications in Health Informatics workshop in KDD 2017 [58].

The rest of this dissertation is organized as follows. Chapter 2 compares MapReduce and parallel databases. Chapter 3 describes CloudMatcher. Chapter 4 describes related work. Chapter 5 concludes this dissertation and describes future research directions.

Chapter 2

MapReduce and Parallel Databases: A Comparison of Two Approaches to Data Analysis

2.1 Introduction

"Cluster computing" is a computing paradigm that entails harnessing large numbers of (low-end) processors working in parallel to solve a computing problem. In effect, this suggests constructing a data center by lining up a large number of low-end servers instead of deploying a smaller set of high-end servers. With this rise of interest in clusters has come a proliferation of tools for programming them. One of the earliest and best known such tools in MapReduce (MR) [40]. MapReduce is attractive because it provides a simple language through which users can express relatively sophisticated distributed programs, leading to significant interest in industry and academia.

Given this interest in MapReduce, it is natural for the database community to ask "Why not use a parallel DBMS instead?" Parallel database systems (which all share a common architectural design) have been commercially available for nearly two decades, and there are now about a dozen in the marketplace, including Teradata, Netezza, Microsoft SQL Server Parallel Data Warehouse, Dataupia, Vertica, ParAccel, Neoview, Greenplum, DB2 (via the Database Partitioning Feature), and Oracle (via Exadata). They are robust, high performance computing platforms. Like MapReduce, they provide a high level programming environment and parallelize readily. Though it may seem that MR and parallel databases target different audiences, it is in fact possible to write almost any parallel processing task as either a set of database queries (possibly using user defined functions and aggregates to filter and combine data) or a set of MapReduce jobs. Inspired by this

question, our goal is to understand the differences between the MapReduce approach to performing large scale data analysis and the approach taken by parallel database systems. The two classes of systems make different choices in several key areas. For example, all DBMSs require that data conform to a well-defined schema, whereas MR permits data to be in any arbitrary format. Other differences also include how each system provides indexing and compression optimizations, programming models, the way in which data is distributed, and query execution strategies.

The purpose of this chapter is to consider these choices, and the trade-offs that they entail. We begin in Section 2.2 with a brief review of the two alternative classes of systems, followed by a discussion in Section 2.3 of the architectural trade-offs. Then, in Section 2.4 we present our benchmark, one from the MR paper [40], and the rest a collection of more demanding tasks. In addition, we present the results of running the benchmark on a 100 node cluster to execute each task. We tested the publicly available open-source version of MapReduce, Hadoop [9], against two parallel SQL DBMSs: Vertica [14] and a second system from a major relational vendor. We also report results on the time each system took to load the test data and report informally on the length of time it took to set up each benchmark and tune the software.

In general, the SQL DBMSs were significantly faster and required less code to implement each task, but took longer to tune and load the data. Hence, we conclude in Section 2.5 with a discussion on the reasons for the differences between the approaches and provide suggestions on the best practices for any large scale data analysis engine.

Some readers may feel the experiments conducted using 100 nodes are not interesting or representative of real world data processing systems. We disagree with this conjecture on two points. First, as we demonstrate in Section 2.4, at 100 nodes the two parallel DBMSs range from a factor of 3.1 to 6.5 faster than MapReduce on a variety of analytic tasks. While MR may indeed be capable of scaling to 1000s of nodes, the superior efficiency of modern DBMSs alleviates the need to use such massive hardware on datasets in the range of 1-2PB (1000 nodes with 2TB of disk/node has a total disk capacity of 2PB). For example, eBay's Teradata configuration uses just 72 nodes (two quad-core CPUs, 32GB RAM, 104 300GB disks per node) to manage approximately 2.4PB of relational data. As another example, Fox Interactive Media's warehouse is implemented using

a 40-node Greenplum DBMS. Each node is a Sun X4500 machine with two dual-core CPUs, 48 500GB disks, and 16 GB RAM (1PB total disk space)[35].

2.2 Two Approaches to Large-Scale Data Analysis

The two classes of systems we consider in this chapter run on a "shared nothing" collection of computers [93]. That is, the system is deployed on a (possibly large) collection of independent machines, each with local disk and local main memory, connected together on a high-speed local area network. Both systems achieve parallelism by dividing any dataset to be utilized into **partitions**, which are allocated to different nodes to allow processing to occur in parallel. In this section, we provide an overview of how both the MR model and traditional parallel DBMSs operate in this environment.

2.2.1 MapReduce

One of the attractive qualities about the MapReduce programming model is its simplicity: an MR program consists only of two functions, called **Map** and **Reduce**, that are written by a user to process key/value data pairs. The input data set is stored in a collection of partitions in a distributed file system deployed on each node in the cluster. The program is then injected into a distributed processing framework and processing occurs in a manner to be described.

The Map function reads a set of "records" from an input file, does any desired filtering and/or transformations, and then outputs a set of intermediate records in the form of new key/value pairs. As the Map function produces these output records, a "split" function partitions the records into R disjoint buckets by applying a function to an attribute of each output record. This split function is typically a hash function, though any deterministic function will suffice. The buckets each Map function produce are written to files on the processing node's local disk. The Map function terminates having produced R output files, one for each bucket.

In general, there are multiple instances of the Map function running on different nodes of a compute cluster. We use the term *instance* to mean a unique running invocation of either the Map or Reduce function. Each Map instance is assigned a distinct portion of the input file by the MR

scheduler to process. If there are M such distinct portions of the input file, then there are R files on disk storage for each of the M Map tasks, for a total of $M \times R$ files; $F_{ij}, 1 \leq i \leq M, 1 \leq j \leq R$. The key observation is that all Map instances use the same hash function; thus, all output records with the same hash value are stored in the same output file.

The second phase of a MR application executes R instances of the Reduce program. Typically there is only one Reduce instance executed for each node in the cluster. The input for each Reduce instance R_j consists of the files $F_{ij}, 1 \leq i \leq M$. These files are transferred over the network from the Map nodes' local disks. Note that again all output records from the Map phase with the same hash value are consumed by the same Reduce instance, regardless of which Map instance produced the data. Each Reduce processes or combines records assigned to it in some way, and then writes records to an output file (in the distributed file system), which forms part of the final output of the MR computation.

It should also be noted that both the Map and Reduce functions are just arbitrary computations written in a general purpose language. Therefore, it is possible for each task to do anything on its input, just as long as its output follows the conventions defined by the model.

The input data set exists as a collection of one or more partitions in the distributed file system. It is the job of the MR scheduler to decide how many Map instances to run and how to allocate them to available nodes. Likewise, the scheduler must also decide on the number and location of nodes running Reduce instances. The MR central controller is responsible for coordinating the system activities on each node. A MR application finishes execution once the final result is written as new files in the distributed file system.

2.2.2 Parallel DBMSs

Database systems capable of running on clusters of "shared nothing" nodes have existed since the late 1980's. These systems all support standard relational tables and SQL, and thus the fact that the data is stored on multiple machines is transparent to the end-user. Many of these systems build on the pioneering research from the Gamma [44] and Grace [51] parallel DBMS projects. The two key aspects that enable parallel execution are that (1) most (or even all) tables are partitioned

over the nodes in a cluster and that (2) the system uses an optimizer that translates SQL commands into a query plan whose execution is divided amongst multiple nodes. Because programmers only need to specify their goal in a high level language, they are not burdened by the underlying storage details, such as indexing options and join strategies.

Consider a SQL command to filter the records in a table T_1 based on a predicate, along with a join to a second table T_2 with an aggregate computed on the result of the join. A basic sketch of how this command is processed in a parallel DBMS consists of three phases. Since the database will have already stored T_1 on some collection of the nodes, partitioned on some attribute, the filter sub-query is first performed in parallel at these sites similar to the filtering performed in a Map function. Following this step, one of two common parallel join algorithms are employed based on the size of data tables. For example, if the number of records in T_2 is small, the DBMS could replicate it on all nodes when the data is first loaded. This allows the join to execute in parallel at all nodes. Following this, the aggregate is then computed by each node using its portion of the answer to the join. A final "roll-up" step is required to compute the final answer from these partial aggregates [43].

If the size of the data in T_2 is large, then T_2's contents will be distributed across multiple nodes. If these tables are partitioned on different attributes than those used in the join, the system will have to hash both T_2 and the filtered version of T_1 on the join attribute using a common hash function. The redistribution of both T_2 and the filtered version of T_1 to the nodes is similar to the processing that occurs between the Map and the Reduce functions. Once each node has the necessary data, it then performs a hash join and calculates the preliminary aggregate function. Again, a roll-up computation must be performed as a last step to produce the final answer for the client.

At first glance, these two approaches to data analysis and processing have many common elements; however, there are notable differences that we consider in the next section.

2.3 Architectural Elements

In this section, we consider aspects of the two system architectures that are necessary for processing large amounts of data in a distributed environment. One theme in our discussion is that

the nature of the MR model is well-suited for development environments with a small number of programmers and a limited application domain. This lack of constraints, however, may not be appropriate for longer-term and larger-sized application development.

2.3.1 Schema Support

Parallel DBMSs require data to fit into the relational paradigm of rows and columns. In contrast, the MR model does not require that data files adhere to a schema defined using the relational data model. That is, the MR programmer is free to structure their data in any manner or even to have no structure at all.

One might think that the absence of a rigid schema automatically makes MR the preferable option. For example, SQL is often criticized for its requirement that the programmer must specify the "shape" of the data in a data definition facility. On the other hand, the MR programmer must often write a custom parser in order to derive the appropriate semantics for their input records, which is at least an equivalent amount of work. But there are also other potential problems with not using a schema for large data sets.

Whatever structure exists in MR input files must be built into the Map and Reduce programs. Existing MR implementations provide built-in functionality to handle simple key/value pair formats, but support for more complex data structures, such as compound keys, must be explicitly written by the programmer. This is possibly an acceptable approach if a MR data set is not accessed by multiple applications. If such data sharing exists, however, a second programmer must decipher the code written by the first programmer to decide how to process the input file. A better approach, followed by all SQL DBMSs, is to separate the schema from the application and store it in a set of system catalogs which can be queried.

But even if the schema is separated from the application and made available to multiple MR applications through a description facility, the developers must also agree on a single schema. This obviously requires some commitment to a data model or models, and the input files must obey this commitment as it is cumbersome to modify data attributes once the files are created. The DBMS community has still not coalesced on how general such a facility should be when input data is not

rigidly structured (semi-structured data, e.g. XML). Although it would require considerably more space than we have available here to consider this debate, it suffices to say that sharing requires a "schema definition language" and that it is a complex decision to decide how general to make this capability.

Once the programmers agree on the structure of data, something or someone must ensure that any data added or modified does not violate integrity or other high-level constraints (e.g., employee salaries must be non-negative). Such conditions must be known and explicitly adhered to by all programmers modifying a particular data set; an MR framework and its underlying distributed storage system has no knowledge of these rules, and thus allows input data to be easily corrupted with bad data. By again separating such constraints from the application and enforcing them automatically by the run time system, as is done by all SQL DBMSs, the integrity of the data is enforced without additional work on the programmer's behalf.

In summary, when no sharing is anticipated, the MR paradigm is quite flexible. If sharing is needed, however, then we argue that it is advantageous for the programmer to use a data description language and factor schema definitions and integrity constraints out of application programs. This information should be stored in a common set of system catalogs accessible to the appropriate users and applications.

2.3.2 Indexing

All modern DBMSs use hash or B-tree indexes to accelerate access to data. If one is looking for a subset of records (e.g., employees with a salary greater than $100,000), then using a proper index reduces the scope of the search dramatically. Database systems also support multiple indexes per table. Thus, the query optimizer can decide which index to use for each query or whether to simply perform a brute-force sequential search.

Because the MR model is so simple, frameworks do not provide built-in indexes. The programmer must implement any indexes that they may desire to speed up access to the data inside of their application. This not easily accomplished, as the framework's data fetching mechanisms must also be instrumented to use these indexes when pushing data to running Map instances. Once more, this

is an acceptable strategy if the indexes do not need to be shared between multiple programmers, despite requiring every MR programmer re-implement the same basic functionality.

If sharing is needed, however, the specifications of what indexes are present and how to use them must be transferred between programmers. It is again preferable to store index information in a standard format in the system catalogs, so that programmers can query this structure to discover such knowledge.

2.3.3 Programming Model

During the 1970's, the database research community engaged in a contentious debate between the relational advocates and the Codasyl advocates [90]. The salient issue of this discussion was whether a program to access data in a DBMS should be written:

1. By stating what you want – rather than presenting an algorithm for how to get it (Relational)

2. By presenting an algorithm for data access (Codasyl)

In the end, the former view prevailed and the last several decades is a testament to the value of relational database systems. Programs in high-level languages, such as SQL, are easier to write, easier to modify, and easier for a new person to understand. Codasyl was criticized for being "the assembly language of DBMS access". We argue that MR programming is somewhat analogous to Codasyl programming: one is forced to write algorithms in a low-level language in order to perform record-level manipulation. On the other hand, to many people brought up programming in procedural languages, such as C/C++ or Java, describing tasks in a declarative language like SQL can be challenging.

Anecdotal evidence from the MR community suggests that there is widespread sharing of MR code fragments to do common tasks, such as joining data sets. To alleviate the burden of having to re-implement repetitive tasks, the MR community is migrating higher level languages on top of the current interface to move such functionality into the run time. Pig [83] and Hive [10] are two notable projects in this direction.

2.3.4 Data Distribution

The conventional wisdom for large-scale databases is to always send the computation to the data, rather than the other way around. In other words, one should send a small program over the network to a node, rather than importing a large amount of data from the node. Parallel DBMSs use knowledge of data distribution and location to their advantage: a parallel query optimizer strives to balance computational workloads while minimizing the amount data transmitted over the network.

Aside from the initial decision on where to schedule Map instances, a MR programmer must perform these tasks manually. For example, suppose a user writes a MR application process a collection of documents in two parts. First, the Map function scans the documents and creates a histogram of frequently occurring words. The documents are then passed to a Reduce function that groups files by their site of origin. Using this data, the user, or another user building on the first user's work, now wants to find sites with a document that contains more than five occurrences of the word 'Google' or the word 'IBM'. In the naive implementation of this, where the Map is executed over the accumulated statistics, the filtration of documents to find those contain 'Google' or 'IBM' is done after the statistics for all documents are computed and shipped to reduce workers, even though only a small subset of documents satisfy the keyword filter.

In contrast, the following SQL view and select queries perform a similar computation:

```
CREATE VIEW keywords AS
SELECT siteid, docid, word, COUNT(*) AS wordcount
  FROM Documents
 GROUP BY siteid, docid, word

SELECT DISTINCT siteid
  FROM Keywords
 WHERE (word = 'IBM' OR word = 'Google')
   AND wordcount > 5
```

A modern DBMS would rewrite the second query such that the view definition is substituted for the Keywords table in the `FROM` clause. Then, the optimizer can push the `WHERE` clause in the query

down so that it is applied to the Documents table before the `COUNT(*)` is computed, substantially reducing computation. If the documents are spread across multiple nodes, then this filter can be applied on each node before documents belonging to the same site are grouped together, generating significantly less network I/O.

2.3.5 Execution Strategy

There is a potentially serious performance problem related to MR's handling of data transfer between Map and Reduce jobs. Recall that each of the M map instances produces R output files, each destined for a different Reduce instance. These files are written to the local disk on the node executing each particular Map instance. If M is 1000 and R is 500, the Map phase produces 500,000 local files. When the Reduce phase starts each of the 500 Reduce instances needs to read its 1000 input files, and must use a file-transfer protocol to "pull" each of its input files from the nodes on which the Map instances were run. With 100s of Reduce instances running simultaneously, it is inevitable that two or more Reduce instances will attempt to read their input files from the same map node simultaneously – inducing large numbers of disk seeks and slowing the effective disk transfer rate significantly. This is why parallel database systems do not materialize their split files and use a push approach to transfer data instead of pull.

2.3.6 Flexibility

Despite its widespread adoption, SQL is routinely criticized for its insufficient expressive prowess. Some believe that it was a mistake for the database research community in the 1970's to focus on data sub-languages that could be embedded in any programming language rather than adding high-level data access to all programming languages. Fortunately, new application frameworks, such as Ruby on Rails [98] and LINQ [79], have started to reverse this situation by leveraging new programming language functionality to implement an object-relational mapping pattern. These programming environments allow developers to benefit from the robustness of DBMS technologies without the burden of writing complex SQL commands.

Proponents of the MR model argue that SQL does not facilitate the desired generality that MR provides. But almost all of the major DBMS products (commercial and open-source) now provide support for user-defined functions, stored procedures, and user-defined aggregates in SQL. Although this does not have the full generality of MR, it does substantially improve the flexibility and generality of database systems.

2.3.7 Fault Tolerance

The MR frameworks provide a more sophisticated failure model than parallel DBMSs. While both classes of systems use some form of replication to deal with disk failures, MR is far more adept at handling node failures during the execution of an MR computation. In an MR system, if a unit of work (i.e. processing a block of data) fails, then the MR scheduler can automatically restart the task on an alternate node. Part of the flexibility is the result of the fact that the output files of the Map phase are materialized locally instead of being streamed to the nodes running the Reduce tasks. Similarly, pipelines of MR jobs such as the one described in Section 2.4.3.4, materialize intermediate results to files each step of the way. This differs from parallel DBMSs, which have larger granules of work (i.e., transactions) that are restarted in the event of a failure. Part of the reason for this approach is that DBMSs avoid saving intermediate results to disk whenever possible. Thus, if a single node fails during a long running query in a DBMS, the entire query must be completely restarted.

2.4 Performance Benchmarks

In this section we present our benchmark consisting of five tasks that we use to compare the performance of the MR model with that of parallel DBMSs. The first task is taken directly from the original MapReduce paper [40] that the authors claim is representative of common MR tasks. Because this task is quite simple, we also developed four additional tasks, comprised of more complex analytical workloads designed to explore the trade-offs discussed in the previous section. We executed our benchmarks on a well-known MR implementation and two parallel DBMSs.

2.4.1 Benchmark Environment

As we describe the details of our benchmark environment, we note how the different data analysis systems we test differ in operating assumptions and discuss the ways in which we dealt with them in order to make the experiments uniform.

2.4.1.1 Tested Systems

Our benchmarks were conducted on the following systems:

Hadoop: The Hadoop system is the most popular open-source implementation of the MapReduce framework, under development by Yahoo! and the Apache Software Foundation [9]. Unlike the Google implementation of the MR framework that is written in C++, the core Hadoop system is written entirely in Java. For our experiments, we used Hadoop version 0.19.0 running on Java 1.6.0. We used the default configuration settings for Hadoop, except for two parameters that we found yielded better performance without diverging from core MR fundamentals: (1) data is stored using 256MB data blocks instead of the default 64MB and (2) each task executor JVM ran with a maximum heap size of 512MB and the DataNode/JobTracker JVMs ran with a maximum heap size of 1024MB (for a total of 3.5GB per node), (3) we enabled Hadoop's "rack awareness" feature for data locality in the cluster, and (4) we allowed Hadoop to reuse the task JVM instead of starting a new process for each Map/Reduce task. Moreover, we configured the system to allow two Map instances and a single Reduce instance to execute concurrently on each node.

The Hadoop framework also provides an implementation of the Google distributed file system [55]. For each benchmark trial, we store all input and output data in the Hadoop distributed file system (HDFS). We used the default settings of HDFS of three replicas per block and without compression; we also tested other configurations, such as using only a single replica per block as well as block and record-level compression, but we found that our tests almost always executed at the same speed or worse with these features enabled (see Section 2.5.1.3). After each benchmark run finishes, we delete the data directories on each node and reformat HDFS so that the next set of input data is replicated uniformly across all nodes.

Hadoop uses a central job tracker and a “master” HDFS daemon to coordinate node activities. To ensure that these processes did not affect the performance of worker nodes, we execute both of these additional framework components on a separate node in the cluster that has the same hardware specifications as the other nodes.

DBMS-X: We used the latest release of DBMS-X, a parallel SQL DBMS from a major relational database vendor that stores data in a row-based format. The system is installed on each node and configured to use 4GB shared memory segments for the buffer pool and other temporary space. Each table is hash partitioned across all nodes on the salient attribute for that particular table, and then sorted and indexed on different attributes (see Section 2.4.2.1 and 2.4.3.1). Like the Hadoop experiments, we deleted the tables in DBMS-X and reloaded the data for each trial to ensure that the tuples were uniformly distributed in the cluster.

By default DBMS-X does not compress data in its internal storage, but it does provide ability to compress tables using a well-known dictionary-based scheme. We found that enabling compression reduced the execution times for almost all benchmark tasks by 50%, and thus we only report results with compression enabled. In only one case did we find that using compression actually performed worse. Furthermore, because all of our benchmarks are read-only, we did not enable replication features in DBMS-X, since that would not have improved performance and complicates the installation process.

Vertica: The Vertica database is a parallel DBMS designed for large data warehouses [14]. The main distinction of Vertica from other DBMSs (including DBMS-X) is that all data is stored as columns, rather than rows [97]. It uses a unique execution engine designed specifically for operating on top of a column-oriented storage layer. Unlike DBMS-X, Vertica compresses data by default since its executor can operate directly on compressed tables. Because disabling this feature is not typical in Vertica deployments, the Vertica results in this paper are generated only using compressed data. Vertica also sorts every table by one or more attributes based on a clustered index.

We found that the default 256MB buffer size per node performed well in our experiments. The Vertica resource manager is responsible for setting the amount of memory given to queries, but

we provide a hint to the system to expect to execute only one query at a time. Thus, each query receives most of the maximum amount of memory available on each node at runtime.

2.4.1.2 Node Configuration

All three systems were deployed on a 100-node cluster. Each node has a single 2.40 GHz Intel Core 2 Duo processor running 64-bit Red Hat Enterprise Linux 5 (kernel version 2.6.18) with 4GB RAM and two 250GB SATA-I hard disks. According to hdparam, the hard disks deliver approximately 7GB/sec for cached reads and approximately 74MB/sec for buffered reads. The nodes are connected with a Cisco Catalyst 3750E-48TD switch. This switch has gigabit Ethernet ports for each node and, according to the Cisco datasheet [34], has an internal switching fabric of 128Gbps. There are 50 nodes per switch. The switches are linked together via Cisco StackWise Plus, which creates a 64Gpbs ring between the switches. Traffic between two nodes on the same switch is entirely local to the switch and does not impact traffic on the ring.

2.4.1.3 Benchmark Execution

For each task in the benchmarks, we will outline both the steps needed to complete the task by an MR program and provide the equivalent SQL statement(s) executed by the two database systems. We executed each task three times and report the average of the three trials. Each system executes the benchmarks separately to ensure that it had exclusive access to the cluster resources. To measure the basic performance without the overhead of coordinating parallel tasks, we first execute each benchmark on a single node. We then execute the task on different cluster sizes to show how each system scales as both the amount of data processed and available resources are increased.

Our results assume that all nodes are available and operating correctly during the benchmark tests. Although each system supports node failure recovery, we always restarted each experiment if a node failed. Thus, are numbers are based on an ideal execution and network environment.

To ensure that no system used cached results from the previous trial, we flushed the disk cache of each node between trials and executed all of the benchmark tasks one after another in a specific trial (rather than executing the same task three times in a row).

We also measured the time it takes for each system to load the test data. The results from the measurements are split between the actual loading of the data and any additional operations after the loading that each system performs, such as compressing or building indexes. The initial input data on each node is stored on one of its two locally installed disks.

Unless otherwise indicated, the final results from the queries executing in Vertica or DBMS-X are piped from a shell command into a file on the disk not used by the DBMS. Although it is possible to do an equivalent operation in Hadoop, it is easier (and more common) to store the results of an MR program into the distributed file system. This procedure, however, is not analogous to how the DBMS produce their output data; rather than storing the results in a single file, the MR program produces one output file for each Reduce instance and stores them in a single directory. The standard practice is for developers then to use these output directories as a single input unit for other MR jobs. If, however, a user wishes to use this data in a non-MR application, they must first combine the results into a single file and download it to the local file system.

