FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Ressourceneffizienz in Tritonsort

Optimierung der Ressourcenverwendung in verteiltem speicher-externem Sortieren großer Datenmengen

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering/Internet Computing

eingereicht von

Alexander Pucher
Matrikelnummer 0525262

an der
Fakultät für Informatik der Technischen Universität Wien

(Durchführung am Department for Computer Science and Engineering, University of California San Diego)

Betreuung
Betreuer: Prof. Dr. Stefan Biffl (TU Wien)
Betreuer: Prof. Dr. Amin E. Vahdat (UCSD)

Wien, 09.10.2010 _____ _____

(Unterschrift Verfasser/in) (Unterschrift Betreuer/in)

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Resource Efficiency in Tritonsort

Optimization of Resource Utilization in Large-Scale Distributed External Sorting

MASTER THESIS

to obtain the academic degree

Master of Science

Curriculum

Software Engineering/Internet Computing

submitted by

Alexander Pucher
Matriculation Number 0525262

Vienna University of Technology, Department for Informatics

(Research at Department for Computer Science and Engineering, University of California San Diego)

Scientific Supervision
Supervisor: Dr. Stefan Biffl (Vienna UT)
Supervisor: Dr. Amin E. Vahdat (UCSD)

Vienna, 10/09/2010 _____ _____

(Signature Author) (Signature Supervisor)

Erklärung zur Verfassung der Arbeit

„Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe."

Wien, 09.10.2010

Alexander Pucher

Kurzfassung

Internetanwendung in der Größenordnung von Yahoo's Webportal oder Google's Such- und Cloud-Diensten arbeiten mit massive Datenmengen. Die rechtzeitige Verarbeitung von Tera- und Perabytes an Daten macht die Verwendung von DISC Systemen notwendig. Der jährliche Sort Benchmark vergleicht die Performance aktueller Systeme und hat über die vergangenen Jahre einen stetigen Leistungszuwachs beobachtet, der jedoch mit drastischen Einschnitten in Hardwareeffizienz erkauft ist. Die „Tritonsort" Fallstudie wurde initiiert, ein leistungsstarkes und kosteneffizientes System für Sort Benchmark zu entwickeln, dessen Design primär auf Ressourceneffizienz achtet. Diese Arbeit beschreibt die Entwicklung zweier Systemkomponenten von Tritonsort und bietet die systematische Evaluierung von Ressourceneffizienz in Tritonsort.

Ziel. Das Ziel der Arbeit ist die Entwicklung zweier Komponenten für Zwischenspeicherung von Daten und speicher-internes Sortieren von Daten, sowie die Evaluierung der Ressourceneffizienz im Vergleich zu state-of-the-art Systemen. Die Lösung wird als erfolgreich angesehen wenn Tritonsort in den Benchmarks 100TB „Gray Sort Indy" und 60 Sekunden „Minute Sort Indy" Höchstleistung liefert und verbesserte Hardware- und Kosteneffizienz bietet.

Methode. Literaturstudie über existierende Ansätze zu verteiltem parallelem Sortieren, Design und Implementierung von Systemkomponenten für den Tritonsort Prototypen und Evaluierung des Ergebnisses durch systematischen Vergleich mit existierenden Systemen im Bezug auf Leistung und Effizienz.

Resultat. Tritonsort erreicht 2010 Bestleistung in den Kategorien „Gray Sort Indy" und „Minute Sort Indy" mit viermal höherer Leistung pro Maschine als andere Systeme. Zusätzlich wird die Durchschnittsleistung pro CPU und Festplatte erhöht, wodurch bessere Kosteneffizienz erzielt wird.

Literatur. Anderson und Tucek begründen die Notwendigkeit von ressourceneffizienten Systemen und liefern eine Effizienz-Systematik. Vitter schafft die Grundlage für effizienten Festplattenzugriff in externem Sortieren, während existierende Systeme Ansätze zu internem Sortieren inspirieren.

Abstract

Internet scale services like Yahoo's web portal and Google Search and Cloud services operate on massive amounts of information. Timely processing of this data at the scale of Tera- and Petabytes requires the use of DISC systems, orchestrating large assets of hardware with frameworks such as Apache Hadoop. Annually, DISC system performance is compared by Sort Benchmark and benchmark results over the past years show gains in performance, although a substantial loss in resource efficiency is found. The Tritonsort case study is set up to create a top-performing and cost-effective system for large-scale Sort Benchmarks by emphasizing resource efficiency in design primarily. This paper describes design and implementation of two core components and provides a systematic evaluation of resource efficiency in Tritonsort.

