

SCALING SORTED NEIGHBORHOOD BLOCKING FOR ENTITY MATCHING

by

Jeffrey R. Ballard

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

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2019

Date of final oral examination: 7/30/2019

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

AnHai Doan, Professor, Computer Sciences

Kristin R. Eschenfelder, Professor and Director, The Information School

Paraschos Koutris, Assistant Professor, Computer Sciences

Theodoros Rekatsinas, Assistant Professor, Computer Sciences

© Copyright by Jeffrey R. Ballard 2019

All Rights Reserved

*To my parents, Joyce and Robert
and, my in-laws, Lusia and Robert
and, most of all, my wife, Jo*

ACKNOWLEDGMENTS

What you are holding¹ is the artifact of a long journey. It is almost paralyzing to try to put into words that will in any way acknowledge everyone whom I need to thank for their help in this process. I am sure I have forgotten many people. If I have forgotten you, I am really sorry. Tell me and I will buy you an appropriate beverage at your favorite establishment.

As I find myself finally at the end of the long and winding PhD road, I am afforded the opportunity to reflect on the journey. This is a story of two major forces: *passion* and *persistence*. Both forces were absolutely required to get me to this point.

Probably not surprising for someone pursuing a doctorate in Computer Sciences, computers have always played a role in my life. For as long as I can remember, my parents made computers available to me.² This was the early fuel for my passion. Over time, this passion was honed into building systems. In this dissertation you will find an experiment that included *a billion records*. Truly this was only made possible from years of building things “on the computer”, and my father provided me the early path to follow.

Persistence is the other major force that drove me to actually climb the mountains to get a doctorate. This was not learned at the end of my university education, but at the beginning. As an undergraduate, for a time, I was a member of the University of Wisconsin–Madison intercollegiate rowing team. I was by no means a particularly good rower, however I was lucky enough to receive advice from Freshman Coach Dan Gehn on the merits of “sticktoitiveness”. While crew was something I eventually drifted away from, the persistence it requires remained.

What stuck with me was computers, and as a student employee for the College of Engineering’s Computer–Aided Engineering Center (CAE) I was able to grow my skill base and experiment on much larger systems. While working for the Model Advanced Facility (MAF) within CAE, I had access to cutting-edge resources. Working in the MAF I was able to collaborate with art professor George Cramer. George was the first faculty member who earnestly told me “you should go to grad school”. George believed in me a full decade before I believed in myself.

As I did not initially listen to George, I graduated with my undergraduate degree and started to work as Academic Staff for the University of Wisconsin–Madison, eventually settling back in at CAE. About a decade later, with the encouragement of my wife, I enrolled in the Computer Sciences graduate program, intending on getting a masters degree part time. After I got the masters degree, I thought to myself “how hard could a PhD be?” (Answer: really, really, hard.)

¹The irony here is that nearly everyone will be reading the electronic version.

²And, at the time, a modem *with* an extra phone line. I suppose they actually wanted to use the phone every so often.

Pursuing a doctorate while working full time required a lot of coordination, balance, and grace from the various realms in my life: academics, work, *and* home. I have been blessed to have these all work out.

Realm One: Academics First and foremost I have had the wonderful opportunity to work with my advisor, professor AnHai Doan. It is palpably intimidating to try to put into words what AnHai has given me. He was tremendously graceful and flexible towards my study, far beyond my expectation of the accommodations a “typical” computer sciences professor would provide (and beyond what other professors have provided). The single most important concept that I have learned from AnHai is communication. Being able to logically structure an explanation to maximize accessibility by your audience is key (and surprisingly difficult to learn). Truly one of my happiest times was when I was able to fully explain something to AnHai without him getting confused. Beyond communication, AnHai provided me with a wonderful set of opportunities including teaching, consulting, and attending conferences. I cannot possibly thank him enough.

I also have learned a great deal from the other CS Database faculty, through discussions, classes, and weekly seminars. Amongst all of those, this dissertation especially benefited from my discussions with Paris Koutris. Paris provided much-needed rigor to my cost model, as well as providing keen insight to many of my ideas. Also, thank you to the other members of my committee, Krisin Eschenfelder and Theo Rekatsinas.

On the technical side, the rigorous empirical study you will find in this dissertation would not have been possible without the tremendous resources available from CloudLab [12]. To get to this point I’ve used *262,876 hours (29.98 years)* of machine time. I would not have been able to explore as far as I have without this truly wonderful resource.

Realm Two: Work The second realm has been my truly wonderful friends and colleagues at CAE. I cannot adequately express the gratitude that I have for Robert Kohlhepp, CAE Director. He has been extremely supportive of my graduate work, and his support was key to its success. At CAE the people are wonderful, and the systems to build and maintain are large and compelling to work on. Specifically, I had the pleasure to work closely with Ken Hahn, Brian Kroth, and Brian Wilt. No matter the obstacles we kept advancing CAE’s infrastructure, services were always available, and customers were happy.

Realm Three: Home The trials and tribulations of anyone getting a PhD will have effects on those around them. My wife endured many sacrifices on my path to a PhD. Yet she put up with me, especially when I have been sleep deprived, wickedly stressed, and generally awful to be around. Her love and support, and push to help me complete this process is the only reason I have the best dissertation.³ To this I am grateful beyond words.

Finally, my in-laws were incredibly supportive of my graduate studies, and I am sorry they were not able to be around to see that I finished my PhD.

³By best I mean done.

TABLE OF CONTENTS

	Page
LIST OF TABLES	vii
LIST OF FIGURES	viii
LIST OF ALGORITHMS	xi
ABSTRACT	xiii
1 Introduction	1
1.1 Entity Matching	1
1.2 Prior Sorted Neighborhood Blocking (SNB) Work and its Limitations	2
1.3 Goals and Contributions of this Dissertation	4
1.4 Roadmap of the Rest of the Dissertation	5
2 Blocking for Entity Matching	7
2.1 Basic Methods	7
2.1.1 Key-Based Blocking	7
2.1.2 Clustering-based Methods	11
2.1.3 Rules	12
2.2 Composite Blocking Methods	13
2.2.1 Combine Variations of One Blocking Method	13
2.2.2 Combine Different Blocking Methods	13
2.3 Scaling Blocking	14
3 Sorted Neighborhood Blocking: Background and Prior Work	16
3.1 The Sorted Neighborhood Blocking Algorithm	16
3.2 Existing SNB Solutions	17
3.3 MapReduce Overview	17
3.4 A simple SNB on MapReduce	21
3.5 RepSN	25

	Page
4 Analyzing Current MapReduce SNB Solutions	38
4.1 SNB on a Single Machine	38
4.2 SNB-single	40
4.2.1 Comparing RepSN with SNB-Single	43
4.2.2 Explaining RepSN Performance	44
5 Developing Better SNB Solutions on MapReduce	49
5.1 SNB-MR-Simple	49
5.1.1 SNB-MR-Simple in Detail	50
5.2 Vanilla Sorted Neighborhood Blocking using MapReduce (VSNB)	53
5.2.1 VSNB in Detail	54
5.3 Modeling Cost and Runtime	61
5.3.1 Tunable Parameters for VSNB	62
5.3.2 Cost Model	63
5.3.3 Calibration Procedure	66
5.3.4 Plan Search	66
5.4 Empirical Study	67
5.4.1 Overall Performance	68
5.4.2 Effects of the Optimizer	71
5.4.3 Sensitivity Analysis	75
6 Analyzing Current Adaptive SNB Solutions	79
6.1 Motivation for Adaptive SNB	79
6.2 Single-Machine Adaptive SNB	80
6.2.1 Duplicate Count Strategy (DCS++)	80
6.2.2 Adaptive Sorted Neighborhood Method (A-SNM)	83
6.3 MultiMR-DCS++: A MapReduce Adaptive SNB	84
6.3.1 Discussion	85
7 Developing an Adaptive SNB Solution on MapReduce (ASNB)	88
7.1 Challenges for Adaptive SNB in a Distributed Environment	88
7.1.1 Single-Pass Adaptive SNB Computation	88
7.1.2 Distribute DCS++-style Computation	91
7.2 Introducing ASNB	92
7.2.1 Overview	93
7.2.2 Detail	95

	Page
7.3 Cost Model	110
7.4 Empirical Study	112
7.4.1 Performance	112
7.4.2 Effects of the Optimizer	116
7.4.3 Sensitivity Analysis	119
8 Conclusion and Future Work	122
BIBLIOGRAPHY	124
APPENDIX A	127
TERMS AND ABBREVIATIONS	157

LIST OF TABLES

Table	Page
3.1 Running Example. BK is the first letter of the lname attribute.	28
3.2 Output by each additional TID with $w = 4$	29
3.3 Final output with $w = 4$	29
4.1 Datasets used to compare RepSN with a single machine implementation of SNB . . .	43
5.1 Source data used for the examples in this chapter.	55
5.2 Example: sliding window computation in VSNB	58
5.3 Datasets used for VSNB empirical study	68
7.1 VSNB versus ASNB strategy	111

LIST OF FIGURES

Figure	Page
1.1 An EM Example	1
1.2 Illustration of neighborhoods of size 3.	2
2.1 Survey of blocking methods	8
2.2 An Example of AE Blocking	9
2.3 An Example of SNB	9
3.1 Distributed SNB: window spans splitpoint, $w = 3$	23
3.2 Replication of tuples after processing on each node: tuples near a boundary are sent to another MapReduce job, $w = 3$	24
3.3 RepSN overview	25
3.4 RepSN: first MapReduce Job, $w = 3, r = 2$	31
3.5 KPM Construction during the second MapReduce job in RepSN: $w = 3, r = 2$	31
3.6 RepSN: second MapReduce job, $w = 3, r = 2$	32
4.1 SNB-Single: $w = 4$	47
4.2 Relative performance: RepSN vs SNB-single	48
5.1 SNB-MR-Simple: first MapReduce job	50
5.2 SNB-MR-Simple: second MapReduce job and post process task, $w = 3$	51

Figure	Page
5.3 VSNB overview	54
5.4 VSNB: Load Dataset into HDFS	56
5.5 VSNB: first MapReduce job, $w = 3, s = 2$	57
5.6 VSNB: second MapReduce job, $w = 3, s = 2$	58
5.7 VSNB: third MapReduce job, $w = 3, s = 2$	60
5.8 VSNB: copy data from HDFS	61
5.9 Relative performance of each SNB method.	69
5.10 Relative performance of each SNB method, scaling up the number of records.	70
5.11 Cross optimization. Each dataset is run with a set of optimized plans.	72
5.12 Cross optimization using double the source data. Each dataset is run with a set of optimized plans.	73
5.13 Cross optimization using triple the source data. Each dataset is run with a set of optimized plans.	73
5.14 VSNB Sensitivity Analysis: adjust probability of skew	76
5.15 VSNB Sensitivity Analysis: adjust window size	76
5.16 VSNB Sensitivity Analysis: adjust number of worker machines	78
6.1 Window expansion in DCS++: $w = 5$. The shaded records match, so the next 4 records need to be part of the window. Thus, 6 and 7 are added to window.	81
7.1 ASNB overview	93
7.2 ASNB: load data into MapReduce	95
7.3 ASNB: first MapReduce job	96
7.4 ASNB: second MapReduce job	96
7.5 ASNB: third MapReduce job	97

Figure	Page
7.6 ASNB example: MapReduce job 2, machine one	101
7.7 ASNB example: MapReduce job 2, machine two	102
7.8 ASNB: input to MapReduce job 3, prior to partitioning	104
7.9 ASNB: third MapReduce job, after partitioning and sorting	105
7.10 ASNB: MapReduce job 3, reduce task one	107
7.11 ASNB: MapReduce job 3, reduce task two	107
7.12 ASNB: output from third MapReduce job	108
7.13 ASNB: fourth MapReduce job	109
7.14 ASNB: output from fourth MapReduce job	109
7.15 Relative performance of each ASNB method	113
7.16 Relative performance of each ASNB method, scaling up the number of records	114
7.17 ASNB cross optimization. Each dataset is run with a set of optimized plans.	117
7.18 ASNB cross optimization using double the source data. Each dataset is run with a set of optimized plans.	118
7.19 ASNB cross optimization using triple the source data. Each dataset is run with a set of optimized plans.	118
7.20 ASNB Sensitivity Analysis: adjust probability of skew	120
7.21 ASNB Sensitivity Analysis: adjust window size	121
7.22 ASNB Sensitivity Analysis: adjust number of worker machines	121

LIST OF ALGORITHMS

Algorithm	Page
3.1 MapReduce Pseudocode: Load Data	18
3.2 MapReduce Pseudocode: Mappers	19
3.3 MapReduce Pseudocode: Partitioners	20
3.4 MapReduce Pseudocode: Reducers	21
3.5 Simple SNB on MapReduce Pseudocode: job 1, job _{mapfn}	22
3.6 Simple SNB on MapReduce Pseudocode: job 1, job _{partfn}	23
3.7 Simple SNB on MapReduce Pseudocode: job 1, job _{reducefn}	33
3.8 RepSN Pseudocode: job 1, job _{mapfn}	33
3.9 RepSN Pseudocode: job 1, job _{partfn}	34
3.10 RepSN Pseudocode: job 1, job _{reducefn}	34
3.11 RepSN Pseudocode: job 2, job _{mapfn}	35
3.12 RepSN Pseudocode: build_KPM	35
3.13 RepSN Pseudocode: get_partition	36
3.14 RepSN Pseudocode: needs_replication	36
3.15 RepSN Pseudocode: job 2, job _{partfn}	36
3.16 RepSN Pseudocode: job 2, job _{reducefn}	37

Algorithm	Page
4.1 Baseline SNB pseudocode	39
4.2 SNB-single pseudocode	46

ABSTRACT

Entity matching (EM) finds data pieces referring to the same real-world entity such as tuples (David Smith, UW-Madison) and (David Smith, UWisc). The ideal EM workflow would match every data piece with every other piece within a dataset. Unfortunately, that is intractable for all but the smallest datasets. Blocking is a step within EM to reduce the quadratic cost of matching within EM to a manageable amount. One popular method is Sorted Neighborhood Blocking (SNB). Many methods have been proposed to scale SNB to large datasets. However, current methods suffer from a variety of challenges that limit the scale of data that they can process. In this dissertation I propose two methods to solve these challenges.

First, I describe a novel method called Vanilla Sorted Neighborhood Blocking (VSNB). VSNB implements an efficient SNB method using MapReduce. My contribution in VSNB is to address the technical limitations found in prior MapReduce-based SNB solutions. The empirically-backed improvements in VSNB are an efficient sort for SNB in MapReduce, as well as techniques to reduce redundant work.

Second, I describe a MapReduce-based method called Adaptive Sorted Neighborhood Blocking (ASNB). Traditional SNB such as VSNB has a fixed window within which it considers matching data pieces. By extending VSNB, ASNB can adaptively extend the amount of work SNB performs, but only in the areas required. As dense areas of actually-matching tuples are encountered, ASNB will extend SNB's reach within the dataset in order not to miss matching data. Prior work has scalability challenges, which ASNB addresses.

The key contribution of my work in the MapReduce-based adaptive SNB process can be decomposed into two phases. The first phase creates partial **ASNB** results. Then, the partial results are merged to produce the final output. Extensive experiments demonstrate the scalability of the newly developed **ASNB**. Taken together, **VSNB** and **ASNB** significantly advance the state-of-the-art in scaling SNB, a major blocking technique for EM.

Chapter 1

Introduction

This dissertation studies blocking in Entity Matching (EM). EM is the problem of identifying data pieces that refer to the same real-world entity, such as tuples (David Smith, UW-Madison) and (David Smith, UWisc). This problem has been a long-standing challenge and will become even more important for big data and data science [5, 9, 10, 13, 34].

I begin this chapter with a definition of EM and blocking within EM in Section 1.1. Then, I cover prior work in Sorted Neighborhood Blocking (SNB) in Section 1.2. Next, in Section 1.3, I will describe the goals of this dissertation. Finally, I describe the layout of the rest of this dissertation in Section 1.4.

1.1 Entity Matching

Entity Matching (EM) is the task of finding data pieces that refer to the same real-world entity. For example, Figure 1.1 shows an example of a set of data where there are redundant data pieces. Figure 1.1 depicts a *table*, where a table is composed of a set of *tuples* (also referred to as *records*). Each tuple has a unique tuple ID (TID).

ID	Surname	Age	First Name	...
3	Cooper	63	Evans	...
5	Evans	33	Cooper	...
1	Armstrong	42	Betty	...
6	Fvans	33	Cooper	...
2	Betty Armstrong	42		...
4	David	45	Susan	...

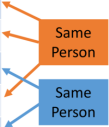


Figure 1.1: An EM Example

Many EM scenarios have been considered, however in this dissertation I will only consider the scenario of EM within a single table. Often EM within a single table is referred to as *deduplication*. The goal of deduplication is to find tuple pairs within a table that refer to the same real-world entity. When there is no ambiguity, I will use the term *dataset* to refer to a single table of tuples.

Nearly every large-scale EM workflow contains a blocking step to reduce the number of tuple pairs to be matched. Deduplicating a dataset that consists of just 100,000 tuples each will result in nearly *ten billion* comparisons without using blocking. For most datasets, the resulting quantity of comparisons will be very expensive to compute, wasteful as the vast majority of comparisons are for obviously non-matching tuple pairs, and highly impractical for contemporary datasets in the big data age. Many blocking techniques have been developed (see Chapter 2 for a survey).

1.2 Prior Sorted Neighborhood Blocking (SNB) Work and its Limitations

One popular blocking method is Sorted Neighborhood Blocking (SNB). The key idea behind SNB is that for each tuple in the dataset, use attribute values from that tuple to create a *key*. Then, when the dataset is sorted on that key, likely-matching tuples will be near each other.

Therefore, conceptually, SNB uses a Blocking Key (BK) function to produce a Blocking Key Value (BKV) for each tuple t . Then, SNB sorts the tuples based upon that value. Next, for every tuple t in sorted order, SNB defines a neighborhood of size w for t , where the neighborhood will consist of all tuples within w positions before or after t . In other words, in the sorted list of tuples $(t_1, t_2, t_3, \dots, t_n)$, tuple t_i at position i and t_j at position j are in the same neighborhood if $|i - j| < w$. Finally, SNB outputs all pairs (t, u) where u is in the neighborhood of t .

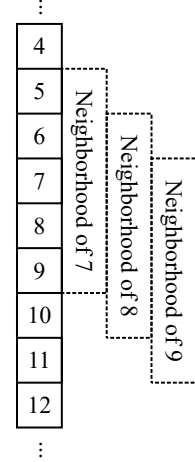


Figure 1.2: Illustration of neighborhoods of size 3.

For example, once again consider Figure 1.1. This dataset has tuples with various attributes, including Surname, Firstname, Age and others not shown. One potential BK could be age, or surname, or any arbitrary function of the attributes. If age was chosen, the dataset would be sorted on age, and SNB would consider tuples within w positions of each other in that sorted order.

Despite SNB’s popularity as a blocking method, there are several limitations.

Not Studied at Scale SNB has not yet been studied at big data scale. In fact, many research papers use relatively small quantities of source data with their experiments. For example, over the last decade, papers examining blocking have used 89,070 records [1], 200,000 records [7], 100,000 records [14], 64,263 records [18], 114,000 records [22], 250,000 records [33], and 861,567 records [35]. The largest datasets (used by some papers) did go over 1 million tuples: 2 million records [3], 1 million records [2], 1 million records [11], 1.4 million records [20, 21], 3.4 million records [24], and 2 million records [31]. However, in the age of big data, many real-world datasets deal with tens or hundreds of millions of tuples, so clearly there is a need to study how blocking algorithms perform at such scale.

Current Solutions do not Scale As far as I can tell the most advanced work to scale SNB is RepSN [21]. RepSN uses MapReduce over a cluster of machines. However, RepSN has two problems. Surprisingly, when attempting to scale RepSN, severe performance problems were encountered. First, RepSN uses map tasks to perform redundant work, which is only efficient for small datasets. Second, RepSN inefficiently implements the sorting and replication needed for SNB.

In Chapter 4, I show that on a large number of tuples (in the hundreds of millions), RepSN, which uses a cluster of machines, actually performs *worse* than a relatively simple single-machine based SNB solution. This is unexpected, and I examine the reasons behind this phenomenon.

Adaptive Neighborhoods not Demonstrated at Scale Traditional (or, vanilla) SNB uses a fixed neighborhood size to cutoff comparisons between tuples. Unfortunately, in areas of densely-matching tuples, this causes SNB to miss actually-matching tuple pairs. Setting of the neighborhood size is therefore of great importance. However, setting the neighborhood size too large

will cause many non-matching tuple pairs to be compared. Therefore, it is desirable to adaptively enlarge the neighborhood size only in areas that warrant a larger neighborhood. This is referred to as *adaptive SNB*, and several solutions have been proposed.

One set of adaptive SNB solutions have been developed for a single machine. I will describe them further in Section 6.2, however there is a need to scale up adaptive SNB to clusters of machines. As I describe in Section 6.3, a method called Multi-pass MapReduce Duplicate Count Strategy (MultiMR-DCS++) has been proposed in [27]. Unfortunately, MultiMR-DCS++ skips an important step: establishing an actually-sorted list of tuples. While it is likely that MultiMR-DCS++ will return answers, they are unlikely to be accurate, and MultiMR-DCS++ will not actually scale up to a cluster of machines. As I will examine this further in (Section 6.3), this underscores the difficulty faced in this area.

1.3 Goals and Contributions of this Dissertation

In this dissertation, I address the three limitations discussed in Section 1.2.

Study SNB at Scale I use several datasets in this dissertation to study SNB at scale. I use five datasets ranging from the smallest of 45 thousand tuples (233 megabytes), to the largest of 36.4 million tuples (168 gigabytes). This is the largest dataset that has been used to study scaling blocking methods in published literature. Furthermore, I also synthetically extend the datasets in order to examine performance on even larger datasets of sizes up to *one billion* tuples.

Using these large datasets reveals issues with current SNB solutions. I address these limitations in Chapter 4.

New Scalable MapReduce SNB Algorithm Based on my analysis of current SNB methods in Chapter 4, I propose an improved MapReduce-based SNB solution (Chapter 5). This method is called Vanilla Sorted Neighborhood Blocking (VSNB). VSNB is inspired by typical MapReduce sorting algorithms. Furthermore VSNB uses a sophisticated cost model to effectively adjust parameters for efficient execution in a MapReduce cluster.

New Scalable MapReduce Adaptive SNB Algorithm Adding adaptivity to SNB is a desirable feature that has yet to be successfully demonstrated on a large scale. I propose a new adaptive SNB blocking technique called Adaptive Sorted Neighborhood Blocking (ASNB) in Chapter 7.

Contributions In summary, this dissertation makes the following contributions:

- Demonstrates the best current SNB method, called RepSN, does not scale to large datasets. In fact, for very large datasets RepSN takes *longer* single machine methods to compute SNB. I show that this slowdown is due to RepSN’s skew mitigation called the Key Partition Matrix (KPM). The KPM is very expensive, and the wall-clock time to generate the KPM *increases* as the dataset grows.
- Develop Vanilla Sorted Neighborhood Blocking (VSNB). With VSNB the key technical innovation is, instead of an expensive KPM to handle skew, VSNB uses a long-established probabilistic sampling method, with a sophisticated cost model, to sort the dataset.
- Develop Adaptive Sorted Neighborhood Blocking (ASNB). With ASNB the key technical innovation is extending VSNB to compute distributed adaptive SNB efficiently. To compute adaptive SNB effectively, ASNB first computes adaptive SNB within each partition across the cluster. Then, ASNB uses a sophisticated method to merge the adaptive SNB results that span nodes, saving considerable time.
- Experiment with datasets from 45 thousand to 1 billion tuples. These experiments demonstrate the promise of VSNB and ASNB. As far as I can tell this is the largest scale datasets for SNB ever published.

1.4 Roadmap of the Rest of the Dissertation

The rest of this dissertation is as follows. In Chapter 2 I conduct a survey of blocking methods, examining the literature describing blocking methods and focusing on scaling.

In Chapter 3 I provide an overview of the MapReduce framework, relevant examples of SNB in the literature, and a description of RepSN (the current state-of-the-art MapReduce-based SNB

solution). In Chapter 4 I analyze RepSN, starting from a simple baseline. Next, in Chapter 5 I introduce an improved MapReduce solution called VSNB.

As I described, adding adaptivity in SNB is desirable. In Chapter 6 I describe current adaptive SNB approaches, and their limitations. Then in Chapter 7 I introduce a novel Adaptive Sorted Neighborhood Blocking (ASNB) method that uses MapReduce to scale.

Finally, I discuss future work and conclude this dissertation in Chapter 8.

Chapter 2

Blocking for Entity Matching

As introduced in Chapter 1, blocking is part of an EM pipeline. The high level goal of blocking is to reduce the number of tuple pairs for consideration. This allows for faster runtime and more expensive matching functions.

In this chapter I briefly survey blocking methods. For other surveys see [5, 6, 10, 13, 34]. For the rest of this chapter I will group blocking methods and techniques into three major focus areas as illustrated in Figure 2.1. The first focus is on the distinct blocking methods in Section 2.1. Individual blocking methods are often combined with other blocking methods, as I explore in Section 2.2. Finally, in Section 2.3, I survey scaling blocking methods.

2.1 Basic Methods

In this section I will describe the different major blocking techniques. The techniques are broken into three different categories: key-based blocking techniques, clustering techniques, and rules.

2.1.1 Key-Based Blocking

The idea behind key-based blocking algorithms is that the algorithm will first use a BK function to create a BKV for each tuple and then an algorithm-specific process will compare the BKVs. BK functions have many options to create keys. The simplest option to create keys is to directly use attribute values (see Attribute Equivalence, below). An example would be to use `zip` code as the key. Linguistic encodings have been popular for quite some time [28, 29], contemporary linguistic

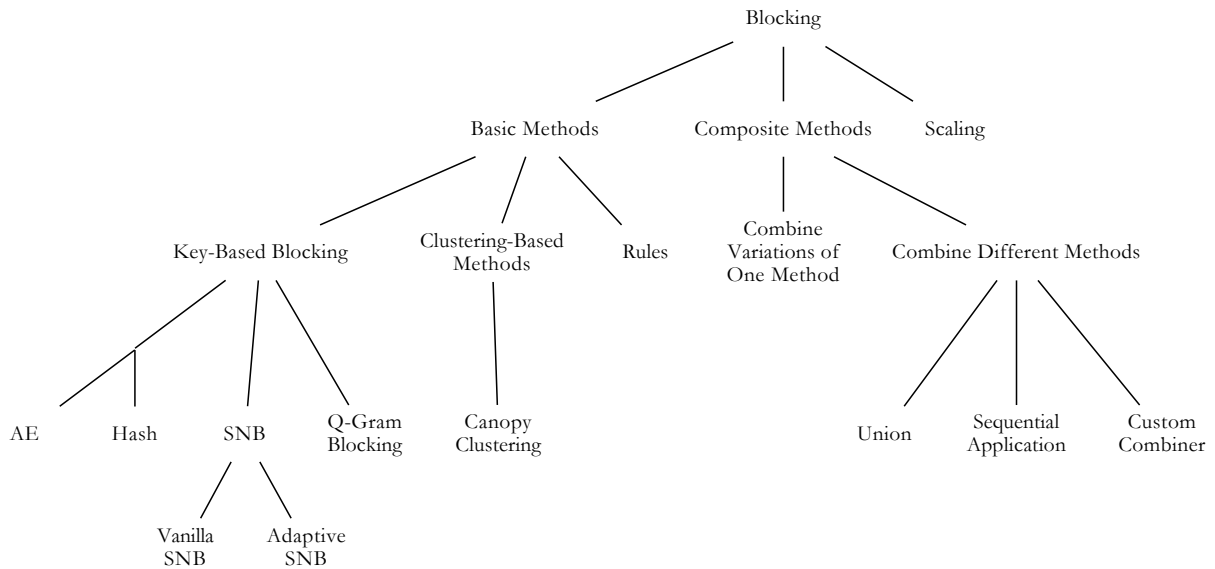


Figure 2.1: Survey of blocking methods

encodings include Soundex, Double Metaphone, NYIIS, and others. For example, consider the following commonly mistaken surnames encoded in Double Metaphone: Ballard, Pollard, and Balard. All three will receive the encoding PLRT.

Attribute Equivalence/Hashing: Attribute Equivalence (AE) is one of the oldest (and simplest) blocking methods [29]. The basic idea is similar to a database join: if specified attributes from two tuples are equal, then the tuple pair is a candidate for matching. Typically AE is implemented as a *hash* (hence it is commonly referred to as a hash join).

A hash join will return all tuple pairs where the hashed output of specified attributes is equal. The implementation uses a set of hash buckets, and each tuple is placed into a bucket based on the hash value of its specified attributes. Obviously, a sufficient number of hash buckets is required to reduce wasted computation from hash collisions.

Implementing hash based blocking is straightforward. For each tuple in the dataset, use the BK function to create a BKV for this tuple. Then, hash the BKV to place the tuple in a bucket corresponding to this hash value. Do this for all tuples. Then, for each bucket, make a tuple pair

from each tuple with every other tuple in the bucket. Output the tuple pairs as candidates for matching.

Sorted Neighborhood Blocking: The basic idea behind SNB is with an appropriate BK function, actually-matching tuples will have BKVs that are close to each other in sorted order.

Introduced in the 1990's [16], SNB is a popular blocking method today (described in detail in Chapters 3 through 5). SNB uses a BK function that produces values that bring matching tuples near each other when sorted. Thus, keys that are *close* together in sorted order are candidates for matching.

Many real-world identifiers offer such properties. For example, attributes that are typically associated with people include: zip code, area code, Soundex encoding of surname, and many others.

SNB uses a neighborhood size to determine whether two tuples are near each other. Typically computation of the neighborhood is implemented as a sliding window over the tuples sorted on their key. The sliding window creates an advantage: you know in advance the size of the output. The size of the output is a linear function of the number of tuples in the dataset and the size of the sliding window.

SNB's fixed output size comes with a downside, though. A common problem for SNB is having too small of a neighborhood size. For example, consider the case where the neighborhood size is 10, and there are 11 tuples that share a key value – some tuples with the same key value will not be returned. To solve this, [11] introduced an extension to SNB to dynamically adapt the window

Example of Blocking

Source Table				Blocks by Zip Code			
ID	Name	Zip Code	(other attrs)	ID	Name	Zip Code	(other attrs)
1	Tom	53706	...	1	Tom	53706	...
2	Sue	54111	...	3	Tommy	53706	...
3	Tommy	53706	...	6	Tammy	53706	...
4	Jeff	53711	...	ID	Name	Zip Code	(other attrs)
5	Geoff	53711	...	4	Jeff	53711	...
6	Tammy	53706	...	5	Geoff	53711	...
Cross product: (15 pairs)				2	Sue	54111	...
1-2, 1-3, 1-4, 1-5, 1-6, 2-3, 2-4, 2-5, 2-6, 3-4, 3-5, 3-6, 4-5, 4-6, 5-6				Candidate pairs for matching: (4 pairs)			
				1-3, 1-6, 3-6, 4-5			

3

Figure 2.2: An Example of AE Blocking

SNB Example

ID	Surname	Age	...	Key	ID	Key	ID	
3	Cooper	63	...	C	3	A	1	1-2
5	Evans	33	...	E	5	B	2	1-3
1	Armstrong	42	...	A	1	C	3	2-3
6	Fvans	33	...	F	6	D	4	2-4
2	Betty Armstrong	42	...	B	2	E	5	3-4
4	David	45	...	D	4	F	6	3-5
Table				Apply blocking key function		Sort		4-5
								4-6
								5-6

5

Figure 2.3: An Example of SNB

size if the density of actually-matching tuple pairs within the windows is over a threshold, which I refer to as adaptive SNB. Adaptive SNB strategies are explored in Chapters 6 and 7.

Q-Gram Blocking: The basic idea behind q-gram blocking is that actually-matching tuples will have BKVs that are approximate string matches. Unlike equality comparisons with AE/hash, and key range comparisons with SNB, q-grams use an overlap threshold to determine if two BKV are candidates for matching.

- **Febri:** Febri [4, 7], is a q-gram blocking algorithm where actually-matching tuple pairs will share a common variation of the q-grams in their respective BKVs.

The main goal of Febri is to find tuple pairs where each pair's key values split into q-grams intersect by a threshold t , where t where $0 < t < 1$.

Febri implements q-gram blocking as follows. For each tuple T in the dataset, Febri computes a key K , and splits that key into a set of q-grams Q . Febri then computes all possible combinations c of the q-grams in Q , where each c is of length $|Q| \times t$. Then, for each q-gram combination in c , Febri inserts T into a c -specific block (or bucket). After all tuples have been inserted into blocks, for each block, Febri returns pairs where each pair is in a common block.

For example, consider a sample key value of `boggle`. Using bigrams (q-grams of length 2), the `boggle` is comprised of 5 bigrams: `bo`, `og`, `gg`, `gl`, `le`. If t is 0.8, then the desired length of the bigram lists would be $5 \times 0.8 = 4$. So, for each variation from the resulting bigrams of length 4 ((`bo`, `og`, `gg`, `gl`), (`bo`, `og`, `gg`, `le`), (`bo`, `og`, `gl`, `le`), and so forth), Febri inserts this tuple into a block for each variation. Then, after all tuples have been inserted into blocks, then for each block, return all pairs of tuples within that block for matching.

- **DDEBIT:** The key intuition behind DDEBIT [14], is similar to Febri, except instead of materializing all possible q-gram variations, DDEBIT uses an inverted index. DDEBIT finds the set of q-grams from each BKV. Then DDEBIT builds an inverted index based upon the

q-grams, inserting the ID of this record onto the posting list for each q-gram. Then, using the posting lists, DDEBIT aggregates the posting lists to quickly find all other BKVs that overlap by at least the desired threshold.

- **Multiblock:** Similarly, Multiblock [17] also uses positional q-grams. Multiblock employs a multidimensional index to provide for blocking.
- **Q-Gram Fingerprinting:** A similar method, referred to as Q-Gram Fingerprinting [3], similarity between tuples is computed assigning q-grams to bloom filters to find similar keys.

First, for each attribute in the schema for the tuples to be compared, a per-attribute set of hash functions k_a are selected. Each attribute has the same number of hash functions.

Once for each dataset the following algorithm is used to build the indexes. For each tuple t in the dataset, a bloom filter of a predetermined size is created. Next, for each attribute a in t , Q-Gram Fingerprinting makes a set of q-grams q . Then, for each q , Q-Gram Fingerprinting calls a each hash function in k_a . Q-Gram Fingerprinting then inserts the output from each $k_a(q\text{-gram})$ into the bloom filter for this tuple. When all q-grams for this attribute are inserted, then Q-Gram Fingerprinting moves to the next attribute. When all attributes have finished, the bloom-filter-based fingerprint for this tuple is complete. This continues for all tuples in the dataset.

The next phase is to query the fingerprints to find fingerprints that overlap. The intuition is that the overlap between q-grams of two tuples correspond to the number of matching 1 bits in the respective bloom filters from each tuple.

2.1.2 Clustering-based Methods

The basic idea of clustering methods is that two matching tuples are likely to be in the same cluster. The main algorithm in this area is Canopy Clustering [25]. Canopies are created using an approximate distance measurement between tuples to place each tuple into one or more overlapping clusters.

At a high level, Canopy Clustering has two steps. First, using a cheap distance function, assign tuples to overlapping subsets called *canopies*. Then, use a more rigorous distance function for tuples that share at least one canopy.

Canopy Clustering uses two distance thresholds: T_1 and T_2 where $T_1 > T_2$. To create the canopies, start by adding all tuples to a pool of available tuples p .

While there are still tuples remaining in p , do the following. First, pick any tuple a at random from p and create a new canopy c_a . Find all tuples using the cheap distance function where the distance is within T_1 . Add those tuples to c_a . Tuples that are within T_2 are very close to each other, so they should not be allowed to be placed into another canopy. Since they cannot be placed into another canopy, remove points within T_2 distance from p . Repeat this whole process until p is empty.

Often the cheap distance function is an inverted index. The inverted index allows for extremely fast methods to approximately measure distance (similar to DDEBIT, above).

In [6] an extension to Canopy clustering is proposed, where instead of using threshold values, nearest neighbor values are used. Thus, if it is desired to have a particular number of tuples per cluster, start with the nearest neighbor and keep adding until the cluster is full. This has the same downside as sorted neighborhood blocking, insofar as if the number of key values that are equal are larger than the desired size of the cluster, then some true matches will not be returned as candidates.

2.1.3 Rules

The basic idea behind rules-based blocking is that in many domains, what constitutes a match is expressed by simple rules. Rules-based blocking directly uses those rules to determine if tuples are candidates for matching.

Systems such as Magellan [23] offer the ability to write arbitrary blocking rules, often in combination with other methods (see next section). Often additional rules are added until all remaining tuple pairs fit in memory [15]. Additionally, rules are human-accessible and understandable to most analysts. Some examples of rules are:

- **Band Join:** e.g. Prices must be within 25% of each other.

- **String Similarity:** e.g. Titles must match within a specific edit distance.
- **DNF Rules:** DNF (Disjunctive Normal Form) rules are often implemented as decision trees or random forests, often as the output from data mining or crowdsourced learning [15].
- **Arbitrary Rules:** e.g. the second to the last digit in two product codes must match.

2.2 Composite Blocking Methods

The basic idea behind composite methods is one individual blocking method might not be sufficient to find the desired number of matches. Composite methods fall into two groups: methods that combine the same blocking method, and methods that combine different methods.

2.2.1 Combine Variations of One Blocking Method

Many blocking methods offer the ability to union the output from multiple variations of a single method in one pass (for example [21]). Typically this is done such such that the input data is not read multiple times.

For example, in SNB, an analyst may want to combine the output of blocking based upon zipcode, combined with the blocking based upon age, and combined with the blocking based upon salary. The entire algorithm could be run multiple times, however, that is wasteful. Ideally you should be able to run these all in one pass over the input data.

2.2.2 Combine Different Blocking Methods

Combining different blocking methods is typical in EM workflows such as Magellan [23]. There are three different ways that blocking methods can be combined.

First, multiple blocking methods with lower than desired recall can be combined through a *union* of the blocking methods to provide adequate coverage of actually-matching pairs. This will increase the number of pairs returned. Also, it will likely contain duplicates, although this can be cleaned up with a relatively simple sort and deduplicate.

The second way blocking methods can be combined is through sequential application, to further reduce the number of pairs. Sequential application composite methods take the output from one blocking function and use that as input to a subsequent blocking function.

The third way blocking methods can be combined is through custom combiners. Custom combiners, like the name suggests, use custom functions to determine whether tuple pairs should be kept or discarded. This might be through looking at the output of blocking functions to determine if tuple pairs are likely to be actually-matching, or through other functions. One example is Meta-blocking [33]. Meta-blocking seeks to look at the results of other blocking methods to further prune the candidate set of matching pairs. Blocking algorithms are split into three groups.

Redundancy-positive blocking algorithms are ones where the more blocks a candidate pair is located, the more likely the pair is to be a match. Key-based blocking methods are redundancy-positive, insofar as each use some similarity to discern candidates, and as such, the more times a pair is seen, the more likely it is to be a match.

Redundancy-negative blocking methods are ones where the fewer blocks a candidate pair is located, the more likely the pair is to be a match. Canopy clustering, for example, is redundancy-negative.

Redundancy-neutral blocking methods are ones where the number of blocks that a candidate pair is located does not provide any guidance as to whether it is actually-matching. Methods like SNB have the same number of blocks per run, and unfortunately that then may have many pairs showing up many times. This results in limited ability to reason about whether the pairs are actually matching.

2.3 Scaling Blocking

There are techniques to scale that are common across most methods. First, I will begin by looking at common scaling methods for implementing blocking on a single machine and on MapReduce. Then, I will describe some common optimizations.

Single Machine On a single machine, reasonable steps must be taken to use all the available CPU cores and memory. Disk access patterns are important, therefore typical database-style optimizations are needed depending upon the storage technology used (disk, flash, or something else).

MapReduce In MapReduce there are a number of features to be aware of. First, be careful with map-side computation that is not a direct function of the input records (for example, computing an index prior to processing input records). Since MapReduce will always start at least one map task for each HDFS block, if the number of blocks is greater than the number of slots that MapReduce can run them, the result will be *waves* of map tasks. Therefore, be careful with computation performed in the map task that is not per input record (see my analysis of RepSN in Chapter 4 for an example).

Optimizations: There are several optimizations that have been proposed.