Because of this discrepancy, we execute an extra Reduce function for each MR benchmark task that simply combines the final output into a single file in HDFS. Our results differentiate between the execution times for Hadoop running the actual benchmark task versus the additional combine operation. Thus, the Hadoop results displayed in the graphs for this chapter are shown as stacked bars: the lower portion of each bar is the execution time for just the specific benchmark task, while the upper portion is the execution time for the single Reduce function to combine all of the program's output data into a single file.

2.4.2 The Original MR Benchmark

Our first benchmark is the "Grep task" taken from the original MapReduce paper, which the authors describe as "representative of a large subset of the real programs written by users of MapReduce" [40]. For this task, each system must scan through a data set of 100-byte records looking for

a three-character pattern. Each record consists of a unique key in the first 10 bytes, followed by a 90-byte random value. The search pattern is only found in the last 90 bytes once in every 10,000 records. The input data is stored on each node in plain text files, with one record per line. For the Hadoop trials, we uploaded these files unaltered directly into HDFS. To load the data into Vertica and DBMS-X, we execute each system's proprietary load commands in parallel on each node and store the data using the following schema:

```
CREATE TABLE Data (
   key   VARCHAR(10) NOT NULL PRIMARY KEY,
   field VARCHAR(90) NOT NULL
);
```

We executed the Grep task using two different amounts of input data. In the original MR paper, the authors processed 1TB of data on approximately 1800 nodes, which is 5.6 million records or roughly 535MB of data per node. Thus, in our first experiment, for each system we measured the processing time to find the search pattern on different cluster sizes of 1, 10, 25, 50, and 100 nodes, where each node has 535MB of data. The total number of records processed is therefore 5.6 million times the number of nodes. The performance of each system not only illustrates how each system scales as the amount of data increases, but also allows us to (to some extent) compare the results to the original MR system.

While our first dataset fixes the size of the data per node to be same as original MR benchmark and only varies the number of nodes, our second dataset fixes the total dataset size to be the same as the original MR benchmark (1TB) and evenly divides the data amongst a variable number of nodes. This task measures how well each system scales as the number of available nodes is increased. Since Hadoop needs a total of 3TB of disk space in order to store three replicas of each block in HDFS, we were limited to running this benchmark only on 25, 50, and 100 nodes (at fewer than 25 nodes, there is not enough available disk space to store 3TB).

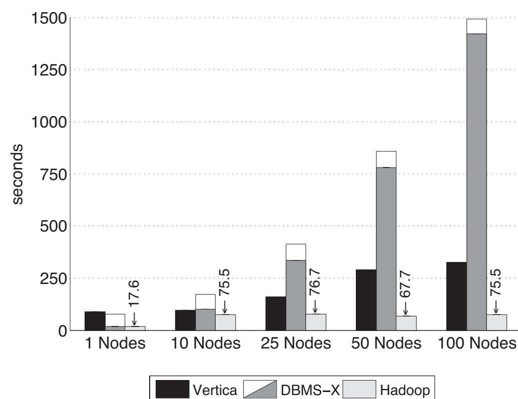

Figure 2.1: Load Times – Grep Task Data Set (535MB/node)

2.4.2.1 Data Loading

We now describe the procedures used to load the data from the nodes' local files into each system's internal storage representation.

Hadoop: There are two ways to load data into Hadoop's distributed file system: (1) use Hadoop's command-line file utility to upload file data stored on the local filesystem or (2) create a custom data loader program that writes data using Hadoop's internal I/O API. We did not need to alter the input data for our MR programs, therefore we loaded the files on each node in parallel directly into HDFS as plain text using the command-line utility. Storing the data in this manner enables MR programs to access data using Hadoop's `TextInputFormat` data format, where the keys are line numbers in each file and their corresponding values are the contents of each line. We found that this approach yielded the best performance in both the loading process and task execution, as opposed to using Hadoop's serialized data formats or compression features.

DBMS-X: The loading process in DBMS-X occurs in two phases. First, we execute the `LOAD` SQL command in parallel on each node in the cluster to read data from the local filesystem and insert its content into a particular table in the database. We specify in this command that the local data is delimited by a special character, therefore we did not need to write a custom program to transform the data before loading it. But because our data generator simply creates random keys for each record on each node, the DBMS-X must redistribute the tuples to other nodes in the cluster

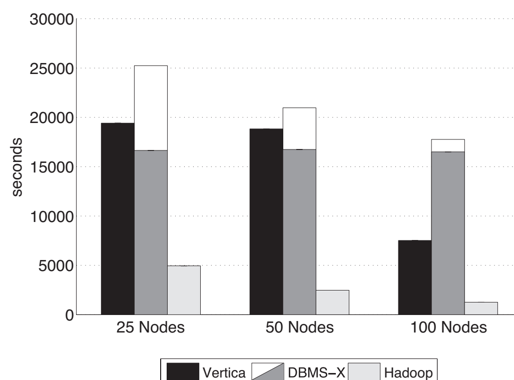

Figure 2.2: Load Times – Grep Task Data Set (1TB/cluster)

as it reads each record from the input files based on the target table's partitioning attribute. It would be possible to generate a "hash-aware" version of the data generator that would allow DBMS-X to just load the input files on each node without this redistribution process, but we do not believe that it would have improved load times very much.

Once the initial loading phase is complete, we then execute an administrative command to reorganize the data on each node. This process executes in parallel on each node to compress data, build each table's indexes, and perform other housekeeping.

Vertica: Vertical also provides a `COPY` SQL command that is issued from a single host and then coordinates the loading process on multiple nodes in parallel in the cluster. The user gives the `COPY` command as input a list of nodes to execute the loading operation for. This process is similar to DBMS-X: on each node the Vertica loader splits the input data files on a delimiter, creates a new tuple for each line in an input file, and redistributes that tuple to a different node based on the hash of its primary key. Once the data is loaded, the columns are automatically sorted and compressed according the physical design of the database.

Results and Discussion: The results for loading both the 535MB/node and 1TB/node datasets are shown in Figures 2.1 and 2.2, respectively. For DBMS-X, we separate the times of the two loading phases, which are shown as a stacked bar in the graphs: the bottom segments represent the execution time of the parallel `LOAD` commands and the top segment is the reorganization process.

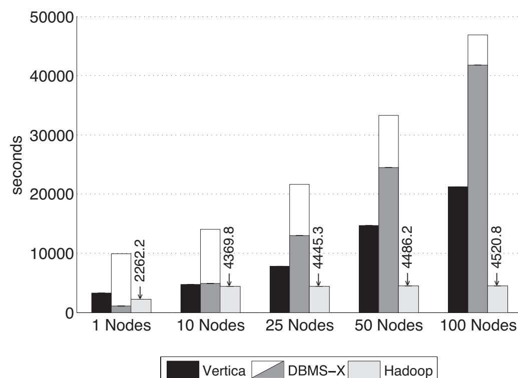

Figure 2.3: Load Times – UserVisits Data Set (20GB/node)

The most striking feature of the results for the load times in the 535MB/node dataset shown in Figure 2.1 is the difference in performance of DBMS-X compared to Hadoop and Vertica. Despite issuing the initial `LOAD` command in the first phase on each node in parallel, the data was actually loaded on each node sequentially. Thus, as the total amount of data is increased, the load times also increased proportionally. This also explained why, for the 1TB/cluster data set, the load times for DBMS-X do not decrease as less data is stored per node. However, the compression and housekeeping on DBMS-X can be done in parallel across nodes, and thus the execution time of the second phase of the loading process is cut in half when twice as many nodes are used to store the 1TB of data.

Without using either block- or record-level compression, Hadoop clearly outperforms both DBMS-X and Vertica since each node is simply copying each data file from the local disk into the local HDFS instance and then distributing two replicas to other nodes in the cluster. If we load the data into Hadoop using only a single replica per block, then the load times are reduced by a factor of three. But as we will discuss in Section 2.5, the lack of multiple replicas often increases the execution times of jobs.

2.4.2.2 Task Execution

SQL Commands: A pattern search for a particular field is simply the following query in SQL. Neither SQL system contained an index on the field attribute, so this query requires a full scan.

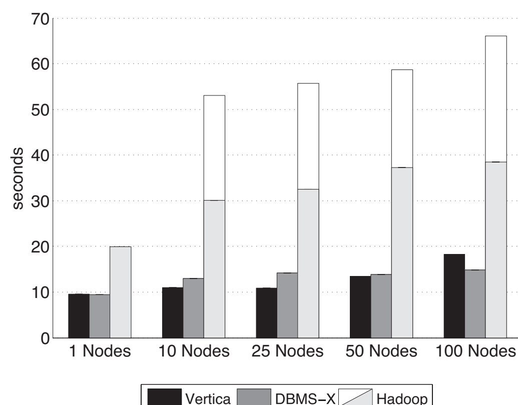

Figure 2.4: Grep Task Results – 535MB/node Data Set

```
SELECT field WHERE field LIKE '%XYZ%';
```

MapReduce Program: The MR program consists of a single Map function that is given each record already split into the appropriate key/value pair and then performs a sub-string match on the value. If the search pattern is found, the Map function outputs the input key/value pair to HDFS. Because no Reduce function is defined, the output of the Map task is the final output of the program.

Results & Discussion: The performance results for the three systems are shown in Figures 2.4 and 2.5. Surprisingly, the relative differences between the systems are not consistent in the two figures. In Figure 2.4 the two parallel databases perform about the same, more than a factor of two faster than Hadoop. But in Figure 2.5, both DBMS-X and Hadoop perform more than a factor of two slower than Vertica. The reason is that the amount of data processing varies substantially from the two experiments. For the results in Figure 2.4, very little data is being processed (535MB/node). This causes Hadoop's non-insignificant start-up costs to become the limiting factor in its performance. As will be described in Section 2.5.1.2, for short-running queries (i.e., queries that take less than a minute), Hadoop's start-up costs can dominate the execution time. In our observations, we found that takes 10-25 seconds before all map tasks have been started and are running at full speed across the nodes in the cluster. Furthermore, as the total number of allocated Map tasks increase, there is additional overhead required for the central job tracker to coordinate node activities. Hence, this overhead increases slightly as more nodes are added to the cluster and

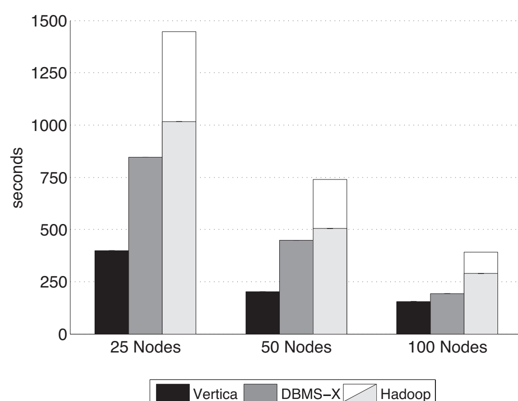

Figure 2.5: Grep Task Results – 1TB/cluster Data Set

for longer data processing tasks, as shown in Figure 2.5, this fixed cost is dwarfed by the time to complete the required processing.

The upper segments of each Hadoop bar in the graphs represent the execution time of the additional MR job to combine the output into a single file. Since we ran this as a separate MapReduce job, these segments consume a larger percentage of overall time in Figure 2.4, as the fixed start-up overhead cost again dominates the work needed to perform the rest of the task. Even though the Grep task is selective, the results in Figure 2.5 show how this combine phase can still take hundreds of seconds due to the need to open and combine many small output files. Each Map instance produces its output in a separate HDFS file, and thus even though each file is small there are many Map tasks and therefore many files on each node.

For the 1TB/cluster data set experiments, Figure 2.5 shows that all systems executed the task in nearly half the amount of time when using twice as many nodes, as one would expect since the total amount of data was held constant across nodes for this experiment. Hadoop and DBMS-X performs approximately the same, since Hadoop's startup cost is amortized across the increased amount of data processing for this experiment. However, the results clearly show that Vertica outperforms both DBMS-X and Hadoop. We attribute this to Vertica's aggressive use of data compression, (see section 2.5.1.3) which becomes more effective as more data is stored per node.

2.4.3 Analytical Tasks

To explore more complex uses of both types of systems, we developed four tasks related to HTML document processing. We first generate a collection of random HTML documents, similar to that which a web crawler might find. Each node is assigned a set of 60,000 unique HTML documents, each with a unique URL. In each document, we randomly generate links to other pages set using a Zipfian distribution.

We also generated two additional data sets meant to model log files of HTTP server traffic. These data sets consist of values derived from the HTML documents, as well as several randomly generated attributes. The schema for these three tables is as follows:

```
CREATE TABLE UserVisits (
  sourceIP VARCHAR(16) NOT NULL PRIMARY KEY,
  destinationURL VARCHAR(100),
  visitDate TIMESTAMP,
  adRevenue FLOAT,
  userAgent VARCHAR(64),
  countryCode VARCHAR(3),
  languageCode VARCHAR(6),
  searchKeyword VARCHAR(32),
  duration INT
);
CREATE TABLE Rankings (
  pageURL VARCHAR(100) NOT NULL PRIMARY KEY,
  pageRank INT,
  avgDuration INT
);
CREATE TABLE Documents (
  url VARCHAR(100) NOT NULL PRIMARY KEY,
  contents TEXT
```

```
);
```

Our data generator created unique files with 155 million UserVisit records (20GB/node) and 18 million Rankings records (1GB/node) on each node. The visitData, adRevenue, and sourceIP fields are picked uniformly at random from specific ranges. All other fields are picked uniformly from sampling real-world data sets. Each data file is stored on each node as a column-delimited text file.

2.4.3.1 Data Loading

We now describe the procedures for loading the UserVisits and Rankings data sets. For reasons to be discussed in Section 2.4.3.5, only Hadoop needs to directly load the Documents files into its storage system. DBMS-X and Vertica both execute a UDF that process the Documents on each node at runtime and loads the data into a temporary table. We account for the overhead of this approach in the benchmark time, rather than in load times. Therefore, we do not provide results for loading the Documents data set.

Hadoop: Unlike the Grep task's data set, which was uploaded directly into HDFS, the UserVisits and Rankings data sets need to be modified so that the first and second columns are separated by a tab delimiter and all other fields in each line are separated by a unique delimiter. Because there are no schemas in the MR model, in order to access the different attributes at run time, the Map and Reduce functions in each task must manually split the value by the delimiter character into an array of strings.

We wrote a custom data loader executed in parallel on each node to read in each line of the data sets, prepare the data as needed, and then write the tuple into a plain text file in HDFS. Loading the data sets in this manner was roughly three times slower than using the command-line utility, but did not require us to write custom input handlers in Hadoop, the MR programs are able to use Hadoop's `KeyValueTextInputFormat` interface on the data files to automatically split lines of text files into key/value pairs by the tab delimiters. Again, we found that other data format options, such as `SequenceFileInputFormat` or custom `Writable` tuples, resulted in both slower load and execution times.

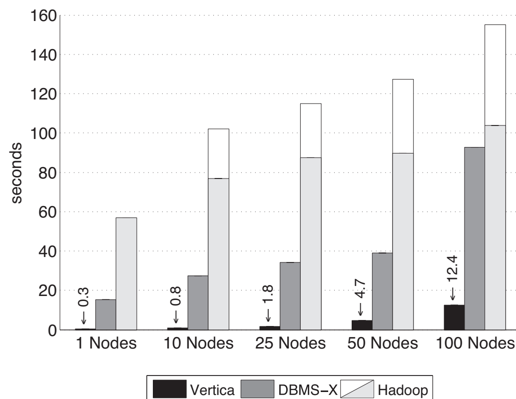

Figure 2.6: Selection Task Results

DBMS-X: We used the same loading procedures for DBMS-X as discussed in Section 2.4.2.1. The Rankings table was hash partitioned across the cluster on pageURL, and the data on each node was sorted by pageRank. Likewise, the UserVisits table was hash partitioned on destinationURL and sorted by visitDate on each node.

Vertica: Similar to DBMS-X, Vertica used the same bulk load commands discussed in Section 2.4.2.1 and sorted the UserVisits and Rankings tables by the visitDate and pageRank columns, respectively.

Results and Discussion: Since the results of loading the UserVisits and Ranking data sets are similar, we only provide the results for the loading of the larger UserVisits data in Figure 2.3. Just as with loading the Grep 535MB/node data set (2.1), the loading times for each system increased in proportion to the number of nodes used.

2.4.3.2 Selection Task

The Selection task is a lightweight filter to find the pageURLs in the Rankings table (1GB/node) with a pageRank above a user-defined threshold. For our experiments, we set this threshold parameter to 10, which yields approximately 36,000 records per data file on each node.

SQL Commands: The DBMSs execute the selection task using the following simple SQL statement:

```
SELECT pageURL, pageRank FROM Rankings WHERE pageRank > X;
```

MapReduce Program: The MR program uses only a single Map function that splits the input value based on the field delimiter and outputs the record's pageURL and pageRank as a new key/value pair if its pageRank is above the threshold. This task does not require a Reduce function, since each pageURL in the Rankings data is unique across all nodes.

Results and Discussion: As was discussed in the Grep task, the results from this experiment, shown in Figure 2.6, demonstrate again that the parallel DBMSs outperform Hadoop by a rather significant factor across all cluster scaling levels. Although the relative performance of all systems degrade as both the number of nodes and the total amount of data increase, Hadoop is the most affected. For example, there is almost a 50% difference in the execution time between 1 node and 10 node experiments. This is again due to Hadoop's increased start-up costs as more nodes are added to the cluster, which takes up a proportionately larger fraction of the total query time for short-running queries.

Another important reason for why the parallel DBMSs are able to outperform Hadoop is that both Vertica and DBMS-X use an index on the pageRank column and store the Rankings table already sorted by pageRank. Thus, executing this query is trivial. It should also be noted that although Vertica's absolute times remain low, its relative performance degrades as the number of nodes increases. This is in spite of the fact that each node still executes the query in the same amount of time (approximately 170ms). But because the nodes finish executing the query so quickly, the system becomes flooded with control messages from too many nodes, which then takes a longer time for the system to process. Vertica uses a reliable message layer for query dissemination and commit protocol processing [19], which we believe has considerable overhead when more than a few dozen nodes are involved in the query.

2.4.3.3 Aggregation Task

Our next task requires each system to calculate the total adRevenue generated for each sourceIP in the UserVisits table (20GB/node), grouped by the sourceIP column. We also ran a variant of this query where we grouped by the seven-character prefix of the sourceIP column to measure the

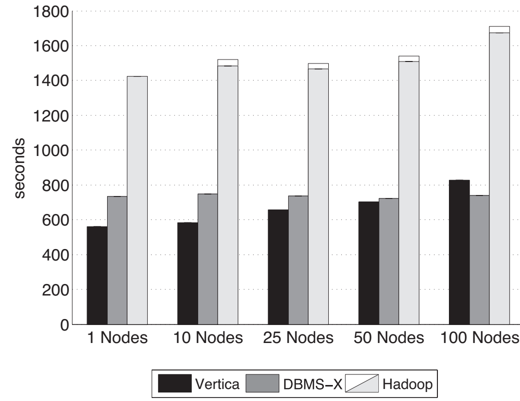

Figure 2.7: Aggregation Task Results (2.5 million Groups)

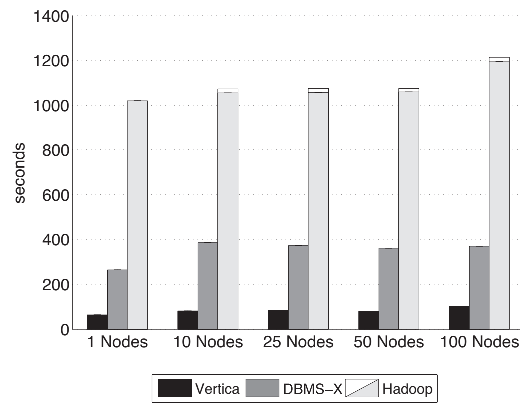

Figure 2.8: Aggregation Task Results (2000 groups)

effect of reducing the total number of groups on query performance. We designed this task to measure the performance of parallel analytics on a single read-only table, where nodes needs to exchange intermediate data with one another in order to compute the final value. Regardless of the number of nodes in the cluster, this task always produces 2.5 million records (53 MB); the variant query produces 2,000 records (24KB).

SQL Commands: The SQL command to calculate the total adRevenue is straightforward:

```
SELECT sourceIP, SUM(adRevenue)
  FROM UserVisits
 GROUP BY sourceIP;
```

The variant query is:

```
SELECT SUBSTR(sourceIP, 1, 7), SUM(adRevenue)
  FROM UserVisits
 GROUP BY SUBSTR(sourceIP, 1, 7);
```

MapReduce Program: Unlike the previous tasks, the MR program for this task consists of both a Map and Reduce function. The Map function first splits the input value by the field delimiter, and then outputs the sourceIP field (given as the input key) and the adRevenue field as a new key/value pair. For the variant query, only the first seven characters (representing the first two octets, stored as three digits) of the sourceIp are used. These two Map functions share the same Reduce function that simply adds together all of the adRevenue values for each sourceIP and then outputs the prefix and revenue total. We also used MR's *Combine* feature to perform the pre-aggregate before data is transmitted to the Reduce instances, improving the first query's execution time by a factor of two [40].

Results & Discussion: The results of the aggregation task experiment in Figures 2.7 and 2.8 show once again that the two DBMSs outperform Hadoop. The DBMSs execute these queries by having each node scan its local table, extract the sourceIP and adRevenue fields, and perform a local *Group-By*. These local groups are then merged at the query coordinator, which outputs results to the user. The results in Figure 2.7 illustrate that the two DBMSs perform about the same

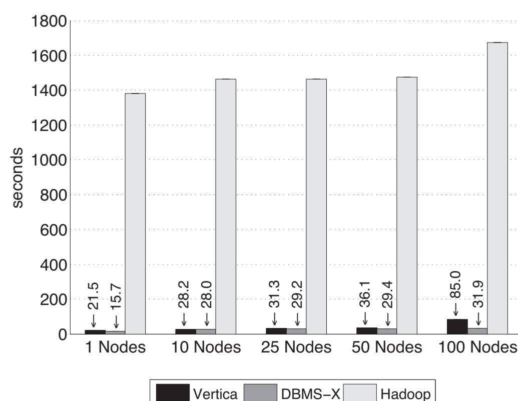

Figure 2.9: Join Task Results

for a large number of groups, as their runtime is dominated by the cost to transmit the large number of local groups and merge them at the coordinator. For the experiments using fewer nodes, Vertica performs somewhat better, since it has to read less data (since it can directly access the sourceIP and adRevenue columns), but it becomes slightly slower as more nodes are used.

Based on the results in Figure 2.8, it is more advantageous to use a column-store system when processing fewer groups for this tasks. This is because the two columns accessed (sourceIP and adRevenue) consist of only 20 bytes out of more than 200 bytes per UserVisits tuple, and therefore there are relatively few groups that need to be merged so communication costs are much lower than in the non-variant plan. Vertica is thus able to outperform the other two systems from not reading unused parts of the UserVisit tuples.

Note that the execution times for all systems are roughly consistent for any number of nodes (modulo Vertica's slight slow down as the number of nodes increases). Since this benchmark task requires the system to scan through the entire data set, the run time is always bounded by the constant sequential scan performance and network repartitioning costs for each node.

2.4.3.4 Join Task

The join task consists of two sub-tasks that perform a complex calculation on two data sets. In the first part of the task, each system must find the sourceIP that generated the most revenue within a particular date range. Once these intermediate records are generated, the system must

then calculate the average pageRank of all the pages visited during this interval. We use the week of January 15-22, 2000 in our experiments, which matches approximately 134,000 records in the UserVisits table.

The salient aspect of this task is that it must consume two different data sets and join them together in order to find pairs of Ranking and UserVisits records with matching values for pageURL and destURL. This task stresses each system using fairly complex operations over a large amount of data. The performance results are also a good indication of how well the DBMS's query optimizer produces efficient join plans.