Objective. The objective is development of well-performing intermediate data storage and internal sorting for Tritonsort and an in-depth evaluation of resource efficiency compared to state-of-the-art systems. The solution is deemed effective if Tritonsort outperforms in Sort Benchmark categories 100TB Gray Sort Indy and 60 seconds Minute Sort Indy and provides competitive hardware and cost efficiency.

Method. Survey in literature about existing approaches to distributed parallel sort, design and implementation of internal sorting and disk I/O components, and evaluation by systematic comparison to existing systems in terms of performance and resource efficiency.

Result. Tritonsort uses both subsystems and outperforms state-of-the-art systems in the 2010 "Gray Sort Indy" and "Minute Sort Indy" challenge by a factor of four per cluster node. Also, it improves upon average throughput per CPU core and disk which leads to higher cost efficiency.

Literature. Anderson and Tucek emphasize the potential of resource efficient systems and provide a systematic listing of different aspects of efficiency. Vitter creates the foundation for efficient disk I/O in external sorting while different Sort Benchmark systems inspire design and optimization of internal sort.

Table of Contents

Index of Figures

Index of Tables

1 Introduction

Data warehouses of Internet companies store vast amounts of data. The ability to process this data in a timely manner is vital to company revenues through advertising and retailing. Accessing and mining these amounts of data is therefore a major challenge, alongside failure-redundant storage, security and others.

Such amounts of data are typically found in internet scale services and cloud computing applications. (1) The need to deal with large data sets gave birth to a series of new batch-processing frameworks such as Apache Hadoop (2), Google MapReduce (3) or Microsoft Dryad.(4) The analysis of stored information is typically distributed across hundreds of individual machines for raw storage requirements and I/O bandwidth. This focus on parallel processing of data instead of expensive arithmetic computation distinguishes “Data-intensive Super computing” (DISC) from traditional “High-performance Computing” (HPC). (5)

One example for the scale of data processing is provided by Jeffrey Dean in a 2009 talk about utilization of MapReduce in Google’s data centers. In September 2009 Google’s data centers processed 540 Petabyte of input data, using 25,500 machine years worth of processing time.(6)

At these scales it seems favorable to operate a cluster close to its hardware limits - with high resource efficiency - to keep initial investments and running costs as low as possible. (7) This motivates the design of “balanced systems” (8) that optimize software and hardware for I/O bound workloads that are found in DISC applications.

All major DISC frameworks are work in progress, and hence, developers frequently compete in benchmarks and publish performance numbers (9), although fine-grained comparability of the processing frameworks is a mostly unsolved issue (8). This phenomenon is nothing new - in 1985 the Datamation benchmark suite was proposed by (10) and has since been adapted multiple

times to account for rapid changes in hardware performance and capacity (11)(7). Part of this suite is the Sort Benchmark: it requires a set of unordered records to be read from disk storage, sorted and written to a sequential output file. Sort benchmarks have been used since to compare performance and efficiency of parallel processing systems close to real workloads for a long time. (12) The task of sorting is simple enough to be performed with any system, yet it is an I/O bound task and touches most aspects of the system, including CPU, memory, disk and potentially network I/O. Different metrics are used to evaluate competing systems in terms of performance and efficiency. In general, systems dealing with large amounts of data are compared in terms of performance only, while small scale systems use metrics targeted at cost and energy efficiency.

There have been considerable advances to benchmark performance over the past 20 years (12). This is mainly credited to increases in hardware performance as well as the use of Beowulf-type clusters instead of mainframe systems. (13) However, the growth in performance comes at increasingly high costs due to the use of large amounts of cluster nodes. It was pointed out recently that system hardware efficiency suffered heavily and the explosion in scale may mislead future developments into a competition of pure economic investment. In (8) E. Anderson and J. Tucek estimate an average throughput of less than 12 MB/s per node for modern enterprise-class DISC clusters competing in large scale sort benchmark. Compared to more than 100MB/s per server on small systems built from consumer quality hardware for reasons of cost-efficiency there seems to be a potential for improvement.