When combining variations of a single method, or multiple different methods, often redundant computation becomes a problem. Some work has been published to address this issue and reduce the wasteful computation [22, 32].

When distributing computation across a cluster of machines, one of the common areas of concern is skew. Skew is the difference between the amount of work one node of a cluster has to do versus the average amount of work each node in the cluster has to do. Higher skew will result in slower execution. RepSN attempted to address skew avoidance [20], however, I provide an improved solution in Chapter 5.

Finally, not all optimizations are ones that are beneficial. Take care to make sure that the actual gains from optimizations outweigh a simple solution. [26] An example of this is how a single-machine solution out-scaled RepSN, as discussed in Chapter 4.

Chapter 3

Sorted Neighborhood Blocking: Background and Prior Work

So far I have motivated my work on blocking methods (Chapter 1) and I have surveyed the literature for blocking methods (Chapter 2). Over the next few chapters, I will examine a popular blocking method called Sorted Neighborhood Blocking (SNB).

In this chapter, I will begin by examining SNB in general. Then, I will give an overview of SNB in the literature, and provide a high-level overview of MapReduce, specifically tailored to issues important to SNB. After that, I will describe implementing SNB on MapReduce. Finally, I will describe RepSN, the current state-of-the-art algorithm implementing SNB on MapReduce.

3.1 The Sorted Neighborhood Blocking Algorithm

SNB sorts the dataset and identifies potentially-matching tuple pairs based upon tuples *near* each other in sorted order. Thus, SNB leverages a sorting function to bring *similar* (potentially matching) tuples near each other.

The SNB process is as follows. First, use a function to produce a value for each tuple t . Then, sort the tuples based upon that value. Next, for every tuple t in sorted order, define a neighborhood of size w for t , where the neighborhood will consist of all tuples within w positions before or after t . In other words, in the sorted list of tuples $(t_1, t_2, t_3, \dots, t_n)$, tuple t_i at position i and t_j at position j are in the same neighborhood if $|i - j| < w$. Then, output all tuple pairs (t, u) where u is in the neighborhood of t . The candidate matching tuple pairs are passed to a subsequent process for matching. The matching process itself is orthogonal to the choice of blocking function, and as such is beyond the focus of this work.

An example of a set of neighborhoods is illustrated in Figure 1.2 on page 2. Note that the tuples 7, 8, and 9 each have different neighborhoods, and thus will output slightly different tuple pairs.

3.2 Existing SNB Solutions

There are several SNB solutions described in the literature. In 1995 Hernández and Stolfo first formalized SNB [16]. Through the early/mid 2010's new refinements to SNB have been proposed. These include extensions to dynamically adjust the neighborhood size by Draibach et al. in [11]. Importantly, Kolb, Thor, and Rham produced a number of works [18, 20, 21, 22] exploring SNB in a cluster environment. In [21] they explored implementing SNB on a MapReduce cluster. Despite being several years old, this remains the state-of-the-art solution for implementing SNB on MapReduce.

While SNB has existed in the literature for many years, it remains relevant and vital to EM systems. Systems such as Magellan [23] and DeDoop [19] offer SNB as blocking methods. With this as the backdrop, I will go over implementing SNB on MapReduce. I will start by giving an overview of the components of the MapReduce platform relevant for SNB.

3.3 MapReduce Overview

Before I describe SNB on MapReduce, I will first give a high-level overview of the MapReduce system as implemented by the popular open-source software package Apache Hadoop. In particular, I will examine the components that are important to consider as I describe the design of SNB on MapReduce.

MapReduce runs on a collection of computers, called a *cluster*. Each computer in the cluster is called a *node*. Users submit *jobs* to MapReduce that describe the desired computation. Each job takes input from a file. This file must be stored in HDFS, so the first thing is to load the file into HDFS.

Load Data The first task is to load the input file into the Hadoop Distributed File System (HDFS). Optionally, to adjust how HDFS stores the file, two additional parameters might be provided: a

Algorithm 3.1 MapReduce Pseudocode: Load Data

Input:

A file on disk
 (optional) a block size bs
 (optional) a replication factor k

Output:

A set of blocks, in HDFS, representing job_{if}

```

1:  $bs \leftarrow$  HDFS Blocksize (if specified, otherwise 128M)
2: Read input file into memory buffer
3:  $pieces \leftarrow$  Chop buffer into  $bs$ -sized pieces.
4:  $k \leftarrow$  HDFS replication factor (if specified, otherwise 3)
5: for each block  $b \in pieces$  do
6:   loop  $k$  times
7:     Find HDFS node  $n$  with available disk space.
8:     Write block  $b$  into a file, on disk, on node  $n$ .
9:   end loop
10: end for
  
```

block size bs , and an HDFS replication factor k . If not specified, cluster-wide defaults will be applied (typically 128 megabytes and 3, respectively.) MapReduce defaults to $k = 3$, although it is common for ephemeral data to be loaded with $k = 1$ (no replication). The output is a set of blocks in HDFS, representing the file.

To load the file into HDFS, first the file is chopped into *blocks*, which are segments of the file that are each bs long, except the last piece which may be shorter. For each block, locate an HDFS node n that does not already have a copy of this block, and write a copy of the block to a file on disk. Copies of this block are placed a total of k times, where k is the HDFS replication factor. This is illustrated in pseudocode in Algorithm 3.1.

Mappers After data is loaded into HDFS, the MapReduce job itself begins. The first step in the MapReduce job are the *mappers*, illustrated in pseudocode in Algorithm 3.2, where each mapper will run one map task. The input to the mappers is the set of blocks placed into HDFS in the previous phase, as well as a *map task* job_{mapfn} specified by the MapReduce job. The output is a set of intermediary (Key, Value) pairs to be passed to the next step in MapReduce.

For each block b , MapReduce finds one copy of b written to disk on some node n . Then, on node n , MapReduce will launch the mapper. The block b consists of a set of entries, where each entry e is a unique piece of data in the input file. Each mapper will execute the map task job_{mapfn}

Algorithm 3.2 MapReduce Pseudocode: Mappers

Input:

A set of potentially replicated blocks, in HDFS, representing the MapReduce job's input file

A map task job_{mapfn} specified by the MapReduce job

Output:

(Key, Value) pairs in *map_output_buffer* in RAM (spilled to disk if needed)

```

1: for each block  $b \in \text{HDFS}$  do
2:    $n \leftarrow$  node with a copy of  $b$  on disk
3:   Wait for slot on  $n$  to become available.
4:    $e \leftarrow$  set of entries from  $b$ 
5:   while output  $\leftarrow job_{mapfn}(e)$  do
6:      $map\_output\_buffer_b \leftarrow$  output
7:     if memory > 80% full then
8:       spill_to_disk( $map\_output\_buffer_b$ )
9:     end if
10:  end while
11: end for

```

passing the entries from b as input. Each e will produce 0, 1, or many intermediate (Key, Value) pairs as output. The mapper will write any (Key, Value) pairs output from the map task into an in-memory buffer on its node n . If memory becomes low, the in-memory buffer is spilled to the local disk.

Partitioners The next step of the MapReduce job is to run the *partitioners* (see pseudocode in Algorithm 3.3). On all nodes where mappers ran, one partitioner per node will take as input the intermediary (Key, Value) pairs from all of the mappers on that node. The partitioner will call the job-specified *partitioning function* job_{partfn} for each intermediary (Key, Value) also passing in the number of reducers r . One common partitioning function is to hash the key from the (Key, Value) pair into r buckets (where r is the number of reducers to be run in the next phase of the MapReduce job). For each (Key, Value) pair, the job_{partfn} will return a number p , which will refer to a particular reduce partition in the next phase of the MapReduce job. The partitioner then appends the (Key, Value) to an in-memory buffer specific for this value of p (spilling to disk if necessary).

Then, for each p returned from job_{partfn} , MapReduce sorts the in-memory (or spilled to disk) buffer. Next, MapReduce writes the now-sorted buffer to the local disk.

Algorithm 3.3 MapReduce Pseudocode: Partitioners

Input:

set of in-memory (or spilled to disk) map_output_buffers on each node, containing (Key, Value) pairs
 a partitioning function job_{partfn} , specified by the MapReduce job

Output:

On each node, one file on disk per each reduce partition, containing (Key, Value) pairs.

```

1: for each node  $n$  do
2:   for  $i = 1$  to  $r$  do
3:     partition_buffer $_i \leftarrow$  initialize
4:   end for
5:   for each map_output_buffer  $m \in$  map_output_buffer $_*$  do
6:     for each (Key, Value) pair  $\in m$  do
7:        $p \leftarrow job_{partfn}(\text{Key}, \text{Value}, r)$  ▷ Often hash(Key) modulus  $r$ 
8:       partition_buffer $_p \leftarrow$  (Key, Value)
9:     end for
10:  end for
11:  for each pb  $\in$  partition_buffer $_p$  do
12:    Sort and write pb to file $_p$ 
13:  end for
14: end for
  
```

Reducers The next phase of the MapReduce job are the *reducers* (as illustrated in pseudocode in Algorithm 3.4). The MapReduce job specifies a number of reducers r to run. For each partition number p , a reducer will be responsible for processing the p^{th} partition of the (Key, Value) pairs from the partitioning step, above. The reducer will run once on one of the nodes of the MapReduce cluster.

Thus, for each reducer r_p , first find a node n for r_p to run. On node n , the reducer first copies all the files destined for partition p written by the partitioners on nodes in the MapReduce cluster where mappers have been run. These files contain a sorted list of (Key, Value) pairs, and the reducer merges these files into a sorted in-memory buffer buf_p , spilling to disk if necessary.

Next, the reducer starts the *reduce task* $job_{reducefn}$ on the same node n , which takes as input the set of (Key, Value) pairs from buf_p . Any output returned from $job_{reducefn}$ is written into an in-memory buffer in the reducer on n (spilling to disk if necessary).

When complete, the in-memory buffer containing output from $job_{reducefn}$ is written into a file in HDFS, in a directory named for this MapReduce job, with a filename of this reducer.

When all reducers have finished, the MapReduce job has completed.

Algorithm 3.4 MapReduce Pseudocode: Reducers

Input:

A set of files for each reduce partition, on many nodes of the MapReduce cluster.
 A reduce task $job_{reducefn}$, specified by the MapReduce job.
 r reducers to run.

Output:

A set of blocks representing output files in HDFS, one set of blocks for each reduce partition, in a job-specific output directory

```

1: for each reduce partition  $p$  do
2:    $n_p \leftarrow$  node with available slot to run reduce partition
3:   for each node  $n$  do
4:     Copy  $file_p$  from  $n$  to node  $n_p$  as  $file_{p_n}$ 
5:     Merge sort all  $file_p$  into  $reduce\_buffer_p$ 
6:   end for
7:    $reduce\_output\_buffer_p \leftarrow job_{reducefn}(reduce\_buffer_p)$ 
8:    $load\_hdfs(output\_file_p) \leftarrow reduce\_output\_buffer_p$ 
9: end for
  
```

▷ See Algorithm 3.1

Output Collection Finally, if desired, the output files can be copied from HDFS into a single file on the local filesystem. Naturally, some uses of MapReduce may wish to leave the file in HDFS for further processing. In particular, some algorithms may require more than one MapReduce job to complete and, naturally, will typically leave the inter-job output in HDFS.

3.4 A simple SNB on MapReduce

Using the framework from the previous section, I will now describe a simple SNB using MapReduce. The inputs to this simple SNB are the following: a table of tuples in a file on disk, an HDFS replication factor k , a neighborhood size w , a number of reducers r to run, and a set of $(r - 1)$ split points sp . The split points are the values that will mark the boundaries between the r reducers. This simple SNB on MapReduce will output a single file on disk with TID pairs of potentially matching tuples.

The first step is to load the input file from the local filesystem into HDFS. MapReduce will place this into HDFS with replication factor k as described in Section 3.3. As HDFS is used only for temporary storage in this algorithm, k is typically set to one.

The MapReduce job begins with the mappers. As described, MapReduce will start a set of mappers, one mapper for each block representing the input file in HDFS. MapReduce will start

Algorithm 3.5 Simple SNB on MapReduce Pseudocode: job 1, $\text{job}_{\text{mapfn}}$

Input:

A set of tuples (tuple_set)
 A Blocking Key function BK

Output:

A set of (BKV, TID) pairs

```

1: procedure  $\text{job}_{\text{mapfn}}(\text{tuple\_set})$ 
2:   for each tuple  $t$  in  $\text{tuple\_set}$  do
3:      $\text{BKV} \leftarrow BK(t)$ 
4:      $\text{TID} \leftarrow t_{\text{ID}}$ 
5:      $\text{EMIT}(\text{BKV}, \text{TID})$ 
6:   end for
7: end procedure

```

the mapper on a node where one copy of the block is already written to disk. The mappers will each start a map task specific for SNB (as shown in Algorithm 3.5). The map task will read in the block of tuples assigned to this mapper. For each tuple t in the block, the following will occur. The map task calls $BK(t)$ to create the BKV for t . The map task then creates the TID from t 's ID or primary key. The map task emits a (BKV, TID) pair for each tuple. This continues for all tuples in the block.

Next, MapReduce starts the partitioners, with one partitioner on each node in the cluster where a mapper has been run for this MapReduce job. Each partitioner will call the SNB specific partition function for each (BKV, TID) pair, described in Algorithm 3.6. The partition function will require a set of split points provided by the user. The split points will dictate the boundaries between reduce partitions. These split points are in the form of a set $(B, D, G, K, \dots)$, and state that the first partition are any BKVs alphanumerically less than "B", the second is for BKVs greater or equal to "B", but less than "D", and so forth. Using the boundaries between reduce partitions, the partition function will look up which partition this BKV is in, and return a number p , which will refer to the reduce partition. MapReduce will use p to place the (BKV, TID) pair into a buffer for the p^{th} reduce partition. This continues until all (BKV, TID) pairs have been put into buffers for their respective reduce partitions.

Next, MapReduce will start the r reducers, which will run on available nodes across the MapReduce cluster. For each reducer, when placed on a node n , MapReduce copies the input

Algorithm 3.6 Simple SNB on MapReduce Pseudocode: $\text{job } 1, \text{job}_{\text{partfn}}$

Input:

A (Key, Value) pair
User supplied split points sp

Output:

A number between 0 and r

```

1: procedure  $\text{job}_{\text{mapfn}}(\text{Key}, \text{Value}, r)$ 
2:   for  $i \leftarrow 0$  to  $r$  do
3:     if  $\text{Key} < sp[i]$  then
4:       return  $i$ 
5:     end if
6:   end for
7:   return  $r$ 
8: end procedure

```

for each reduce partition from every node in the cluster to n and then starts the reducer for that reduce partition, again on node n . The reducer then starts the reduce task, specific for SNB as described in Algorithm 3.7. For each (BKV, TID) pair in the input to this reduce task, the reduce task will emit the TID paired up with the last $(w - 1)$ previously seen TIDs. For each entry in the input buffer, the reduce task determines TID t_{new} representing the tuple. Then for each of the last $(w - 1)$ seen TIDs each as t_{old} , the reducer emits $(t_{\text{new}}, t_{\text{old}})$. This continues for each (BKV, TID) pair in the reduce partition. MapReduce writes the output into HDFS.

As described, this computes SNB within each partition. However, a problem remains, SNB requires the sliding window to cross partition boundaries as in Figure 3.1. In that figure, tuple pairs, such as $((B, 2), (D, 4))$ are not yet returned as $(B, 2)$ is on node 1 and $(D, 4)$ is on node 2. The solution is to send the $(w - 1)$ tuples near a boundary to another MapReduce job as shown in Figure 3.2.

Thus, the process works as follows. For every partition, the first and last $(w - 1)$ tuples are written into files named by the boundary the tuples are near. Then, after the first MapReduce job finishes, a second MapReduce job is submitted. The second MapReduce job has no map task,

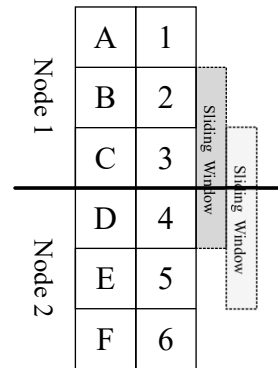


Figure 3.1: Distributed SNB: window spans splitpoint, $w = 3$

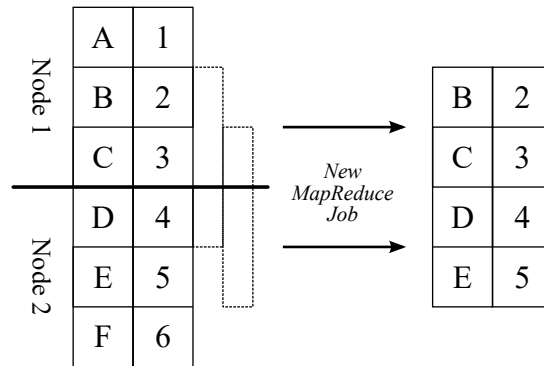


Figure 3.2: Replication of tuples after processing on each node: tuples near a boundary are sent to another MapReduce job, $w = 3$

so tuples are passed to the partitioner on the local node. The partitioner partitions based upon hashing the partition boundary. The reducers then compute the sliding window among the tuples that cross the partition boundary. For example, in Figure 3.2, the tuple pair $((B, 2), (C, 3))$ was already returned (by the previous MapReduce job) and thus it is not returned by this MapReduce job.

When all the reducers have finished, the output from both MapReduce jobs is copied from HDFS into a single file on disk. With that the simple SNB on MapReduce is finished.

Issues with the Simple SNB Sketch Unfortunately, there are two main issues with the simple SNB on MapReduce sketch I have illustrated.

First, the method to partition (BKV, TID) pairs across the reducers is unaddressed, and requires the user to know the distribution of BKV a priori. Thus, to partition in ranges requires the user to have knowledge of where to define the boundaries such that each range will produce the same amount of work from each reducer. This is not generally reasonable to expect nor desirable to require the user to have this knowledge. Furthermore, without automatic range partitioning, this will be highly vulnerable to skew in the output ranges.

Second, as every tuple has a unique id, it is typically unnecessary to either retain the entire tuple in memory or output the entire tuple in final output. For example: systems such as [23] only

use references to id numbers to represent the tuples. Retaining the entire tuple through processing will cause substantial additional processing cost without benefit.

I will now present RepSN, which was previously introduced in the literature to address these two issues.

3.5 RepSN

RepSN [18, 19, 20, 21] implements SNB on Apache MapReduce.¹ RepSN addresses the issue described in the previous section: RepSN avoids skew through careful automatic partitioning. At a high level, RepSN consists of two MapReduce jobs as shown in Figure 3.3.

Input into RepSN is a file that has been loaded into HDFS. The file is a table of tuples, where each tuple refers to a real-world entity. Also input to RepSN is a BK function, a neighborhood size w , an HDFS replication factor, and a number of reduce tasks r . RepSN will output a single file on the local filesystem. In this file will be TID pairs of potentially matching tuples.

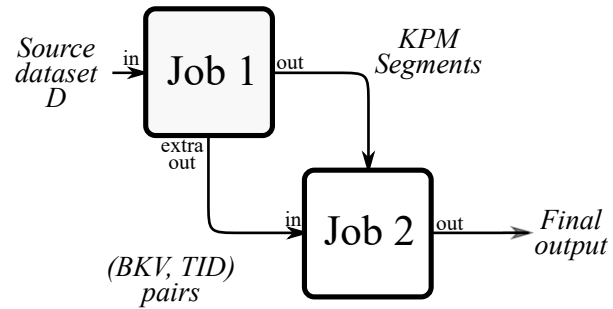


Figure 3.3: RepSN overview

MapReduce Job One After loading the input file into HDFS, RepSN starts the first MapReduce job. At a high level, this job will first apply the BK to each tuple, to create a (BKV, TID) pair, which the job writes to HDFS. Also, the first job writes out the BKV to the reduce tasks. The reduce tasks count each BKV into the data that will become the KPM, computed in the next MapReduce job.

Specifically RepSN takes the following actions in the first MapReduce job. As always, the first step in the MapReduce job are the mappers. The mappers each are assigned one (potentially replicated) block of the input file and a node n where one copy of the block exists on disk. Each

¹RepSN uses metadata added to each record to combine multiple BK functions and neighborhood sizes into a single execution of the workflow. To ease exposition, I have simplified this part of the algorithm for one pass, and direct the reader to [21] for additional detail.

mapper starts by executing the RepSN map task, as described in Algorithm 3.8. The map task starts by establishing this task's block number BN. Then, RepSN opens for writing a RepSN-specific temporary cache file Cache_{BN} . The contents Cache_{BN} will be used later in RepSN. Then, the map task will process each tuple in its block. For each tuple t in the block, the map task will find two values. First, to determine the BKV, the map task will call $BK(t)$ which returns the BKV. Next, the map task will assign the TID from the ID or primary key of the tuple. After determining the BKV and TID, the map task determines the Block Number (BN) for this block. The BN is a number that refers to this block of the input file in HDFS. Then, to the temporary file Cache_{BN} , the map task will write the triple (BKV, TID, BN). After that, the map task will emit to MapReduce the (Key, Value) pair $((\text{BKV}, \text{BN}), 1)$ where the key is a composite key of (BKV, BN). This continues for all tuples in the block. When that is finished, RepSN copies Cache_{BN} to HDFS for the next MapReduce job.

When all mappers have finished, MapReduce then starts the partitioners. MapReduce starts one partitioner per node in the cluster where a mapper has been run. Each partitioner will take as input the $((\text{BKV}, \text{BN}), 1)$ pairs output by the mappers on that node. For each $((\text{BKV}, \text{BN}), 1)$ pair, the partitioner hashes the BKV and applies a modulus of r , to produce a number p (Algorithm 3.9). The number p will refer to a particular reduce partition. MapReduce will write each $((\text{BKV}, \text{BN}), 1)$ into a file specific for reduce partition p on its node. This continues until all pairs have been processed.

Next, after all partitioners are finished, MapReduce starts the reducers. Each reducer is assigned one reduce partition and a node n to run on. For each reducer, MapReduce copies the files from all nodes for the reduce partition p to node n . The reducer then starts the reduce task as described in Algorithm 3.10. For each $((\text{BKV}, \text{BN}), 1)$ input to this reducer, the reduce task will count the total number of (BKV, BN) pairs. For each distinct pair, the map task will write out $((\text{BKV}, \text{BN}), \text{count})$. This continues until all pairs have been processed. MapReduce writes the output into HDFS. The complete set of output is known as the KPM, and will be used by the next MapReduce job.

MapReduce Job Two After the first MapReduce job ends, RepSN will start the second MapReduce job. At a high level, the second MapReduce job computes the KPM in each map task, and uses the KPM to distribute and replicate tuples to the reducers. Then, RepSN computes SNB within each reducer, writing the output to MapReduce, which writes the output to HDFS.

Specifically the second MapReduce job takes the following actions. The input to the MapReduce job is the directory of Cache_{BN} files. Since the input is a directory of files, MapReduce treats each Cache_{BN} file as a block. Thus, each mapper will be assigned one Cache_{BN} on a node where a copy of its assigned Cache_{BN} resides on disk. Also, every map task reads in the entire set of output from MapReduce job one. Each mapper starts a map task as illustrated in Algorithm 3.11.

Every map task begins by assembling the KPM. The KPM consists of an index of BKVs per input block. To build the index of BKVs for the KPM, every map needs the output from the last MapReduce job (which is broadcast to every node). The KPM is constructed in two phases. The first phase build an index based upon $(\text{BKV}, \text{BN}, \text{count})$ of all the entries from the previous MapReduce job.

The second phase constructs *this block's* KPM. The goal of the KPM itself is twofold: determine destination for each BKV, and determine which tuples need replication. To determine the destination and replication, RepSN each map task finds boundary points evenly through the BKVs. Thus as every map task knows the total number of tuples (from the index), the boundary points BP are simply points every $\left\lceil \frac{\text{total pairs}}{r} \right\rceil$ tuples through the dataset. Therefore, to generate *this block's* KPM, first the index is sorted on (BKV, BN) . RepSN sets the current partition cp to zero. Then, for each entry i in the index, RepSN performs the following. If i 's BN is this block's BN, then the map task inserts the offset into $KPM(\text{BKV})$. In all cases, RepSN increments cp by the number of entries by count. This repeats for all entries in the index. At this point the map task has its KPM, which is a set of offsets of the BKVs that occur *in this partition*.

With the KPM and BP , the map task now processes the entries from the Cache_{BN} file. For each entry in the Cache_{BN} file, the map task must calculate reduce partition for this BKV, and whether replication is required. To calculate the reduce partition, the map task reads offset i from $KPM(\text{BKV})$ (which causes the offset for this BKV in the KPM to increment by one). Then, to

tuples	BKV	TID	BKV	TID
lname="Cooper", age=63, id=3, ...	C	3	A	1
lname="Evans", age=33, id=5, ...	E	5	B	2
lname="Armstrong", age=42, id=1, ...	A	1	C	3
lname="Fvans", age=33, id=6, ...	F	6	D	4
lname="Betty Armstrong", age=42, id=2, ...	B	2	E	5
lname="David", age=45, id=4, ...	D	4	F	6

(a) Example: table of tuples

(b) after BKV generation in three workers

(c) sorted stream of TIDs

Table 3.1: Running Example. BK is the first letter of the lname attribute.

find which partition offset i belongs, the map task iterates through the split points, stopping when it finds the partition number p where i fits. If i is within $(w - 1)$ positions after a split point, it is replicated to partition $(p - 1)$.

Note that in the KPM the offset is incremented *each time* it is read. Thus, each time the KPM is called for a particular BKV, the answers will be unique, and may span a reduce partition. In other words, a very long segment of identical BKV may span reduce partitions, yet that long segment will be precisely partitioned so the correct number of tuples are sent to each reducer.

Now the map task will emit this (BKV, TID) pair to MapReduce to be routed to the reducers. For routing, RepSN adds explicit partition information to the pair: the destination partition number dp , as well as the original partition op number. For the second copy of a replicated tuple $dp \neq op$, otherwise $dp = op$. Thus, the map task emits a quadruple of (dp, op, BKV, TID) , using the values (p, p, BKV, TID) . Next, the map task checks to see if the offset for this BKV is near a partition boundary and requires replication. To check to see if the offset for this BKV is near a boundary, the map task checks to see if the offset is with $(w - 1)$ positions of a split point. If the offset is within $(w - 1)$ positions of a split point, the map task emits a copy of this (BKV, TID) pair, but destined to the previous partition. To route the (BKV, TID) to the previous partition, the map task emits a second quadruple of $((p - 1), p, BKV, TID)$. The second quadruple will be sent to the previous partition, and due to the standard MapReduce sorting, will always be sorted to the end.

TID	pairs added to output
1	
2	(1-2)
3	(1-3) (2-3)
4	(1-4) (2-4) (3-4)
5	(2-5) (3-5) (4-5)
6	(3-6) (4-6) (5-6)

Table 3.2: Output by each additional TID
with $w = 4$

1-2
1-3
2-3
1-4
2-4
3-4
2-5
3-5
4-5
3-6
4-6
5-6

Table 3.3: Final output with $w = 4$

This process continues for each of the entries in the Cache_{BN} file. MapReduce writes the output from each map task to an in-memory buffer on the same node.

Then, MapReduce starts the partitioners, one for each node where a mapper has run. Each partitioner reads all the output from the mappers. For each quadruple, the partitioner uses a SNB specific partition function (see Algorithm 3.15). The partition function emits the partition number based solely upon the value of first entry in the quadruple, the destination partition. As always, MapReduce writes the quadruple into a file specific for this partition number on disk on the same node.

Next, MapReduce starts the reducers. Each reducer is assigned a reduce partition and a node n to execute on. MapReduce routes all tuples for this reduce partition to this machine, merging the sorted per-machine results. MapReduce starts the reduce tasks as illustrated in Algorithm 3.16. For each entry e in the input buffer, the reduce task determines TID t_{new} representing the tuple. Then for each of the last $(w - 1)$ seen TIDs as t_{old} , the reducer emits $(t_{\text{new}}, t_{\text{old}})$. Then the reduce task adds t_{new} to the queue of $(w - 1)$ tuples. This process repeats for all e .

Next, MapReduce copies the output from the reduce tasks into HDFS. Then, RepSN copies the output from HDFS to a single file on the local filesystem. With this, RepSN is complete.

Example Now, I will describe an example of RepSN. An example of the logical flow of the first MapReduce job in RepSN is shown in Figure 3.4. In this example, I illustrate a tiny MapReduce

cluster with two nodes. As each node can run one reducer, r is 2. The sliding window size w is 3. For ease of exposition, the BK function is a simple function – it returns the first letter of the last name. The output will be to a single file on disk.

An example table of tuples is shown in Table 3.1a. In this example, the source data is a table of tuples referring to real-world people. The neighborhood size in the example is 4. Some occurrences of multiple tuples refer to the same real-world person. For ease of explanation, in this example the BK is a simple function that only returns the first character of the last name (family name). Thus, the BKV (the output from the BK) for `lname="Armstrong"` will be A. For instance, note that resulting from human error, the entry

```
[lname="Betty Armstrong", id=2]
```

is actually the same as

```
[lname="Armstrong", id=1]
```

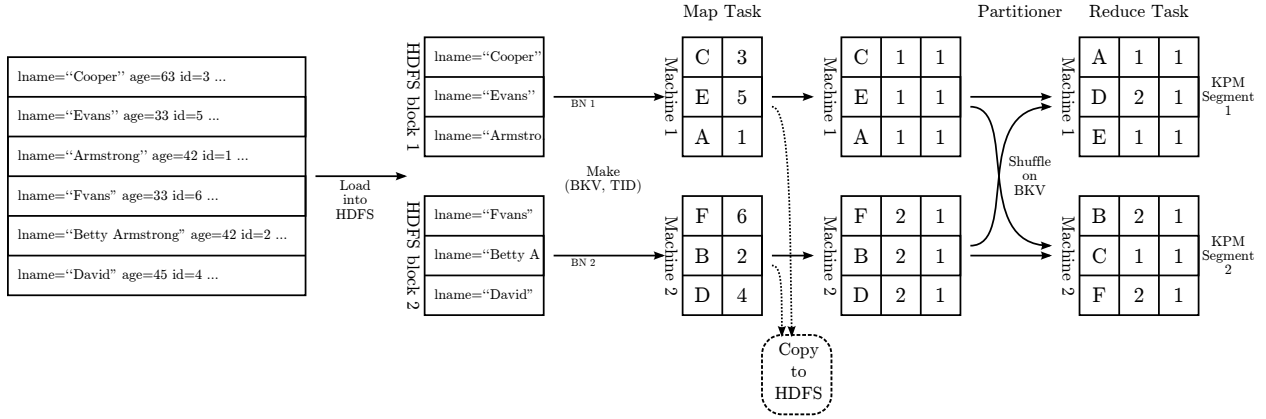
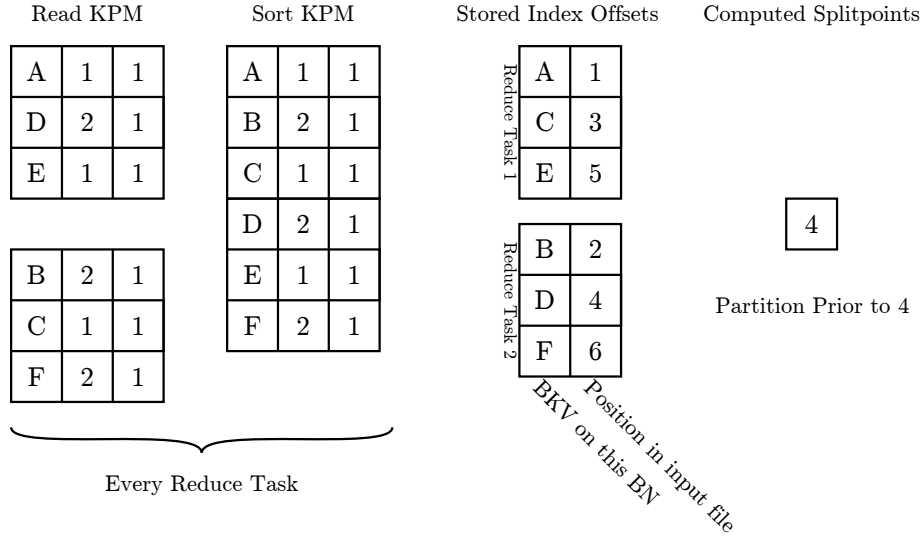
Furthermore, the entry for `[lname="Fvans"]` contains a typographical error and is a duplicate of `[lname="Evans"]`.

First, the on-disk input file to RepSN is copied into HDFS. HDFS places blocks of the file across nodes (in this case 2 nodes.) The mappers begin, each taking a block as input. The map task emits (BKV, BN, total count) to MapReduce. Next, the map task writes (BKV, TID, BN) to a cache file. When the map task is complete the cache file is copied into HDFS.

The partitioner shuffles on hashing the BKV, and each reduce task creates a sorted KPM segment.

The second MapReduce job then begins. In the second MapReduce job, each node computes the KPM in parallel (as in Figure 3.5a.) Each map task will read in the entire KPM and compute the final KPM offsets relevant to its input block as shown in Figure 3.5a. The map task only retains information regarding the BKVs in its input block (Figure 3.5b) as well as split points (Figure 3.5c.)

After the map task computes the KPM for its input partition, the map task begins reading the cached (BKV, TID) tuples for this BN from HDFS (Figure 3.6.) The map task calls `part(BKV)`,

Figure 3.4: RepSN: first MapReduce Job, $w = 3$, $r = 2$ Figure 3.5: KPM Construction during the second MapReduce job in RepSN: $w = 3$, $r = 2$.

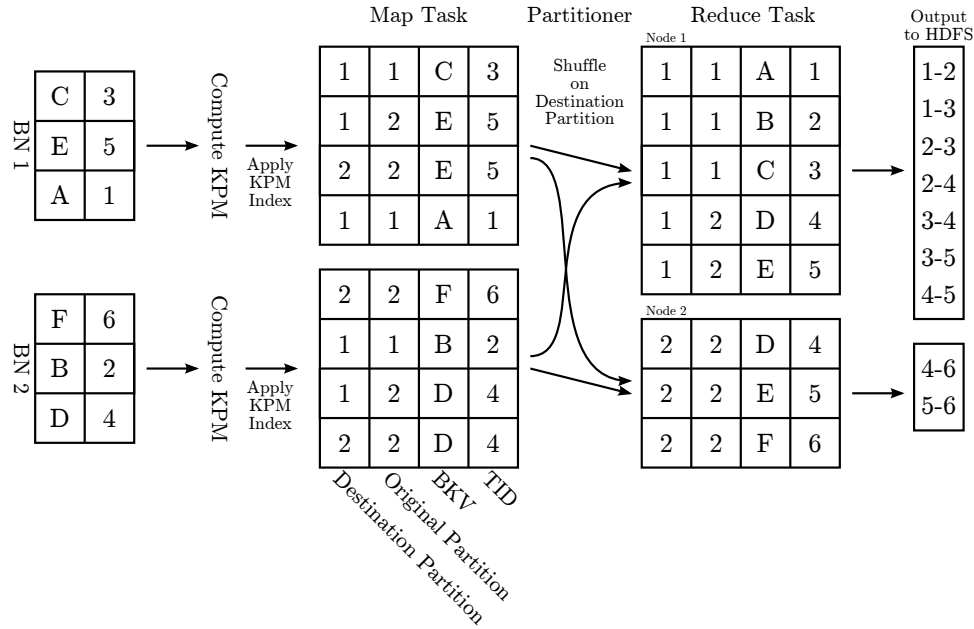


Figure 3.6: RepSN: second MapReduce job, $w = 3$, $r = 2$.

which computes the reduce partition. Also the map task computes if this BKV needs replication. In this example, with w of three, BKVs at positions 4 and 5 must be duplicated.

The partitioner hashes the destination partition to determine the reduce task for each quadruple.

The reduce task outputs each TID paired up with the last $(w - 1)$ TIDs, accounting for the overlapping BKVs D and E. For example, looking at Figure 3.6, on the final output, notice that the pair 4-5 was output by node 1 and not node 2. Machine 2 skipped outputting that pair because it was within $w - 1$ (in this case 2) positions of a split boundary.

The output is then written to HDFS, and then RepSN copies the output to a single file on disk (not shown). With this RepSN is complete. While RepSN is a robust solution, it was not tested against a simple baseline. I will describe that in the next chapter.

