5-2015

Optimal "Big Data" Aggregation Systems - From Theory to Practical Application

William J. Culhane IV
Purdue University

Graduate School Form 30
(Revised 08/14)

PURDUE UNIVERSITY
GRADUATE SCHOOL
Thesis/Dissertation Acceptance

This is to certify that the thesis/dissertation prepared

By William John Culhane IV

Entitled
Optimal "Big Data" Aggregation Systems - From Theory to Practical Application

For the degree of Doctor of Philosophy

Is approved by the final examining committee:

Patrick Eugster

Charles Killian

Sonia Fahmy

Dongyan Xu

To the best of my knowledge and as understood by the student in the Thesis/Dissertation Agreement, Publication Delay, and Certification/Disclaimer (Graduate School Form 32), this thesis/dissertation adheres to the provisions of Purdue University's "Policy on Integrity in Research" and the use of copyrighted material.

Patrick Eugster

Approved by Major Professor(s): _____

_____

Approved by: Sunil Prabhakar/William J. Gorman 04/15/2015

Head of the Department Graduate Program Date

OPTIMAL "BIG DATA" AGGREGATION SYSTEMS – FROM THEORY TO PRACTICAL APPLICATION

A Dissertation

Submitted to the Faculty

of

Purdue University

by

William John Culhane IV

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

May 2015

Purdue University

West Lafayette, Indiana

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Culhane, William John IV PhD, Purdue University, May 2015. Optimal "Big Data" Aggregation Systems – From Theory to Practical Application. Major Professor: Patrick Eugster.

The integration of computers into many facets of our lives has made the collection and storage of staggering amounts of data feasible. However, the data on its own is not so useful to us as the analysis and manipulation which allows manageable descriptive information to be extracted. New tools to extract this information from ever growing repositories of data are required.

Some of these analyses can take the form of a two phase problem which is easily distributed to take advantage of available computing power. The first phase involves computing some descriptive partial result from some subset of the original data, and the second phase involves aggregating all the partial results to create a combined output. We formalize this *compute-aggregate* model for a rigorous performance analysis in an effort to minimize the latency of the aggregation phase with minimal intrusive analysis or modification.

Based on our model we find an aggregation overlay attribute which highly affects aggregation latency and its dependence on an easily findable trait of aggregation. We rigorously prove the dependence and find optimal overlays for aggregation. We use the proven optima to create simple heuristics and build a system, NOAH, to take advantage of the findings. NOAH can be used by big data analysis systems.

We also study an individual problem, top-k matching, to explore the effects of optimizing the computation phase separately from aggregation and create a complete distributed system to fulfill an economically relevant task.

1 INTRODUCTION

It is fair to call this the advent of the age of information. The sheer amount of data available dwarves that of even a generation ago, and the rate at which new data is generated and collected is only increasing. Data in itself is typically only a tool in a process to understand phenomena, so increased storage capabilities are insufficient to fully utilize this vast increase in available data. It is necessary to also progress our tools for analysis and processing to distill the data to their actionable core.

1.1 Distributed Big Data Processing

There is an obvious need for systems which have high availability, simplified service capabilities, and higher computational power than single machines [1]. Furthermore, the economics have increasingly pushed toward large scale datacenters [2].

Approaches to distributed data processing including Microsoft's LINQ [3] and Yahoo!'s Pig Latin [4] allow the user to not only define the structure of the data, but also to attempt to instruct the program with some guidance on program flow. This requires more effort on the part of the programmer and produces more specialized implementations. This is certainly manageable for some processes, but isn't always as scalable or simple to use.

Another idea is to create simplified data abstraction for creating powerful programmable and, more importantly, reusable frameworks. Because the process for distilling data depends on the desired outcomes and the nature of the data itself, frameworks including the Hadoop [5] implementation of MapReduce [6] create intuitive data models to allow the user to abstract the data into more manageable chunks. In the case of the MapReduce model this is via organizing the data via *key-value* pairings, in which keys are descriptive attributes for partitioning the data via *mapping*

for processing, or *reducing*, *value* data items which should be processed together. Intelligently placing data for processing in this manner allows local computations when doing the brunt of the calculations, which is dersireable for both latency and memory access reasons [7]. The frameworks themselves are flexibile enough for users to program in their familiar environments while expending little extra effort to use an efficient and robust distribution system.

For users who cannot afford to maintain their own large distributed system there are an increasing number of cloud services available for deploying these solutions [8–10]. The additional on-demand computing power with no hardware maintenance worries creates and attractive draw for many users [11].

1.2 Top-k Matching

Many times a user or process needs a list of items which fulfill a set of criteria, but the full result set is much larger than what is needed, or even potential what can be used. To solve this problem top-k matching attempts to return subset of results of a given size k such that the returned items fit some definition of "best" matching. Fagin explored this problem extensively on specialized databases [12–14], and subsequent research has pointed out the need for it in various web based services [15, 16].

In order to perform a top-k matching algorithm requires some function to score potential entities for the final list and determine which ones are the best match. Any matching model must be clear in what it allows both for clarity to the user and to set limits on the algorithm. More expressive models allow users to create more precise definitions of "best", but they often do so at the expense of disallowing performance enhancing shortcuts in the matching algorithm.

Scoring all entities and sorting them is tremendously time consuming when datasets get too large, which negatively impacts a system which is expected to be responsive. Thus a more efficient algorithm is necessary which works within the confines of the matching model.

1.3 Limitations of Current Art

Existing frameworks, especially MapReduce implementations, have proven wildly popular in practice. According to Jeff Dean in September of 2009 Google ran 3467K MapReduce jobs on 544PB of input data, a more than 100× increase over the numbers from 5 years prior, and the numbers were still growing [17]. The data and computational model is easy to manipulate to solve many problems for which it is not suited for efficient computation [18]. For example, consider the top-k problem. Mappers can find their top-k matches locally, then a reducer uses all the local results to find the top-k of the sets it receives. However, the scores from the local results are used to filter out items from other local result sets, so a single reducer is required to make sure the filtering comes to the correct result [19]. This mitigates some of the use of distribution because the second phase has to be performed at a single location.

Thus it makes sense to introduce frameworks with new distributed computation models which provide users with intuitive high level abstractions for segmenting and combining data. Rather than having only a single tool available, a user should be able to choose among multiple tools to create a system which is best suited to the type of computation being done without having to worry about implementing the distributive properties that are reused on multiple problems.

Another limitation of many aggregation specific systems is their adaptive nature, which means they are more reactive than predictive. Several current systems use metrics to determine places in the overlay which are ripe for local tuning and apply it [20–22]. While this is good for log lived systems with changing workloads, it typically does not do the best job of global tuning, and it requires some amount of activity before local optimization can be effective.

Likewise, the top-k matching solutions are very good at performing the tasks for which they were specifically designed. However, as the data collection mechanisms available grow there are several underaddressed areas the model for matching can be expanded. Current approaches typically limit matching to attributes which are

universally available, and limit the scoring function to only allow monotonic score changes. In some tasks where top-k matching makes sense, such as advertisement serving on a web platform, different pieces of data will be available for different users based on usage history, applicable laws on data collection, and privacy concerns. In those same applications some traits will be seen as positive and others as negative when creating a truly targetted match.

1.4 Our Solution

In this dissertation we approach both the problem of distributed computation and top-k matching from the model point of view first. We create mathematically rigorous definitions of the problems at hand. Based on these definitions we can clearly state our assumptions and the goals of our work in relation to the relevant variable.

In the case of distributed data aggregation the goal is minimal latency of a general purpose aggregation (sub)system for use by big data analysis engines. Through our slightly simplified model we create we are able to clearly identify the attributes of aggregation functions which affect the latency in a general purpose system without need for in depth analysis of each aggregation function. By fully exploring the model we find provably optimal aggregation overlays for a variety of cases.

The next step creates a usable system based on the provably optimal overlays. We introduce NOAH which trims the use cases down to a minimum to create heuristics for building very good aggregation overlays. NOAH also incorporates several practical mechanisms to maintain the base assumptions as much as possible and maintain overlays efficiently after the initial aggregation is completed.

Finally we build a full top-k matching system with NOAH. The aggregation happens after initial computation, we note that the two phases can be independently optimized. We expand the top-k matching model over the prior art to allow better use of the type of data that may be available in current applications and introduce

the FX-TM algorithm to efficiently match data in this expanded model. FX-TM is run within NOAH to create a full distributed top-k matching system.

1.5 Problem Statement

There are classes of distributed data analysis models which are underdefined and thus poorly addressed for optimized running by current solutions. Top-k matching algorithms need to be more expressive to deal with new data models. This includes obtaining all data available for more targetted matching than limit matching to attributes which are universally available and allowing entities to weight attributes positively and negatively.

1.6 Thesis Statement

This dissertation presents a mathematical model for the compute-aggregate class of distributed data problems and finds provably optimal aggregation overlays. The optima are distilled into heuristics for a practical general purpose system to create on-demand aggregation overlays. A distributed top-k matching system is created with this system and a novel, more expressive, top-k matching approach.

1.7 Organization

In Chapter 2 we precisely define the compute-aggregate family of problems and rigorously prove optimal aggregation overlays. In Chapter 3 we distill heuristics from the optimal overlays and use them to create a system called NOAH for general purpose aggregation in a larger big data analysis framework. In Chapter 4 we create an algorithm for use with top-k matching and integrate it into NOAH, resulting in a distributed top-k matching system. Chapter 5 discusses the related work, and Chapter 6 concludes the dissertation.

2 THEORETICALLY OPTIMAL AGGREGATION OVERLAYS

This chapter lays out the theoretical model for our aggregation work and proves the optimalilty of a targetted overlay approach. With the exception of portions of Section 2.3.2 the contents of this chapter chapter are William Culhane, Kirill Kogan, Patrick Eugster, and Chamikara Jayalath, *Optimal Communication Structures for Big Data Aggregation*, INFOCOM, April 2015 [23].

2.1 Overview

Big data processing underpins many efforts of modern computing usage. Given the size of data and the amount of resources that go into processing them, there is a decided motivation to optimize the processing as much as possible.

We consider *compute-aggregate* problems, which can be broken into an initial computation phase on data distributed across a set of nodes and an aggregation phase to collect and process the results of the computation phase. An example is word count, where counts from all nodes can be aggregated by summing the counts for each word.

A natural choice is to aggregate results along a communication structure such as a tree connecting leaf nodes (where original computations occur) to a root (where the final result will be available). It becomes clear that there are simple customizations to such *aggregation trees* created for a broad range of aggregation functions. Exactly which customizations are applicable – most prominently affecting *fan-in* of the tree – depends on how the input data is to be processed: at once on an entire "block" or in a stream of chunks, and the characteristics of the aggregation function which affect the size of the data.

Intuitively, aggregation can be considered the "inverse" of *multicast*: in the latter, data is typically propagated along a tree, and a function applied at every node. The function may simply copy input to all outgoing links; in more complex cases nodes may run transformation functions on data prior to forwarding (e.g., for interoperability), or collect acknowledgements or negative acknowledgements (for reliability).

Much effort has been invested in optimizing multicast *spanning trees*, but aggregation is less studied. In practice aggregation is very common, especially with the advent of the big data era. While the familiar problems – e.g., merging sorted lists or combining word counts – have outputs at least as large as each input, the average MapReduce job at Google [6] and the common "aggregate" jobs at Facebook and Yahoo! [24] decrease the size of data. Despite the large potential performance impact of simple traits of aggregation trees, most systems described in literature performing aggregation are agnostic to them.

The contributions of this chapter are as follows:

- We identify and define a model for a class of problems termed *compute-aggregate* whose distributed execution is optimized by manipulation of an underlying aggregation tree without requiring revision to applications themselves.

- Within the compute-aggregate subset we identify two distinct input models requiring different optimizations – (a) aggregation of a single block and (b) streaming input.

- We identify parameters for aggregation trees with respect to both problem variants and prove their optimal values via mathematical models for various cases.

- We provide mechanisms to reduce the physical overhead or increase the bandwidth over a naïve implementation with streaming input. These mechanisms are particularly useful for the common real-world applications which decrease data size during aggregation.

- We evaluate the accuracy of our models with microbenchmarks performed in the Amazon Elastic Compute Cloud (EC2) and measure the actual impact that aggregation tree decisions make in practice, confirming our models.

2.2 Model Intuition and Definitions

In this section we explain the compute-aggregate model and what it means to have blocks versus streaming input.

2.2.1 Compute-Aggregate

Compute-Aggregate is a straightforward family of problems where computation is applied to multiple subsets of an input dataset, presumably in parallel, and the outputs from those computations are aggregated to create the final output. This is formally defined in Definition 2.2.1 and illustrated in Figure 2.1.

Definition 2.2.1. *Compute-aggregate runs on decomposable input* $\overline{z} = z_1, \ldots, z_n$ *to produce desired output* $h(\overline{z})$ *when* $h(\overline{z})$ *can be decomposed into a computation on subsets of the data,* $c(z_1), \ldots, c(z_n)$*, and function* $a()$ *can operate on those results such that* $h(\overline{z}) \equiv a(c(z_1), \ldots, c(z_n))$.

Each computation node contains some subset of the intial data. After computation (c), a system aggregates (a) the results along an *aggregation tree* (or henceforth simply tree) communication structure to create the final output. With the exception of passing the results of the computation to the aggregation tree the two phases are independent from each other. At each leaf, data z is in memory prior to computation, and computation applies function c such that $x = c(z)$, where x is data formatted for aggregation. There is no requirement on the type or complexity of the computation. It can be as simple as reading data from a source, such as a log file.

We consider optimizating the aggregation phase. Optimizing computation requires knowledge about the data, data structures, and algorithms for each specific problem.

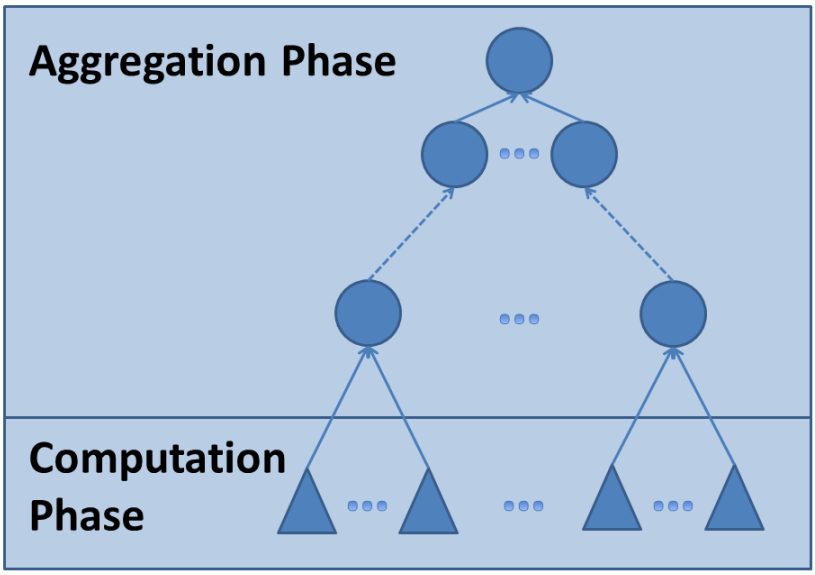


Figure 2.1.: Visual representation of the computation and aggregation phases.

We show optimizing the tree often only requires knowing very basic information about the aggregation function.

Often the initial computation reduces the size of the data for aggregation significantly because the analysis is interested in some feature of the data rather than the data itself. A user, for instance, might want to find the entries in a log pertaining to a particular bug, retrieve the number of references to a set of websites, or reduce the data to a finite number of clusters for a machine learning algorithm. The application must analyze all data, but the output from each slice of data is smaller than the data itself.

Aggregation can be triggered by the completion of the computation phase or run periodically on the current state of the data, as long as the data is formatted for aggregation. Aggregation applies some function a to all of the outputs of the computation nodes, $a\left(c\left(z_1\right), \ldots, c\left(z_n\right)\right)$. This does not have to be done in a single step. Aggregation can be applied to the results of previous aggregation. When aggregation begins, each output from a leaf is sent to a single aggregation node. The aggregation is

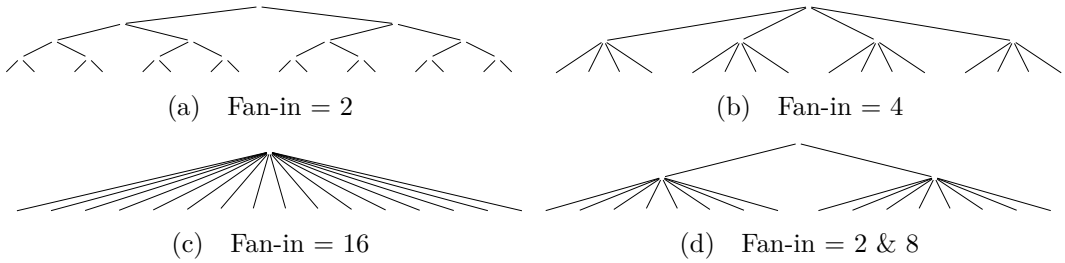


Figure 2.2.: Four balanced aggregation trees with 16 leaves.

applied to all inputs received at the node, and the node outputs the aggregated result. The outputs from those nodes, if there are indeed multiple such nodes, are in turn aggregated. The final aggregate result contains exactly one path to each leaf, so each computation output is included exactly once, resulting in an explicit tree structure. Figure 2.2 shows how 16 leaf nodes can be placed in two different trees with fan-ins of 2 and 16. These are the extreme scenarios, and it is possible to use any fan-in between the two values, although such fan-ins will not result in full trees unless the number of leaf nodes is a power of the fan-in.

Later in Section 2.3.2 we discuss the idea of "implicit" aggregation. If portions of the data do not interact with each other, like subsequent chunks n-gram counting with no overlapping n-grams, but have an implicit order based on their locations in the input streams the tree may aggregate them separately without explicitly requiring them to be aggregated through the same channels as long as the ordering is maintained.

2.2.2 Single Block

The first case we consider is input processed in a single block, and each aggregation results in a single block of its own. This means all of the data is available and computed at a single step at the leaves, then the aggregation tree reads the entire results and outputs a single output without natural breaks which can be used to distribute the work.

In this case we are concerned with the optimal latency of processing the block. The number of inputs – including the direct outputs from the computation phase and aggregations of them – that can be aggregated at each node in the tree is variable, as long as each result from the computation phase is included exactly once. Reducing the fan-in means increasing the number of siblings in the tree, which in turn means that more aggregation is done in parallel at each level. However, subsequent levels have to repeat some of the work their children did. Therefore the optimal fan-in depends on the ratio of work that is repeated to the amount of parallelism gained.

2.2.3 Streaming

While the trees we will propose for single blocks of input are provably good, they only use one level of the aggregation tree at a time. Thus we consider a second case of input using multiple levels at a time. In order to do this we split the input into chunks according to rules laid out later in this section. By splitting the input we allow aggregation on some segment of the input data to be propagated up the tree while aggregating the next chunk at the lower levels. Thus we utilize more nodes in parallel. While this approach may recompute portions of the aggregation in a way that was avoided in the single block case, the additional parallelism can overcome that.

Rather than consider optimal latency per chunk we note that for any sufficiently large finitely sized input the optimal latency will be on a tree which has the highest bandwidth of processing the chunks. Of course, this extends to infinitely sized input as well, i.e. continuous input, and incrementally updated data where the results of the aggregation are updated, or even fed back into the original data for reaggregation.

We note that if a single block of input is sufficiently large it may be chunked and turned into a streaming problem if the data is amenable to such an operation, which means the input follows rules we define later in this section. In practice the input must be sufficiently large to overcome the warmup and cooldown times of the tree,

and for the increased parallelism used to increase the bandwidth of the streaming situation to overcome the reduction in latency with respect to one block.

2.2.4 In-Memory Data

Like the recent big data processing methods in our prior art search, we use in-memory computation (and aggregation). Because we are worried about high performance, disk accesses are a significant penalty on today's hardware. In addition, mixing RAM and disk swapping leads to unpredictable latencies which can differ by orders of magnitude.

RAM constraints may be one thing that forces the use of the streaming model in practice. If the input size is too large to fit on a single machine, it will require chunking. Even if there are not enough chunks to overcome the warmup and cooldown phases, this may be preferable to requiring disk access.

2.2.5 Notation and Assumptions

We rely on several assumptions to calculate optimality. These do *not* affect the correctness of the output. Table 2.1 shows the notation used to rigorously define our assumptions and prove optimal fan-ins for the trees.

We assume time complexities of aggregation functions depend solely on input size. Some aggregation algorithms have time complexities expressed in terms of the fan-in and the size of individual inputs, e.g. $O(fan\text{-}in \times \log(average\ input\ size))$. Our experiments show this deviation is a small factor.

We model communication time as linear on the amount of data transferred. Contemporary streams to a node share bandwidth, so the communication time at a node is the same as long as all children start sending data before the node finishes receiving the data from any set of other nodes. Communication at a node can be affected by other nodes on the network, especially when TCP incast is present [25]. We assume that TCP incast is resolved on the TCP level as in [26] or communication

Table 2.1.: Aggregation model notation.

Token	Meaning
n	Number of computation/leaf nodes.
f	Fan-in of the tree, making the height $\log_f n$.
$a(\overline{x})$	The aggregation function for a set of inputs x.
$a^t(\overline{x})$	Returns the time taken for $a(\overline{x})$ (with communication).
$c()$	Initial computation function.
$h()$	(Composed) function transforms original input to final output.
t	Time per unit of data for linear $a^t(\overline{x})$ and $a^t(\emptyset) = 0$.
z	Data prior to computation.
x	Data formatted for aggregation.
x_0	Output from a computation node. This is a special case of z.
y	Ratio of output sizes of consecutive levels.
y_0	Ratio of the final aggregate output size to $\|x_0\|$.
$r(a,b)$	$TRUE$ iff item b follows item a with ordering function $r()$.
m	A single chunk in a streamed input.

time is relatively inconsequential. We wait for all streams when aggregating at a node to avoid higher programming complexity to account for locking and race conditions. These assumptions require, performance-wise, a pseudo-synchrony which requires a degree of homogeneity among nodes of a level. Heterogeneity may exist across levels.

We do ignore some of the other complexities that can be associated with communication, especially with regard to node location on the network and shared resources or conflicts. We are considering mostly the case of using third party cloud offerings, which often hide such things from the user. As such we consider the network architecture a black box best approximated in the average case as mostly uniform.

Homogeneous hardware is a fair premise when datacenters mass order standard hardware, and cloud services provide tiers of service based on performance. The minor service variations can be unpredictable. Homogeneous input follows from our model

for data distribution. If aggregation time depends on the data traits other than size (e.g., element order), those traits must be distributed. We find the leeway in synchrony and the impact of tree customization mask enough hetergeneity in practice.

Aggregation costs are modeled as monotonically increasing on input size, and are zero for no input. Modeling non-linear setup overheads complicates analysis, and the practical impact of these overheads are very small. When aggregation changes the size of the data we model ratio of output to input at each level to be the same. This is or can be made to be true for many applications. In many other cases the ratio stays within a range, e.g. below 1, and the optimal fan-in is unaffected.

We assume the same ideal fan-in is used at all levels of the tree as possible. Without a rigorous proof of this, we note that once y is applied to the first level the optimal remaining fan-in is essentially being applied to $\frac{1}{f}$ nodes with input different by a factor of y, which is essentially the same problem.

We model full and balanced trees. Fulfilling fullness for arbitrary numbers of leaves can be difficult, as it requires the fan-ins at each level to multiply to the number of leaves. It is simple to balance within a level, however, and fullness and balance follow when the number of leaves is a power of fan-in.

2.2.6 Function Requirements

We consider aggregation functions which take $x_1 \dots x_i$ and output an aggregate $x^{1..i}$, i.e., $x^{1..i} = a(\overline{x})$. Functions must be able to handle any number of inputs in order for the increased fan-in to be effective, as there is no advantage to having multiple inputs available it they cannot be used.

Functions are cumulative, commutative, and associative. This essentially means inputs may be aggregated in any order with any group of inputs, including those which are outputs of non-root nodes of the tree. Definitions 2.2.2, 2.2.3, and 2.2.4 capture the properties more precisely. Definitions require equivalency ($\equiv$), not necessarily identical output. For example, if a system is supposed to output the single word with

the maximum number of occurrences (word count) and two words are tied for that distinction, either word may be returned.

Definition 2.2.2 (Cumulative Aggregation)**.** $a\left(a\left(\overline{x}\right), a\left(\overline{x}'\right)\right) \equiv a\left(\overline{x}, \overline{x}'\right)$

Definition 2.2.3 (Commutative Aggregation)**.** $a\left(\overline{x}', \overline{x}\right) \equiv a\left(\overline{x}, \overline{x}'\right)$

Definition 2.2.4 (Associative Aggregation)**.** $a\left(a\left(\overline{x}, \overline{x}'\right), \overline{x}''\right) \equiv a\left(\overline{x}, a\left(\overline{x}', \overline{x}''\right)\right) \equiv a\left(\overline{x}, \overline{x}', \overline{x}''\right)$

Because our streaming model outputs results before receiving the entirety of the input, there are additional rules on the inputs which must be satisfied. This applies both to the original input and the output of nodes which is used as input higher in the tree. The inputs must all use a known ordering function, and side effects can only propagate from earlier chunks of data to later chunks rather than the opposite way around, as one would expect. Definitions 2.2.5, 2.2.6, and 2.2.7 define these in more rigorous language. We also provide counterexamples to each rule using a word count application.