This motivates the “Tritonsort” project to approaches large-scale benchmarks from a resource efficiency point of view. Tritonsort aims at improving upon existing large-scale sort benchmark results while using significantly less hardware resources. Hence, it is designed with efficiency considerations in the first place and incorporates experience and lessons learned from state-of-the-art systems.

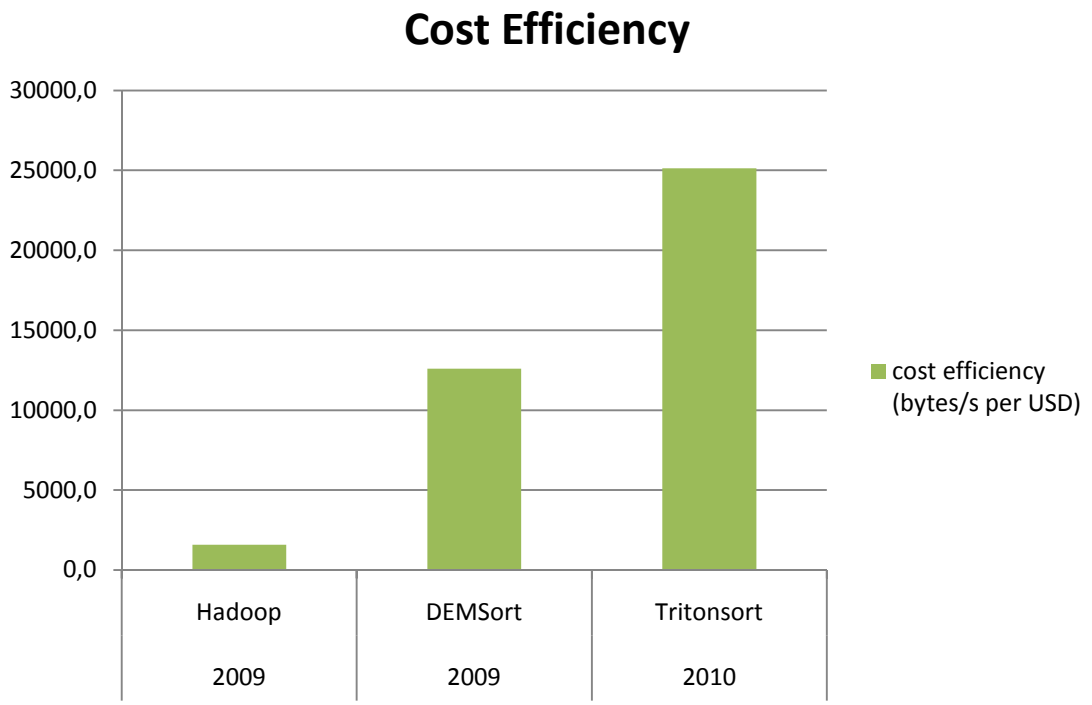


Figure 1 - Large-scale benchmark cost-efficiency

The figure shows 2010 Sort Benchmark cost-efficiency for Hadoop, DEMSort and Tritonsort. All three systems compete in the Sort Benchmark "Gray Sort" category, handling large-scale datasets of 100TB or more. Benchmark performance of these systems is almost identical, with Tritonsort being the top-performing system with a slight advantage. From a cost-efficiency point of view the differences become substantial, however. (Cost estimations for DEMSort and Hadoop are based on (8))

This paper specifically focuses on two aspects in the development of Tritonsort. First, two core components for intermediate data storage and internal sorting are designed and implemented. Both affect disk I/O efficiency and computational efficiency of Tritonsort substantially. Second, a systematic evaluation of Tritonsort's resulting resource-efficiency is provided, comparing Tritonsort to state-of-the-art systems in large-scale Sort Benchmark.

The key contributions of the thesis are:

1. **Overview of state-of-the-art systems**

An overview of state-of-the-art systems competing in Sort Benchmark is given. Architecture and algorithms of systems are described and specific strengths are analyzed. Experience from small- and large-scale systems is collected and discussed in the context of resource efficiency. This leads to a series of open questions and challenges in creating resource-efficient data persistence and sort implementation.

2. **Data persistence for sustained rate parallel file writes**

The Tritonsort prototype is supplemented with a disk I/O layer for writing to large amounts of files that overcomes file-system weaknesses for sustained rate transfers to multiple files co-located on a physical disk. An optimal approach from literature is adapted to satisfy benchmark requirements and hardware limitations and performance is compared to alternative approaches.