Algorithm 3.7 Simple SNB on MapReduce Pseudocode: job 1, $\text{job}_{\text{reducefn}}$

Input:

a buffer with (Key, Value) pairs
 neighborhood size w
 partition p

Output:

($\text{TID}_i, \text{TID}_j$) pairs

```

1: procedure  $\text{job}_{\text{reducefn}}$ (buffer)
2:    $sw \leftarrow \text{initialize}$ 
3:    $\text{first} \leftarrow \text{initialize}$ 
4:    $\text{last} \leftarrow \text{initialize}$ 
5:   for each (Key, Value) pair  $\in$  buffer do
6:      $\text{TID}_{\text{new}} \leftarrow \text{Key}$ 
7:     if  $\text{length}(\text{first}) < (w - 1)$  then
8:        $\text{first} \leftarrow \text{append}(\text{TID}_{\text{new}})$ 
9:     end if
10:    for each  $\text{TID}_{\text{seen}} \in sw$  do
11:       $\text{EMIT}(\text{TID}_{\text{new}}, \text{TID}_{\text{seen}})$ 
12:    end for
13:     $sw \leftarrow \text{append}(\text{TID}_{\text{new}})$ 
14:    if  $\text{length}(sw) \geq w$  then
15:       $sw \leftarrow \text{remove\_oldest}(sw)$ 
16:    end if
17:  end for
18:   $\text{last} \leftarrow sw$ 
19:   $\text{write}(p_{\text{first}}, \text{first})$ 
20:   $\text{write}(p_{\text{last}}, \text{last})$ 
21: end procedure

```

▷ At the end, sw needs replication

Algorithm 3.8 RepSN Pseudocode: job 1, $\text{job}_{\text{mapfn}}$

Input:

A set of tuples (tuple_set)
 A Blocking Key function BK
 BN

Output:

A set of (BKV, TID) pairs
 A cache file in HDFS for this BN

```

1: procedure  $\text{job}_{\text{mapfn}}$ ( $\text{tuple\_set}$ )
2:    $\text{Cache}_{\text{BN}} \leftarrow \text{open}(\text{temporary file})$ 
3:   for each tuple  $t$  in  $\text{tuple\_set}$  do
4:      $\text{BKV} \leftarrow BK(t)$ 
5:      $\text{TID} \leftarrow t_{\text{ID}}$ 
6:      $\text{Cache}_{\text{BN}} \leftarrow \text{write}(\text{BKV}, \text{TID}, \text{BN})$ 
7:      $\text{EMIT}((\text{BKV}, \text{BN}), 1)$ 
8:   end for
9:    $\text{load\_hdfs}(\text{Cache}_{\text{BN}})$ 
10: end procedure

```

Algorithm 3.9 RepSN Pseudocode: job 1, $\text{job}_{\text{partfn}}$

Input:

A (Key, Value) pair
 number of reducers r

Output:

A number between 0 and r

```

1: procedure  $\text{job}_{\text{mapfn}}$ (Key, Value,  $r$ )
2:   return hash(Key) mod  $r$ 
3: end procedure

```

Algorithm 3.10 RepSN Pseudocode: job 1, $\text{job}_{\text{reducefn}}$

Input:

a buffer with (Key, Value) pairs

Output:

file with sorted (BKV, BN, count) triples

```

1: procedure  $\text{job}_{\text{reducefn}}$ (buffer)
2:   lastBKV  $\leftarrow$  0
3:   lastBN  $\leftarrow$  0
4:   lastcount  $\leftarrow$  0
5:   for each (Key, Value) pair  $\in$  buffer do
6:     BKV  $\leftarrow$  KeyBKV
7:     BN  $\leftarrow$  KeyBN
8:     count  $\leftarrow$  Value
9:     if lastcount  $>$  0 and (BKV  $\neq$  lastBKV or BN  $\neq$  lastBN) then
10:      EMIT(lastBKV, lastBN, lastcount)
11:      lastBKV  $\leftarrow$  BKV
12:      lastBN  $\leftarrow$  BN
13:      lastcount  $\leftarrow$  0
14:     end if
15:     lastcount  $\leftarrow$  lastcount + count
16:   end for
17:   EMIT(lastBKV, lastBN, lastcount)
18: end procedure

```

Algorithm 3.11 RepSN Pseudocode: job 2, $\text{job}_{\text{mapfn}}$

Input:A set of tuples (tuple_set)

KPM files (entire output from all reducers in job 1)

Output:

A set of (destination partition, original partition, BKV, TID) quadruples

```

1: procedure  $\text{job}_{\text{mapfn}}(\text{tuple\_set})$ 
2:    $\text{KPM} \leftarrow \emptyset$ 
3:   for each tuple  $t \in \text{tuple\_set}$  do
4:      $\text{BKV} \leftarrow t_{\text{BKV}}$ 
5:      $\text{TID} \leftarrow t_{\text{ID}}$ 
6:      $\text{BN} \leftarrow t_{\text{BN}}$ 
7:     if  $\text{KPM} = \emptyset$  then
8:        $(\text{KPM}, \text{splitpoints}) \leftarrow \text{build\_KPM}(\text{KPM\_directory}, \text{BN}, r, w)$  ▷ See Algorithm 3.12
9:     end if
10:     $p \leftarrow \text{get\_partition}(\text{KPM}, \text{splitpoints}, \text{BKV})$ 
11:     $\text{EMIT}(p, p, \text{BKV}, \text{TID})$ 
12:    if  $\text{needs\_replication}(\text{KPM}, \text{BKV})$  and  $p > 0$  then ▷ See Algorithm 3.14
13:       $\text{EMIT}((p - 1), p, \text{BKV}, \text{TID})$ 
14:    end if
15:  end for
16: end procedure

```

Algorithm 3.12 RepSN Pseudocode: build_KPM

```

1: procedure BUILD_KPM( $\text{KPM\_directory}, \text{BN}, r, w$ )
2:    $\text{KPM\_files} \leftarrow \text{open all files in HDFS directory KPM\_directory}$ 
3:    $\text{KPM\_entries} \leftarrow \text{start sort merge (KPM\_files)}$ 
4:    $\text{offset} \leftarrow 0$ 
5:   for each entry  $e \in \text{KPM\_entries}$  do
6:      $\text{this\_BKV} \leftarrow e_{\text{BKV}}$ 
7:      $\text{this\_BN} \leftarrow e_{\text{BN}}$ 
8:      $\text{this\_count} \leftarrow e_{\text{count}}$ 
9:     if  $\text{this\_BN} = \text{BN}$  then
10:       $\text{KPM}[\text{this\_BKV}] \leftarrow \text{offset}$ 
11:    end if
12:     $\text{offset} \leftarrow \text{offset} + \text{this\_count}$ 
13:  end for
14:   $\text{total\_tuples} \leftarrow \text{offset}$ 
15:   $\text{offset} \leftarrow 0$ 
16:   $\text{tuples\_left} \leftarrow \text{total\_tuples}$ 
17:  while  $\text{tuples\_left} > 0$  do
18:     $\text{tuples}_p \leftarrow \min \left( \left\lceil \frac{\text{total\_tuples}}{r} \right\rceil, \text{tuples\_left} \right)$ 
19:     $\text{offset} \leftarrow \text{offset} + \text{tuples}_p$ 
20:     $\text{splitpoints} \leftarrow \text{append}(\text{offset})$ 
21:     $\text{tuples\_left} \leftarrow \text{tuples\_left} - \text{tuples}_p$ 
22:  end while
23:  return  $(\text{KPM}, \text{splitpoints})$ 
24: end procedure

```

Algorithm 3.13 RepSN Pseudocode: get_partition

```

1: procedure GET_PARTITION(KPM, splitpoints, BKV)
2:    $i \leftarrow \text{KPM}[\text{BKV}]$ 
3:    $\text{KPM}[\text{BKV}] \leftarrow \text{KPM}[\text{BKV}] + 1$  ▷ Increment in case this BKV called again
4:    $j \leftarrow 0$ 
5:   for  $s \leftarrow 0$  to  $\text{length}(\text{splitpoints})$  do
6:     if  $\text{splitpoints}[s] \leq i$  then
7:        $j \leftarrow s$ 
8:     end if
9:   end for
10:  return  $j$ 
11: end procedure

```

Algorithm 3.14 RepSN Pseudocode: needs_replication

```

1: procedure NEEDS_REPLICATION(KPM, BKV)
2:    $i \leftarrow \text{KPM}[\text{BKV}]$ 
3:    $\text{KPM}[\text{BKV}] \leftarrow \text{KPM}[\text{BKV}] + 1$ 
4:   for  $s \leftarrow 0$  to  $\text{length}(\text{splitpoints})$  do
5:      $\text{distance} \leftarrow i - \text{splitpoints}[s]$ 
6:     if  $\text{distance} \geq 0$  and  $\text{distance} < w$  then
7:       return True
8:     end if
9:     if  $i < \text{splitpoints}[s]$  then
10:      return False
11:    end if
12:  end for
13:  return False
14: end procedure

```

Algorithm 3.15 RepSN Pseudocode: job 2, job_{partfn}

Input:

A (Key, Value) pair

Output:

A number between 0 and r

```

1: procedure jobmapfn(Key, Value,  $r$ )
2:   (destination_partition, original_partition, BKV)  $\leftarrow$  split(Key)
3:   return destination_partition
4: end procedure

```

Algorithm 3.16 RepSN Pseudocode: job 2, $\text{job}_{\text{reducefn}}$

Input:

a buffer with (Key, Value) pairs
neighborhood size w

Output:

($\text{TID}_i, \text{TID}_j$) pairs

```

1: procedure  $\text{job}_{\text{reducefn}}$ (buffer)
2:    $sw \leftarrow \text{initialize}$ 
3:    $\text{extra} \leftarrow 0$ 
4:   for each (Key, Value) pair  $\in$  buffer do
5:     (destination_partition, original_partition, BKV)  $\leftarrow \text{split}(\text{Key})$ 
6:      $\text{TID}_{\text{new}} \leftarrow \text{Value}$ 
7:     if  $\text{extra} < w$  then
8:       for each  $\text{TID}_{\text{seen}} \in sw$  do
9:          $\text{EMIT}(\text{TID}_{\text{new}}, \text{TID}_{\text{seen}})$ 
10:      end for
11:       $sw \leftarrow \text{append}(\text{TID}_{\text{new}})$ 
12:      if  $\text{length}(sw) \geq w$  then
13:         $sw \leftarrow \text{remove\_oldest}(sw)$ 
14:      end if
15:      if destination_partition  $\neq$  original_partition then
16:         $\text{extra} \leftarrow \text{extra} + 1$ 
17:      end if
18:    end if
19:  end for
20: end procedure

```

Chapter 4

Analyzing Current MapReduce SNB Solutions

So far I have surveyed the related work on blocking in Chapter 2 and existing solutions for SNB in Chapter 3. In this chapter I will evaluate RepSN, introduced in Section 3.5, through an empirical evaluation which compares RepSN with a robust implementation on a single machine.

4.1 SNB on a Single Machine

To analyze the performance of RepSN, I will compare RepSN against a baseline solution: running the SNB algorithm on a single machine. In this section, I will cover the basic implementation of implementing SNB on a single machine, and point out some enhancements.

To run SNB on a single machine, the same three items described seen in Chapter 3 are needed for input. The first input is a file on disk. In this file is a table of tuples. Each tuple refers to real-world tuples, and in the table some of the tuples might be regarded as duplicates (they refer to the same real-world entity.) The second input parameter is a dataset-specific BK function which will create a key for each tuple, which I will refer to as the Blocking Key Value (BKV). When the tuples are sorted on the BKV, possibly-matching tuples will be sorted near each other. The final input parameter is the size of the neighborhood w .

The output from SNB will be a file on disk containing a set of tuple pairs that are in the same neighborhood and thus are candidates for matching.

The baseline solution for implementing SNB on a single machine is as follows. First, create an in-memory datastructure to hold a bag of keys and values (such that a key may be repeated). Next, open the input file on disk. Then, for each entry in the file, read the string representing a tuple from

Algorithm 4.1 Baseline SNB pseudocode

```

1:  $b \leftarrow \emptyset$  ▷ Initialize  $b$  with an empty set
2:  $recent \leftarrow \emptyset$  ▷ Initialize  $recent$  with an empty set
3: for all  $e \in dataset$  do
4:    $t \leftarrow decode(e)$  ▷ the tuple is encoded with an on-disk format
5:    $b \leftarrow append(BKV(t), t)$ 
6: end for
7:  $b \leftarrow sort(b)$ 
8: for all  $i \in \text{results of } b$  do
9:   for all  $r \in recent$  do
10:     $EMIT((i, r))$ 
11:   end for
12:    $ENQUEUE(recent) \leftarrow i$ 
13:   if  $length(recent) \geq w$  then
14:      $DEQUEUE(recent)$  ▷ removes oldest from  $recent$ 
15:   end if
16: end for

```

the file into an in-memory buffer. Then, from the buffer, decode each string into an in-memory tuple. Next, pass the tuple to the BK, which will return the BKV for the tuple. Insert the BKV and the tuple into the in-memory key, value, datastructure, where the BKV is the key. This continues for all entries in the input file.

The next step is to sort the in-memory key/value datastructure based upon the keys (BKV's). Finally, pass a sliding window of size w through the now sorted key/value datastructure. At each stop of the sliding window, write out to the output file on disk each tuple paired up with each of the other tuples within the sliding window. As the sliding window moves, only return new tuple pairs not already output. When the sliding window has reached the end of the in-memory key/value datastructure, the algorithm is complete. This is illustrated in pseudocode in Algorithm 4.1.

Unfortunately, this algorithm as described has several problems.

The primary issue is that this algorithm does not take into account how much memory is consumed. This algorithm uses an in-memory sort. Thus, if system memory is full, this algorithm will run into problems. While this algorithm could potentially rely on virtual memory, in contemporary systems this will be prohibitively slow. Even then, virtual memory could also become fully consumed, which then would then cause this algorithm to fail.

The second issue is, as described, only a single core is used. A ubiquitous feature of modern machines is the presence of multiple CPU cores. Thus, potential gains from parallelism are not realized. Finally, operations are not pipelined, missing out on additional performance benefits.

4.2 SNB-single

To address the limitations of the simple single machine algorithm above, I now introduce **SNB-single**. Using a single computer, **SNB-single** uses in-memory datastructures, multiple cores, and pipelined operations. Algorithm 4.2 illustrates **SNB-single** in pseudocode and Figure 4.1 shows the conceptual data flow.

SNB-single takes four items as input. First, **SNB-single** requires a table of tuples in a single file on disk, where each tuple has a unique tuple id TID. The second input is a BK function which will produce a BKV for each tuple. Third, **SNB-single** needs a neighborhood size w . Finally, **SNB-single** needs an amount of system memory to reserve m_{res} .

SNB-single outputs a single file on disk containing potentially-matching TID pairs.

The implementation of **SNB-single** is as follows. First, the controller starts one worker for each CPU core. Then, the controller creates two in-memory buffers for each worker: one buffer will be used as input to the worker from the controller, the other buffer will be used for output from the worker back to the controller. The worker input buffer will hold both strings from the file (undecoded tuples) as well as commands from the controller. The worker output buffer will hold (BKV, TID) pairs and status updates from the worker to the controller.

Next, the controller opens the input file on disk, and creates an input buffer to read strings from the file. Until the input file is empty, the following is repeated. From the input file, the controller will read from the file a bulk of strings into the controller's input buffer. The controller then moves this bulk of strings into the input buffer for the next worker. The next worker is chosen in round robin fashion. The moving of bulks of strings continues until the input file is empty. When the file is empty, the controller puts an "end" command into every worker's input buffer, instructing the workers to return their results.

Each worker waits for bulks of strings to be put into its input buffer. When work is put into the buffer, some number of string-encoded tuples are now available for the worker. Then, for each tuple in the buffer, the worker first decodes the tuple into an in-memory datastructure. Next, the BK is called for the tuple, returning a BKV. The worker inserts the (BKV, TID) into its in-memory sorted datastructure. The worker then deletes the tuple from memory.¹ This repeats for every tuple in the input buffer. When the input buffer is empty the worker waits for the buffer to be refilled.

When the worker receives the “end” command from the controller to return its results, the worker fills its output buffer with its sorted list of (BKV, TID) pairs. If the buffer is full, the worker blocks until there is buffer space available. While (BKV, TID) pairs remain in memory, the worker will continue to refill the output buffer. When zero (BKV, TID) pairs remain, the worker puts an “end” status update into its the output buffer and terminates.

Meanwhile, the controller reads the output buffer from every worker. Similar to a merge sort, the controller examines each worker’s output buffer to find the smallest (BKV, TID) pair. The controller will then remove that (BKV, TID) pair from the worker’s output buffer and put the TID into the controller’s output buffer. Then, the controller will remove the next smallest (BKV, TID) pair and will insert that TID into the controller’s output buffer. When the controller received the “end” status update from a worker, the controller removes that worker’s buffer. This continues until no output buffers from any worker remain.

For each TID inserted into the controller’s output buffer the controller computes the sliding window. To compute the sliding window, the controller pairs up the new TID with each of the last $(w - 1)$ TIDs in the output buffer. For each of those last $(w - 1)$ TIDs, the controller will write to the output file on disk the pair of the new TID and the prior TID. When all TIDs have been inserted into the output buffer, and the output TID pairs have been written to disk, the algorithm is complete.

Memory Control I will now zoom in on how memory is handled in SNB-single. SNB-single uses an in-memory sort. With very large input files or low system memory, however, available system memory could be exhausted during SNB-single.

¹If matching is also including then the whole tuple is retained.

As the controller is reading the input file, the controller also monitors the amount of system memory available. On Linux, one method is to read `/proc/meminfo`. When the free system memory drops below m_{res} , the controller inserts a “flush to disk” command into the input buffer for each worker.

When each worker receives the flush to disk command, each worker flushes its sorted in-memory datastore to a new temporary file on disk for this worker. The temporary file is written in sorted order. The in-memory datastore is now empty, and the worker resumes waiting for tuples to be placed into its buffer. For large data sources, the memory may be flushed to disk many times, so each worker might have many temporary files on disk.

Then, when a worker receives the command to return the results, the worker creates a buffer for each temporary file written to disk. Next, the worker fills each buffer with the beginning of each sorted temporary file. Similar to a sort-merge, the worker will merge (BKV, TID) pairs from the in-memory datastore with the (BKV, TID) pairs in the buffers from each file on disk. The smallest (BKV, TID) is placed into the output buffer for this worker. As needed, the worker fills the buffers with data from the on-disk temporary files. As above, when all tuples have been returned, the worker writes an “end” status update into its output buffer and terminates.

Example In this example, the source data is a table of tuples referring to real-world people (as in Section 3.5). An example table of tuples is shown in Table 3.1a (on page 28). The neighborhood size in the example is 4. Some occurrences of multiple tuples refer to the same real-world person. For ease of explanation, in this example the BK is a simple function that only returns the first character of the last name (family name). Thus, the BKV (the output from the BK) for `lname="Armstrong"` will be A. For instance, note that resulting from human error, the entry

```
[lname="Betty Armstrong", id=2]
```

is actually the same as

```
[lname="Armstrong", id=1]
```

Furthermore, the entry for `[lname="Fvans"]` contains a typographical error and is a duplicate of `[lname="Evans"]`.

Dataset	size
DBLP	4.6M records
FEC Individual Donors	25.2M records
Users Database	36.4M records

Table 4.1: Datasets used to compare RepSN with a single machine implementation of SNB

The controller starts reading the on-disk table of tuples (again Table 3.1a, page 28). The workers process the tuples to create sorted runs of (BKV, TID) pairs as in Table 3.1b (page 28). Then, as in Table 3.1c, the master merges the sorted runs from the workers into a single sorted list of (BKV, TID) pairs. While the master creates the sorted list, it passes a sliding window of size w over the TIDs as in Table 3.2 (page 29). The final output is shown in Table 3.3 (page 29).

Observations SNB-single works well on a single machine. However, to scale SNB, more machines will be needed. One popular method to use many machines is with MapReduce, as we saw with RepSN in Section 3.5. Next, I will give an empirical evaluation of SNB-single versus RepSN.

4.2.1 Comparing RepSN with SNB-Single

In this section I will empirically evaluate the relative performance of RepSN and SNB-single. For these experiments I used three datasets: the DBLP Computer Science bibliography, a set of individual donors to United States Federal Elections from the United States Federal Elections Commission (FEC), and a “users database” found online from an online website. See Table 5.3 for a brief description of the datasets. See Appendix A.1 for a detailed description of the datasets, and Appendix A.3 for the experimental setting

The graphs in Figure 4.2 (page 48) show the results of each algorithm with respect to each dataset. The horizontal axis represents the number of tuples given as source input to each SNB algorithm. The vertical axis is the number of hours taken for each type of blocking. For my work, the total time an algorithm takes is measured from the time the single file of source tuples on a local filesystem is opened until resultant candidate matches are finished being written to a single output

file on the local filesystem. SNB-Single is the baseline solution, and illustrates the execution of SNB on a single machine.

Looking at the graphs in Figure 4.2 closely reveals an interesting phenomenon. The performance of RepSN initially performs better than a single machine. However, as the dataset size increases the scale-up of RepSN is poor, and the time gain decreases. Eventually, RepSN using a cluster of 10 machines is *slower* than a single machine SNB solution.

4.2.2 Explaining RepSN Performance

It is clear from Figure 4.2 that RepSN runs into an issue and performs worse than SNB-single when datasets are large. I have identified two issues.

Issue One: Waves of Map Tasks The first issue is the number of map tasks RepSN needs to run. As defined in [21], the RepSN uses a standard HDFS block size (128 megabytes). When loaded into HDFS, the input file is split into blocks. MapReduce will start one mapper per block in the input file. As the size of the input file is increased, the number of blocks also increases. As the number of blocks increases, the number of required mappers increases. Each MapReduce cluster has a configured number of mappers that can run per node, leading to a fixed number of *slots* that mappers can run in. When the number of mappers exceeds the number of slots available, then *waves* of mappers will run. For example, if there are ten computers, and each computer can run four mappers, then 40 mappers can run concurrently. If 41 mappers need to run, the first 40 run to completion. When a slot is free the 41st mapper begins, and runs to completion.

Thus, when the source input file is larger than 128 megabytes times the number of mapper slots available, the MapReduce job will need to operate in waves. For example, imagine a Hadoop cluster that is able to run 40 map tasks at once. If the input file of tuples is 4500MB (4.5GB), 36 map tasks will be required ($\frac{4500}{128} = 35.16$). Since 36 map tasks is less than 40, all tasks can run on the cluster simultaneously. However if the input file of tuples is 5500 MB (5.5GB), 43 map tasks will be required ($\frac{5500}{128} = 42.97$). This means that 40 tasks will run to completion in the first wave. Then in the second wave, 3 tasks will run to completion. If the input file is 25,000 MB, then 196 map tasks will be required, and will run in 5 waves. The equation to calculate the number of waves

needed is straightforward: $\left\lceil \frac{\frac{\text{size of input file}}{\text{block size}}}{\text{number of map tasks a cluster can run}} \right\rceil$. It is the ceiling of the size of the input file divided by the block size divided by the number of map tasks a cluster can run.

This explains the substantial superlinearity shown by RepSN in Figure 4.2. Increasing waves of map tasks computing the KPM account for the superlinear run time expansion. Herein lies the problem with RepSN: RepSN assumes that the KPM will be built by all mappers in parallel and the wall-clock cost is hidden because with small datasets the KPM is created at *the same time by each map task*. However, instead of the end-to-end time having one KPM computation in parallel, with many waves of map tasks, the cost of the KPM is paid many times.

However, the first issue does not completely explain the performance of RepSN. Even when adapting the blocksize such that there is always at most one wave of mappers, the performance is no longer superlinear, is and slightly better than a single machine, but not substantially. Furthermore, very large blocks causes problems with MapReduce. For example, one problem with very large blocks is that it is very RAM intensive, and will lead to crashes due to running out of program memory. While compensation is possible for these issues, the second issue remains in all cases.

Issue Two: Cost of the KPM The second issue is fundamental to RepSN’s algorithm. Fixing just the number of blocks so that the KPM can be generated in one wave will reduce the superlinearity caused by multiple map task waves. However, the KPM generation cost itself is still an issue.

As discussed in Section 3.5, the goal of the KPM is establish split points that will balance the output from each reducer in the second MapReduce job. The KPM is an *exact* measurement. However, looking at sorting in Hadoop, sorting algorithms use a *sampled* part of the data to choose split points. This leads to innovations upon RepSN, which I will introduce in the next chapter.

Algorithm 4.2 SNB-single pseudocode

```

1:  $b \leftarrow \emptyset$ 
2:  $recent \leftarrow \emptyset$ 
3:  $workers \leftarrow$  Initialize one per CPU Core
4:  $worker\_queues \leftarrow$  Initialize one per worker
5: for all  $q \in worker\_queues$  do
6:   START WORKER( $q$ )
7: end for
8:  $current\_worker \leftarrow 0$ 
9: for all  $e \in dataset$  do
10:   $worker\_queues[current\_worker].enqueue(e)$ 
11:   $current\_worker \leftarrow next\_worker()$ 
12:  if memory is low then
13:    for all  $q \in worker\_queues$  do
14:       $worker\_queues[q].flush()$ 
15:    end for
16:  end if
17: end for
18: for all  $i \in$  results of  $merge\_sort(worker\_queues.dequeue())$  do
19:  for all  $r \in recent$  do
20:    EMIT(( $i.TID, r.TID$ ))
21:  end for
22:  ENQUEUE( $recent$ )  $\leftarrow i$ 
23:  if  $length(recent) \geq w$  then
24:    DEQUEUE( $recent$ )
25:  end if
26: end for
27: procedure WORKER( $queue$ )
28:  for all  $e \in queue.dequeue()$  do
29:     $t \leftarrow decode(e)$ 
30:     $b \leftarrow append\_sorted(BKV(t), TID(t))$ 
31:  end for
32:  for all  $pair \in b$  do
33:     $queue.enqueue(pair)$ 
34:  end for
35: end procedure

```

▷ Initialize b with an empty set
 ▷ Initialize $recent$ with an empty set

▷ Launch worker

▷ Typically e will be in bulks.

▷ Round robin

▷ Flushes tuples to disk

▷ $sort(b)$ is pipelined

▷ removes oldest from $recent$

▷ the tuple is encoded with an on-disk format

▷ If flushed to disk, merge on-disk BKV,TID pairs

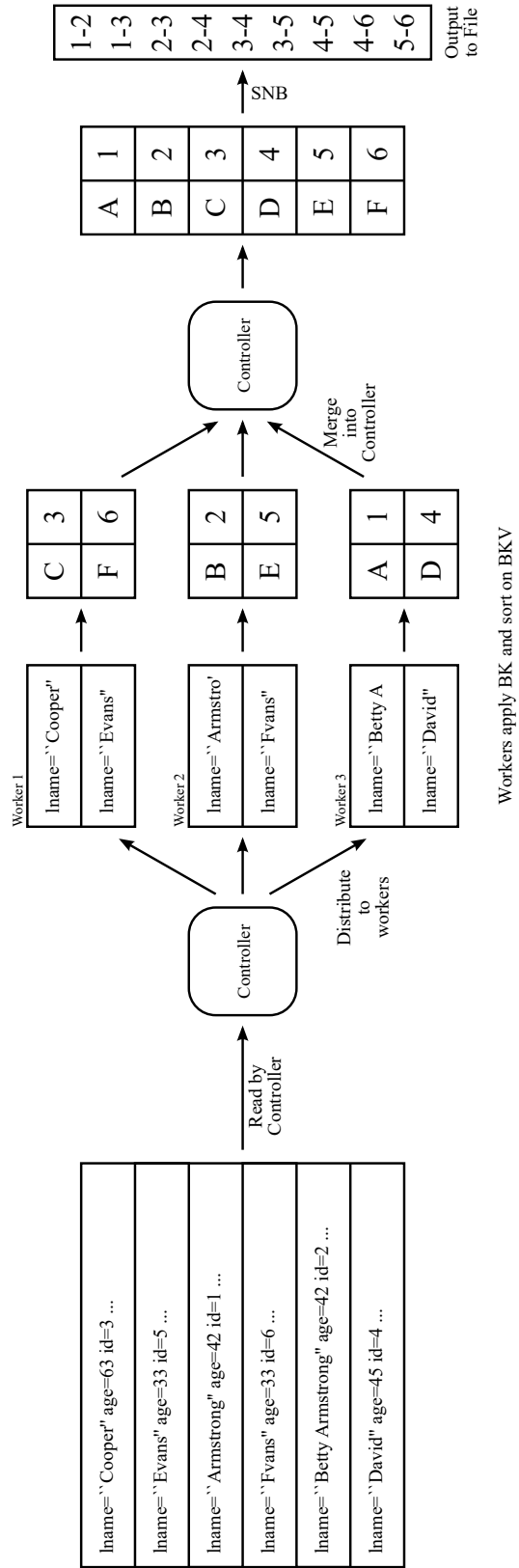
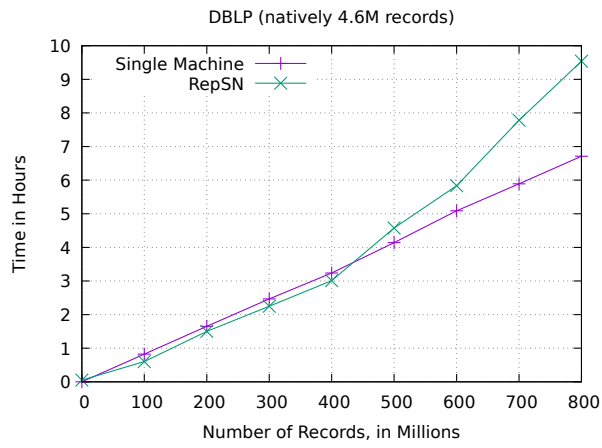
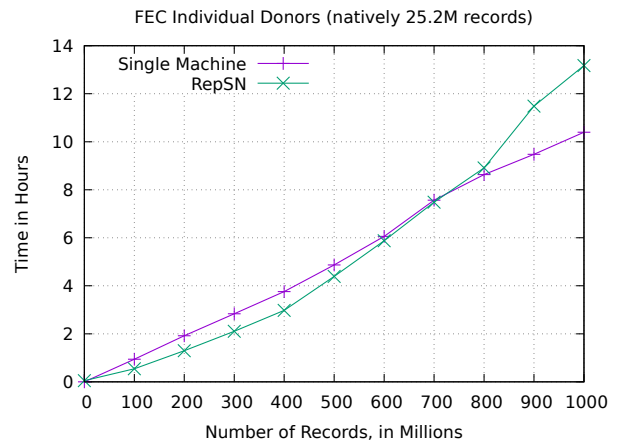


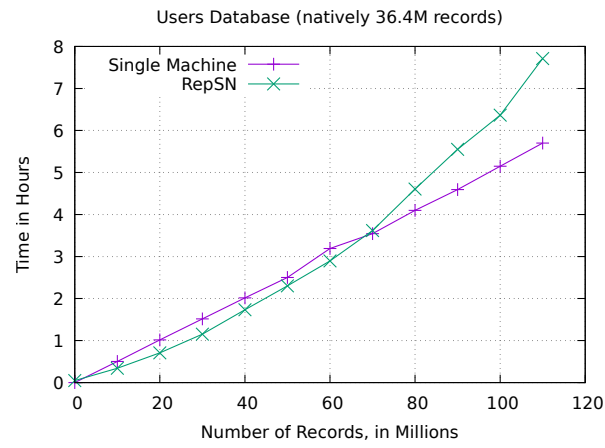
Figure 4.1: SNB-Single: $w = 4$



(a) DBLP



(b) FEC Individual Donors



(c) Users Database

Figure 4.2: Relative performance: RepSN vs SNB-single

Chapter 5

Developing Better SNB Solutions on MapReduce

As shown in Chapter 4, there are substantial challenges regarding the performance of RepSN. In particular, the cost to create the KPM is prohibitive for large amounts of data. In this chapter I will address these challenges.

Specifically, I describe two methods to compute SNB using MapReduce. Both methods seek to improve performance over RepSN. First, in Section 5.1, I will introduce a simple method to compute SNB using MapReduce: SNB-MR-Simple. SNB-MR-Simple improves upon RepSN through using *off-the-shelf* components. Then, in Section 5.2, I describe VSNB, which improves upon SNB-MR-Simple by streamlining the workflow to compute SNB on MapReduce in an efficient manner.

5.1 SNB-MR-Simple

When considering the performance challenges of RepSN, I start with the observations in Figure 2.3 on page 9. As shown in that figure, at its core, SNB is a set of three operations: generate a key for each record, sort the records by that key, then slide a window across the sorted order. MapReduce naturally sorts large amounts of data very effectively [30]. The key idea behind of SNB-MR-Simple is to use an existing off-the-shelf sorting method within MapReduce to implement the sort with SNB.

The inputs to SNB-MR-Simple are a table of tuples in a single file on the local disk, a BK function, and a window size w . The output from SNB-MR-Simple is a set of TID pairs in a single file on the filesystem.

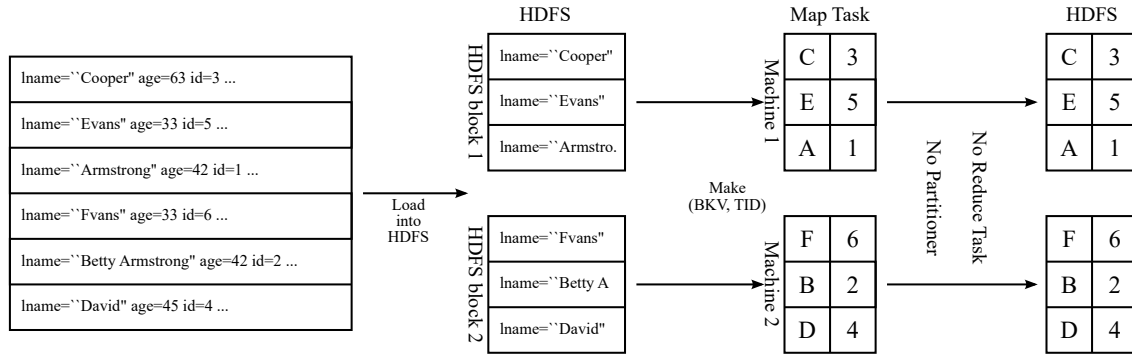


Figure 5.1: SNB-MR-Simple: first MapReduce job

In RepSN, the intuition of the KPM is to exactly control how much work each final reduce task will need to do. In contrast, standard sorting example provided by Hadoop uses a sampled subset of the dataset to determine splitpoints. SNB-MR-Simple itself is agnostic to the inner workings of how the file is sorted.

5.1.1 SNB-MR-Simple in Detail

SNB-MR-Simple consists of two MapReduce jobs. After the MapReduce jobs, a post-process on a single machine prepares the output. The key idea behind SNB-MR-Simple is that by using simple off-the-shelf components¹ performance can be increased over RepSN. I found performance to be surprisingly better than RepSN.

Similar to the other algorithms, first SNB-MR-Simple loads the input table of tuples from a single file into HDFS. As we have already seen, HDFS loads the file into a set of blocks, which are spread across the MapReduce cluster.

MapReduce Job One The goal of the first MapReduce job is to convert the raw tuples into (BKV, TID) pairs. So, for each block of the input file in HDFS, MapReduce starts a map task on a node with a copy of the block on disk. Then, the map task will read each tuple from its input block. For each tuple, the mapper will emit a (BKV, TID) pair from that tuple. This continues for each tuple

¹By off-the-shelf, I mean components literally distributed with the Hadoop software.

in the block. In the first MapReduce job, there are no partitioners or reducers. Thus, the output is written into HDFS. An example of this is shown in Figure 5.1.

MapReduce Job Two The second MapReduce job uses a generic MapReduce sort to sort the tuples by BKV. For this document, I used the sort function implemented in `hadoop-mapreduce-examples` distributed as an example file with Apache Hadoop. For example, see Figure 5.2. Here, the sort program reads the (BKV, TID) tuples from HDFS, runs a standard sorting algorithm, and writes the result into another file in HDFS.

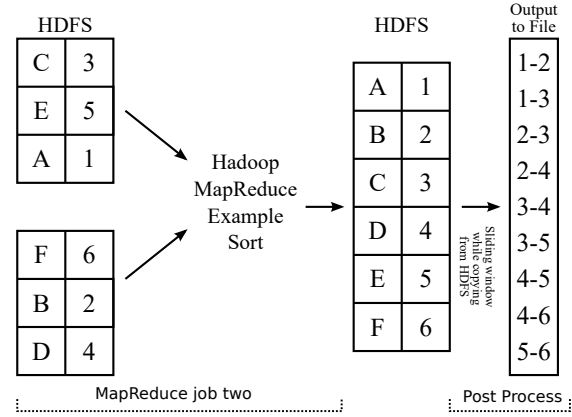


Figure 5.2: SNB-MR-Simple: second MapReduce job and post process task, $w = 3$

Post Process The last step for SNB-MR-Simple is to read the file from HDFS and com-

pute TID pairs of records within a neighborhood. First, SNB-MR-Simple creates a queue q of maximum length $(w - 1)$. Then, SNB-MR-Simple reads the first (BKV, TID) pair in the output from the last MapReduce job. Then for each position q_j in q , SNB-MR-Simple outputs (TID, q_j) to the output file. Finally, SNB-MR-Simple adds the TID to q , and if the length is over $(w - 1)$, removes the oldest entry. This continues for every (BKV, TID) pair in the output from the file in HDFS.

5.1.1.1 MapReduce Example Sort

It is instructive to look at how the standard MapReduce sort code operates. The key idea is that sorting efficiently in MapReduce is achieved by first selecting split points (or boundary points). The split points will segment the file to be sorted into similar-sized partitions. Then, MapReduce uses the split points to send the smallest records to the first partition, the next largest to the next

partition, up to the largest records to the last partition. Note this is at the heart of what SNB needs to do: sort.

Specifically the “example sort” code works as follows It takes as input a file of records, a sampling percentage, a maximum number of samples to keep, and a number of reducers r . The output is a sorted file in HDFS.

First, if not already in HDFS, the file to be sorted is loaded into HDFS. Then, on a single machine, the sort code samples records from the input file. The samples are both as a percentage as well as a maximum number of samples. Then, the samples are sorted. Next, $(r - 1)$ samples are selected evenly through the sorted samples (to become the boundaries for r reducers). Those $(r - 1)$ samples are written to a file in HDFS.

Next the sort code submits a MapReduce job. There is no map task, so MapReduce simply copies each input block to the partitioner.

After the map tasks have finished, MapReduce starts a partitioner on each machine a map task has run. Then, the partitioner reads the split points from HDFS, and establishes the ranges for each reduce partition.

Next, the partitioner reads every record on the machine. Then, for each record on a machine, a partitioner determine a partition number p , which will refer to a particular reduce partition. The reduce partition will be input to the next phase of the MapReduce job. The partitioner appends the record to an in-memory buffer specific for this value of p (spilling to disk if necessary.) When the partitioner is finished, MapReduce (as always) sorts each in-memory buffer.

When all partitioners are finished, MapReduce will then start r reducer tasks. Each reduce task will be assigned to a node n , and a reduce partition number p . As always, MapReduce copies all records destined for p from each node across the cluster to n . MapReduce merges the sorted buffers from each node into a single sorted list.

Note that the input to the reduce task is the segment of the entire sorted list, in sorted order. As there is no additional work to do, MapReduce writes the records in sorted order to HDFS. When all reducers are finished, then the sort is complete.

Limitations of SNB-MR-Simple Clearly, SNB-MR-Simple suffers from two performance issues. First, the sliding window computation is a single machine, which will be a bottleneck. Second, SNB-MR-simple generates all the (BKV, TID) pairs, and then has the standard MapReduce sort re-scans the file. Next, I will propose VSNB, which addresses these two issues.

5.2 Vanilla Sorted Neighborhood Blocking using MapReduce (VSNB)

I now introduce my solution which improves RepSN: Vanilla SNB using MapReduce, or VSNB. VSNB takes as input dataset D , which is a file of records on the local filesystem of a single machine. VSNB also requires typical SNB-specific variables: a window size w , a blocking key function BK (and, optionally, a matching function M). VSNB also determines the number of reduce tasks r , from MapReduce. VSNB will output to the local filesystem on a single machine a file containing a set of matches from D , in an efficient manner.

Overview Intuitively, as we saw in Figure 2.3 on page 9, SNB is composed of two high-level operations: sort dataset D by a key, and then use a sliding window of size w to block (and, optionally, match using M) records within w positions in the sorted list.² Therefore, in MapReduce, VSNB will need to both sort D , and block (or match) records within w positions in sorted order. However, to scale VSNB, D will be partitioned across a cluster of computers. By partitioning D across a cluster, some record pairs needing blocking (or matching) will be on different partitions, potentially on different computers.

Thus, for VSNB on MapReduce, I will use three MapReduce jobs as illustrated in Figure 5.3:

1. A MapReduce job to compute keys and identify key ranges suitable to evenly partition the dataset into r partitions.
2. A MapReduce job to sort the dataset and compare records within w positions within those ranges.

²Note that in this section, I generally omit the matching function from the description. However, in Chapter 7 I extend VSNB to add adaptivity, which requires a matching function. See that chapter for more detail.

3. A MapReduce job to handle the record pairs that span ranges, potentially on different computers in the cluster.

At a high level, VSNB operates as follows. The first MapReduce job in VSNB takes in dataset D and writes out a new dataset $\bar{D}$ of (BKV, TID) pairs.³ Additionally, the first MapReduce job takes a sample of the key values to produce a set of key ranges P . The goal is that when $\bar{D}$ is partitioned using the ranges defined by P , each partition will contain roughly the same number of records.

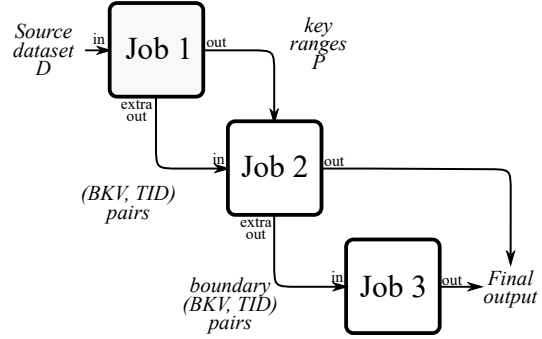


Figure 5.3: VSNB overview

The second MapReduce job in VSNB reads in $\bar{D}$, and partitions the records across the reduce tasks using P . Then, each reduce task blocks (and, optionally, matches using M) all record pairs within w positions of each other using a sliding window.

The areas near the beginning or end of a partition (near the *boundary* to another partition) will contain records that must be compared with records across the boundary in another partition. Therefore, for each record close to a boundary b within P , the reduce task outputs the record along with id number for b and the id of its partition.

The third, and final, MapReduce job is a small job that takes as input the boundary records from the second MapReduce job. Then the third MapReduce job blocks (and, optionally, matches using M) the records that came from different partitions in the previous MapReduce job.

Finally, VSNB copies the blocking (or matching) output from the second and third MapReduce jobs to a single file on disk on a single machine.

5.2.1 VSNB in Detail

Now, I will describe VSNB in detail. VSNB uses a MapReduce cluster containing n identical machines, where each machine has k cores, resulting in a cluster-wide total of $n \times k$ cores N_k .

³If M is defined, then $\bar{D}$ will be the pairs (BKV, original record).

lname	id	age	fname	...
Cooper	id3	63	Evans	...
Evans	id5	33	Cooper	...
Armstrong	id1	42	Betty	...
Fvans	id6	33	Cooper	...
Betty Armstrong	id2	42	[null]	...
David	id4	45	Susan	...

Table 5.1: Source data used for the examples in this chapter.

VSNB has two tunable parameters: the HDFS blocksize B , and a sampling frequency s . B and s are returned from a cost model described further in Section 5.3.

First, VSNB loads D into HDFS, with block size B . During loading, VSNB counts the number of records N_D in the dataset. VSNB begins by copying the table of tuples from a single file on disk into HDFS. As always, HDFS segments the file into blocks, and places the blocks onto nodes in the MapReduce cluster, potentially replicated.

For example, please see Table 5.1, which is a dataset representing a set of people. In this dataset some records are to be identified as duplicates. In this example, VSNB is called with window size $w = 3$, the Blocking Key Function BK returns the first letter of the lname field, and there are two machines, each with one core. The cost model determines HDFS block size B to be equivalent to three records.

Load Data VSNB first loads the source data file into HDFS using blocksize B . During loading, VSNB counts the number of records N_d . The result is shown in Figure 5.4: there are now two HDFS blocks, each containing three records. HDFS places the blocks onto the HDFS worker nodes, and each block is on a different machine, with no redundancy.