Definition 2.2.5 (Known Ordering)**.** $\exists$ *some known transitive function* r *on pairs of keys s.t. for any pair* $k_1 \neq k_2$ *of keys* $r\left(k_1, k_2\right)$ *returns* $TRUE$ *iff* $k_1 \leq k_2$*.*

Counterexample If the input is a hashmap with string:key mappings, the ordering of a traversal of the list of entries is not a lexical ordering of the keys, but a numerical ordering of the hashes of the keys. The programmer must understand the datastructure and account for this.

Definition 2.2.6 (Consistent Ordering)**.** $\forall$ *keys* k_1, k_2 *such that* k_1 *precedes* k_2 *in ordered list* c_i *used as input for aggregation function* $a\left(x_0, \ldots, x_d\right)$*, if* $k_1, k_2 \subset x_j \implies r_j\left(k_1, k_2\right) = TRUE$*.*

Counterexample One list could be derived from a hashmap, ordered on the hashes of the keys. Another list may have been processed to follow lexical ordering of the keys or be sorted on the number of occurrences of each string.

Definition 2.2.7 (Unidirectional Side Effects)**.** *In aggregation function* $a\left(x_0, \ldots, x_d\right)$*, list* x_i *can be decomposed into concatenated lists* $x_i^0 \cdot x_i^1 \cdot x_i^2$ *and* x_j *into* $x_j^0 \cdot x_j^1 \cdot x_i^2$ *for* $i \neq j$ *such that for any element* $l^0 \subset x^0$ *and* $l^1 \subset x^1$*,* $r(l^0, l^1)$*.*

If aggregation function $a\left(\right)$ *is applied to chunks* x_i^0 *and* x_j^0 *to create output* x_a^0*,* $\exists$ *items* $k_1^0, k_2^0 \subset x_a^0$ *such that* $r\left(k_1^0, k_2^0\right) = TRUE$*, then* $a\left(x_i^{0\ldots2}, \ldots, x_j^{0\ldots2}\right)$ *contains* $k_1^0, k_2^0 \subset x_a^{1\ldots2}$ *such that* $r\left(k_1^0, k_2^0\right) = TRUE$*.*

Counterexample If the output is ordered on the number of occurrences of each string, but string s occurs in x_i^1 and x_j^2, the value used in the ordering of s, and thus the ordering of s relative to other elements, can change when x_j^2 is aggregated with the output from the original aggregation.

2.3 Optimality Proofs and Mechanisms

2.3.1 Single Block Optima

Here we prove theoretical (near-)optimality of the fan-in f for minimizing latency given n and $a^t\left(\overline{x}\right)$ for several cases of y_0 when there is a single block of input at each tree leaf.

Lemma 2.3.1. *The total aggregation time with linear* $a^t\left(\overline{x}\right)$ *and* $y_0 \neq 1$ *is* $g\left(f, n, y_0\right) = \frac{t|x_0|f(y_0-1)}{\log_f \sqrt[n]{y_0}-1}$*, and* $\frac{\partial}{\partial f} g(f, n, y_0) = \frac{t|x_0|(y_0-1)y_0^{\log_n d}-\left(1+\frac{\log y}{\log n} y^{\log_n f}\right)}{\left(y_0^{\log_n f}-1\right)^2}$*.*

Proof. *For aggregation linear on input size, the time at a level is constant* $t \times$ *(the input size at the level). Initial input size is* $|x_0|$*, and size changes by a factor of* y *at each of the subsequent* $\log_f n$ *levels. Thus total aggregation time is* $\sum_{z=1}^{\log_f n} tfy^{z-1}x_0$*. Pulling out the constants gives* $tf\left|x_0\right| \sum_{z=1}^{\log_f n} y^{z-1} = \frac{tx_0fy^{\log_f n}-1}{y-1}$*. Recalling* $y = \sqrt[\log_f n]{y_0}$ *gives* $\frac{t|x_0|f(y_0-1)}{\log_f \sqrt[n]{y_0}-1}$*.*

Lemma 2.3.2. *The total aggregation time with $y_0 = 1$ is $a^t(f|x_0|)\log_f n$.*

Proof. *Reusing the logic from Lemma 2.3.1 without introducing y_0 gives us $\sum_{z=1}^{\log_f n} a^t(f|x_0|)$. This simplifies to $a^t(f|x_0|)\sum_{z=1}^{\log_f n} 1$, and then to $a^t(f|x_0|)\log_f n$.*

Theorem 2.3.3. *The optimal fan-in is 2 when $y_0 < 1$ and $a^t(\overline{x})$ is linear or superlinear.*

Proof. *By Lemma 2.3.1, the total aggregation time taken is $g(f,n,y_0) = \frac{tf|x_0|(y_0-1)}{\log_f \sqrt[n]{y_0}-1}$, and $\frac{\partial}{\partial f}g(f,n,y_0)$ is $t|x_0|(y_0-1)\frac{y_0^{\log_n f} - \left(1+y_0^{\log_n f}\log_n y_0\right)}{\left(y_0^{\log_n f}-1\right)^2}$. $0 < \log_n f \leq 1$ for $2 \leq f \leq n$, so $\frac{\partial}{\partial f}g(f,n,y_0) > 0$ for $0 < y_0 < 1$.*

$\therefore$ The optimal f is 2.

We assumed $a^t(\overline{x}_0) + \ldots + a^t(\overline{x}_z) = a^t(\overline{x}_0 + \ldots + \overline{x}_z)$. As f grows there are more inputs and input size is reduced less, and $a^t(\overline{x}_0) + \ldots + a^t(\overline{x}_z) < a^t(\overline{x}_0 + \ldots + \overline{x}_z)$ for superlinear $a^t(\overline{x})$, so superlinear $a^t(\overline{x})$ is more sensitive to f than linear $a^t(\overline{x})$.

$\therefore$ This result holds for superlinear $a^t(\overline{x})$.

Theorem 2.3.4. *The optimal fan-in is e when $y_0 = 1$ and $a^t(\overline{x})$ is linear.*

Proof. *With $a^t(f|x_0|)\log_f n$ from Lemma 2.3.2 and linear $a^t(x)$, $g(f,n,y_0) = f|x_0|t\log_f n$. $\frac{\partial}{\partial f}g(f,n,y_0) = \frac{|x_0|t(\log f - 1)\log n}{\log^2 f}$, which is 0 iff $f = e$. $\frac{\partial^2}{\partial f^2}g(f,n,y_0) = -\frac{|x_0|t(\log f - 2)\log n}{f\log^3 f}$. At $f = e$, $\frac{\partial^2}{\partial f^2}g(f,n,y_0) > 0$, so this is a minimum.*

$\therefore$ The optimal f is e.

Theorem 2.3.5. *The optimal fan-in is $[2,e)$ when $y_0 = 1$ and $a^t(\overline{x})$ is superlinear.*

Proof. *By Lemma 2.3.2 the total aggregation time is $g_{lin.}(f,n,y_0) = a^t(f|x_0|)\log_f n$. As shown in Theorem 2.3.4, $lim_{a^t(\overline{x})\to linear}$ is e. We can assume $\frac{\partial^2}{\partial f^2}g_{superlin.}(f,n,y_0) > \frac{\partial^2}{\partial f^2}g_{linear}(f,n,y_0)$. Thus $\frac{\partial^2}{\partial f^2}g_{linear}(f,n,y_0) > 0$ for $f \geq e \implies \frac{\partial^2}{\partial f^2}g_{superlin.}(f,n,y_0) > 0$. Thus any minimum occurs at $f < e$.*

$\therefore$ The optimal value of f is in the range $[2,e)$.

Theorem 2.3.6. *The optimal fan-in is $(1-\log_n y_0)^{-\log_{y_0} n}$ when $1 < y_0 < n$ and $a^t(\overline{x})$ is linear.*

Proof. *From Lemma 2.3.1, the amount of time taken to aggregate is* $g(f,n,y_0) = \frac{tf|x_0|(y_0-1)}{\log_f \sqrt[n]{y_0}-1}$, *and* $\frac{\partial^2}{\partial f^2} g\left(f,n,y_0\right) = \frac{t|x_0|(y_0-1)}{\left(y_0^{\log_n f}-1\right)^2}\left(y_0^{\log_n f} - \left(1+y_0^{\log_n f}\log_n y_0\right)\right)$. *For* $y_0 > 1$, $\frac{t|x_0|(y_0-1)}{\left(y_0^{\log_n f}-1\right)^2} > 0$, *so the expression is 0 iff* $y_0^{\log_n f} = \left(1+y_0^{\log_n f}\log_n y_0\right)$, *which happens at* $f = \left(1-\log_n y_0\right)^{-\log_{y_0} n}$.

$\frac{\partial^2}{\partial f^2} g\left(f,n,y_0\right) = \frac{t|x_0|(y_0-1)y_0^{\log_n f}\log y_0\left(\log n+\log y_0-(\log n-\log y_0)y_0^{\log_n f}\right)}{f\log^2 n\left(y_0^{\log_n f}-1\right)^3}$. *Because* $\frac{t|x_0|(y_0-1)y_0^{\log_n f}\log y_0}{f\log^2 n\left(y_0^{\log_n f}-1\right)^3} > 0$, $\frac{\partial^2}{\partial f^2} g\left(f,n,y_0\right) > 0$ *iff* $\log n+\log y_0-(\log n-\log y_0)\, y_0^{\log_n f} > 0$. *Substituting the extrema value for* f *gives* $\frac{y_0^{-\log_{y_0}(1-\log_n y_0)}-1}{y_0^{-\log_{y_0}(1-\log_n y_0)}+1} - \log_n y_0 < 0$. *To prove this we fix* n *and find* y_0 *to maximize* $h\left(f,y_0\right) = \frac{y_0^{-\log_{y_0}(1-\log_n y_0)}-1}{y_0^{-\log_{y_0}(1-\log_n y_0)}+1} - \log_n y_0$. $\frac{\partial}{\partial y_0} h\left(f,y_0\right) = \frac{2\log^2 n+\log^2 y_0-4\log n\log y_0}{y_0\log n(\log y_0-2\log n)^2}$.

$\frac{\partial}{\partial y_0} h\left(f,y_0\right) > 0$ *for* $1 < y_0 < n$. *Thus* $max\left(h\left(n,y_0\right)\right)$ *occurs at* $\lim_{y_0\to n}$, *and* $\lim_{y_0\to n} h\left(n,y_0\right) < 0$. *The extrema of* $g(f,n,y_0)$ *is a minimum.*

$\left(1-\log_n y_0\right)^{-\log_{y_0} n} > n \implies d > n$, *which is impossible. Since the only local extrema is a minimum,* $\frac{\partial}{\partial f} g\left(f,n,y_0\right) < 0$ *for* $f = \left[2, \left(1-\log_n y_0\right)^{-\log_{y_0} n}\right]$, *so the optimal fan-in is the largest possible value, i.e.* n, *in this case.*

$\therefore$ *The optimal* f *is* $min(n, \left(1-\log_n y_0\right)^{-\log_{y_0} n})$.

Theorem 2.3.7. *The optimal fan-in is* n *when* $y_0 \geq n$.

Proof. *The time taken by the root node is* $a^t\left(y^{\log_f n}|x_0|\right)$. $y_0 \geq n \implies y \geq f$ *and* $\log_f n \geq 1$, *so this is minimal at* $\log_f n = 1$. *In addition, for* $f < n$ *the rest of the tree takes non-zero time.*

$\therefore$ *The optimal* f *is* n.

Table 2.2 summarizes the results from our proofs for single blocks of input. There are still unproven cells where the degree of sub- or superlinearity of the aggregation is required to find the optimal value. There is always an aspect of linearity to $a^t\left(\overline{x}\right)$ due to communication time, so it makes sense to use the results from the linear cases on the sublinear cases, which would fill most of the table.

Table 2.2.: The optimal value for f to minimize the latency of a single input block and the applicable proof when available.

$\mathbf{y_0}$	**Model Runtime**	**Optimal Fan-in**	**Sublin. $\mathbf{a^t}$ ()**	**Linear $\mathbf{a^t}$ ()**	**Superlinear $\mathbf{a^t}$ ()**
$y_0 < 1$	$\frac{t\lvert x_0\rvert f(y_0-1)}{\log_f \sqrt[n]{y_0}-1}$	2	*unproven*	Thm. 2.3.3	Thm. 2.3.3
$y_0 = 1$	$tf\,\lvert x_0\rvert \log_f n$	e	*unproven*	Thm. 2.3.4	Thm. 2.3.5 **near optimal*
$1 < y_0 < n$	$\frac{t\lvert x_0\rvert f(y_0-1)}{\log_f \sqrt[n]{y_0}-1}$	$min(n, (1-\log_n y_0)^{-\log_{y_0} n})$	*unproven*	Thm. 2.3.6	*unproven*
$y_0 \geq n$	$\frac{t\lvert x_0\rvert f(y_0-1)}{\log_f \sqrt[n]{y_0}-1}$	n	Thm. 2.3.7	Thm. 2.3.7	Thm. 2.3.7

2.3.2 Streaming Optima

The case for streaming input is different than a single block. In this case we are worried about the bandwidth instead of the latency. As such, we explore the tree which can process subsequent input chunks the fastest. Intuitively this means finishing the initial processing of the current chunk the fastest so the tree can start processing the next chunk.

To achieve this we make an additional assumption about the tree. The first is that we can predict a single layer which is the bottleneck for the system. The rest of the tree can process its input faster than the limiting level, which we refer to as "keeping pace". Assuming keeping pace simplifies finding the optima because we just have to minimize the latency of the bottleneck level.

Trees for problems with $y_0 \leq 1$, which, as mentioned in the introduction, account for most of the aggregation seen in real world usage, intuitively have a bottleneck at the first level of aggregation. When $y_0 < 1$ they reduce the size of the data and thus the amount of time required at levels above. When $y_0 = 1$ the amount of data at each level is the same, so aggregation time should be roughly the same.

Theorem 2.3.8. *For $y_0 \leq 1$ The maximal bandwidth for streaming input when the entire tree keeps pace with the first level of aggregation happens when the $f = 2$ at the first level.*

Proof. *Bandwidth is throughput per time unit. Assuming the rest of the aggregation tree keeps pace with the lowest level, we only have to consider the bandwidth of the lowest level. The throughput is equal to the amount of input, which is $|m|$ at each of n nodes. The latency of the first level is $a^t(\overline{x})$ where $|\overline{x}|$ is $f \times m$. This makes the bandwidth $\frac{n|m|}{a^t(\overline{x})}$. By definition $a^t(\overline{x})$ monotonically increases with input size, so for a given m the the denominator is minimized by minimizing f.*
$\therefore$ The optimal f at the lowest level of aggregation is 2.

It is interesting to know that while this result is independent of the degree of function $a^t(\overline{x})$, there is an additional implied optimization that is possible in the case the function is not linear on $|m|$. Because the numerator is linear on $|m|$, sublinear occurrences in the denominator mean that $|m|$ should be maximized. Similarly $|m|$ should be minimized when the aggregation cost function is superlinear on $|m|$. In the case the function is linear, $|m|$ does not affect optimality.

As long as $y_0 \leq 1$ an entire tree with $f = 2$ will keep pace with the first level: simply create the rest of the tree with $f = 2$ as well. Since the output of each level is not growing, each level has no more aggregation work to do than the level below it. Thus no level is a bottleneck compared to its level below.

If $y_0 > 1$ this does not work. In fact the top level of aggregation is the bottleneck, as the amount of work increases at each level and culminates at the final level. Theorems 2.3.9 and 2.3.10 cover these cases. Table 2.3 summarizes the results from all our proofs for streaming input.

Theorem 2.3.9. *For $1 < y_0 < n$ and linear or superlinear $a^t(\overline{x})$ the maximal bandwidth for streaming input happens when the $f = 2$.*

Proof. *Bandwidth is throughput per time unit. Assuming the output grows at subsequent levels, the final level of aggregation is the bottleneck for the bandwidth. The*

throughput is equal to the amount of original input for the chunk, $n \times |m|$, divided by the time to process the input at the final level, $a^t(f\,|m|\,y^{\#of\ previous\ levels})$. For linear $a^t(\overline{x})$ and substituting $y = \sqrt[\log_f n]{y_0}$ this becomes $B(y_0, n, f) = \frac{n|m|}{\frac{(\log_f n)-1}{y_0^{\log_f n}} f|m|t}$ for some constant t, and $|m|$ factors out.

$\frac{\partial}{\partial f} B(y_0, n, f) = \frac{y_0^{\log_n f-1} n}{tf^2}(\frac{\log y_0}{\log n} - 1)$. For $y_0 < n$ this is always negative, indicating that bandwidth decreases as f increases for this range, making the minimal fan-in the maximal bandwidth.

$\therefore$ the optimal f is 2.

This analysis finds that the increased parallelism dominates the increased time at the final level. Therefore when the increase in time is more pronounced with respect to f in the case of a superlinear $a^t(\overline{x})$ the increase in parallelism continues to dominate.

$\therefore$ this result holds for superlinear $a^t(\overline{x})$.

Theorem 2.3.10. *For $y_0 > n$ and linear or sublinear $a^t(\overline{x})$ the maximal bandwidth for streaming input happens when the $f = n$.*

Proof. *Bandwidth is throughput per time unit. Assuming the output grows at subsequent levels, the final level of aggregation is the bottleneck for the bandwidth. The throughput is equal to the amount of original input for the chunk, $n \times |m|$, divided by the time to process the input at the final level, $a^t(f\,|m|\,y^{\#of\ previous\ levels})$. For linear $a^t(\overline{x})$ and substituting $y = \sqrt[\log_f n]{y_0}$ this becomes $B(y_0, n, f) = \frac{n|m|}{\frac{(\log_f n)-1}{y_0^{\log_f n}} f|m|t}$ for some constant t, and $|m|$ factors out.*

$\frac{\partial}{\partial f} B(y_0, n, f) = \frac{y_0^{\log_n f-1} n}{tf^2}(\frac{\log y_0}{\log n} - 1)$. For $y_0 > n$ this is always negative, indicating that bandwidth increases as f increases for this range, making the maximal fan-in the maximal bandwidth.

$\therefore$ the optimal f is n.

This analysis finds that the increased time at the highest level dominates the increase parallelism. Therefore when the increase in time is less pronounced with respect to f in the case of a sublinear $a^t(\overline{x})$ the increase in parallelism is even less significant.

$\therefore$ this result holds for sublinear $a^t(\overline{x})$.

Table 2.3.: The optimal value for f to maximize the bandwidth of streaming input and the applicable proof when available.

y_0	Bottleneck Level	Model Bandwidth	Optimal Fan-in	Sublin. $a^t()$	Linear $a^t()$	Superlin. $a^t()$
$y_0 \leq 1$	Lowest	$\frac{n\|m\|}{t\bar{x}}$	2	Thm. 2.3.8	Thm. 2.3.8	Thm. 2.3.8
$1 < y_0 < n$	Highest	$\frac{n\|m\|}{y_0^{\frac{(\log_f n)-1}{\log_f n}} f\|m\|t}$	2	*unproven*	Thm. 2.3.9	Thm. 2.3.9
$y_0 \geq n$	Highest	$\frac{n\|m\|}{y_0^{\frac{(\log_f n)-1}{\log_f n}} f\|m\|t}$	n	Thm. 2.3.10	Thm. 2.3.10	*unproven*

In the case that the lowest level is the bandwidth bottleneck each level takes longer than the level below it, so input will buffer in ever increasing amounts between levels. Eventually it will be necessary for lower levels to wait to send the next because the levels to which they are sending will not be able to store it. We propose two mechanisms to better use the computational power of the tree.

Modifying Chunk Size

The first mechanism prevents buffering which could lead to buffer overloads when $y_0 > 1$ and ameliorates some warmup and cooldown inefficiencies: rather than fixing $|m|$, we change the size of each chunk. If each subsequent chunk is a factor of y larger each level takes as long to aggregate a chunk as the next level up takes to aggregate the previous chunk.

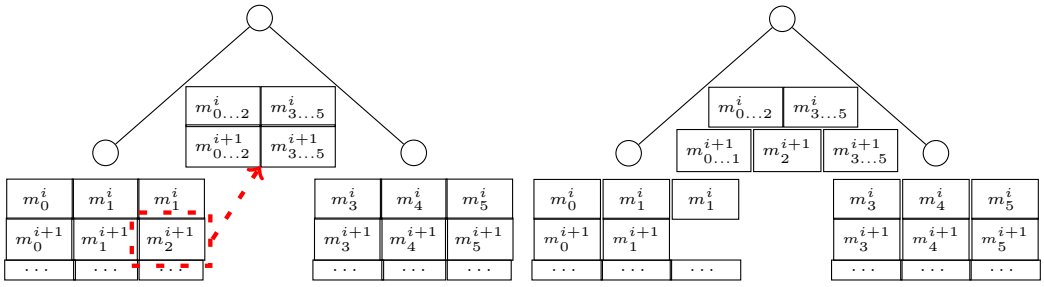
In practice it is impractical to increase the size of chunks indefinitely. Therefore we suggest specifying a minimum and maximum chunk size. If $y_0 > 1$ start with the minimum chunk size. Chunk sizes are increased until the maximum size is encountered, which will take $\log_y \frac{MAX(|m|)}{MIN(|m))}$ iterations.

The tree requires the occasional cooldown period clear to let the largest chunk propagate through the tree before the aggregation power is available to the next

chunk. There is also a warmup period when the first chunk, or the smallest chunk when the chunk size is reset, propagates through the tree. When the chunk size is reset these two periods can overlap, but the warmup phase cannot overtake the cooldown phase because the hardware is not available. It is not necessary to know the exact value of y *a priori* in this case. The first level of aggregation can use the size that it is outputting for a chunk divided by the original size of the input for that chunk.

Since the tree must still wait when resetting the size of the chunks this is not perfectly ideal. However, the waiting is predictable rather than occurring whenever the buffers happen to fill up. In addition, since it handles several chunks before bogging down it may be sufficient for bursty streams.

Modifying Fan-in



(a) Before moving buffers (b) After moving buffers

Figure 2.3.: Fractional fan-in through buffer transport.

When $y_0 < 1$, aggregation happens faster higher in the tree due to decreasing input size. Levels of the tree above the first are underutilized after finishing aggregation but before receiving the next chunk. The previous mechanism shifts this wait time to when chunk size resets, but does not alleviate it completely. Since we are using the proven optimal bandwidth in this case, there is no opportunity to improve bandwidth. Rather, this suggests that we can get comparable performance with less hardware.

To address this we consider how to implement higher fan-ins, even fractional fan-ins. This is not possible with single chunks because there is no way to break up a

chunk in the middle. However, with streaming input we already have natural breaking points. Figure 2.3 shows how we can use these to move aggregation to different points in the tree. In the figure the lower level aggregates 3 inputs, and the higher level aggregates the results from those two nodes. Assume the higher level is completing its work faster than the lower level. As a response it takes a chunk away from one of its children. As a result, in the $i + 1$ step the top node has more aggregation to slow it down, and the child only has two inputs for that chunk, speeding it up. This means that the higher level has a higher effective average fan-in than the original 2, and the child which transferred a chunk has a lower one that the original 3.

There are a couple of caveats to making this mechanism work well. Nodes should take chunks in equal amounts from each child. Using this to approach the optimal fan-in is not intuitive, especially as the mechanism propagates down the tree. To solve both these problems we suggest exercising transfers on demand. When a node detects it is running low on buffered chunks, it asks a child to transfer one. It asks the child which seems to be the furthest behind, since this will speed up the subtree of that child. This dynamically balances the tree.

Another limitation is that the requested chunk cannot be aggregated as it is received. Rather, the chunk must be buffered until the other chunks from the ith step are received. This means that the speed up or slow down at a node from sending or receiving a chunk will not be seen immediately.

Initial input chunks can be filtered immediately into the tree, and the full tree may be fully utilized as soon as the initial warmup phase is complete. This does open the possibility of fan-ins below 2 at the lowest level, which would theoretically decrease per-chunk latency and increase bandwidth. Perhaps a user could actually specify a number of available nodes and increase bandwidth by creating an effective fan-in below 2 at the lowest level, but we do not explore this.

A more practical use of this is choosing a size of tree which has different fan-ins at each level and balancing the effective fan-in at each level with this mechanism. We can thus create trees which have effective fan-ins of exactly the calculated optimum,

even if that optimum is fractional. This means having a complete and fully utilized tree, giving us the performance which can match the performance of a fan-in of 2 at the lowest level with the lowest number of nodes, thus decreasing the cost to maintain a system implementing this.

This approach allows the rest of the tree to keep pace with the lowest level even for $y_0 > 1$ if one additional property is met: aggregation must include no side effects, as precisely captured by Definition 2.3.1. This is reasonable for applications such as n-gram counting and merging time sorted lists where chunking is done on preselected time values.