SQL Commands: In contrast to complexity of the MR program described below, the DBMSs need only two fairly simple queries to complete the task. The first statement creates a temporary table and uses it to store the output of the `SELECT` statement that performs the join of UserVisits and Rankings and computes the aggregates. Once this table is populated, it is then trivial to use a second query to output the record with the largest totalRevenue field.

```
SELECT INTO Temp sourceIP,
                 AVG(pageRank) as avgPageRank,
                 SUM(adRevenue) as totalRevenue
  FROM Rankings AS R, UserVisits AS UV
 WHERE R.pageURL = UV.destinationURL
   AND UV.visitDate BETWEEN Date('2000-01-15')
                        AND Date('2000-1-22')
 GROUP BY UV.sourceIP;


SELECT sourceIP, totalRevenue, avgPageRank
  FROM Temp
 ORDER BY totalRevenue LIMIT 1;
```

MapReduce Program: Because the MR model does not have an inherent ability to join two or more disparate data sets, the MR program that implements the join task must be broken out into

three separate phases. Each of these phases is implemented together as a single MR program in Hadoop, but does not begin executing until the previous phase is complete.

Phase 1 – The first phase filters UserVisits records that are outside the desired data range and then joins the qualifying records with records from the Rankings file. The MR program is initially given all of the UserVisits and Rankings data files as inputs.

Map Function: For each key/value input pair, we determine its record type by counting the number of fields provided when splitting the value on the delimiter. If it is a UserVisits record, we apply the filter based on the date range predicate. These qualifying records are emitted with composite keys of the form (destURL, K_1), where K_1 indicates that it is a UserVisits records. All Rankings records are emitted with composite keys of the form (pageURL, K_2), where K_2 indicates that it is a UserVisits record. These output records are repartitioned using a user-supplied partitioning function that only hashes on the URL portion of the composite key.

Reduce Function: The input to the Reduce function is a single sorted run of records in URL order. For each URL, we divide its values into two sets based on the tag component of the composite key. The function then forms the cross product of the two sets to complete the join and outputs a new key/value pair with the sourceIP as the key and the tuple (pageURL, pageRank, adRevenue) as the value.

Phase 2 – The next phase computes the total adRevenue and average pageRank based on the sourceIP of records generated in Phase 1. This phase uses a Reduce function in order to gather all of the records for a particular sourceIP on a single node. We use the identity Map function in the Hadoop API to supply records directly to the split process [9, 40].

Reduce Function: For each sourceIP, the function adds up the adRevenue and computes the average pageRank, retaining the one with maximum total ad revenue. Each Reduce instance outputs a single record with sourceIP as the key and the value as a tuple of the form (avgPageRank, totalRevenue).

Phase 3 – In the final phase, we again only need to define a single Reduce function that uses the output from the previous phase to produce the record with the largest total adRevenue. We only

execute one instance of the Reduce function on a single node to scan all the records from Phase 2 and find the target record.

Reduce Function: The function processes each key/value pair and keeps track of the record with the largest totalRevenue field. Because the Hadoop API does not easily expose the total number of records that a Reduce function will process, there is no way for the Reduce function to know that it is processing the last record. Therefore, we override the closing callback method in our Reduce implementation so that the MR program outputs the largest record right before it exits.

Results & Discussion: The performance results for this task is displayed in Figure 2.9. We had to slightly change the SQL used in 100 node experiments for Vertica due to an optimizer bug in the system, which is why there is an increase in the execution time for Vertica going from 50 to 100 nodes. But even with this increase, it is clear that this task results in the biggest performance difference between Hadoop and the parallel database systems. The reason for this disparity is two-fold.

First, despite the increased complexity in the query, the performance of Hadoop is yet again limited by the speed with which the large UserVisits table (20GB/node) can be read off disk. The MR program has to perform a complete table scan, while the parallel database systems were able to take advantage of clustered indexes on UserVisits.visitDate to significantly reduce the amount of data that needed to be read. When breaking down the costs of the different parts of the Hadoop query, we found that regardless of the number of nodes in the cluster, phase 2 and phase 3 took on average 24.3 seconds and 12.7 seconds, respectively. In contrast, phase 1, which contains the Map task that reads in the UserVisits and Rankings tables, takes an average of 1434.7 seconds to complete. Interestingly, it takes approximately 600 seconds of raw I/O to read the UserVisits and Rankings tables off of disk and then another 300 seconds to split, parse, and deserialize the various attributes. Thus, the CPU overhead needed to parse these tables on the fly is the limiting factor for Hadoop.

Second, the parallel DBMSs are able to take advantage of the fact that both the UserVisits and the Rankings tables are partitioned by the join keys. This means that both systems are able to do the join locally on each node, without any network overhead of repartitioning before the join.

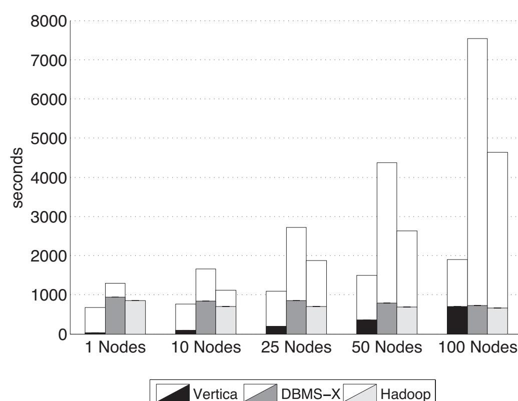

Figure 2.10: UDF Aggregation Task Results

Thus, they simply have to do a local hash join between the Rankings table and a selective part of the UserVisits table on each node, with a trivial `ORDER BY` clause across nodes.

2.4.3.5 UDF Aggregation Task

The final task is to compute the inlink count for each document in the dataset, a task that is often used as a component of PageRank calculations. Specifically, for this task, the system must read each document file and search for all the URLs that appear in the contents. The system must then, for each unique URL, count the number of unique pages that reference that particular URL across the entire set of files. It is the type of task that MR is believed to be commonly used for.

We make two adjustments for this task in order to make processing easier in Hadoop. First, we allow the aggregate to include self-references, as it is non-trivial for a Map function to discover the name of the input file it is processing. Second, on each node we concatenate the HTML documents into larger files when storing them in HDFS. We found this improved Hadoop's performance by a factor of two and helped avoid memory issues with the central HDFS master when a large number of files are stored in the system.

SQL Commands: To perform this task in a parallel DBMS requires a user-defined function F that parses the contents of each record in the Documents table and emits URLs into the database. This function can be written in a general-purpose language and is effectively identical to the Map

program discussed below. With this function F, we populate a temporary table with a list of URLs and then can execute a simple query to calculate the inlink count:

```
SELECT INTO Temp F(contents) FROM Documents;
SELECT url, SUM(pageRank) FROM Temp GROUP BY url;
```

Despite the simplicity of this proposed UDF, we found that in practice it was difficult to implement in the DBMSs.

For DBMS-X, we translated the MR program used in Hadoop into an equivalent C program that used the POSIX regular expression library to search for hyperlinks in the document. For each URL found in the document contents, the UDF returns a new tuple (URL,1) to the database engine. We originally intended to store each HTML document as a character BLOB in DBMS-X and then execute the UDF on each document completely inside the database, but were prevented from doing so by a known bug in the release of DBMS-X we were using. Instead, we modified the UDF to open a file on the local hard drives containing the HTML data and return the URLs the UDF discovers. This is similar to the approach taken with Vertica (see below). In contrast to Vertica, the UDF did not run external to the database, and did not use any bulk-loading tools to import the extracted URLs.

Vertica does not currently support UDFs, therefore we had to implement this benchmark task in two phases. In the first phase, we used a modified version of the DBMS-X UDF to extract URLs from the files, but then write the output to files on each node's local filesystem. Unlike DBMS-X, this program executes as a separate process outside the database system. Each node then loads the contents of these files into a table using Vertica's bulk-loading tools. Once this is completed, we then execute the query as described above to compute the inlink count for each URL.

MapReduce Program: To fit into the MR model where all data must be defined in terms of key/value pairs, each HTML document is split by lines and given to the Map function with the line contents as the value and the line number in which it appeared in the file as its key. The Map function then uses a regular expression to find all of the URLs in each line. For every URL found, the function outputs the URL and the integer 1 as a new key/value pair. Given these records, the

Reduce function then simply counts the number of values for a given key and outputs the URL and the calculated inlink count as the program's final output.

Results & Discussion: The results in Figure 2.10 show that both DBMS-X and Hadoop (not including the extra Reduce process to combine the data) have approximately constant performance this task, since each node has the same amount of Document data to process and this amount of data remains constant (7GB) as more nodes as added in the experiments. As we expected, the additional operation for Hadoop to combine data into a single file in HDFS gets progressively slower since the amount of output data that the single node must process gets larger as new nodes are added. The results for both DBMS-X and Vertica are shown in Figure 2.10 as stacked bars, where the bottom segment represents the time it took to execute the UDF/parser and load the data into the table and the top segment is the time to execute the actual query. DBMS-X performs worse than Hadoop due to the added overhead of row-by-row interaction between the UDF and the input file stored outside of the database. Vertica's poor performance is the result of having to parse data outside of the DBMS and materialize the intermediate results on the local disk before it can load it into the system.

2.5 Discussion

We now discuss broader issues about the benchmark results and comment on particular aspects of each system that the raw numbers may not convey. In the benchmark above, both DBMS-X and Vertica execute most of the tasks much faster than Hadoop at all scaling levels. The next subsections describe, in greater detail than the previous section, the reasons for this dramatic performance difference.

2.5.1 System Aspects

In this section, we describe how architectural decisions made at the systems level affect the relative performance of the two classes of data analysis systems. Since installation and configuration parameters can have a significant difference in the ultimate performance of the system, we begin with a discussion of the relative ease with which these parameters are set. Afterwards, we discuss

some lower level implementation details. While some of these details affect performance in fundamental ways (e.g., the lack of a loading phase in MR precludes various I/O optimizations and necessitates runtime parsing which increases CPU costs), others are more implementation specific (e.g., the high start-up cost of MR).

2.5.1.1 System Installation, Configuration, and Tuning

We were able to get Hadoop installed and running jobs with little effort. Installing the system only requires setting up data directories on each node and deploying the system library and configuration files. Configuring the system for optimal performance was done through trial and error: We found that certain parameters, such as the size of the sort buffers had no affect on execution performance, whereas other parameters, such as using larger block sizes, improved performance significantly.

The DBMS-X installation process was relatively straightforward. A GUI leads the user through the initial steps on one of the cluster nodes, and then prepares a file that can be fed to the installer in parallel on the other nodes that completes the installation. Despite this simple process, we found that DBMS-X was complicated to configure in order to start running queries. Initially, we were frustrated by the the failure of anything but the most basic of operations. We eventually discovered that each node's kernel was configured to limit the total amount of allocated virtual address space. When this limit was hit, new processes could not be created and DBMS-X operations would fail. We mention this even though it was our own administrative error, as we were surprised that DBMS-X's extensive system probing and self-adjusting configuration was not able to detect this limitation. This was disappointing after our earlier Hadoop successes.

Even after these earlier issues were resolved and we had DBMS-X running, we were routinely stymied by other memory limitations. We found that certain default parameters, such as the sizes of the buffer pool and sort heaps, were too conservative for modern systems. Furthermore, DBMS-X proved to be ineffective at adjusting memory allocations for changing conditions. For example, the system automatically expanded our buffer pool from the default 4MB to only 5MB (we later forced it to 512MB). It also warned us that performance could be degraded when we increased our

sort heap size to 128MB (in fact, performance improved by a factor of 12). Manually changing some options resulted in the system automatically altering others. On occasion, this combination of manual and automatic changes resulted in a configuration for DBMS-X that refused to boot the next time the system started. As most configuration settings required DBMS-X to be running in order to adjust them, it was unfortunately easy to lock ourselves out with no fail-safe mode to restore to a previous state.

Vertica was relatively easy to install as an RPM that we deployed on each node. An additional configuration script bundled with the RPM is used to build catalog meta-data and modify certain kernel parameters. Database tuning is minimal and is done through hints to the resource manager; we found that the default settings worked well for us. The downside of this simplified tuning approach, however, is that there is no explicit mechanism to determine what resources were granted to a query nor is there a way to manually adjust per-query resource allocation.

The take-away from our efforts is that we found the database systems to be much more challenging than Hadoop to install and configure. There is, however, a significant variation with respect to ease of installation and configuration across the parallel database products. One small advantage for the database systems is that the tuning that needed to be done for the parallel database systems was largely be performed in advance, and that certain tuning parameters (e.g., sort buffer sizes) are suitable for all tasks. In contrast, for Hadoop, we not only had to tune the system (e.g., block sizes), but we also occasionally needed to tune each individual task to work well with the system (e.g., changing code). Finally, the parallel database products came with tools to aid in the tuning process whereas with Hadoop we were forced to resort to trial and error tuning; clearly a more mature MapReduce implementation could include such tuning tools as well.

2.5.1.2 Task Startup

We found that our MR programs took some time before all nodes were running at full capacity. On a cluster of 100 nodes, it takes 10 seconds from the moment that a job is submitted to the JobTracker before the first Map task beings to execute, and 25 seconds until all the nodes in the cluster are executing the job. This coincides with the results in [40], where the data processing rate

does reach its peak for nearly 60 seconds on a cluster of 1800 nodes. The "cold start" nature is symptomatic to Hadoop's (and apparently Google's) implementation and not inherent to the actual MR model itself. For example, we also found that prior versions of Hadoop would create a new JVM process for each Map and Reduce instance on a node, which we found increased the overhead of running jobs on large data sets; enabling the JVM reuse feature in the latest version of Hadoop improved our results for MR by 10-15%.

In contrast, parallel DBMSs are started at boot time, and thus are considered to be always "warm", waiting for a task to perform. Moreover, all modern DBMSs are designed to execute using multiple threads and processes, which allows the currently running code can accept additional tasks and optimize its execution schedule. Minimizing startup time was one of the early optimizations of DBMSs, and is certainly something that MR systems should be able to incorporate without a large rewrite of the underlying architecture.

2.5.1.3 Compression

Almost every parallel DBMS (including DBMS-X and Vertica) allows for optional compression of stored data. It is not uncommon for compression to result in a factor of 6-10 space savings. Vertica's internal data representation, in fact, is highly optimized for data compression and has an execution engine that is capable of operating directly on compressed data (i.e., it avoids decompressing the data during processing whenever possible). In general, since analysis tasks on large data sets are often I/O bound, trading CPU cycles (needed to decompress input data) for I/O bandwidth (compressed data means that there is less data to read) is a good trade-off and translates to faster execution. In situations where the executor can operate directly on compressed data, there is no trade-off at all and compression is an obvious win.

Hadoop and its underlying distributed filesystem support both block-level and record-level compression on input data. We found, however, that neither technique improved Hadoop's performance and in some cases actually slowed execution. It also required more effort on our part to either change code or prepare the input data. It should also be noted that compression was also not used in the original MR benchmark [40].

In order to use block-level compression in Hadoop, we first had to split the data files into multiple, smaller files on each node's local file system and then compress each file using the *gzip* tool. Compressing the data in this manner reduced each data set by 20-25% from its original size. These compressed files are then copied into HDFS just as if they were plain text files. Hadoop automatically detects when files are compressed and will decompress them on the fly when they are fed into Map instances, thus we did not need to change our MR programs to use the compressed data. Despite the longer load times (if one includes the splitting and compressing), Hadoop using block-level compression slowed most of the tasks by a few seconds while CPU-bound tasks executed 50% slower.

We also tried executing the benchmarks using record-level compression. This required us to (1) write to a custom tuple object using Hadoop's API, (2) modify our data loader program to transform records to compressed and serialized custom tuples, and (3) refactor each benchmark. We initially believed that this would improve CPU-bound tasks, because Map and Reduce tasks no longer needed to split the fields by the delimiter. We found, however, that this approach actually performed worse than the block-level compression while only compressing the data by 10%.

2.5.1.4 Loading and Data Layout

Parallel databases have the opportunity to reorganize the input data file at load time. This allows for certain optimizations, such as storing each attribute of a table separately (as done in column-stores such as Vertica). For read-only queries that only touch a subset of the attributes of a table, this optimization can significantly improve performance by allowing the attributes that are not accessed by a particular query to be left on disk and never read. Similar to the compression optimization above, this saves critical I/O bandwidth. MR systems by default do not transform the data when it is loaded into their distributed file system, and thus are unable to change the layout of input data, which precludes this class of optimization opportunities. Furthermore, Hadoop was always much more CPU intensive than the parallel DBMS in running equivalent tasks because it must parse and deserialize the records in the input data at run time, whereas parallel databases do the parsing at load time and can quickly extract attributes from tuples at essentially zero cost.

But MR's simplified loading process did make it much easier and faster to load than with the DBMSs. Our results in Sections 2.4.2.1 and 2.4.3.1 show that Hadoop achieved load throughput of up to three times faster than Vertica and almost 20 times faster than DBMS-X. This suggests that for data that is only going to be loaded once for certain types of analysis tasks, it may not be be worth it to pay the cost of the indexing and reorganization costs in a DBMS. This also strongly suggests that a DBMS would benefit from an "in-situ" operation mode that would allow a user to directly access and query files stored in a local file system.

2.5.1.5 Execution Strategies

As noted earlier, the query planner in parallel DBMSs are careful to transfer data between nodes only if it is absolutely necessary. This allows the systems to optimize the join algorithm depending on the characteristics of the data and perform push-oriented messaging without writing intermediate data sets. Over time, MR advocates should study the techniques used in parallel DBMSs and incorporate the concepts that are germane to their model. In doing so, we believe that again the performance of MR frameworks will improve dramatically.

Furthermore, parallel DBMSs construct a complete query plan, which is sent to all processing nodes at the start of the query. Because data is "pushed" between sites when only necessary, there are no control messages during processing. In contrast, MR systems use a large number of control messages to synchronize processing, resulting in poorer performance due to increased overhead; Vertica also experienced this problem but on a much smaller scale.

2.5.1.6 Failure Model

As discussed earlier, while not providing support for transactions, MR is able to recover from faults in the middle of query execution in a way that most parallel database systems cannot. Since parallel DBMSs will be deployed on larger clusters over time, the probability of mid-query hardware failures will increase. Thus, for long-running queries, it may be important to implement such a fault tolerance model. While improving the fault tolerance of DBMSs is clearly a good idea, we are wary of devoting huge computational clusters to "brute force" approaches to computation

when sophisticated software could do the same processing with far less hardware and consume far less energy, or in less time, thereby obviating the need for a sophisticated fault tolerance model. A multi-thousand-node cluster of the sort Google, Microsoft, and Yahoo! run uses huge amounts of energy, and as our results show, for many data processing tasks a parallel DBMS can often achieve the same performance using far fewer nodes. As such, the desirable approach is to use high-performance algorithms with modest parallelism rather than brute force approaches on much larger clusters.

2.5.2 User-level Discussion

A data processing system's performance is irrelevant to a user or an organization if the system is not usable. In this section, we discuss aspects of each system that we encountered while conducting the benchmark study that may promote or inhibit application development and adoption.

2.5.2.1 Ease of Use

Once the system is on-line and the data has been loaded, the programmer then begins to write the query or the code needed to perform their task. Like other kinds of programming, this is often an iterative process: the programmer writes a little bit of code, tests it, and then writes some more. The programmer can easily determine whether his/her code is syntactically correct in both types of systems: the MR framework can check whether the user's code compiles and the SQL engines can determine whether the queries parse correctly. Both systems also provide runtime support to assist users in debugging their programs.

It is also worth considering the way in which the programmer writes the query. MR programs in Hadoop are primarily written in Java (though other language bindings exist). Most programmers are more familiar with object-oriented, imperative programming than with other language technologies, such as SQL. That said, SQL is taught in many undergraduate programs and is fairly portable—we were able to share the SQL commands between DBMS-X and Vertica with only minor modifications.

In general, we found that getting an MR application up and running with Hadoop for the benchmark tasks took significantly less effort than with the other systems. We did not need to construct a schema or register user-defined functions in order to begin processing the data. However, after obtaining our initial results we expanded the number of benchmark tasks, causing us to add new columns to our data set. In order to process this new data, we had to modify our existing MR code and retest each MR application to ensure that it worked with the new assumptions about the data's schema. Furthermore, some API methods in Hadoop were deprecated after we upgraded to newer versions, which required us to rewrite portions of our applications. In contrast, once we had built our initial SQL applications, we did not have to modify the code despite several changes to our benchmark schema.

We argue that although it may be easier to for developers to "get up and running" with MR, maintenance of MR applications—particularly in the face of schema changes—is likely to lead to significant pain for applications developers over time. Furthermore, as we argued in Section 2.3.1, reusing MR code between two deployments or on two different data sets is difficult, as there is no explicit representation of the schema for data used in the MR model.

2.5.2.2 Additional Tools

At the time that this benchmark was developed, Hadoop came with a rudimentary web interface that allows you to browse HDFS and monitor the execution of your jobs. Any additional tools would have had to been developed in-house.

SQL Databases, on the other hand, have tons of existing tools and applications for reporting and data analysis. Entire software industries have developed around providing DBMS users with third-party extensions. The types of software that many of these tools include (1) data visualization, (2) business intelligence, (3) data mining, (4) data replication, and (5) automatic database design. Because MR technologies are still nascent, the market for such software for MR is limited; however, as the user base grows, many existing SQL-based tools will likely support MR systems.

2.6 Conclusions

There are a number of interesting conclusions that can be drawn from the results presented in this chapter. First, at the scale of the experiments we conducted, both parallel database systems displayed a significant performance advantage over Hadoop MR in executing a variety of data intensive analysis tasks. Averaged across all five tasks at 100 nodes, DBMS-X was 3.2 times faster than MR and Vertica was 2.3 times faster than DBMS-X. While we cannot verify this claim, we believe that the system would have the same relative performance on 1,000 nodes (the largest Teradata configuration is less than 100 nodes managing over four petabytes of data). The dual of these numbers is that a parallel database system that provides the same response time with far fewer processors will certainly use far less energy; the MapReduce model on multi-thousand node clusters is a brute force solution that wastes vast amounts of energy. While it is rumored that the Google version of MapReduce is faster than the Hadoop version, we did not have access to this code and hence could not test it. We are doubtful again, however, that there would be a substantial difference in the performance of the two version as MR is always forced to start a query with a scan of the entire input file.

This performance advantage that the two database systems share is the result of a number of technologies developed over the past 25 years, including (1) B-tree indices to speed the execution of selection operations, (2) novel storage mechanisms (e.g. column-orientation), (3) aggressive compression techniques with the ability to operate directly on compressed data, and (4) sophisticated parallel algorithms for querying large amounts of relational data. In the case of a column-store database like Vertica, only those columns that are needed to execute a query are actually read from disk. Furthermore, the column-wise storage of data results in better compression factors (approximately a factor of 2.0 for Vertica, versus a factor of 1.8 for DBMS-X and 1.24 for Hadoop); this also further reduces the amount of disk I/O that is performed to execute a query.

Although we were not surprised by the relative performance advantages provided by the two parallel database systems, we were impressed by how easy Hadoop was to set up and use in comparison to the databases. The Vertica installation process was also straight-forward but temperamental to certain system parameters. DBMS-X, on the other hand, was difficult to configure properly and required repeated assistance from the vendor to obtain a configuration that performed well. For a mature product such as DBMS-X, the entire experience was indeed disappointing. Given the upfront cost advantage that Hadoop has, we now understand why it has quickly attracted such a large user community.

Extensibility was another area where we found the database systems we tested lacking. Extending a DBMS with user-defined types and functions is an idea that is now over 25 years old [84]. Either of the parallel DBMSs we tested did a good job on the UDF aggregation task, forcing us to find workarounds when we encountered limitations (e.g. Vertica) or bugs (e.g. DBMS-X).

While all DB systems are tolerant of a wide variety of software failures, there is no question that MR does a superior job of minimizing the amount of work that is lost when a hardware failure occurs. This capability, however, comes with a potentially large performance penalty, due to the cost of materializing the intermediate files between the map and reduce phases. Left unanswered is how significant this performance penalty is. Unfortunately, to investigate this question properly requires implementing both the materialization and non-materialization strategies in a common framework, which is an effort beyond the scope of this work. Despite a clear advantage in this domain, it is not completely clear how significant a factor Hadoop's ability to tolerate failures during execution really is in practice. In addition, if an MR system needs 1,000 nodes to match the performance of a 100 node parallel database system, it is ten times more likely that a node will fail while a query is executing. That said, better tolerance to system failures is a capability that any database user would appreciate.