VSNB runs the calibration procedure to produce s (see Section 5.3 for details). For our running example, the cost model returns $s = 2$.

MapReduce Job One Next, VSNB starts the first MapReduce job, illustrated in Figure 5.5. The goal of the first MapReduce job is twofold: compute keys for every record and create a set of ranges P to sort the keys across the MapReduce cluster.

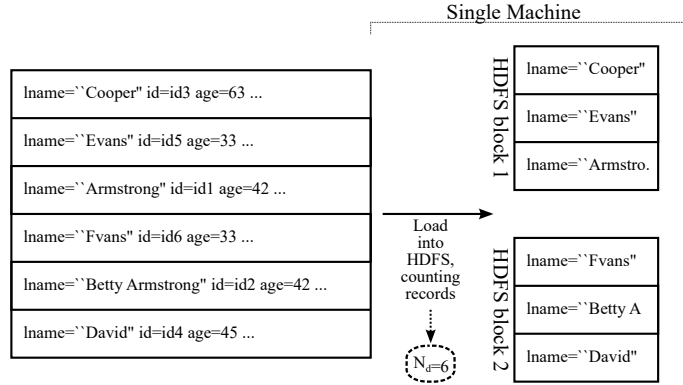


Figure 5.4: VSNB: Load Dataset into HDFS

As always, MapReduce assigns each map task one block of records from D in HDFS. MapReduce starts the map task on an available core. If the number of blocks exceeds N_k , then the map tasks will also exceed N_k , and the map tasks will run in *waves* (batches of N_k map tasks that run concurrently) until complete.

For each map task, the map task first opens an output file H_1 in HDFS. Next, the map task must sample the key from one out of every s records. Then, once every s records, output (BKV, TID) to H_1 .

Looking at the example in the dataset in Figure 5.5, the BKV is C (the first letter of the lname attribute Cooper). As the TID is 3, and no matching is being performed, VSNB writes (C, 3) to HDFS. Additionally the map task determines this record is to be part of the sample, so C is output back to MapReduce. Next, the map task computes the BKV for the next record, resulting in E, and writes (E, id5) to HDFS. This BKV is not part of the sample, so no output is made.

When the map tasks have finished, the first MapReduce job in VSNB has a single reduce task, and as a result there is no partitioner. MapReduce copies the sample *BKV*s to the machine in the cluster that will run the reduce task. As always, MapReduce sorts the sample *BKV*s. In Figure 5.5, note that the output from Machine 1 and Machine 2 are combined, and sorted.

The reduce task begins. The reduce task selects $(N_k - 1)$ values from the sample *BKV*s. VSNB takes the $(N_k - 1)$ values and produces partitions P . For the example in Figure 5.5, only two ranges are needed, so one split point must be chosen.

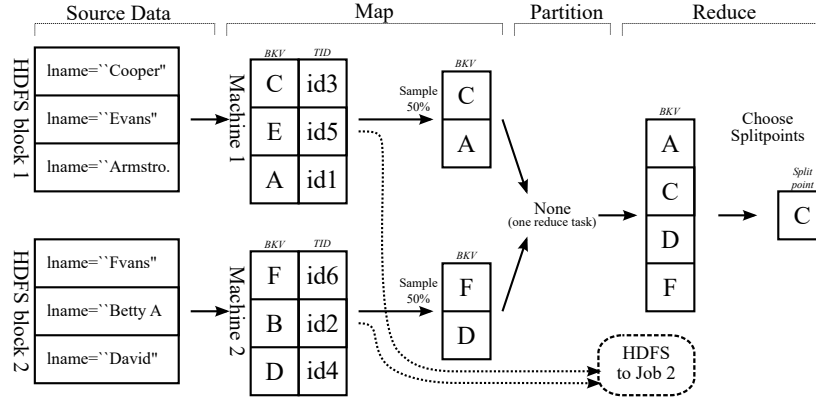


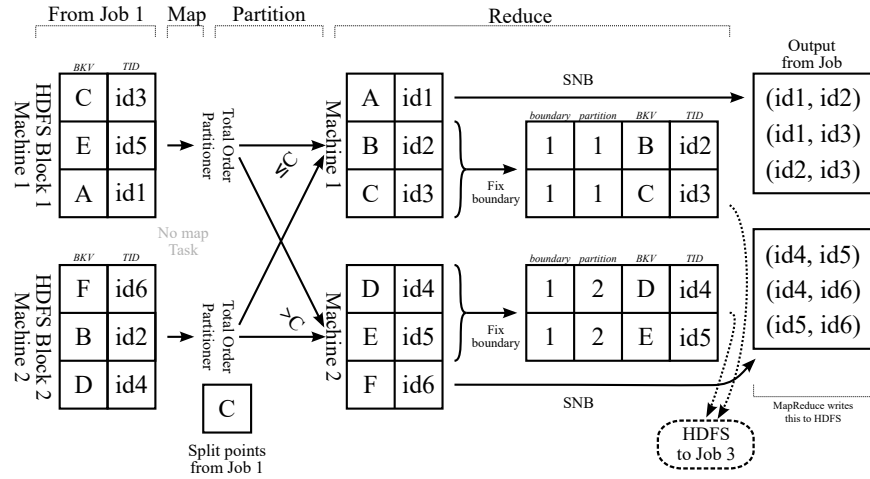
Figure 5.5: VSMB: first MapReduce job, $w = 3$, $s = 2$

VSMB picks a record half-way through the sample records (in this case breaking the tie between C and D by choosing the former). This results in two ranges: keys less than or equal to C , and keys larger than C .

MapReduce Job Two VSMB then starts the second MapReduce job. The second MapReduce job outputs record pairs that block (or, optionally, match) within each partition. Additionally, the second MapReduce job outputs records that are near boundaries of the partition so the next MapReduce job can handle them.

The second MapReduce job takes the (BKV, TID) pairs from the first MapReduce job as input. There is no map task, so MapReduce copies all (BKV, TID) pairs in H_1 to the MapReduce partitioner. This MapReduce job uses the Apache Hadoop provided `TotalOrderPartitioner`. The Total Order Partitioner operates differently than the more commonly-used MapReduce hash-based partitioner. The hash-based partitioner distributes records according to the hashed value of some (or all) of the attributes of a record. The Total Order Partitioner, however partitions the records into ranges and sends each range to a different reducer. The first reducer receives the lowest-valued records, the second reduce receives the next bunch of records, and so forth, up to the last reducer which receives the largest records.

For each tuple, the `TotalOrderPartitioner` uses the BKV from the tuple to choose which of the N_k reducers the tuple should be sent to. The smallest BKV are sent to the first reducer, the next largest to the second reducer, and so on with the largest BKV sent to the last reducer.

Figure 5.6: VSNB: second MapReduce job, $w = 3, s = 2$

	incoming record	queue O	output		
Time 1:	(A, id1)	<table><tr><td></td></tr><tr><td></td></tr></table>			
Time 2:	(B, id2)	<table><tr><td></td></tr><tr><td>id1</td></tr></table>		id1	(id1, id2)
id1					
Time 3:	(C, id3)	<table><tr><td>id1</td></tr><tr><td>id2</td></tr></table>	id1	id2	(id1, id3) (id2, id3)
id1					
id2					

Table 5.2: Example: sliding window computation in VSNB

Therefore, in this toy example, there are only two ranges: tuples with BKVs that are alphanumerically less than or equal to C and tuples with BKVs that are alphanumerically greater than C. With this configuration, as seen in Figure 5.6, the tuples with BKVs of A, B, and C, are sent to the reduce task that will run on machine 1, and the others are sent to the reduce task on Machine 2.

When the partitioning is complete, MapReduce starts the reduce tasks. Each reduce task starts by initializing the set of open windows O to be empty. Next, the reduce task opens output file H_2 in HDFS. Then, the reduce task starts to read the input from MapReduce, which is the input (BKV, TID) pairs I . Finally, the reduce task finds its partition number pn , where the first partition is 1 and the last partition is N_k .

For each record i in I , the reduce task performs two operations. The first operation is to output the result from blocking. For every record o in O , the pair (i_{TID}, o_{TID}) is written⁴ to output file H_2 . Next, i is added to O . If the size of O is greater than $(w - 1)$, then the oldest record is removed from O .

So, as seen in Table 5.2, for the first reduce task the BKV A and the corresponding TID id1 is read. O is empty, so no comparisons are made. Then, (A, id1) is added to O . Next, the reduce task reads in (B, id2). O contains (A, id1), the reduce task outputs the pair (id1, id2). The reduce task pushes (B, id2) into O . Finally, the reduce task reads (C, id3). The reduce task returns all pairs between id3 with all the entries in O , returning (id1, id3) and (id2, id3). Concurrently, a similar process occurs on machine 2, with its set of records.

The second operation the reduce task performs is to identify records near the partition boundaries. Records within a window size of the boundary will require additional processing. Then, if r is in the first $(w - 1)$ entries, then the boundary number bn is set to $(pn - 1)$. If r is in the last $(w - 1)$ entries, then bn is set to pn . Otherwise bn is not set. Finally, if bn is set, and $0 < bn < N_k$, then the reduce task writes (bn, pn, BKV, r) as output to MapReduce, which MapReduce writes to HDFS.

As shown in Figure 5.6, there is only one boundary, so the records required for the next MapReduce job are the ones at the end of the reduce partition on machine 1 and the beginning of the reduce

⁴If matching is desired, the record pair is written only when $M(i, o)$ returns true.

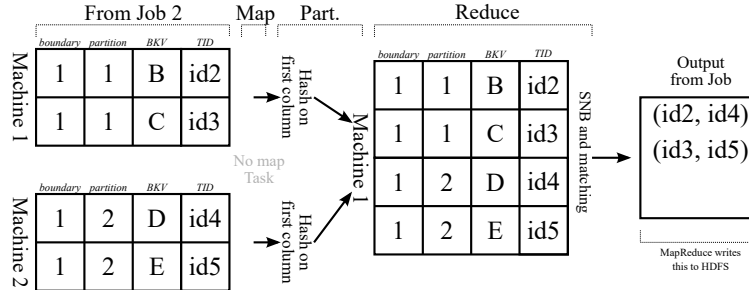


Figure 5.7: VSNB: third MapReduce job, $w = 3$, $s = 2$

partition on machine 2. As there is only one boundary, $bn = 1$ for all records. This will become the input to the next MapReduce job.

This whole process continues for all r in R . When this is complete, the reduce task is also complete. Then, when all the reduce tasks are complete, this MapReduce job is also complete.

MapReduce Job Three VSNB starts the third MapReduce job which takes as input the output from the second MapReduce job. The third MapReduce job handles the record pairs that span reduce partitions, as illustrated in Figure 5.7.

In this MapReduce job there is no map task, so each (bn, pn, BKV, TID) tuple is copied to the MapReduce partitioner. The MapReduce partitioner used by this MapReduce job is the standard hash-based partitioner. The partitioner chooses the destination reduce partition based on bn (the first entry of the tuple). MapReduce sorts by the entire tuple.

Then, MapReduce starts the reducers. Each boundary bn will have two sets of tuples: tuples from the end of one partition, and tuples from the beginning of the next partition. The goal of the reduce task is to compute SNB on records between these two sets.

Each reduce task takes as input a set of tuples T . The reduce task starts by initializing a set of open windows O to be empty. The reduce task then, one at a time, reads in a tuple t in T from MapReduce. Each tuple t is the quadruple (bn, pn, BKV, r) .

For each tuple t in T , the reduce task performs one of the following. If the $t_{bn} = t_{pn}$, then this tuple is before the split point, and the reduce task loads t into O .

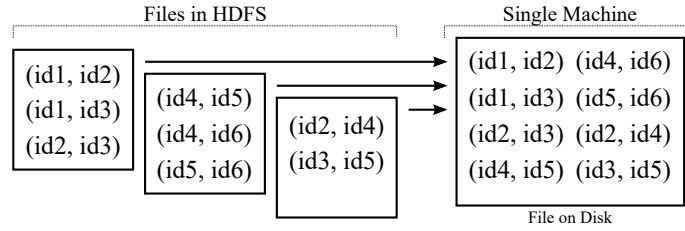


Figure 5.8: VSNB: copy data from HDFS

Otherwise, $t_{bn} \neq t_{pn}$ means that t is after the split point, and due to sorting, all future t for this bn will be after the split point. If t is after the split point, the reduce task compares t against every record o in O . The map task outputs each (t_{TID}, o_{TID}) combination.⁵ Then, the oldest record is removed from O . This continues until all input tuples have been processed.

Note that in Figure 5.7, the pairs $(id2, id3)$ and $(id4, id5)$ were already output in Figure 5.6, and were not output a second time.

Output Collection Once the MapReduce jobs finish, then VSNB collects the output from HDFS. As seen in Figure 5.8, VSNB copies the output from MapReduce jobs 2 and 3 and writes them to a single file on disk. With this VSNB is complete.

Discussion Note that the tuple pair $(id3, id6)$ was not returned, and note that from Table 5.1 those tuples actually match the same real-world person “Cooper Evans”. This is due to a limitation of the static window sizes in standard VSNB.

Assume the end-user could identify a situation like this in an actual run of VSNB. The only way he or she could fix this problem in VSNB is to re-run with a larger w . This window size limitation is addressed by adding adaptive window sizes to SNB. Adaptive window sizes is described in Chapter 6, and I introduce my enhanced adaptive SNB (ASNB) in Chapter 7.

5.3 Modeling Cost and Runtime

I will now describe how to estimate the runtime of VSNB, in order to find a good plan to execute VSNB efficiently on a MapReduce cluster. The goal is to perform SNB, using MapReduce,

⁵If matching is desired, then $M(t, o)$ must return true in order to be output.

on dataset D , which starts as a file of records on disk. The same standard SNB parameters are required, as we saw in Section 5.2.1, including: a sliding window size w , a Blocking Key function BK , and an optional matching function M . The output of VSNB is a set of matches to a single file on disk, in an efficient manner.

The rest of this section is as follows. In Section 5.3.1 I describe tunable parameters for VSNB, and the corresponding plan space produced by these parameters. Next, in Section 5.3.2 I describe the cost model for VSNB. Then, in Section 5.3.3 I describe how to calibrate the cost model. Finally, in Section 5.3.4 I describe how to search for optimal parameters to minimize run time.

5.3.1 Tunable Parameters for VSNB

VSNB runs using MapReduce on a cluster of n identical machines. Each machine has k cores, for a cluster total of N_k cores. D is in a single file of size $|D|$ megabytes, with N_D records, on a single machine outside the cluster.

I will now describe a set of tunable parameters K for VSNB. There are three parameters K that can be adjusted. Of these, two parameters significantly affect runtime: the HDFS blocksize B , and the sample frequency s . The third parameter (skew computation correctness probability P_s) can be set to a reasonable value ($P_s = 0.01$) without affecting performance (see Section 5.4.3).

The first tunable parameter in K is the HDFS blocksize B (Apache Hadoop uses a default blocksize of 128 megabytes). VSNB begins by copying D to HDFS. HDFS partitions D into blocks of size B , resulting in N_B blocks. Additionally, MapReduce *always* schedules at least one map task per HDFS block.

The second tunable parameter in K is the number of items in the sample, produced by the sampling frequency s . At the core of VSNB is a probabilistic distributed sorting algorithm that sorts the dataset by the blocking key BK . The algorithm chooses $(N_k - 1)$ boundary points that define N_k partitions of the dataset. To compute the $(N_k - 1)$ boundary points, first sort the items in s , and then choose every $\frac{s}{N_k}$ -th record as a boundary.

The plan space P is the Cartesian product of s and B . The blocksize B can be varied from as low as the length of one record in D , to as high as the Java heap size as configured in the

MapReduce JVM. For the sample size, one can choose $(N_k - 1) \leq s \leq N_D$. When the sample size is the minimum $(N_k - 1)$ records are used for the sample, random split points are chosen. With the maximum sample size, every record in D is used to determine the boundary points.

5.3.2 Cost Model

In order to find the best plan from P , I constructed a cost model $CM(p)$ that maps a plan $p \in P$ to an estimated runtime cost for our algorithm. The model uses the following set of parameters:

- Cluster-specific variables:
 - n identical machines, each having k cores, resulting in N_k total cores
 - C_{MR} : average time to start a map or reduce task
 - J : maximum usable Java JVM memory size
- Dataset-specific derived variables:
 - N_D : total number of records in dataset D
 - $|D|$: size of D in megabytes
- Dataset-specific variables:
 - C_k time to compute blocking key for each record
 - C_s time to move one sample to the reducer
 - C_b time to compute blocking for a record pair
 - C_M time to compute the matching function between a pair of records (if used)

Ideally, using a particular plan p , $CM(p)$ is a combination of the time taken by the five steps of VSNB:

$$CM_{\text{total}}(p) = T_l + T_g + T_c + T_o + T_u \quad (5.1)$$

Where:

- T_l : Time to load data into HDFS
- T_g : Time to generate Blocking Key values for every record in D , and pick $(N_k - 1)$ boundary points (VSNB MapReduce job 1)
- T_c : Time to compute SNB within a partition (VSNB MapReduce job 2)
- T_o : Time to compute SNB overlap between partitions (VSNB MapReduce job 3)
- T_u : Time to unload data from HDFS

However, T_l and T_u are independent of K , thus these times can be omitted from CM to decide the plan with the lowest runtime. Hence, it suffices to estimate the following runtime cost:

$$CM(p) = T_g + T_c + T_o \quad (5.2)$$

Next, we will examine T_g , T_c , and T_o .

T_g : Generate Blocking Key Values and Boundary Points To distribute the sorted records across N_k reducers, $(N_k - 1)$ boundary points must be chosen to define the boundaries between the partitions. T_g is the time needed to compute these points in the first MapReduce job of VSNB.

The goal of the first MapReduce job of VSNB is for every record, create a BKV, and store this to HDFS. Also, the items in the sample must be sent to the reducer. Therefore, VSNB uses a counter to send one of every $\frac{N_D}{s}$ records to the reducer.

MapReduce schedules by default one map task per HDFS block, so it is expected there will be N_B map tasks. MapReduce will run at most N_k map tasks concurrently (called a *wave*). The expected number of waves of map tasks is: $\left\lceil \frac{N_B}{N_k} \right\rceil$, and the time for each wave is $\left(C_k \cdot \frac{N_D}{N_B} \right) + C_{MR}$, where C_k is the average time to compute the blocking key for each record.

The expected time for each sample is $(C_s \cdot s)$, where C_s is the cost to move each sample plus per-record sorting time plus per-record scanning time cost. Thus,

$$T_g = \left\lceil \frac{N_B}{N_k} \right\rceil \cdot \left(C_k \cdot \frac{N_D}{N_B} + C_{MR} \right) + C_s \cdot s \quad (5.3)$$

T_c : Compute SNB Within a Partition The second MapReduce job in SMB-MR computes the sliding window within each partition, and is estimated to take T_c of wall-clock time. First, I will describe the ideal case without considering skew. Then, I will describe how to factor skew into the equation.

The typical time it would take for the second MapReduce job is, ignoring skew:

$$T_{c_{ideal}} = \frac{(C_b + C_M) \cdot N_D \cdot w}{N_k} + C_{MR} \quad (5.4)$$

where C_b is the per-record cost for blocking and C_M is the per-record cost for matching. In terms of time, the N_k reduce tasks run concurrently.

Obviously, there will be skew from non-ideally chosen boundary points: here I define *skew* as the ratio between the maximum number of records within any partition and the average number of records per partition. From [8], with probability $(1 - P_s)$, the skew γ is given by the following formula:

$$\gamma = \frac{-N_k \cdot \ln\left(\frac{P_s}{N_k}\right) + \sqrt{\ln\left(\frac{P_s}{N_k}\right)^2 \cdot (N_k)^2 - 2 \cdot \ln\left(\frac{P_s}{N_k}\right) \cdot s \cdot N_k + s}}{s} \quad (5.5)$$

As an example, consider the case of no skew: the value returned would be 1.0 and the computation of every partitions would take exactly T_c to compute. However, if one partition ends up with an 150% of the average number of records spread across all partitions, then the skew returned would be 1.5. Therefore as this reduce task has 50% more records than the average, this reduce task will take 50% longer than average. Since the overall MR job will take as long as the slowest reduce task, T_c will also increase by 50%.

Thus, adding this to the cost:

$$T_c = \left(\frac{-N_k \cdot \ln\left(\frac{P_s}{N_k}\right) + \sqrt{\ln\left(\frac{P_s}{N_k}\right)^2 \cdot (N_k)^2 - 2 \cdot \ln\left(\frac{P_s}{N_k}\right) \cdot s \cdot N_k + s}}{s} \right) \cdot \left(\frac{(C_b + C_M) \cdot N_D \cdot w}{N_k} \right) + C_{MR} \quad (5.6)$$

T_o : Compute SNB Overlap Between Partitions The third, and final, MapReduce job in VSNB computes SNB windows that cross a partition boundary. The expected time is:

$$T_o = (C_b + C_M) \cdot \left(\frac{(w-1) \cdot w}{2} \right) + C_{MR} \quad (5.7)$$

5.3.3 Calibration Procedure

To compute the optimal values for K , first the parameters in N must be calibrated. In this section I will discuss the method to estimate the parameters in N .

Estimating Cluster-specific Parameters To find n and N_k is straightforward: programmatically ask Hadoop for the current cluster configuration. To calculate C_{MR} , run a very short MapReduce job with a single map and single reduce task. Calculate the average time it takes to schedule and run a task as C_{MR} . The usable Java JVM Memory size J is taken from the Java parameter MaxHeapSize.

Estimating Dataset-specific Parameters First, N_D is the number of records in D , and B is provided, so $N_B = \frac{|D|}{B}$ where $|D|$ is the size of D in megabytes. Next, take a small sample of D as D_s . Using D_s , compute the average time to compute a blocking key value for each record C_k . Likewise, to compute the average time to take sample (C_s). Also, using the entire D_s , compute the average time per record to perform blocking (C_b) and matching (C_M).

5.3.4 Plan Search

With VSNB, there are two independent input parameters to search for: B and s .

Searching for B The naïve value for B is established directly via Equation 5.3, by ensuring that only one wave of map tasks are required, thus:

$$\frac{N_B}{N_k} = 1 \quad \text{and} \quad B = \frac{|D|}{N_B} \quad \text{thus} \quad B = \frac{|D|}{N_k} \quad (5.8)$$

However, values of B greater than the usable JVM memory size J will cause Java to crash. Therefore, we must force a number of waves of computation. Each wave needs to use all available cores N_k . Therefore we choose B to both fit in memory, and utilize all available cores N_k . Thus, B becomes:

$$B = \frac{|D|}{(\# \text{ waves}) \cdot N_k} \quad \text{thus} \quad B = \frac{|D|}{\left\lceil \frac{|D|}{J \cdot N_k} \right\rceil \cdot N_k} \quad (5.9)$$

Searching for s Once B has been established, then we compute s . Two factors impact s : the sampling time from T_g , and the extra skew from T_c . Therefore, to calculate s , minimize the sum of $(C_s \cdot s)$ from T_g plus all of T_c . The resulting objective function is:

$$\begin{aligned} \text{Objective} = & (C_s \cdot s) + \\ & \left(\frac{-N_k \cdot \ln\left(\frac{P_s}{N_k}\right) + \sqrt{\ln\left(\frac{P_s}{N_k}\right)^2 \cdot (N_k)^2 - 2 \cdot \ln\left(\frac{P_s}{N_k}\right) \cdot s \cdot N_k + s}}{s} \right) \cdot \\ & \left(\frac{(C_b + C_M) \cdot N_D \cdot w}{N_k} \right) \end{aligned} \quad (5.10)$$

Intuitively, the greater the cost of the blocking and matching functions $(C_b + C_M)$, the more the skew will affect overall runtime. To compute the s that minimizes Equation 5.10, find the minimum value of the objective function using a simple descent search between $s = N_k$ and $s = \frac{N_D}{2}$. Starting at $s = N_k$, increase s by multiples of N_k until the objective function is no longer decreasing.

5.4 Empirical Study

So far I have described the algorithm (Section 5.2) and the cost model (Section 5.3), now I move on to an empirical study of VSNB.

Dataset	Size (records)
Kaggle Movies Database	45K
One Million Songs Dataset	1M
DBLP	4.6M
FEC Individual Donors	25.2M
Users Database	36.4M

Table 5.3: Datasets used for VSNB empirical study

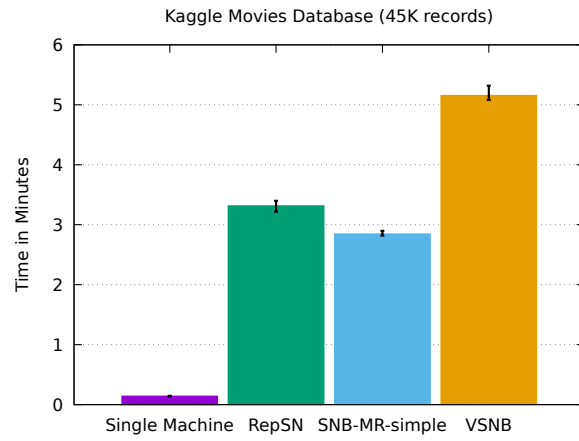
In this section I will empirically evaluate my single machine solution, RepSN, SNB-MR-Simple, and VSNB. I started with the data files as in Section 4.2 (see page 40), and added two additional datasets (a database of 45K movies, and a dataset of 1M songs). The datasets I used for the experiments on VSNB are shown in Table 5.3. Also, examples records from the datasets are available in Appendix A.1, and the experimental setting is described in Appendix A.3.

5.4.1 Overall Performance

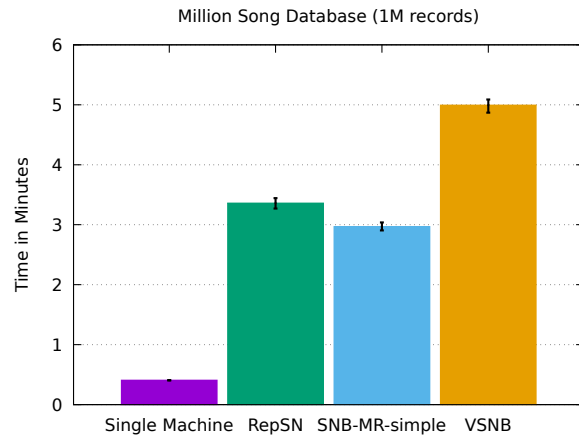
In this section I will investigate the performance of VSNB relative to the other methods.

VSNB’s Runtime on Large Datasets For large datasets, I will show that VSNB finishes faster than any of the other SNB solutions: single machine SNB, RepSN, or SNB-MR-Simple. To support this claim, I ran each algorithm on five datasets. For each dataset, I ran each algorithm ten times, removing the highest and lowest values, then averaging the remaining values. The graphs for each dataset are shown in Figure 5.9.

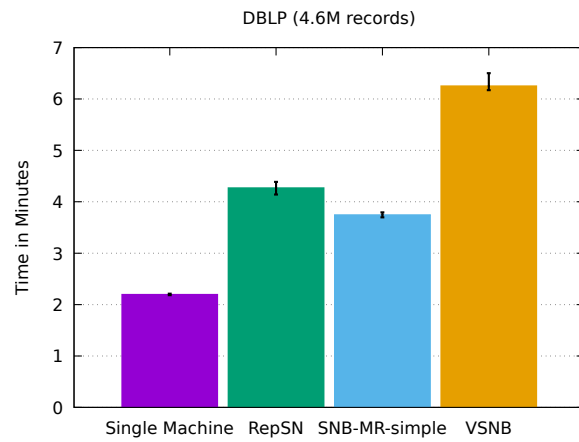
As expected, the results show that a single machine outperforms VSNB and all other methods for small datasets (less than 4.6M records). As can be seen in Figures 5.9a-5.9c, the single machine SNB solution is faster than all other methods by between 1.5 to 2.7 minutes (with the latter showing a single machine to be 20x faster than the next-best SNB-MR-Simple). VSNB is slower with those small datasets due to the extra overhead of MapReduce. However, in the largest datasets I used, a single machine becomes the slowest way to compute SNB (by 5.3 minutes in Figure 5.9d and 52 minutes in Figure 5.9e). At 36.4M tuples, VSNB computes SNB in the least time (12 minutes faster than SNB-MR-Simple and 33 minutes faster than RepSN).



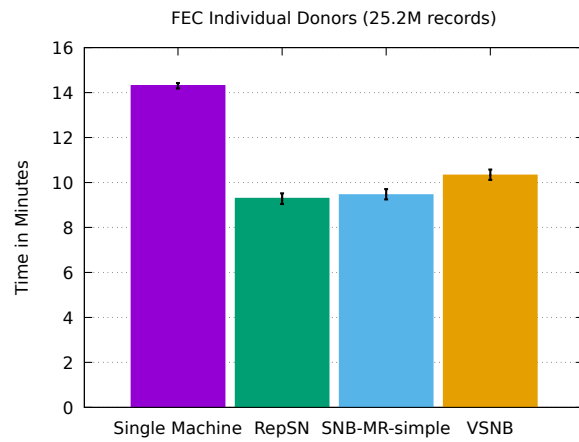
(a) Kaggle Movies



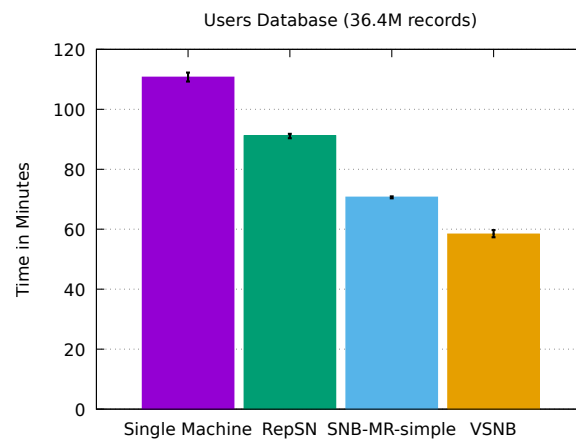
(b) Million Song Dataset



(c) DBLP



(d) FEC Individual Donors



(e) Users Database

Figure 5.9: Relative performance of each SNB method.

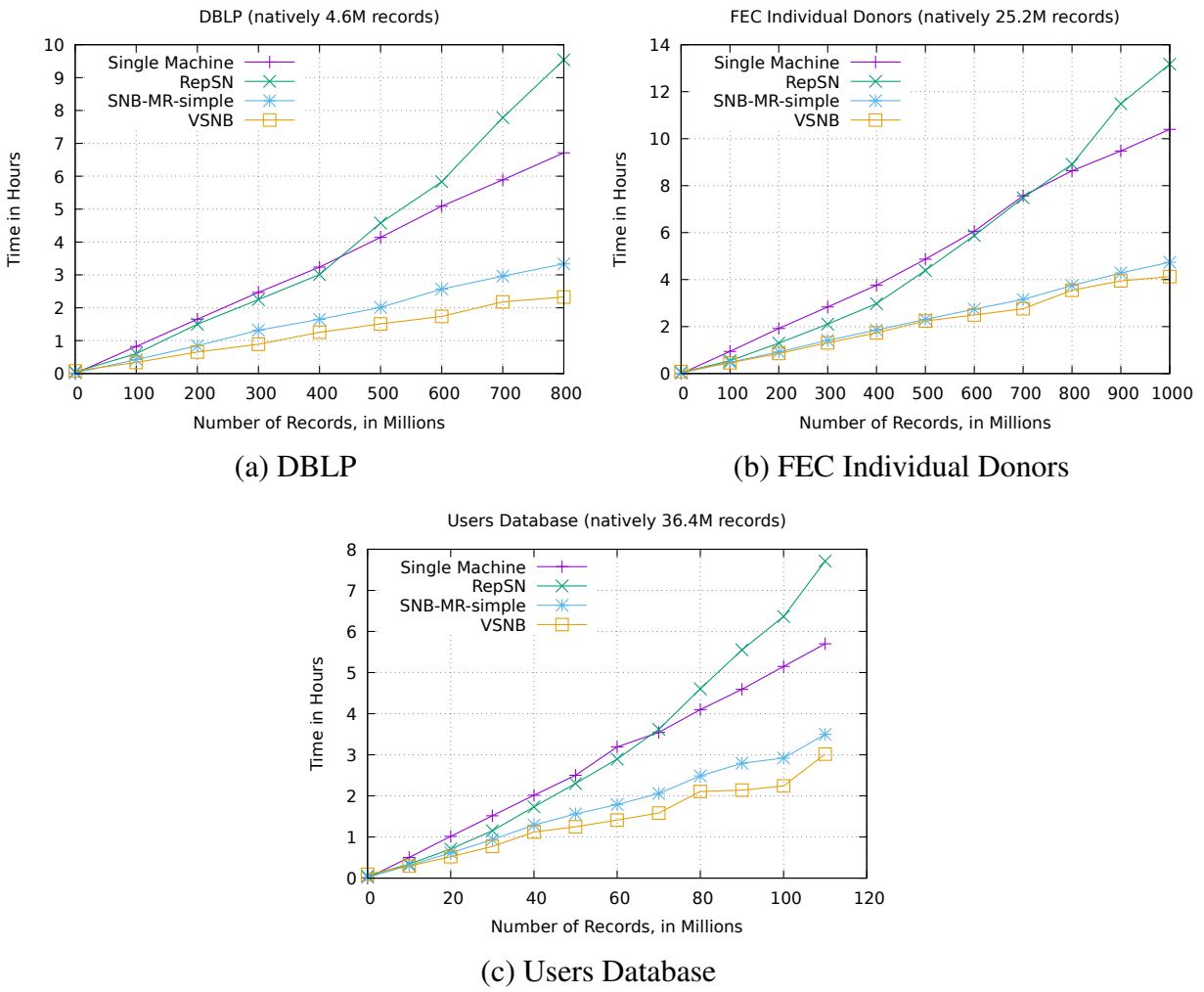


Figure 5.10: Relative performance of each SNB method, scaling up the number of records.

Performance as Input Data Grows With very large datasets, I claim that not only is VSNB faster, but the gap between VSNB, RepSN and Single machine grows. To examine this claim I looked at the trend of the time each method took as the dataset size was adjusted. To adjust the dataset size, I first wrote a per-dataset method to clone the dataset in order to be able to make a dataset of an arbitrary number of rows, where each row has a unique TID. Therefore, for each dataset, I wrote a method to prepend a *pass number* to the TID for each tuple. Then, I continued to cloning the dataset until the required number of tuples are in the source file. I ran each method against each number of tuples and graphed the results.

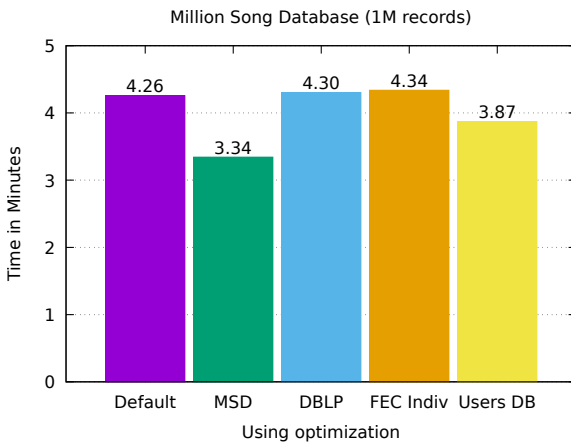
The results are in Figure 5.10. From those results, for large values (800M tuples, 1B tuples, 110M tuples) VSNB performed faster than the SNB-MR-Simple method by 30-60 minutes. However, for each dataset RepSN was three to four times slower than VSNB, and slower than the single machine solution (by between 45 minutes and nearly 3 hours).

Increase Dataset Size Finally, I will show that as the dataset sizes grow, all other methods other than RepSN will have linear speedup.

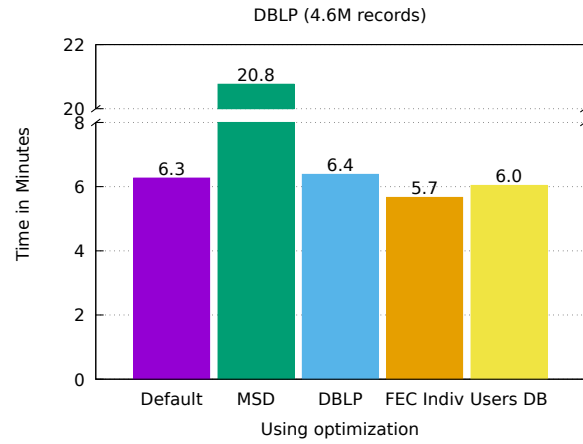
Going back to the graphs from Figure 5.10, after 400M tuples (DBLP), 500M tuples (Federal Elections Commission), and about 30M tuples (Users Database) the time RepSN takes grows by a superlinear amount relative to the size of the dataset. The decreasing scale up is caused by two reasons. First, as the input dataset grows in size, RepSN must make a larger KPM (which grows linearly with the input dataset). Second, map tasks cannot reuse the KPM computation, so each *wave* of map tasks is slowed down by the time cost to compute the KPM. Therefore, while the KPM is getting larger, there are more waves of map tasks, causing the time for RepSN to grow superlinearly. As the other methods do not compute a KPM, they do not have this issue and scale roughly linearly in the range I tested.

5.4.2 Effects of the Optimizer

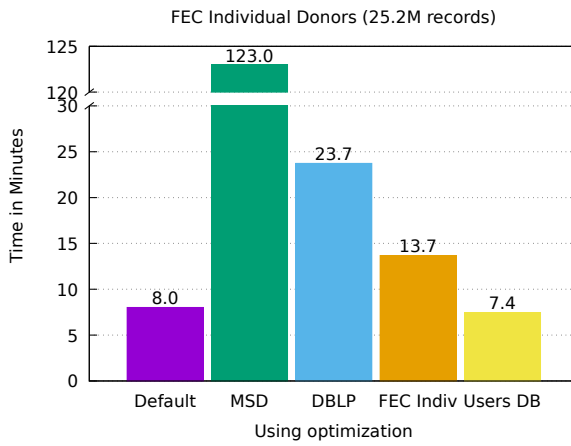
In Section 5.3, I proposed an optimizer to pick parameters for VSNB. This section will investigate these claims.



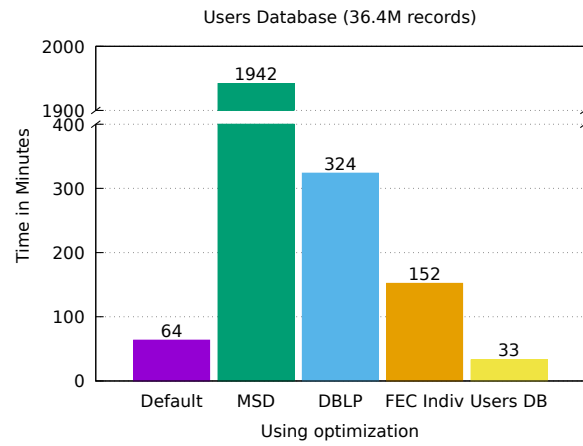
(a) Million Song Dataset



(b) DBLP

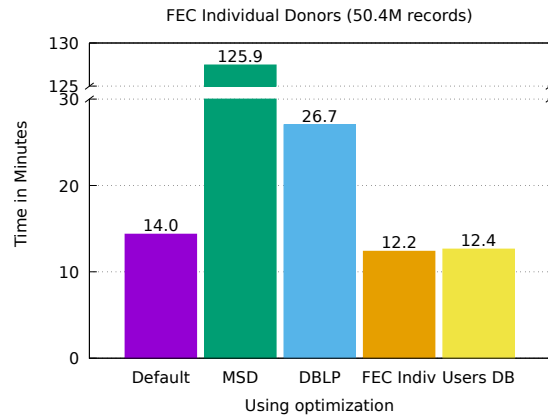


(c) FEC Individual Donors

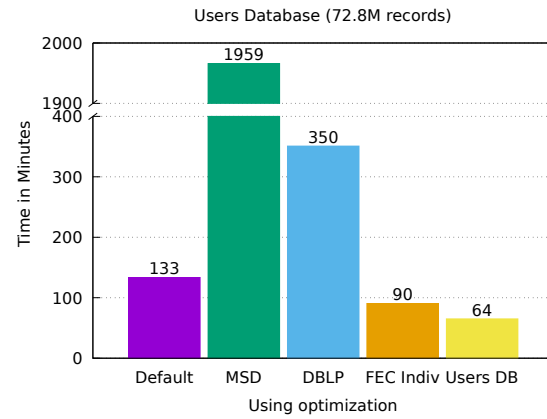


(d) Users Database

Figure 5.11: Cross optimization. Each dataset is run with a set of optimized plans.