3. **Memory-efficient linear-time sort**

A memory-efficient linear-time internal sort component is designed and implemented. Radix sort is adapted to the specific use case and memory requirements are reduced to O(1.3n). Additionally, a performance evaluation is performed to ensure sufficient throughput for on-the-fly processing of data.

4. **Concept for evaluation of resource-efficiency at large-scale**

A concept for evaluation of resource-efficiency of large-scale systems in Sort Benchmark is provided. An approach suggested by (8) is adapted to quantify aspects of computational, I/O and cost efficiency.

5. **Systematic evaluation of performance and resource-efficiency**

Performance and resource-efficiency of the integral Tritonsort prototype is evaluated using the above approach. Large-scale benchmark results of state-of-the-art systems are compared to Tritonsort and discussed in the light of resource-efficiency. This is used to identify specific strengths and potential for optimization in the future.

The document is structured as follows: Section 2 summarized relevant related work and gives an overview of the state of the art in Sort Benchmark. In Section 3 design and architecture of Tritonsort are presented. Section 4 derives research issues and summarizes challenges of architecting for resource efficiency. Practical work of the thesis is described in Section 5, the evaluation is performed in Section 6. Practical work and evaluation represent the main part of the thesis; they describe the disk I/O layer and Radix Tag sort implementation and quantify resource-efficiency of Tritonsort and state-of-the-art systems. Section 7 summarizes and discusses results and provides perspectives for further research.

2 Related work

This section provides an overview of related work and summarizes important concepts. First, distinct aspects of efficiency in DISC systems are introduced and a number of benchmarks and metrics quantifying them are presented. Then fundamental algorithmic approaches to external memory sort are provided. Finally, state-of-the-art systems in Sort Benchmark are presented and compared in terms of architecture and efficiency.

2.1 Types of efficiency

In a 2010 paper E. Anderson and J. Tucek describe eight aspects of efficiency in DISC systems. These are computational, I/O, storage, memory, programmer, management, energy and cost efficiency. The following listing summarizes definition and potential impacts.

1. **Computational efficiency**

There are two major variables affecting computational efficiency. At first, the amount of CPU cycles required to generate the desired result and second, the amount of cycles being wasted due to idling. The first aspect can be addressed by the choice of algorithm (14) and programming tools (8). The second aspect can be dealt with mainly by process scheduling. A side-effect of computational efficiency is an impact on energy and cost efficiency.

2. **Input/Output efficiency**

The I/O efficiency of a device is determined by the average throughput achieved divided by its theoretical maximum. The overall result is the average of all relevant-system devices and is mainly impacted by replication redundancies and idle times. The use of replication is an architectural decision while idle times may be caused by inappropriate design or bottlenecks elsewhere.

3. Storage efficiency

Efficient storage minimizes the overhead of physical data storage relative to the amount of logical data in the system. Overheads are mainly generated by replication and additional metadata. For example, a system with 10 percent file-system overheads using 2-way replication would provide a storage efficiency of 0.45.

E. Anderson and J. Tucek argue that compression can improve this kind of efficiency by a multitude. While this is valid for production systems, rules of Sort Benchmark do not allow any kind of data compression due to the emphasis on I/O bandwidth dependent metrics.

4. Memory efficiency

Memory efficiency focuses at the overhead of data held in volatile storage. This is impacted by data structures, heap requirements and memory fragmentation. When performing memory-intensive tasks, such as write caching or internal sorting, the choice of algorithm may have severe impact too.

5. Programmer efficiency

Programmer efficiency is directly related to productivity, and hence, it is difficult to measure. It is argued the choice of tools, programming languages and frameworks heavily influences this aspect. Generally, this is a factor related to the broader topic of development processes and reusable software designs and lies out of the scope of this work.

6. Management efficiency

Management efficiency describes the effort required to maintain the system infrastructure relative to the minimum amount required. Any quantitative results in this are specific to a site or technology. Qualitatively, it can be observed that increased specialization of hardware leads to decreased management efficiency.

7. **Energy efficiency**

The overall energy consumption of system infrastructure to complete a given task is compared to previous systems in benchmarks to obtain a relative energy efficiency measure. It is suggested that a considerable portion of the total consumption may be caused by auxiliary devices such as cooling, therefore emphasizing small hardware appliances.

8. **Cost efficiency**

There are multiple ways to measure cost efficiency, e.g. records/dollar (data processed per investment) or records/second/dollar (data throughput per investment). The preferred metric depends on specific application requirements and may be biased towards certain benchmarks or configurations.