Many people find SQL difficult to use. This is partially due to having to think differently when solving a problem and that SQL has evolved into a complex language that is quite different than the original design by Don Chamberlin in the 1970s. Though most languages become more complex

over time, SQL is particularly bad as many of its features were designed by competing database companies who each sought to include their own proprietary extensions.

Despite its faults, SQL is still a powerful tool. Consider the following query to generate a list of Employees ordered by their salaries and the corresponding rank of each salary (i.e., the highest paid employee gets a rank of one):

```
SELECT Emp.name, Emp.salary,
     RANK() OVER (ORDER BY Emp.salary)
FROM Employees as Emp
```

Computing this in parallel requires producing a total order of all employees followed by a second phase in which each node adjusts the rank values of its records with the counts of the number of records on each node to its "left" (i.e. those nodes with salary values that are strictly smaller). Although an MR program could perform this sort in parallel, it is not easy to fit this query into the MR paradigm of group-by aggregation. `RANK` is just one of the many powerful analytic functions provided by modern parallel database systems. For example, both Teradata and Oracle support a rich set of functions, such as functions over windows of ordered records.

Two architectural differences are likely to remain in the long run. MR makes a commitment to a "schema later" or even a "schema never" paradigm. But this lack of a schema has a number of important consequences. Foremost, it means that parsing records at run time is inevitable, in contrast to DBMSs, which perform parsing at load time. This difference makes compression less valuable in MR and causes a portion of the performance difference between the two classes of systems. Without a schema, each user must write a custom parser, complicating sharing data among multiple applications. Second, a schema is needed for maintaining information that is critical for optimizing declarative queries, including what indices exist, how tables are partitioned, table cardinalities, and histograms that capture the distribution of values within a column.

In our opinion there is a lot of learn from both kinds of systems. Most importantly is that higher level interfaces, such as Pig [83] and Hive [99], are being put on top of the MR foundation, and a number of tools similar in spirit but more expressive than MR are being developed, such as Dryad [66] and Scope [26]. This will make complex tasks easier to code in MR-style systems and

remove one of the big advantages of SQL engines, namely that they take much less code on the tasks in our benchmark. For parallel databases, we believe that both commercial and open-source systems will dramatically improve the paralellization of user-defined functions. Hence, the APIs of the two classes of systems are clearly moving towards each other. Early evidence of this is seen in the solutions for integrating SQL with MR offered by Greenplum and Asterdata.

Chapter 3

CloudMatcher: A Cloud/Crowd Service for Entity Matching

3.1 Introduction

Entity matching (EM) finds disparate data instances that refer to the same real-world entity. For example, given the two tables in Figure 3.1.a, where each tuple describes a person, we want to find all tuples across the tables that refer to the same real-world person, such as tuples a_1 and b_1 in the figure. Figure 3.1.b shows another example of matching drugs across two tables [74]. Matching tuple pairs are often referred to as *matches*, and variations of this problem are known as record linkage, entity resolution, reference reconciliation, deduplication, etc. (see the related work in Chapter 4).

EM is critical in numerous data management applications, and will become even more so in the age of Big Data and data science. EM is also well-known to be very difficult, raising both accuracy and scalability challenges. As a result, it has been studied intensively over the past several decades, by the database, AI, KDD, and WWW communities, among others. Many EM algorithms have been proposed. Building on these algorithms, many EM systems have been developed (see [71] for a discussion of 33 recent open-source and proprietary EM systems).

Today, however, it is still very difficult for domain scientists to use such EM systems. First, it is often non-trivial and time-consuming to install and learn to use such systems. Second, many such systems do not scale to large tables (e.g., those with several hundreds of thousands of tuples). Third, systems that scale often do so by using a cluster of machines (running Hadoop or Spark). However, many domain scientists do not know how to, or do not want to, install and use a machine cluster. Finally, and most seriously, to use such systems effectively, domain scientists often need

(a)

Table A

	Name	City	State
a_1	Dave Smith	Madison	WI
a_2	Joe Wilson	San Jose	CA
a_3	Dan Smith	Middleton	WI

Table B

	Name	City	State
b_1	David D. Smith	Madison	WI
b_2	Daniel W. Smith	Middleton	WI

Matches: (a_1, b_1), (a_3, b_2)

(b)

Table A

	Name	Generic name	Strength
a_1	Dolorex	Acetaminophen/ Phenyltolx	500MG/ 30MG
a_2	Wart Remover	Salicylic Acid	17%
a_3	Maki-DM	Guaifenesin/ Dextromethorphan	500MG/ 300MG

Table B

	Name	Generic name	Strength
b_1	Q-Tussin DM	Guaifenesin/ Dextromethorphan	100-10M G/5
b_2	Dolorex	Acetaminophen/ Phenyltolx	500MG-3 0MG

Matches: (a_1, b_2), (a_3, b_1)

Figure 3.1: Examples of EM across two tables.

to know quite a bit about EM, e.g., knowing about string similarity measures (e.g., edit distance, Jaccard, TF/IDF, etc.) and when to use which measure, about machine learning models and when to use which model, etc. (see Sections 3.2-3.3). Obtaining such knowledge is difficult even for EM experts, let alone for most domain scientists.

The CloudMatcher Service: To address these problems, in the past few years we have been building CloudMatcher, a cloud/crowd service for EM. We envision CloudMatcher to be a fast, easy-to-use, scalable, and highly available service on the Web. Specifically, to use this service, a user simply needs to go to CloudMatcher's Web site, uploads two tables to be matched, performs some basic pre-processing, then pushes a button. CloudMatcher will perform EM end to end. To do so, it will use crowd workers on Amazon's Mechanical Turk (or some other crowdsourcing platform) to label tuple pairs (as matched / no-matched). The user just has to pay for the labeling. Alternatively, instead of using crowdsourcing, the user can just label these tuple pairs. At the end, CloudMatcher will return the desired matches. In the backend, CloudMatcher performs EM using a machine cluster that our group will maintain.

As described, when using CloudMatcher, the user does not need to install or learn how to use any complicated system (using CloudMatcher should be very straightforward). The user does not have to know EM (e.g., knowing string similarity measures). He or she will only perform simple actions such as labeling a tuple pair as matched / no-matched. Alternatively, if the user is not even willing to label the tuple pairs, then he or she can pay to "outsource" that work to a crowd of

workers (assuming that the data is not sensitive and that crowd workers can be quickly trained to label tuple pairs). Finally, the system can scale to tables of millions of tuples and can automatically add more machine resources as necessary.

Our initial motivation for building CloudMatcher is to serve the EM needs of domain scientists at the University of Wisconsin, Madison, and affiliated research institutions (e.g., Marshfield Clinic), and in fact, we have deployed such a service at UW-Madison, with highly promising results (see Section 3.6). In the near future, we will open up the service to the broader public, and make the service open source, so that it can also be deployed at other places, as appropriate.

Outline and Contributions: In the rest of this chapter, we will describe our ongoing work developing CloudMatcher, our initial experience using it, and lessons learned. Specifically, we first describe the EM problem and typical EM solutions (Section 3.2). Next, we describe Corleone, which performs EM end to end, using only crowdsourcing. We then describe Falcon, which uses a cluster of machines to scale up Corleone to tables of millions of tuples (Section 3.3). Both the Corleone and Falcon works have been recently published [56, 39].

We then describe CloudMatcher, which implements Falcon as a cloud service. It turns out that doing so raises challenges in terms of effective user interaction, fault tolerance, crash recovery, and scalability (to hundreds of EM tasks that users may submit at any time). In this chapter we will describe these challenges in detail, then describe our initial solutions (Section 3.4).

Using this initial solution, we have implemented a first version of CloudMatcher, and used it for a variety of real-world EM tasks on real datasets from industry and academic partners. We describe our experience and lessons learned, regarding debugging and explaining, understanding data/problem/solution, and interaction with the user, among others (Section 3.6).

We then perform scalability experiments with a deployment of CloudMatcher that mimics how we could eventually deploy the system at UW-Madison. In these experiments, we vary the workload placed on CloudMatcher and report numbers relevant to the end-user experience. Finally, we present several scenarios of how a CloudMatcher user can use the system in ways beyond the Falcon EM workload. As far as we can tell, this is the first work that publicly describes how to

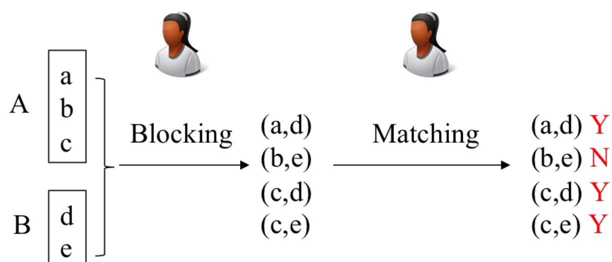

Figure 3.2: Most current EM solutions consist of a blocking step and a matching step.

develop a cloud-based EM service and discusses initial experience and lessons learned (see also the related work chapter).

CloudMatcher is a part of a major project at UW-Madison on developing data cleaning and integration tools for data scientists. Another major part of this project develops Magellan, a Python package that helps the user perform entity matching end to end [71, 12].

3.2 Preliminaries

In this section we describe the EM problem considered in this chapter, the blocking and matching steps of typical EM solutions, and recent crowdsourcing solutions. Subsequent sections will build on these to discuss the Corleone, Falcon, and CloudMatcher solutions.

Entity Matching: Many EM variations exist, such as matching across two tables, matching within a single table, matching mentions in text documents into a knowledge base, etc. (see the related work chapter). In this chapter, we will consider the problem of matching across two tables, specifically, given two tables A and B, find all tuple pairs $(a \in A, b \in B)$ that refer to the same real-world entity (see Figures 3.1.a-b). This problem setting is very common in practice. Our solution however can also be applied to two other common settings: matching tuples within a single table (known as *deduplication*), and matching a set of tuple pairs.

Blocking and Matching Steps: Most current EM solutions perform a blocking step then a matching step. The blocking step applies a heuristic to remove tuple pairs judged obviously not matched (i.e., "blocking" these tuple pairs from further consideration). The matching step then

predicts Y/N, i.e., matched/not-matched, for each remaining tuple pair. Figure 3.2 illustrates these two steps (here the blocking step removes two tuple pairs out of six possible pairs).

Blocking heuristics are typically specified by the user. For example, when matching the two tables of persons in Figure 3.1.a, a user may specify that "two persons whose states disagree do not match". Using this heuristic, the blocking step would remove the tuples (a_2, b_1) and (a_2, b_2) because their states (CA and WI) are not the same.

Blocking is necessary because matching all tuple pairs in the Cartesian product of the two input tables A and B would be too expensive, e.g., if each table has 100K tuples, $A \times B$ would have 10B tuple pairs. Hence, we need a way to quickly remove as many obviously non-matched tuple pairs as possible, before applying the time-consuming matching step to the remaining tuple pairs.

Of course, it would not make sense to do blocking by *enumerating* all tuple pairs in $A \times B$ then removing those judged obviously non-matched, because the enumeration step alone is already very time-consuming. Instead, blocking is typically done by using the blocking heuristic to enumerate only those tuple pairs judged possibly matched. For example, given the above heuristic about disagreeing states, we can build an inverted index over Table B, such that given a state, the index will return all tuples in B with that state value. Next, given a tuple in Table A, we can consult the index to find only those tuples in Table B that share the same state (e.g., WI), then enumerate only those tuple pairs.

Numerous solutions have been proposed for the blocking and matching steps, focusing on accuracy and scalability (see the related work chapter).

Crowdsourcing: In the past few years, crowdsourcing has been increasingly applied to EM. In crowdsourcing, certain parts of a problem are "farmed out" to a crowd of workers to solve. As such, crowdsourcing is well suited for EM, and indeed many crowdsourced EM solutions have been proposed (see Chapter 4).

To illustrate such crowdsourcing solutions, consider again the EM workflow in Figure 3.2. In this workflow, recall that after the blocking step, we apply a matcher to predict Y/N for the surviving four tuple pairs. We can now send the three pairs with the Y prediction to crowd workers to help verify these predictions. Suppose that the crowd verifies that (a, d) and (c, e) are indeed

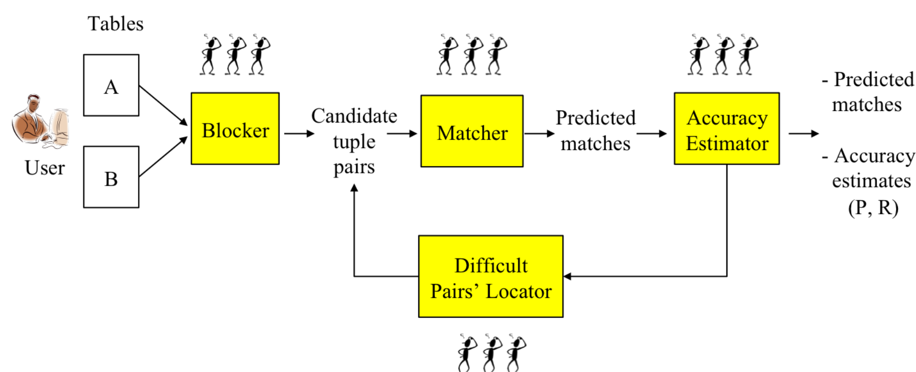

Figure 3.3: The EM workflow of Corleone.

matches, but (c, d) is not. Then we would output Y predictions for only the two pairs (a, d) and (c, e), thereby improving the precision of the matching process. To increase the reliability of the answers obtained from the crowd, a typical solution is to obtain three answers from three crowd workers for each question, then take the majority answer to be the final answer from the crowd. Current works use the crowd to verify predicted matches (as illustrated above), find the best questions to ask the crowd, and find the best UI to pose such questions, among others (Chapter 4). Many crowdsourcing platforms can be used for the above purpose. The most popular one is Amazon's Mechanical Turk. Others include CrowdFlower, Samasource, oDesk, Elance, etc.

3.3 The Corleone & Falcon Systems

We now describe Corleone and Falcon, which form the basis for the CloudMatcher cloud/crowd EM service.

3.3.1 The Corleone System

Corleone was motivated by the fact that while recent crowdsourced EM works are promising, they are limited in that they crowdsource only parts of the EM workflow, *requiring a developer* who knows how to code and match to execute the remaining parts. For example, several recent solutions require a developer to write heuristic rules, called *blocking rules*, to reduce the number of candidate pairs to be matched, then train and apply a matcher to the remaining pairs to predict

matches. The developer must know how to code (e.g., to write rules in Python) and match entities (e.g., to select learning models and features).

As such, it is very difficult for an organization to concurrently deploy multiple crowdsourced EM solutions, because crowdsourcing each still requires a developer and there are simply not enough developers. The Corleone [56] system, a solution that crowdsources the *entire* EM workflow, thus requiring no developers, addresses this problem. For example, in the blocking step, instead of asking a developer to come up with blocking rules, Corleone asks a crowd to label certain tuple pairs as matched/no-matched, uses these pairs to learn a classifier, then extracts blocking rules from the classifier (as we will explain soon). Other steps in the EM workflow also heavily use crowdsourcing, but no developers. Thus, Corleone is said to perform *hands-off crowdsourcing* for entity matching.

Specifically, given two tables A and B, Corleone applies the EM workflow in Figure 3.3 to find all tuple pairs $(a \in A, b \in B)$ that match. This workflow consists of four main modules: Blocker, Matcher, Accuracy Estimator, and Difficult Pairs Locator.

The Blocker generates and applies blocking rules to $A \times B$ to remove obviously non-matched pairs (Figure 3.4.b shows two such rules). Since $A \times B$ is often very large, considering all tuple pairs in it is impractical. So blocking is used to drastically reduce the number of pairs that subsequent modules must consider. The Matcher uses active learning to train a random forest classifier, then applies it to the surviving pairs to predict matches. The Accuracy Estimator computes the accuracy of the Matcher. The Difficult Pairs Locator finds pairs that most likely the current Matcher has matched incorrectly. The Matcher then learns a better random forest to match these pairs, and so on, until the estimated matching accuracy no longer improves.

Corleone is distinguished in that the above four modules use no developers, only crowdsourcing. For example, to perform blocking, most current works would require a developer to examine Tables A and B to come up with heuristic blocking rules (e.g., "If prices differ by at least $20, then two products do not match"), code the rules (e.g., in Python), then execute them over A and B. In contrast, the Blocker in Corleone uses crowdsourcing to learn such blocking rules (in a

machine-readable format), then automatically executes those rules. Similarly, the remaining three modules also heavily use crowdsourcing but no developers.

Corleone can also be run in many different ways, giving rise to many different EM workflows. The default is to run multiple iterations until the estimated accuracy no longer improves. But the user may also decide to just run until a budget (e.g., $300) has been exhausted, or to run just one iteration, or just the Blocker and Matcher, or just the Matcher if the two tables are relatively small, making blocking unnecessary, etc.

3.3.2 The Falcon System

As described, Corleone is highly promising. But it suffers from a major limitation: it executes mostly a single-machine in-memory EM workflow, and thus does not scale at all to tables of moderate and large sizes. For example, using Corleone to match tables of 50K-200K tuples would take weeks, rendering the system impractical.

The Falcon, system, is a solution that scales up Corleone to tables of millions of tuples. Falcon introduced three key ideas. First, it defines basic operators and use them to model the EM workflow of Corleone as a directed acyclic graph (DAG). Next, it scales up the operators, using MapReduce if necessary. Finally, it optimizes within and across operators.

In what follows we discuss these ideas, but only to the extent necessary for the purpose of this chapter (see [39] for a complete description of Falcon).

3.3.2.1 The Entity Matching Workflow Considered by Falcon

Currently, Falcon considers only EM workflows that consist of the Blocker followed by the Matcher, or just the Matcher. We now describe the Blocker and the Matcher, focusing only on the aspects necessary to understand Falcon.

The Blocker: The key idea underlying this module is to use crowdsourced active learning to learn a random forest based matcher (i.e., binary classifier) M, then extract certain paths of M as blocking rules.

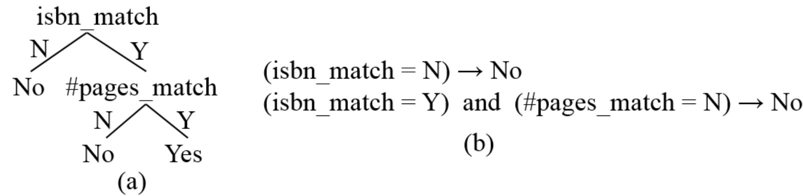

Figure 3.4: (a) A decision tree learned by Corleone and (b) blocking rules extracted from the tree.

Specifically, learning on $A \times B$ is impractical because it is often too large. So this module first takes a small sample of tuple pairs S from $A \times B$ (without materializing the entire $A \times B$), then uses S to learn matcher M.

To learn, the module first asks the user to supply two positive examples (i.e., two tuple pairs labeled matched) and two negative examples (i.e., two tuple pairs labeled non-matched). Next, it uses these "seed" examples to train an initial random forest matcher M, uses M to select a set of controversial tuple pairs from sample S, then asks the crowd to label these pairs as matched / no-matched. In the second iteration, the module uses these labeled pairs to re-train M, uses M to select a new set of tuple pairs from S, and so on, until a stopping criterion has been reached.

At this point the module returns a final matcher M, which is a random forest classifier consisting of a set of decision trees. Each tree when applied to a tuple pair will predict if it matches, e.g., the tree in Figure 3.4.a predicts that two book tuples match only if their ISBNs match and the number of pages match. Given a tuple pair p, matcher M applies all of its decision trees to p, then combines their predictions to obtain a final prediction for p.

Next, the module extracts all tree branches that lead from the root of a decision tree to a "No" leaf as candidate blocking rules. Figure 3.4.b shows two such rules extracted from the tree in Figure 3.4.a. The first rule states that if two books do not agree on ISBNs, then they do not match.

Next, for each extracted blocking rule r, the module computes its precision. The basic idea is to take a sample T from S, use the crowd to label pairs in T as matched / no-matched, then use these labeled pairs to estimate the precision of rule r. To minimize crowdsourcing cost and time, T is constructed (and expanded) incrementally in multiple iterations, only as many iterations as necessary to estimate the precision of r with a high confidence (see [56]).

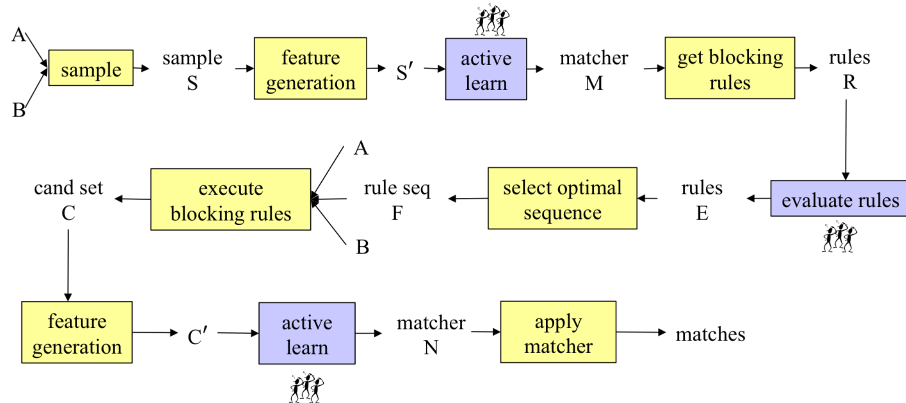

Figure 3.5: The EM workflow of Falcon as a DAG of basic operators.

Finally, the Blocker applies a subset of high-precision blocking rules to $A \times B$ to remove obviously non-matched pairs. The output is a set of candidate tuple pairs C to be passed to the Matcher.

The Matcher: This module applies crowdsourced active learning on C to learn a new matcher N, in the same way that the Blocker learns matcher M on sample S. The module then applies N to match the pairs in C.

3.3.2.2 Modeling the EM Workflow as a DAG of Basic Operators

As described, the workflow of Falcon can be modeled as a DAG of basic operators as shown in Figure 3.5. In this DAG, given two tables A and B to be matched, Falcon first takes a sample S of tuple pairs. Next, Falcon uses the schemas of A and B to automatically generate a set of features (not shown in the figure), then use these features to convert each tuple pair in S into a feature vector, thereby converting S into a set of feature vectors S'.

Next, Falcon performs active learning with the crowd[1] over S' to learn a matcher M, then extracts a set of blocking rules R from M. It then uses the crowd to evaluate these rules and select a sequence of rules F judged to be optimal (see [56]). Next, it executes F over the tables A and B. This produces a set of candidate tuple pairs C.

[1]For clarity, we omit the step of asking the user to supply "seed" examples to avoid making the figure too cluttered.

At this point, the blocking step ends, and the matching step begins. Falcon first converts each tuple pair in C into a feature vector, thereby converting C into a set of feature vectors C'. Then it performs active learning (again) with the crowd to learn a matcher N. Finally, it applies N to the feature vectors in C' to predict matches.

The above workflow uses eight basic operators. As described, these operators involve complex rules, crowdsourcing, and machine learning, and can be used to compose a variety of EM workflows (see [39]).

It is important to note that Falcon includes efficient implementations for these operators (using MapReduce where necessary), and there are techniques to optimize within and across operators. We omit further details here (see [39]).

3.4 The CloudMatcher Service

We are now in a position to discuss CloudMatcher, the cloud/crowd service that we have been building. In what follows we discuss the motivations, goals, and then our ongoing work on CloudMatcher.

3.4.1 Motivations and Goals

As mentioned in the introduction, we want to provide EM services to hundreds of domain scientists at UW-Madison and affiliated institutions. Domain scientists often do not know how to, or are reluctant to, deploy EM systems locally (such systems often require a Hadoop cluster, as discussed earlier). So we want to provide such EM services on the cloud, supported in the backend by a cluster of machines maintained by our group.

During any week, we may have tens to hundreds of submitted EM tasks running. Many of these tasks require blocking, but the users do not know how to write blocking rules (which often involve string similarity functions, e.g., edit distance, Jaccard, TF/IDF), and we simply cannot afford to ask our busy developers to assist the users in all of these tasks.