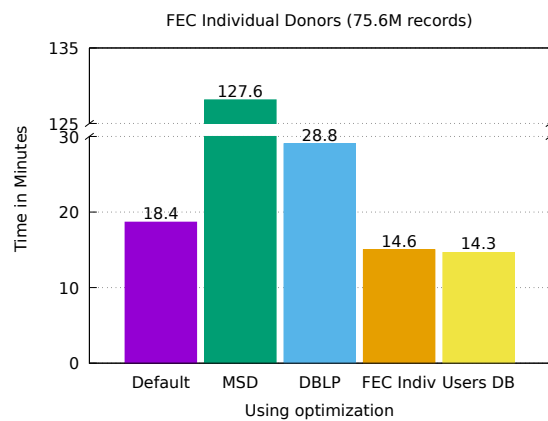


(a) FEC Individual Donors

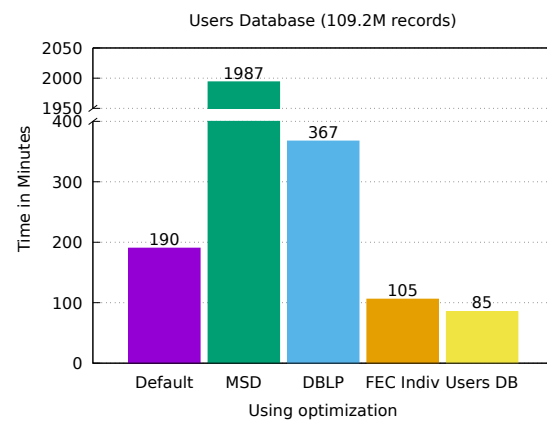


(b) Users Database

Figure 5.12: Cross optimization using double the source data. Each dataset is run with a set of optimized plans.



(a) FEC Individual Donors



(b) Users Database

Figure 5.13: Cross optimization using triple the source data. Each dataset is run with a set of optimized plans.

The Optimizer Picks a Good Plan I will show that the optimizer I have implemented for VSNB will select a good plan in the plan space. To examine this claim, I selected the four largest datasets from Table 5.3. Then, for each dataset, I optimized VSNB for that dataset, and saved the selected plan.

Intuitively, one way to investigate the performance of the optimizer is to examine the results of using the wrong optimization. To investigate this scenario, first I ran the optimizer for each of the four datasets, saving each result. Then, for each dataset, I executed VSNB with five *different plans*. I first ran VSNB using the default optimization (128MB block size, sample one out of every 1,000 records). Then, for each of four optimized plans, I executed VSNB using that optimized plan. I refer to this as *cross optimization*.

I ran each experiment 10 times, discarding the high and low values, and averaged the results. The results are in Figure 5.11. Notice that the Million Song Dataset (MSD) was the smallest dataset, in Figure 5.11a. While the optimizer picked a good plan, the overall difference is fairly minor (within 1 minute). The plan for MSD, however, was a poor choice for the rest of the datasets (Figures 5.11b through 5.11d.) where the default plan would have been a much better choice (by between 14 minutes and 1.3 *days*). In Figure 5.11b, optimizer picked a reasonable but not best plan (within 45 seconds or about 10% of the best plan). Moving on to Figure 5.11c, we can see that the optimizer picked a fairly bad plan for this dataset (off by about 7 minutes, or about 100% of the best plan and nearly that much to the default plan). Notice, however, the largest dataset is in Figure 5.11d. In this case the optimizer picked the best plan, at about half the cost of the default plan (33 minutes vs 64 minutes).

Optimizing Larger Datasets I now claim that as the dataset size increases the optimizer will pick a plan that executes faster than either the default plan or other optimized plans, and that gap will increase.

To examine how the optimizer handles larger data, I selected the two largest datasets (FEC Individual Contributions and the Users Database). For each of those datasets, using the method from Section 5.4.1 I created two new datasets — one dataset double the original size and one dataset

triple the original size. I then ran the same cross optimization procedure on the new datasets. The results are shown in Figure 5.12 for the doubled datasets, and Figure 5.13 for the tripled datasets.

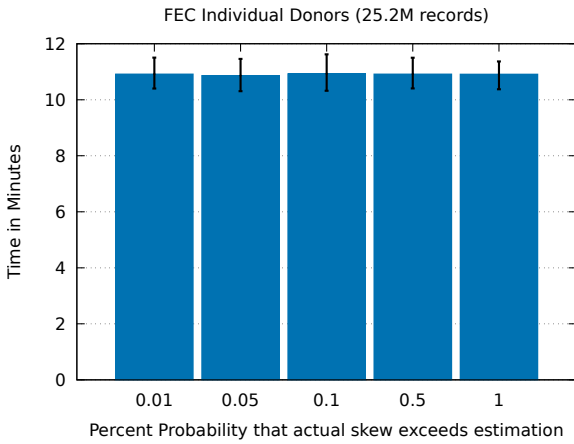
First, considering the FEC Individual Donors dataset, the relative time between the default plan the optimized plan went from 72% slower in Figure 5.11c to 14% faster in Figure 5.12a to 23% faster in Figure 5.13a. Similarly, the Users Database also saw an increase of performance relative to the default plan, from 93% faster in Figure 5.11d to 108% faster in Figure 5.12a to 124% faster in Figure 5.13a. In both Figures 5.12 and 5.13, the optimized-for plan is either fastest or second fastest (fastest in all except in Figure 5.12a where it is about 2% behind the plan optimized for the Users Database).

5.4.3 Sensitivity Analysis

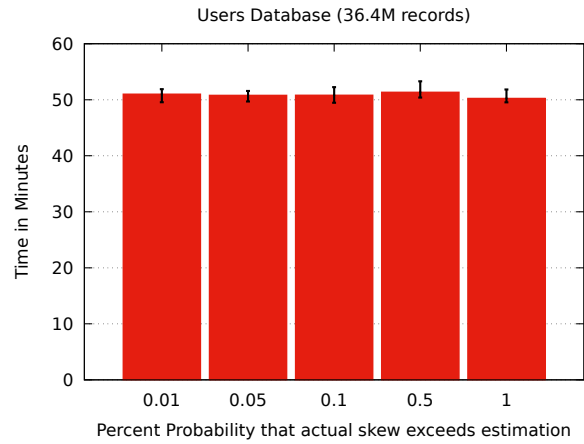
Now I move on to determining the sensitivity of the parameters to VSNB. In this section, note that these experiments were run *with* a matching function. I considered the case of no matching function, however just pairwise outputs from SNB is so fast that there was no discernible difference. Thus, to amplify the effect of the sensitivity analysis, I used appropriate matching functions (see Appendix A for examples of the matching functions).

P_s Does not Drastically Affect Runtime I will show that the exact choice of P_s (the probability that the actual skew will exceed the estimated value) does not have a drastic effect runtime. At the heart of VSNB (and many other MapReduce sorting algorithms), is a probabilistic sort as described in Section 5.3.

To investigate this claim, I again took the two largest datasets from Table 5.3. Here I ran VSNB setting the probability of skew P_s to a series of values: (0.01, 0.05, 0.1, 0.5, 1.0). For each dataset, I ran VSNB adjusting the probability. For each adjustment of the probability, I ran VSNB 10 times, discarding the high and low values, and averaging the remaining values. The result was that the runtime of the algorithm using each of the series of values was substantially the same, as shown in Figure 5.14. This shows that altering the probability of actual skew exceeding estimated value does not drastically effect runtime.

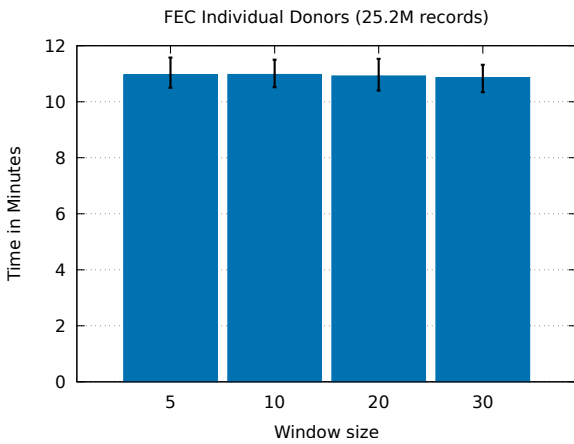


(a) FEC Individual Donors

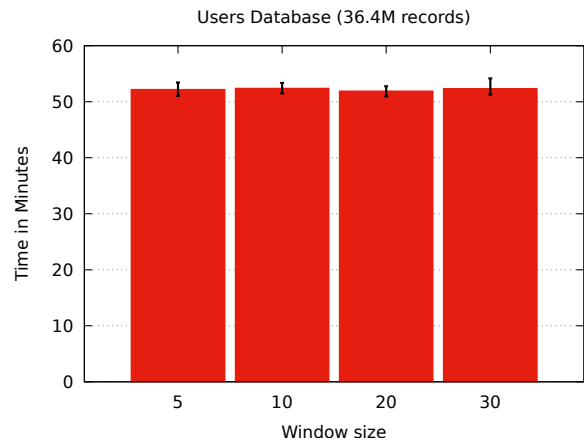


(b) Users database

Figure 5.14: VSNB Sensitivity Analysis: adjust probability of skew



(a) FEC Individual Donors



(b) Users database

Figure 5.15: VSNB Sensitivity Analysis: adjust window size

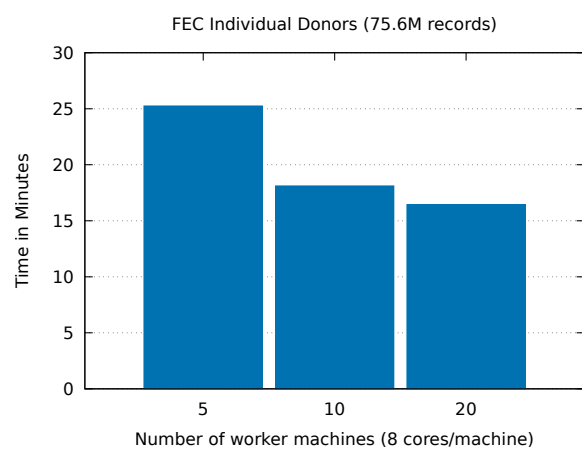
As I showed in Section 5.3, from [8] we leverage a formula to determine the number of items should be in the sample in order to have skew less than a particular amount. Specifically P_s is the probability that n items in the sample will exceed skew s , thus lower should result in less skew. Note that in the experiments from Section 5.4, I used a static probability of 0.1%.

Window Size For relatively minor changes in window size, I claim that the performance of VSNB will be stable.

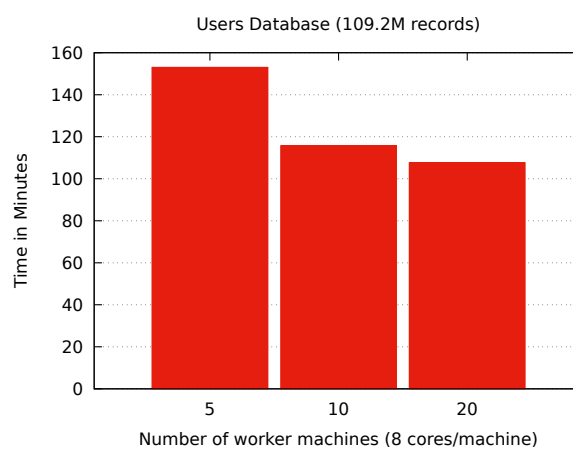
To investigate this claim, I once again took the two largest datasets from Table 5.3. This time, I ran VSNB setting the window size w to a series of values: (5, 10, 20, 30). For each dataset, I ran VSNB adjusting the window size. For each adjustment of the window size, I ran VSNB 10 times, discarding the high and low values, and averaging the remaining values. The results were once again that despite the window size, the overall runtime was substantially the same, as shown in Figure 5.15. Therefore, the runtime of VSNB is not affected by minor adjustments to the window size.

Cluster Size Finally, I claim that altering the size of the cluster of worker machines for MapReduce will increase performance in a reasonable manner.

To investigate this claim, I once again took the two largest datasets from Table 5.3 and tripled their size. In this set of experiments, I first adjusted the number of worker nodes in the cluster through a series of values: 5, 10, and 20 cluster nodes. After adjusting the worker nodes, I recreated the HDFS datastore to ensure it was healthy. Then, for each dataset, I ran VSNB 10 times, discarding the high and low values, and averaging the remaining values. The result in Figure 5.16 shows that as the number of machines is increased (from 5 to 20) the time decreases (by 9 and 45 minutes). As VSNB counts the time it takes to load the dataset from a single file on disk into HDFS, this is a substantial and fixed cost. Therefore, I conclude the runtime of VSNB gains speedup from additional cluster nodes.



(a) FEC Individual Donors



(b) Users database

Figure 5.16: VSNB Sensitivity Analysis: adjust number of worker machines

Chapter 6

Analyzing Current Adaptive SNB Solutions

In this chapter I will describe existing adaptive SNB methods. First, in Section 6.1, I describe the motivation for adaptivity in SNB. Next, in Section 6.2, I examine two single-machine methods for adaptive SNB proposed in the literature. Finally, in Section 6.3, I examine a recently published MapReduce method for adaptive SNB.

6.1 Motivation for Adaptive SNB

As I discussed in Section 2.1.1, traditional SNB has a limitation: the fixed window size. If, during the SNB execution, a dense area of matching records spanning a distance wider than the SNB window length is encountered, some record pairs will not be examined. Recall that as with other blocking methods, SNB is a *filtering* process. Thus, by not examining certain record pairs, those pairs will be completely missed (effectively marked as non-matching). Thus, some actually-matching records will not be returned, hurting accuracy. There are two general strategies to address dense areas of matching records.

First, in traditional (vanilla) SNB, the main strategy to address this issue is to set the window size to always be longer than the largest possible set of matches. The challenge is how to know what the largest set is, either *a priori* or *a posteriori*. *A priori*, the end-user might not have a clear understanding of the dataset they face. Then, even after the SNB method has been executed, determining if there are missing records is non-obvious. Since the records were not compared during the SNB run, it is completely unknown that the record pairs were missed. To compensate, in vanilla SNB, an overly-large window size must be chosen with enough of a safety buffer to

ensure all actually-matching record pairs are found. This safety buffer in the window size will cause many more comparisons with non-matching record pairs. As these non-matching records are matched, that will hurt performance.

The second method is to adaptively expand the window size — make the window bigger only as necessary. When a dense area of records is found, expand the window size until it no longer finds matching records. Therefore, the end-user will not be required to know the largest set of actually-matching records to establish the window size. He or she will only provide a reasonable minimum-sized window, and that window size will be lengthened as necessary. The extra comparisons only are encountered in areas of dense matches, saving performance.

In the rest of this chapter, I will describe the current solutions to adaptively adjust the window size.

6.2 Single-Machine Adaptive SNB

In this section I describe two methods for adaptive SNB on a single machine.

6.2.1 Duplicate Count Strategy (DCS++)

The current state-of-the art adaptive SNB method for a single-machine is Duplicate Count Strategy (DCS++) [11]. DCS++ uses a matching function to measure the density of matching records within the current window. If the current window has a density of matches greater than a threshold, DCS++ expands the window. After expanding, DCS++ re-evaluates the window's density, further extending as necessary. This continues until no further matches are found, and the density falls below the configured threshold. Finally, DCS++ performs a transitive closure over all record pairs found, to find any other matching pairs.

As with traditional SNB, DCS++ takes as input a set of records, a window size w , and a BK function. However, DCS++ requires two other inputs: a dataset-specific matching function M , and an density threshold ϕ where $0 < \phi \leq \frac{1}{w-1}$. DCS++ will output a set of record pairs that are determined to be matches.

Proposition 3.1 in [11] shows that DCS++ is transitive as long as $\phi \leq \frac{1}{w-1}$. The intuition is that when DCS++ finds a match, DCS++ always extends the window by *at least* as much as had DCS++ started a new window with the matching record. Since transitivity is a key part of DCS++, the windows would have been merged, so instead, DCS++ extends the current window.

The DCS++ method is as follows. First, DCS++ sorts the records per the BK. Then, DCS++ starts the process of sliding a window through the dataset. At each position j in the dataset in sorted order, DCS++ resets the window size to w . DCS++ initializes the number of matches m and the number of comparisons c to zero. DCS++ loads records at position j (r_j) through the position $j + w$ ($r_{(j+w)}$) into the sorted window W . The current window offset k (the distance from the starting record) is set to zero.

Until k is longer than the size of W , the DCS++ will perform the following. First, DCS++ increases k by one. Then, DCS++ calls $M(r_j, r_{(j+k)})$, and increases c by one.

If M returns true, then DCS++ has found a match.

DCS++ will adjust the window size as follows.

First, increase m by one to account for the new match. Next, DCS++ will add pair $(r_j, r_{(j+k)})$ to the output. After that, DCS++ will make sure the next $(w - 1)$ records are part of this window. Thus, DCS++ will add records $r_{(j+k+1)}$ through $r_{(j+k+(w-1))}$ to the window (if they are not already in the window).

For example see Figure 6.1. In Figure 6.1, the window size is 5. As records number 1 and 3 match, 4 through 7 need to be part of this window. Now, as records 4 and 5 are already in the window, no action is taken with them. Thus, DCS++ adds records 6 and 7 to the window.

Finally, if r_j and $r_{(j+k)}$ match, DCS++ adds $r_{(j+k)}$ to a *skip list*. Note carefully that if a window of length w was started at record $r_{(j+k)}$ it will extend until $r_{(j+k+(w-1))}$. Those records are now

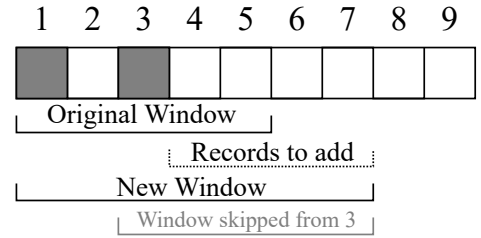


Figure 6.1: Window expansion in DCS++: $w = 5$. The shaded records match, so the next 4 records need to be part of the window. Thus, 6 and 7 are added to window.

added to the window starting at r_j , so the computation of a window at $r_{(j+k)}$ is not necessary. DCS++ adds $r_{(j+k)}$ to a *skip list*, so it will skip that window position in the future.

For example, once again see Figure 6.1. Imagine the SNB computation moved to record 3. The record in position 3 matches position 1 and all of the records that would have been in the window starting at position 3 have already been evaluated (as part of the window starting at position 1). Since matching is assumed to be transitive, all records that would have matched position 3 matched position 1, and were output as matching the record at position 1. All records matching starting at position 3 will be found through the transitive closure. Thus, evaluating the window at position 3 is redundant, and therefore unnecessary.

Then, DCS++ increases k by one, and repeats the matching process. When k reaches the length of W , DCS++ evaluates if the density $\frac{m}{c} \geq \phi$. If true, then DCS++ will expand the window by one record, and repeats the matching process. The loop continues until $\frac{m}{c} < \phi$ or the end of the dataset is reached.

If DCS++ is not at the end of the dataset and $\frac{m}{c} < \phi$, the window is then closed. DCS++ adds one to j (sliding the window one position), resets c , m , and k to zero, and trims the current window to length w . Then, DCS++ looks to see if j is in the skip list. If so, then DCS++ moves the window forward by removing r_j from the window, and adding $r_{(j+w+1)}$ to the window. DCS++ then increases j by 1. This repeats until DCS++ finds a record position not in the skip list. After finding a record position not in the skip list, DCS++ then repeats the matching process for the new window at that position.

This entire pattern continues until j reaches the end of the dataset.

When all the matching is completed, DCS++ performs a transitive closure over the found pairs of records. This transitive closure has several practical effects. First, if two matches are found within a window $(r_a, r_m), (r_a, r_n)$, a third match is generated through transitivity: (r_m, r_n) . Second, if the matching function isn't perfectly transitive, a record might match multiple windows, for example: $(r_a, r_m), (r_b, r_m)$. In that case the windows will be *combined* through transitivity.

Discussion There are two issues with DCS++. First, DCS++ is only a single-machine solution. Furthermore, with the expanding and contracting of the window, it is not clear how to directly

extend DCS++ to use a cluster of computers efficiently. For example, it is possible that a window might extend through the *entire* dataset! I will examine a MapReduce method proposed in the literature in Section 6.3, and introduce my solution in Chapter 7.

Second, for large datasets with large windows, the working set of records needing to be examined may exceed memory capacity, and thus will cause DCS++ to potentially read records from disk multiple times. I will also address this issue in Chapter 7.

6.2.2 Adaptive Sorted Neighborhood Method (A-SNM)

In the literature, another single-machine solution has been proposed: Adaptive Sorted Neighborhood Method (A-SNM) [36]. The key idea of A-SNM is, after sorting, to split the sorted list into *non-overlapping* blocks by using a distance function on the fields used for sorting. The intuition is that, if the blocking has been done perfectly, all actually-matching records will be directly adjacent, and the only goal is to discover the pairs of records where the blocking fields differ substantially as those will be the *boundary pairs* between blocks.

A-SNM takes as input a set of records, a BK function, a distance function d . Output from A-SNM are a set of non-overlapping sets of potentially-matching records. A-SNM needs to calculate a distance threshold ϕ such that if $d(r_1, r_2) \leq \phi$ then the records are in the same block.¹ If $d(r_1, r_2) > \phi$ then the records belong to two different blocks.

At a high level A-SNM starts with a window size of $w = 2$. A-SNM then compares $d(r_1, r_w)$ where w is the length of the current window. If $d(r_1, r_w) < \phi$, the window is enlarged, if $d(r_1, r_w) > \phi$ then the window is retracted.² When $d(r_1, r_w) \approx \phi$, then the window is complete, and a new window is started at the record r_{w+1} . This continues until the end of the dataset is reached.

There are several challenges with A-SNM making it unattractive for an adaptive SNB method. First, it remains unclear how ϕ should be established on a dataset of any reasonable size.³

¹It is not clear exactly how ϕ should be set in practice, in [36] the authors stated they used a full *quarter of the dataset* to compose a validation set to tune ϕ .

²[36] proposes a set of incremental, linear, geometric, and a combination of methods to adjust the window size. I direct the reader there if additional detail is desired.

³The datasets used in [36] (7.8K records, 214K records, and 1.4M records) are significantly smaller than the large datasets used in this work.

Second, A-SNM shifts much of the burden to the end-user. The choice of a new distance function is important, and difficult to establish. Additionally, in A-SNM the choice of BK is now more critical, and less adaptable to record pairs that may have identical *BKV*s, yet do not match. Furthermore, the end-user will be expected spend significant time tuning ϕ , with unclear feedback from A-SNM as to how to make good choices.

6.3 MultiMR-DCS++: A MapReduce Adaptive SNB

In this section I will describe MultiMR-DCS++, a MapReduce-based adaptive SNB method. In work developed somewhat currently to this work, MultiMR-DCS++ has been proposed in [27]. This work is an extension of both A-SNM to use a distance function to find blocks, as well as DCS++ to compute adaptive SNB *within* each block. To distribute the computation in MapReduce, MultiMR-DCS++ uses a Partition Matrix and data routing approach similar to RepSN.⁴

MultiMR-DCS++ takes as input a set of records, a BK function, a distance function d , a distance threshold $\phi_{\max}$, and a distance interval value α (which MultiMR-DCS++ uses to calculate $\phi_{\min} = \phi_{\max} - \alpha$). MultiMR-DCS++ outputs a set of matching record pairs.

At a high level, MultiMR-DCS++ is a set of three MapReduce jobs. The first MapReduce job in MultiMR-DCS++ randomly distributes the records to the reducers, and each reducer sorts its records and produces a materialized sorted run. The second MapReduce job calculates the similarity between these sorted runs.⁵ The third MapReduce job sends the sorted runs to a reducer, along with a copy of all the subsequent sorted runs that are within $\phi_{\min}$ similarity, so that the reducer will have all similar records to perform DCS++. The reduce task then calculates DCS++ within records sent to it.

Let us look into MultiMR-DCS++ in a bit more detail. In the first MapReduce job, each map task calls the BK to produce a *BKV* for every record in the map task. MultiMR-DCS++ randomly

⁴As we saw with RepSN in Section 3.5, MultiMR-DCS++ also handles multiple BK functions in a single execution of the workflow. As this is a straightforward extension by adding a tiny bit of metadata to track each individual BK “pass” over the dataset and sort it correctly; it adds a cumbersome level of detail to describe. I omit it here and direct the reader to [27] for additional detail.

⁵Please note that [27] conflates a set of sorted runs with a sorted order that has been partitioned. Unfortunately, this has rather profound negative implications that I discuss at the end of this section.

distributes the enriched records to the reducers. Next, each reducer materializes a sorted run to HDFS, named for the number of this reducer. Also, the reduce task outputs the first record with its partition number, which is input to the next MapReduce job.

The second MapReduce job in MultiMR-DCS++ takes the output from the first MapReduce job (the first record of each sorted run) to create the Multi-pass Partition Allocation Matrix (Multi-PAM). In the second MapReduce job, each map task takes as input one set of output from the first MapReduce job (the first record of each sorted run). The map task assigns each record from the input tuple (`record`, `partition_number`) to each reducer where the reducer number is greater than the `partition_number`. The MapReduce partitioner partitions the tuples based upon the destination reducer. The reduce task compares each record to see if the similarity is greater than $\phi_{\min}$. If the records are similar, then MultiMR-DCS++ considers their sorted order partitions similar. In that case MultiMR-DCS++ adds a 1 to the MultiPAM for the pair of sorted order partitions to note that the first sorted order partition requires the second sorted order partition to compute DCS++, thus they must be co-located.

In the third MapReduce job in MultiMR-DCS++, each map task first loads all of the MultiPAM. The map task determines which reduce tasks will receive *this entire sorted run*. For each record in the sorted run, the map task sends to the partitioner a copy of the record for each reduce task. The partitioner partitions the records based upon the destination reducer. Each reducer processes DCS++ within the records it receives.

6.3.1 Discussion

There are two main issues with MultiMR-DCS++.

No Order Between Sorted Runs First, note that MultiMR-DCS++ materializes a set of *overlapping* sorted runs in the first MapReduce job. In [27], the authors refer to these as if they were an entire sorted order that was partitioned — sorted orders that are partitioned is a common strategy with SNB methods using MapReduce. Unfortunately, MultiMR-DCS++ does not create a fully sorted order prior to partitioning (as we saw with RepSN in Section 3.5). To compute DCS++ in

this manner, MultiMR-DCS++ needs the next *partition* of records — not an overlapping run — to ensure the correct records are available for matching.

This causes a set of problems. First, these overlapping sorted runs have no inherent order *relative to each other* — they are exactly like sorted runs in a database sort merge join. As the sorted runs have no inherent order relative to each other, with a high likelihood, every sorted run will need *every other* subsequent sorted run to compute DCS++. Consider that the first record in each sorted run is the *smallest*, and thus each are likely to match the other.

Consider the consequences attempting to use $\phi_{\min}$ similarity amongst the first records in each run. For example, the beginning and end of the dataset could be a set of records that are non-matching, but some set of records in the middle are actually-matching records. MultiMR-DCS++ will erroneously determine that none of the sorted runs are within $\phi_{\min}$ and thus DCS++ will only be executed within the sorted runs. This will cause MultiMR-DCS++ to miss all actually-matching tuples that are not co-located in the same sorted run!

Next, consider the case where all of the sorted runs are within $\phi_{\min}$. MultiMR-DCS++ will consider all the sorted runs to be matching. Since they are all matching, in the final MapReduce task, the first reducer will be sent all the tuples. The second reducer will be sent the tuples from the second sorted order partition onwards (randomly missing the tuples from the first partition). The third reducer will be sent all the tuples from the third sorted order partition onwards (randomly missing the tuples from the first and second partitions). This pattern continues to the end, where the last partition receives only tuples from the last sorted run! This creates one full DCS++ run at the first reduce task, and a set of partial DCS++ runs at the subsequent reduce tasks, where data is increasingly missing the further down the task list you go. This is a major performance problem.⁶

In other words, effectively the final MapReduce reduce task for the smallest partition number is a single-machine DCS++ computation *over the entire dataset*. Every other reduce task is a DCS++ computation with missing data. The potential exists that additional matches might be found this way. However, in real-world scenarios, the dataset must have been selected to be amenable to a

⁶The reader is directed to Figure 11 from [27], where the performance of MultiMR-DCS++ is consistent to (albeit marginally better than) RepSN. When compared with the graphs in Figure 5.10 in Section 5.4, if the performance is consistent with RepSN, MultiMR-DCS++ will not scale.

SNB method. Thus, given that a SNB method was chosen, the DCS++ runs with data missing are highly unlikely to return valuable results.

A potential fix for this would be to bolt MultiMR-DCS++ onto RepSN directly: first sort the dataset as in RepSN, find partitions of the sorted order as in RepSN, *then* calculate which partitions will be necessary per MultiMR-DCS++, *then* calculate DCS++. However, at best, this adds another MapReduce job to RepSN, thus that would make MultiMR-DCS++ always slower than RepSN.⁷

Inconsistent with DCS++ Finally, [27] claims that since A-SNM also uses a similarity threshold to cutoff to end the block, the DCS++ comparisons can be stopped at the end of the block using a watered-down distance function (watered down by the distance interval α). DCS++ itself, however, does not limit the computations in this manner.

Let us assume the algorithmic challenges mentioned above have been worked out. It is true that if the distance measure correctly determines two records are not similar, DCS++ will be unlikely to find another actually-matching record pair that spans that point. However, this relies heavily upon the end-user to supply a dataset-specific distance measure function d , and distance interval α . These are both non-trivial items for the end-user to produce, and inaccuracies will have substantial and non-obvious negative impacts upon MultiMR-DCS++. For these reasons I will not consider this work further here.

⁷Unless the gains from DCS++ are so substantial as to overcome the additional overhead from another MapReduce job, which is unlikely.

Chapter 7

Developing an Adaptive SNB Solution on MapReduce (ASNB)

In this chapter I develop a new solution for computing SNB with adaptively-sized windows on MapReduce, which I call Adaptive Sorted Neighborhood Blocking (ASNB). In Section 7.1, I explore the challenges for adaptive SNB strategies in a distributed environment. In Section 7.2, I describe my solution, ASNB, and its implementation in MapReduce. In Section 7.3 I describe the cost model for ASNB. Finally, in Section 7.4, I provide empirical results demonstrating the promise of ASNB.

7.1 Challenges for Adaptive SNB in a Distributed Environment

As I discussed in Section 6.1, a limitation of SNB is that with a fixed window size, SNB misses areas of densely-matching records. In this section I will introduce ASNB, a new method that extends DCS++, the current state-of-the-art in adaptive SNB, into MapReduce. I described DCS++ in detail in Section 6.2.1. Importantly, DCS++ is a single machine solution for adaptive SNB. To scale up adaptive SNB from a single-machine solution by using a cluster, there are two challenges: how to compute adaptive SNB in a single pass, and how to distribute adaptive SNB computation across a cluster. I first address these challenges in Section 7.1.1, and then go into the details of ASNB in Section 7.2.

7.1.1 Single-Pass Adaptive SNB Computation

At the end of Section 6.2.1, I discussed one of the challenges with DCS++: it computes one window at a time. In DCS++, a window can be started at every position in sorted order. When

DCS++ moves to the next position to start a new window, the window size is reset (because this is now adaptive SNB and the window may have been extended). When the window size is reset, extra tuples are removed until the window size is down to the minimum. Those tuples will be needed again as the window keeps moving forward through the dataset. However, this means that each record in the dataset may be added and removed from the window multiple times.

As an optimization, when the first record in a window (r_{head}) matches a record in the window (r_i), the window is extended by an additional window size (including that record, so records $r_{(i+1)}$ through $r_{(i+w-1)}$). As records $r_{(i+1)}$ through $r_{(i+w-1)}$ are now contained within the window that starts at r_{head} , r_i is added to a *skip list* to prevent DCS++ from re-calculating the window at position r_i . In other words, all of the records in the window starting at r_i have already been compared to something that matches r_i – the record r_{head} .

The DCS++ method evaluates exactly one window, then moves to the next window. Thus, it can be said that DCS++ computes SNB *window-at-a-time*. Window-at-a-time computation works well for small datasets. However, if the memory of the machine is limited and/or the dataset is large and highly-matching, some or all of the dataset will need to be spilled to disk. Records of the dataset that are spilled to disk will, naturally, need to be read back in before they can be used again. Thus, the drawback of window-at-a-time is that it can be dependent upon reading and writing to disk, instead of keeping everything in memory. If memory is very tight, the same records may need to be read in *many times*.

I will now introduce a new method for computing adaptive SNB *record-at-a-time*, with a single scan through the dataset. The intuition behind record-at-a-time is that instead of having one window open, we will keep a *set* of windows open.

At a high level, record-at-a-time operates as follows. For each record r in the dataset, either match r with one or more already existing windows. If r matches more than one window, merge them together. If r matches zero windows, create a new window for this record. Then, before moving to the next record, evaluate if any of the currently open windows are at least as long as the initial window size and does not have the density of matches to support extending the window further. If the window no longer has sufficient match density, or adaptive SNB is at the end of

the dataset, then the window is no longer viable to add new records to it. When a window can no longer have records added to it, it is finished and referred to as *closed*. Within the window, all matches are considered transitive. Thus, when the window is closed, the cross-product of matching record pairs found by this window is written as output.

Intuitively, this is similar to how windows are handled in the non-adaptive SNB methods I have described (see Section 4.1). In adaptive SNB, however, a window can have arbitrary length. Therefore to handle windows of different lengths, instead of a single queue (as in Section 4.1), we have a set of open windows.

One concern is the amount of memory required to maintain the set of open records. First, recall that the goal of **ASNB** is to return a set of matching TID pairs (and not the records themselves). Therefore, again assuming all matches are transitive, all records within a window will match the first record, or the *head* record, for the window. This means that for each window, only the head record needs to be fully retained. For every other record within the window, it is only necessary to retain the TID.

Thus, the memory requirements of the set of open windows would be expected to be substantially smaller than available memory in contemporary systems, for most reasonable scenarios. While not described further here, in cases where the set of open windows are not able to be retained in memory, some or all of the open window datastructures could be written to the local disk. If the entire window datastructure would need to be written to disk, then the performance would be the same as DCS++. However, as we will see in Section 7.1.2, there are benefits even in this case.

Single Machine Adaptive SNB To evaluate my adaptive SNB claims, I developed a single-pass adaptive SNB method for a single machine. This method is a straightforward extension of **SNB-single** from Section 4.2 (on page 4.2), by altering the final SNB computation exactly as described above.

7.1.2 Distribute DCS++-style Computation

Next, we consider how to compute adaptive SNB, as described above, by partitioning the dataset across a cluster of computers. We will extend how we compute VSNB as described in Section 5.2 to compute adaptive SNB.

The key insight from prior work’s description of DCS++ [11] starts with the requirement that the density function ϕ must be $\phi \leq \frac{1}{w-1}$. As a result, when an adaptive SNB neighborhood window expands due to a match, the window must always expand by *at least* by $(w - 1)$ positions from the matching record. In other words, when a matching is found, the window is extended by at least as far as a new window from that record.

When computing adaptive SNB in a clustered partitioned environment, some windows will always be near enough to a partition boundary, and will span two partitions, just like as in VSNB in Section 5.2. While VSNB handled the cross-partition windows with a single “fix-up” MapReduce job, in the adaptive SNB setting, windows can be of arbitrary length, including the worst case: a window that spans the entire dataset.

My contribution here starts with the observation that as shown in [11], since $\phi \leq \frac{1}{w-1}$, when the window expands, it expands by at least one window size. In [11] that information is used to motivate the skip list — the other window does not need to be evaluated as it will be redundant. Now, consider the situation in reverse: two head records from different windows match, and their windows overlap (some records are common between the two windows). In the DCS++ case, this situation does not occur — the second window is never created because the head record is on the skip list and the contents of the second window are part of the first window. In the distributed case, however, the work to compute the second window is already done, and *later* finds that the windows overlap. The solution is to *merge* the second window into the first window.

For example, imagine two windows in a distributed setting. Window a is open at the end of partition one. Window b is at the very first record of partition two. The head records for both a and b match. Had adaptive SNB been computed on a single machine, as Window a was being computed, the record that starts b , and all of the entries from the-now-not-created Window b would have been incorporated into a . With distributed adaptive SNB, we have two windows that overlap

and match, thus we merge b into a . After merging, $\frac{\text{combined number of matches}}{\text{combined number of comparisons}}$ for Window a requires Window a to be expanded beyond the end of both Windows a and b . Thus, distributed adaptive SNB may be required look even further to see if the now-expanded a matches with any other partially-computed windows.

A clear concern is the number of merging rounds that will be required, as each merging round becomes an extra MapReduce job. First, note that for an adaptive SNB computation to take many subsequent rounds, *very* large windows must exist, consuming an increasingly large amount of records per window per each additional round. The worst-case scenario is the case where a single window spans all partitions. In order for a window to span all partitions, the window will need to match a minimum of $\lceil \phi N_D \rceil$ records for a window that starts from the first record in the dataset. This single window will require a substantial fraction of all records in the dataset to match. That said, very large windows are antithetical to making the choice of SNB for a blocking strategy. In SNB-style strategies, records close to each other are more likely to match. As the distance grows, the probability that a record pair will match should decrease exponentially.

That decrease in probability motivates the expected case. Consider the case where every partition has an exactly equal number of records N_P . In order for a window to be longer than one partition, the window would need $\lceil \phi N_P \rceil$ matches to extend N_P positions. In common entity integration workloads, the maximum number of duplicated records is less than $\lceil \phi N_P \rceil$ (typically far less). Thus, it is expected that a single window merging MapReduce job will be required to handle the windows that were initially found near the end of a Partition. Since no windows are expected to be longer than N_P positions, all windows will close during the first merging round, and thus only a single merging MapReduce job is required.

7.2 Introducing ASNB

Now I will discuss my method to distribute adaptive SNB across a MapReduce cluster, called ASNB. ASNB takes as input a dataset D which is a file of records on the local filesystem of a single machine. ASNB also requires the following SNB-specific variables: a starting window size w , a blocking key function BK , a matching function M , and a matching density function ϕ where

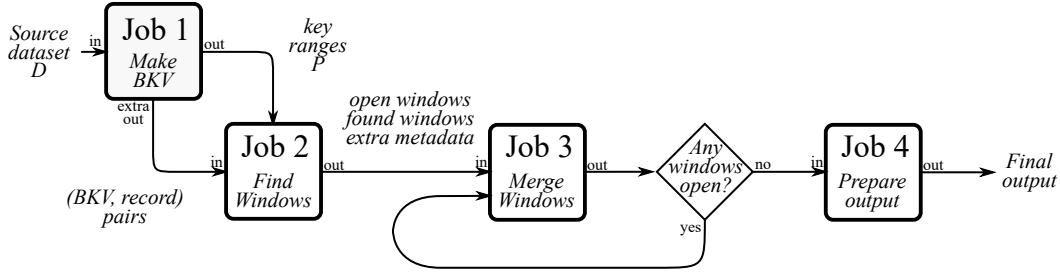


Figure 7.1: ASNB overview

$\phi \leq \frac{1}{w-1}$. ASNB will output to the local filesystem on a single machine a file containing a set of matching TIDs from D , in an efficient manner.

7.2.1 Overview

ASNB has four phases, which correspond to four different MapReduce jobs, as illustrated in Figure 7.1.

1. Use a MapReduce job to apply the BK to every record in the dataset. Take a sample of some records to find split points (exactly like VSNB).
2. Use a second MapReduce job to split and sort the data, just like VSNB. Then, extend VSNB's second job to compute ASNB only *within* each partition. Write out all windows with routing information.
3. Then, submit another MapReduce job. In this job the routing information from the previous job moves the open windows. The open windows from the previous job are moved to the next partition in this MapReduce job. These windows are still open and may overlap, and potentially match, windows found previously in this partition. Then, if the windows actually overlap, and match, ASNB *merges* them. If any windows are *still* open at the end of step three, repeat step three until no windows are open at the end of any partition.
4. Finally, when all windows are closed, compute cross-product of matching TIDs.

With this structure in place, I will now describe how to compute ASNB at a high level. First, similar to VSNB, ASNB loads the dataset into HDFS. Then, ASNB uses a MapReduce job (job

one) to choose a set of partitions. Please note that job one is *exactly* the same¹ as first MapReduce job of VSNB in Section 5.2.