Definition 2.3.1 (No Side Effects)**.** *In aggregation function $a\,(x_0, \ldots, x_f)$, list x_i can be decomposed into concatenated lists $x_i^0 \cdot x_i^1 \cdot c_i^2$ and x_j into $x_j^0 \cdot x_j^1 \cdot x_i^2$ for $i \neq j$ where the chunking happens on some delimiter s.*

If aggregation function $a\,()$ is applied to chunks x_i^0 and x_j^0 to create output x_a^0, then x_i^1 and x_j^1 to derive output x_a^1, and $a\,()$ is applied separately to only x_i^1 and x_j^1 to derive output $x_a'^1$, $x_a^1 \equiv x_a'^1$.

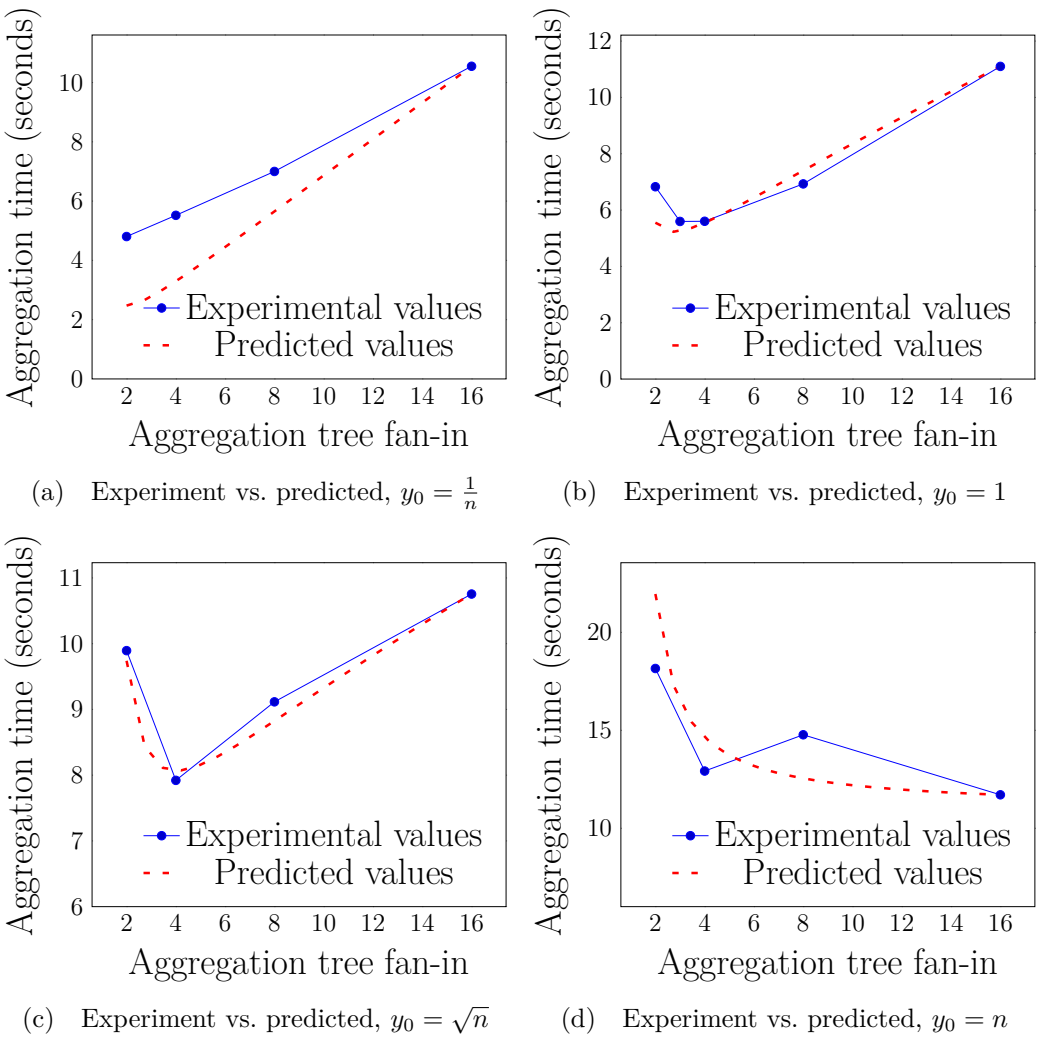
With this we can build side-by-side trees to handle different chunks. For example, one tree may take even numbered chunks, and another take the odd chunks, reducing the work of a tree by a factor of two. The order of the chunks imparts an implicit aggregation that requires no work to maintain explicitly.

2.4 Evaluation

To evaluate the model we run several microbenchmarks on a set of m1.medium nodes in an Amazon EC2 datacenter. We run a tightly controlled compute-aggregate solution compare the results against the modeled expectations. The compute phase consists of randomly generating a set size of integers. The aggregation at each node receives a set of lists from its children and performs some rather intensive computation which takes time which is almost perfectly linear on the total input size to produce

an output list of integers which is a factor of the given y different from the average size of its inputs.

2.4.1 Single Block



(a) Experiment vs. predicted, $y_0 = \frac{1}{n}$

(b) Experiment vs. predicted, $y_0 = 1$

(c) Experiment vs. predicted, $y_0 = \sqrt{n}$

(d) Experiment vs. predicted, $y_0 = n$

Figure 2.4.: Microbenchmark results.

Figure 2.4 shows the latency of running our microbenchmarks for a single block of input on 16 nodes with an initial input size of 100000 items. We vary the fan-in from the minimum value of 2 to the maximum value of n. We use a bottom-to-top

greedy algorithm to assign node placement. That is, the bottom most level is filled up as much as possible from left to right. When there are no more nodes to act as children at that level, we move to the next level up.

Each graph has a line of "Predicted values" which is a line generated by the appropriate mode with the constants set so it goes through the point when $f = 16$. In general, the trends predicted by the model are seen in the data. The minima occur close to the predicted places. We note that the actual optima do vary from 2 to n depending on y_0, so there is no constant optima. The difference in choosing the optimal fan-in and the worst case scenario is always at least 40%, and the predicted optima always the experimental optimum in these conditions. We use 3 to approximate e.

We only include the points in the graph which have full and balanced trees, as well as when $d = 3$ when the predicted optimum is e. We use a bottom up greedy approach to fill the tree. Nodes at one level are grouped together with one parent until no nodes at the lowest level remain. The process is then repeated with the parent nodes. This means that for a fan-in of 8 the lower level has the appropriate fan-in, but the top level only has a fan-in of 2. We note that this is still balanced. For other fan-ins this approach creates high levels of imbalance. Algorithms which strive for balance by starting top down appear to be possible, but they are more involved to implement than our greedy approach.

16 nodes is not a lot in cloud computing systems. Figure 2.5 extends the results for $y_0 = 1$ to 96 nodes, which is more realistic. In these results we see an even bigger penalty for choosing a non-optimal fan-in. The difference beween the optimal fan-in and a fan-in of 64 is 500%. So choosing an optimal fan-in is more important for larger. We omit the data point for $d = 96$ because at the input was not able to fit in memory at the root, and latency increased by almost an order of magnitude due to the time required for disk access. This just reinforces the idea of using a streaming model when possible with big data to avoid such problems.

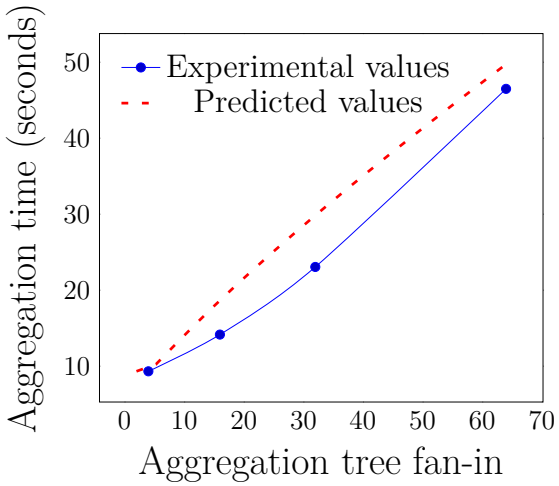


Figure 2.5.: Scalability test with $y_0 = 1$ and $n = 96$.

2.4.2 Streaming

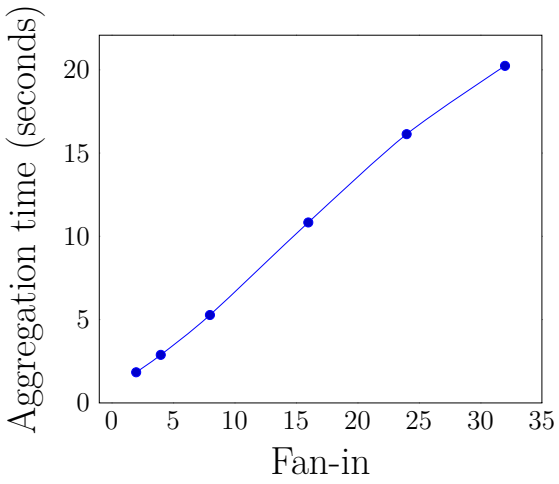
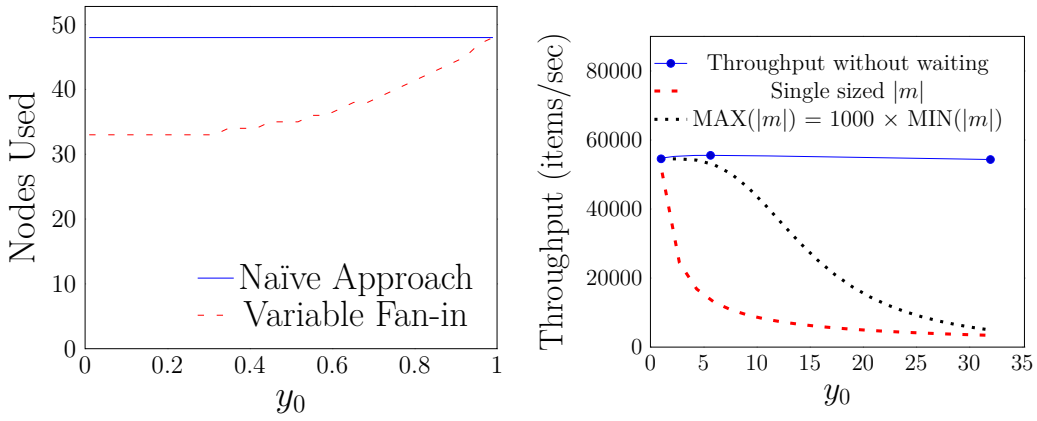


Figure 2.6.: Streaming time of lowest level vs. fan-in.

Figure 2.6 shows the latency of the lowest level of aggregation only for one chunk in streaming input when the lowest level is the bottleneck. We remind the reader that as the fan-in increases there is no change the size of the output per set of chunks read in, so this merely increases the time unit. Thus the increase in latency corresponds

directly to a decrease in bandwidth, and it is clear the optimal bandwidth occurs with minimal fan-in at the first level of aggregation.



(a) Nodes needed for varying fan-in vs. maintaining $f = 2$.

(b) Throughput penalty of resetting the size of m.

For the streaming experiments with $y_0 \leq 1$ and fan-in of 2 at the lowest level of aggregation, an entire tree with fanout 2 sufficient to keep up. Figure 2.7a shows the number of nodes needed in total for such a tree with 32 at the lowest level as the naïve approach. The second line in the graphs shows the number of nodes needed if we instead use the approach in Section 2.3.2 to create a variable fan-in tree.

Fractional fan-in above the first level is calculated based on y_0 with minimal overhead to implement the chunk transfer function. We then determine the minimal number of nodes to implement that fan-in. Since we must have at least enough capacity to handle the flows, rounding is done up.

Near $y_0 = 1$, there are no potential node savings. Removing a single node reduces the capacity of the tree to the point that the tree cannot keep up with its first level. As the output size of the first level decreases with respect to its input size the rest of the tree does not need as much computational ability to keep up to the extreme the point that it can use a single node. This reduces the number of nodes required by nearly a factor of two, although all the bottom level nodes are still required. This is beneficial for cost and maintenance savings.

With $y_0 > 1$, variable fan-in does not help unless Definition 2.3.1 is satisfied and the system can create parallel trees with implied ordering to handle different chunks of input. If that is not the case, we have to use decreasing chunk sizes with the occasional pause to allow the tree to cool down and warm up again as chunk size hits its limit and is reset.

Figure 2.7b shows the impact of this approach with 32 nodes at the lowest level of aggregation. The top line is based on experiments were the first level of aggregation continues uninterupted, and the results are thrown away – the theoretical bandwidth limit. The bottom line shows the opposite approach. At each level of the tree, the aggregation is paused until the next level up is ready to process the last chunk to be completed by the level. Because the output size grows at each level, every level is waiting for the level above it. The bandwidth essentially becomes constrained by the root node.

The middle line comes from varying $|m|$. Maximum and minimum chunk sizes are chosen with a ratio of 1000:1. As the bottom level outputs an aggregated chunk, it reads a new chunk the same size as its output, which is a factor of y times larger than the previous chunk. When the maximum chunk size is hit, the tree simultaneously cools down and warms up the new smallest chunk using the same pausing strategy as used in the case with fixed chunk size. Each level waits until the level above it is ready to continue before continuing itself.

Especially when y_0 is not very large, this approach is better than using a single size chunk. However, there is still a penalty suffered from not having a tree which is able to keep up with the lowest level. This may be enough to handle bursty traffic where the cooldown and warmup may initiated early done during lapses in input. Otherwise this suggests a user should make efforts to abide by Definition 2.3.1 and build parallel aggregation structures when possible.

3 A HEURISTICS BASED AGGREGATION OVERLAY SYSTEM

In this chaper we extend the theory contributions from the previous chapter into a practical system, NOAH. A preliminary version of this work was published in HotCloud [27].

3.1 Overview

The most intuitive and widely used approach to big data problems is splitting data across nodes and *computing* relevant information from subsets of the data before *aggregating* the partial results. In practice, almost every existing big data processing framework aggregates along a one-size-fits-all "overlay" instead of tailoring the overlay structure – typically a tree – to the individual semantics of aggregation functions in order to minimize execution latency.

Without the benefit of an analysis as to which form of tree provides less latency, many systems choose a one-size-fits-all approach which is inefficient for a problem it will encounter. For instance, an overlay with a fan-in of 2 (e.g., Presto [28]) is not ideal for sorting as we end up comparing same items at multiple levels. Conversely, a system with a fixed single coordinator performing all aggregation (e.g., a specialized top-k matching system [29]) has non-optimal latency for executing top-k matching since part of the aggregation job could have been parallelized.

Given that any one-size-fits-all aggregation overlay is sub-optimal for some set of applications, choosing an overlay per-application makes more sense if it can be done without resource-intensive analysis of an application. In the previous chapter, we proposed an analytically provable way to do this using a single value – the ratio of the size of an aggregated output to the size of one input – which is easily known for many functions, and varies drastically for different functions. For instance, in top-k

matching, the overlay filters multiple lists of k items into a list of the k top scoring items, so the output is the same size as one input – a ratio of 1. The ratio for merging sorted lists is higher because the output is the combination of all incoming lists. Table 3.1 shows some common aggregation problems organized by the relevant ratio.

In this paper we describe how we have put into practice our theoretical results [23] in building a system called NOAH capable of executing near-optimal aggregating overlays. In particular, we contribute the following:

- Distill the theoretical optima into simple heuristics.
- Introduce a mechanism to balance an overlay with a given fan-in to reduce the latency of the slowest branch; imbalance causes a non-ideal performance because branches with more nodes in an unbalanced overlay take more time than branches with less nodes since they have more input, causing a performance skew.
- Introduce novel mechanisms to (i) dynamically add new nodes to the overlay, and (ii) co-locate overlay nodes on the same virtual machine when the underlying dataset is dynamic, as in incremental computations. Both mechanisms take advantage of already executed aggregations on existing nodes to save time wasted on performing re-aggregations in all nodes. In addition, the last mechanism helps release unused or under-utilized machines. By co-locating nodes we can save more than 95% on maintenance costs while reaggregating in 80% less time when only a single leaf node is changed.
- Via microbenchmarks and real world compute-aggregate tasks in Amazon EC2 we show NOAH can improve latency by more than 75% over a one-size-fits-all aggregation tree, even without synthetically enforcing the assumptions of the model. We show the relevance of this improvement in iteratively computed problems and as part of a system involving more general workloads.

Like other recent work in big data processing [30] NOAH works in-memory. This reduces latency by a significant factor and reduces the unpredictable timing due to

Table 3.1.: Some common aggregation functions.

Common Problems	**Output to input size ratio**
Average MapReduce jobs at Google [6], average "aggregate" jobs at Facebook and Yahoo! [24], finding unique occurrences	< 1
Top-k match, word count with a fixed dictionary, multiplying square matrices, k-means clustering	$= 1$
Sort, concatenate, word count with mismatched dictionaries	> 1

random disk accesses. In fact, many "big data" sets are small enough to fit in memory without processing on a small number of machines, but are distributed to take advantage of greater parallelism [31]. NOAH is thus a parallel and distributed sub-engine that can be used within a big data analysis framework. We run microbenchmarks and real world tasks on our system in Amazon EC2 and show NOAH can improve latency by more than 75% over a one-size-fits-all aggregation tree. Our node co-location mechanism can improve that time by 80% when results are reagregated after changing data at a single leaf node.

3.1.1 System Comparison

In this section, we compare (Table 3.2) the implementation capabilities of NOAH to a representative sample of existing big data analysis systems.

MapReduce [6] is the de factor standard for much big data processing. The fan-in is configurable in the sense that the programmer can choose the number of reducers through choosing keys. This works very well for problems like word count where data can be partitioned independently. However, in the case of problems like top-k matching, where all results have to be directly compared to filter each other out, there are problems. The programmer must either use a single reducer (effective fan-in of

Table 3.2.: Aggregation system comparison.

System	**Fan-in**	**In-Mem**	**Aggregation Limitations**	**Data Format**
NOAH	Adaptive & Balanced	✓	Total Aggregation	Java Collections
LOOM [27]	Adaptive	✓	Total Aggregation	Java Collections
Presto [28]	2	✓	None	Distributed Arrays
Spark [32]	Configurable	✓	By Partition or Add-only semantics	Key-Value
MapReduce [6]	Configurable		By Partition	Key-Value

n) or run iterations of the problem to prune data at the cost of remapping at each iteration.

Spark [32], in addition to using an in-memory model for lower latency, adds an additional aggregator functionality to the MapReduce model. This allows aggregation information to be compiled across multiple reducers. However, these aggregators require "add-only" semantics. This means they can only be used to track variables in which all changes are additive. This is insufficient for problems like top-k matching which require filtering. As a result, Spark is also limited to an effective fan-in of n or the extra overhead of remapping intermediate results when implementing certain problems.

Presto [28] is a system based on the parallelization of R and array based analysis. A master node splits work across worker nodes and manages the workflow. In practice aggregation happens pairwise, creating a one-size-fits all aggregation overlay with fan-in 2.

Our initial efforts with LOOM and our refined work with NOAH consider the aggregation function and create a near optimal aggregation overlay. However, there is an inefficiency for problems like word count which can be intelligently partitioned.

These systems explicitly aggregate at each level regardless of how data is distributed, and are thus unable to take advantage of intelligent partitioning.

Ostensibly, these systems use different data types, but there is a degree of flexibility within them. For instance, a Java Collection may contain exactly one item of any type, and the keys of a key-value pair may be ignored when a single reducer is required to compare results from all mappers, such as with top-k matching.

Key-value mapping is a great advantage when data can be partitioned intelligently to avoid complete aggregation. It is worth noting that this may require extra effort to the programmer. For instance, if a user wants all items in sorted order by an integer value the high order bits may be used as a key, and the reducers may sort based on the low order bits. When the reducers produce their sorted output the results will already be in sorted order with other reducers. Without this insight (or in the case that order can only be determined by comparison rather than bits) all values must be sent to a single reducer.

3.2 Heuristics and Mechanisms

Table 3.3.: Relevant notation reprisal.

Token	**Meaning**
n	Number of computation/leaf nodes.
f	Fan-in of the overlay, making the height $\log_d n$.
x	Data formatted for aggregation.
y_0	Ratio of sizes for final output to one aggregation input.
$c()$	Initial computation function.
$a()$	Aggregation function.
$h()$	(Composed) function transforms original input to final output.

In this section, we first provide a definition of the two phase *compute-aggregate* data processing family of problems our system handles. We then explain heuristics and their intuitions of why and how overlay structures affect latency of such problems.

Finally, we introduce our novel mechanisms for balancing, and incremental updating of an overlay tree. Table 3.3 reprises the relevant notation.

3.2.1 Heuristics and Their Intuitions

The overlay aggregates the outputs from all the computation nodes, and the two phases can be optimized independently. The aggregation function is determined by the problem at hand, but the fan-in, f, can be chosen to minimize the aggregation time for a number of leaves, n.

The ratio of the size of the aggregated output to the size of one input is y_0. While the exact value may vary slightly based on the input, we have found that most individual aggregation functions are fairly predictable and stay within ranges that use the same heuristic.

The aggregation time at a single node depends on the size of all the inputs at that level. Each level of aggregation changes the size of the data processed at the next level while taking some time to run the transformation. At lower levels there is more parallelism because of the branching factor of the tree, but subsequent levels may have to redo some of the work. For instance, in merging sorted lists each element is considered at each level.

Intuitively, smaller fan-ins increase the work done in parallel because there are more active branches further down the tree. Unfortunately, the parallelism may not be productive if work is recomputed at multiple levels. For instance, when an aggregation function adds occurrences from words mapped to occurrences in a word count application, it considers each word once and adds the values from all maps. With a small fan-in the lowest levels combine more maps in parallel, but each level considers each word again. The value of y_0 determines if the time saved by the parallelism offsets the time required by extra levels.

While these values are not proven for all aggregation functions, particularly those whose run time are related to some combination of variables (e.g. f and $|x|$

Table 3.4.: The heuristics for f.

$\mathbf{y_0}$	**fan-in Heuristic**
$y_0 < 1$	2
$y_0 = 1$	3
$1 < y_0 < n$	$min(n, \lceil (1 - \log_n y_0)^{-\log_{y_0} n} \rceil)$
$y_0 \geq n$	n

independently) rather than directly to total input size, we empirically show the values are still practically reasonable. In doing so we also reduce the variables needed to only y_0, thus simplifying the input needed from the user.

We use the nearest integer to implement the fractional fan-ins suggested by the proofs. In particular, e rounds to 3, and values of $(1 - \log_n y_0)^{-\log_{y_0} n}$ can likewise require rounding. Our only remaining assumption is that y_0 is known. Other assumptions made by our model are either mitigated by system design or are shown to be experimentally negligible.

3.2.2 Balancing Mechanism

Our heuristics are based on a model where all overlay trees are perfectly balanced. This often requires fractional tree heights, which are of course not possible. Hence, in order to put our heuristics into practice, we first need to lift this assumption. We note there is no advantage to increasing fan-in, which increases the time at each node, unless a discrete change in height offsets it.

If, for instance, we have 25 leaf nodes and a fan-in of 6 there are 2 levels of aggregation. However, some nodes have their full complements of children while others do not. The branches with more nodes take longer than those with less because they have larger amounts of input. The slowest branch determines the latency of the system.

If we change the fan-in to 5 we have a perfectly balanced overlay *with the same height*. This reduces the performance skew between branches by balancing the work done, and the slowest branch is made faster by shifting work to other branches with available resources without increasing the length of the longest branch.

To implement this, we create a balancing mechanism. First we find the actual (non-fractional) height of a tree created by the overlay implementing the heuristic. For a heuristic i this is simply $\lceil \log_i n \rceil$.

We next look to see if another fan-in gives a more balanced overlay of the same height. We note that a smaller fan-in creates a more balanced overlays, so we inductively reason we should find the smallest fan-in which creates an overlay with the same height as the heuristic. To find this smallest value we take the log of the number of nodes divided by the height found earlier. This value must be rounded up because we do not allow fractional fan-ins, and any smaller fan-in results in an overlay of greater height.

Therefore the balancing mechanism uses the fan-in determined by the equation $\lceil \log n / \lceil \log_i n \rceil \rceil$ when i is the value returned by the original heuristic.

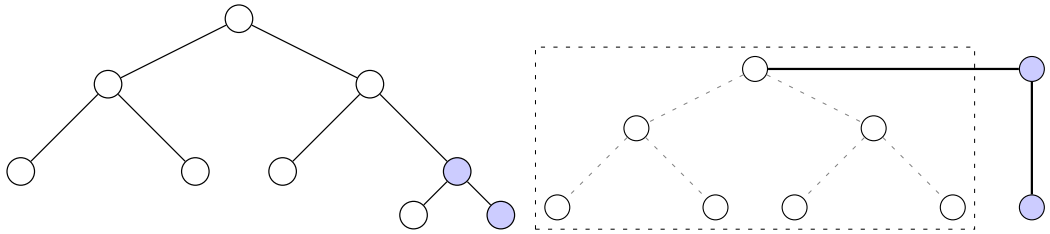
This only affects $1 \leq y_0 < n$; there is no fan-in less than 2, and any fan-in less than n results in greater tree height.