Thus, we planned to deploy the hands-off solution of Corleone. A user can just submit the two tables to be matched on a Web page and specify the crowdsourcing budget. We will run Corleone

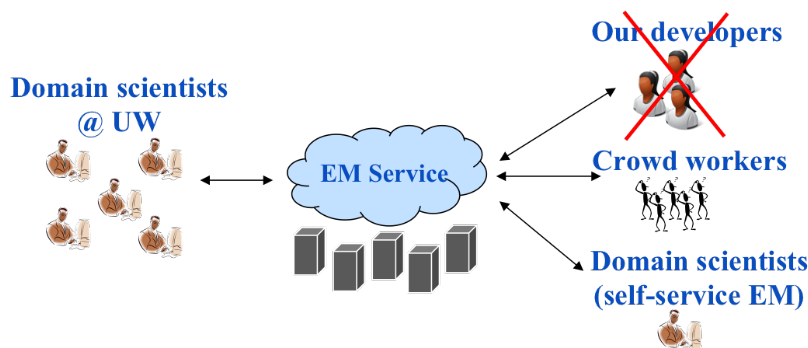

Figure 3.6: CloudMatcher as a cloud/crowd EM service.

internally, which uses the crowd to match. As described, Corleone seems perfect for our situation. Unfortunately, it executes mostly a single-machine in-memory EM workflow, and does not scale at all to tables of moderate and large sizes. So we will use Falcon, which scales to tables of millions of tuples. In particular, we will execute the EM workflow of Falcon described in Figure 3.5.

It is important to note that if users do not want to engage the crowd, they can label the tuple pairs themselves. This in effect provides a self-service EM for the users. Most users we have talked to, however, prefer if possible (e.g., if the data is not sensitive or too difficult for the crowd to match) to just pay a few hundred crowdsourcing dollars to obtain the result in 1-2 days. Figure 3.6 illustrates both the crowdsourcing and the self-service options discussed above.

Our goals for CloudMatcher are as follows:

- *Efficient resource consumption:* Use minimal machine and crowd resources to perform EM tasks.

- *Fault tolerance:* If a machine or process crashes, can recover and continue gracefully.

- *Crash recovery:* Executing an EM task can take hours (or days if crowdsourcing is involved). As a result, crash recovery is critical. Specifically, if CloudMatcher crashes in the middle of an EM task, when resumed, it should continue where it crashed, instead of restarting the task from scratch.

- *Scaling:* CloudMatcher should scale, both for a single EM task and for multiple EM tasks. That is, a single EM task should execute as fast as possible, and the system should be able to handle hundreds of EM tasks concurrently, without being slow on anyone of them.

- *Optimization:* In order to scale, ideally the system must be such that there are multiple opportunities for optimization, and the system can make use of these opportunities.

- *Efficient management of heterogeneous execution environments:* When executing an EM workflow, each step in the workflow may require its own execution environment. For example, one step is to be executed in Python on a single machine, whereas another step requires Java over a Hadoop cluster. CloudMatcher should be able to handle a broad range of such heterogeneous environments.

- *Smooth user experience:* The user should have a very smooth experience with the system. The GUI must be intuitive and requires very little guessing to work with. The system latency should be at interactive speed, i.e., it should not take more than a few seconds to respond to the user. If the user works in a browser, then stops the work (say for lunch), or closes the browser and opens another one in the same or another machine, then the user should be able to seamlessly continue working.

- *Progress report:* The system should tell the user where it is in the EM process and give estimations on how much longer it will take to complete certain tasks.

- *Visualization:* The system should provide as much visual information to the user as possible, especially in terms of its progress.

The above set of goals makes it clear that developing CloudMatcher is not a simple matter of deploying Falcon. For example, Falcon focuses on techniques to scale up a single EM workflow. It does not focus on goals such as fault tolerance, crash recovery, smooth user experience, etc.

3.4.2 Limitations of Current Solutions

To implement CloudMatcher, the simplest solution is to convert each submitted EM task into a Falcon DAG, as shown in Figure 3.5, then execute the DAG using a workflow management system (WMS) such as Luigi, Airflow, or Pinball. Many such WMSs have been developed. In theory, they can guarantee certain kinds of fault tolerance and crash recovery. For example, the WMS can write the output of each node in the DAG to disk, and thus can guarantee that in the case of a crash, DAG execution does not have to restart from scratch. In addition, such WMSs can easily handle multiple Falcon DAGs being run concurrently, as is common in cloud settings.

There are however two major problems with the Falcon DAG. First, the DAG's granularity is too coarse, rendering it not effective for crash recovery, optimization, and best usage of resources. Specifically, some of the steps in this DAG can take a very long time, and currently there is no easy way to save their partial results for crash recovery. Consider for example a step that performs active learning with the crowd to learn a matcher. This step can perform up to 30 iterations of active learning, and can take hours or days (if the crowd is slow). Ideally, we should be able to save the output of each iteration, so that in the case of a crash, we can resume at the crashed iteration. However, since currently the entire step is modeled as a single node in the Falcon DAG, it is difficult for us to perform such partial saving. Also, coarse steps make it difficult to optimize within and among the steps.

Another problem is that some of the steps in the Falcon DAG involve user interaction. For example, before we can start the first active learning process (on sample S'), we need to ask the user to supply at least two positive examples and two negative examples. (This step is not shown in Figure 3.5, to avoid making the figure too cluttered.) Machine-wise this step does not take long to execute. But it involves asking for an input from the user, and this can often cause a problem. The user may stop in the middle, go to lunch, have a phone call, etc., in which case this step will be left "hanging", waiting for the user to get back. If not implemented carefully, this step will continue to "hog" resources until the user responds. The same problem arises for any step involving crowdsourcing.

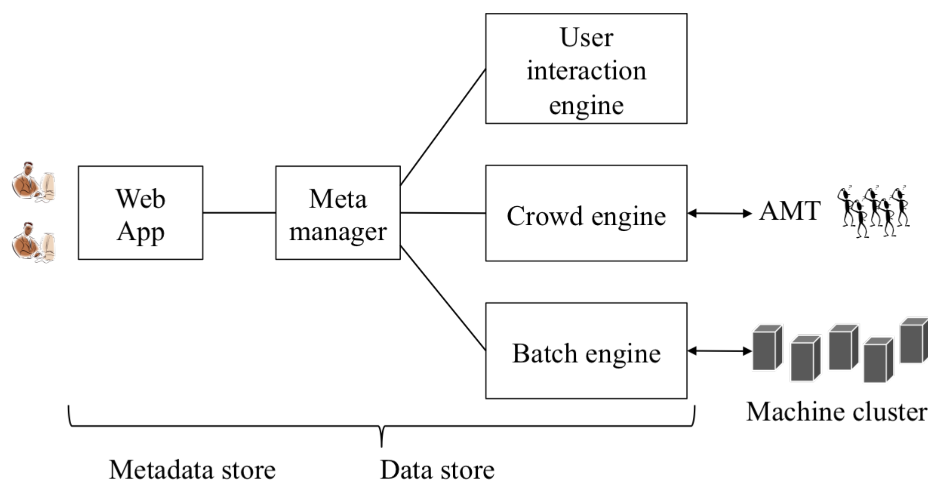

Figure 3.7: The CloudMatcher architecture

Note that this second problem is relatively new, because the current workflow management systems (e.g., Luigi, Airflow, etc.) typically are designed to execute DAGs that can be run in batch mode. They are not designed for efficiently executing DAGs that can involve user interaction in the middle. In theory, they can still be used to execute such DAGs, but it will typically result in inefficient use of resources (as the executor waits for the user to get back from lunch, say) and can negatively affect many other DAGs that are being concurrently executed.

3.4.3 Key Ideas of the CloudMatcher Solution

To address the above limitations, in CloudMatcher we employ the following key ideas:

- We convert the Falcon DAG into an EM workflow at a much finer granularity, to maximize the opportunities for crash recovery, optimization, and efficient resource usage. The new EM workflow is not a DAG, as it involves loops (in addition to conditionals).

- For the new CloudMatcher EM workflow, we clearly define tasks (i.e., nodes of the workflow graph) that are *interactive* (i.e., interacting with a user or a crowd to request some input).

- We partition the CloudMatcher workflow into "pieces" such that each piece is either an *interactive* task or a workflow fragment that can be executed entirely in *batch* mode.

- We define three kinds of execution engines: user interaction (UI) engine, crowd engine, and batch engine. The UI engine is designed to execute interactive tasks efficiently, and similarly the crowd engine is designed for executing tasks that require crowdsourcing. Finally, the batch engine is designed for executing batch-mode workflow fragments.

- We use a meta-manager to execute the entire CloudMatcher workflow, by executing each piece of the workflow using the appropriate execution engine.

- Finally, we divide the responsibilities for managing fault tolerance and crash recovery appropriately among the meta-manager and the execution engines.

Putting these ideas together produces the CloudMatcher architecture shown in Figure 3.7. In this figure, the Web app module is responsible for authentication, account creation, and processing GET/POST requests from users. Given a submitted EM task, the meta-manager converts it into an EM workflow, then partitions the workflow into pieces, where each piece is interactive or batch by nature, as described earlier. The meta-manager then executes these pieces using the appropriate execution engines. The meta-manager and the execution engines will coordinate the management of fault tolerance and crash recovery, using the meta-data store (which records for example which nodes in which graph fragments have been executed) and the data store (which stores the input/output and intermediate data for the workflow nodes).

We now elaborate on the most important aspects of the above solution architecture.

3.4.4 The Entity Matching Workflow of CloudMatcher

Recall that we want to break each long-running step in the Falcon DAG into many much smaller steps, whenever possible, and isolate all "points" in the Falcon DAG where we need user interaction, then make those "points" into their own steps.

Figure 3.8 shows the resulting workflow for CloudMatcher. This workflow is quite long and consists of three parts: Part (a) followed by Part (b) followed by Part (c). We now briefly describe this workflow, and contrast it to the Falcon one.

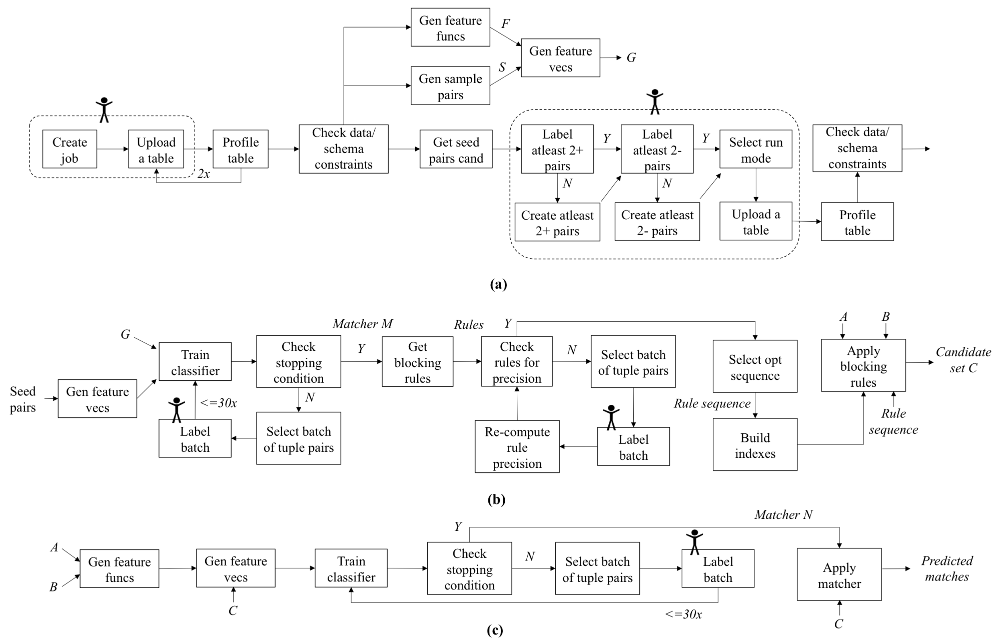

Figure 3.8: A refinement of the Falcon DAG, to create the workflow for CloudMatcher. The resulting workflow consists of three parts, as shown in (a)-(c).

- The very first task in the CloudMatcher workflow is “Create job” (see Figure 3.8.a). In this task, the user goes to the CloudMatcher Web page, creates a job (whose goal is to match two tables), and supplies some job related information (e.g., contact email). Next, the user uploads the first table, Table A. The system then profiles this table, e.g., counting the total number of tuples in the table and showing that to the user (to confirm that the table has indeed been uploaded and everything appears to be in order). See the next two tasks on the figure. These two tasks will be repeated one more time for Table B (the number “2x” on the arrow from “Profile table” back to “Upload table” indicates the maximal number of iterations for this loop).

- The first two tasks, “Create job” and “Upload a table”, are *interactive*, in that they must interact with the user to obtain information. As discussed earlier, we “isolate” such interactions into their own tasks. On the figure, we show such tasks either with a small human figure or inside a dotted box with a small human figure. The third task, “Profile table”, is not interactive. We refer to such tasks as *batch tasks*, as they can be executed in batch mode.

- The next task, "Check data/schema constraints", verifies certain integrity constraints, e.g., trying to identify a key column, and if none found, then create a key column for the table. The subsequent tasks on Figure 3.8.a obtain a sample S of tuple pairs, convert S into a set of feature vectors G, and obtain at least two positive examples and two negative examples from the user.

- Once the workflow fragment on Figure 3.8.a ends, we continue with the workflow fragment on Figure 3.8.b. Here, we first convert the two positive and two negative examples (called "seeds") into feature vectors, then start the active learning process. Notice that this active learning process was earlier just a single task in the Falcon workflow. Here it has been broken into a loop of four tasks: "Train classifier", "Check stopping condition", "Select batch of tuple pairs", and "Label batch". We repeat this loop up to 30 times. Notice also that the first three tasks of this loop are batch task, whereas the last one ("Label batch") is an interactive task.

- Once the active learning finishes, we obtain a matcher M, from which we obtain a set of candidate blocking rules, then evaluate the rules and so on. We omit further description of the rest of the tasks in Figure 3.8.b, as well as the tasks in Figure 3.8.c (as they are similar to those in Figure 3.8.b).

Compared to the Falcon DAG, the CloudMatcher workflow is different in the following aspects. First, it is at a much finer granularity, with all long-running tasks being broken down into as many smaller tasks as possible. Second, the workflow is no longer a DAG. It now has loops (in addition to conditionals). Finally, the workflow is a combination of interactive tasks and batch tasks.

3.4.5 Partitioning the Entity Matching Workflow

Given an EM workflow as described above, the meta-manager of CloudMatcher partitions it into workflow fragments, such that each fragment is strictly interactive or batch by nature. Figure 3.9 shows how the first part of the CloudMatcher workflow (which is the part shown in Figure 3.8.a) is partitioned into eight interactive fragments (each of which is in yellow) and six batch

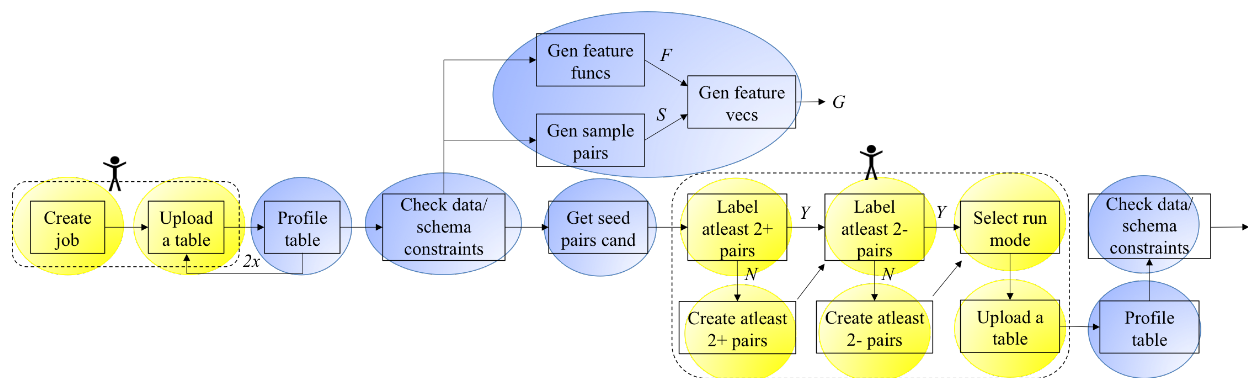

Figure 3.9: A partitioning of the first part of the CloudMatcher workflow into interactive and batch fragments.

fragments (in blue). The uppermost batch fragment, for example, consists of three tasks: "Gen feature funcs", "Gen sample pairs", and "Gen feature vecs".

3.4.6 Executing the Workflow Fragments

After partitioning, the meta-manager executes the workflow fragments, each in the appropriate execution engine. Specifically, each batch fragment will be executed using the batch engine, which uses a well-known current workflow management system, such as Luigi, Airflow, or Pinball. If an interactive workflow fragment does not require crowdsourcing, then it only needs to interact with a single user to request some input. In this case, we execute the fragment using the user interaction (UI) engine. Otherwise, we execute the fragment using the crowd engine.

The meta-manager uses the metadata store and the data store to coordinate the execution of the various workflow fragments, and to handle fault tolerance and crash recovery.

3.4.7 The User Interaction/Crowd Engines

Finally, we describe the working of the UI engine and the crowd engine. Consider executing an UI task E. The UI engine starts by sending a request for the user to do an action (e.g., providing the name of the job to be created and a contact email address) to the user's Web browser (via the

Web app). At some point, after the user has filled out the requested information, he or she will click the submit button, which sends a request to the Web app, which in turn contacts the UI engine.

The engine then processes the information from the user. If this information is complete, then the engine indicates to the meta-manager that this UI task E has been completed. Otherwise, if the information is incomplete (e.g., only the job name is provided, the requested email address is still missing), then the UI engine sends another request (for email address) to the user's Web browser, and so on.

As described, the UI engine does not run a process that is dedicated with trying to interact with the user. Instead, it operates in a "transactional" mode, in which it sends a request to the user, then "moves on to something else", and returns only when the user has sent a response. This transactional mode is desired because we simply do not know how long it would take for the user to process the request (he or she may stop for lunch in the middle, etc.), and hence we do not want to run a dedicated process to wait on the user.

If the task is not UI, but instead requires crowdsourcing, then the situation is a bit more involved. Suppose the task is to label 20 examples (for active learning). To execute, the crowd engine will send these 20 examples to a crowdsourcing platform, say Amazon's Mechanical Turk (AMT), for labeling. The problem is that AMT does not get back to the crowd engine (i.e., there is nothing that is equivalent to "a user clicking the submit button" in the AMT case). So the crowd engine needs to "ping" AMT at regular intervals to check on the progress of the labeling task.

Figure 3.10: The services of CloudMatcher.

3.5 Workflows as Composites of Services

Version 1.0 of CloudMatcher implemented only the above Falcon EM workflow. As we interacted with real users, however, we observed that many users want to flexibly customize and experiment with different EM workflows. As a result, in Version 2.0, we solved this problem by (a) extracting a set of basic services from the Falcon EM workflow and making them available on CloudMatcher, and (b) allowing users to flexibly combine them to form different EM workflows, including the original Falcon workflow. Figure 3.10 shows examples of services that we currently provide.

Basic services include uploading a dataset, profiling a dataset, edit the metadata of a dataset, sampling, generating features, training a classifier, etc. We have combined these basic services to provide *composite services*, such as active learning, obtaining blocking rules, and Falcon (see the bottom of the Figure 3.10). For example, the user can invoke the “Active Learning” service to ask CloudMatcher to automatically take the user through several more basic services. Consider the

final portion of the Falcon workflow for matching (part ‘c’ in Figure 3.8). The middle portion of the flow, from “Train Classifier” up until the input of “Apply Matcher” is an active learning process. The “Active Learning” composite service in Figure 3.10 uses the “Create a classifier”, “Train Classifier”, “Identify informative examples”, and “Label data” basic services to automatically run an Active Learning process. The composite service implements the logic for the “Check stopping condition” step and re-invokes the “Identify informative examples” and “Label data” services as necessary. As another example, the user can invoke the “Falcon” service to execute the end-to-end Falcon EM workflow, which will automatically guide the use through each service seamlessly. CloudMatcher is useful for more than just entity matching: users will be able to use the basic services in interesting ways, as we will show later.

3.6 Deployment and Architecture

We now describe the initial deployment of CloudMatcher in terms of hardware configuration and software deployment. For the experiments in Section 3.7.1, we have deployed CloudMatcher on a 4-node Amazon EC2 cluster, where each node has a 16-core Intel Xeon E5-2676 2.4GHz processor and 64GB of RAM. For crowdsourcing, we use Mechanical Turk and assign each question to three crowd workers, paying 2 cents per answer. In each run we used common turker qualifications to avoid spammers, such as allowing only turkers with at least 100 approved HITs and 95% approval rate.

For the experiments in Section 3.7.2, we have opted to use less powerful EC2 servers in our experiment. This is in part to keep the costs down, and in part to use machines that are more similar to resources we will ultimately use at UW-Madison. The cluster configuration we have chosen for these experiments is a 4-node EC2 cluster of `M4-Xlarge` instances. Each instance (node) has a 4-core Intel Xeon 2.4GHz Broadwell or Haswell processor and 16GB of RAM. For storage, each instance has an attached 512GB Elastic Block Store (EBS) volume. These are the “General-Purpose” SSD drives, with the default AWS configuration of 384 MB/s bandwidth (1536 256KB IOPS). The network performance on these instances are rated as “high”.

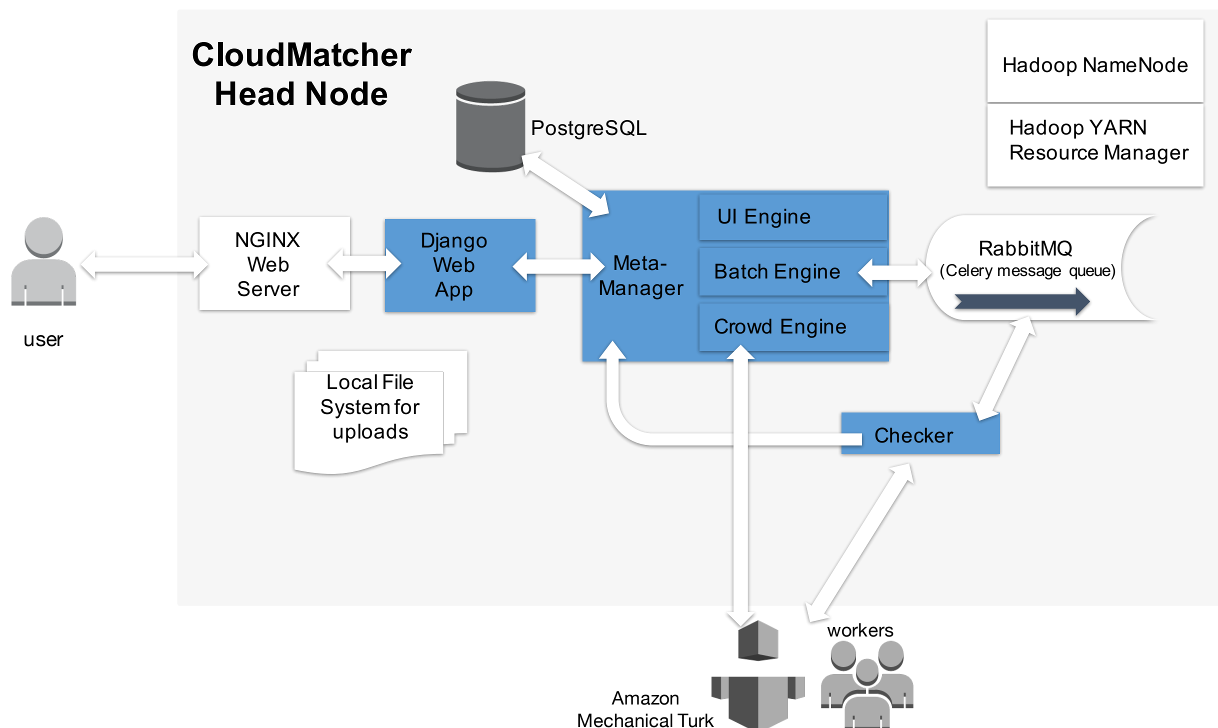

Figure 3.11: CloudMatcher Head Node Diagram. Components in blue are software written for CloudMatcher

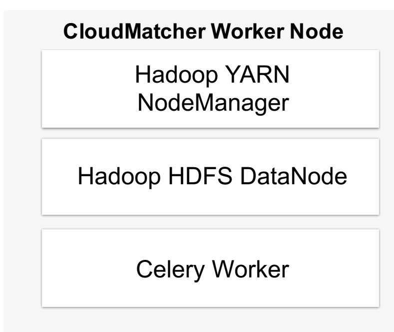

Figure 3.12: CloudMatcher Worker Node Diagram

In a cluster of four nodes, the software components are distributed between one node designated as the "head" node, and three nodes that serve as "workers". On the "head" node (Figure 3.11), we run the core of the CloudMatcher system. The "web app" is built using the Django web framework, and constructs the HTML user interface for the user. The "web app" is also responsible

for managing user logins. For most operations, however, the "web app" communicates with the "meta-manager" to request new workflows and to pass the requests of the "UI Engine" to and from the user. The "web app" sits behind an NGINX web server, which is a best practice with Python web applications.