Next, ASNB executes MapReduce job two: make windows. The second MapReduce job computes record-at-a-time adaptive SNB, as described in Section 7.1.1, for all records within each partition. By the end of this MapReduce job, ASNB has written out two sets of data: the found windows for this partition, and the set of windows that are still open and need to be evaluated if they merge with windows found in the next partition. Windows that are still open on the partition at the end of the dataset are closed. ASNB writes out all windows, enriched with routing information for the next MapReduce job (described further in Section 7.2.2).

Now begins the process of merging the open windows that cross a partition boundary with already-found windows. ASNB starts MapReduce job three to merge windows. In this MapReduce job, in the reduce tasks for each partition, the task is to identify if any incoming still-open windows (that originated on a previous partition) overlap with a set of windows already found for this partition. If any of the found windows overlap with the incoming windows, then ASNB checks to see if their head (first) records match. If the head records match, then ASNB merges the previously-found windows into the incoming window. Again, previously-found window would have been part of the incoming window if ASNB had been run on a single machine.

At the end of MapReduce job three, if any of the incoming windows are still open, then a window spans multiple partitions. While unlikely, a single open window might span many partitions, or even every partition. Therefore, the expected case there will be $O(1)$ merging rounds for p partitions, with the worst case of $O(p - 1)$ merging rounds.

Otherwise, if all of the incoming windows are closed, then ASNB submits MapReduce job four to compute pairwise crossproduct of TIDs within each window. When that is complete, then ASNB copies all of the output from HDFS to a single file on disk. With this, ASNB is complete.

¹In fact, in my empirical study, I use the exact same code.

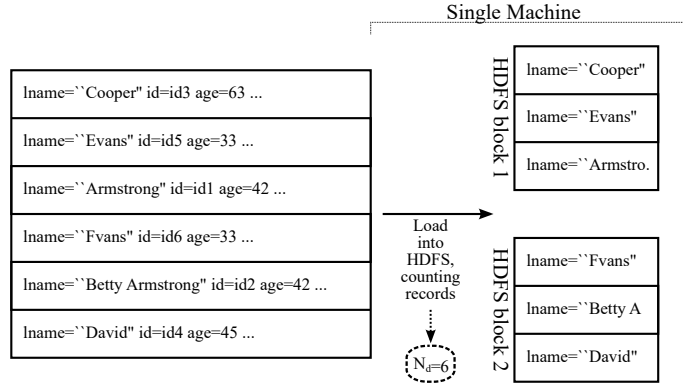


Figure 7.2: ASNB: load data into MapReduce

7.2.2 Detail

In this section, I will describe ASNB in detail. Similar to VSNB in Section 5.2, ASNB uses a MapReduce cluster containing n identical machines, where each machine has k cores, resulting in a cluster-wide total of $n \times k$ cores N_k . Also similar to VSNB, ASNB has two tunable parameters: the HDFS blocksize B , and a sampling frequency s . B and s are returned from a cost model that I will describe further in Section 7.3. ASNB also requires a matching function M and a matching density function ϕ , where $\phi \leq \frac{1}{w-1}$.

To assist with explanation, I will continue the running example from Table 3.1a. In the running example, ASNB also has window size $w = 3$, the Blocking Key Function BK returns the first letter of the lname field, the matching function M is a black box, and there are two machines, each with one core. The HDFS block size B is identified to be 3 records in length.

Thus, in my example, exactly like VSNB in Section 5.2, ASNB loads D into HDFS, with block size B . During loading, ASNB counts the number of records N_r , which in this case is 6. The result is shown in Figure 7.2: there are now two HDFS blocks, each containing three records. HDFS places the blocks onto the HDFS worker nodes, and each block is on a different machine, with no redundancy.

Again, exactly the same as VSNB, ASNB calculates the required number of records to sample, which in this example is 3. With the total number of records counted, ASNB calculates that 50% of the records ($s = 2$) must be sampled to adequately produce split points.

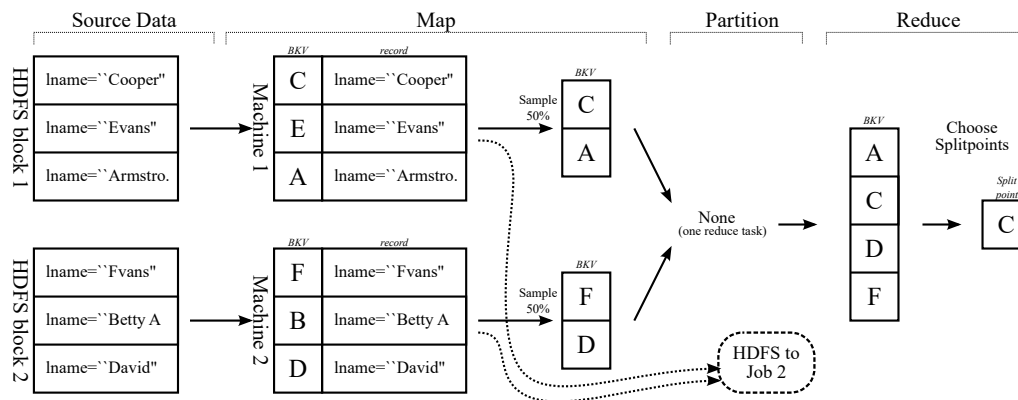


Figure 7.3: ASNB: first MapReduce job

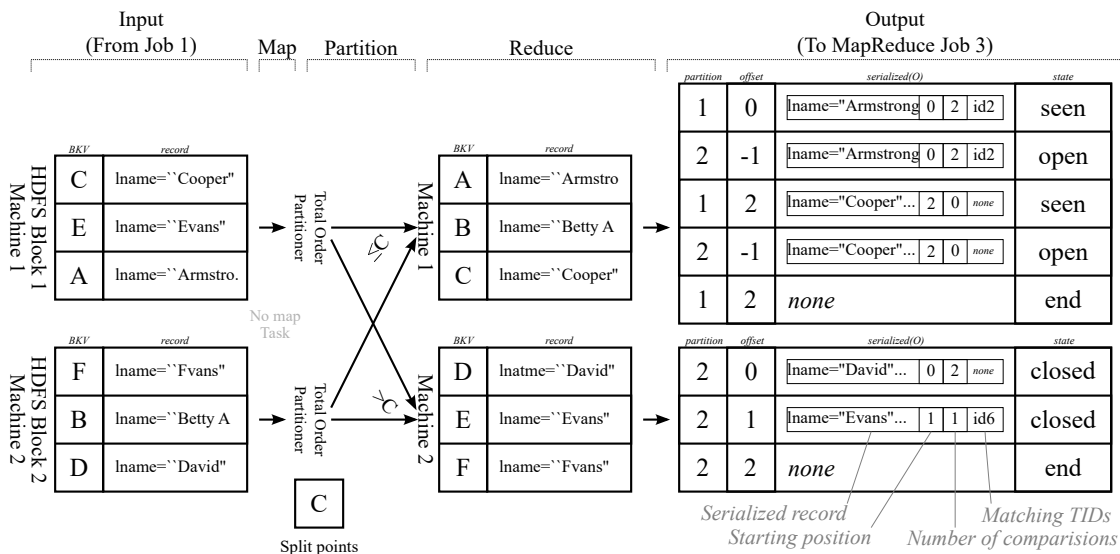


Figure 7.4: ASNB: second MapReduce job

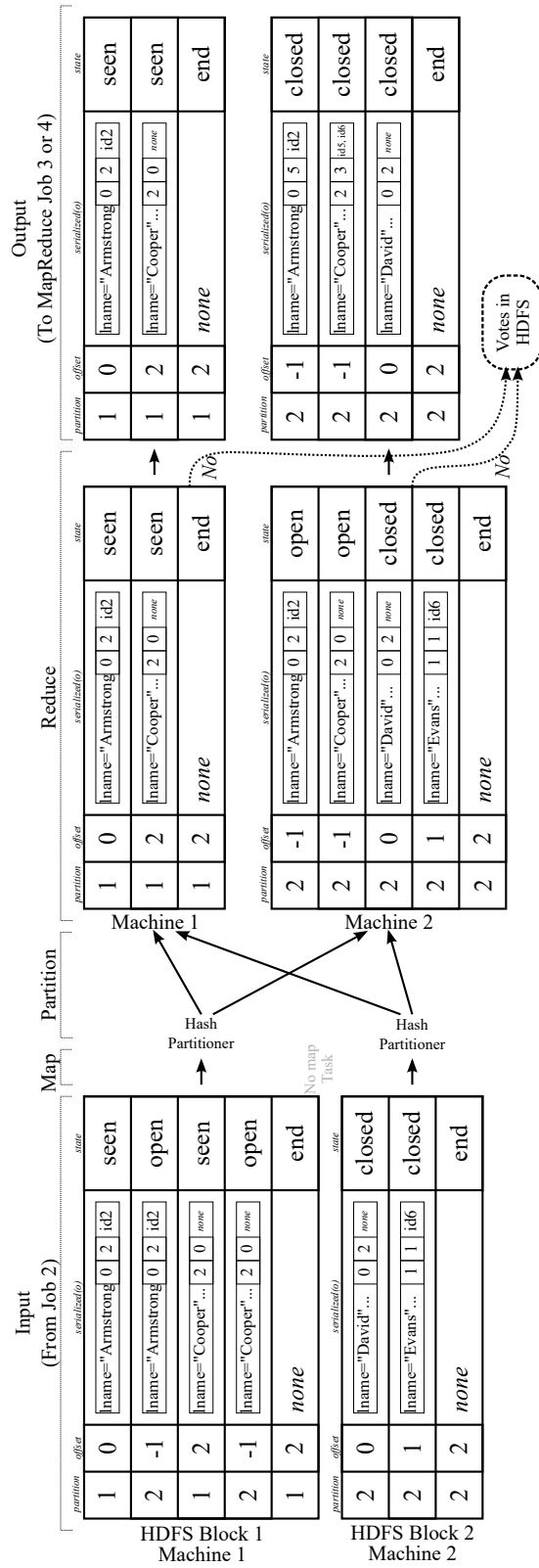


Figure 7.5: ASNB: third MapReduce job

7.2.2.1 MapReduce Job 1: Key and Partition Boundary Generation

As illustrated in Figure 7.3, the first MapReduce job in ASNB is the same as the first MapReduce job in VSNB. However, note that as matching is required, the entire record is copied (not just the TIDs): See Section 5.2. Output from the first MapReduce job is an enriched dataset $\bar{D}$ in HDFS as File H_1 , and a set of partitions P .

7.2.2.2 MapReduce Job 2: Find Windows

ASNB then starts the second MapReduce job. At a high level, the second MapReduce job computes windows within each partition. Open windows the last partition (at the end of the sorted order), are closed. For all other partitions, each MapReduce job outputs windows that are still open at the end of the partition with routing information so they will be sent to the beginning of the next partition, in the next MapReduce job. The first part of this MapReduce job is identical to VSNB.

ASNB starts the second MapReduce job, as shown in Figure 7.4. The input to the map task is the HDFS file that was written in the first MapReduce job. In my running example, the HDFS file from the first MapReduce job was written in two blocks, one block written to machine 1, and one block written to Machine 2. In typical operation, MapReduce starts the map task on the same machine the HDFS block was written to in the first MapReduce job. However, in this MapReduce job, there is no map task, so the input is copied to the partitioner on this machine.

MapReduce job two has no map tasks, so MapReduce copies all (BKV, r) pairs in H_1 to the MapReduce partitioner. MapReduce job two uses the MapReduce standard TotalOrderPartitioner. The TotalOrderPartitioner uses P to partition (BKV, r) pairs across N_k reducers such that the smallest BKV are sent to the first reducer, and the largest BKV are sent to the last reducer. Therefore, in this toy example, there are only two ranges: tuples with BKVs that are alphanumerically less than or equal to C and tuples with BKVs that are alphanumerically greater than C. With this configuration, as seen in Figure 7.4, the tuples with BKVs of A, B, and C, are sent to the reduce task that will run on machine 1, and the others are sent to the reduce task on Machine 2.

When the partitioning is complete, MapReduce starts the reduce tasks. Each reduce task starts by initializing the set of open windows O to be empty. When created, an open window contains the following attributes: a *head* record, the partition offset of the head record, number of comparisons made for this window, and the set of found record ids.

Next, the reduce tasks begin. Each reduce task takes as input its partition of sorted records R .

For each record r in R , ASNB first identifies if r matches any open window o in O . To match r into O , for each record o in O , ASNB calls $M(r, \text{head record of } o)$. If $M(r, \text{head record of } o)$ returns true, then ASNB inserts the id of r into o 's set of found record ids, and stops matching r to other entries in O . This continues for all o in O .

Next, ASNB checks to see if any window o in O must be closed. To check if window o must be closed, ASNB first increases the number of comparisons in o because this position has been “compared” with all entries O . Then, ASNB checks to see if the density of matches is less than ϕ . If the window has had at least $(w - 1)$ comparisons, and $\frac{\text{matching records}}{\text{compared records}} \leq \phi$, then ASNB closes the window. To close the window ASNB writes to MapReduce the tuple: (partition number, o 's original offset, $\text{serialized}(o)$, ‘closed’), and removes o from O .

If r did not match any window in O , then ASNB creates a new window for r . To create a new window for r , ASNB inserts a new o in O , with r as the head record, the offset of r in R as the partition offset, with no comparisons, and an empty set of matching record ids.

This continues for all r in R .

At the end of R , all records have been read and the windows are written as output. However, all the entries in O that extend into the next partition. The windows are still open because they are within $(w - 1)$ positions of the end of the partition, or $\frac{\text{matching records}}{\text{compared records}} \not\leq \phi$. ASNB makes an extra copy of these windows, so they will be sent to the next partition in the next MapReduce job. In ASNB, the output is enriched with routing information to distribute tuples to the correct reduce partition (similar to [21]). The output will be in the form of (partition number + 1, -1, $\text{serialized}(o)$, ‘open’), which will direct this window to the beginning of the next partition in the next MapReduce job. Additionally, ASNB writes out a *seen* record for this window: (partition number, o 's original offset, $\text{serialized}(o)$, ‘seen’). The ‘seen’

record is used in the next MapReduce job to potentially match windows incoming into this partition.

Finally, after any still-open windows are written out, **ASNB** writes out an end of partition record: (partition, max offset, none, 'end'). This will be used to account for the number of records in each partition, used in the next MapReduce job.

Example Now, I will go back to the running example. Figure 7.6a shows the input after reading the first record on machine one. As O was just initialized and empty, the reduce task opens a window with record id1 as the head. The reduce task then inserts that window into O , and the resulting state is shown in Figure 7.6b.

The reduce task then reads in record id2, as shown in Figure 7.6c. The reduce task attempts to see if id2 matches an entry in O . The only record in O is the entry for id1. The reduce task calls matching function M on $M(id1, id2)$, which returns true. Since id1 and id2 match, id2 is added to the id1 entry in O . Also, the number of comparisons for the id1 entry in O has been increased by one. As id1 has not reached the minimum window size, the window for id1 is not closed. The final result after id2 is shown in Figure 7.6d.

After that, the reduce task reads in record id3, as shown in Figure 7.6e. The reduce task will again attempt to match id3 to the only record in O . The reduce task calls $M(id1, id3)$, which returns false. The number of comparisons for the id1 entry in O has been increased by one. As id3 does not match any window in O , a new window is created with id3 as the head node. As id1 still has not been open for at least w positions, the window for id1 is not closed. Figure 7.6f shows the result of adding the window id3 to O .

Now, all records for the reduce task on machine one have been read, so the map task must write out the open windows. In O there are two open windows that extend into the next partition. For each open window, the map task writes out the 'open' and 'seen' records (described above). Finally, the reduce task outputs a record denoting the maximum partition offset for partition 1, with a record of type 'end'. The first partition's reduce task output to MapReduce is shown in Figure 7.6g.

offset	BKV	record	
0	A	lname="Armstrong", id="id1", ...	←
1	B	lname="Betty Armstrong", id="id2", ...	
2	C	lname="Cooper", id="id3", ...	

(a) After id1: Records

head record	offset	number of comparisons	matching record ids
lname="Armstrong", id="id1", ...	0	0	<i>none</i>

(b) After id1: Open windows O

offset	BKV	record	
0	A	lname="Armstrong", id="id1", ...	←
1	B	lname="Betty Armstrong", id="id2", ...	
2	C	lname="Cooper", id="id3", ...	

(c) After id2: Records

head record	offset	number of comparisons	matching record ids
lname="Armstrong", id="id1", ...	0	1	id2

(d) After id2: Open windows O

offset	BKV	record	
0	A	lname="Armstrong", id="id1", ...	←
1	B	lname="Betty Armstrong", id="id2", ...	
2	C	lname="Cooper", id="id3", ...	

(e) After id3: Records

head record	offset	number of comparisons	matching record ids
lname="Armstrong", id="id1", ...	0	2	id2
lname="Cooper", id="id3", ...	2	0	<i>none</i>

(f) After id3: Open windows O

partition number	partition offset	serialized(<i>o</i>)	state
1	0	[lname="Armstrong", id="id1", ..., 0, 2, id2]	'seen'
2	-1	[lname="Armstrong", id="id1", ..., 0, 2, id2]	'open'
1	2	[lname="Cooper", id="id3", ..., 2, 0, <i>none</i>]	'seen'
2	-1	[lname="Cooper", id="id3", ..., 2, 0, <i>none</i>]	'open'
1	2	<i>none</i>	'end'

(g) Job 2 output to MapReduce

Figure 7.6: ASNB example: MapReduce job 2, machine one

offset	BKV	record	
0	D	lname="David", id="id4", ...	←
1	E	lname="Evans", id="id5", ...	
2	F	lname="Fvans", id="id6", ...	

(a) After id4: Records

head record	offset	number of comparisons	matching record ids
lname="David", id="id4", ...	0	0	<i>none</i>

(b) After id4: Open windows O

offset	BKV	record	
0	D	lname="David", id="id4", ...	←
1	E	lname="Evans", id="id5", ...	
2	F	lname="Fvans", id="id6", ...	

(c) After id5: Records

head record	offset	number of comparisons	matching record ids
lname="David", id="id4", ...	0	1	<i>none</i>
lname="Evans", id="id5", ...	1	0	<i>none</i>

(d) After id5: Open windows O

offset	BKV	record	
0	D	lname="David", id="id4", ...	←
1	E	lname="Evans", id="id5", ...	
2	F	lname="Fvans", id="id6", ...	

(e) After id6: Records

head record	offset	number of comparisons	matching record ids
lname="David", id="id4", ...	0	2	<i>none</i>
lname="Evans", id="id5", ...	1	1	id6

(f) After id6: Open windows O

partition number	partition offset	serialized(o)	state
2	0	[lname="David", id="id4", ..., 0, 2, <i>none</i>]	'closed'
2	1	[lname="Evans", id="id5", ..., 1, 0, id6]	'closed'
2	2	<i>none</i>	'end'

(g) Job 2 output to MapReduce

Figure 7.7: ASNB example: MapReduce job 2, machine two

Now focus on partition two on machine two in Figure 7.7. A similar process to what occurred on machine 1 occurs on machine 2. See Figures 7.7a through 7.7g for the intermediate steps.

With this, the reduce task on machine two has read in all input. Partition two is the final partition, so the reduce task closes all the open windows. As the windows are closed, the reduce task writes their output to MapReduce as shown in Figure 7.7g. Finally, the reduce task writes out an ‘end’ record, with the maximum offset of 2 for this partition. With this, the second MapReduce job is complete.

7.2.2.3 MapReduce Job 3: Merge Windows

The goal of this MapReduce job is to merge any windows that were open beyond the end of a partition to matching windows in the next partition. Therefore, the input to this MapReduce job is a set of records (partition number, partition offset, serialized o , record state) from the previous MapReduce job. The output will be a set of found windows, as well as any open windows that need to be passed to the next partition in another MapReduce job (if necessary).

In this MapReduce job there is no map task, and all tuples are passed to the partitioner. The partitioner is a standard hash-based partitioner on the `partition number` attribute of the records. As always during partitioning, MapReduce sorts the records.

Then, the reduce tasks begin. First, each reduce task reads any records with a negative offset and a state of ‘open’. These records represent the incoming open windows. Next, ASNB needs to find the *maximum offset* of each incoming window. The maximum offset is how far each window can *extend* into this partition. The maximum offset determines which windows that this incoming window overlaps, and thus must be investigated for a possible match.

The maximum offset for each record is defined as $\lceil \frac{m}{\phi} - c \rceil$, where m is the length of the set of matching records from the previous MapReduce job ids plus one, and c is the number of comparisons plus one. As the records are given to the reduce task in sorted order by the offset, the reduce task can quickly check to see if any of the incoming window overlaps using that window’s maximum offset.

Input on machine one			
partition number	partition offset	serialized(<i>o</i>)	state
1	0	[lname="Armstrong", id="id1", ... , 0, 2, id2]	'seen'
2	-1	[lname="Armstrong", id="id1", ... , 0, 2, id2]	'open'
1	2	[lname="Cooper", id="id3", ... , 2, 0, none]	'seen'
2	-1	[lname="Cooper", id="id3", ... , 2, 0, none]	'open'
1	2	none	'end'

Input on machine two			
partition number	partition offset	serialized(<i>o</i>)	state
2	0	[lname="David", id="id4", ... , 0, 2, none]	'closed'
2	1	[lname="Evans", id="id5", ... , 1, 0, id6]	'closed'
2	2	none	'end'

Figure 7.8: ASNB: input to MapReduce job 3, prior to partitioning

Then, ASNB inserts the incoming window (head record, offset, c , set of matching record ids, maximum offset) into the set of incoming windows I . This repeats for all incoming windows (with offset -1).

The rest of the records as input to this reduce task are the set of windows F that were found by this partition during the last MapReduce job..

For each record f , ASNB compares f with the windows in I . For each entry i in I , ASNB first determines if f and i *overlap*. f and i overlap if $f_{\text{offset}} \leq i_{\text{max offset}}$.

If i overlaps with f , then ASNB checks to see if the head records match. If the head records match, then i and f are actually part of the same window, and must be merged. To merge f into i , the id from f 's head record is added to i 's matching records. Also, any matching records from f are added to i 's matching records. Then i 's maximum offset is set to $m \times (w - 1) - c$, where m is the length of matching records + 1, and c is the number of comparisons made by this window in a previous MapReduce job. If f matches more than one record in I , then the records in I are merged, and the maximum offset of the merged record is adjusted accordingly. Finally, f is deleted from F .

Otherwise, if i did not overlap with f , then the reduce task knows all other entries in F will not overlap with i (all later entries in F will have a larger offset than f). Thus there are no

Input on machine one			
partition number	partition offset	serialized(<i>o</i>)	state
1	0	[lname="Armstrong", id="id1", ... , 0, 2, id2]	'seen'
1	2	[lname="Cooper", id="id3", ... , 2, 0, none]	'seen'
1	2	none	'end'

Input on machine two			
partition number	partition offset	serialized(<i>o</i>)	state
2	-1	[lname="Armstrong", id="id1", ... , 0, 2, id2]	'open'
2	-1	[lname="Cooper", id="id3", ... , 2, 0, none]	'open'
2	0	[lname="David", id="id4", ... , 0, 2, none]	'closed'
2	1	[lname="Evans", id="id5", ... , 1, 0, id6]	'closed'
2	2	none	'end'

Figure 7.9: ASNB: third MapReduce job, after partitioning and sorting

other windows that i might match with, i is closed. To close i , ASNB writes out to MapReduce (partition, offset, serialized (i), 'closed'), and removes i from I .

If f did not match any record in I , then ASNB writes to MapReduce: (partition, offset, serialized(f), flag), where flag indicates whether this window is 'seen' or 'closed'.

This continues for all f in F .

After all records are processed, if any records remain in I , then that means windows in I spanned this entire partition. If there are any open windows in I , the reduce task votes 'yes' that at least one more merging round is required.² Then, the reduce task will process each open window i in I . ASNB increases the number of comparisons for i by the max_offset. Then, ASNB writes the record for the next partition: (partition+1, -1, serialized(i), 'open'). Finally, since other windows from a previous partition might be brought forward to this partition, the reduce task writes a copy of this open window as a *seen* window: (partition, offset, serialized(i), 'seen'). As above, a seen window is not brought in as an incoming window, and can only be merged with an incoming window from a previous partition.

²The particular strategy I used when I implemented voting in ASNB was to write a zero length file in a known location in HDFS.

Finally, the map task writes out the maximum offset record for this partition: (partition, maximum offset, *none*, 'end'). The maximum offset record is carried forward in case there needs to be another MapReduce job. The map task is complete.

After all map tasks complete, and the third MapReduce job is complete, ASNB must determine if another merging round is required.

At this point ASNB looks to see if *any* partition had voted 'yes' to needing another merging round. If any votes 'yes', then ASNB starts another MapReduce job three. If no partitions voted for another round, then ASNB moves to MapReduce job four.

Running Example The input to the third MapReduce job is the output from the second MapReduce job, and is shown in Figure 7.8. With the third MapReduce job, there is no map task, and the input is passed to the MapReduce partitioner. The MapReduce partitioner partitions based upon the 'partition' attribute, and during partitioning MapReduce sorts the data. The result is shown in Figure 7.9. Note that the open windows from partition one have now been successfully copied to the beginning of partition two.

The reduce tasks then start. The first job of the reduce tasks is to read the incoming open windows I . As shown in Figure 7.10, the first partition has no incoming open windows, and therefore I is empty. Thus there is no work for this partition to do, and all input is rewritten to output. This reduce task votes 'no' for another merging round.

On the second partition, there are incoming windows. The first task is for each window, calculate the maximum offset for this window. Again, the maximum offset is how far this window can expand into this partition. For the *id1* entry, the maximum offset is $\lceil \frac{m}{\phi} - c \rceil = \frac{2}{\frac{1}{2}} - 2 = 4 - 2 = 2$, as shown in Figure 7.11. For the *id3* entry, the maximum offset is also 2, as is also shown in Figure 7.11. Thus, both incoming windows can currently match with windows that start on or before offset 2.

With the maximum offsets established, the reduce task now starts iterating through the found windows F . The reduce task reads in record *id4*, and attempts to match it with each window in I . Thus, the reduce task identifies the offset of *id4* is less than or equal to the maximum offset of *id1*, calls $M(id4, id1)$, which returns false, therefore the windows do not match and should not

Partition 1 incoming open windows I				
head record	offset	number of comparisons	matching record ids	maximum offset
<i>none</i>				
Partition 1 found windows F				
head record	offset	number of comparisons	matching record ids	
lname="Armstrong", id="id1", ...	0	2	id2	
lname="Cooper", id="id3", ...	2	0	<i>none</i>	

Figure 7.10: ASNB: MapReduce job 3, reduce task one

Partition 1 incoming open windows I				
head record	offset	number of comparisons	matching record ids	maximum offset
lname="Armstrong", id="id1", ...	-1	2	id2	2
lname="Cooper", id="id3", ...	-1	0	<i>none</i>	2
Partition 1 found windows F				
head record	offset	number of comparisons	matching record ids	
lname="David", id="id4", ...	0	2	<i>none</i>	
lname="Evans", id="id5", ...	1	1	id6	

Figure 7.11: ASNB: MapReduce job 3, reduce task two

Output from machine one			
partition number	partition offset	serialized(<i>o</i>)	state
1	0	[lname="Armstrong", id="id1", ... , 0, 2, id2]	'seen'
1	2	[lname="Cooper", id="id3", ... , 2, 0, none]	'seen'
1	2	none	'end'

Output from machine two			
partition number	partition offset	serialized(<i>o</i>)	state
2	-1	[lname="Armstrong", id="id1", ... , 0, 2, id2]	'closed'
2	-1	[lname="Cooper", id="id3", ... , 2, 0, (id5, id6)]	'closed'
2	0	[lname="David", id="id4", ... , 0, 2, none]	'closed'
2	2	none	'end'

Figure 7.12: ASNB: output from third MapReduce job

be merged. Next, the reduce task identifies the offset of id4 is less than or equal to the maximum offset of id3, calls $M(id4, id3)$, which also returns false. Therefore window id4 does not match any open windows in I , and window id4 is written to output.

Next, the reduce task reads in id5, and once again tries to match it with the open windows in I . The reduce task identifies id1 overlaps with id5, and the reduce task calls $M(id5, id1)$ which returns false. Next, the reduce task identifies id5 also overlaps with id3, the reduce task calls $M(id5, id3)$ which returns true. The reduce task merges windows id5 and id3. To merge windows id5 and id3, the reduce task adds the TIDs of id5 and id6 to the matching record TIDs of window id3. Next, the reduce task calculates the new maximum offset for id3, $\lceil \frac{m}{\phi} - c \rceil = \frac{3}{\frac{1}{2}} - 0 = 6$. Note that c was not updated, and contains the incoming value.

Now, the reduce task is finished reading records in F . Since the second reduce task is the partition at the end of the dataset, the reduce task closes all open windows. Next, the reduce task writes out the end record. The reduce task writes its output to MapReduce, as shown in Figure 7.12.

Finally, as this reduce task has no open windows, it votes 'no' for another MapReduce merging round. ASNB tallies the votes. Since zero 'yes' votes were recorded, the merging rounds are complete. ASNB then moves on to MapReduce job 4.

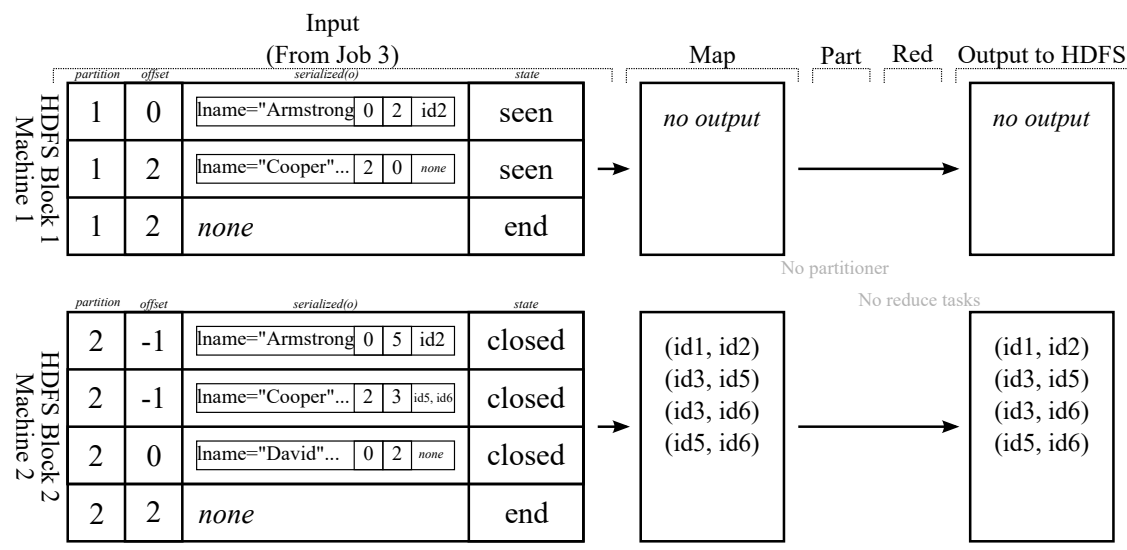


Figure 7.13: ASNB: fourth MapReduce job

Output from machine one	
none	
Output from machine two	
records	ASNB output
id1, id2	→ (id1, id2)
id3, id5, id6	→ (id3, id5), (id3, id6), (id5, id6)

Figure 7.14: ASNB: output from fourth MapReduce job

7.2.2.4 MapReduce Job 4: Prepare Output

Once all open windows have been closed, ASNB runs a final MapReduce job to produce the final output from ASNB as shown in Figure 7.13.

MapReduce job 4 is a map-only job to compute the *per-window* transitive closure as shown in Figure 7.13. The transitive closure finds all other matching tuple pairs. The full transitive closure is not necessary as any matching tuples already caused windows to merge.

Each map task reads each tuple output from the previous MapReduce job. If the tuple type is ‘closed’, and there are at least two matching TIDs, then the map task will compute a pairwise cross product of the TIDs from this window.³ The reduce task writes pairwise TID output to MapReduce, which as always writes this to HDFS. All other tuple types (‘open’, ‘end’, or ‘seen’) are ignored.

This continues until all tuples have been processed. With this MapReduce job is complete, and the ASNB output has been computed. Finally, after the MapReduce job is complete, ASNB copies all the output from HDFS to a single file on disk. When this is complete, ASNB is also complete.

Running Example Going back to my running example, see Figure 7.14. The map task for partition one has no tuples with state ‘closed’, therefore that map task produces no output. The map task for partition two has two closed windows. The window id1 has one additional “matching record id” and therefore the map task outputs (id1, id2). The window id3 has two additional “matching record ids” and outputs (id3, id5), (id3, id6), (id5, id6). This output is written to MapReduce, which writes it to HDFS, and is shown in Figure 7.14.

Comparison with VSNB Note that the pair (id3, id6) was returned by ASNB in Figure 7.14, but not by VSNB in Figure 5.8. This illustrates the benefits of ASNB.

7.3 Cost Model

In this section I will describe the cost model for ASNB. The goal is to pick a good plan for ASNB, similar to VSNB in Section 5.3.

³Optionally, with datasets with tuples that each have many duplicates, it may be desirable to return the connected component instead of the pairwise output.

MapReduce Job	VSNB	ASNB
1	Make BKV	Make BKV
2	Make SNB output within partition and output answers	Make partial ASNB output within partition
3	Fixup SNB output between partitions and output answers	Merge partial ASNB windows (typically in 1 job)
4		Compute pairwise crossproduct of TIDs within windows

Table 7.1: VSNB versus ASNB strategy

At a high level, both ASNB and VSNB share a similar strategy. Like VSNB, ASNB runs using MapReduce on a cluster of n identical machines. Each machine has k cores, for a cluster total of N_k cores. D is in a single file of size $|D|$ megabytes, with N_D records, on a single machine outside the cluster.

If you compare the overview figure for VSNB in Figure 5.3 on page 54 with the overview figure for ASNB in Figure 7.1 on page 93, you will notice a substantial commonality. To look in more detail, see Table 7.1.

Looking at the first MapReduce job, it is exactly the same, thus the same optimization is required. For the second MapReduce job, a similar amount of work is needed. It is true that ASNB may do more work as windows are larger, and matching is required. Matching is accounted for in the VSNB cost model. With regards to the window size in ASNB, it is not predictable how large the windows will be *a priori*. The cost for matching ASNB base window sizes is the exact cost used for VSNB, thus the same optimization is required.

The third MapReduce job in VSNB is invariant to the tunable parameters. It is a static cost, based upon the number of partitions. In ASNB, the third job is also invariant to the tunable parameters. It, too, is essentially a static cost: just as with job 2, since the actual size of the windows cannot be determined *a priori*, they cannot be optimized. The number of MapReduce jobs also is argued to be one job (see previous section). Even if ASNB takes multiple merging rounds (which become multiple MapReduce jobs), it is not possible to tune any of the knobs to adjust the time those jobs take.

The fourth job in **ASNB** does not exist in **VSNB**, but is a function solely of the total number of windows. Thus, it is invariant to the tunable parameters. Therefore, as all of the operations that differ between **ASNB** and **VSNB** are invariant to the tunable parameters, the same calibration procedure will be required in **ASNB** as was used in **VSNB**.

7.4 Empirical Study

In this chapter, I have described **ASNB** in Section 7.2, and motivated the reuse of the **VSNB** cost model in Section 7.3. Now I move on to an empirical study of **ASNB** performance.

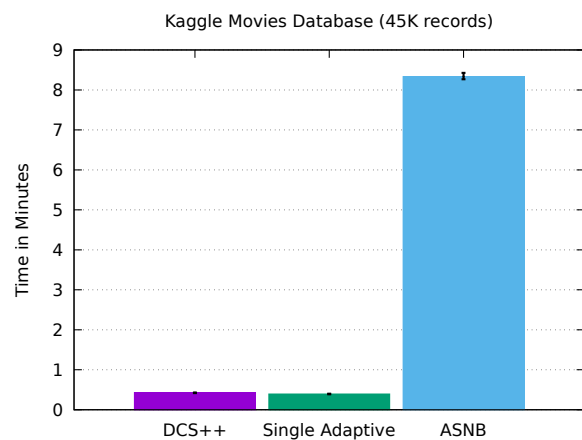
7.4.1 Performance

Exactly like **VSNB**, for all of my **ASNB** empirical studies, I will use the datasets shown in Table 5.3 (page 68).

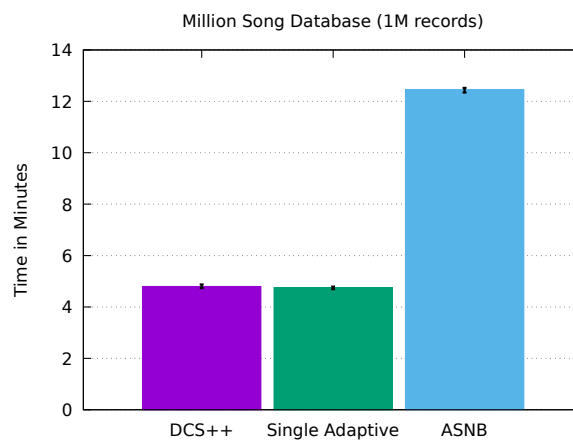
ASNB Performs Better on Large Datasets I claim **ASNB** will process large datasets faster than either **DCS++** or a simple single machine adaptive **SNB** (described in Section 7.1.1). To evaluate this claim, I ran each algorithm ten times for each dataset, removing the high and low values, and averaging the result. The results are in Figure 7.15

Similar to **VSNB**, for the smaller datasets (Figures 7.15a, 7.15b, and 7.15c) single machine adaptive **SNB** solutions (both **DCS++** and my single machine adaptive **SNB**) finish faster than **ASNB** by 6 to 8 minutes. However, in the larger datasets (Figures 7.15d and 7.15e), **DCS++** becomes the slowest (by 12 minutes to 10 hours). **ASNB** is faster than my simple single machine adaptive **SNB** method by about 35 minutes to 2 hours.

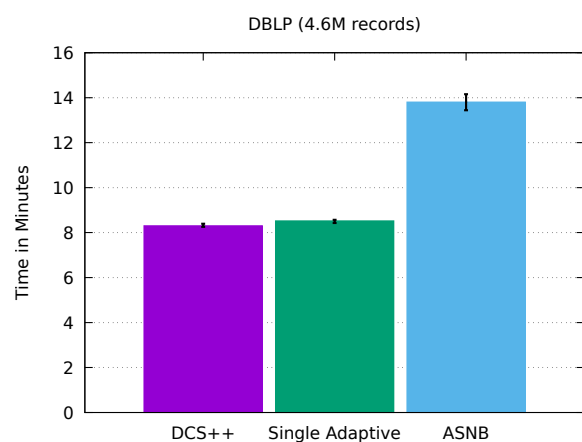
Similar to **VSNB**, these results demonstrate that a single machine solution remains the best choice for small datasets. As I tested, after about 5 million records the MapReduce-based **ASNB** solution has best performance. Further, **DCS++** or my single machine adaptive **SNB** method have similar performance, with my method having a slight advantage in the input dataset sizes where single machine solutions are warranted.