3.2.3 Incremental Update Mechanisms

In this section, we augment our heuristics, which are good for creating one-shot overlays to aggregate all data, with two practical mechanisms to efficiently maintain overlays beyond their initial aggregation phase.

Adding New Nodes

Many big data problems deal with datasets which are growing, and it is necessary to update the final output as new data enters the system. There are basically two options on how to handle this, as shown in Figure 3.1.



(a) Recompute the overlay (b) Use the prior in-memory result

Figure 3.1.: Maintaining results to avoid reaggregation.

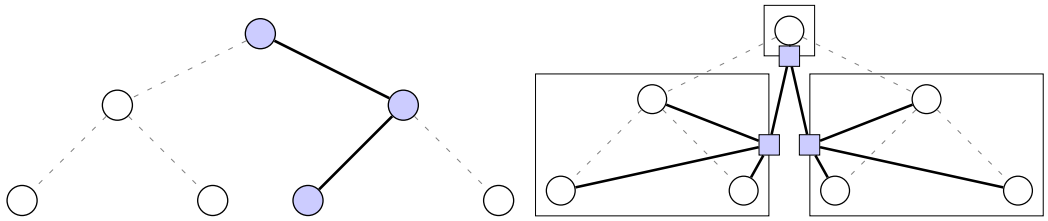
Simply reusing the heuristics, as shown in Figure 3.1a, creates a new overlay with the expanded set of leaf nodes and reaggregates the entire set of inputs. A large part of the new overlay redoes work done by the previous overlay to obtain results which are already available in memory, which is highly unproductive. By reusing the previous result and simply aggregating it with the new incoming data, as shown in Figure 3.1b, we can not only save a lot of time wasted on reaggregation but also focus the freed up resources on performing only the new aggregation for a double boost in performance.

This refined approach can be extended to adding multiple new nodes at once. Rather than compute an overlay with all leaf nodes an overlay can be established with all new leaf nodes and the previous root node. All leaf nodes and intermediate nodes of the previous tree would be explicitly omitted because they are implicitly included with the previous root node.

If the entire overlay is expected to be reused, such as when the underlying data at several nodes are changed in a real time monitoring system, there is a tradeoff between total reaggregation and the and refined solution. The refined solution is much faster and resource efficient for integrating the data from the new nodes because it is not taking the time to reaggregate anything from the existing nodes, but if the resulting overlay is used again the result will be suboptimal for those future uses of the overlay. In practice append only data is the norm for some applications [33], and HDFS even makes files append only [34]. If data is only appended without updates or deletions this tradeoff is never encountered.

Thus if NOAH is configured to use append only data, the overlay is updated by pairing the old root with new nodes. If the data may be changed at any node then adding new nodes triggers a recomputation of the overlay and reaggregation of the existing data.

Co-locating Nodes



(a) Update Path (b) Virtual and Physical Overlays

Figure 3.2.: Exploiting the update path for lower maintenance with infrequent updates.

We can only exploit the parallelism of multiple machines while there is work to be done. If updates to existing data are expected to be infrequent it makes sense to colocate aggregator nodes to fewer machines. Unused machines may be released so the user does not have to pay for them. There is an additional trait of the aggregation overlay which makes this more useful. When the data at a leaf node is changed it only affects the aggregation along its path to the output. Figure 3.2a shows the reaggregations that must happen if a change occurs at the data in one leaf.

The shaded nodes run sequentially, not in parallel, and unshaded nodes need not run at all. There is no penalty to colocate nodes along the path, as they do not compete for computational power at the same time. Furthermore, multiple update paths do not conflict until the paths merge.

Rather than maintain separate machines for each node, nodes may be virtually moved to the same machine. Figure 3.2b shows 3 machines with 7 nodes. Boxes represent physical machines containing nodes. The overlay is shown in dashed lines,

and solid lines show the communication lines. Each machine has a communications manager (CM) to handle communication for the nodes on the machine.

The CM has a few jobs. One is ensuring communication is still valid. If it receives communication meant for a node which has moved, it forwards the packets and informs the sender of the updated location. Another job is to control network access. If a communication arrives at the CM and is meant for another node on the same machine, the CM sends the communication directly to the recipient without using the network. This is similar to what is exploited in previous works on high performance distributed computing networking research [35]. The speed difference between RAM and the network saves significant time.

When the system controller selects a node to move the CM serializes it and forwards the node to the CM at the destination. It informs the CMs of nodes which communicate with the moving node of the new location. If the node is the last node on a machine the CM waits a set period of time to intercept and forward any communications started before knowledge of the move propagated, then shuts down the machine.

A node needs the inputs from each of its children in the hierarchy to aggregate. Thus each node keeps copies of the latest input from each child for future use. When it receives a new input from a child it replaces the copy of the input from the appropriate child and runs the aggregation with all available inputs before forwarding the result.

Colocation uses fewer machines, reducing system maintenance costs. When nodes are placed to avoid conflicting update paths the aggregation can also be faster due to the use of RAM for communication instead of the network.

This is different than traditional multi-tenency in which multiple virtual machines are placed on a physical machine. Cloud providers often mask virtual machine placement, so we cannot rely on preformance gains from traditional multi-tenency without full control over placement on the network and hooks into the networking layer via our CM. Even then there would be no reduction in virtual machines, so there would be no cost savings.

3.3 Experiments

Here we empirically show the performance of NOAH via microbenchmarks and real world problems. We show the effects of changing the fan-in and the accuracy of the heuristics. We also note how one-size-fits-all overlays such as those used by other systems would fare.

3.3.1 Setting

For our experiments we use nodes in a single Amazon EC2 datacenter. Unless noted otherwise the experiments use m3.medium nodes. While we did see measurable difference if we terminated a set of nodes and reran an experiment on a new set, probably due to different placement in the network, the differences were not significant. Thus we consider it sufficient to obtain average readings for each set of experiments on a single set of instances rather than capturing averages across multiple deployments, which would hide the variability in EC2 performance.

When isolating the aggregation phase we wait for the leaves to complete their computation and inform the controller. The controller then sends a signal to begin aggregation and times how long it takes from that point to receive the final result from the root. On the other tests we start timing from the beginning of the computation phase.

3.3.2 Microbenchmarks

Our first set of tests are microbenchmarks with varying values for y_0, the variable used to choose the fan-in, to validate the heuristics. In the computation phase leaf nodes generate random lists of integers of a given size. Aggregators generate a series of random numbers proportional to the size of their inputs and prune the list to the size dictated by y. Thus y_0 is maintained perfectly.

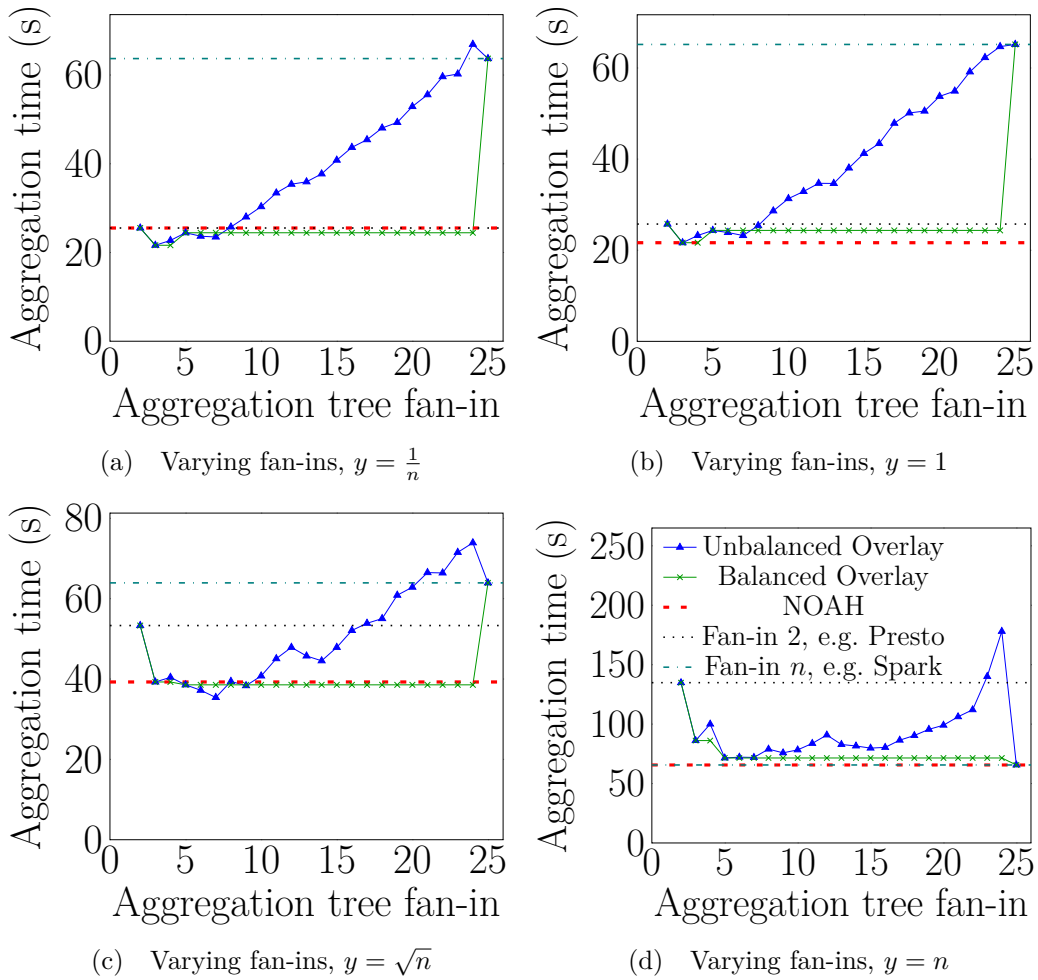
We use 25 leaves. For every fan-in from 2 to 25 we show two overlays. The unbalanced overlay assigns children to aggregators until the prescribed fan-in is hit, then continues with the next node until all nodes are included. The balanced overlay first determines the lowest fan-in with the same height as the given fan-in to build the most balanced tree with the same height.

We also project the aggregation time from three points across the graphs. The red dashed line is the aggregation time of the balanced overlay when the fan-in equals the heuristic for the given value of y_0, which corresponds to the overlay NOAH builds. The other two projections are the aggregation times for one-size-fits-all aggregation trees with fan-ins of 2, such as that used by Presto, and n, such as that used by Spark or MapReduce when a single reducer has to compare all items to filter them.

Figure 3.3 shows the microbenchmark results. In Figures 3.3a-3.3c predictable trends in the non-balanced overlay emerge, and the balanced version remains steady for all values with the same height as expected. The balanced overlay regularly outperforms the unbalanced one, sometimes quite significantly. In the rare cases the balanced overlay underperforms it does so only very slightly. The heuristic choices of 2, 3, and 4 are close to the experimental optima – both in the value of the fan-in and in the resulting relative performance.

In Figure 3.3d there are noticeable peaks and valleys in the unbalanced overlay's performance as the overlay gets closer and farther from balanced. The balanced overlay performs much better in general, although the two approaches reconverge at fan-in 25, which is the heuristic fan-in and experimental minimum.

In Figures 3.3a and 3.3d the heuristic aligns with different one-size-fits-all overlays. In the first case it is with the fan-in 2 preferred by Presto. In the latter case it is with the fan-in of n required by Spark when all data must be compared to each other. Additionally, in 3.3b the overlay with fan-in 2 performs very close to the heuristic. The more interesting results in Figure 3.3c show both one-size-fits-all approaches underperforming the heuristic significantly.



(a) Varying fan-ins, $y = \frac{1}{n}$

(b) Varying fan-ins, $y = 1$

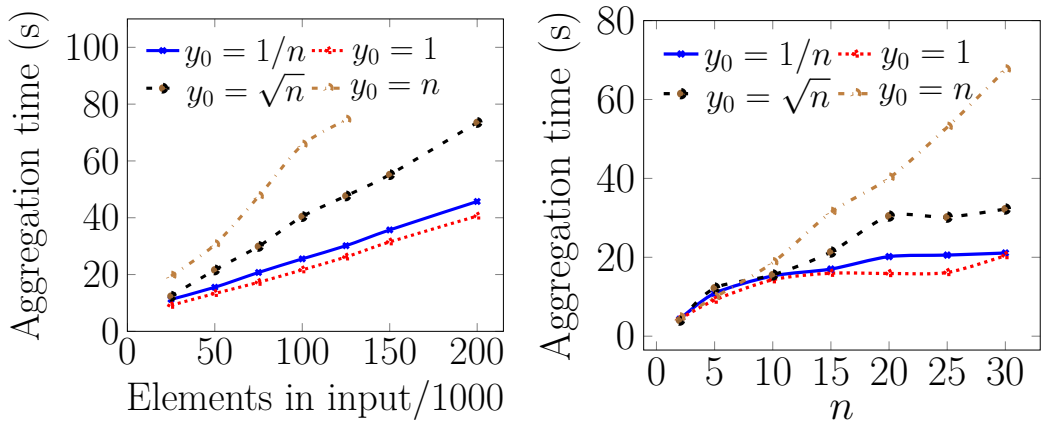
(c) Varying fan-ins, $y = \sqrt{n}$

(d) Varying fan-ins, $y = n$

Figure 3.3.: Microbenchmarks for various values of y_0. The legend in Figure 3.3d applies to all.

We also consider the effects of the number of leaf nodes and the size of the input data even though they are not controlled by the aggregation overlay. In this way we can understand how NOAH responds to different loads. For each of the microbenchmarked values of y_0 we use NOAH and vary n and input size independently while keeping the other fixed. n is varied from 3 to 30, and input size is varied from 25000 elements per leaf to 200000.

Figure 3.4a shows the effect of changing input size on 25 nodes using the same y_0 values as the earlier experiments. The final two points for $y_0 = n$ are omitted as the



(a) Varying input size

(b) Varying n

Figure 3.4.: Effects of variables other than fan-in on aggregation latency.

memory required at the root exceeds the available RAM, and the latency increases dramatically with disk accesses. In all other cases the latency is very predictably linearly correlated with the input size.

Figure 3.4b shows how the number of leaves affects the latency of the system for a given input size at each leaf and the heuristic fan-in. In the case of $y_0 = n$ the relationship is predictably linear. Smaller values of y_0, which have smaller fan-ins and thus higher parallelism, are less sensitive to changes in the number of nodes, but the latency still grows.

3.3.3 Common Problems

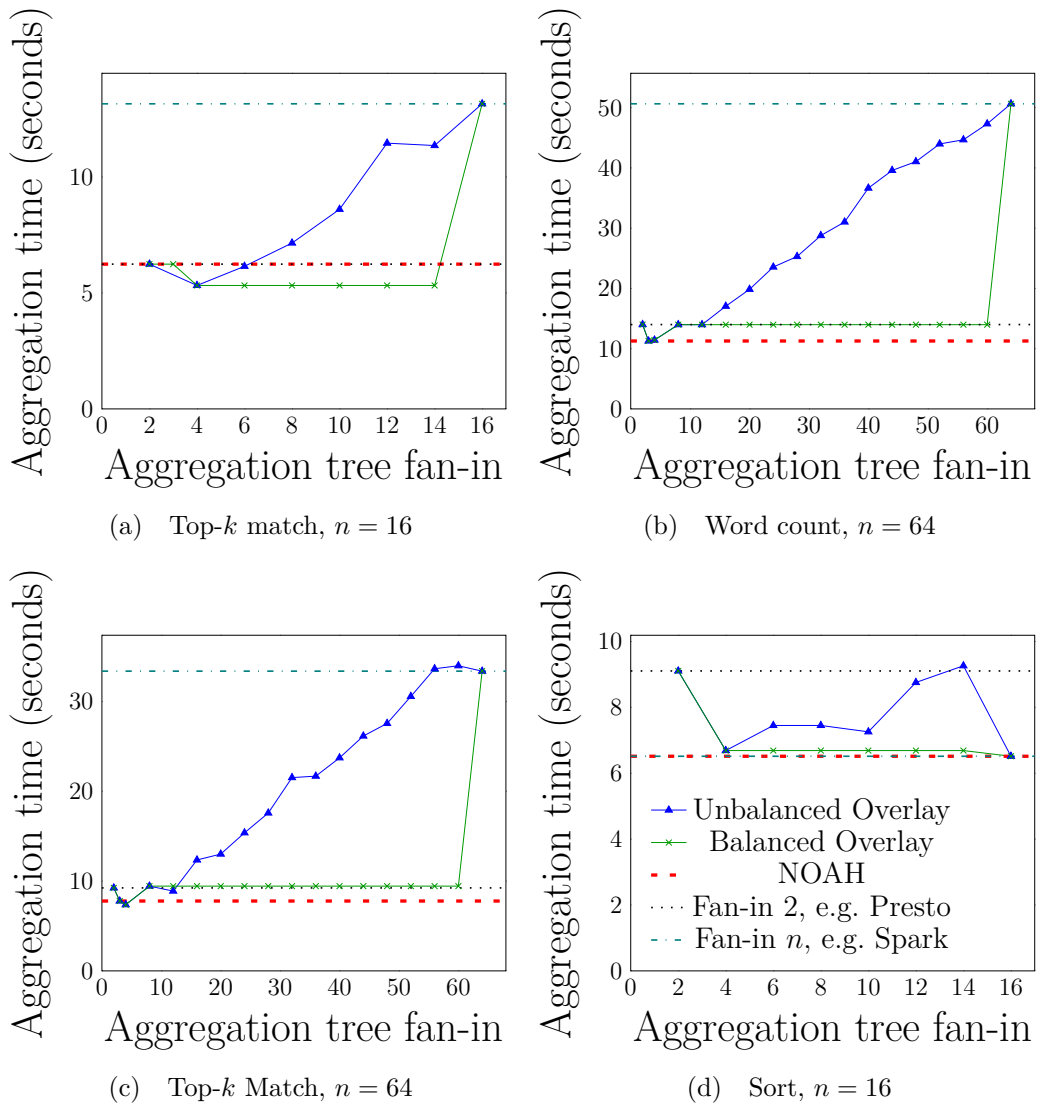
Next we evaluate the heuristics with common aggregation tasks. We perform top-k matching on generated data which, when placed in data structures in memory, take up a little over 200MB per node. The computation phase prunes this to less than 10MB per node for aggregation. We also run word count and sort applications on a dataset consisting of log files of Yahoo's Hadoop clusters [36]. 34MB of input data is distributed such to each leaf node. In more detail the application are:

- Top-k match – a simple equation "scores" how well generated entries match a filter. The k entries with the highest score are returned in sorted order from each leaf. Aggregator nodes forward the k highest scores across their inputs, which means $y_0 = 1$. The final aggregate result is the k highest scoring matches from all generated data. The complexity of our aggregation function is $O(|x| \log d)$, and thus depends on the fan-in and size of input separately.
- Word count – counts the occurrences of each word in the log files at the leaves, and then sums the results. The final aggregate result is a map of each word in the logs and the number of times it appeared across all logs. Log entries are not very disparate, so any word which appears in the logs at one leaf has a high probability of appearing in the logs at every other leaf, so if y_0 is greater than 1, it is negligibly so.
- Sort – Using the scoring mechanism similar to the top-k match, all lines are scored and sorted on how well they match a filter. All lines are returned in sorted order, and vertices forward a sorted merging of all inputs, so $y_0 = n$. The complexity of our aggregation function is $O(d\,|x| \log d)$, and thus depends on the fan-in and size of input separately as well as together.

We note that the size of the data for the top-k match and word count application could easily be scaled up at each node at the expense of longer computation phases. Since both application reduce the size of the data to a reasonably sized data structure explaining the nature of the data this would not affect the aggregation phase.

Figure 3.5 shows the results from our common aggregation problems on real world data for a selection of 16 and 64 leaf nodes. Sorting is shown with 16 leaves due to unpredictable behavior when larger input overflowed the RAM in larger overlays. We show 16 and 64 nodes for top-k matching to compare scalability. In all cases the heuristics are close to the experimental optima.

The relative impact of choosing the right fan-in increases as n increases. For the top-k matching applications with 16 nodes the worst case fan-in takes 247% as much time as the best case fan-in. When we increase n to 64 that number jumps to 453%.



(a) Top-k match, $n = 16$

(b) Word count, $n = 64$

(c) Top-k Match, $n = 64$

(d) Sort, $n = 16$

Figure 3.5.: Experiments with aggregation with common real-world problems. The legend in Figure 3.5d applies to all.

The sorting application in Figure 3.5d sees less improvement. While the heuristic properly chooses the best fan-in, the latency is only 2.5% faster than the next fastest point and only 40% better than the worst case. It is worth noting that points in which the full and balanced assumption is met are at or below the trendline, while the other points, which have slight imbalances, are above, thus reinforcing our statements

that the assumptions affect the ability of the model to predict the time accurately, although all overlays presented correct output.

These tests are also a good indication of why a one-size-fits-all approach does not work across different problems. We note that while the fan-in of 2 like that used by Presto performs only slightly worse than overlays chosen by NOAH for top-k matching and word count in Figures 3.5a-3.5c, it is noticeably slower for sorting in Figure 3.5d– purely as a result of the aggregation overlay.

Word count would actually be better performed on Spark or another in-memory MapReduce style system, as the keys would allow data to be partitioned to avoid complete aggregation. Sort may or may not have that advantage depending on whether smart partitioning is possible, but at worst it would have a fan-in of n, which is the heuristic. However, for the top-k matching it is necessary to run a single reducer, which is effectively the same as an overlay with a fan-in of n in Figures 3.5a and 3.5c. Based on these experiments such a setup would perform significantly worse than an overlay with a smaller fan-in, and it appears that it will only get worse as the number of nodes gets larger, as is the case for many big data applications.

3.3.4 Iterative Computations

NOAH's in-memory aspect makes it well suited to iterative problems. A good example of this is k-means clustering, a common machine learning problem. Some selection of points is chosen for likely centers of clusters. Points are then assigned to clusters based on the nearest estimated center, and a new cluster center is computed for each cluster. The process repeats with the new estimates.

To distribute the problem the points are distributed across the leaf nodes. Each leaf determines its next set of 100 estimated cluster centers and sends that list to its parent. The parent runs the k-means algorithm on the suggested centers of clusters to create an aggregate set of suggestions. k is fixed, so $y_0 = 1$. It is worth noting that the aggregation method is of the order $O(k^2)$ because each suggested center is

tested against all centers from the previous iteration, so this is a superlinear problem. k-means often does not arrive to a perfect answer, but tends to gets closer with each iteration [37]. Thus it is advantageous to run as many iterations as possible in a given amount of time.

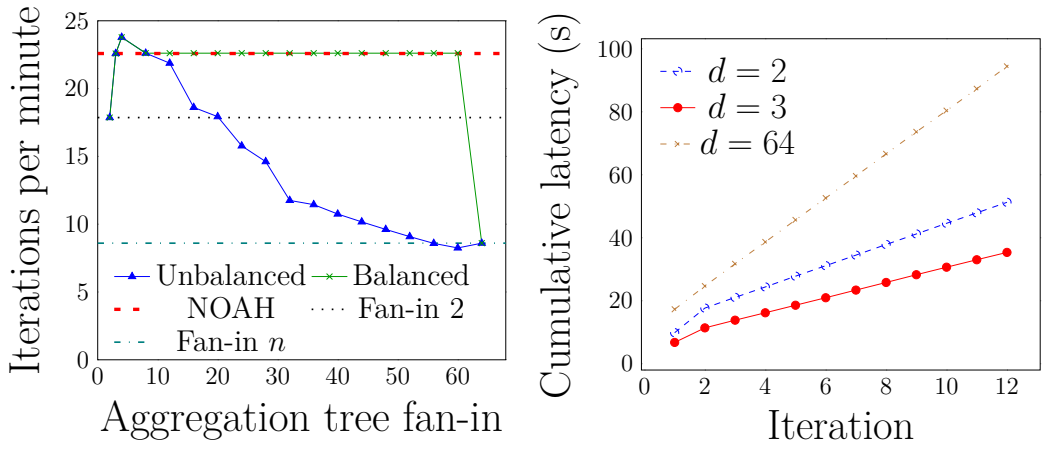
We run the k-means test on a set of 3-dimensional road coordinates in Denmark containing longitude, latitude, and elevation [38]. 400000 entries are distributed across 64 leaf nodes, and we vary the fan-in from 2 to 64. We also project the performance from the overlay with NOAH's heuristic and the one-size-fits-all overlays across the graph for comparison again. After a 2 iteration warm-up phase the times for iterations stabilize. The results in Figure 3.6a include the stable rates after the warm-up phase is complete, but the analysis including the warm-up phase is consistent with the results.

This single application is a stark indicator of why one-size-fits-all overlays are sub-optimal. In this particular case MapReduce and Spark are once again forced to use a single reducer of fan-in n in order to account for all cluster centers in each iteration, which is the worst case scenario. The NOAH overlay outperforms that worst case by 270%, and is within 5% of the experimental optimum of fan-in 4. The overlay with the heuristic fan-in also outperforms the overlay with a fan-in of 2 favored by Presto by a less spectacular, but still significant, 26%.