The head node runs the management infrastructure for a Hadoop cluster, specifically, a Hadoop NameNode and a Hadoop YARN Resource manager. The worker nodes (Figure 3.12) each run a Hadoop DataNode process and a Hadoop YARN NodeManager process. This means that MapReduce jobs and the block data of HDFS are only present on the worker nodes.

The "meta-manager" (MM) is also written in Python, using the Twisted network framework. The MM stores its internal metadata and state in a PostgreSQL database. The three engines are internal to the MM and are not stand-alone processes. The "Crowd" engine formats and sends requests to the Amazon Mechanical Turk services, and stores a "handle" to that request in the MM metadata. This handle can be used later to track the status of the MTurk request. The "Batch" engine submits jobs to the Celery distributed task queue. Celery does not use a server directly, and instead rides on top of a messaging service, for CloudMatcher, we chose RabbitMQ as the messaging layer for the distributed task queue. Like the "Crowd" engine, the "Batch" engine stores a handle to jobs that have been submitted to Celery in the MM metadata.

Celery worker processes attach to the RabbitMQ messaging system to learn about new jobs in the system; RabbitMQ manages concurrent access and ensures that a job goes to only one of the attached workers. Celery decodes the message and invokes the job. For some jobs, the Celery portion is a simple pass-through to in turn submit the job to Hadoop and wait for completion, for other jobs the Celery job implements the job directly using a mixture of PyData libraries.

The MM is meant to be simple and maintains no in-memory state between requests. This design allows the MM to scale and recover from failure like a web application in that each request is independent, served immediately, and then cleaned up. This makes a restart of the MM trivial, but presents a challenge for asynchronous operations like batch jobs or requests to MTurk: there is no active thread waiting for completion of those operations in the MM. To work around this, we have introduced the "checker" process. The checker reads the handles to "batch" or "crowd"

engine jobs from the MM metadata, and connects to either Celery or MTurk periodically to obtain updated state information for those jobs. When a job changes state in one of these external systems, the "checker" notifies the MM using the same request serving mechanisms as the "web app" to advance the workflow.

Regardless of the engine, at the beginning and end of each task in a workflow, the task copies any data it may need from HDFS to the local disk at task startup, and back to HDFS at the end of the task. We do this so that we can schedule different tasks in a workflow on different nodes, and to support fault tolerance. However, at the moment, we only run the Celery worker on one node and are not yet taking advantage of the scheduling freedom the data movement allows us; worker nodes only execute Hadoop jobs.

3.7 Experience with CloudMatcher

This section describes our experience with CloudMatcher and makes the following three claims:

- CloudMatcher can be effective for multiple datasets. (Subsection 3.7.1)
- CloudMatcher will scale sufficiently to build a real-world service for a moderate-sized organization, such as the UW-Madison campus, using reasonable resources, and meet what we believe is a realistic expected demand. (Subsection 3.7.2)
- CloudMatcher can be used for multiple workflows and flexibly combined into novel uses. (Subsection 3.7.3)

3.7.1 Entity Matching on Real-World Datasets

CloudMatcher has been developed for over 1.5 years, in a combination of Python and Java, at cloudmatcher.io. It is not yet available to the general public (we still need to work out issues such as how to let a "public" user pay easily and how to securely store his/her data).

CloudMatcher however has been applied to many datasets at UW-Madison, Johnson Controls Inc. (JCI), and WalmartLabs, and has been opened to several other users, including biomedical

Dataset	Table A	Table B	Precision (in %)	Recall (in %)	Cost (# of Questions)	Machine Time	Crowd Time	Total Time	Candidate Set |C|
People	9751	706,878	93.75 – 96.32	95.50 – 97.76	$57.6 (960)	23 m	23h 14m	23h 37m	3,013
Addresses	90,673	231,081	93.22 – 95.72	76.93 – 81.01	$72.0 (1200)	38m	36h 48m	37h 26m	1,095,414
Vendors	50,295	50,292	29.95 – 38.04	91.89 – 98.10	$69.6 (1160)	58m	30h 31m	31h 29m	6,754,287
Vendors (no Brazil)	28,152	28,149	95.44 – 97.75	88.82 – 92.41	$72.0 (1200)	22m	22h 19m	22h 41m	177,337
Drugs	446,048	440,048	99.14 – 99.63	98.45 – 99.14	User labeling (1162)	8h 40m	1h 10m	9h 50m	143,479,768

Table 3.1: The results of applying CloudMatcher to several representative datasets.

researchers in a joint project between Marshfield Clinic and UW-Madison, as well as on datasets representative of problems from the political science domain.

The EM results have been good to very good on some datasets, and not so good on some others. A close examination of these results reveal several interesting issues. To discuss them, we now describe applying CloudMatcher to four representative datasets. The first three columns of Table 3.1 describes these datasets. People identifies the same persons across two tables, given their names and addresses. The smaller of the two tables is a list of elected officials in Wisconsin, and the larger table is a list of Wisconsin residents who have signed a petition. There are political scientists who would like to know which elected officials signed this petition, which is an EM problem. Addresses attempts to match addresses of Johnson Controls customers, using data from two different ERP systems. Vendors identifies the same JCI vendors across the tables, given their names and addresses. (We will explain the dataset Vendors (no Brazil) later.) Finally, Drugs matches drug descriptions in the Marshfield-UW research team. (We do not yet have the permission to disclose EM results on matching products at WalmartLabs.) Note for these real world datasets we don't have gold matches.

The rest of Table 3.1 describes the results of applying CloudMatcher to these datasets. These columns are self-explanatory, except for the last one, which lists the size of the set of tuple pairs obtained after blocking. To compute accuracy, we took a sample of 500-1000 pairs from the output of CloudMatcher, manually labeled them, then followed the accuracy estimation procedure in [56] to estimate precision and recall (see Columns "Precision" and "Recall"). A value such as "93.75-96.32" in Column "Precision" means the precision is in that range with 0.95 confidence.

From the table, we can see that CloudMatcher achieves high accuracy on People, with precision in 93.75-96.32 range and recall in 95.5-97.76. This demonstrates CloudMatcher "at its finest". It shows how an "ordinary" user at a non-profit organization could simply just upload two tables, wait 24 hours, and pay under $60 (to the crowd), to obtain good matching results.

Unfortunately the results for the remaining datasets are less stellar. For Addresses, the precision is high (93-96%), but the recall is somewhat low (77-81%). A close examination reveals that this dataset is quite dirty. For example, addresses often contain extra strings, such as "AS AGENT FOR 105 East 17th St Associates 423 W 55th St, New York, NY, 10019". This suggests data cleaning is necessary. Further, the matching instruction for crowd workers was incomplete, so when crowd workers saw addresses that are identical except for the "P.O. Box" numbers, they did not know if those should be matched.

For Vendors, the recall is good (92-98%), but the precision is very low (30-38%). This dataset contains many vendors in Brazil, and it turned out there are serious problems with their descriptions. Specifically, many Brazilian vendors have the same address and the same last name but different first name, e.g., "LUCIANA DOS SANTOS, RUA JOAO TIBIRICA - 900, VILA ANASTACIO, SAO PAULO, BRA, 34756800" vs "FERNANDA DOS SANTOS, RUA JOAO TIBIRICA - 900, VILA ANASTACIO, SAO PAULO, BRA, 34756800" (note that a vendor can be a person representing a small business, or a large company; the above two tuples represent two small businesses). Many crowd workers declared such tuples matched. In reality they do not. An extensive examination reveals some problem with the Brazil data, namely, when two vendors were entered into the JCI system, instead of entering their real addresses, often the address of the JCI office that they were doing business with were entered. Hence two vendors with different names often end up "sharing" the same address.

Thus, the Brazil data is simply incorrect. So the JCI users removed all Brazil tuples from the two tables and applied CloudMatcher again. The results now improved significantly (see the row starting with Vendors (no Brazil)), achieving precision of 95-98% and recall of 89-92%.

Another problem with the above dataset is that some of the tuple pairs are difficult even for domain experts to match, e.g., "Juan Carlos Caldelas, Monterrey, MX" vs "Juan Carlos Espinosa Mari, Monterrey, MX".

The last dataset, Drugs, was not hard to match accuracy-wise (achieving precision of 99% and recall of 98-99%), but was hard to manage runtime and space-wise. Specifically, different runs of this dataset produced different blocking results, and these results vary drastically. In some cases the size of the blocking result was quite reasonable, in the millions (of tuple pairs). But we have also seen cases of 1-2 billions of tuple pairs. Table 3.1 shows a case in the middle, where we "only" have 143M tuple pairs after blocking. This raises the question of what CloudMatcher should do if blocking produces very large outputs. If left unmanaged, the blocking process can take a very long time, consume all available memory and disk space, and stall or crash.

3.7.2 Scaling to Support a Campus-wide Entity Matching Service

In the previous subsection, all experiments were performed sequentially and one at a time. To make CloudMatcher available as a campus service, however, we must be able to support multiple concurrent jobs by multiple users. We envision running a CloudMatcher service at the UW-Madison using resources from HTCondor as the backend for CloudMatcher. This will allow us to scale up and scale down as users come in to use the system. In these experiments, however, for ease of experimentation we are using a set of resources from AWS in place of the HTCondor resources. We have allocated a set of 10 clusters of 4 instances each, configured as described at the start of this section. For our experiments, we have implemented a basic queuing service: each job gets one cluster to use, when we are out of clusters, jobs will queue up and wait for an available cluster. This is similar to how the JupyterHub server [81] provides a secure multi-user notebook service for PyData users at universities and research labs. While there are certainly opportunities to better share cluster resources, this does give us a baseline performance number.

We compare two scenarios. The first assumes a uniform arrival rate of jobs, and the second assumes that jobs arrive in bursts. In these experiments, we used five different datasets: the "People", "Address" and "Vendors" datasets without the Brazil data, described previously, and two datasets

from the Corleone paper: restaurants, which is a very small dataset, and a citations dataset matching between DBLP and Google Scholar. For these experiments we did not use the "Drugs" dataset. For any given job in the experiments, we randomly selected one of the five datasets.

The experiments are driven by a Python script using the Selenium browser automation library. Selenium activates a web browser and simulates the mouse clicks, scrolls, and keyboard input that a human user would perform. For active learning, we have a set of "golden" data that the automation script can compare against to label pairs as "matching" or "non-matching". For each pair to be labeled, the automation script pauses for 3 seconds to simulate "think-time" for the human.

Uniform Arrival Rate. In this experiment, a job arrives at the service once every twenty minutes, and is dispatched to an available cluster. We run the experiment for 24 hours, then pause job submission, and wait until the last job has completed. The purpose of this experiment is to show that CloudMatcher could handle what we claim is a reasonable workload of EM jobs that a campus like the UW-Madison might have.

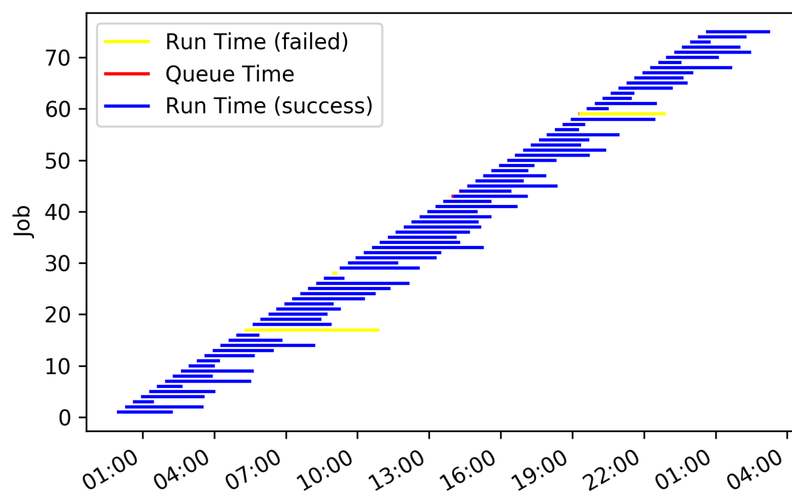

Figure 3.13: Uniform arrival rate, jobs arriving for a minimum of 24 hours

The graph is shown in Figure 3.13. The average job runtime is 2 hours, 20 minutes, and 45 seconds, though runtime is variable, with a standard deviation of 1 hour, 0 minutes and 59 seconds.

Queue time is very low - on average, just 5 seconds, which is the time it takes for the job to be distributed to a cluster and the job to become active on a cluster node. When we designed this experiment, we chose a conservative 3 jobs/hour arrival rate, and used a conservative estimate of 3 hours for an average job runtime. Using Little's Law $L = \lambda W$, where $\lambda = 3$ jobs an hour and the measured average runtime W = 2.33 hours, we expect that at any given time there are on average about 7 jobs being serviced. Given that we have enough clusters to service 10 jobs, jobs should rarely queue and the system should remain stable over time. In the experiment, there was only two times that queue time were more than a few seconds. In both cases the queued job arrived after a stretch of jobs with runtimes slightly longer than usual, and the maximum queue time was never more than five minutes.

Note that 3 jobs failed during the experiment, and several of the jobs in the next set of experiments also failed. These failures were due to known bugs in CloudMatcher that have since been fixed. Two of them are not very interesting (a multi-threaded race condition in a background file copy, and a case where we mishandled a boundary condition on sampling). The third class of failures were due to CloudMatcher enforcing a maximum size of the candidate set remaining after applying the blocking rules (we will return to discuss the motivation for this size limit in Section 3.8). For now, we mention these failures because their runtimes are included in these experiments and the experiments below. Although the individual job fails, the CloudMatcher system continues to process other jobs without issue and the entire set of jobs complete.

Bursty Arrival Rate. In this experiment, again a fixed number of jobs (K) arrive over the course of 24 hours, but arrive in "bursts". We experimented with four sets of job: K= 80, 120, 160, and 200 jobs, and we again submit all K jobs within a 24 hour window and then wait until all jobs have completed. For each experiment E_k, we fix the number of bursts B to be 8. Each burst B_i is given a random portion of the E_K jobs, and all E_K jobs are allocated to only one of the B bursts in E_K. For each burst, we pick a time for the start of the burst B_i within the 24 hours and then randomly chose a time within one hour of the start of that burst for each individual job in that burst, in effect smearing the start times of the jobs in burst out over a window rather than all jobs starting at exactly the same time.

K	**Average Total Time**	**Average Queue Time**	**Average Run Time**
80	02:58:32	00:39:46	02:18:45
120	05:48:42	03:27:00	02:21:42
160	06:23:47	04:03:59	02:19:48
200	13:56:08	11:40:10	02:15:58

Table 3.2: Total Time, Queue Time, and Run Times of the "Bursty" experiments. The time is in hours:minutes:seconds

The purpose of this experiment is to show that even in a busy and more realistic environment, where jobs do not arrive predictably, and when there may be stretches of time where amount of submitted work is greater than the immediate processing ability of the system, a user of CloudMatcher is still likely to get reasonable service from the system. Table 3.2 shows the average times jobs experience. The "Total Time" is the combined time a job spends in the queue and then running: this is the takeaway number a user sees from when they decide to run a job to when they get the results back. The "Queue Time" and "Run Time" break that number down into the two components.

Table 3.3 shows the minimum and maximum times of the same "Total", "Queue", and "Run" times. In particular, we focus on the maximum "Total Time" metric. In our experiments, the very worst performance a user would have seen was a bit over thirty hours from job submission until the results are available, which we claim is a reasonable service level for a campus Entity Matching service.

Figures 3.14 through 3.17 show a pictorial representation of the experiments. Each horizontal line is a job, and the X axis is the time in the course of the experiment. The start of each horizontal line is when the job was queued into the system, and the end of the line is when the job completed. The red portions of the line show the time when the job was queued, and the blue portion shows the job while running. A yellow line shows a job that ultimately failed.

K	Min Total Time	Max Total Time	Min Queue Time	Max Queue Time	Min Run Time	Max Run Time
80	00:44:14	10:00:36	00:00:03	04:24:26	00:44:09	09:13:31
120	00:45:58	15:32:56	00:00:05	08:09:41	00:44:24	08:46:57
160	00:46:44	15:02:14	00:00:05	11:46:18	00:43:46	07:46:08
200	00:47:09	30:34:34	00:00:05	27:34:47	00:44:08	07:58:53

Table 3.3: Min and Max Total Times, Queue Times, and Run Times of the "Bursty" experiments. The time is in hours:minutes:seconds

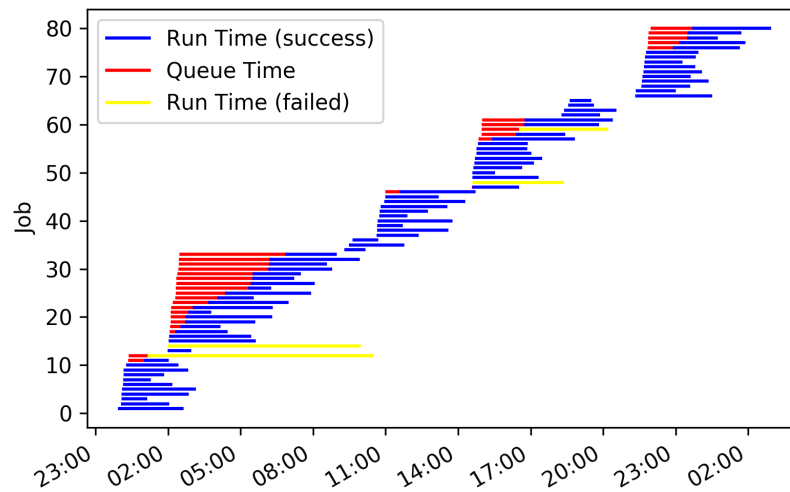

Figure 3.14: "Bursty" run, 80 submitted jobs

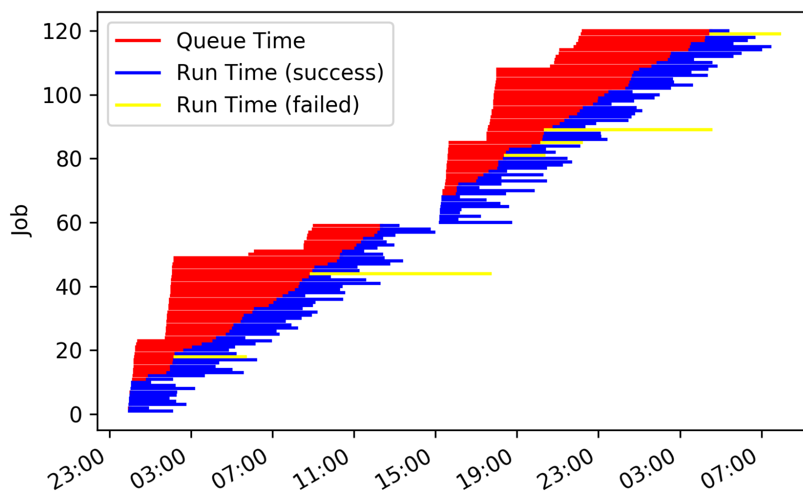

Figure 3.15: “Bursty” run, 120 submitted jobs

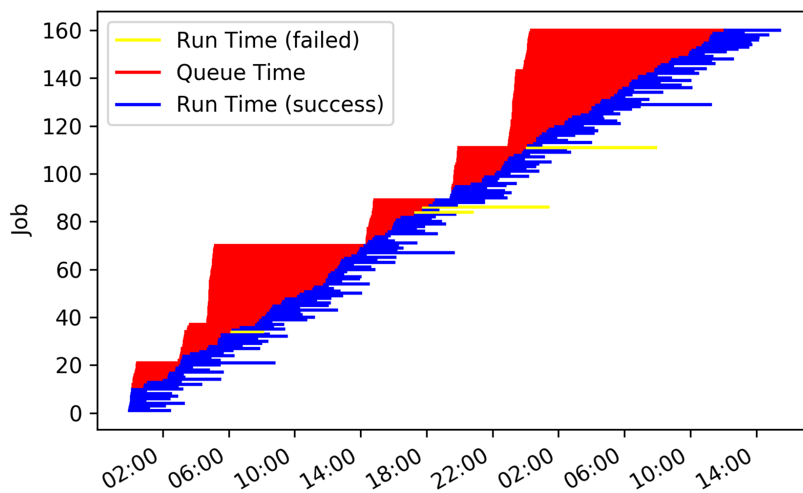

Figure 3.16: “Bursty” run, 160 submitted jobs

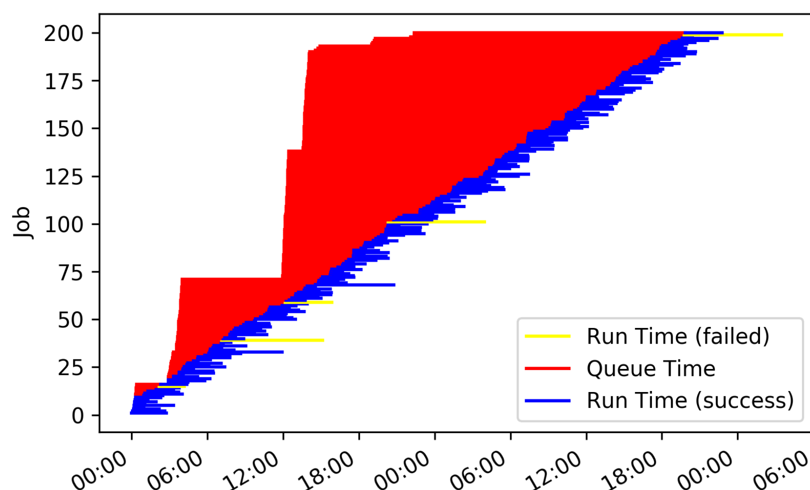

Figure 3.17: “Bursty” run, 200 submitted jobs

3.7.3 Using Basic Services in Novel Ways

We will now turn to show how users can easily customize and experiment with a wide range of EM workflows, by combining basic services provided by CloudMatcher on an easy-to-use cloud-based UI. We will show a scenario in two parts. First, though, assume a user has gone through much of the Falcon workflow - they have uploaded two tables, A and B, and converted the tables into “feature vectors” suitable for processing.

Apply Custom Blocking Rules In the typical EM flow that Falcon follows, the system helps a user determine effective blocking rules based on some training data. Suppose, however, the user suspects that they know an effective blocking rule already, and would prefer to skip the step of learning blocking rules and instead directly specify a rule. Such a rule (which we will call R_1) could be “if two items disagree on the value of the city attribute, then they will not match and should not pass through the blocking stage”.

However, just because a user believes rule R_1 is a good rule does not mean that it is indeed a good rule. For example, if many tuples have dirty values for “city”, then the rule may not be as effective as the user hopes. As a result, the user can invoke the “Evaluate Blocking Rules” service

and enter their rule R_1, as shown in Figure 3.18, and asks CloudMatcher to evaluate the rule. CloudMatcher then selects a set of tuples to be labeled, either by the user interactively or sent to the crowd (shown in Figure 3.19) to label. If R_1 indeed turns out to be a good blocking rule, then the user can invoke the "Apply Blocking Rules" basic service to apply the rule to produce a candidate set, and apply other services to learn and apply a matcher.