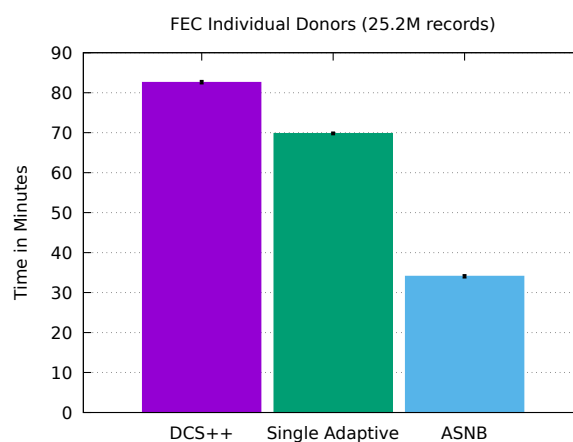
(a) Kaggle Movies



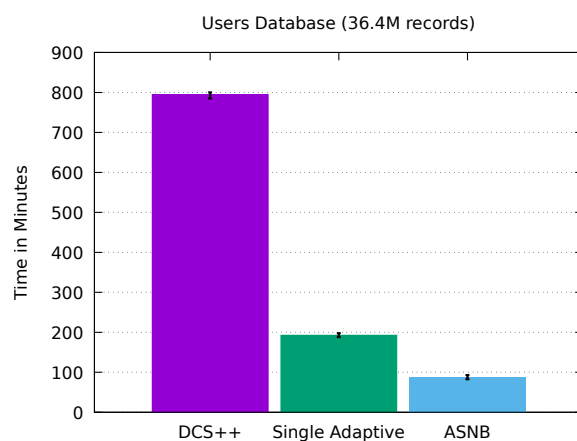
(b) Million Song Dataset



(c) DBLP



(d) FEC Individual Donors



(e) Users Database

Figure 7.15: Relative performance of each ASNB method

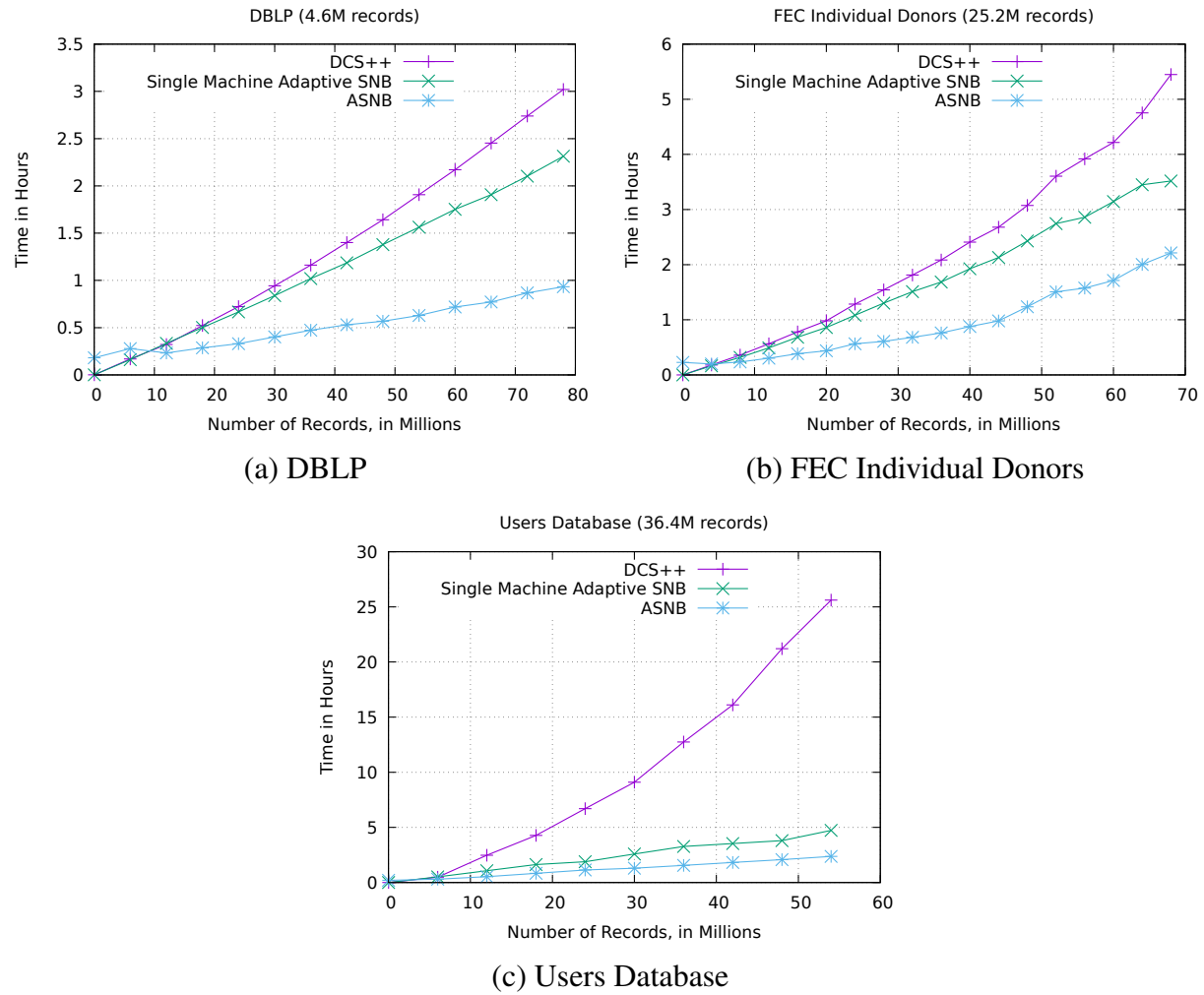


Figure 7.16: Relative performance of each ASNB method, scaling up the number of records

Performance as Input Data Grows I will show that with very large datasets, **ASNB** finishes processing faster, and the gap increases. To examine this claim, I once used the same process to create arbitrary-length datasets as I used in Section 5.4.1 on page 71. I then ran each method with a sequence of increasing input sizes. The output from those sequences is shown in Figure 7.16.

As shown in Figure 7.16, as the dataset sizes grow, **ASNB** is the fastest by about 2 hours for 78M tuples (Figure 7.16a), 1.25 hours for 68M tuples (Figure 7.16b), and about 2 hours for 54M tuples (Figure 7.16c). **DCS++** was the slowest, by up about a whole day (54M tuples, Figure 7.16c). This performance decrease was due to the dataset needing to spill to disk, thus **DCS++** became disk bound.⁴

Increase in Dataset Size Finally, I will show that as the dataset sizes grow, the time to process adaptive SNB will grow linearly. Looking at Figure 7.16, all the adaptive SNB solutions grow at a substantially linear rate up to the native size of their dataset (unlike the performance in RepSN in Section 5.4.1).

Specifically, for Figure 7.16a, the single machine adaptive SNB method and **ASNB** both scale linearly. **DCS++** tends to grow slightly more than linearly. The superlinearity is caused by the increasingly duplicated tuples in the dataset. As I mentioned above, I synthetically extend the dataset by duplicating tuples (and adjusting the TIDs). This causes there to also be an increasing amount of duplicate tuples after 4.6M records. Both **DCS++** and my single machine adaptive SNB solution have to keep track of ever-increasing windows as the density of actually-matching records increases. **ASNB**, however, masks this time with a slightly slower initial window computation, but a fast merging phase.

That is not the only place this occurs, however. The same effect is seen to a lesser degree in Figure 7.16b, where the processing time **DCS++** versus my single machine adaptive SNB method is comparable through about the native dataset size (25.2M records). After that point, **DCS++** starts to slow down as the window sizes are larger (by approximately 2 hours at 68M records). Finally,

⁴In my experimental setup a standard 10K PRM disk was available. Faster storage would have reduced, but not eliminated, the difference.

a similar effect occurs in Figure 7.16c, where again DCS++ scales linearly until the native size of the dataset, where it starts to grow.

7.4.2 Effects of the Optimizer

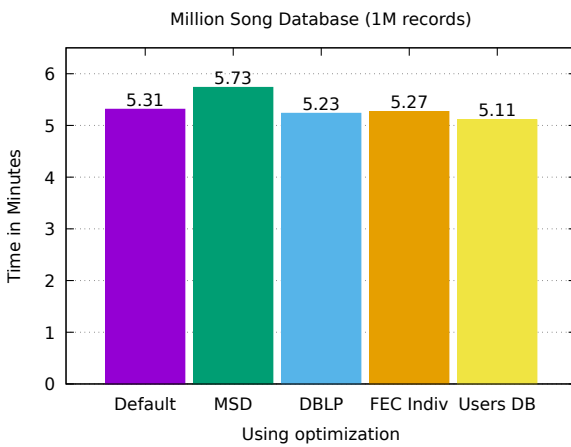
I proposed in Section 7.3, the VSNB optimizer can be reused for ASNB. This section will evaluate that claim.

The Optimizer Picks a Good Plan I claim that the optimizer I have implemented for VSNB will select a good plan for ASNB in the plan space. To examine this claim, similar to VSNB in Section 5.4.2 (page 71), I selected the four largest datasets from Table 5.3 on page 68. Then, for each dataset, I optimized ASNB for that dataset using the VSNB optimizer, and saved the selected plan. The results are in Figure 7.17.

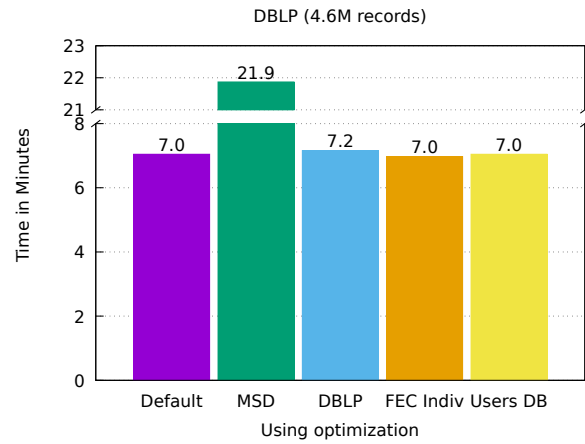
As expected the results for ASNB are similar to the VSNB cross optimization graphs in Section 5.4.2. Similar to the results I found for VSNB, the million song dataset is so small the performance is comparable (within a minute in Figure 7.17a). For the other datasets, the million song dataset optimization is a poor choice (by 3x in Figure 7.17b, 12x in Figure 7.17c, and 58x in Figure 7.17d). For DBLP in Figure 7.17b, the choice was not the fastest choice, however, it was very close (under 30 seconds). The optimized plan for the FEC Individual Donors dataset in Figure 7.17c also is very close (under 15 seconds). Finally, the Users Database in Figure 7.17, shows that, like VSNB, the optimizer for ASNB worked better for large datasets (2x faster than the next-best plan).

Optimizing Larger Datasets As I found with VSNB, I claim the optimizer will pick a plan for ASNB that is faster than the default plan or other optimized plans, as the datasets sizes increase.

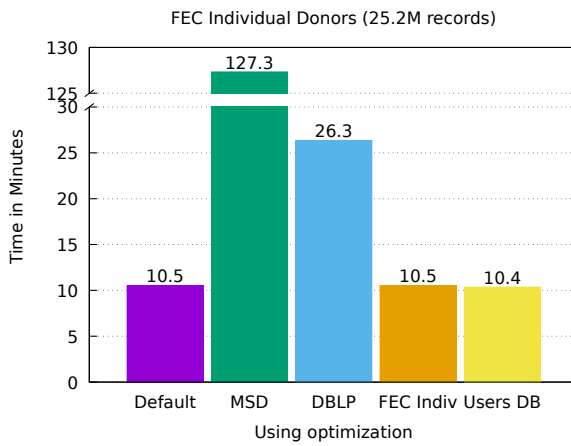
To empirically test this claim, I once again took the datasets for FEC Individual Donors dataset (Figure 7.17c) and the Users Database (Figure 7.17d) and created two new datasets, one doubled in size and one tripled in size. Then, I ran the optimizer against each of the new data sources, and ran each the new dataset against each of the optimized plans. The results are shown in Figures 7.18 and 7.19.



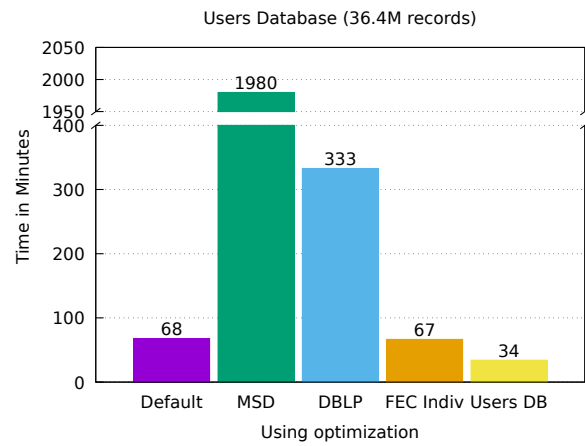
(a) Million Song Dataset



(b) DBLP

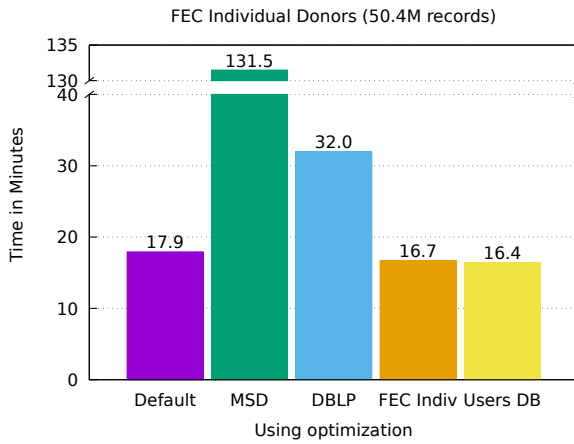


(c) FEC Individual Donors

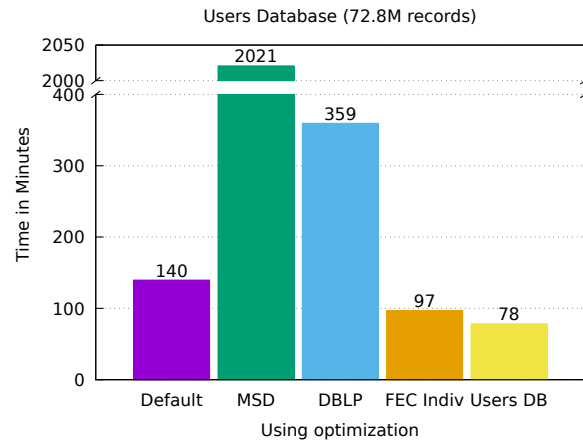


(d) Users Database

Figure 7.17: ASNB cross optimization. Each dataset is run with a set of optimized plans.

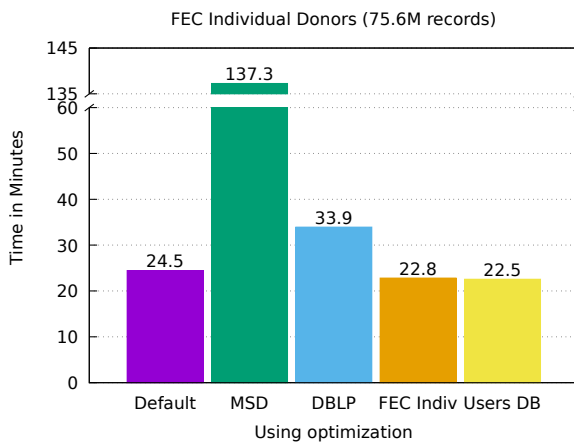


(a) FEC Individual Donors

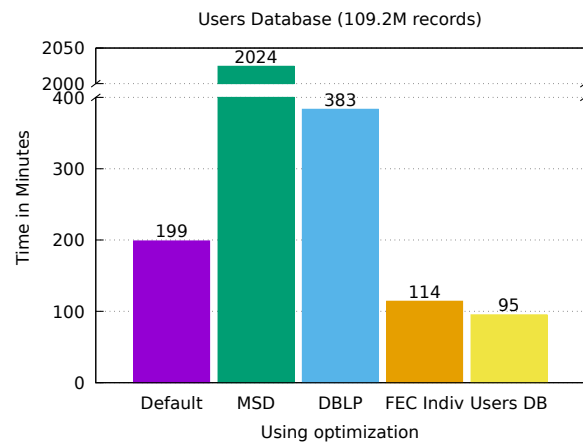


(b) Users Database

Figure 7.18: ASNB cross optimization using double the source data. Each dataset is run with a set of optimized plans.



(a) FEC Individual Donors



(b) Users Database

Figure 7.19: ASNB cross optimization using triple the source data. Each dataset is run with a set of optimized plans.

In Figure 7.18a, the performance between the optimizations for the FEC Individual Donors Dataset and the Users Database are comparable (within 30 seconds). However, in Figure 7.18b, the performance of the Users Database optimization is the best, by 19 minutes. Tripling the data, in Figure 7.19a the difference was 15 seconds, although with the Users Database optimization providing the slightly better plan. In Figure 7.19a the performance of the Users Database optimization was once again 19 minutes faster.

7.4.3 Sensitivity Analysis

So far I have shown that the optimizer of VSNB picks good plans, and thus can be reused for ASNB. Here, I will check to ensure that the sensitivity analyses also hold.

P_s Does Not Drastically Affect Runtime Here I will show that altering P_s within reasonable amounts still does not have a drastic effect on runtime (as I showed with VSNB in Section 5.4.3 on page 75). To verify that this holds in ASNB, I again ran the two largest datasets from Table 5.3 (page 68), altering P_s . The results are shown in Figure 7.20, where in either graph, the varying of the probability does not alter the runtime substantially.

Window Size I also will show that for ASNB, adjusting the starting window size does not drastically affect the runtime (similar to Section 5.4.3 on page 77).

To investigate this claim, I used the same method as in Section 5.4.3. I once again took the two largest datasets from Table 5.3 (page 68). This time, I ran ASNB setting the window size w to a series of values: (5, 10, 20, 30). For each dataset, I ran ASNB adjusting the window size. For each adjustment of the window size, I ran ASNB 10 times, discarding the high and low values, and averaging the remaining values. The results were once again that despite the window size, the overall runtime was substantially the same, as shown in Figure 7.21. Therefore, the runtime of ASNB is not affected by minor adjustments to the window size.

Cluster Size Finally, I claim that altering the size of the cluster of MapReduce worker machines for ASNB will produce reasonable speedup as was shown in Section 5.4.3 (page 77).

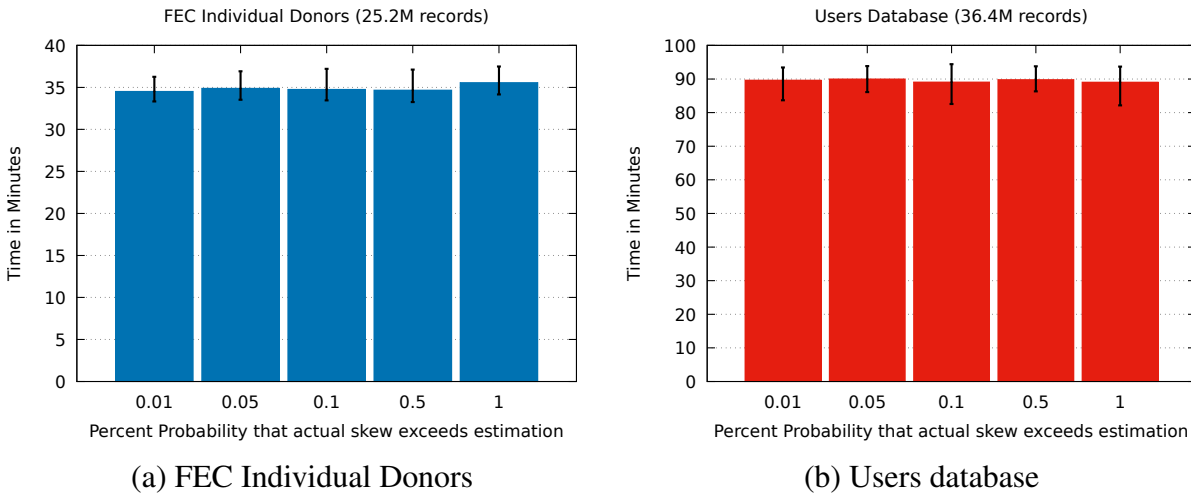
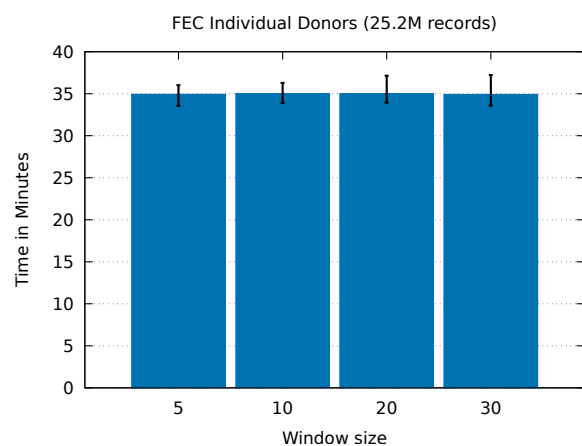
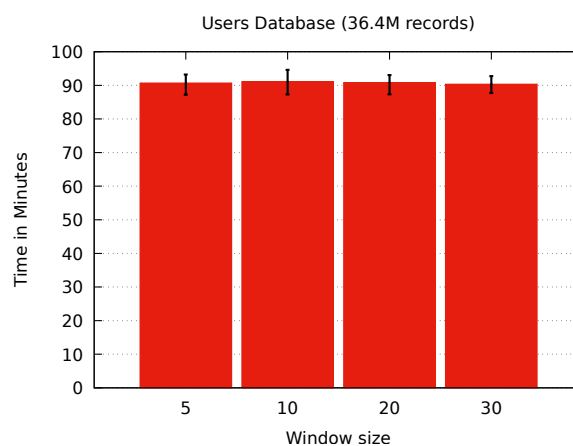


Figure 7.20: ASNB Sensitivity Analysis: adjust probability of skew

To investigate this claim, I used the same method as in Section 5.4.3. I once again took the two largest datasets from Table 5.3 (page 68) and tripled their size. In this set of experiments, I first adjusted the number of worker nodes in the cluster through a series of values: 5, 10, and 20 cluster nodes. After adjusting the worker nodes, I recreated the HDFS datastore to ensure it was healthy. Then, for each dataset, I ran ASNB 10 times, discarding the high and low values, and averaging the remaining values. The result in Figure 7.22 shows that as the number of machines is increased (from 5 to 20) the time decreases (by 23 minutes and 7 hours). As ASNB counts the time it takes to load the dataset from a single file on disk into HDFS, this is a substantial and fixed cost. Therefore, I conclude the runtime of ASNB gains speedup from additional cluster nodes.

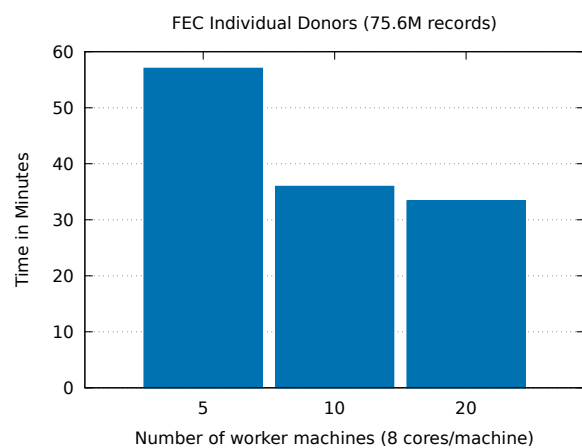


(a) FEC Individual Donors

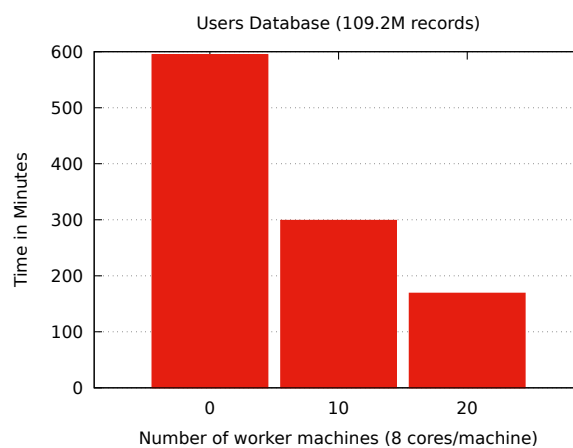


(b) Users database

Figure 7.21: ASNB Sensitivity Analysis: adjust window size



(a) FEC Individual Donors



(b) Users database

Figure 7.22: ASNB Sensitivity Analysis: adjust number of worker machines

Chapter 8

Conclusion and Future Work

This dissertation studied Sorted Neighborhood Blocking (SNB) an important component of many EM pipelines. Prior to this work, SNB methods were studied, but only with relatively small datasets. When looking at SNB methods with large datasets I identified several issues with current solutions. I started this dissertation with a survey of blocking methods in Chapter 2. Next, I described current blocking methods in Chapter 4, and in particular their limitations. After that, I introduced a new SNB method, called **VSNB** in Chapter 5. **VSNB** outperforms the current state-of-the-art MapReduce SNB solution on large datasets. Furthermore, **VSNB** uses a cost model to effectively adjust parameters for efficient execution in a MapReduce cluster.

As adding adaptivity is a desirable feature for SNB, I described current solutions in Chapter 6. Using that background, in Chapter 7, I proposed a new Adaptive SNB method called **ASNB**. **ASNB** leveraged a novel way to compute SNB on MapReduce by merging partially-computed windows. First, partition the dataset and compute adaptive SNB within each partition. Then, copy partially-computed adaptive SNB answers to the next partition and *merge* the results. For both **VSNB** and **ASNB**, I showed empirical studies that demonstrate the promise of my solutions. While promising, much work remains. Important future work directions include the following.

Adaptive SNB through Transitive Closure When extending DCS++ to become **ASNB**, it is clear that transitivity is important. The strategy I used was to compute DCS++ exactly in MapReduce. One criticism of DCS++ is that the density function is insensitive to the length of the window. As observed in Section 7.1.2, the choice of SNB as a blocking strategy is one where you expect actually-matching records to be very close to each other in sorted order. However, if DCS++

encounters a very dense area of actually-matching records, the window will extend quite far — potentially very far after the last actually-matching record. For example, imagine if 100 duplicates of a record are found, and $w = 101$ and $\phi = \frac{1}{100}$. In this case, unless the end of the dataset is reached first, this window will always expand to be at least 10,000 records wide *even if no other matching records are found in the last 9,900 records*.

It would seem that adaptive SNB would benefit from DCS++’s expand-by-one-window-size method, and could omit the density function. Note that in this case, this is exactly VSNB with a transitive closure over all records at the end. Thus, with a clever distributed transitive closure method, an even simpler adaptive SNB function could be developed.

Spark Other distributed contribution paradigms other than MapReduce exist. For example, Apache Spark uses in-memory datasets to distribute computation, and is not limited to map-partition-reduce steps.

Debugging SNB Finally, adding debugging to SNB could be very helpful. For example, consider the problem faced by the data scientist who is trying to decide how big to make the window size. How does he or she know if her window size is big enough? Here is a method to allow for examining different window sizes in the standard (vanilla) SNB setting. Instead of outputting (TID_a, TID_b) output (TID_a, TID_b, d) where d is the distance between the records in sorted order. This way, the analyst can run SNB once with a window size larger than typical, and use the distance information to determine the span of the largest record pair that he or she finds interesting.

A similar method could be used for adaptive SNB. One method would be to keep the longest distance between matching pairs in each window. This will give the analyst valuable information about what the minimum window size should be.

Bibliography

- [1] H. Altwaijry, S. Mehrotra, and D. V. Kalashnikov. Query: a framework for integrating entity resolution with query processing. *Proceedings of the VLDB Endowment*, 9(3):120–131, 2015.
- [2] A. Arasu, M. Götz, and R. Kaushik. On active learning of record matching packages. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 783–794. ACM, 2010.
- [3] T. Bachteler, J. Reiher, and R. Schnell. Similarity filtering with multibit trees for record linkage. *German Record Linkage Center, Nuremberg, Working Paper WP-GRLC-2013-02*, 2013.
- [4] R. Baxter, P. Christen, and T. Churches. A comparison of fast blocking methods for record linkage. In *ACM SIGKDD*, volume 3, pages 25–27. Citeseer, 2003.
- [5] P. Christen. *Data matching: concepts and techniques for record linkage, entity resolution, and duplicate detection*. Springer Berlin Heidelberg, 2012.
- [6] P. Christen. A survey of indexing techniques for scalable record linkage and deduplication. *Knowledge and Data Engineering, IEEE Transactions on*, 24(9):1537–1555, 2012.
- [7] P. Christen, T. Churches, and M. Hegland. Febrl—a parallel open source data linkage system. In *Advances in knowledge discovery and data mining*, pages 638–647. Springer, 2004.
- [8] D. J. DeWitt, J. F. Naughton, and D. A. Schneider. Parallel sorting on a shared-nothing architecture using probabilistic splitting. In *Parallel and distributed information systems, 1991., proceedings of the first international conference on*, pages 280–291. IEEE, 1991.
- [9] A. Doan, A. Halevy, and Z. Ives. *Principles of data integration*. Elsevier, 2012.
- [10] X. L. Dong and D. Srivastava. Big data integration. *Synthesis Lectures on Data Management*, 7(1):1–198, 2015.
- [11] U. Draisbach, F. Naumann, S. Szott, and O. Wonneberg. Adaptive windows for duplicate detection. In *Data Engineering (ICDE), 2012 IEEE 28th International Conference on*, pages 1073–1083. IEEE, 2012.

- [12] D. Duplyakin, R. Ricci, A. Maricq, G. Wong, J. Duerig, E. Eide, L. Stoller, M. Hibler, D. Johnson, K. Webb, A. Akella, K. Wang, G. Ricart, L. Landweber, C. Elliott, M. Zink, E. Cecchet, S. Kar, and P. Mishra. The design and operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 1–14, July 2019.
- [13] A. K. Elmagarmid, P. G. Ipeirotis, and V. S. Verykios. Duplicate record detection: A survey. *Knowledge and Data Engineering, IEEE Transactions on*, 19(1):1–16, 2007.
- [14] A. Ferro, R. Giugno, P. L. Puglisi, and A. Pulvirenti. An efficient duplicate record detection using q-grams array inverted index. In *Data Warehousing and Knowledge Discovery*, pages 309–323. Springer, 2010.
- [15] C. Gokhale, S. Das, A. Doan, J. F. Naughton, N. Rampalli, J. Shavlik, and X. Zhu. Corleone: hands-off crowdsourcing for entity matching. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 601–612. ACM, 2014.
- [16] M. A. Hernández and S. J. Stolfo. The merge/purge problem for large databases. *ACM SIGMOD Record*, 24(2):127–138, 1995.
- [17] R. Isele, A. Jentzsch, and C. Bizer. Efficient multidimensional blocking for link discovery without losing recall. In *WebDB*, 2011.
- [18] L. Kolb and E. Rahm. Parallel entity resolution with dedoop. *Datenbank-Spektrum*, 13(1):23–32, 2013.
- [19] L. Kolb, A. Thor, and E. Rahm. Dedoop: efficient deduplication with hadoop. *Proceedings of the VLDB Endowment*, 5(12):1878–1881, 2012.
- [20] L. Kolb, A. Thor, and E. Rahm. Load balancing for mapreduce-based entity resolution. In *Data Engineering (ICDE), 2012 IEEE 28th International Conference on*, pages 618–629. IEEE, 2012.
- [21] L. Kolb, A. Thor, and E. Rahm. Multi-pass sorted neighborhood blocking with mapreduce. *Computer Science-Research and Development*, 27(1):45–63, 2012.
- [22] L. Kolb, A. Thor, and E. Rahm. Don’t match twice: redundancy-free similarity computation with mapreduce. In *Proceedings of the Second Workshop on Data Analytics in the Cloud*, pages 1–5. ACM, 2013.
- [23] 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. *Proceedings of the VLDB*, 2016.
- [24] H. Liang, Y. Wang, P. Christen, and R. Gayler. Noise-tolerant approximate blocking for dynamic real-time entity resolution. In *Advances in Knowledge Discovery and Data Mining*, pages 449–460. Springer, 2014.

- [25] A. McCallum, K. Nigam, and L. H. Ungar. Efficient clustering of high-dimensional data sets with application to reference matching. In *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 169–178. ACM, 2000.
- [26] F. McSherry, M. Isard, and D. G. Murray. Scalability! but at what cost? In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, 2015.
- [27] D. G. Mestre, C. E. S. Pires, and D. C. Nascimento. Towards the efficient parallelization of multi-pass adaptive blocking for entity matching. *Journal of Parallel and Distributed Computing*, 101:27–40, 2017.
- [28] H. B. Newcombe and J. M. Kennedy. Record linkage: Making maximum use of the discriminating power of identifying information. *Commun. ACM*, 5(11):563–566, Nov. 1962.
- [29] H. B. Newcombe, J. M. Kennedy, S. Axford, and A. P. James. Automatic linkage of vital records computers can be used to extract “follow-up” statistics of families from files of routine records. *Science*, 130(3381):954–959, 1959.
- [30] O. O’Malley. Terabyte sort on apache hadoop. *Yahoo, available online at: <http://sortbenchmark.org/Yahoo-Hadoop.pdf>*, (May), pages 1–3, 2008.
- [31] G. Papadakis, G. Alexiou, G. Papastefanatos, and G. Koutrika. Schema-agnostic vs schema-based configurations for blocking methods on homogeneous data. *Proceedings of the VLDB Endowment*, 9(4):312–323, 2015.
- [32] G. Papadakis, E. Ioannou, C. Niederée, T. Palpanas, and W. Nejdl. Eliminating the redundancy in blocking-based entity resolution methods. In *Proceedings of the 11th annual international ACM/IEEE joint conference on Digital libraries*, pages 85–94. ACM, 2011.
- [33] G. Papadakis, G. Koutrika, T. Palpanas, and W. Nejdl. Meta-blocking: Taking entity resolution to the next level. *Knowledge and Data Engineering, IEEE Transactions on*, 26(8):1946–1960, 2014.
- [34] W. E. Winkler. Approximate string comparator search strategies for very large administrative lists. *Statistics*, 2, 2005.
- [35] C. Xiao, W. Wang, X. Lin, J. X. Yu, and G. Wang. Efficient similarity joins for near-duplicate detection. *ACM Transactions on Database Systems (TODS)*, 36(3):15, 2011.
- [36] S. Yan, D. Lee, M.-Y. Kan, and L. C. Giles. Adaptive sorted neighborhood methods for efficient record linkage. In *Proceedings of the 7th ACM/IEEE-CS joint conference on Digital libraries*, pages 185–194. ACM, 2007.

APPENDIX

A

A.1 Datasets used

For my implementation, I converted each dataset from the form provided provided format into JSON format. Each tuple in the dataset is individually JSON encoded and separated by newlines. As often the datasets came with inherent skew from the dataset provider, during conversion to JSON format I randomized the order of each dataset to ensure there was no inherent by-record skew. The exact same order was used for all experiments.

A.1.1 Kaggle Movies

The first dataset was the Movies dataset from Kaggle (45K entries). The each record contains a large number of attributes, making the per-record size quite large. An example entry is as follows, newlines and spaces have been added for readability:

```
{
  "status" : "Released",
  "video" : "False",
  "vote_count" : "5415",
  "production_countries" :
    "[{'iso_3166_1': 'US', 'name': 'United States of America'}]",
  "original_language" : "en",
  "original_title" : "Toy Story",
  "runtime" : "81.0",
  "title" : "Toy Story",
  "homepage" : "http://toystory.disney.com/toy-story",
  "spoken_languages" : "[{'iso_639_1': 'en', 'name': 'English'}]",
  "poster_path" : "/rhIRbceoE9lR4veEXuwCC2wARtG.jpg",
  "revenue" : "373554033",
  "release_date" : "1995-10-30",
```

```

"belongs_to_collection" : "{
  'id': 10194,
  'name': 'Toy Story Collection',
  'poster_path': '/7G9915LfUQ2lVfwMEehDsn3kT4B.jpg',
  'backdrop_path': '/9FBwqcd9IRruEDUrTdcaafOMKUq.jpg'}",
"production_companies" :
  "[{'name': 'Pixar Animation Studios', 'id': 3}]",
"popularity" : "21.946943",
"genres" : "[{
  'id': 16,
  'name': 'Animation'},
{'id': 35,
  'name': 'Comedy'},
{'id': 10751,
  'name': 'Family'}]",
"vote_average" : "7.7",
"crew" : "[{'credit_id': '52fe4284c3a36847f8024f49',
'department': 'Directing',
'gender': 2,
'id': 7879,
'job': 'Director',
'name': 'John Lasseter',
'profile_path': '/7EdqiNbr4FRjIhKHYPdFfEEEEFG.jpg'},
{'credit_id': '52fe4284c3a36847f8024f4f',
'department': 'Writing',
'gender': 2,
'id': 12891,
'job': 'Screenplay',
'name': 'Joss Whedon',
'profile_path': '/dTIVsuaTVTeGmvkhcyJvKp2A5kr.jpg'},
{'credit_id': '52fe4284c3a36847f8024f55',
'department': 'Writing',
'gender': 2,
'id': 7,
'job': 'Screenplay',
'name': 'Andrew Stanton',
'profile_path': '/pvQWsu0qc8JFQhMVJkTHuexUAa1.jpg'},
{'credit_id': '52fe4284c3a36847f8024f5b',
'department': 'Writing',
'gender': 2,
'id': 12892,
'job': 'Screenplay',

```

```

'name': 'Joel Cohen',
'profile_path': '/dAubAiZcvKFbboWlj7oX0kZnTSu.jpg'},
{'credit_id': '52fe4284c3a36847f8024f61',
'department': 'Writing',
'gender': 0,
'id': 12893,
'job': 'Screenplay',
'name': 'Alec Sokolow',
'profile_path': '/v79vIRYi94BZUQnkkyznbGUZLjT.jpg'},
{'credit_id': '52fe4284c3a36847f8024f67',
'department': 'Production',
'gender': 1,
'id': 12894,
'job': 'Producer',
'name': 'Bonnie Arnold',
'profile_path': None},
{'credit_id': '52fe4284c3a36847f8024f6d',
'department': 'Production',
'gender': 0,
'id': 12895,
'job': 'Executive Producer',
'name': 'Ed Catmull',
'profile_path': None},
{'credit_id': '52fe4284c3a36847f8024f73',
'department': 'Production',
'gender': 2,
'id': 12896,
'job': 'Producer',
'name': 'Ralph Guggenheim',
'profile_path': None},
{'credit_id': '52fe4284c3a36847f8024f79',
'department': 'Production',
'gender': 2,
'id': 12897,
'job': 'Executive Producer',
'name': 'Steve Jobs',
'profile_path': '/mOMP3SwD5qWQSR0ldCIByd3guTV.jpg'},
{'credit_id': '52fe4284c3a36847f8024f8b',
'department': 'Editing',
'gender': 2,
'id': 8,
'job': 'Editor',

```

```

'name': 'Lee Unkrich',
'profile_path': '/bdTCCXjgOV3YyaNmLGYG0xFQM0c.jpg'},
{'credit_id': '52fe4284c3a36847f8024f91',
'department': 'Art',
'gender': 2,
'id': 7883,
'job': 'Art Direction',
'name': 'Ralph Eggleston',
'profile_path': '/uUfcGKDsK01aROMpXR67Hn6RvR.jpg'},
{'credit_id': '598331bf925141421201044b',
'department': 'Editing',
'gender': 2,
'id': 1168870,
'job': 'Editor',
'name': 'Robert Gordon',
'profile_path': None},
{'credit_id': '5892168cc3a36809660095f9',
'department': 'Sound',
'gender': 0,
'id': 1552883,
'job': 'Foley Editor',
'name': 'Mary Helen Leasman',
'profile_path': None},
{'credit_id': '5531824d9251415289000945',
'department': 'Visual Effects',
'gender': 0,
'id': 1453514,
'job': 'Animation',
'name': 'Kim Blanchette',
'profile_path': None},
{'credit_id': '589215969251412dcb009bf6',
'department': 'Sound',
'gender': 0,
'id': 1414182,
'job': 'ADR Editor',
'name': 'Marilyn McCoppen',
'profile_path': None},
{'credit_id': '589217099251412dc500a018',
'department': 'Sound',
'gender': 2,
'id': 7885,
'job': 'Orchestrator',

```

```

'name': 'Randy Newman',
'profile_path': '/w0JzfoiM25nrnxY0zosPHRq6mlE.jpg'},
{'credit_id': '5693e6b29251417b0e0000e3',
'department': 'Editing',
'gender': 0,
'id': 1429549,
'job': 'Color Timer',
'name': 'Dale E. Grahm',
'profile_path': None},
{'credit_id': '572e2522c3a36869e6001a9c',
'department': 'Visual Effects',
'gender': 0,
'id': 7949,
'job': 'CG Painter',
'name': 'Robin Cooper',
'profile_path': None},
{'credit_id': '574f12309251415ca1000012',
'department': 'Writing',
'gender': 2,
'id': 7879,
'job': 'Original Story',
'name': 'John Lasseter',
'profile_path': '/7EdqiNbr4FRjIhKHYPpDfEEFEFG.jpg'},
{'credit_id': '574f1240c3a3682e7300001c',
'department': 'Writing',
'gender': 2,
'id': 12890,
'job': 'Original Story',
'name': 'Pete Docter',
'profile_path': '/r6ngPgnReA3RHmKjmSoVsc6Awjp.jpg'},
{'credit_id': '574f12519251415c92000015',
'department': 'Writing',
'gender': 0,
'id': 7911,
'job': 'Original Story',
'name': 'Joe Ranft',
'profile_path': '/f1BoWC2JbCcP1e5hKfGsxxHzVU.jpg'},
{'credit_id': '574f12cec3a3682e82000022',
'department': 'Crew',
'gender': 0,
'id': 1629419,
'job': 'Post Production Supervisor',