2.2 Benchmark and Metrics

Initially defined in “A Measure of Transaction Processing Power” by Anon et Al. in 1985 the benchmark suite later named “Datamation” has continually been refined and extended (11)(10)(7). It is designed to provide a minimal level comparability of real-world performance of different systems and platforms. Multiple aspects of a system are tested instead of relying on vendor whitepapers and theoretical metrics such as maximum MIPS. For this purpose each mini-benchmark includes a range of I/O operations testing the system’s interface to its environment. The first generation of benchmarks included “DebitCredit”, “Scan” and “Sort”. The initial metric used to rank systems was the cost/performance ratio based on elapsed time and total throughput.

“DebitCredit” measures remote transaction performance and latency. It simulates a banking scenario timing system and database transactions initiated from remote ATM terminals. “Scan” tests throughput achievable through high-level interfaces used by application developers. Finally, “Sort” benchmarks maximum I/O performance by measuring elapsed time for reading and processing a fixed set of input data and writing the sorted sequence back to disk.

The original “Datamation” sort benchmark operated on a 100MB input data and was replaced by a series of different categories to account for increased

hardware capacity and emerging topics such as energy efficiency. As of June 2010 there are four different metrics applied to two different types of datasets and multiple input sizes. The dataset types are split into general purpose “Daytona” tuples with variable length and key size and performance-focused “Indy” fixed-length 100-bytes records with a 10-bytes key. Each of the metric described below favors a different system setup and imposes certain restrictions to ensure comparability.

1. **Gray sort**

The input size is restricted by a lower bound of 100TB and system performance is measured by throughput (TB/min). The benchmark replaced the Terabyte sort challenge and is named in honor of Jim Gray after his disappearance at sea in 2007. (9) Due to the emphasis on large datasets, systems need to rely on distributed processing on clusters in general.

2. **Minute sort**

The amount of data processed in less than a minute is used as metric and includes time for system startup and shutdown. (A maximum runtime of 60 seconds) Current results supersede 1TB which requires a substantial amount of disks to provide the required bandwidth. Minute sort was introduced by (11).

3. **Penny sort**

The amount of data sorted for the equivalent of one penny system cost, assuming the hardware has a lifetime of three years. This category was first suggested by (11) too and favors cheap hardware setups. Competing systems utilize hardware efficiently, but their absolute scale currently does not surpass several hundred Gigabytes of total storage capacity.

4. **Joule sort**

The amount of data sorted for the equivalent of one Joule, including total energy consumption of any hardware and auxiliary devices used. This was introduced by (7) and further emphasized by (8) due to drastically decreasing resource efficiency of systems competing in Gray- and Terasort benchmarks. The benchmark is currently performed for multiple input sizes between 10GB and

100TB. The focus on energy consumption favors expensive flash-based storage devices that are currently not suitable for large-scale deployments.

Benchmark rules do not restrict the scale of a hard- or software, however, they require commodity components to be used. In addition, any kind of pre-caching and compression is disallowed to support the focus on I/O efficiency. (9)

2.3 Sorting Algorithms

The following part gives an overview of memory external sort. Merge and distribution based approaches are presented

Any kind of internal sort algorithm at least requires memory equal to the size of a run being sorted. Given a memory starved environment external sorting is required, i.e. a cluster providing 1TB of main memory while sorting 100TB of data. External sort passes through data two times or more. It first partitions data into smaller subsets and then iteratively aggregates them to a global result. (14)

Vitter (15) describes the two fundamental approaches to external sorting, sorting by distribution and sorting by merging. Vitter also takes different approaches for I/O into account and derives a duality in terms of I/O operations for both approaches.

2.3.1 Distribution Sort

Distribution sort is a recursive process that partitions a set of items into multiple disjoint buckets. Each bucket only contains items that are smaller than items in subsequent buckets. Hence, when all buckets are sorted individually and concatenated, the result is sorted globally.