Figure 3.18: Defining custom blocking rules in CloudMatcher

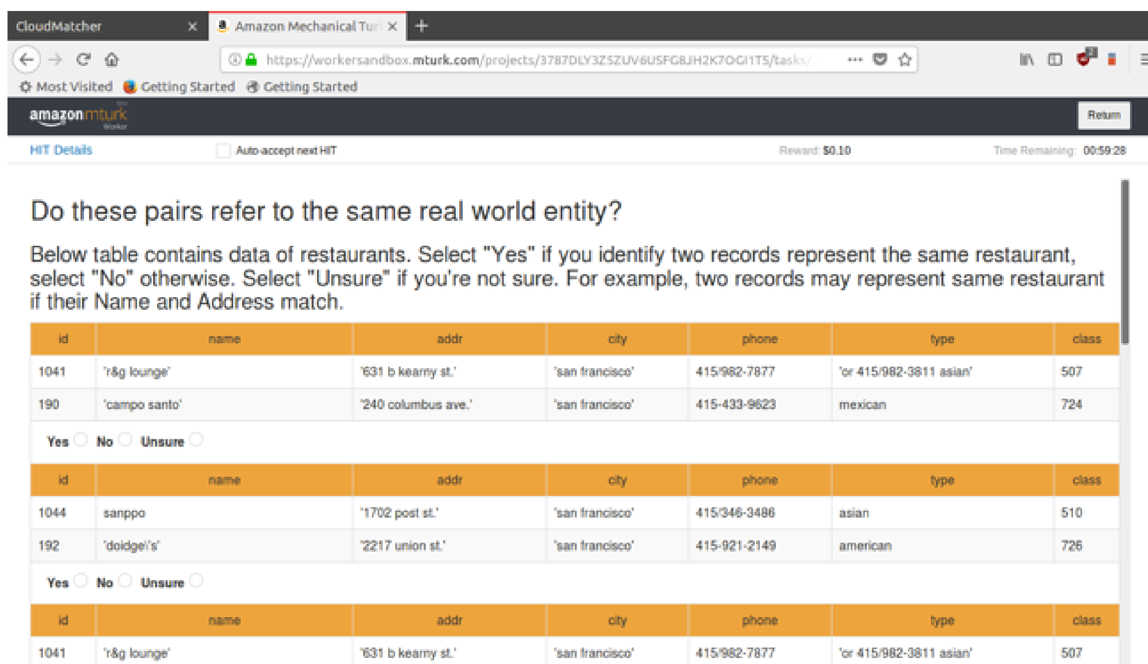

Figure 3.19: Crowdsourcing tuple labels on Amazon MTurk

Cross Validate and Apply a Matcher: To continue with this scenario, let C be the candidate set of tuple pairs obtained after applying the blocking rule R_1 to the two input tables. Earlier, executing the Falcon EM workflow, CloudMatcher would have interacted with the user or the crowd in an active learning fashion to learn a matcher M and then apply M to C. Suppose, however, the user does not want to do a full active learning cycle of labeling and training multiple times, but would prefer to simply label some examples, learn a set of models, and use k-fold cross-validation to choose a good model and then apply that model to the full dataset. This could be a faster turn-around time for an EM job by avoiding the latency of multiple active learning rounds, which can have high latency, especially when the crowd is used to label pairs.

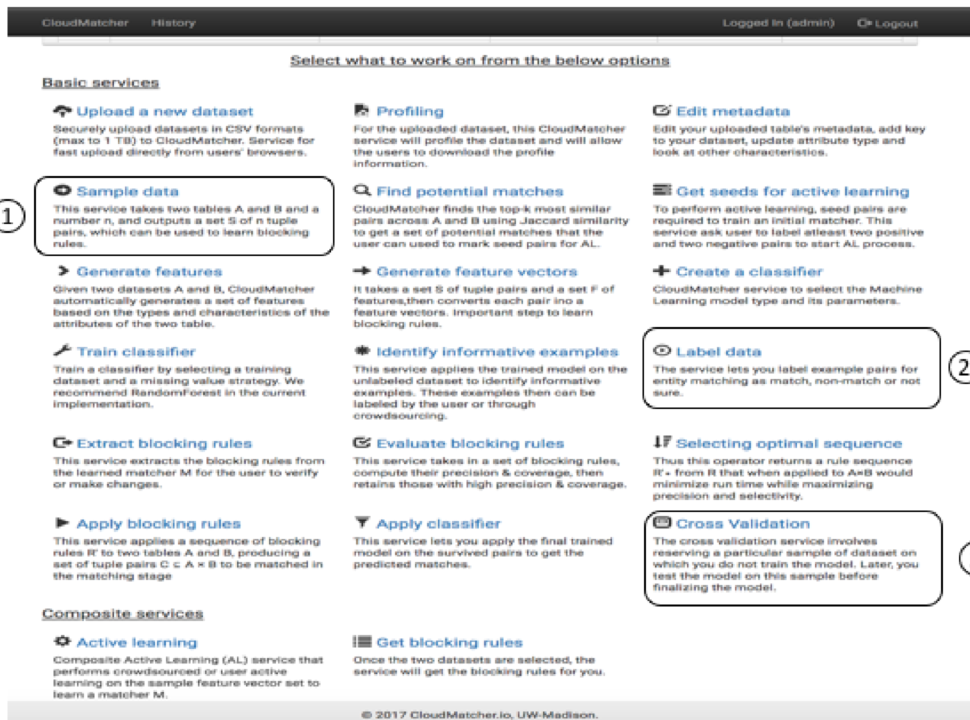

Figure 3.20: Basic services to be invoked for the cross validation workflow

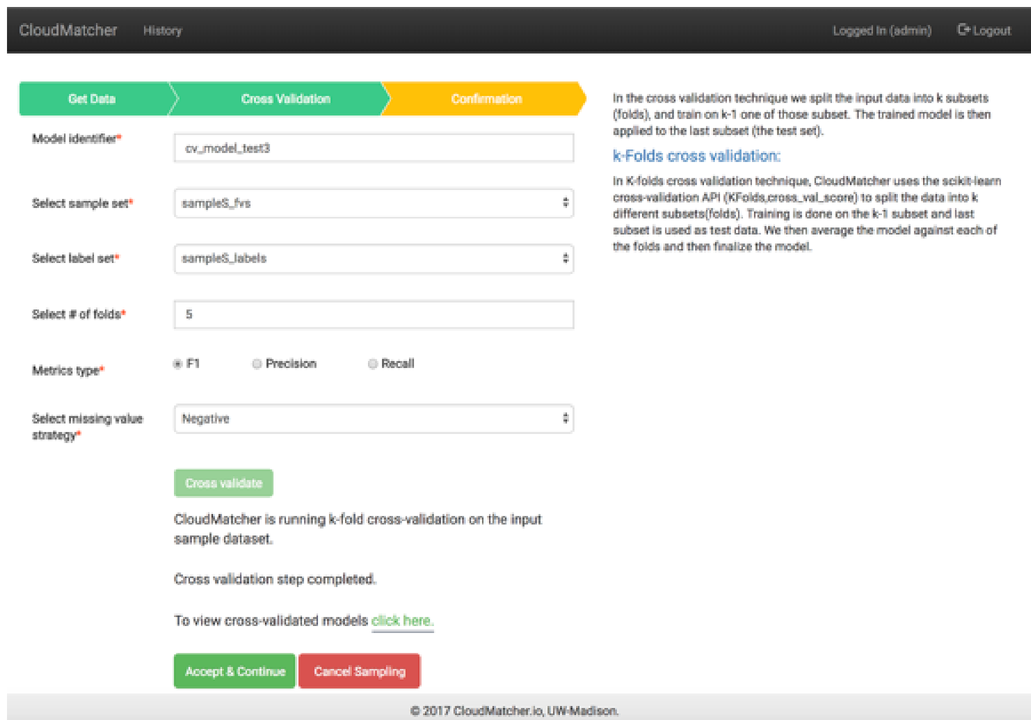

Figure 3.21: Setting parameters for the cross validation service in CloudMatcher

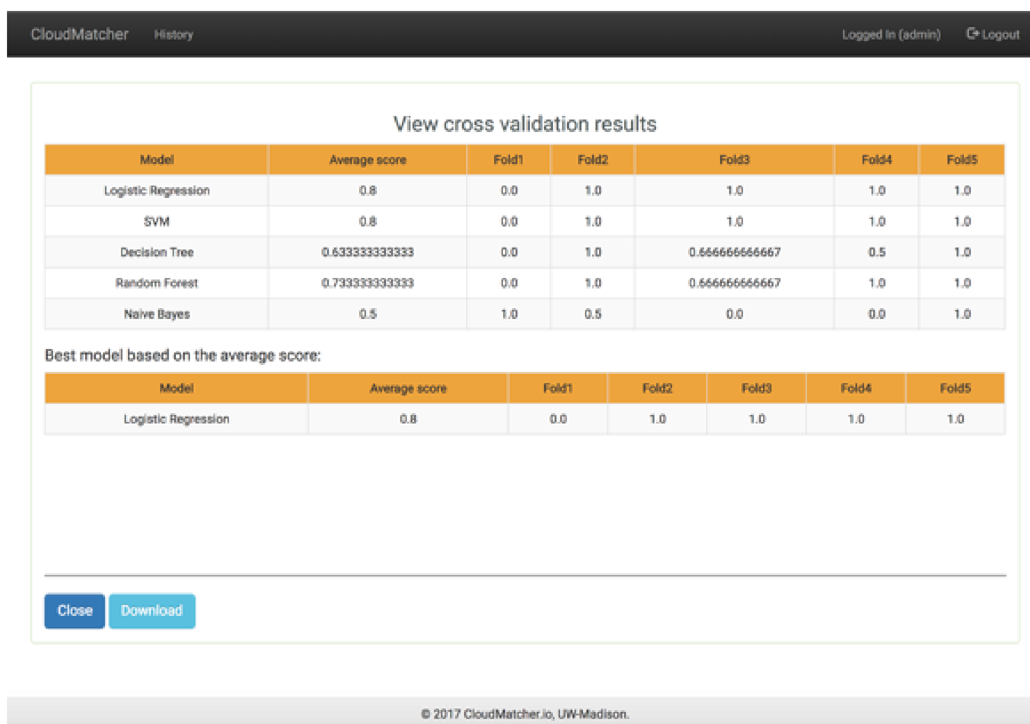

Figure 3.22: Example cross validation service results in CloudMatcher

The user can invoke a sequence of basic services, shown in Figure 3.20, to execute this workflow. First, in Step (1) of Figure 3.20, the user can create a random sample of their dataset, and then with Step (2), invoke the "Label data" service, and either label the data themselves interactively or send the data to the crowd to be labeled as shown previously in Figure 3.19. Once labeled, the user can invoke the "Cross Validation" (Figure 3.21) service highlighted in Step (3) to train and evaluate several different classes of classifiers. The "Cross Validation" service will return the most effective matcher, which can then be applied to the full dataset. (Figure 3.22)

3.8 Lessons Learned

From our experience with CloudMatcher so far, we have learned a set of interesting lessons. Some of the lessons are not so surprising. For example, users would like to have a way to estimate accuracy (e.g., precision and recall) at the end of the EM process. This is understandable and can be solved by implementing the crowdsourced accuracy estimation procedure in Corleone [56].

Other requests include ways to store EM models, data, and results, and a dashboard to monitor the EM process in real time. Other lessons that we have learned are more significant, as we describe below.

Debugging and Explaining: If there is some problem, such as low recall on Addresses, low precision on Vendors, or the blocking process taking too long on Drugs, the user is often at a loss as to why. They highly desire tools that they can use to debug or find explanations, so that they know what they can do next to improve the EM process.

Understanding Data/Problem/Solution: This is perhaps the most important lesson we learned from the CloudMatcher experience. It is clear that in many cases, the user starts out with a very limited understanding of the data, the problem, and the capabilities of the solution (which is CloudMatcher in this case). First, the user may have no idea that the data is dirty (e.g., addresses containing extra strings), that parts of the data are simply incorrect (e.g., Brazil data in Vendors), that parts of the data are so incomplete or ambiguous that even domain experts cannot match.

Second, the user may also think he/she knows the problem, i.e., the "match" definition, e.g., what it means for two tuples to match. But we have found that this is rarely the case in practice, and it can have serious consequences. For example, the user thinks he/she knows the match definition. So he/she will write an instruction to crowd workers based on that knowledge. Then crowd workers run into ambiguous cases not covered by the instruction (e.g., addresses that are the same except P.O. Box numbers in Addresses). They do not know what to do. So some will say yes, and some will say no. As a result, CloudMatcher learns incorrect blocking rules and matching models, which in turn seriously degrades the quality of the EM process. Some crowd workers may ask about such ambiguous cases in their emails. Often that is when the user realizes that the current match definition is not complete. He/she may need to revise it, then run the EM process again, incurring unnecessary time and expenses.

Finally, the user may have no idea what a tool such as CloudMatcher is capable of. Recall that CloudMatcher produces precision of 94-96% and recall of 95-98% on People dataset. Suppose

the user wants to increase precision to 99%. Can he/she still use CloudMatcher? Or would it already reach its limit and a new tool needs to be explored?

A New Way to Do EM? As a result of these observations, we do not believe practical EM can be done in "one shot". Instead, it appears to require multiple iterations. Each iteration is used to gain increasingly more knowledge about the data, the problem, and the capabilities of the tools. To do so, in each iteration we must have an arsenal of tools to help users profile, explore, and understand these three targets. We must also have a clear methodology to guide users on how to use these tools.

How to execute such multiple-iteration processes is an interesting question. Perhaps at the start, a system (such as CloudMatcher) can help the user execute EM on small data samples (selecting a variety of data samples so that the user can be exposed to all the diversity in the data, the problem, and the tool capabilities). Subsequent iterations can operate on large samples, and then eventually when the user has been satisfied, then a final EM process over the entirety of the data is launched. This is an open research question. But we believe solving it is critical for successful EM in practice.

Different Solutions for Different Parts of the Data: Another important observation is that the vast majority of current EM works treat the input data as of *uniform* quality, but in practice this is rarely the case. Instead, the data commonly contains dirty data of varying degree (e.g., as in the Addresses dataset), incorrect data (e.g., Brazil data in Vendors), and incomplete data that even domain experts cannot match (e.g., as in the Vendors dataset). It makes no sense trying to debug the system, then spending more time and money to match incorrect and incomplete data. As a result, it is important to have tools that help the user explore and understand the data (as discussed earlier), then ways to help the user "split" the data into different parts and develop different EM strategies for different parts.

User Needs Iteration with Crowd Workers: For crowd workers, we found that the most serious problem is giving them clear instructions of what it means to be a match. As discussed earlier, at the start the user often does not have a complete knowledge yet of what it means to be a match.

So he/she will give incomplete instructions to the crowd. This can have serious consequences, as discussed in that section.

As a result, we believe the problem of how to work with the crowd to arrive at the clearest instructions possible (as quickly as possible) is critical to enable practical crowdsourcing. Ultimately, the larger challenge here is that working with a crowd is not an "one-way" street. The user needs to be interacting with the crowd, possibly in multiple iterations, in order to perform EM efficiently.

More Expressive UIs: For in-house users (who label the tuple pairs), we found that they do not like the labeling UI (of seeing tuple pairs and labeling them). They find this wasteful and inefficient. To explain, observe that most current crowdsourced EM works do not consider the time it takes for a crowd worker to *understand* a tuple pair. They often focus instead on minimizing the total number of pairs that the crowd must label (as we also do in this work).

In practice, however, there is a real cost of trying to understand a tuple. For example, in the Drugs dataset, each drug description is a complex tuple that describes many ingredients. A domain expert needs 5-6 seconds to understand such a drug. Suppose this expert has just labeled drug pair (a, b), then suppose 3 minutes later he/she needs to label another pair (a, c). In this case the expert needs to spend time trying to understand a again, i.e., recognizing that this new tuple a is the same as the old tuple a. This wastes the expert's time and cause resentments. A suggestion that we have heard is to have a more expressive and efficient UI, e.g., one that shows a cluster of drugs, so that an expert needs only to try to understand a drug once, yet can still efficiently match it with many other drugs. It is interesting to explore whether more expressive UIs like this can work with crowd workers as well (e.g., on AMT), not just with in-house workers.

3.9 Conclusions

We have described CloudMatcher a cloud/crowd service for EM. We have motivated CloudMatcher then described its design and implementation. Finally, we have described its deployment

and lessons learned. These lessons point to challenges in understanding the EM problem, crowdsourcing, and general human interaction (e.g., with in-house users). We conclude that these challenges must be addressed to develop truly successful EM services, both for health informatics and for other general domains. Much more work remains to be done on CloudMatcher, but the initial results suggest the high promise of this EM-as-a-service approach.

Chapter 4

Related Work

Big Data Systems: The work in Chapter 2 was one of the opening works in studying systems for Big Data analytics. Shortly after the work was published, the authors of the original Google work appeared alongside us in a point/counterpoint pair of articles [41, 94] in the *Communications of the ACM*. In the *CACM* article [41], the authors took issue with several points raised in the benchmark around the construction of the benchmark, the functionality tested, and the MapReduce programming model vs the Hadoop implementation of the MapReduce model. The benchmark itself has been incorporated into other benchmark suites. The AMPlab used it as the basis for their package to benchmark additional systems [5, 85] and Intel incorporated the benchmark into their HiBench benchmark [65], which tested a number of scenarios, including machine learning, streaming, and graph analysis tasks. In [25], Mike Carey notes that what he terms the "CALDA" benchmark of Chapter 2 was a good start, and identifies unmet needs and future directions for Big Data benchmarks.

Optimizations suggested in Chapter 2 have since been studied in other systems. MapReduceOnline [36] studied pipelining results without an intermediate materialization step. HadoopDB [16, 21] built a parallel database on top of Hadoop, but replaced the underlying task execution system with a direct execution of SQL in local PostgreSQL databases, while still using MapReduce as the task distribution/parallelization system. Microsoft Polybase [45, 53] is also a split-execution system, but has an execution engine that can execute part of a query plan as a MapReduce job, and other parts of the plan as a SQL query in SQL Server, and dynamically decide which strategy to use at runtime. For data stored in HDFS, Polybase can store the index to the data in SQL Server,

and the SQL Server query optimizer can use the index and statistics in its planning. Floratou et al studied improved columnar formats for MapReduce jobs [50] and found that they could give jobs improved performance. Blanas et al wrote one of the first of many papers to study joins on MapReduce [23].

Hive, the canonical example of SQL on Hadoop, has incorporated a number of ideas from the database community in recent releases: an improved on-disk file format, vectorized execution of operators, and a more-informed query planner [63]; the "Tez" execution system replaced MapReduce with a set of data-flow operators [91]; and the "Live Long and Process" feature [11] leaves a running process "warm" so to avoid task startup costs for short queries. Impala [22] is another popular SQL-on-Hadoop system, which like Tez avoids MapReduce and executes queries itself over data stored in HDFS.

There are many other SQL-on-Hadoop systems, for a few surveys, see [28, 75, 49].

Spark brought a new approach to intermediate data with its "Resilient Distributed Dataset" [106] and lineage tracking to intelligently reduce the need for materialization while still having strong fault tolerance capabilities. In particular, compared to MapReduce, Spark dramatically improved the performance of iterative algorithms, which are at the heart of many machine learning problems.

To address different data types for which MapReduce and SQL were not natural processing fits, more specialized systems have been proposed. One such system is Pregrel [77], which brought the BSP parallel programming model to cluster computing. Further specialized graph processing systems that relax the rigid synchronization step of BSP, such as GraphLab [76] and GraphX [57] (the latter of which is built on top of Spark) are just two more examples for graph analytics. For fast reporting and aggregation, Dremel [80] was created, which brought faster execution for analytic queries using pre-aggregated data and a specialized query execution engine that executes queries directly without translating them to MapReduce. SciDB [95] is a specialized system for array databases, common in scientific computing.

Setting up and tuning the RDBMSs in Chapter 2 was far more difficult than running the queries. Now, cloud-hosted massively parallel processing SQL databases are available as on-demand services, for example, AWS RedShift [59] and Google BigQuery [92]. Systems like Presto [100]

and AWS Athena [1] bring queries over cloud filesystems like S3 without requiring a cluster or any data loading. AWS EMR [2], previously ElasticMapReduce, provides an on-demand parallel computing environment capable of running many different systems, including MapReduce and Spark.

Newer systems have augmented the distributed filesystem by storing structured data in the filesystem along with untyped byte arrays. Examples include Azure Data Lake Store and Analytics [88, 4] and Apache Kudu [3].

More recently, there has been a shift away from the batch processing model of MapReduce jobs and SQL queries, and renewed interest in streaming computations. The Google FlumeJava system [27] envisions computation as pipelines of MapReduce jobs, and Millwheel [17] proposes an updated take on a continuous query system. The two models are unified in DataFlow [18], which positions "batch" computations as a special case of streaming systems, where the stream is "bounded", e.g. a table is the result of a stream of `INSERT` statements bounded from time T_i to time T_j. Along with systems like Apache Flink [24] and Spark Structured Streaming [20], there is considerable enthusiasm for building systems that specify the logic once for data, regardless if the data source is a "bounded" table of data, or an "unbounded" stream.

Entity Matching: Entity Matching (EM) has received enormous attention in the past few decades [30, 48, 46, 72, 82, 31]. Prior works have addressed various EM scenarios such as matching tuples across two tables [56], matching tuples within a single table [48], matching into a knowledge base [54], matching XML data [103], etc. CloudMatcher considers matching tuples across two tables which is a very common setting in practice.

Crowdsourcing for EM and Hands-Off EM: Crowdsourced EM has received increasing attention in academia [78, 101, 102, 104, 42] and industry (e.g., CrowdFlower [7], CrowdComputing, SamaSource [13] etc.). Most of the works employ crowd to verify the predicted matches [101, 102, 42, 104]. Recent works Corleone [56] and Falcon [39] consider learning blockers and matchers using crowdsourcing. CloudMatcher builds on Falcon to deploy it as a service in the cloud. As far as we can tell, CloudMatcher is the first hands-off EM service on the cloud.

Cloud-Based EM: In the recent years, cloud-based analytics has become popular [61, 64]. However, very limited work has addressed building an EM service in the cloud. A recent effort, Dedupe [8], is a cloud-based EM service to match tuples within a single table. Specifically, it learns a matcher using active learning, then using the labeled data it learns a blocker. However, Dedupe uses only simple types of blockers and requires the user to label the tuple pairs selected using active learning. In contrast, CloudMatcher can employ crowdsourcing to label the pairs (CloudMatcher also supports user mode). As far as we can tell, CloudMatcher is the first cloud-based EM service providing support for crowdsourcing.

Building EM Systems and EM in Industry: The vast majority of work on EM has focused on developing EM *algorithms* (e.g., for blocking and matching steps), trying to improve their accuracy and minimizing their runtime [30, 48]. In the past few years, however, there has been effort towards building EM systems in academia [71, 37, 29] and industry [96]. Magellan [71] develops an end-to-end EM system built on the top of Python data ecosystem. It clearly distinguishes between the development and production stages, and provides tools to help users perform EM end to end. CloudMatcher leverages these tools to build a cloud-based service.

Recently, there has also been efforts towards building open-source ecosystems for data integration. A recent effort, BigGorilla [6], is an open-source data integration and data preparation ecosystem built on the top of Python data ecosystem, to enable data scientists to perform integration and analysis of data. We believe that such ecosystems can benefit from the services developed by CloudMatcher.

There has been relatively little published about EM in industry [68, 38, 54, 96]. [68] matches unstructured product offers to structured product records using a probabilistic approach, and [54] links tweets into a knowledge base.

Scaling EM: Most works of scaling EM focus on efficiently executing the blocking step. Prior works have addressed scaling specific blocking approaches such as sorted neighborhood blocking [70], key-based blocking [32], meta blocking [47] and so on. A recent work, Falcon [39], considers more general blocking rules (each being a Boolean expression of predicates) and develops a

MapReduce solution to efficiently execute such rules over two tables. CloudMatcher deploys the Falcon solution in the cloud.