```

```

'name': 'Patsy Bouge',
'profile_path': None},
{'credit_id': '574f14f19251415ca1000082',
'department': 'Art',
'gender': 0,
'id': 7961,
'job': 'Sculptor',
'name': 'Norm DeCarlo',
'profile_path': None},
{'credit_id': '5751ae4bc3a3683772002b7f',
'department': 'Visual Effects',
'gender': 2,
'id': 12905,
'job': 'Animation Director',
'name': 'Ash Brannon',
'profile_path': '/6ueWgPEEBHvS3De2BHYQnYjRTig.jpg'},
{'credit_id': '5891edbe9251412dc5007cd6',
'department': 'Sound',
'gender': 2,
'id': 7885,
'job': 'Music',
'name': 'Randy Newman',
'profile_path': '/w0JzfoiM25nrnxY0zosPHRq6mlE.jpg'},
{'credit_id': '589213d39251412dc8009832',
'department': 'Directing',
'gender': 0,
'id': 1748707,
'job': 'Layout',
'name': 'Roman Figun',
'profile_path': None},
{'credit_id': '5892173dc3a3680968009351',
'department': 'Sound',
'gender': 2,
'id': 4949,
'job': 'Orchestrator',
'name': 'Don Davis',
'profile_path': None},
{'credit_id': '589217cec3a3686b0a0052ba',
'department': 'Sound',
'gender': 0,
'id': 1372885,
'job': 'Music Editor',

```

```

'name': 'James Flamberg',
'profile_path': None},
{'credit_id': '58921831c3a3686348004a64',
'department': 'Editing',
'gender': 0,
'id': 1739962,
'job': 'Negative Cutter',
'name': 'Mary Beth Smith',
'profile_path': None},
{'credit_id': '58921838c3a36809700096c0',
'department': 'Editing',
'gender': 0,
'id': 1748513,
'job': 'Negative Cutter',
'name': 'Rick Mackay',
'profile_path': None},
{'credit_id': '589218429251412dd1009d1b',
'department': 'Art',
'gender': 0,
'id': 1458006,
'job': 'Title Designer',
'name': 'Susan Bradley',
'profile_path': None},
{'credit_id': '5891ed99c3a3680966007670',
'department': 'Crew',
'gender': 0,
'id': 1748557,
'job': 'Supervising Technical Director',
'name': 'William Reeves',
'profile_path': None},
{'credit_id': '5891edcec3a3686b0a002eb2',
'department': 'Sound',
'gender': 2,
'id': 7885,
'job': 'Songs',
'name': 'Randy Newman',
'profile_path': '/w0JzfoiM25nrnxY0zosPHRq6mlE.jpg'},
{'credit_id': '5891edf9c3a36809700075e6',
'department': 'Writing',
'gender': 2,
'id': 7,
'job': 'Original Story',

```

```

'name': 'Andrew Stanton',
'profile_path': '/pvQWsu0qc8JFQhMVJkTHuexUAa1.jpg'},
{'credit_id': '58920f0b9251412dd7009104',
'department': 'Crew',
'gender': 2,
'id': 12890,
'job': 'Supervising Animator',
'name': 'Pete Docter',
'profile_path': '/r6ngPgnReA3RHmKjmSoVsc6Awjp.jpg'},
{'credit_id': '58920f1fc3a3680977009021',
'department': 'Sound',
'gender': 2,
'id': 2216,
'job': 'Sound Designer',
'name': 'Gary Rydstrom',
'profile_path': '/jZprinVf071ldWIOYtmP1FGw7Rj.jpg'},
{'credit_id': '58920f389251412dd700912d',
'department': 'Production',
'gender': 0,
'id': 12909,
'job': 'Production Supervisor',
'name': 'Karen Robert Jackson',
'profile_path': None},
{'credit_id': '58920fbd9251412dcb00969c',
'department': 'Crew',
'gender': 0,
'id': 953331,
'job': 'Executive Music Producer',
'name': 'Chris Montan',
'profile_path': None},
{'credit_id': '589210069251412dd7009219',
'department': 'Visual Effects',
'gender': 0,
'id': 7893,
'job': 'Animation Director',
'name': 'Rich Quade',
'profile_path': None},
{'credit_id': '589210329251412dcd00943b',
'department': 'Visual Effects',
'gender': 0,
'id': 8025,
'job': 'Animation',

```

```

'name': 'Michael Berenstein',
'profile_path': None},
{'credit_id': '5892103bc3a368096a009180',
'department': 'Visual Effects',
'gender': 0,
'id': 78009,
'job': 'Animation',
'name': 'Colin Brady',
'profile_path': None},
{'credit_id': '5892105dc3a3680968008db2',
'department': 'Visual Effects',
'gender': 0,
'id': 1748682,
'job': 'Animation',
'name': 'Davey Crockett Feiten',
'profile_path': None},
{'credit_id': '589210669251412dcd009466',
'department': 'Visual Effects',
'gender': 0,
'id': 1454030,
'job': 'Animation',
'name': 'Angie Glocka',
'profile_path': None},
{'credit_id': '5892107c9251412dd1009613',
'department': 'Visual Effects',
'gender': 0,
'id': 1748683,
'job': 'Animation',
'name': 'Rex Grignon',
'profile_path': None},
{'credit_id': '5892108ac3a3680973008d3f',
'department': 'Visual Effects',
'gender': 0,
'id': 1748684,
'job': 'Animation',
'name': 'Tom K. Gurney',
'profile_path': None},
{'credit_id': '58921093c3a3686348004477',
'department': 'Visual Effects',
'gender': 2,
'id': 8029,
'job': 'Animation',

```

```

'name': 'Jimmy Hayward',
'profile_path': '/lTDRpudEY7BDwTefXbXzMlmb0ui.jpg'},
{'credit_id': '5892109b9251412dcd0094b0',
'department': 'Visual Effects',
'gender': 0,
'id': 1426773,
'job': 'Animation',
'name': 'Hal T. Hickel',
'profile_path': None},
{'credit_id': '589210a29251412dc5009a29',
'department': 'Visual Effects',
'gender': 0,
'id': 8035,
'job': 'Animation',
'name': 'Karen Kiser',
'profile_path': None},
{'credit_id': '589210ccc3a3680977009191',
'department': 'Visual Effects',
'gender': 0,
'id': 1748688,
'job': 'Animation',
'name': 'Anthony B. LaMolinara',
'profile_path': None},
{'credit_id': '589210d7c3a3686b0a004c1f',
'department': 'Visual Effects',
'gender': 0,
'id': 587314,
'job': 'Animation',
'name': 'Guionne Leroy',
'profile_path': None},
{'credit_id': '589210e1c3a36809770091a7',
'department': 'Visual Effects',
'gender': 2,
'id': 7918,
'job': 'Animation',
'name': 'Bud Luckey',
'profile_path': '/pcCh7G19FKMNijmPQg1PMH1btic.jpg'},
{'credit_id': '589210ee9251412dc200978a',
'department': 'Visual Effects',
'gender': 0,
'id': 1748689,
'job': 'Animation',

```

```

'name': 'Les Major',
'profile_path': None},
{'credit_id': '589210fa9251412dc8009595',
'department': 'Visual Effects',
'gender': 2,
'id': 7892,
'job': 'Animation',
'name': 'Glenn McQueen',
'profile_path': None},
{'credit_id': '589211029251412dc8009598',
'department': 'Visual Effects',
'gender': 0,
'id': 555795,
'job': 'Animation',
'name': 'Mark Oftedal',
'profile_path': None},
{'credit_id': '5892110b9251412dc800959d',
'department': 'Visual Effects',
'gender': 2,
'id': 7882,
'job': 'Animation',
'name': 'Jeff Pidgeon',
'profile_path': '/yLddkg5HcgbJg00cS13GVBnPOHY.jpg'},
{'credit_id': '58921113c3a36863480044e4',
'department': 'Visual Effects',
'gender': 0,
'id': 8017,
'job': 'Animation',
'name': 'Jeff Pratt',
'profile_path': None},
{'credit_id': '5892111c9251412dcb0097e9',
'department': 'Visual Effects',
'gender': 0,
'id': 1184140,
'job': 'Animation',
'name': 'Steve Rabatich',
'profile_path': None},
{'credit_id': '58921123c3a36809700090f6',
'department': 'Visual Effects',
'gender': 0,
'id': 8049,
'job': 'Animation',

```

```

'name': 'Roger Rose',
'profile_path': None},
{'credit_id': '5892112b9251412dcb0097fb',
'department': 'Visual Effects',
'gender': 0,
'id': 1509559,
'job': 'Animation',
'name': 'Steve Segal',
'profile_path': None},
{'credit_id': '589211349251412dc80095c3',
'department': 'Visual Effects',
'gender': 0,
'id': 1748691,
'job': 'Animation',
'name': 'Doug Sheppeck',
'profile_path': None},
{'credit_id': '5892113cc3a3680970009106',
'department': 'Visual Effects',
'gender': 0,
'id': 8050,
'job': 'Animation',
'name': 'Alan Sperling',
'profile_path': None},
{'credit_id': '58921148c3a3686b0a004c99',
'department': 'Visual Effects',
'gender': 0,
'id': 8010,
'job': 'Animation',
'name': 'Doug Sweetland',
'profile_path': None},
{'credit_id': '58921150c3a3680966009125',
'department': 'Visual Effects',
'gender': 0,
'id': 8044,
'job': 'Animation',
'name': 'David Tart',
'profile_path': None},
{'credit_id': '589211629251412dc5009b00',
'department': 'Visual Effects',
'gender': 0,
'id': 1454034,
'job': 'Animation',

```

```

'name': 'Ken Willard',
'profile_path': None},
{'credit_id': '589211c1c3a3686b0a004d28',
'department': 'Visual Effects',
'gender': 0,
'id': 7887,
'job': 'Visual Effects Supervisor',
'name': 'Thomas Porter',
'profile_path': None},
{'credit_id': '589211d4c3a3680968008ed9',
'department': 'Visual Effects',
'gender': 0,
'id': 1406878,
'job': 'Visual Effects',
'name': 'Mark Thomas Henne',
'profile_path': None},
{'credit_id': '589211f59251412dd4008e65',
'department': 'Visual Effects',
'gender': 0,
'id': 1748698,
'job': 'Visual Effects',
'name': 'Oren Jacob',
'profile_path': None},
{'credit_id': '58921242c3a368096a00939b',
'department': 'Visual Effects',
'gender': 0,
'id': 1748699,
'job': 'Visual Effects',
'name': 'Darwyn Peachey',
'profile_path': None},
{'credit_id': '5892124b9251412dc5009bd2',
'department': 'Visual Effects',
'gender': 0,
'id': 1748701,
'job': 'Visual Effects',
'name': 'Mitch Prater',
'profile_path': None},
{'credit_id': '58921264c3a3686b0a004dbf',
'department': 'Visual Effects',
'gender': 0,
'id': 1748703,
'job': 'Visual Effects',

```

```

'name': 'Brian M. Rosen',
'profile_path': None},
{'credit_id': '589212709251412dcd009676',
'department': 'Lighting',
'gender': 1,
'id': 12912,
'job': 'Lighting Supervisor',
'name': 'Sharon Calahan',
'profile_path': None},
{'credit_id': '5892127fc3a3686b0a004de5',
'department': 'Lighting',
'gender': 0,
'id': 7899,
'job': 'Lighting Supervisor',
'name': 'Galyn Susman',
'profile_path': None},
{'credit_id': '589212cdc3a3680970009268',
'department': 'Visual Effects',
'gender': 0,
'id': 12915,
'job': 'CG Painter',
'name': 'William Cone',
'profile_path': None},
{'credit_id': '5892130f9251412dc8009791',
'department': 'Art',
'gender': 0,
'id': 1748705,
'job': 'Sculptor',
'name': 'Shelley Daniels Lekven',
'profile_path': None},
{'credit_id': '5892131c9251412dd4008f4c',
'department': 'Visual Effects',
'gender': 2,
'id': 7889,
'job': 'Character Designer',
'name': 'Bob Pauley',
'profile_path': None},
{'credit_id': '589213249251412dd100987b',
'department': 'Visual Effects',
'gender': 2,
'id': 7918,
'job': 'Character Designer',

```

```

'name': 'Bud Luckey',
'profile_path': '/pcCh7G19FKMNijmPQg1PMH1btic.jpg'},
{'credit_id': '5892132b9251412dc80097b1',
'department': 'Visual Effects',
'gender': 2,
'id': 7,
'job': 'Character Designer',
'name': 'Andrew Stanton',
'profile_path': '/pvQWsu0qc8JFQhMVJkTHuexUAa1.jpg'},
{'credit_id': '58921332c3a368634800467b',
'department': 'Visual Effects',
'gender': 0,
'id': 12915,
'job': 'Character Designer',
'name': 'William Cone',
'profile_path': None},
{'credit_id': '5892135f9251412dd4008f90',
'department': 'Visual Effects',
'gender': 0,
'id': 1748706,
'job': 'Character Designer',
'name': 'Steve Johnson',
'profile_path': None},
{'credit_id': '58921384c3a3680973008fd4',
'department': 'Visual Effects',
'gender': 0,
'id': 1176752,
'job': 'Character Designer',
'name': 'Dan Haskett',
'profile_path': None},
{'credit_id': '5892138e9251412dc20099fc',
'department': 'Visual Effects',
'gender': 0,
'id': 1088034,
'job': 'Character Designer',
'name': 'Tom Holloway',
'profile_path': '/a0r0T2usTBpgMI5aZbRBDW1fTl8.jpg'},
{'credit_id': '58921395c3a368097700942f',
'department': 'Visual Effects',
'gender': 0,
'id': 1447465,
'job': 'Character Designer',

```

```

'name': 'Jean Gillmore',
'profile_path': None},
{'credit_id': '589213e2c3a3680973009026',
'department': 'Directing',
'gender': 0,
'id': 1748709,
'job': 'Layout',
'name': 'Desirée Mourad',
'profile_path': None},
{'credit_id': '589214099251412dc5009d57',
'department': 'Art',
'gender': 0,
'id': 1748710,
'job': 'Set Dresser',
'name': '\"Kelly O'Connell\"',
'profile_path': None},
{'credit_id': '58921411c3a3686b0a004f70',
'department': 'Art',
'gender': 0,
'id': 1443471,
'job': 'Set Dresser',
'name': 'Sonoko Konishi',
'profile_path': None},
{'credit_id': '58921434c3a368096a00956e',
'department': 'Art',
'gender': 0,
'id': 1748711,
'job': 'Set Dresser',
'name': 'Ann M. Rockwell',
'profile_path': None},
{'credit_id': '5892144ac3a36809680090de',
'department': 'Editing',
'gender': 0,
'id': 1748712,
'job': 'Editorial Manager',
'name': 'Julie M. McDonald',
'profile_path': None},
{'credit_id': '58921479c3a368096800910f',
'department': 'Editing',
'gender': 0,
'id': 1589729,
'job': 'Assistant Editor',

```

```

'name': 'Robin Lee',
'profile_path': None},
{'credit_id': '5892148b9251412dd10099cc',
'department': 'Editing',
'gender': 0,
'id': 1748716,
'job': 'Assistant Editor',
'name': 'Tom Freeman',
'profile_path': None},
{'credit_id': '589214959251412dcb009b1f',
'department': 'Editing',
'gender': 0,
'id': 1748717,
'job': 'Assistant Editor',
'name': 'Ada Cochavi',
'profile_path': None},
{'credit_id': '5892149ec3a3686348004798',
'department': 'Editing',
'gender': 0,
'id': 1336438,
'job': 'Assistant Editor',
'name': 'Dana Mulligan',
'profile_path': None},
{'credit_id': '589214adc3a368096a0095db',
'department': 'Editing',
'gender': 0,
'id': 1748718,
'job': 'Editorial Coordinator',
'name': 'Deirdre Morrison',
'profile_path': None},
{'credit_id': '589214c7c3a368097700952b',
'department': 'Production',
'gender': 0,
'id': 1748719,
'job': 'Production Coordinator',
'name': 'Lori Lombardo',
'profile_path': None},
{'credit_id': '589214cec3a368096a009603',
'department': 'Production',
'gender': 0,
'id': 1748720,
'job': 'Production Coordinator',

```

```

'name': 'Ellen Devine',
'profile_path': None},
{'credit_id': '589214e39251412dc8009904',
'department': 'Crew',
'gender': 0,
'id': 1468014,
'job': 'Unit Publicist',
'name': 'Lauren Beth Strogoff',
'profile_path': None},
{'credit_id': '58921544c3a3686b0a00507d',
'department': 'Sound',
'gender': 2,
'id': 2216,
'job': 'Sound Re-Recording Mixer',
'name': 'Gary Rydstrom',
'profile_path': '/jZprinVf071ldWIOYtmP1FGw7Rj.jpg'},
{'credit_id': '5892154c9251412dd1009a56',
'department': 'Sound',
'gender': 0,
'id': 1425978,
'job': 'Sound Re-Recording Mixer',
'name': 'Gary Summers',
'profile_path': None},
{'credit_id': '58921555c3a36809680091bd',
'department': 'Sound',
'gender': 2,
'id': 8276,
'job': 'Supervising Sound Editor',
'name': 'Tim Holland',
'profile_path': None},
{'credit_id': '589215c39251412dcb009c12',
'department': 'Sound',
'gender': 0,
'id': 7069,
'job': 'Sound Effects Editor',
'name': 'Pat Jackson',
'profile_path': None},
{'credit_id': '58921698c3a368096a009788',
'department': 'Crew',
'gender': 2,
'id': 15894,
'job': 'Sound Design Assistant',

```

```

'name': 'Tom Myers',
'profile_path': None},
{'credit_id': '589216a89251412dc2009ca4',
'department': 'Sound',
'gender': 0,
'id': 1414177,
'job': 'Assistant Sound Editor',
'name': 'J.R. Grubbs',
'profile_path': None},
{'credit_id': '589216c19251412dc2009cb9',
'department': 'Sound',
'gender': 1,
'id': 1748724,
'job': 'Assistant Sound Editor',
'name': 'Susan Sanford',
'profile_path': None},
{'credit_id': '589216ccc3a3680973009274',
'department': 'Sound',
'gender': 0,
'id': 1748725,
'job': 'Assistant Sound Editor',
'name': 'Susan Popovic',
'profile_path': None},
{'credit_id': '589216d79251412dc8009aa0',
'department': 'Sound',
'gender': 0,
'id': 8067,
'job': 'Assistant Sound Editor',
'name': 'Dan Engstrom',
'profile_path': None},
{'credit_id': '589216e49251412dcd009a4f',
'department': 'Production',
'gender': 1,
'id': 7902,
'job': 'Casting Consultant',
'name': 'Ruth Lambert',
'profile_path': None},
{'credit_id': '589216f39251412dc2009cf3',
'department': 'Production',
'gender': 0,
'id': 84493,
'job': 'ADR Voice Casting',

```

```

'name': 'Mickie McGowan',
'profile_path': '/k7TjJBfINsg8vLQxJwos6XObAD6.jpg'}]]",
  "overview" : "Led by Woody, Andy's toys live happily in his room until
Andy's birthday brings Buzz Lightyear onto the scene. Afraid of losing his
place in Andy's heart, Woody plots against Buzz. But when circumstances
separate Buzz and Woody from their owner, the duo eventually learns to put
aside their differences.",
  "tagline" : "",
  "adult" : "False",
  "cast" : "[{'cast_id': 14,
'character': 'Woody (voice)',
'credit_id': '52fe4284c3a36847f8024f95',
'gender': 2,
'id': 31,
'name': 'Tom Hanks',
'order': 0,
'profile_path': '/pQFoyx7rp09CJTAb932F2g8Nlho.jpg'},
{'cast_id': 15,
'character': 'Buzz Lightyear (voice)',
'credit_id': '52fe4284c3a36847f8024f99',
'gender': 2,
'id': 12898,
'name': 'Tim Allen',
'order': 1,
'profile_path': '/uX2xVf6pMmPepxnvFWyBtjexzgY.jpg'},
{'cast_id': 16,
'character': 'Mr. Potato Head (voice)',
'credit_id': '52fe4284c3a36847f8024f9d',
'gender': 2,
'id': 7167,
'name': 'Don Rickles',
'order': 2,
'profile_path': '/h5BcaDMPRVLHLDzbQavec4xfSdt.jpg'},
{'cast_id': 17,
'character': 'Slinky Dog (voice)',
'credit_id': '52fe4284c3a36847f8024fa1',
'gender': 2,
'id': 12899,
'name': 'Jim Varney',
'order': 3,
'profile_path': '/eIo2jVVXYgjDtaHoF19Ll9vtW7h.jpg'},
{'cast_id': 18,
```

```

'character': 'Rex (voice)',
'credit_id': '52fe4284c3a36847f8024fa5',
'gender': 2,
'id': 12900,
'name': 'Wallace Shawn',
'order': 4,
'profile_path': '/oGE6JqPP2xH4tNORKNqxbNPYi7u.jpg'},
{'cast_id': 19,
'character': 'Hamm (voice)',
'credit_id': '52fe4284c3a36847f8024fa9',
'gender': 2,
'id': 7907,
'name': 'John Ratzenberger',
'order': 5,
'profile_path': '/yGechiKWL6TJDfVE2KPSJYqdMsY.jpg'},
{'cast_id': 20,
'character': 'Bo Peep (voice)',
'credit_id': '52fe4284c3a36847f8024fad',
'gender': 1,
'id': 8873,
'name': 'Annie Potts',
'order': 6,
'profile_path': '/eryXT84RL41jHSJcMy4kS3u9y6w.jpg'},
{'cast_id': 26,
'character': 'Andy (voice)',
'credit_id': '52fe4284c3a36847f8024fc1',
'gender': 0,
'id': 1116442,
'name': 'John Morris',
'order': 7,
'profile_path': '/vYGyvK4LzeaUCoNSHtsuqJUY15M.jpg'},
{'cast_id': 22,
'character': 'Sid (voice)',
'credit_id': '52fe4284c3a36847f8024fb1',
'gender': 2,
'id': 12901,
'name': 'Erik von Detten',
'order': 8,
'profile_path': '/twNF1ZaJ1FUNUuo6xLXwcxjayBE.jpg'},
{'cast_id': 23,
'character': 'Mrs. Davis (voice)',
'credit_id': '52fe4284c3a36847f8024fb5',

```

```

'gender': 1,
'id': 12133,
'name': 'Laurie Metcalf',
'order': 9,
'profile_path': '/unMMIT60eoBM2sN2nyR7EZ2BvvD.jpg'},
{'cast_id': 24,
'character': 'Sergeant (voice)',
'credit_id': '52fe4284c3a36847f8024fb9',
'gender': 2,
'id': 8655,
'name': 'R. Lee Ermey',
'order': 10,
'profile_path': '/r8GBqFBjypLUP9VVqDqfZ7wYbSs.jpg'},
{'cast_id': 25,
'character': 'Hannah (voice)',
'credit_id': '52fe4284c3a36847f8024fbd',
'gender': 1,
'id': 12903,
'name': 'Sarah Freeman',
'order': 11,
'profile_path': None},
{'cast_id': 27,
'character': 'TV Announcer (voice)',
'credit_id': '52fe4284c3a36847f8024fc5',
'gender': 2,
'id': 37221,
'name': 'Penn Jillette',
'order': 12,
'profile_path': '/zmAaXUdx12NRsssgHbk1T31j2x9.jpg'}]],
  "imdb_id" : "tt0114709",
  "id" : "862",
  "budget" : "30000000"
}

```

Blocking Key Function The BK for this dataset was: adult attribute-title-release date-runtime-id

Thus, first sort the dataset by whether this is an “adult movie” (adult movies are typically regarded to be pornography and often try to be clever and have similar-sounding titles to non-pornographic films so I sort them out first), then by the title of the film, then by the release date

of the film, and then by the runtime of the film, in minutes. Finally to break ties, sort by the id attribute.

Matching Function The matching function for this dataset was: Both movies have to either be adult movies (or not), the titles must be within an edit distance of 10%, the release date has to be within an edit distance of 10%, and the runtime has to be within 10 minutes.

A.1.2 Million Song Dataset

The second dataset is the Million Song Dataset (naturally 1M records). Here is an example entry:

```
{
  "artist_hotttnesss" : "0.394031892714",
  "duration" : "252.05506",
  "artist_name" : "Faster Pussy cat",
  "title" : "Silent Night",
  "year" : "2003",
  "artist_familiarity" : "0.649822100201",
  "id" : "1",
  "release" : "Monster Ballads X-Mas"
}
```

Blocking Key Function The BK for this dataset was: title-artist-id. Thus, first sort by title, then by artist, and then finally break ties by using the id attribute.

Matching Function The matching function for this dataset was: The edit distance of the artist name must be within 10 percent, *and* the edit distance of the song title must be within 10 percent.

A.1.3 DBLP

The next dataset was a snapshot of DBLP (4.6M entries). An example entry is as follows, again newlines and spaces have been added for readability:

```
{"@mdate": "2013-12-09",
  "@key" : "0conf/sigmod/PatelyKTNBHRLEKGDN97",
  "author": ["Jignesh M. Patel", "Jie-Bing Yu", "Navin Kabra",
```

```

        "Kristin Tufte", "Biswadeep Nag", "Josef Burger",
        "Nancy E. Hall", "Karthikeyan Ramasamy", "Roger Lueder",
        "Curt J. Ellmann", "Jim Kupsch", "Shelly Guo",
        "David J. DeWitt", "Jeffrey F. Naughton"],
"title": "Building a Scaleable Geo-Spatial DBMS: Technology,
        Implementation, and Evaluation.",
"pages": "336-347",
"year": "1997",
"crossref": "conf/sigmod/97",
"booktitle": "SIGMOD Conference",
"url": "db/conf/sigmod/sigmod97.html#PatelYKTNBHRLEKGDN97",
"ee": "http://doi.acm.org/10.1145/253260.253342",
"cdrom": "SIGMOD97/P336.PDF",
"cite": [{"@label": "Beck90", "#text": "conf/sigmod/BeckmannKSS90"},
        {"@label": "Bonc96", "#text": "conf/edbt/BonczQK96"},
        {"@label": "Care94", "#text": "conf/sigmod/CareyDFHMNSSTTWZ94"},
        {"@label": "DCW92", "#text": "..."},
        {"@label": "DeW90", "#text": "journals/tkde/DeWittGSBHR90"},
        {"@label": "DeWi92", "#text": "journals/cacm/DeWittG92"},
        {"@label": "DeWi92", "#text": "conf/vldb/DeWittNSS92"},
        {"@label": "DeWi94", "#text": "conf/vldb/DeWittKLPY94"},
        {"@label": "EOS96", "#text": "..."},
        {"@label": "Grae90", "#text": "conf/sigmod/Graefe90"},
        {"@label": "Gutm84", "#text": "conf/sigmod/Guttman84"},
        {"@label": "Hua91", "#text": "conf/vldb/HuaL91"},
        {"@label": "Info96", "#text": "..."},
        {"@label": "Illu96", "#text": "..."},
        {"@label": "Moha92", "#text": "journals/tods/MohanHLPS92"},
        {"@label": "Kab96", "#text": "..."},
        {"@label": "Kits90", "#text": "conf/vldb/KitsuregawaNT89"},
        {"@label": "Koud96", "#text": "conf/edbt/KoudasFK96"},
        {"@label": "Kim96", "#text": "..."},
        {"@label": "Pate96", "#text": "conf/sigmod/PatelD96"},
        {"@label": "Pate97", "#text": "..."},
        {"@label": "Prep88", "#text": "..."},
        {"@label": "Rous95", "#text": "conf/sigmod/RoussopoulosKV95"},
        {"@label": "Shat95", "#text": "conf/sigmod/ShatdalN95"},
        {"@label": "Shat96", "#text": "..."},
        {"@label": "Ston93", "#text": "conf/sigmod/StonebrakerFGM93"},
        {"@label": "SQL3", "#text": "..."},
        {"@label": "Suni94", "#text": "conf/icde/SarawagiS94"},
        {"@label": "Tera85", "#text": "..."}],

```

```
{ "@label": "Yu96", "#text": "conf/ssdbm/YuD97"},
  {"@label": "Walt91", "#text": "conf/vldb/WaltonDJ91"},
  {"@label": "Welc84", "#text": "journals/computer/Welch84"}],
"type": "inproceedings"}
```

Blocking Key Function The BK used for this dataset was: the title, then a dash, then the first author, then a dash, then the year of publication, then the type of article. For example, the BKV for the sample entry above is Building a Scaleable Geo-Spatial DBMS: Technology, Implementation, and Evaluation.-Jignesh M. Patel-1997.

Matching Function The matching function for this dataset was: the types must equal (and not be type ‘www’), *and* the titles must be within edit distance of 2, *and* the first author must be within edit distance of 2, then the two tuples match.

A.1.4 FEC Individual Donor Information

The second dataset I used was Individual Donor Information from the United States Federal Elections Commission (FEC) from 1980 through January 2016 (25.2M entities). The order of the entries were randomized. An example entry is as follows, again newlines and spaces have been added for readability:

```
{"SUB_ID": "4051020051057252568",
  "OCCUPATION": "PROFESSOR",
  "ENTITY_TP": "IND",
  "OTHER_ID": "",
  "TRANSACTION_AMT": "1000",
  "TRANSACTION_TP": "15",
  "IMAGE_NUM": "25990716842",
  "ZIP_CODE": "53726",
  "RPT_TP": "12G",
  "MEMO_TEXT": "",
  "AMNDT_IND": "A",
  "NAME": "DEWITT, DAVID",
  "STATE": "WI",
  "MEMO_CD": "",
  "TRANSACTION_DT": "10132004",
  "EMPLOYER": "UNIVERSITY OF WISCONSIN",
```

```
"CITY": "MADISON",
"CMTE_ID": "C00010603",
"TRAN_ID": "0014653058",
"FILE_NUM": "173455",
"TRANSACTION_PGI": ""}
```

Blocking Key Function The BK function used was, in pseudocode: the first word in the name field (typically the family name) encoded in Soundex, then a dash, then the rest of the name field encoded in Soundex, and then a dash, then the state, then the zip code. For example, the BKV for the sample entry above is D300-D130-WI-53726.

Matching Function The matching function used for this dataset was: If the names match, and the distance between the zip codes is less than 100 miles, *or* the edit distance between the names is less than 3 and the zip codes are less than 50 miles apart, *or* the edit distance between the names is less than 6 and the edit distance of the employer is less than 3 and the zip codes are less than 50 miles apart, then two tuples match.

A.1.5 Users Database

The third dataset I used was a database dump from a large commercial website (36.4M entries). Each entry was denormalized from the source database. Then, the order of the dataset was randomized and then JSON encoded. An example of the dataset is below. Newlines and spaces inserted for readability, and I have redacted some of the information with #'s. The running dataset used unredacted records.

```
{"am_am_member__nickname": "B#####9",
"am_am_member__gender": 2,
"aminno_member__gender": 2,
"aminno_member_email__email": "j#####r@#####.###",
"aminno_member__state": 52,
"am_am_member__pref_opento_other": "",
"aminno_member__adsouce": 0,
"am_am_member__street1": "3### s#####",
"am_am_member__street2": "",
"aminno_member__timezone": 0,
"aminno_member__credits": 90,
```

```

"member_details__profile_smoke": 1,
"am_am_member__pref_lookingfor_abstract": "",
"aminno_member__signupip": "",
"member_details__hair_color": 0,
"aminno_member__dob": "19##-##-##",
"aminno_member__turnsmeon": "",
"aminno_member__lookingfor_abstract": "",
"member_login__username": "b#####9",
"am_am_member__pref_opento": "|13|15|16|18|21|",
"aminno_member__photos_private": 0,
"member_details__dob": "19##-##-##",
"am_am_member__createdby": 0,
"aminno_member__approved": 1,
"member_details__profile_ethnicity": 1,
"aminno_member__ishost": 0,
"aminno_member__zip": "537##",
"aminno_member__smoking": null,
"aminno_member_email__notify_login": 1,
"aminno_member_email__isvalid": 1,
"am_am_member__updatedby": 11,
"member_details__profile_height": 1##,
"member_details__profile_caption": "I've been there and liked it",
"aminno_member__longitude": -89.#####,
"aminno_member__nickname": "B#####9",
"am_am_member__state": 52,
"am_am_member__profile_number": 2####,
"am_am_member__security_answer": "",
"member_login__password": "#####",
"am_am_member__profile_smoke": 1,
"aminno_member_email__html": 0,
"am_am_member__pref_lookingfor": "|7|9|20|21|28|29|33|",
"aminno_member_email__notify_newmail": 1,
"aminno_member__restrict_global": 0,
"aminno_member__bc_mail_last_time": null,
"am_am_member__updatedon": "2011-##-##T##:##:##",
"am_am_member__admin": 3,
"am_am_member__zip": "537##",
"am_am_member__dob": "19##-##-##",
"member_login__notify": 0,
"aminno_member__set_chat_available": 1,
"am_am_member__country": 1,
"aminno_member__flags": 0,

```

```

"aminno_member__opento": "",
"aminno_member_email__notify_offer": 1,
"aminno_member__accept_collect": 1,
"member_login__loginkey": "#####",
"aminno_member__city": "Madison",
"am_am_member__pref_turnsmeon": "|6|11|16|22|",
"aminno_member__keywords": "",
"am_am_member__main_photo": 0,
"am_am_member__pref_lookingfor_other": "",
"am_am_member__profile_drink": 1,
"am_am_member__latitude": 43.####,
"aminno_member__seeking": 2,
"am_am_member__security_question": 0,
"aminno_member__lookingfor_other": null,
"aminno_member__opento_other": null,
"aminno_member__limits": null,
"am_am_member__profile_height": 188,
"am_am_member__createdon": "2002-##-##T##:##:##",
"aminno_member__fraud_flag": 0,
"am_am_member__first_name": "J###",
"aminno_member__ethnicity": null,
"aminno_member__flatrate": null,
"am_am_member__pref_turnsmeon_abstract": "",
"am_am_member__profile_initially_seeking": 5,
"aminno_member__set_chat_enabled": 1,
"aminno_member__lookingfor": "",
"member_details__profile_initially_seeking": 5,
"am_am_member__profile_weight": 8###2,
"am_am_member__city": "Madison",
"am_am_member__profile_ethnicity": 1,
"am_am_member__longitude": 89.####,
"aminno_member_email__notify_newmember": 1,
"aminno_member__turnsmeon_abstract": "",
"aminno_member_email__optin": 9,
"am_am_member__ad_source": 0,
"member_details__profile_bodytype": 2,
"aminno_member__set_view_rated": 0,
"am_am_member__pref_turnsmeon_other": "",
"aminno_member__bodytype": null,
"aminno_member__turnsmeon_other": null,
"aminno_member__updatedon": "2013-01-24T##:##:##",
"am_am_member__last_name": "M#####",

```

```

"aminno_member__security_answer": "",
"aminno_member__mail_auto_responder": 0,
"am_am_member__account_type": 1,
"am_am_member__pref_opento_abstract": "I'm looking for
#####
#####.",
"am_am_member__status": 2,
"am_am_member__profile_bodytype": 2,
"am_am_member__work_phone": "6082####",
"aminno_member__accept_host_contact": 1,
"aminno_member__sponsor": 0,
"aminno_member__restrict_bc": 0,
"aminno_member__mail_auto_responder_msg": null,
"aminno_member__accept_mail_auto_responder": 1,
"aminno_member__opento_abstract": "",
"aminno_member__signupvid": "",
"aminno_member__set_show_online": 1,
"aminno_member__weight": null,
"am_am_member__profile_relationship": 2,
"aminno_member__hair_color": 0,
"pnun": ####,
"aminno_member__country": 1,
"aminno_member__photos_public": 0,
"aminno_member__caption": null,
"aminno_member__set_show_profile": 0,
"aminno_member__eye_color": 0,
"am_am_member__phone": "6082####",
"aminno_member__bc_chat_last_time": null,
"member_details__profile_weight": 8###2,
"am_am_member__membership_status": 1,
"aminno_member__reply_mail_last_time": null,
"member_details__eye_color": 0,
"member_details__profile_drink": 1,
"aminno_member__security_question": 0,
"am_am_member__profile_caption": "I### #### ##### ### ##### ##",
"am_am_member__mobile_phone": "",
"aminno_member__height": null,
"aminno_member__latitude": 43.#####}

```

Blocking Key Function The BK function I used was gender then a dash then the soundex of the last name then a dash then the well-known Soundex encoding of the first name, then the profile number. For example, the BKV for the sample entry above is 2-M460-J200-####.

Matching Function The matching function used for this dataset was: the last names must be within an edit distance of 10% of their length, *and* the first names must be within an edit distance of 10% of their length, *and* the zip codes must be within 100 miles, then the two tuples match.

A.2 Extending datasets

The experiments varied the amount of entries from each dataset. As needed, more entries were synthesized for each dataset. To extend a dataset, a new pass over the dataset is started, and each TID is prepended with a number representing the number of times the source dataset has been read in. This preserves the distribution of tuples, yet provides uniquely addressable entries in the dataset. The desired number of entries in the input dataset are written to a per-experiment file. This file is recreated for experiments with a different number of input entries.

A.3 Experimental setting

My experiments were performed using CloudLab [12], using the Wisconsin Cluster. Each experiment used the CloudLab Hadoop 2.7.1 profile — with one node as the Hadoop namenode, one as the resourcemanager, and ten Hadoop slaves. Each machine was a 4 processor 8 core 2.4 GHz Intel processor, with 1 TB of local disk and 128–192GB of RAM. The machines were interconnected with 10 gigabit Ethernet networking. Prior to starting each data point, system memory caches were flushed on all machines.

The algorithms were written in Python, with as much code reuse as possible between methods. For the MapReduce setting, I used MapReduce streaming with my Python code.

List of Terms

AE Attribute Equivalence. A key-based blocking method (see Section 2.1.1). 8

ASNB Adaptive SNB Method using MapReduce. 5, 6, 61, 88, 90, 92–95, 98–100, 103–106, 108, 110–112, 115, 116, 119, 120, 122

A-SNM Adaptive Sorted Neighborhood Method. An Adaptive SNB strategy for a single machine proposed in [36]. 83, 84, 87

BK Blocking Key function. Takes a tuple and returns a BKV. 2, 3, 7–9, 25, 30, 38, 40–42, 80, 81, 83, 84, 93, 148, 149, 151, 152, 156

BKV A string representation of a tuple returned from a BK. Term used in [6]. 2, 7–11, 22, 25–28, 30, 32, 38–42, 50, 51, 56, 64, 84, 111

BN Block Number. Used in RepSN to keep track of which input partition this tuple came from. 26, 27, 30

DCS++ Duplicate Count Strategy. An Adaptive SNB strategy for a single machine proposed in [11]. 80–91, 112, 115, 116, 122, 123

EM Entity Matching. 1, 2, 122

HDFS Hadoop Distributed File System. Used by Apache Hadoop’s implementation of MapReduce. 17, 21, 25–27, 29, 30, 50–52, 55, 56, 59, 61, 62, 64, 77, 105, 120

KPM Key Partition Matrix. Used in RepSN to determine how to partition data to completely avoid skew. 5, 25–27, 45, 49, 50, 71

MapReduce The Apache Hadoop MapReduce Framework. 3, 16–30, 49–57, 59–66, 68, 75, 77, 83, 94, 98, 99, 106, 108, 110, 111, 119, 122, 123

MultiMR-DCS++ Multi-pass MapReduce Duplicate Count Strategy. A MapReduce based adaptive SNB strategy proposed in [27]. 4, 84–87

MultiPAM Multi-pass Partition Allocation Matrix (MultiPAM). Used in MultiMR-DCS++. 85

RepSN A MapReduce implementation of SNB proposed in [21]. 3, 5, 6, 15, 16, 25–30, 32, 38, 43–45, 49, 50, 68, 71, 84–87, 115

SNB Sorted Neighborhood Blocking. A popular key-based blocking style. 1–6, 9, 13, 14, 16, 17, 23, 38, 43, 44, 49, 52, 61, 79, 80, 84, 85, 87–92, 94, 122, 123

SNB-MR-Simple A Simple method to compute SNB over MapReduce using off-the-shelf components. 49–51, 53, 68, 71

SNB-single A single machine implementation of SNB. 40, 41, 43, 44, 90

TID Tuple Identifier. A unique identifier of a tuple (similar to a primary key). 1, 21–23, 25, 26, 29, 32, 40, 41, 43, 50, 51, 56, 71, 90, 93, 94, 98, 108, 110, 111, 115

VSNB Vanilla, or static window sized, SNB. 4–6, 49, 53–55, 60–62, 64, 66–68, 71, 74, 75, 77, 91, 93–95, 98, 110–112, 116, 119, 122, 123