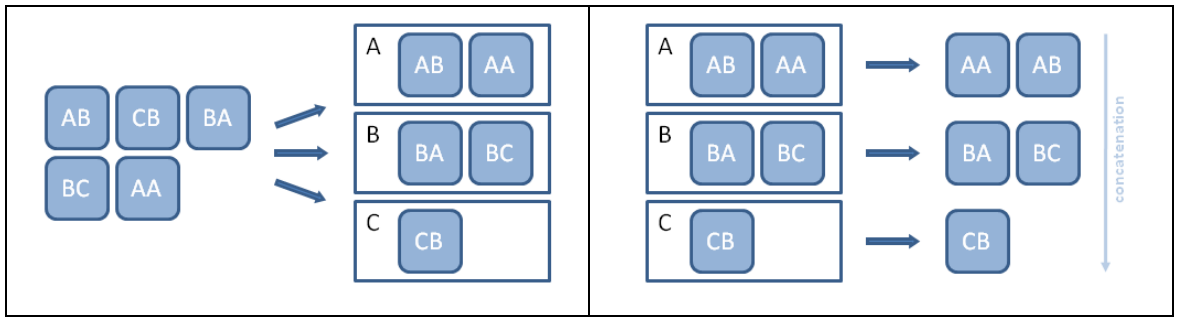


Figure 2 - Distribution Sort

Before distribution-sort iterations can be performed the partitioning elements have to be determined. Their values represent the boundaries of a bucket. All items found to be in between the two partitioning elements are put into the same bucket. When partitioning elements are selected optimally, items are distributed across buckets evenly. There a multiple deterministic and probabilistic methods to select partition boundaries with a general trade-off between expense and accuracy. (16)(17)

When applying distribution sort in an environment with multiple parallel disks many I/O-operations can be load-balanced to improve performance. Vitter (15) shows that randomization-based approaches provide optimal I/O behavior.

2.3.2 Merge Sort

Merge sort is a two step process. First, a number of sorted runs are generated by filling memory, performing internal sort and writing them back to disk repeatedly. Then, these runs are merged into a single sequence by reading each sequence from disk in small chunks and heap-sorting individual items.

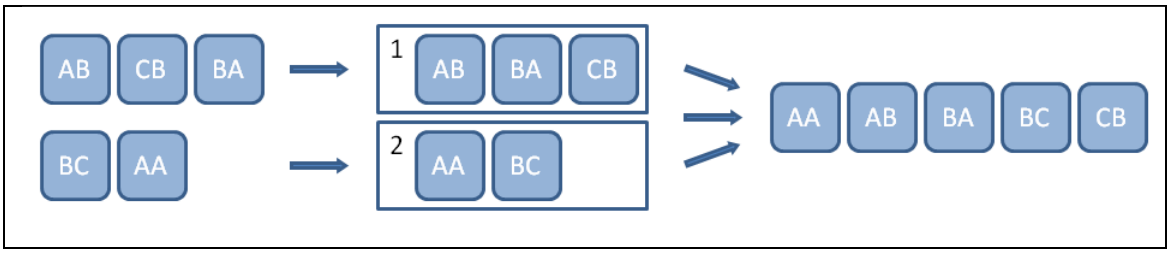


Figure 3 - Merge Sort

Merge sort does not require partitioning elements as partitioning is based on run size only. Hence, Merge sort can be easily adapted to different amounts of main memory by changing the size (and number) of runs. The merge operation is based on comparison and cannot be run in linear time.

This approach allows for load-balancing across multiple disks too, optimal pre-fetching of chunks requires relatively complex algorithms, however.(15)

2.4 Architectural approaches

Numerous architectural approaches to distributed sorting can be found in literature. In the history of Sort Benchmark early systems were built on shared memory mainframes (11) and were replaced by distributed systems built from large amounts of similar commodity hardware components. (18)(13) Also, some hybrid systems proved to be successful using heterogeneous (19) or specialized hardware (20) and mainframes operating on external disk clusters. (21)

2.4.1 Shared memory and Data partitioning

Access to main memory and secondary storage in parallel sort implementations is popularly modeled either by a shared-memory or a data-partitioning approach, as denoted by (11).

Shared-memory systems assume access to all data by any node in the system. This model is used by frameworks that transparently handle the exchange of data across the network or by mainframe systems that provide hardware support for large amounts of main memory. Data-partitioning approaches operate on node-local datasets only and explicitly exchange messages and data between nodes.

The general advantage of shared-memory is simple implementation of algorithms while data-partitioning does not require any additional hardware or software support in order to function on networks of commodity workstations. (18)

2.4.2 Single- and Multi-pass sorting

The number of I/O operations mainly depends on the amount of memory in the system relative to the total amount of data being sorted (11) (18). A memory-rich environment can rely on single I/O-pass design. In a memory-starved environment two read/write passes have to be performed. Depending on the design, additional passes may be introduced. (22) In general this leads to longer runtimes and decreases I/O efficiency, but may be necessary to overcome memory constraints.