Figure 3.6b shows the cumulative effect of choosing a one-size-fits-all versus the heuristic based approach on the first 12 iterations. Latency is strongly linear with the number of iterations, so the absolute difference predictably grows with each iteration. The figure clearly shows the improvements in absolute time over either one-size-fits-all overlay, and the difference grows with the number of iterations.

3.3.5 NOAH as a Subsystem

Iterative computation can be viewed as a special case of a more general workflow containing an aggregation problem (executed iteratively). NOAH is designed precisely as a subsystem in a larger workflow execution environment when aggregation time is a



(a) Iteration rate

(b) Cumulative latency

Figure 3.6.: Iterative k-means clustering.

significant contributor to overall latency. To show the impact of this one component we use NOAH as part of a larger analysis job to find the most active nodes participating in MapReduce at a datacenter. For the compute-aggregate portion of this problem, 50MB of logs are placed in each of 16 leaf nodes. The compute phase greps the logs for host names and counts the occurrences in their respective logs. The aggregate phase sums occurrences across leaves. Finally, a post-aggregation phase takes the result, sorts it by number of occurrences, and retrieves the top-k active nodes.

We run this problem three times each for fan-ins 2, 3, 4, and 16 and present the averages for each phase. With the balancing mechanism all fan-ins from [4,15] effectively become a fan-in of 4. 3 is the heuristic selection for the word count family of problems, which is the aggregation in this case. The results are shown in Figure 3.7.

It is immediately clear that the only measureable variation between the runs is caused by the aggregation phase, which is to be expected. The compute phase is the same for each run, and the top-k phase is working on equivalent input. Aggregation takes 48-91% of the total time, which is significant. In fact, the minimal latency happens at the heuristically chosen value of $f = 3$. There is a modest 47% time

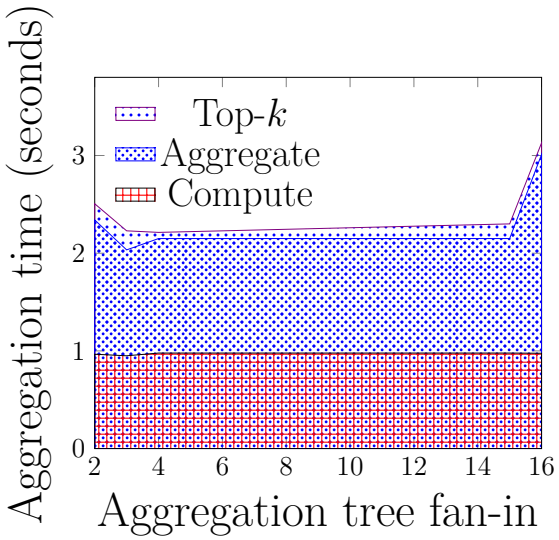


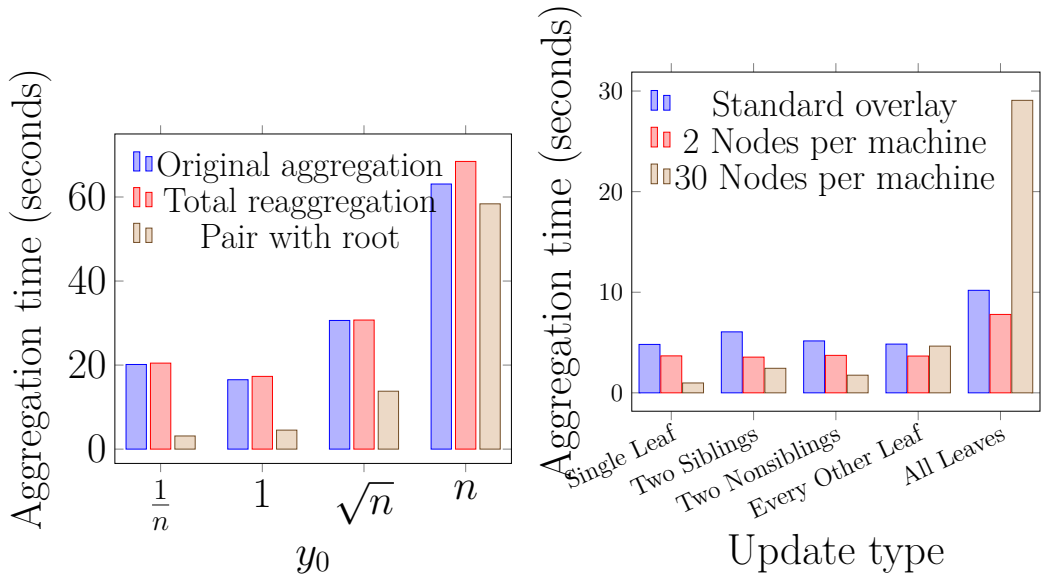
Figure 3.7.: Impact of NOAH as a subsystem, $n = 16$

savings in aggregation over the naïve approach of sending everything to a single node, which corresponds to a 29% lower latency of the system as a whole.

3.3.6 Incremental Overlay Updates

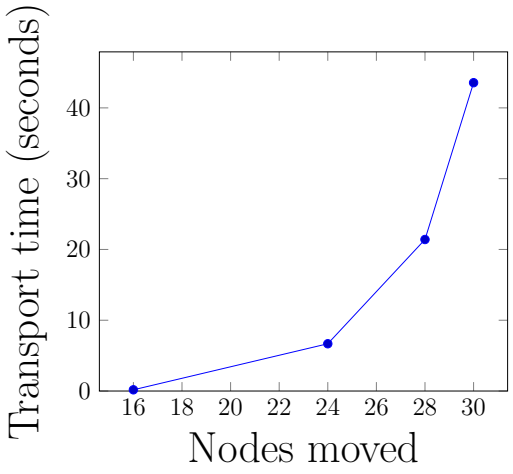
Given the 25 nodes from the microbenchmarks we explore what it would take to add a single new node. In Figure 3.8a we show the difference between creating a new overlay with 26 leaves and aggregating from scratch versus simply aggregating the prior output with the new data. We also include the original aggregation time for comparison. For both the 25 and 26 leaf overlays the heuristic for choosing the preferred fan-in for a given y_0 is used.

From the graph it is clear that reaggregating takes very slightly longer than the original aggregation took, as we would expect. Choosing to instead just aggregate the previous results with the new data is faster in all cases, although the amount of savings depends on y_0, or more specifically, on the size of the output of the previous data. In the case that $y_0 = \frac{1}{n}$ the approach of aggregating the prior output with the



(a) Adding a node, reaggregation vs. pairing with previous root

(b) Results for combining nodes on machines



(c) Time to transfer nodes to other machines

Figure 3.8.: Maintenance mechanism performance.

new data directly is 85% faster than a new overlay. When $y_0 = n$ the difference is only 15% faster.

We note that if the overlay is reused each subsequent time, the second approach would require time equal to the original aggregation time plus the time to aggregate those new results with the data from the new node. In all cases that combined time

is greater than the cost of a new overlay. So it makes sense to use this incremental approach if the overlay will not be required to reaggregate or perform a new aggregation function, such as would happen if data at a node is modified. However, if the data is write only this shows a piecemeal approach makes sense.

3.3.7 Node Colocation

Our final experiment combines nodes onto the same machine. To show the behavior at without the noise of excess data we present only the results from the extremes – combining 2 nodes per machine and 30 nodes per machine. We also use micro instances on EC2, which have low priority on the network, and microbenchmarks that require relatively little computational power, thus exacerbating the effects of using the network. The same trends hold in other environments where communication dominates computation. Figure 3.8b shows the results on latency of reaggregation after updates to the data. Nodes store and reuse the output from unaffected paths, so reaggregation only recomputes along affected paths. We test updating a single leaf, updating two sibling leaves, updating two leaves on opposite sides of the tree, updating every other leaf, and updating all leaves. $y_0 = 1$ for this test.

Combining two nodes per machine always outperforms the original overlay because of the CM allows data transfer through RAM instead of relying on the network, as many nodes share a machine with their parent. Others share a machine with a sibling, causing resource contention, but the significant savings in communication overcome this.

The higher degree of combination saves even more time when updating a single leaf. Updating a single node runs a single branch of the tree without conflicts. The difference between this and a branch of original overlay is the network usage. Thus the latency difference of 80% between the two physical overlays (while maintaining the same virtual overlay) is explained by the difference in the traffic over the network.

The ideal degree of combination depends on the network attributes, frequency and number of conflicting updates, and the aggregation method.

We consider two other costs associated with colocation. The first is the time to move nodes to their new homes, shown in Figure 3.8c. The time taken is not linear. Nodes, including the stored prior results, are large. Saturation of the network, the costs of serializing and deserializing multiple streams, and overloaded CMs sending and processing updates slow things considerably. A better approach is combining nodes piecemeal rather than all at once.

Lastly we note the cost savings of colocation. As EC2 and other cloud providers charge per unit time a machine is running, an overlay maintained indefinitely ameliorates the cost of the initial aggregation and the cost of the system relates to the inverse of the degree of combination. Ergo 30 nodes per machine leads to savings of 97%.

4 TOP-K MATCHING – A PRACTICAL APPLICATION

In this chapter we concentrate on one particular task that can be run on a NOAH leaf node – top-k matching. We extend the matching algorithm over prior art and given an efficient solution before creating a distributed solution with NOAH. This work appeared in Middleware [39].

4.1 Overview

Top-k matching is an important computation problem requiring fast execution [40, 41]. In top-k matching, entities wishing to be matched, e.g. advertisers, request to add subscriptions to a system, where each subscription corresponds to an advertisement ("ad"). When an event, such as a browser event representing a webpage view, triggers the matching algorithm, the algorithm finds and returns the k best matching subscriptions based on the user's interests and attributes (e.g., sex, age, location, etc.). Corresponding middleware solutions are widely used for online ads, e.g., text or banner ads based on user profiles and browsing histories, streaming videos ads, social network "online deals", etc.

4.1.1 Expressive Matching

Targeted ads may affect purchasing intent by 65% [42] and click through rates by 670% [43], so *effective* matching is vital to advertisers. As online behavior and personal information are more effectively tracked, especially via social media, using those data for matching becomes more relevant. In 2011 93% of companies reported using social media for marketing [44], and not too long ago Facebook announced a program to serve demographically targeted video ads to its users [45]. Users of social

media are however quickly deterred by unfocused ads [46]. This can be avoided by *more expressive* events, subscriptions, and matching between these.

Consider *ad exchanges* – marketplaces for ads used by Google, Facebook, Yahoo!, and others. Ad exchanges can be viewed as networks of processes routing advertisers to consumers (email users, website visitors, video viewers, etc.). A consumer's arrival can be modelled as an event with attribute values determined by scripts (read from cookies or otherwise inferred). The values of such attributes representing user information are typically actual values. Some attributes may instead have "*undefined*" values or be captured by *intervals* of values. An example event for a user arrival is `(fName: Jack, lName: UNKNOWN, age: [18..29], state: Indiana)`.

Advertisers typically do not need individual constraints to match a consumer entirely. For example, an advertiser for spring break airfares from Chicago to Cancun submits a subscription with preferences for consumers of the ad, e.g., `(age` $\in$ `[18,24]` $\wedge$ `state` $\in$ {`Indiana, Illinois, Wisconsin`}`)`. Advertisers assign a *weight* to each attribute and request the ad be delivered to consumers with a significant combined weight of matching attributes. The weight on the attribute signifies the relevance of serving the ad to the consumer; the advertiser assigns higher weights to more relevant attributes.

Campaigns have fixed budgets to spend in a specified time period, so there is an advantage to *dynamically adapt the scoring mechanism* to fully utilize budgets without overspending. Consider an ad campaign for a concert with a fixed budget for a given period of time. The campaign does not want to exhaust the budget within a day nor serve so few ads as not to use it entirely. Rather than force the campaign's owner to monitor progress and change the weights in an attempt to match at the desired rate, the system can track the budget, the length of the campaign, the desired rate of matching, and the historical rate of matching to *adjust a score multiplier* to achieve the desired matching rate.

For a consumption medium, a number of ads per access should be chosen which is minimally intrusive and economically viable. Each access generates an event with the

available known attributes and sends it to the exchange. The exchange returns the k ads best suited to display for while balancing the advertiser preferences as well.

4.1.2 Fast *and* Expressive Top-k Matching

One key bottleneck in the design of middleware systems for top-k matching is the latency of the matching algorithm, i.e, time taken to compute matches for events. Several specialized approaches have been developed for top-k matching. Examples are Fagin's seminal aggregation algorithm [12], SOPT-R Trees and scored segment and interval trees in combination with Fagin's algorithm [40], and BE*-Trees [41]. For scalability, a distributed overlay can be used (e.g., [27, 29]). *Distributed* top-k matching aggregates and prunes sets from top-k matching at individual nodes, so any expressiveness desired for the system must be implemented centrally.

Motivated by recent application scenarios, we propose in this paper a more *expressive* model of weighted partial top-k matching than considered in prior art, and an *efficient*, novel algorithm dubbed Fast eXpressive Top-k Matching (FX-TM) for implementing it. Our model supports:

(a) explicit handling of *interval attributes* and optional *prorated scoring* when there is partial overlap between subscriptions and events. This corresponds to scenarios where exact values for user attributes are not known (e.g., targeted age [18,24] and consumer age [20, 30]).

(b) attribute weight on *either* the subscription *or* the event used during aggregation. For instance, a company may favor experience over applicant location while a job seeker may prefer proximity over experience requirements. Our model allows each of these, and can switch between approaches for each matching iteration.

(c) *non-monotonic aggregation functions* and *mixed positive and negative weights* forbidden by some of the state-of-the-art. This is useful when some attribute

values are undesirable, e.g. an age range below the voting age, while others, e.g. income, are desirable.

(d) missing attributes and wildcards, finding matches with the best scores even if a match is not *exact* (i.e., does not match all attributes). This is useful as some groups are more prone to self disclosure [47]. Available data is matched; missing data does not disqualify a match.

(e) *dynamic alteration of scores* based upon the rate a subscription is matched to events and versus its specified preferred rate. This helps an ad campaign match as many events as its budget allows over a specified period of time without depleting its budget early.

Our output-sensitive algorithm runs a single match in time $O(M \log N + S \log k)$ for N subscriptions, M attributes in the event, and S matching elementary constraints. The complexities hold regardless of chosen expressiveness options.

The concrete contributions of this chapter are as follows:

- a model of expressive weighted top-k matching with the features outlined above.
- an efficient algorithm FX-TM of our model, including a score adjustment device to achieve desired matching rates for each subscription.
- a formal complexity analysis of our algorithm qualifying its scalability.
- an implementation of our FX-TM algorithm including its distributed application over several nodes with subsequent aggregation.
- an empirical evaluation consisting of statistical micro-benchmarks and two benchmarks derived from real-life data. We show that FX-TM performs at least as well as state-of-the-art solutions before considering our added expressiveness and dynamic score adjustment; when considering these, FX-TM surpasses existing approaches after attempting to upgrade those.

4.2 Model

This section defines events, subscriptions, and matching.

4.2.1 Matching Events and Subscriptions

We consider an event e to be a set of attribute/interval pairs $\{a_1 : [v_1, v_1'], \ldots, a_l : [v_l, v_l']\}$. Events are only required to include attributes whose values are known, but may specify attributes with `UNKNOWN`. For simplifying presentation and comparison with predating work, we consider subscriptions to be *conjunctions* of constraints.

A subscription is thus in the following represented as a predicate ϕ based on the following grammar (extended BNF):

$$\begin{aligned} \textit{Predicate} \quad & \phi \;:=\; \phi \wedge \delta \mid \delta \\ \textit{Constraint} \quad & \delta \;:=\; a \;\in\; [v, v'] \;:\; w \end{aligned}$$

Interval subscriptions are predicates where the only boolean operator is $\in$. We call to subscriptions where predicates have relational operators *regular* subscriptions. A predicate `x>100` for some integer `x` is expressed as `x`$\in$ [`101`, `MAX_INT`]. A predicate for a single value or member of a set is represented by $v = v'$ so the "interval" contains only a single value. Each Constraint has an optional weight w attached.

To support weights in events instead of subscriptions, one can consider the events being augmented with weights w on attributes, $\{\langle a_1 : [v_1, v_1'], w_1\rangle, \ldots, \langle a_l : [v_l, v_l'], w_l\rangle\}$, which, when they exist, override the weights in subscriptions.

Interval subscriptions are as expressive as regular subscriptions. Matching involves substitution, i.e., substituting the values of attributes in events into constraints in a subscription. We denote this for a single constraint by $\delta(e)$. For example, if $e = \{a_1 : [v_1, v_1'], \ldots, a_l : [v_l, v_l']\}$, and $\delta = a \in [v_1'', v_1''']$, $\delta(e) = [v_1, v_1'] \in [v_1'', v_1''']$. $\delta(e)$ evaluates to TRUE or FALSE. When an attribute for some event is missing or `UNKNOWN`,

$\delta(e)$ for that attribute evaluates to FALSE, as an unknown value cannot reasonably match a known interval.

Definition 4.2.1 (Match Score)**.** *Given event e and subscription $\phi = \bigwedge_{i=1}^{n} \delta_i : w_i$, with $\delta_i = a_i \in [v_i, v_i']$, $score(\phi, e) = \sum_{i=1}^{n} w_i \mid \delta_i(e) =$* TRUE*.*

For proration, the weight for an attribute is multiplied by the ratio of the size of the interval intersection to the size of the interval of the event. We note that proration is used as a confidence metric, and this practice can be extended to other confidence or similarity metrics.

Definition 4.2.2 (Prorated Match Score)**.** *Given event $e = \{a_1 : [v_1, v_1'], \ldots, a_l : [v_l, v_l']\}$ and subscription $\phi = \bigwedge_{i=1}^{n} \delta_i : w_i$, with $\delta_i = a_i \in [v_i'', v_i''']$, $score(\phi, e) = \sum_{i=1}^{n} w_i \times \left(C + \frac{\text{MIN}(v_i''', v_i') - \text{MAX}(v_i, v_i'')}{v_i' - v_i} \right) \mid \delta_i(e) =$* TRUE*. $C = 0$ for continuous intervals, and $C = 1$ for discrete intervals to account for the overlapping at the endpoints.*

The top-k matching problem consists in determining the top-k matching set, defined as follows:

Definition 4.2.3 (Top-k Matching Set)**.** *Given a set of subscriptions Θ, and an event e, the weighted partial top-k matching set $\Phi \subseteq \Theta$ is defined as $\Phi =$ $\{\phi \mid score(\phi, e) > 0 \wedge score(\phi, e) \geq score(\phi', e)\ \forall\ \phi' \in \Theta \setminus \Phi\}$ and $|\Phi| = k$.*

This definition does not specify handling of ties. For example, if Θ contains $k + 1$ subscriptions yielding the same score for an event the implementation must pick k of those subscriptions. The definition also requires at least k subscriptions match each event with a positive score. Otherwise the implementation chooses between returning fewer than k results or including some results with scores of 0.

4.2.2 The Budget Window

In some systems which use top-k matching, advertisers specify a budget and a time period to serve their ads. The goal is to spend as close to the budget as possible

in the given time when paying per match. If a subscription is matched too often, the budget is exceeded. If not matched enough, the advertiser is underserved. Bhalgat et al. [48] explain smooth delivery over the time window is preferred.

Hitting the target budget for a subscription is highly unlikely unless the scoring mechanism adjusts for the time window and budget. When a subscription is added, two values accompany it – the length of the window and the *budget*. The *begin time* is when the subscription is added, and amount *spent* is set to 0. The *end time* is *begin time* + *window length*. There is an additional optional configuration, $g(t)$, to specify the preferred matching distribution over time. $\int g(t)\,dt$ must be zero at *begin time* and monotonically increase to model ideal spending in the absence of negative spending. When not specified $g(t)$ defaults to $g(t) = 1$.

The multiplier must be less than 1 for subscriptions matching too often, and reduce their likelihood to be in the top-k. The opposite occurs for subscriptions not matching often enough. This leads to the following definition:

Definition 4.2.4 (Budget Window Score)**.** *Given event e, subscription ϕ, and score(ϕ, e)' from Definition 4.2.2, score(ϕ, e) = score$(\phi, e)'$* $\times \frac{budget}{spent} \times \frac{\int_{begin\ \ time}^{current\ \ time} g(t)dt}{\int_{begin\ \ time}^{end\ \ time} g(t)dt}$

4.3 FX-TM Algorithm

This section presents our novel top-k matching algorithm.

4.3.1 Overview

Some existing approaches, including BE* trees [41], use a single data structure, and thus exhibit a worst case complexity linear in the number of subscriptions. The key strategy adopted by FX-TM is *partitioning by attribute*, i.e., to use one data structure *per attribute* (see Algorithms 1 and 2). We assume subscriptions from clients are stored on a *matcher*, which is a node to match events to subscriptions. Events arrive at the matcher which returns k matching subscriptions. A subscription is of the form $\phi = \delta_1 \wedge \cdots \wedge \delta_n$ where each δ_i is on a different attribute a_i. Every subscription ϕ

Table 4.1.: The data structures used in FX-TM with relevant function call names and execution time bounds.

Data Structure	Insertion	Deletion	Retrieval
Interval Trees [49]	TREE-INSERT: $O(log\, n)$	TREE-DELETE: $O(log\, n)$	GET-MATCHING-INTERVALS: $O(log\, n + s)$ for s matches
Tree Set [50]	TREESET-ADD: $O(log\, n)$	TREESET-REMOVE-MIN: $O(log\, n)$ TREESET-REMOVE-ID: $O(log\, n)$	TREESET-FIND-MIN: $O(log\, n)$ TREESET-GET-ALL: $O(n)$
Hash Map	HMAP-PUT: $O(1)$	HMAP-DELETE: $O(1)$	HMAP-GET: $O(1)$ GET-ALL-KEYS: $O(1)$ (plus traversal)

is uniquely identifiable by *sid*. Weightings on elementary constraints of the attributes in the aggregation function can be either specified by subscriptions or specified by the event, with the latter option taking precedence. Proration can be used in either case.

In this section we refer to three main data structures which are used in FX-TM. The names of the relevant operations and their complexity bounds are in Table 4.1. Some interval tree operations may appear faster than the accepted operation, but they are achievable with some implementations, including that proposed by Arge and Vitter [49].

4.3.2 Subscription Partitioning

FX-TM employs a two-level index for subscriptions. Constraints on attributes are grouped together in an attribute-level index. A "master index" (Line 1) maps attribute names to (pointers to) data structures. Subscriptions are stored on matchers with predicates split by attribute name. Figure 4.1 shows a visual representation of the two-level approach. The structure for each attribute contains indexing information only for subscriptions containing that attribute.

We implement two types of structures for attributes. Attributes with discrete individual values use a hash map with the values as the keys and a tree set of

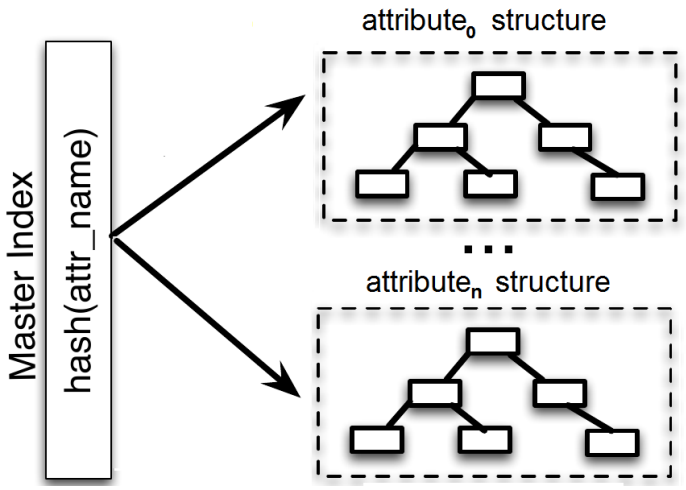


Figure 4.1.: A representation of the two-level index.

matching subscriptions as the values. The tree set is ordered on subscription ids *sid* for quick insertion and deletion, but retrieval returns a list of all items in that tree. The second type of attribute structure is an interval tree for ranged attributes. Attributes are defined as ranged or exact matching so that the proper structure can be selected, and the selection must be consistent for all subscriptions with constraints on that attribute.

For example, a subscription `x = 'Indiana'` $\wedge$ `y`$\in$ `[60,100]` $\wedge$ `z` $\in$ `[0,2000]` is partitioned into its elementary constraints by attribute, and each of the constraints is stored in a separate attribute structure. Finding the structure into which a constraint on an attribute has to be inserted is done with the master index. Retrieving the top-k subscriptions that match an event is done by searching each of the relevant structures (possibly in parallel), and combining the results of the individual subscriptions. Thus, partitioning can be more advantageous when the matcher executes on multi-core machines or if each interval tree is on a separate node. Here we only consider single threaded single compute node matching because most state-of-the-art top-k matching algorithms do not consider concurrency or distribution; this ensures a fair empirical comparison of FX-TM against the same.