Novel User Interfaces for Crowd Workers: Prior works have addressed designing novel user interfaces for crowd workers [105, 101, 67]. For example, [101] clusters the tuples into groups, shows the workers a group of tuples and asks them to find all duplicate tuples in the group, rather than asking the workers to label tuple pairs as match/non-match. These works are complementary to ours and CloudMatcher can benefit from them.

Data Profiling, Exploration, and Cleaning: Numerous works have addressed data profiling and exploration [15, 62, 89]. Based on our experiences with CloudMatcher, we observe that users often need to profile and explore the data during an EM task. For example, a user may profile the input tables to identify the various matching definitions, so that he/she can provide better instructions to the crowd workers. However, most current works on data profiling and exploration do not provide tools specific for EM tasks. Hence, we believe that there needs to be more effort towards developing profiling and data exploration capabilities specific for EM tasks.

Data cleaning has received enormous attention [33, 52, 73, 69, 62, 60]. Many works address EM as a part of the data cleaning workflow [60, 52]. Based on our experiences with CloudMatcher we believe that there needs to be more effort towards data cleaning solutions specific for EM tasks.

Chapter 5

Conclusion

Big Data is now pervasive. This has driven a critical need to develop novel methods to store and process data at large scale, as well as to develop new applications to use and make sense of this data. In this dissertation, we have made two contributions toward addressing this need. First, we study methods for large-scale data analysis. In particular, we compare the popular MapReduce model to parallel relational database management systems, and empirically analyze their strengths and weaknesses. The MapReduce (MR) paradigm for large-scale data analysis has received significant attention [86]. Although the basic control flow of this framework has existed in parallel relational database management systems (DBMSs) for over 20 years, some have called MR a dramatically new computing model [40, 86]. We evaluate both kinds of systems in terms of performance and development complexity. To this end, we define a collection of benchmarks that we have run on an open-source version of MR as well as on two parallel DBMSs. For each benchmark, we measure each systems performance for various degrees of parallelism on a cluster of 100 shared-nothing nodes. Our results reveal some interesting trade-offs. We speculate about the causes of the dramatic performance difference and consider implementation concepts that future systems should take from both kinds of architectures.

In the second contribution, we examine how Big Data scaling methods, such as MapReduce, can be used to build a scalable and flexible cloud-based entity matching applications, and what lessons can be learned from this effort for the future development of similar applications. Entity matching (EM) finds disparate data instances that refer to the same real-world entity. EM has been long studied and is crucial to many fields, and will become even more so in the age of Big Data and data science. However, it is still very difficult for domain scientists to use such EM

systems, especially at scale. In response, we have developed CloudMatcher, a cloud/crowd service for EM. CloudMatcher aims to be a fast, easy- to-use, scalable, and highly available EM service on the Web. As far as we can tell, no such application has been developed for EM in the data management research community. We describe CloudMatcher's development in the past two years and its deployment in the past six months, providing a detailed analysis of its performance over several representative datasets and in several scale-up experiments, and discussing lessons learned. Taken together, our contributions in this dissertation advance the topic of Big Data analytics, for both aspects of methods and applications.

There are many interesting challenges and research directions that could follow this dissertation. In what follows I discuss these directions, focusing on my recent work on CloudMatcher.

- **Further Optimizing CloudMatcher:** There are likely many low-hanging fruit optimization opportunities in CloudMatcher. For example, intermediate data is always written out in CSV file format, which is known to be slow to parse and is not space efficient. Converting intermediate data into a format such as Parquet or Feather would speed up many tasks, especially those that were already relatively slow and spend much of their time parsing data. Caching and optimizing file transfers would also increase performance. For fault tolerance, a task in a CloudMatcher workflow copies data to the local node from HDFS and stores the results back in HDFS at the end. If the output files were also in a local cache, some of this transfer cost could be avoided. To make the most effective use of the cache, the scheduler should attempt to place a task on the same node as the previous tasks.

  CloudMatcher should embrace Cloud-Native infrastructure. We currently store data in HDFS, though we run in AWS. Storing data in S3 instead of an HDFS cluster we run specifically for CloudMatcher would make managing CloudMatcher easier. For fault-tolerance and management of the CloudMatcher components, we have a mostly ad-hoc set of scripts and operating procedures to run our cluster. We should embrace Kubernetes or some similar orchestration system to make it easier to on-board new developers to CloudMatcher, and to make it easier for sites to deploy CloudMatcher.

CloudMatcher should support additional types in its schema, to create new features beyond numerical comparisons and string distance metrics. For example, if a given column was labeled as an “address”, CloudMatcher could both filter the address through an external address normalization service, as well as convert it to a geocoded point, and compute actual distance metrics between tuples. Other columns could benefit from matching with synonyms before computing pairwise comparisons. For example, a column listed as “Names” could know that “John” and “Jack” match much better than “John” and “Josh”.

- **Resource Management and Scheduling in CloudMatcher:** We have been challenged in CloudMatcher with understanding how to share resources between workflows. A large part of this is driven by not having a good understanding of what a given task will actually require for resources, especially in terms of disk space and memory usage. For now, we have set conservative limits on what we expect worst-case output would be and assumed concurrent execution would require worst-case resource usage from all tasks. We could improve on this with better profiling of the resource usage of individual tasks. For many operators, we need the equivalent of cardinality estimates for the output of the operator on a given dataset, which we could translate into resource usage estimations and feed into a planner/optimizer/scheduler component. In RDBMSs, understanding how to do this was developed over time by the community, but no such guidance exists for EM.

  The rule-based blockers such as those found in Falcon in particular seem to generate large amounts of intermediate data that they store on the local disks of their execution machines. This is data that is only used during the execution of a given Map or Reduce task and is not formally part of the MR model. The built-in Hadoop schedulers are effective at scheduling jobs that specify their memory and CPU usage, but do not consider local disk usage, as it is not typical of MR jobs. Exposing disk usage as an allocatable resource in YARN is likely important for the continued usage of MR by the rule-based blocking systems.

- **CloudMatcher Deployments Beyond Entity Matching:** CloudMatcher is designed to make it easier to deploy new services to system and to interact with existing services. Johnson Controls plans to use CloudMatcher for entity matching lists of customers to an existing "gold" set of customers, but to use it to protect the "gold" data. A user will be able to upload a table to be matched, but need not (and in fact, cannot) upload or download the 2nd table, as it will already be stored in the system. As another scenario at Johnson Controls, they have a Bayesian classification problem where they have solved the core operator, but do not have an "end-to-end" workflow. CloudMatcher implements that workflow, so by adding their classifier as a service to CloudMatcher they can deploy the full service.

  From talking with users, it seems as a popular use case for CloudMatcher will be as a collaborative labeling tool. This may be as a "first step" towards entity matching, or it could be simply as a way for a team to label a dataset or as a preparation tool for sending data to the crowd for labeling. The UIs that are developed to support this labeling should be useful to many other workflows. One of the key lessons we learned in CloudMatcher was that the user does not always know the problem that they're actually trying to solve. As we interact more and more with different users and different datasets, we can better develop "how-to" guides to help users more quickly understand their problem and develop effective ways to address it.

Bibliography

[1] Amazon Web Services Athena https://aws.amazon.com/athena/.

[2] Amazon Web Services EMR https://aws.amazon.com/emr/.

[3] Apache Kudu https://kudu.apache.org/kudu.pdf.

[4] Azure Data Lake Analytics https://docs.microsoft.com/en-us/azure/data-lake-analytics/data-lake-analytics-overview.

[5] Big Data Benchmark https://amplab.cs.berkeley.edu/benchmark/.

[6] BigGorilla http://www.biggorilla.org/.

[7] CrowdFlower https://www.crowdflower.com/.

[8] Dedupe https://dedupe.io/.

[9] Hadoop. http://hadoop.apache.org/.

[10] Hive. http://mirror.facebook.com/facebook/hive/.

[11] Hive Live Long and Process https://cwiki.apache.org/confluence/display/Hive/LLAP.

[12] Magellan project https://sites.google.com/site/anhaidgroup/projects/magellan.

[13] Samasource https://www.samasource.org/.

[14] Vertica. http://www.vertica.com/.

[15] Z. Abedjan, L. Golab, and F. Naumann. Profiling relational data: A survey. *PVLDB*, 24(4):557–581, 2015.

[16] A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi, A. Silberschatz, and A. Rasin. Hadoopdb: An architectural hybrid of mapreduce and dbms technologies for analytical workloads. *Proc. VLDB Endow.*, 2(1):922–933, Aug. 2009.

[17] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle. Millwheel: Fault-tolerant stream processing at internet scale. *Proc. VLDB Endow.*, 6(11):1033–1044, Aug. 2013.

[18] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and S. Whittle. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proc. VLDB Endow.*, 8(12):1792–1803, Aug. 2015.

[19] Y. Amir and J. Stanton. The spread wide area group communication system. Technical report, TR CNDS-98-4, The Center for Networking and Distributed Systems, The Johns Hopkins University, 1998.

[20] M. Armbrust, T. Das, J. Torres, B. Yavuz, S. Zhu, R. Xin, A. Ghodsi, I. Stoica, and M. Zahari. Structured streaming: A declarative api for realtime applications in apache spark. In *Proceedings of the 2018 ACM International Conference on Management of Data*, SIGMOD '18, New York, NY, USA, 2018. ACM.

[21] K. Bajda-Pawlikowski, D. J. Abadi, A. Silberschatz, and E. Paulson. Efficient processing of data warehousing queries in a split execution environment. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, pages 1165–1176, New York, NY, USA, 2011. ACM.

[22] M. Bittorf, T. Bobrovytsky, C. Erickson, M. G. D. Hecht, M. J. I. J. L. Kuff, D. K. A. Leblang, N. L. I. P. H. Robinson, D. R. S. Rus, J. R. D. T. S. Wanderman, and M. M. Yoder. Impala: A modern, open-source sql engine for hadoop. In *Proceedings of the 7th Biennial Conference on Innovative Data Systems Research*, 2015.

[23] S. Blanas, J. M. Patel, V. Ercegovac, J. Rao, E. J. Shekita, and Y. Tian. A comparison of join algorithms for log processing in mapreduce. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 975–986, New York, NY, USA, 2010. ACM.

[24] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.

[25] M. J. Carey. Bdms performance evaluation: practices, pitfalls, and possibilities. In *Technology Conference on Performance Evaluation and Benchmarking*, pages 108–123. Springer, 2012.

[26] R. Chaiken, B. Jenkins, P.-A. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. Scope: Easy and efficient parallel processing of massive data sets. *Proc. VLDB Endow.*, 1(2):1265–1276, Aug. 2008.

[27] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum. Flumejava: Easy, efficient data-parallel pipelines. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 363–375, New York, NY, USA, 2010. ACM.

[28] Y. Chen, X. Qin, H. Bian, J. Chen, Z. Dong, X. Du, Y. Gao, D. Liu, J. Lu, and H. Zhang. A study of sql-on-hadoop systems. In *Workshop on Big Data Benchmarks, Performance Optimization, and Emerging Hardware*, pages 154–166. Springer, 2014.

[29] P. Christen. Febrl -: An open source data cleaning, deduplication and record linkage system with a graphical user interface. In *SIGKDD*, 2008.

[30] P. Christen. *Data Matching: Concepts and Techniques for Record Linkage, Entity Resolution, and Duplicate Detection*. Springer Publishing Company, Incorporated, 2012.

[31] P. Christen. A survey of indexing techniques for scalable record linkage and deduplication. *IEEE TKDE*, 24(9):1537–1555, 2012.

[32] X. Chu, I. F. Ilyas, and P. Koutris. Distributed data deduplication. *PVLDB*, 9(11):864–875, 2016.

[33] X. Chu, I. F. Ilyas, S. Krishnan, and J. Wang. Data cleaning: Overview and emerging challenges. In *SIGMOD*, 2016.

[34] Cisco Systems. *Cisco Catalyst 3750-E Series Switches Data Sheet*, June 2008.

[35] J. Cohen, B. Dolan, M. Dunlap, J. M. Hellerstein, and C. Welton. Mad skills: New analysis practices for big data. *Proc. VLDB Endow.*, 2(2):1481–1492, Aug. 2009.

[36] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. Mapreduce online. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, NSDI'10, pages 21–21, Berkeley, CA, USA, 2010. USENIX Association.

[37] M. Dallachiesa et al. Nadeef: A commodity data cleaning system. In *SIGMOD*, 2013.

[38] N. Dalvi, R. Kumar, B. Pang, and A. Tomkins. Matching reviews to objects using a language model. In *EMNLP*, 2009.

[39] S. Das et al. Falcon: Scaling up hands-off crowdsourced entity matching to build cloud services. In *SIGMOD*, 2017.

[40] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI '04*, pages 10–10, 2004.

[41] J. Dean and S. Ghemawat. Mapreduce: A flexible data processing tool. *Commun. ACM*, 53(1):72–77, Jan. 2010.

[42] G. Demartini, D. E. Difallah, and P. Cudré-Mauroux. Zencrowd: Leveraging probabilistic reasoning and crowdsourcing techniques for large-scale entity linking. In *WWW*, 2012.

[43] D. J. DeWitt and R. H. Gerber. Multiprocessor Hash-based Join Algorithms. In *VLDB '85*, pages 151–164, 1985.

[44] D. J. DeWitt, R. H. Gerber, G. Graefe, M. L. Heytens, K. B. Kumar, and M. Muralikrishna. GAMMA - A High Performance Dataflow Database Machine. In *VLDB '86*, pages 228–237, 1986.

[45] D. J. DeWitt, A. Halverson, R. Nehme, S. Shankar, J. Aguilar-Saborit, A. Avanes, M. Flasza, and J. Gramling. Split query processing in polybase. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 1255–1266, New York, NY, USA, 2013. ACM.

[46] A. Doan, A. Halevy, and Z. Ives. *Principles of Data Integration*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2012.

[47] V. Efthymiou, G. Papadakis, G. Papastefanatos, K. Stefanidis, and T. Palpanas. Parallel meta-blocking: Realizing scalable entity resolution over large, heterogeneous data. In *Big Data*, 2015.

[48] A. K. Elmagarmid, P. G. Ipeirotis, and V. S. Verykios. Duplicate record detection: A survey. *IEEE TKDE*, 19(1):1–16, 2007.

[49] A. Floratou, U. F. Minhas, and F. Özcan. Sql-on-hadoop: Full circle back to shared-nothing database architectures. *Proc. VLDB Endow.*, 7(12):1295–1306, Aug. 2014.

[50] A. Floratou, J. M. Patel, E. J. Shekita, and S. Tata. Column-oriented storage techniques for mapreduce. *Proc. VLDB Endow.*, 4(7):419–429, Apr. 2011.

[51] S. Fushimi, M. Kitsuregawa, and H. Tanaka. An Overview of The System Software of A Parallel Relational Database Machine. In *VLDB '86*, pages 209–219, 1986.

[52] H. Galhardas et al. Ajax: An extensible data cleaning tool. In *SIGMOD*, 2000.

[53] V. R. Gankidi, N. Teletia, J. M. Patel, A. Halverson, and D. J. DeWitt. Indexing hdfs data in pdw: Splitting the data from the index. *Proc. VLDB Endow.*, 7(13):1520–1528, Aug. 2014.

[54] A. Gattani et al. Entity extraction, linking, classification, and tagging for social media: A wikipedia-based approach. *PVLDB*, 6(11):1126–1137, 2013.

[55] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43, 2003.

[56] C. Gokhale et al. Corleone: Hands-off crowdsourcing for entity matching. In *SIGMOD*, 2014.

[57] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. Graphx: Graph processing in a distributed dataflow framework. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 599–613, Berkeley, CA, USA, 2014. USENIX Association.

[58] Y. Govind, E. Paulson, M. Ashok, P. S. GC, A. Hitawala, A. Doan, Y. Park, P. L. Peissig, E. LaRose, and J. C. Badger. Cloudmatcher: A cloud/crowd service for entity matching. *BIGDAS Workshop @ KDD-17*.

[59] A. Gupta, D. Agarwal, D. Tan, J. Kulesza, R. Pathak, S. Stefani, and V. Srinivasan. Amazon redshift and the case for simpler data warehouses. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1917–1923, New York, NY, USA, 2015. ACM.

[60] D. Haas, S. Krishnan, J. Wang, M. J. Franklin, and E. Wu. Wisteria: Nurturing scalable data cleaning infrastructure. In *VLDB*, 2015.

[61] D. Halperin et al. Demonstration of the myria big data management service. In *SIGMOD*, 2014.

[62] J. Heer, J. M. Hellerstein, and S. Kandel. Predictive interaction for data transformation. In *CIDR 2015, Seventh Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2015, Online Proceedings*, 2015.

[63] Y. Huai, A. Chauhan, A. Gates, G. Hagleitner, E. N. Hanson, O. O'Malley, J. Pandey, Y. Yuan, R. Lee, and X. Zhang. Major technical advancements in apache hive. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 1235–1246, New York, NY, USA, 2014. ACM.

[64] B. Huang, S. Babu, and J. Yang. Cumulon: Optimizing statistical data analysis in the cloud. In *SIGMOD*, 2013.

[65] S. Huang, J. Huang, Y. Liu, L. Yi, and J. Dai. Hibench: A representative and comprehensive hadoop benchmark suite. In *Proc. ICDE Workshops*, 2010.

[66] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 59–72, New York, NY, USA, 2007. ACM.

[67] A. Jain, A. D. Sarma, A. G. Parameswaran, and J. Widom. Understanding workers, developing effective tasks, and enhancing marketplace dynamics: A study of a large crowdsourcing marketplace. *CoRR*, abs/1701.06207, 2017.

[68] A. Kannan, I. E. Givoni, R. Agrawal, and A. Fuxman. Matching unstructured product offers to structured product specifications. In *SIGKDD*, 2011.

[69] Z. Khayyat, I. F. Ilyas, A. Jindal, S. Madden, M. Ouzzani, P. Papotti, J.-A. Quiané-Ruiz, N. Tang, and S. Yin. Bigdansing: A system for big data cleansing. In *SIGMOD*, 2015.

[70] L. Kolb, A. Thor, and E. Rahm. Multi-pass sorted neighborhood blocking with mapreduce. *Comput. Sci.*, 27(1):45–63, 2012.

[71] P. Konda, S. Das, P. Suganthan G. C., A. Doan, A. Ardalan, J. R. Ballard, H. Li, F. Panahi, H. Zhang, J. Naughton, S. Prasad, G. Krishnan, R. Deep, and V. Raghavendra. Magellan: Toward building entity matching management systems. *Proc. VLDB Endow.*, 9(12):1197–1208, Aug. 2016.

[72] H. Köpcke and E. Rahm. Frameworks for entity matching: A comparison. *Data Knowl. Eng.*, 69(2):197–210, 2010.

[73] S. Krishnan et al. Activeclean: Interactive data cleaning for statistical modeling. *PVLDB*, 9(12):948–959, 2016.

[74] E. LaRose et al. Entity matching using Magellan: Mapping drug reference tables. In *AIMA Joint Summit*, 2017.

[75] K.-H. Lee, Y.-J. Lee, H. Choi, Y. D. Chung, and B. Moon. Parallel data processing with mapreduce: A survey. *SIGMOD Rec.*, 40(4):11–20, Jan. 2012.

[76] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed graphlab: A framework for machine learning and data mining in the cloud. *Proc. VLDB Endow.*, 5(8):716–727, Apr. 2012.

[77] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM.

[78] A. Marcus and A. Parameswaran. Crowdsourced data management: Industry and academic perspectives. *Found. Trends databases*, 6(1-2):1–161, 2015.

[79] E. Meijer, B. Beckman, and G. Bierman. LINQ: reconciling object, relations and XML in the .NET framework. In *SIGMOD '06*, pages 706–706, New York, NY, USA, 2006. ACM.

[80] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: Interactive analysis of web-scale datasets. *Commun. ACM*, 54(6):114–123, June 2011.

[81] M. Milligan. Interactive hpc gateways with jupyter and jupyterhub. In *Proceedings of the Practice and Experience in Advanced Research Computing 2017 on Sustainability, Success and Impact*, PEARC17, pages 63:1–63:4, New York, NY, USA, 2017. ACM.

[82] F. Naumann and M. Herschel. *An Introduction to Duplicate Detection*. Morgan and Claypool Publishers, 2010.

[83] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: A Not-So-Foreign Language for Data Processing. In *SIGMOD '08*, pages 1099–1110, 2008.

[84] J. Ong, D. Fogg, and M. Stonebraker. Implementation of data abstraction in the relational database system ingres. *SIGMOD Rec.*, 14(1):1–14, Sept. 1983.

[85] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B.-G. Chun. Making sense of performance in data analytics frameworks. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI'15, pages 293–307, Berkeley, CA, USA, 2015. USENIX Association.

[86] D. A. Patterson. Technical Perspective: The Data Center is the Computer. *Commun. ACM*, 51(1):105–105, 2008.

[87] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, pages 165–178, New York, NY, USA, 2009. ACM.

[88] R. Ramakrishnan, B. Sridharan, J. R. Douceur, P. Kasturi, B. Krishnamachari-Sampath, K. Krishnamoorthy, P. Li, M. Manu, S. Michaylov, R. Ramos, N. Sharman, Z. Xu, Y. Barakat, C. Douglas, R. Draves, S. S. Naidu, S. Shastry, A. Sikaria, S. Sun, and R. Venkatesan. Azure data lake store: A hyperscale distributed file service for big data analytics. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 51–63, New York, NY, USA, 2017. ACM.

[89] V. Raman and J. M. Hellerstein. Potter's wheel: An interactive data cleaning system. In *VLDB*, 2001.

[90] R. Rustin, editor. *ACM-SIGMOD Workshop on Data Description, Access and Control*, Ann Arbor, MI USA, May 1974. ACM.

[91] B. Saha, H. Shah, S. Seth, G. Vijayaraghavan, A. Murthy, and C. Curino. Apache tez: A unifying framework for modeling and building data processing applications. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1357–1369, New York, NY, USA, 2015. ACM.

[92] K. Sato. An inside look at google bigquery, white paper. *Google Inc*, 2012.

[93] M. Stonebraker. The Case for Shared Nothing. *Database Engineering*, 9:4–9, 1986.

[94] M. Stonebraker, D. Abadi, D. J. DeWitt, S. Madden, E. Paulson, A. Pavlo, and A. Rasin. Mapreduce and parallel dbmss: Friends or foes? *Commun. ACM*, 53(1):64–71, Jan. 2010.

[95] M. Stonebraker, P. Brown, D. Zhang, and J. Becla. Scidb: A database management system for applications with complex analytics. *Computing in Science & Engineering*, 15(3):54–62, 2013.

[96] M. Stonebraker et al. Data curation at scale: The data tamer system. In *CIDR*, 2013.

[97] M. Stonebraker and J. Hellerstein. What Goes Around Comes Around. In *Readings in Database Systems*, pages 2–41. The MIT Press, 4th edition, 2005.

[98] D. Thomas, D. Hansson, L. Breedt, M. Clark, J. D. Davidson, J. Gehtland, and A. Schwarz. *Agile Web Development with Rails*. Pragmatic Bookshelf, 2006.

[99] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: a warehousing solution over a map-reduce framework. *Proceedings of the VLDB Endowment*, 2(2):1626–1629, 2009.

[100] M. Traverso. Presto: Interacting with petabytes of data at facebook. *Retrieved February*, 4:2014, 2013.

[101] J. Wang, T. Kraska, M. J. Franklin, and J. Feng. Crowder: Crowdsourcing entity resolution. *PVLDB*, 5(11):1483–1494, 2012.

[102] J. Wang, G. Li, T. Kraska, M. J. Franklin, and J. Feng. Leveraging transitive relations for crowdsourced joins. In *SIGMOD*, 2013.

[103] M. Weis and F. Naumann. Detecting duplicate objects in xml documents. In *IQIS*, 2004.

[104] S. E. Whang, P. Lofgren, and H. Garcia-Molina. Question selection for crowd entity resolution. *PVLDB*, 6(6):349–360, 2013.

[105] S. E. Whang, J. McAuley, and H. Garcia-Molina. Compare me maybe: Crowd entity resolution interfaces. Technical report, Stanford University.

[106] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.