The single-pass system reads input data from external storage once. Data is stored in memory and sorted internally. The result is written back to disk in the end.

The two-pass system reads input data until main memory is full, performs an internal sort operation and stores the intermediate result on disk. This process is repeated until all input data has been processed. In a second pass the intermediate data is processed again to generate the final result.

A system may perform additional passes if required by architecture or for memory considerations. For example, this could be necessary when using merge sort in heavily memory starved environments. If main memory cannot hold one tuple of each intermediate result, a separate series of runs needs to be generated by partial merging. Then, the final result can be generated by another merge pass.

2.4.3 Synchronous and interlaced I/O

Independent of the number of passes performed it is desirable to maximize I/O throughput over run-time. As described by (11) computation and I/O can either be performed serially (synchronously), in parallel (threaded) or interleaved (relying on kernel level concurrency only). Parallel pipelining increases throughput rates for external storage and network by eliminating most idle times, but might increase memory requirements. Also, due to read and write access occurring at the same time disk and controller setup have to be adapted for a pipelined approach. Disks have to be partitioned into separate sets for reading and writing to prevent a drop in throughput due to random instead of sequential access patterns.

2.4.4 In-place and out-of-place on storage

Due to architectures considerations and rule restrictions in the Daytona benchmark category most submissions to the sort benchmark use dedicated disks for input and output. This out-of-place approach to processing imposes a penalty to storage efficiency of a factor of two. By using main memory as buffer, input data can be processed and overwritten in-place using minor additional disk space. A recent submission using this approach is DEMSort. (22)

An in-place design has some drawbacks, though. It imposes an additional amount of random disk access and is more complex in design and implementation in order to deal with fluctuations in input data distribution. Also, in real-world applications it is generally not desirable to lose the input data, which is reflected by the Sort Benchmark Daytona rules. (9)

2.4.5 Parallel sort algorithms

The NOW-Sort paper (18) describes an approach for parallel sorting on a network of nodes using data-partitioning. Processing is split into a distribution and a merge phase.

The first phase distributes data in a single pass. It examines each tuple, calculates a hash from the key value and distributes the tuple to the according bucket. This task is performed in parallel on all cluster nodes and tuples are sent to buckets over the network. At the receiver side tuples are buffered and, when a bucket becomes full, it is sorted and written to disk. The second phase locally reads and merges all sorted runs stored on a node and writes the sequential output to disk.

2.5 State-of-the-art systems

The following section gives an overview over some existing systems in the context of Sort Benchmark. This includes a description of architecture and the techniques employed to achieve improve performance. In addition to recent systems, NOW-Sort, an influential implementation of distributed sorting on commodity hardware is presented. However, the listing merely provides an exemplary overview about the broad variety of systems in Sort Benchmark and cannot be regarded as complete reference.

In subsequence, advantages and disadvantages of these systems are summarized and discussed, specifically regarding the applicability for large scale deployments. Some systems are developed with high scalability in mind, whereas others optimize for energy and cost-efficiency. As Tritonsort aims at resource efficiency and large scale valuable approaches to optimization can be derived from all these systems.

2.5.1 DEMSort

DEMSort (22) competes in Gray Sort Indy and Minute Sort Indy and uses a three stage process. First, a global shared memory sort is used to generate sorted runs and globally stripe them across disks. Then, the exact global partitioning of tuples is determined and non-matching tuples are redistributed across nodes. In the end, each node locally merges runs on disk, yielding a globally sorted run. The system reads and writes most data twice and operates near in-place, which comes at the cost of some additional disk and network load due to re-distribution after determining the exact partitioning in the second step.

Configuration	**Gray Sort Indy**	**Minute Sort Indy**
$nodes$	195	195
$cores$	1560	1560
$disks$	780	780
$estimated\ cost$	$10^{5.9}\ USD$	$10^{5.9}\ USD$
Metrics		
$year$	2009	2009
$performance$	$0.565\ TB/min$	$0.955\ TB$
$data$	$100\ TB$	$0.955\ TB$
$time$	$10628\ s$	$60\ s$
$throughput$	$9409.1\ MB/s$	$15916.7\ MB/s$

Table 1 - DEMSort fact sheet

In terms of hardware DEMSort relies on a 200 node cluster. A node contains two quad core Intel Xeon processors with a 2.66GHz clock, 16 GB main memory and 4 disks providing 1TB of storage per node. The network interconnection is provided by a single InfiniBand 4x DDR switch. It is noted by the developer that only 60% of total disk storage is available for sorting, however.