Algorithm 1 FX-TM algorithm: adding and removing subscriptions. For simplicity, checking the attribute for ranged versus discrete values and using the appropriate structure is omitted in favor of assuming interval attributes.

```
 1: masterIndex ← new hash map
 2: budgetInfo ← new hash map
 3: prorate ← TRUE/FALSE

 4: upon RECV(ADD-SUBSCRIPTION, a_1 ∈ [v_1, v'_1] : w_1 ∧ ... ∧ a_n ∈ [v_n, v'_n] : w_n, sid) do
 5:     HMAP-PUT(budgetInfo, sid, φ)
 6:     for i = 1..n do
 7:         tree_i ← HMAP-GET(masterIndex, a_i)
 8:         if tree_i ≠ ⊥ then
 9:             TREE-INSERT(ROOT-OF(tree_i), [v_i, v'_i], w_i, sid)
10:         else
11:             tree_i ←new interval tree
12:             TREE-INSERT(ROOT-OF(tree_i), [v_i, v'_i], w_i, sid)
13:             HMAP-PUT(masterIndex, a_i, tree_i)

14: upon RECV(CANCEL-SUBSCRIPTION, a_1 ∈ [v_1, v'_1] : w_1 ∧ ... ∧ a_n ∈ [v_n, v'_n] : w_n, sid) do
15:     HMAP-DELETE(budgetInfo, sid)
16:     for i = 1..n do
17:         tree_i ← HMAP-GET(masterIndex, a_i)
18:         TREE-DELETE(ROOT-OF(tree_i), [v_i, v'_i], w_i, sid)
19:         if SIZEOF(tree_i) = 0 then
20:             HMAP-DELETE(masterIndex, a_i)
```

4.3.3 Adding and Removing Subscriptions

FX-TM employs one structure per attribute that exists in any subscription. FX-TM splits a subscription by attribute, and adds each constraint to the corresponding structure. If an existing subscription at a matcher has a constraint on a given attribute in a new subscription, then corresponding structure already exists in the master index (Line 8). The structure is retrieved (Line 7), and the constraint on the attribute is added (Line 9). If no existing subscription contains a constraint on the attribute, a new structure is created for the attribute (Line 11 and inserted into the master index with a corresponding key (Line 13) after adding the weighted attribute-level constraint to it. Removing subscriptions is similarly straightforward. The structure corresponding to each attribute in the subscription is retrieved from the master index (Line 17). Each attribute-level constraint is deleted.

Empty structures may be removed from the master index. Partial matching only score attributes which exist in subscriptions *and* events. If an event contains an attribute with no existing structure, no subscriptions match on that attribute, so it does not affect matching.

A separate data structure indexed on *sid* for maintaining the information necessary to use the budget window multiplier is also updated (Lines 5 and 15).

4.3.4 Matching

We use summation as the aggregation function here, as that makes the most sense for weighted matching, but FX-TM supports all the aggregation functions of prior art.

FX-TM uses a hash map (*scoremap*) to track aggregate scores (Line 22), and a tree set to maintain a set of subscription identifiers sorted by their scores. FX-TM retrieves the set of subscription identifiers and weights for constraints matching each attribute. If proration is being used (Line 35), FX-TM computes the intersection of the attribute's value and the value associated with each constraint (Line 36).

The match score for each subscription is updated by querying *scoremap* with the subscription identifier and adding the (prorated) weight of the matched constraint (Lines 37 and 39). After matching all attributes for the event and updating the scores in *scoremap*, the entries in *scoremap* are adjusted by their budget window multipliers and pruned using the tree set *topscores*, which is ordered on match scores. Lines 40-49 show how entries from *scoremap* are inserted into *topscores* so that the size of *topscores* never exceeds k. Subscription identifiers and their associated match scores are added to *topscores* only if the subscription is among the top-k matches already seen in this phase, thereby ensuring that subscriptions known to not be in the top-k are never added to *topscores*. When the size of *topscores* would exceed k, the subscription with the lowest score is discarded.

Algorithm 2 FX-TM algorithm: weighted partial matching of events to subscriptions.

21: **upon** RECV(MSG, $\{a_1 : [v_1, v'_1], \ldots, a_n : [v_n, v'_n]\}, \{a_1 : w_1, \ldots, a_n : w_n\}$) **do**
22: $scoremap \leftarrow$ new hash map *{Tracks match scores of partially matched subscriptions}*
23: $topscores \leftarrow$ new tree set *{Stores the results}*
24: $min \leftarrow 0$ *{Temporary variable used to store minimum score}*
25: **for** $i = 1..n$ **do** *{Loop through all attributes in event}*
26: $structure_i \leftarrow$ HMAP-GET$(masterIndex, a_i)$
27: **if** TYPEOF$(a_i) = interval$ **then** *{Get matching interval attribute constraints}*
28: $matches \leftarrow$ GET-MATCHING-INTERVALS$(structure_i, [v_i, v'_i])$
29: **else** *{Get matching discrete attribute constraints}*
30: $matches \leftarrow$ TREESET-GET-ALL(HMAP-GET$(structure_i, v_i)$)
31: **for all** $\langle sid, [v_r, v'_r], w_r \rangle \in matches$ **do**
32: **if** $\exists w_j \neq 0, 1 \leq j \leq n$ **then**
33: $w_r \leftarrow w_i$ *{If weights are specified on the event, use those }*
34: $score \leftarrow$ HMAP-GET$(scoremap, sid)$
35: **if** $prorate =$ TRUE **then**
36: $pwt \leftarrow$ PRORATE$(w_r, [v_i, v'_i], [v_r, v'_r])$
37: HMAP-PUT$(scoremap, sid, score + pwt)$
38: **else**
39: HMAP-PUT$(scoremap, sid, score + w_r)$
40: **for all** $sid \in$ GET-ALL-KEYS$(scoremap)$ **do**
41: $w \leftarrow$ HMAP-GET$(scoremap, sid)$
42: **if** SIZE-OF$(topscores) < k$ **then** *{The first k matches are the initial top-k}*
43: TREESET-ADD$(topscores, sid, w \times$ BUDGETWINDOWMULTIPLIER$(sid))$
44: **if** $w \times$ BUDGETWINDOWMULTIPLIER$(sid) < min \vee min = 0$ **then**
45: $min \leftarrow w \times$ BUDGETWINDOWMULTIPLIER(sid)
46: **else if** $min < w \times$ BUDGETWINDOWMULTIPLIER(sid) **then***{New score is > than at least one top-k previously seen}*
47: TREESET-REMOVE-MIN$(topscores)$ *{Remove min to maintain max size of k}*
48: TREESET-ADD$(topscores, sid, w \times$ BUDGETWINDOWMULTIPLIER$(sid))$
49: $min \leftarrow$ TREESET-FIND-MIN$(topscores)$ *{Get min for future comparison}*
50: **for all** $sid \in$ GET-ALL-KEYS$(topscores)$ **do**
51: SEND(MATCHED, $\{a_1 : [v_1, v'_1], \ldots, a_n : [v_n, v'_n]\}, \{a_1 : w_1, \ldots, a_n : w_n\}$) to RECEIVER-OF$(sid)$

52: **function** PRORATE$(w, [v_{bl}, v_{br}], [v_{il}, v_{ir}])$
53: **return** $\frac{\text{MIN}(v_{ir}, v_{br}) - \text{MAX}(v_{il}, v_{bl})}{v_{br} - v_{bl}}$ *{Fraction of event which overlaps. Add 1 in the case of inclusive integer intervals.}*

54: **function** BUDGETWINDOWMULTIPLIER(sid)
55: $\langle budget, spent, begin\ time, end\ time, g(t) \rangle \leftarrow$ HMAP-GET$(budgetInfo, sid)$
56: **return** $\frac{budget}{spent} \times \frac{\int_{begin\ time}^{current\ time} g(t)\,dt}{\int_{begin\ time}^{end\ time} g(t)\,dt}$

When message producers specify weights on attributes, the weights from the subscriptions are discarded in Line 33. Otherwise the weights from the subscriptions are used.

4.4 Complexity Analysis

Here we formally analyze the worst case time and space complexity of FX-TM. M is the number of attributes in an event or the number of constraints/attributes in a subscription. We assume M is the same for all N subscriptions and events. When that is not the case, the largest value for M determines a correct upper bound. We use interval trees for the analysis as in every case the $O()$ value is at least as large as the hash map required for discrete attributes.

Theorem 4.4.1. *FX-TM handles* ADD-SUBSCRIPTION *operations in* $O(M \log N)$ *time.*

Proof. *From Algorithm 1. Insertion into an interval tree takes* $O(\log N)$ *time (Lines 9, 12). A subscription is inserted to* M *interval trees. Operations on* $masterIndex$ *(Lines 8, 7,13) take* $O(1)$ *time. Thus the total time complexity is* $O(M \log N)$.

Theorem 4.4.2. *FX-TM handles* CANCEL-SUBSCRIPTION *operations in* $O(M \log N)$ *time.*

Proof. *From Algorithm 1. Deletion from an interval tree takes* $O(\log N)$ *time (see Line 18). The subscription is removed from* M *interval trees. Operations on* $masterIndex$ *(Lines 17,20) take* $O(1)$ *time. Thus the total time complexity is* $O(M \log N)$.

Theorem 4.4.3. *FX-TM performs matching in* $O(M \log N + S \log k)$ *time.*

Proof. *From Algorithm 2.* GET-MATCHING-INTERVALS *in Line 28 takes* $O(\log N + s_i)$ *time, where* s_i *is the number of matching intervals for attribute* a_i. *Let* $S = \sum_{i=1}^{M} s_i$, *i.e.,* S *is the total number of constraints matched across all attributes. To find matching intervals for each of the* M *attributes of an event, the total time complexity of querying the interval tree is thus* $O(M \log N + S)$. *Updating the subscription's score in scoremap is* $O(1)$ *(Lines 37 and 39) as is querying* $masterIndex$ *for interval trees (Line 26).*

$|scoremap| \leq S$. $|topscores| \leq k$, so inserting each subscription from scoremap to topscores takes $O(S \log k)$. There may be a maximum of $S - k$ removals and searches for the tree minimums, which has an additional time complexity bounded by $O(S \log k)$. There are S budget window updates with constant time each for a total of $O(S)$, which is less than $O(S \log k)$. Thus the total time complexity is $O(M \log N + S + S \log k)$, i.e., $O(M \log N + S \log k)$.

Theorem 4.4.4. *FX-TM requires $O(M\,N\, + k)$ space.*

Proof. *Each interval tree uses $O(N)$ space [49]. There are M interval trees. Each scoremap uses $O(N)$ space as no object can appear in it twice. The tree set topscores uses $O(k)$ space. Thus the total space complexity is $O(M\,N)$ for subscription storage and $O(N + k)$ for matching for a total of $O(M\,N + N + k)$, i.e., $O(M\,N + k)$.*

4.5 Implementation

We outline the implementation of FX-TM in a distributed middleware system.

4.5.1 Local Implementation

A local controller has two input streams – one for subscriptions and one for events. The controller parses requests (add subscription, remove subscription, get top-k matches) and the raw data contained within. The controller processes the request by updating the local data, including updating attribute structures and tracking matches for budget window calculations, and returning the matches if applicable. The top-k algorithm component has its own API for managing subscriptions and issuing top-k matching requests and is interchangeable. Subscriptions and events support lists of an arbitrary number of discrete and interval attributes.

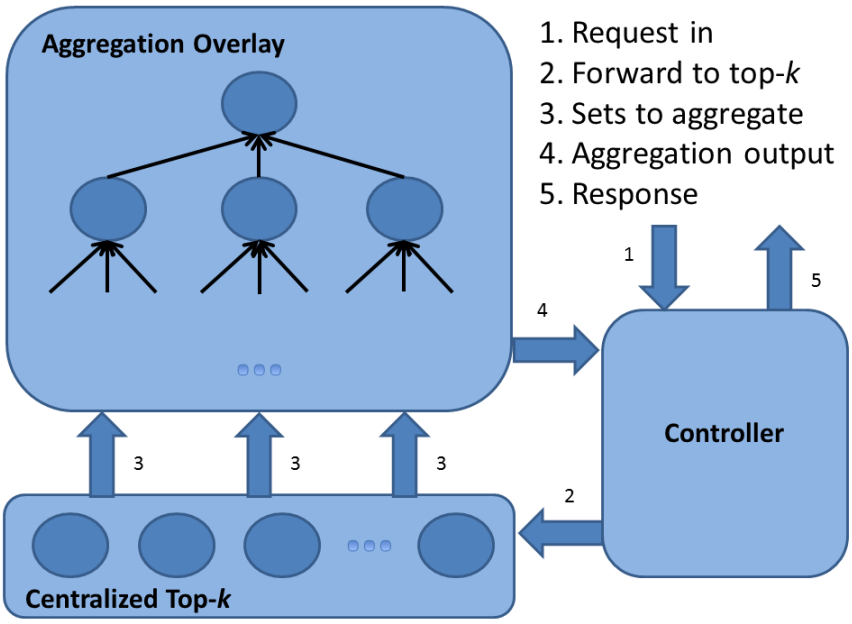


Figure 4.2.: Overview of the full distributed system.

4.5.2 Distributed Aggregation Overlay

There are two reasons distribution is desirable. Firstly, it increases parallel computation for lower system latency. Secondly, it allows scalability beyond the memory limitations of a single machine. We place the local top-k matching implementation on a set of nodes, then provide NOAH with a list of these nodes and a simple merge function which combines sets of top-k results from subsets of the data (see Figure 4.2). NOAH creates an aggregation hierarchy with a heuristically ideal fanout for minimal system latency based on the properties of the merging function. In this case of top-k the fanout is 3. The NOAH controller receives events for the system and forwards each event to every local controller to begin the matching process. The local controllers forward their results to the aggregation overlay, which returns the aggregate top-k results of the system. We use a simple script on the NOAH controller to distribute subscriptions evenly amongst nodes.

4.6 Evaluation

This section empirically compares the performance of FX-TM to the current state-of-the-art. FX-TM is at least as expressive as state-of-the-art top-k matching algorithms, so we emphasize *matching time*, which is the time taken to retrieve the top-k subscriptions that match an event. Since our expectation is that the majority of a system's workload will be consumed by the matching of events rather than subscription additions and removals, our algorithm has relatively low theoretical bounds on addition and removal procedures, and the state-of-the-art alternatives are tuned for matching times, we focus on the matching time for the empirical evaluation.

4.6.1 Systems Compared and Setup

We test FX-TM against three alternatives. First we present an implementation of Fagin's algorithm [12–14]. Summation, our targeted aggregation method, is mentioned in the literature [12, 14], but in the presence of both positive and negative weights, it is not monotonic and thus not be supported. In our experiments, Fagin's algorithm uses $max()$, which is well covered in Fagin's literature. For an additional performance gain we use interval trees instead of a database backend for faster retrieval of stored constraints which overlap the event intervals. Matching intervals are retrieved from the trees and sorted in order of weight$\times$prorated value, and Fagin's algorithm is applied to the resulting lists.

We also modify Fagin's algorithm to allow for summation with mixed weights without breaking monotonicity in an attempt to add greater expressiveness. The magnitude of most negative weight for each attribute is tracked. When an attribute is matched, all scores add that magnitude, including subscriptions which are not matched and have a natural score of 0. Thus no score is below 0, but the list for each contains all subscriptionsand must be sorted. The time to retrieve matches and sort the lists is presented as a lower bound on the execution time without actually matching as this time is already significantly longer than the other implementations.

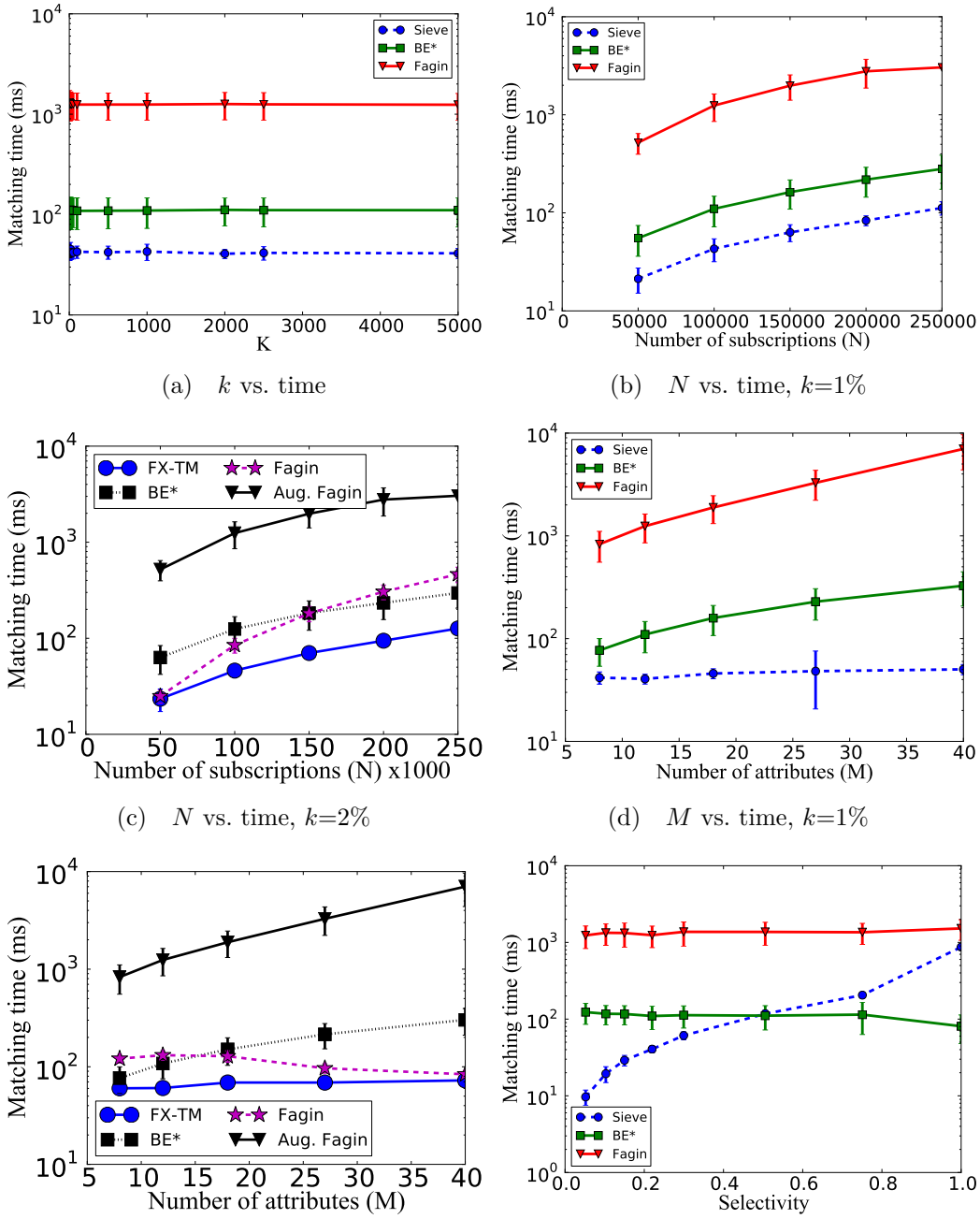
Finally, we implement a BE* tree structure [41]. Rather than dynamically maintaining the structure as new subscriptions are added, we add all subscriptions to a temporary structure and then build the tree for all subscriptions. This creates a tree structure that has the desired attributes of the BE* tree for a static set of subscriptions. The resulting tree is ideal for lookup purposes. (Additions and removals after the initial setup, which we do not consider, require a complete rebuild of the tree.) In addition to the subtrees in a node for intervals which are left, right, and overlapping the partition value, we also have a subtree for subscriptions which do not include the partitioning attribute. When matching on intervals we prorate scores from subtrees with the largest possible intersecting interval in each subtree.

All algorithms are implemented in Java using a single thread. The distributed data access prior to aggregation assumed by Fagin [12] is easily translated into multithreading that is also applicable to FX-TM with an appropriate locking scheme for concurrent updates and matches. [41] does not detail strategies for applying multithreading, so single threaded sequential execution is the fairest comparison.

All centralized evaluations are executed on a machine with an Intel T6600 2.2Ghz processor and 4 GB of RAM. Each algorithm uses the same set of subscriptions and events for an experiment. Unless noted otherwise, averages and standard deviations are calculated from 1000 distinct matches.

Table 4.2.: The default values for the experiments.

Variable	Generated Data	IMDB	Yahoo!
N	100000	100000	10000
k	1% of N 2% of N	1% of N 2% of N	1% of N 2% of N
M	12 per subscription or event out of 100 available	3 out of 3	5.4 (average) out of 22202
S/N	.22	.14	.11



(a) k vs. time

(b) N vs. time, k=1%

(c) N vs. time, k=2%

(d) M vs. time, k=1%

(e) M vs. time, k=2%

(f) Selectivity (S/N) vs. time

Figure 4.3.: Results from varying each variable in the micro-benchmarks.

4.6.2 Micro-Benchmark Description

The variables which affect matching time of events are (1) the number of subscriptions (N), (2) k, (3) the number of attributes per event (M), and (4) the selectivity of the events for the subscriptions present. Selectivity is the fraction of subscriptions where the constraint on at least one attribute matches an event, i.e., S/N. We generate test data to record the effects of the variables independently. Default values, shown in Table 4.2, are reasonable but low enough that the effect of altering each variable is significant. S/N varies per event; the value shown is the average for the experiment.

The generated data contains positive and negative weights, as well as intervals which may overlap to either side for proration. This implements the full expressiveness achievable by FX-TM and a modified BE*. The standard Fagin algorithm uses a different aggregation method to allow it to run on the same data, so it returns a different set of top-k results. This is the only viable way to compare *performance* given the inherent inability to match the expressiveness.

4.6.3 Micro-Benchmark Results

The results for the micro-benchmarks are shown in Figure 4.3. Figure 4.3a shows how k affects the algorithms. FX-TM scales very well, which is expected given the $log\, k$ term in our theoretical bound. The BE* tree also scales well with k, which is indicative of a similar approach for holding intermediate results in a heap and not tailoring other parts of the lookup process to k. Compared to FX-TM, the execution time is still increased by 165% to 200%.

As k impacts the amount of data sorted during Fagin's algorithm, k is expected to have a more dramatic affect on its runtime. Initially, that is the case as the execution time goes from 23% longer than FX-TM as k grows from 1% of N to 120% longer when k is 10% of N. The trend reverses slightly when k is increased further. This is because the number of matches per attribute hits a ceiling, limiting the length of the results to be sorted. The time for the augmented Fagin algorithm hits no such ceiling.

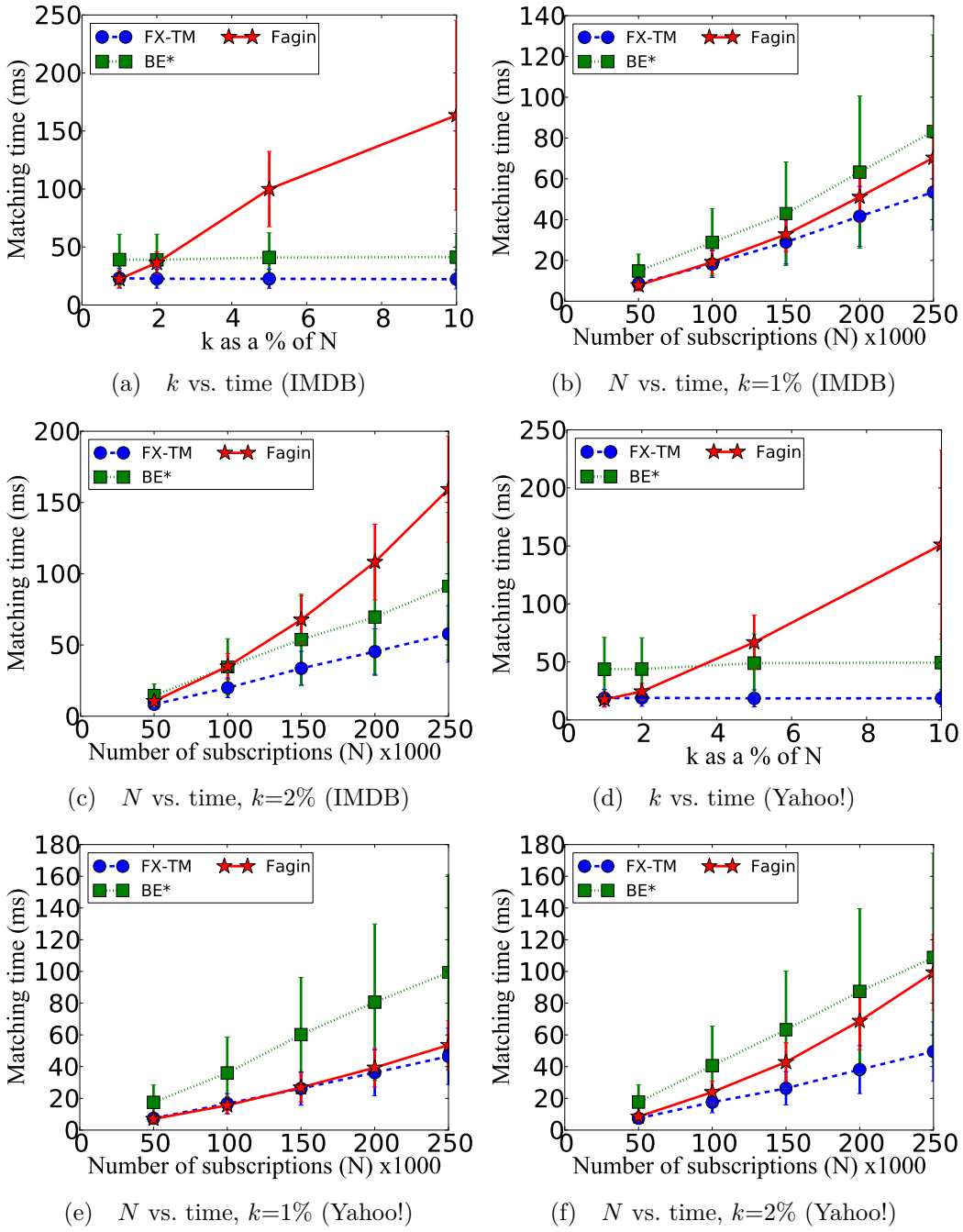
As long as a single negative score exists in an attribute, the retrieval mechanism sorts all subscriptions for that attribute. The graph in Figure 4.3a shows this happens regardless of k, resulting in very slow executions times which are never within 2500% of FX-TM.

The effects of N on the algorithms are shown in Figures 4.3b and 4.3c. Despite the logarithmic term in the theoretical bound for FX-TM the results look more linear. This is because to keep selectivity (S/N) fixed, S increases as well, which affects execution time. The performance of the BE* tree is slightly less sensitive to N and is 190% slower than FX-TM for small values of N and steadily improves to 137% longer for the largest value when k is 1% of N. When k is 2% of N, those numbers are 169% and 134% respectively.

Fagin's algorithm suffers worse as N increases. Higher values of N result in higher values of k, increasing the sorting time. The approach beats FX-TM by a fraction of a percent with low N and k= 1% of N. Increasing N to our maximum, though, results inversely in a 100% longer execution time than FX-TM. When k=2% of N, a 6% longer execution time than FX-TM grows to 167% longer when N is large.

Figures 4.3d and 4.3e show how M affects the algorithms. There is almost no perceivable difference in the execution time of FX-TM, which indicates the low impact of M relative to the other variables. The effect on BE* is more noticeable as it takes 25% to 300% additional execution time compared to FX-TM for both values of k. This suggests that the effectiveness of pruning with single dimensions on a multidimensional index loses its potency when more dimensions vie with equal weight in a partial matching problem.

Figure 4.3e contains an anomaly for the execution time of Fagin's algorithm similar to that seen in Figure 4.3a. Execution time decreases as M increases, bringing the execution time from a time accrued by 100% with respect to FX-TM for a low value of M to within 15% . This is counter to intuition and the trend seen in Figure 4.3d. It is because the number of matches available for each attribute max out below k due to the nature of the data and the value of k. The problem is exacerbated as M grows,



(a) k vs. time (IMDB)

(b) N vs. time, k=1% (IMDB)

(c) N vs. time, k=2% (IMDB)

(d) k vs. time (Yahoo!)

(e) N vs. time, k=1% (Yahoo!)

(f) N vs. time, k=2% (Yahoo!)

Figure 4.4.: Results from IMDB and Yahoo! real-world data benchmarks.

resulting in faster sorting times on smaller lists for each attribute. This happens for data with insufficiently high S/N as M or k grows.

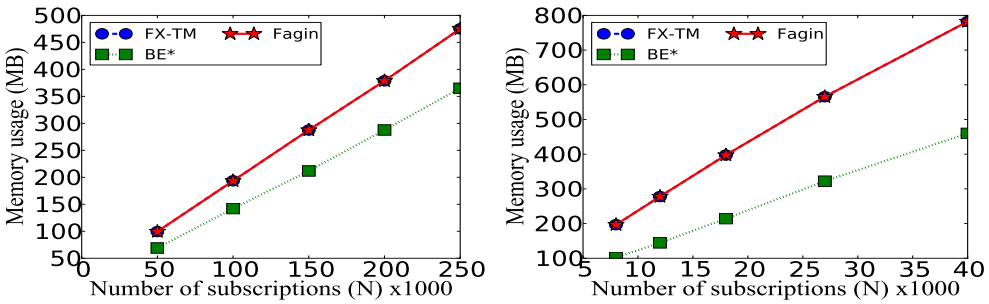
The difference between Fagin's algorithm and an attempt to augment its expressiveness is clear. The performance of the original algorithm appears correlated to that of FX-TM, again indicating that data retrieval time is the bottleneck. The increase of execution time with M is virtually nonexistent. Yet, an increase in M, corresponding to an increase in the number of times each subscription is added to a list and sorted, results in a superlinear increase in execution time of the augmented Fagin algorithm.

Finally, Figure 4.3f shows how selectivity affects execution times. Decreasing S/N, which corresponds to decreasing S when N is fixed, shows an appreciable effect on the execution time of FX-TM. The BE* tree's multi-attribute index and clustering structure allows it to scale with selectivity better, but does not do as well for smaller values of S/N, which we believe are more representative for top-k. For very low S/N, BE* adds just over 1000% execution time to that of FX-TM, but it is 91% faster when S/N approaches 1. With our default values, the trade off point in performance between the algorithms is when S/N is about .5.

As with the other variables affecting mainly data retrieval times, the execution time of Fagin's algorithm is correlated to the execution of FX-TM. Fagin's algorithm does take more than 20% longer with values of S/N between .15 and .22, but at both extremes of the range for S/N the approaches are within 5% of each other. The time taken by the augmented variant of Fagin's algorithm is included in the graph for comparison, but there are no real trends. The existence of a single negative weight on an attribute requires every subscription to be included in that attribute's list prior to sorting, leading to an effective S/N value of 1.0 regardless of the actual selectivity.

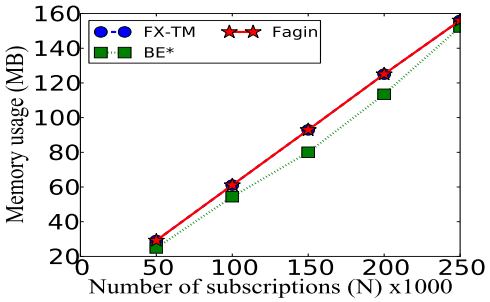
4.6.4 Real-World Data Description

To confirm the relevance of our micro-benchmark results, we use two real-world datasets to generate subscriptions and events for more experiments. The first such dataset uses movie rating data from IMDB [51]. For each movie, IMDB provides the number of users who rated it and the average rating. We build small intervals around

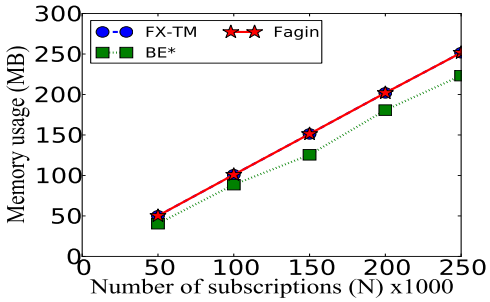


(a) N vs. RAM for subscriptions

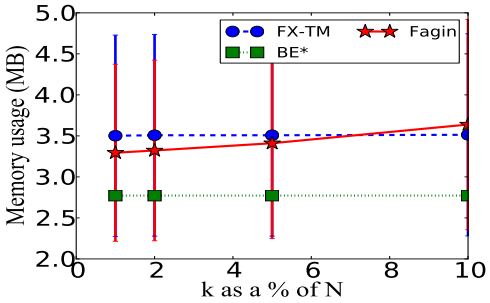
(b) M vs. RAM for subscriptions



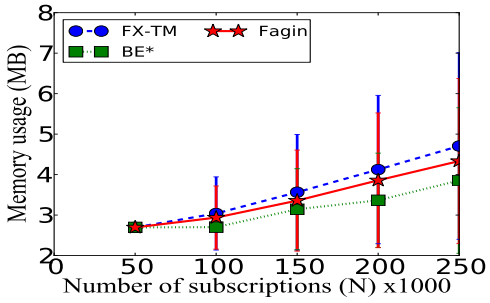
(c) N vs. RAM for subscriptions (IMDB)



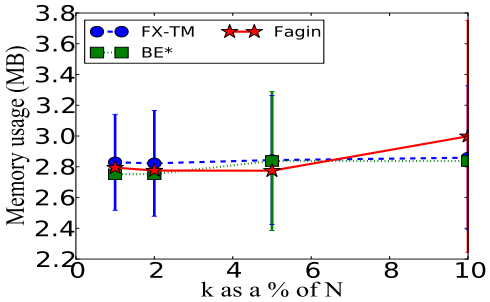
(d) N vs. RAM for subs. (Yahoo!)



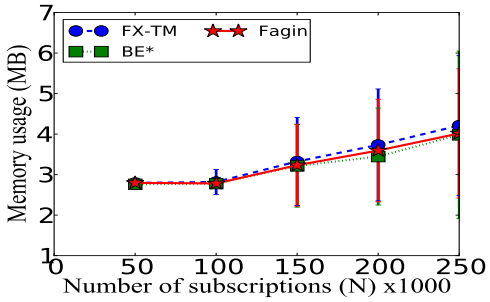
(e) k vs. RAM for matching (IMDB)



(f) N vs. RAM for matching (IMDB)



(g) k vs. RAM for matching (Yahoo!)



(h) N vs. RAM for matching (Yahoo!)

Figure 4.5.: Memory (RAM) usage used for storing subscriptions and for matching.

these values. The year of release is also provided. Thus all subscriptions and events have the same attributes. Subscriptions and events are generated the same way from different sections of the data. The best matches are subscriptions with similar voting patterns to an event and are released in the same year.

Our second real-world dataset is derived from the Yahoo! music rating service [52]. We use the same technique as in the IMDB dataset to build intervals around the number of voters and the average rating. Many songs also have anonymized genre and artist identifiers. These are discrete values. The best matches are subscriptions with similar voting patterns, matching genres, and the same artist as an event.

For each of these datasets, N and k are set to default values and varied as in the micro-benchmarks, but M and selectivity are features of the data. The values for each of the variables, including the defaults for N and k, are in Table 4.2.

4.6.5 Real-World Timing Results

The trends using the real-world data, shown in Figure 4.4, mirror the trends in the micro-benchmarks. We omit the modified Fagin variant so the differences among the other algorithms is clearer. For the IMDB data, the BE* tree is never less than 48% slower than FX-TM in Figure 4.4b. It is as much as 86% slower in Figure 4.4a. This is a smaller difference than seen in the micro-benchmarks because M's value of 3 is smaller than the default value of 12 for the micro-benchmarks, and BE* is sensitive to M.

Fagin's algorithm is 10% faster than FX-TM when N and k are very low in Figure 4.4b. When N or k is increased, that edge disappears. For our highest value of N, Fagin's algorithm takes 31% longer than FX-TM. Varying k in Figure 4.4a results in a quadratic change in execution time for Fagin's algorithm, indicating that the limit on the per-attribute list size seen in the micro-benchmark data is not seen here. When k is $N/10$ in Figure 4.4a, the execution time for Fagin's algorithm is 636%

longer than that of FX-TM. The general trends for N and k on all approaches are similar to those observed in the micro-benchmarks.

The Yahoo! dataset results also corroborate the micro-benchmarking results. Given that the value of M is in between the IMDB dataset and the lowest seen in the micro-benchmarks experiments, it is a little surprising that the execution time of the BE* tree is never less than 115% longer than FX-TM under the same conditions. This best case scenario is shown in Figure 4.4e when both N and k are relatively low. As those variables increase, the performance difference widens to a maximum of 164% in Figure 4.4d. Fagin's algorithm is almost 7% faster than FX-TM for the lowest value of N tested in Figure 4.4e, but adds 100% latency for our highest value of N in Figure 4.4f and more than 700% latency for the maximum value of k in Figure 4.4d.

4.6.6 Memory Usage Results

In emphasizing the execution time, we expect to incur some overhead in memory requirements. We rerun two micro-benchmark tests and the real-world data tests to measure the memory used. We look at the memory used for storing subscriptions and the space used by the matching algorithm per event. We first consider N, which affects the number of subscriptions stored and the tree indexing requirements. We also consider M, which affects the size of subscriptions and requires additional entries in attribute structures.

Memory use depends upon implementation of the data structures, so we realize that the exact values could vary widely. It is not advisable to draw conclusions about the direct comparisons of memory usage among algorithms, but the trends with relation to the variables and general orders of magnitude should be similar between implementations.

To find the amount of memory required to store subscriptions, we first initiate garbage collection and read the memory used by the Java runtime. We populate the structures for each algorithm and manually initiate garbage collection before taking

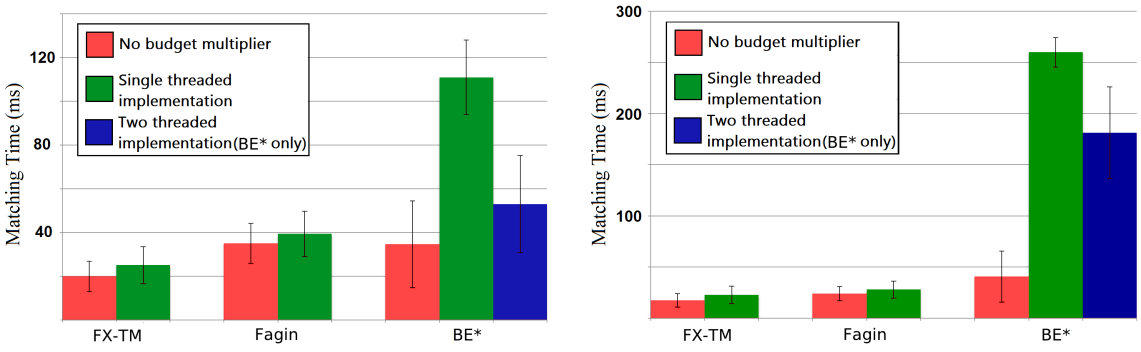
another reading. The difference between the two readings is the memory used to store the subscriptions.

Figure 4.5a shows N versus the memory requirements to store subscriptions. Each subscription requires roughly the same amount of memory independent of the indexing used. The number of nodes in a tree is linear to the number of leaf nodes. Thus the amount of memory used by all approaches is linear on N. A BE* tree uses less space than multiple interval trees due to it's only using the most relevant indexes at each level instead of indexing every attribute of every subscription. The memory required for storing subscriptions is the same for FX-TM and Fagin's algorithm, as shown in Figures 4.5a–4.5d, because we use equivalent structures.

Figure 4.5b shows the effect of varying M on the amount of space required to store subscriptions. As each subscription stores its preferences, the size of subscriptions increases linearly with respect to M. Each new attribute also requires additional space in the indexing trees. In the case of interval trees which index each attribute, that can be expected to also have a linear relationship, which is supported by the data. In the case of BE* trees, the relationship is also linear.

Figures 4.5c and 4.5d show the impact of changing N on the storage space required for the data structures holding the real-world data. Both the IMDB and Yahoo! datasets show linear changes in storage space with respect to N, affirming the results from the micro-benchmark data. The Yahoo! dataset, with its higher value of M, requires more space than the IMDB dataset. This affirms the correlation of M and storage space seen in the micro-benchmarking data.

Matching requires additional memory. The amount depending on the event, we obtain an average reading for 500 different matches. We collect garbage between each match, and check our results to ensure that automatic garbage collection does not impact the numbers. The data collected is memory in use by the Java virtual machine beyond storing the subscriptions, which includes memory used to match including function calls and temporary variables.



(a) IMDB dataset (b) Yahoo! dataset

Figure 4.6.: Overhead of the budget window mechanism.

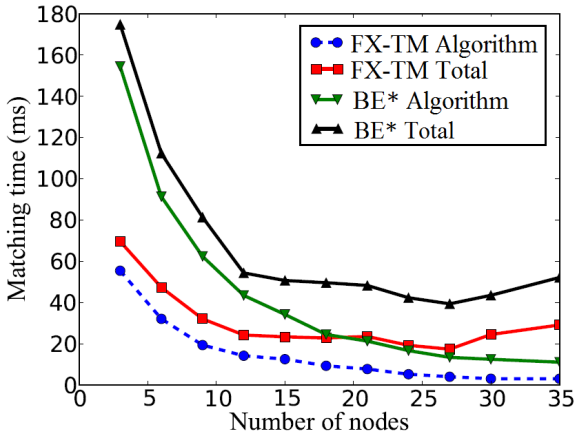


Figure 4.7.: Top-k results with NOAH.

Intuitively, k, which determines the heap size in the approaches using heaps to maintain the top-k already found, should affect this memory usage. However, Figures 4.5e and 4.5g show this not to be the case. k has almost no discernible effect on the amount of space the matching took in either the Yahoo! or IMDB datasets. This is because the size of the heap itself is relatively inconsequential to the overhead incurred by the Java method calls and object creations.

The other variable expected to increase the space required for matching is S, which impacts the size of *scoremap* in FX-TM. For the real-world datasets with fixed S/N, that means increasing N. The results are in Figures 4.5f and 4.5h. Increasing N from

50000 to 250000 results in an increase in memory usage of 75% FX-TM in the IMDB dataset and a 51% in the Yahoo! dataset. The other approaches also require more memory as N increases, but the increases are less significant. In none of these tests does the memory use exceed 5MB, which is at least an order of magnitude less than the space required to store subscriptions.

4.6.7 Budget Window Results

The next experiments test the impact of the budget window mechanism on matching time. To that end, each subscription is added a time window of $[1000000, 10000000]$ units and a budget of $[10000, 100000]$ matches. Every $g(t)$ is set to 1, making $\int g(t)\,dt = t - t_0$ for a homogeneous matching of the subscription across its time window. A time unit is the time taken by a single iteration of the matching algorithm.

To add the functionality to Fagin's algorithm, the multiplier is calculated in the same way as in FX-TM for each attribute before sorting. In both FX-TM and Fagin's algorithm, updates to the amount spent are made between matching iterations to maintain the single threading.

Implementing the mechanism in BE* is more cumbersome. The minimum and maximum multipliers must be propagated up the tree to inform pruning decisions. Since the multipliers change with time, they must be calculated and propagated continuously. We did this calculation and propagation in between each match. Since this process is clearly time intensive, we also moved the mechanism to a separate continuously running thread for a less onerous comparison to the other algorithms. In that case, pruning uses the current information at each level, which may be inconsistent with the state of the subscriptions further down.

For simplicity, Figure 4.6 shows only the results of implementing the budget window mechanism in the real-world data sets using the default N value of 100000 and k at 2%. The first (red) bar in each group is the time taken by each algorithm without the budget window mechanism. The second (green) bar is the time taken

when the mechanism is implemented within the same thread. For the BE* tree, the third (blue) bar is for the multithreaded approach.

The additional time taken for the budget window mechanism in the IMDB dataset is shown in Figure 4.6a. FX-TM and Fagin's algorithm require an additional 25.6% and 12.5% respectively, with the overall time taken for FX-TM still being less than that taken by Fagin's algorithm. The time required with this feature incorporated into the BE* tree is significantly higher. Running completely on a single thread, the BE* tree leads to a 220% time increase. Running the update thread in parallel reduces this overhead to 53.1%, which is still significant compared to the other two approaches running in single threads.

The results with the Yahoo! dataset reiterate the previous results with more accentuated numbers. In this case, FX-TM requires a time increase of 29.7% over not implementing the budget window, and Fagin's algorithm requires 16.6% longer. The total time taken for FX-TM is still less overall than Fagin's algorithm. The BE* tree approach requires 540% longer when updating in the same thread and 346% longer when using a separate thread. This additional time indicates that the pruning mechanism during matching depends heavily on the data, and implementing the budget window mechanism can have a very large impact on the pruning.

4.6.8 Distributed Setup Results

With our JVM and machines, we are not able to run much more than 200000 subscriptions fitting the defaults from the generated microbenchmarks before running out of memory, which is insufficient for our target applications.

We evaluated the distributed setup on a group of blade servers at an IBM research center. We generated 500000 subscriptions using the defaults from the microbenchmarks. We evenly distributed the subscriptions across different numbers of leaves. Increased distribution decreases the load per leaf at the expense of greater aggregation. We tested each setup with 100 events and show the average times in

Figure 4.7. Because we are trying to emulate a real system with full expressiveness, we only consider the two approaches which can handle non-monotonic score, i.e. BE* trees and FX-TM. For each algorithm we show two lines – the average amount of time taken for the local system at the leaves and the total amount of time from when an event enters the system to when the top-k results from the system are available.

Distributing the data decreases the effective N value at individual nodes, and the time for local computation decreases. There is a point of diminishing returns as the relative addition of each new node is less significant, and aggregation time increases with the addition of more nodes. This is particularly noticeable as the number of nodes passes a threshold of a power of 3, which increases the height of the aggregation hierarchy. It is worth noting that while the aggregation time is generally the same between the two algorithms because the outputs from the local nodes are equivalent, it is slightly higher for BE* trees. This is because the variability of the local latency is higher, and the aggregation hierarchy has to receive all results to complete aggregation, so the maximum local computation time is as important as the average time.

For both algorithms the minimal total system time occurs when the data is distributed across 27 nodes. At this point the decrease in local computation time becomes less significant than the increased aggregation time for our data. At this point, the BE* tree takes 330% as long as FX-TM at the local nodes and 226% as long for the entire system. We note the total system time at 27 nodes is less than the local matching time when there were fewer than 12 nodes, so distribution effectively decreases total matching latency.

5 PRIOR ART

5.1 Big Data Aggregation

Several frameworks consider breaking down big data computations into independently optimized phases. MapReduce [6] has become very prevalent, and is the basis for much work and research. Incoop [53] adds a contraction phase between map and reduce in the MapReduce framework by exploiting the Combiners to run aggregation. This contraction phase does not consider the efficiency of different fan-ins for combination. Yu et al. [54] extend MapReduce to efficiently aggregate data between job phases with similar goals to Incoop. These approaches show the usefulness of an aggregation subsystem, but do not optimize the interaction topology of such a mechanism. MapReduceMerge [55] extends the framework with a merge step – a synonym for aggregation – after the reduce phase. MapReduceMerge is motivated by many operations relevant to applications using MapReduce, e.g. full sorts, joins, set unions, and cartesian products, which are not supported efficiently. MapReduce Online [56] allows processing at the reducers to begin before all data is received, thus allowing it to be used for stream processing or more intuitive iterative processes.

CamCube is a network architecture to bypass traditional routing by orienting servers [57] with a coordinate system when full control of the environment is possible. Nodes communicate with up to six neighbors directly, limiting the maximum achievable fan-in. Camdoop shows the ability for CamCube to work efficiently with the existing MapReduce formulation [58]. Camdoop aggregates within the implementation instead of on the edges of the calculation as in some other MapReduce implementations, but is limited by the six neighbor constraint.

Morozov and Weber [22] consider *merge trees* for distributed computations, an abstraction for combining subsets of large structured datasets. Their system monitors

data traits in different branches and recomputes a better tree. It does not optimize for aggregation functions or extrapolate to find optima.

While not specific to aggregation, Kim et al. [59] extend the work by Cheng and Robertazzi [60] to optimize load distribution on processors connected by a tree network. The newer work maximizes parallelization for fastest completion because there is no computation to aggregate results from each processor. Kuhn and Oshman do consider the complexity of aggregation in networks [61], but they concentrate on the number of rounds of communication required in directed networks. Drucker et al. consider the complexity required to split a task across several nodes in the case the results do not have to be aggregated [62].

Work with sensors optimizes overlays for power consumption while conforming to the routing restrictions imposed by the location and communication capabilities of sensors [63, 64]. TinyDB [65] furthers this in determining when to sample, which is equivalent to local computation.

Naiad [66] considers aggregation as part of iterative or cyclic computations. It allows distributed data access and updates to be interleaved, and is not aggregation-specific. Like resilient distributed datasets put forth by Zaharia et al. for Spark [30], Naiad relies on data retained in memory. This offers latencies which can be orders of magnitude better than with disk accesses. This addresses the problem raised by Venkataraman et al. [28] that data access is a significant bottleneck for iterative calculations on many distributed frameworks.

A line of research that considers an overlay specific to aggregation is Astrolabe [21]. Astrolabe is introduced as a summarizing mechanism of bounded size using a hierarchical nature. The aggregation is meant to monitor system state and facilitate scalability of dynamic systems. Summarizing calculations are made and aggregated on the fly using gossip protocols. To a limited fashion they consider the effect the overlay has on the aggregation, but it is not geared toward minimizing the total time of the system as a separate phase involved in computation.

SDIMS [67] presents a rigid overlay for a distributed system used primarily for aggregation. Shruti [68] adds flexibility to the structure by allowing the actual aggregation mechanisms to be tuned across this structure without altering the overlay. The overlay is hierarchical with known fan-in. It is built on top of a distributed hash table for reasons including fault tolerance rather than finding an optimal overlay for total system latency. STAR [20] extends this line of work by using a consistency metric [69] to adaptively set the precision constraints for processing aggregation.

PIER [70] is another system built on DHTs to distribute workload, this time for database use. The overlays are once again restricted by the underlying framework, but the system itself efficiently aggregates results to respond to queries.

5.2 Top-k Matching

5.2.1 Top-k Matching Algorithms

Fagin proposes the seminal algorithm [12] for top-k matching. It runs on databases, so querying sorted lists for individual attributes is easy, and it uses these sorted lists and an aggregation function to create an aggregate order. Matching takes $O(N^{(M-1)/M}\, k^{1/M})$ time *starting from the point the sorted lists are available* – N is the number of objects in the system (our subscriptions) and M the number of attributes.

Fagin and Wimmers [13] expand this to incorporate weights that can vary per attribute without affecting the running time. In line with the database querying model, these weights are determined by the query (corresponding to our events) rather than the objects in the database (our subscriptions).

Machanavajjhala et al. [40] at Yahoo! attempt the first top-k system without a database for the specific use of targeted ads. Their work shows such an approach can efficiently find the attribute level results when attributes can be stored in sorted order, and those results used in Fagin's algorithm.

Sadoghi and Jacobsen also approach the problem from a storage perspective instead of focusing primarily on aggregation [41, 71]. Their BE* tree uses an alternating clustering and dimension partitioning strategy. They choose the partitioning such that the most divergent dimensions are used first to prune the search space quickly. They show that this approach is experimentally competitive to a multi-dimensional storage tree on generated and real-world data.

Recent research into top-k matching focusses on performing rankings with uncertain data. Dylla et al. [72] and Song et al. [73] consider approaches to display the k best suspected matches when some of the values in the data are unknown or known only with some probability. Both methods require some additional overhead to handle this extra requirement.

Search engines and ad companies have their own approaches, but these are proprietary and unavailable to the public.

5.2.2 Distributed Top-k Matching Systems

Several researchers consider *distributing* the top-k matching problem. Most proposals rely on running a matching algorithm on nodes on *partitions* of the data, then aggregating the partial results in some way. Improving the core top-k matching algorithm thus improves the distributed system.

Fagin mentions that the data can be housed on multiple systems, but does require all of the accesses to be done from a single computer to run the algorithm. Cao and Wang [29] distribute Fagin's algorithm by running it at multiple nodes, and then combining the partial results at a single node. The process involves running the algorithm and pruning before sending the partial results to a designated node for merging. The paper focuses on the best way to do this while conserving bandwidth. Fagin proposes a similar threshold algorithm [14]; that paper includes extensive proofs on how well similar algorithms perform under various scenarios.

These approaches rely on in part on Fagin's algorithm. Machanavajjhala et al. [40] theorize about an alternative possibility of using a publish/subscribe system to find matches, presumably similar to what is described by Aguilera et al. [74] with more robust matching, but claim it is inefficient and do not implement it. What they do implement uses Fagin's algorithm. Thus any improvements to a local algorithm are immediately pertinent to distributed top-k matching.

5.2.3 Limitations

Efforts based on Fagin's work exhibit several limitations. Among them is assuming the time for ordered access on attribute matching can be ignored. With proration and dynamic multipliers, scores cannot be known ahead of time, so subscriptions cannot be stored in sorted order, and sorting is run during retrieval. This is required for each of attribute, which takes $O(MS\,logS)$ time for S matching subscriptions per M event attributes prior to running the algorithm.

One requirement of Fagin's algorithm propagated to other work is the explicit need for *monotonicity* in the scoring mechanism [12]: component scores from attributes are considered sequentially, and the aggregate score is required to either exlusively increase or decrease (or stay the same) as the algorithm runs. This works well for some functions, e.g. minimum and maximum subscore. However, even an aggregation function as simple as summation is not monotonic when considering the possibility of positive *and* negative weights. For example, consider aggregating the score over a match with component scores {.2, .2, -.1}. After the initial component, the score is .2. With the next component it *increases* to .4 and then *decreases* to .3. As long as two component scores aside from the first have opposite signs, monotonicity is not guaranteed. Consider a political campaign trying to serve ads. The campaign would prefer to serve ads to users with some attributes (gender, income, etc.). Those below the voting age should be avoided, corresponding to a negative weight for this campaign.

Fagin and Wimmers [13] present a device for weighting attributes in a match, but weighting can only be on what we call an event. In the case of advertising, the ad consumer has a smaller stake in the match than the advertiser. Thus it makes more sense to allow the advertisers to specify weights independent of each other as they will value different attributes for targeting demographics. Advertisers are the objects in the system, the subscriptions, and the ad serving is a result of the events; this example of weighting cannot be done with events as proposed by Fagin and Wimmers.

BE* trees [41] are easily modified to overcome these limitations despite being best tuned to events with attributes containing single values. However, as we will demonstrate empirically, BE* trees are not always the fastest approach.

None of this research considers dynamic score adjustment based on system behavior.

6 CONCLUSIONS

6.1 Big Data Aggregation

Rigourously defining classes of distributed data processing is the first step in creating a portfolio of tools. With a broader range of tools users can select the right abstraction to perform analysis efficiently without giving up intuitive data models or requiring reimplementation of pieces of distributed frameworks that are used across problems.

We define the compute-aggregate family of problems and identify universal characteristics of compute-aggregate tasks that allow to unify design principals of “ideal" aggregation trees so in depth analysis of the aggregation function is not required to determine the aggregation overlay.

We rigorously prove optimality of aggregation trees for different cases of compute-aggregate systems. We clearly state our assumptions for modeling and the conditions we impose. For processing a single block of input we explain that the appropriate measure is latency and that the ratio of output size to input size affects the optimal fan-in.

After the optimal overlays are proven for a slightly simplified models we create heuristics that vary in practical ways and implement them in a new system called NOAH. The heuristics rely on a single variable y – the ratio of the aggregation output to one input – to choose the fan-in to use for an overlay.

We experimentally show these heuristics to be nearly, if not actually, optimal for a variety of microbenchmarks and real world problems in Amazon EC2 without synthetically forcing the assumptions we used to build the model. The overlays outperform naïve one-size-fits-all overlays, such as those currently used in practice, by up to 75%.

We introduce a balancing mechanism to improve performance when the heuristics choose overlays with skewed data distribution. We also introduce two other mechanisms to exploit prior aggregation to reduce the amount of work done or resource inefficiencies caused by maintaining unnecessary parallelism in subsequent aggregation. These mechanisms reduce latency in persistent overlays by up to 80% over reusing overlays without modification.

The latency savings are significant both as part of a larger problem and when running iterative aggregations.

6.2 Top-k Matching

Expressive top-k matching with low latency represents a cornerstone for many "match-making" applications including online advertising, which has been driving a large economy based on online services. We introduce a new, more expressive, model for top-k matching which encompasses the models from prior art. It supports intervals in events and subscriptions with optional prorating for interval intersection. The model allows weights on events *or* subscriptions and supports non-monotonic aggregation.

We present FX-TM, an efficient and scalable algorithm for our model which runs in $O(M\,log\,N + S\,log\,k)$ time including data retrieval. This is experimentally competitive with the existing art, even without considering our extended expressiveness. Fagin's seminal algorithm is competitive for lower values of k as long as its limitations on expressiveness are acceptable. BE* trees, which can be modified to be as expressive as FX-TM, are shown by micro-benchmarks to be slower for selective data. For two real-world data sets BE* exhibited up to 2$\times$ higher matching time than FX-TM.

We believe that the extended expressiveness of our FX-TM and its performance make it competitive for relevant matching problems and systems, especially with regard to serving ads. Existing art does not match the performance of our algorithm while incorporating both the expressiveness *and* ability to dynamically change scores based on budgets.

We present a dynamic mechanism to adjust the matching score based on the rate of matching and a predetermined budget and time window. With dynamic score adjustment, BE* requires up to 11× higher matching time than FX-TM.

We implement distributed top-k matching with NOAH for improved performance and greater scalability. Together with the other work on NOAH this shows how the computation and aggregation phase can be optimized independently.

6.3 Future Work

There are still cases for which our fan-ins remain unproven because the degree of super- or sub-linearity of the aggregation function factors into the model. While the heuristics show good performance in these ranges it is worth further exploration to see if more precise heuristics can be created.

We are also working on optimizing the computation and aggregation in tandem for total optimization instead of considering aggregation in isolation, e.g. by calculating an optimal number of leaf nodes, and specific optimizations for iterative computation.

We are also considering real time pricing for top-k matching. The budget window multiplier does a good job of altering subscription scores relative to each other to prevent over- or under-matching relative to peers. However, there is still a problem of global supply and demand that might cause all subscriptions to get over- or under-matched together. By charging different prices for a match based on the demand for the given event we believe an equilibrium point could be acheived.

REFERENCES

REFERENCES

[1] Eric A. Brewer. Lessons from Giant-Scale Services. *Internet Computing, IEEE*, 5(4):46–55, July 2001.

[2] Jim Gray. Distributed Computing Economics. Technical Report MSR-TR-2003-24, Microsoft Research, March 2003.

[3] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI '08, pages 1–14, Berkeley, CA, USA, December 2008. USENIX Association.

[4] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig Latin: A Not-So-Foreign Language for Data Processing. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 1099–1110, New York, NY, USA, June 2008. ACM.

[5] Apache Software Foundation. Apache Hadoop. `http://hadoop.apache.org`.

[6] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1):107–113, January 2008.

[7] Samuel C. Kendall, Jim Waldo, Ann Wollrath, and Geoff Wyant. A Note on Distributed Computing. Technical report, Sun Microsystems, Inc., Mountain View, CA, USA, 1994.

[8] Amazon Web Services. Amazon Elastic Compute Cloud (EC2). `http://aws.amazon.com/ec2/`.

[9] Vicci. Vicci: A Programmable Cloud-computing Research Testbed. `http://www.vicci.org/`.

[10] IBM. IBM Cloud. `http://www.ibm.com/cloud-computing/us/en/`.

[11] Ali Khajeh-Hosseini, David Greenwood, James W. Smith, and Ian Sommerville. The Cloud Adoption Toolkit: Supporting Cloud Adoption Decisions in the Enterprise. *Software: Practice and Experience*, 42(4):447–465, April 2012.

[12] Ronald Fagin. Combining Fuzzy Information from Multiple Systems. In *Proceedings of the 15th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '96, pages 216–226, New York, NY, USA, June 1996. ACM.

[13] Ronald Fagin and Edward L. Wimmers. A Formula for Incorporating Weights into Scoring Rules. *Theoretical Computer Science*, 239(2):309 – 338, May 2000.

[14] Ronald Fagin, Amnon Lotem, and Moni Naor. Optimal Aggregation Algorithms for Middleware. In *Proceedings of the 20th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '01, pages 102–113, New York, NY, USA, June 2001. ACM.

[15] Amélie Marian, Nicolas Bruno, and Luis Gravano. Evaluating Top-k Queries over Web-Accessible Databases. *ACM Transactions on Database Systems*, 29(2):319–362, June 2004.

[16] Eugene Agichtein, Eric Brill, and Susan Dumais. Improving Web Search Ranking by Incorporating User Behavior Information. In *Proceedings of the 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '06, pages 19–26, New York, NY, USA, August 2006. ACM.

[17] Jeff Dean. Designs, Lessons and Advice from Building Large Distributed Systems, 2009. LADIS '09 Keynote Talk.

[18] Michael Stonebraker, Daniel Abadi, David J. DeWitt, Sam Madden, Erik Paulson, Andrew Pavlo, and Alexander Rasin. MapReduce and Parallel DBMSs: Friends or Foes? *Commununications of the ACM*, 53(1):64–71, January 2010.

[19] Donald Miner. Hadoop MapReduce Can Transform How You Build Top-Ten Lists. *Pivotal P.O.V.*, September 2012.

[20] Navendu Jain, Dmitry Kit, Prince Mahajan, Praveen Yalagandula, Mike Dahlin, and Yin Zhang. STAR: Self-tuning Aggregation for Scalable Monitoring. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, pages 962–973. VLDB Endowment, September 2007.

[21] Robbert Van Renesse, Kenneth P. Birman, and Werner Vogels. Astrolabe: A Robust and Scalable Technology for Distributed System Monitoring, Management, and Data Mining. *ACM Transactions on Computer Systems*, 21(2):164–206, May 2003.

[22] Dmitriy Morozov and Gunther Weber. Distributed Merge Trees. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 93–102, New York, NY, USA, February 2013. ACM.

[23] William Culhane, Kirill Kogan, Chamikara Jayalath, and Patrick Eugster. Optimal Communication Structures for Big Data Aggregation. In *Proceedings of the 34th Annual Joint Conference of the IEEE Computer and Communications Societies*, INFOCOM '15. IEEE, April 2015.

[24] Yanpei Chen, Archana Ganapathi, Rean Griffith, and Randy Katz. The Case for Evaluating MapReduce Performance Using Workload Suites. In *Proceedings of the 19th International Symposium on Modeling, Analysis Simulation of Computer and Telecommunication Systems*, MASCOTS '11, pages 390–399. IEEE, July 2011.

[25] Yan Zhang and Nirwan Ansari. On Architecture Design, Congestion Notification, TCP Incast and Power Consumption in Data Centers. *IEEE Communications Surveys and Tutorials*, 15(1):39–64, February 2013.

[26] Haitao Wu, Zhenqian Feng, Chuanxiong Guo, and Yongguang Zhang. ICTCP: Incast Congestion Control for TCP in Data-Center Networks. *IEEE/ACM Transactions on Networking*, 21(2):345–358, April 2013.

[27] William Culhane, Kirill Kogan, Chamikara Jayalath, and Patrick Eugster. LOOM: Optimal Aggregation Overlays for In-Memory Big Data Processing. In *Proceedings of the 6th USENIX Conference on Hot Topics in Cloud Computing*, HotCloud '14, Berkeley, CA, USA, June 2014. USENIX Association.

[28] Shivaram Venkataraman, Indrajit Roy, Alvin AuYoung, and Robert S. Schreiber. Using R for Iterative and Incremental Processing. In *Proceedings of the 4th USENIX Conference on Hot Topics in Cloud Computing*, HotCloud '12, Berkeley, CA, USA, June 2012. USENIX Association.

[29] Pei Cao and Zhe Wang. Efficient Top-k Query Calculation in Distributed Networks. In *Proceedings of the 23rd Annual ACM Symposium on Principles of Distributed Computing*, PODC '04, pages 206–215, New York, NY, USA, July 2004. ACM.

[30] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI '12, pages 2–2, Berkeley, CA, USA, April 2012. USENIX Association.

[31] Ganesh Ananthanarayanan, Ali Ghodsi, Andrew Wang, Dhruba Borthakur, Srikanth Kandula, Scott Shenker, and Ion Stoica. PACMan: Coordinated Memory Caching for Parallel Jobs. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI '12, pages 20–20, Berkeley, CA, USA, April 2012. USENIX Association.

[32] Apache Software Foundation. Apache Crunch. `http://spark.apache.org`.

[33] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, SOSP '03, pages 29–43, New York, NY, USA, October 2003. ACM.

[34] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop Distributed File System. In *Proceedings of the 26th Symposium on Mass Storage Systems and Technologies*, MSST '10, pages 1–10. IEEE, May 2010.

[35] Wei Huang, Matthew J. Koop, Qi Gao, and Dhabaleswar K. Panda. Virtual Machine Aware Communication Libraries for High Performance Computing. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, SC '07, pages 9:1–9:12, New York, NY, USA, November 2007. ACM.

[36] Yahoo Labs. Yahoo MapReduce Cluster Logs. `http://webscope.sandbox.yahoo.com`.

[37] James MacQueen. Some Methods for Classification and Analysis of Multivariate Observations. In *Proceedings of the 5th Berkeley Symposium on Mathematical Statistics and Probability*, pages 281–297. California, USA, June 1967.

[38] Manohar Kaul, Bin Yang, and Christian S. Jensen. Building Accurate 3D Spatial Networks to Enable Next Generation Intelligent Transportation Systems. In *Proceedings of the 14th International Conference on Mobile Data Management*, volume 1 of *MDM '13*, pages 137–146. IEEE, June 2013.

[39] William Culhane, K. R. Jayaram, and Patrick Eugster. Fast, Expressive Top-k Matching. In *Proceedings of the 15th International Middleware Conference*, Middleware '14, pages 73–84, New York, NY, USA, December 2014. ACM.

[40] Ashwin Machanavajjhala, Erik Vee, Minos Garofalakis, and Jayavel Shanmugasundaram. Scalable Ranked Publish/Subscribe. *Proceedings of the VLDB Endowment*, 1(1):451–462, August 2008.

[41] Mohammad Sadoghi and Hans-Arno Jacobsen. Relevance Matters: Capitalizing on Less (Top-k Matching in Publish/Subscribe). In *Proceedings of the 28th International Conference on Data Engineering*, ICDE '12, pages 786–797. IEEE, April 2012.

[42] Avi Goldfarb and Catherine E. Tucker. Online Advertising, Behavioral Targeting, and Privacy. *Communications of the ACM*, 54(5):25–27, May 2011.

[43] Jun Yan, Ning Liu, Gang Wang, Wen Zhang, Yun Jiang, and Zheng Chen. How Much Can Behavioral Targeting Help Online Advertising? In *Proceedings of the 18th International Conference on World Wide Web*, WWW '09, April 2009.

[44] Michael A Stelzner. Social Media Marketing Industry Report. Technical report, Social Media Examiner, `http://www.socialmediaexaminer.com/SocialMediaMarketingReport2011.pdf`, 2011.

[45] Maura McGowan. Facebook Rolling Out Video Ads to News Feeds Social Network Gives Brands Four Demographics to Target with 15-Second Spots. *Adweek*, May 2013.

[46] Mark Prigg. Dislike: Over HALF of Facebook Users Say they are Fed Up with Constant Adverts and Sponsored Posts. *Mail Online*, July 2012.

[47] Shu-Chuan Chu. Viral Advertising in Social Media: Participation in Facebook Groups and Responses among College-aged Users. *Journal of Interactive Advertising*, 12(1):30–43, July 2011.

[48] Anand Bhalgat, Jon Feldman, and Vahab Mirrokni. Online Allocation of Display Ads with Smooth Delivery. In *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '12, August 2012.

[49] Lars Arge and Jeffrey Scott Vitter. Optimal Dynamic Interval Management in External Memory. In *Proceedings of the 37th Annual Symposium on Foundations of Computer Science*, FOCS '96, pages 560–569, October 1996.

[50] T Cormen, R Rivest, C Leiserson, and C Stein. *Introduction to Algorithms*. MIT Press, 2011.

[51] IMDB. IMDB Movie Ratings. `http://www.imdb.com/interfaces`.

[52] Yahoo! C15 – Yahoo! Music User Ratings of Musical Tracks, Albums, Artists and Genres v 1.0. Yahoo! Webscope `http://webscope.sandbox.yahoo.com`.

[53] Pramod Bhatotia, Alexander Wieder, Rodrigo Rodrigues, Umut A. Acar, and Rafael Pasquin. Incoop: MapReduce for Incremental Computations. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SOCC '11, pages 7:1–7:14, New York, NY, USA, October 2011. ACM.

[54] Yuan Yu, Pradeep Kumar Gunda, and Michael Isard. Distributed Aggregation for Data-parallel Computing: Interfaces and Implementations. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, pages 247–260, New York, NY, USA, October 2009. ACM.

[55] Hung-chih Yang, Ali Dasdan, Ruey-Lung Hsiao, and D. Stott Parker. Map-reduce-merge: Simplified Relational Data Processing on Large Clusters. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, SIGMOD '07, pages 1029–1040, New York, NY, USA, June 2007. ACM.

[56] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M Hellerstein, Khaled Elmeleegy, and Russell Sears. MapReduce Online. In *7th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '10, April 2010.

[57] Hussam Abu-Libdeh, Paolo Costa, Antony Rowstron, Greg O'Shea, and Austin Donnelly. Symbiotic Routing in Future Data Centers. In *Proceedings of the ACM SIGCOMM 2010 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '10, pages 51–62, New York, NY, USA, October 2010. ACM.

[58] Paolo Costa, Austin Donnelly, Antony Rowstron, and Greg O'Shea. Camdoop: Exploiting In-Network Aggregation for Big Data Applications. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI '12, pages 3–3, Berkeley, CA, USA, April 2012. USENIX Association.

[59] Hyoung-Joong Kim, Gyu-In Jee, and Jang-Gyu Lee. Optimal Load Distribution for Tree Network Processors. *Aerospace and Electronic Systems, IEEE Transactions on*, 32(2):607–612, August 1996.

[60] Yuan-Chieh Cheng and Thomas G. Robertazzi. Distributed Computation for a Tree Network with Communication Delays. *Aerospace and Electronic Systems, IEEE Transactions on*, 26(3):511–516, May 1990.

[61] Fabian Kuhn and Rotem Oshman. The Complexity of Data Aggregation in Directed Networks. In David Peleg, editor, *Distributed Computing*, volume 6950 of *Lecture Notes in Computer Science*, pages 416–431. Springer Berlin Heidelberg, 2011.

[62] Andrew Drucker, Fabian Kuhn, and Rotem Oshman. The Communication Complexity of Distributed Task Allocation. In *Proceedings of the 31st ACM Symposium on Principles of Distributed Computing*, PODC '12, pages 67–76, New York, NY, USA, July 2012. ACM.

[63] Jae-Hwan Chang and Leandros Tassiulas. Energy Conserving Routing in Wireless Ad-Hoc Networks. In *Proceedings of the 19th Annual Joint Conference of the IEEE Computer and Communications Societies*, volume 1 of *INFOCOM '00*, pages 22–31 vol.1. IEEE, March 2000.

[64] Hüseyin Özgür Tan and Ibrahim Körpeoğlu. Power Efficient Data Gathering and Aggregation in Wireless Sensor Networks. *ACM SIGMOD Record*, 32(4):66–71, December 2003.

[65] Samuel R. Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. TinyDB: An Acquisitional Query Processing System for Sensor Networks. *ACM Transactions on Database Systems*, 30(1):122–173, March 2005.

[66] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: A Timely Dataflow System. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, SOSP '13, pages 439–455, New York, NY, USA, November 2013. ACM.

[67] Praveen Yalagandula and Michael Dahlin. SDIMS: A Scalable Distributed Information Management System. In *Proceedings of the ACM SIGCOMM 2004 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '04, pages 379–390, August 2004.

[68] Praveen Yalagandula and Mike Dahlin. Shruti: A Self-Tuning Hierarchical Aggregation System. In *Proceedings of the 1st International Conference on Self-Adaptive and Self-Organizing Systems*, SASO '07, pages 141–150. IEEE, July 2007.

[69] Navendu Jain, Prince Mahajan, Dmitry Kit, Praveen Yalagandula, Michael Dahlin, and Yin Zhang. Network Imprecision: A New Consistency Metric for Scalable Monitoring. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI '08, pages 87–102, Berkeley, CA, USA, December 2008. USENIX Association.

[70] Ryan Huebsch, Joseph M. Hellerstein, Nick Lanham, Boon Thau Loo, Scott Shenker, and Ion Stoica. Querying the Internet with PIER. In *Proceedings of the 29th International Conference on Very Large Data Bases*, volume 29 of *VLDB '03*, pages 321–332. VLDB Endowment, September 2003.

[71] Mohammad Sadoghi and Hans-Arno Jacobsen. BE-tree: An Index Structure to Efficiently Match Boolean Expressions over High-dimensional Discrete Space. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, pages 637–648, New York, NY, USA, June 2011. ACM.

[72] Maximilian Dylla, Iris Miliaraki, and Martin Theobald. Top-k Query Processing in Probabilistic Databases with Non-Materialized Views. In *Proceedings of the 29th International Conference on Data Engineering*, ICDE '13, pages 122–133. IEEE, April 2013.

[73] Chunyao Song, Zheng Li, and Tingjian Ge. Top-K Oracle: A New Way to Present Top-k Tuples for Uncertain Data. In *Proceedings of the 29th International Conference on Data Engineering*, ICDE '13, pages 146–157. IEEE, April 2013.

[74] Marcos K. Aguilera, Robert E. Strom, Daniel C. Sturman, Mark Astley, and Tushar D. Chandra. Matching Events in a Content-based Subscription System. In *Proceedings of the 18th Annual ACM Symposium on Principles of Distributed Computing*, PODC '99, pages 53–61, New York, NY, USA, May 1999. ACM.

VITA

VITA

William Culhane was born in Indiana, but moved to Ohio before any long term memories were formed. He graduated from Wyoming High School in Wyoming, Ohio in 2004 and began undergraduate studies at the Ohio State University the same year. In 2008 he graduated *Cum Laude* with distinction and honors with a degree in Computer Science and Engineering with a minor in Music. William started graduate studies at Purdue in the fall of that year and received an M.S. degree in May 2011 and Ph.D. in May 2015 under the direction of Dr. Patrick Eugster. William's research interests include data processing and manipulation, especially in the area of cloud and distributed computing. He has completed internships at NCR, Google, and Qatar University.

Outside of academic interests William enjoys reading and physical activity. He is an accomplished speed skater with seven national medals in USARS competition. He also spent a summer as a whitewater rafting guide on the New River in West Virginia.