The operating system is Suse Linux Enerprise 10 SP 1 running on kernel 2.6.22. The file system XFS is used, backed by a RAID-0 configuration of disks DEMSort is built on C++ using the GCC 4.3 tool chain and the MPI implementation MVAPICH 1.1.

2.5.2 Hadoop

The Hadoop sort benchmark submission (23) is built on Apache Hadoop and leverages from distribution sort inherent to the Map-Reduce programming model. Hadoop competes in Gray Sort Daytona and Minute Sort Daytona, and hence, includes an additional sampling stage to stochastically determine partitioning elements before the actual sorting takes place. In the beginning, input data is read, sampled and partitioning information distributed. In the Map step input files are processed and tuples are sent to their designated target nodes. The Reduction step sorts data locally at each node and saves the result to the distributed file system HDFS. For Gray Sort a replication factor of 2 is used, Minute Sort is executed without replication on a smaller subset of nodes.

Configuration	**Gray Sort Daytona**	**Minute Sort Daytona**
$nodes$	3452	1406
$cores$	27616	11248
$disks$	13808	5624
$estimated\ cost$	$10^{6.8}\ USD$	$10^{6.4}\ USD$
Metrics		
$year$	2009	2009
$performance$	$0.578\ TB/min$	$0.500\ TB$
$data$	$100\ TB$	$0.500\ TB$
$time$	$10380\ s$	$59\ s$
$throughput$	$9633.9\ MB/s$	$8474.6\ MB/s$

Table 2 - Hadoop fact sheet

The hardware is made up from a homogenous cluster of approximately 3800 machines each containing two quad core Intel Xeon processors with a 2.5GHz clocking. A node also holds 4 disks for secondary storage and 8 GB of main memory. Networking is enabled by a 1Gbit Ethernet interface per machine per rack internally. Externally, each 40 machine rack is connected to a central hub with an 8Gbit interface.

The operating environment is provided by Red Hat Enterprise Linux Server Version 5.1 based on the 2.6.18 kernel. Hadoop relies on the Java 6 environment Sun JVM 1.6.0 32bit for small sort runs and 64bit for head nodes

during large sorts. The codebase for the submission is Hadoop 0.20 and integrates a custom shuffle stage, a modified tracker and optimizations to the map and reduce stage. Most notably, some superfluous disk I/O and hard-coded wait-loops are removed and map outputs are compressed. (For the 2010 benchmark a rule change disallowed any kind of data compression)

2.5.3 PSort

Psort (24) leads the Penny sort category for Daytona as well as for Indy input data categories. It employs multi-level merge sort and optimizes for cost-efficiency on small scale, hence putting high emphasis on computational efficiency and I/O efficiency by pipelining. External sort is performed by a two-pass merge sort, reading and writing data at disk twice, processing each datum in memory multiple times depending on the number of internal merge passes. Internal sorting relies on merge-sort for Daytona and a hybrid bucket-sort merge-sort for Indy datasets. Notable optimizations for computational efficiency include the tag-based internal sort, cache-aware buffer sizes and reduction of branching instructions. Application I/O is tuned by relying on parallel disk access, the choice of file system and asynchronous direct I/O provided by commodity Linux.

Configuration	**Penny Sort Daytona**	**Penny Sort Indy**
$nodes$	1	1
$cores$	1	1
$disks$	5	5
$cost$	428 USD	428 USD
Metrics		
$year$	2009	2009
$performance$	225 GB	248 GB
$data$	225 GB	248 GB
$time$	2211 s	2211 s
$throughput$	101.8 MB/s	112.2 MB/s

Table 3 - PSort fact sheet

Being a small-scale system psort's hardware consists of a single machine that does not require any networking. It contains a 2.6GHz AMD Athlon LE

processor, 5 disks and 4GB of memory. The XFS file system is applied to a RAID-0 configuration and the operating system is Gentoo Linux, a concrete version is not specified by the developers.

2.5.4 EcoSort

EcoSort (25) uses flash-based storage for JouleSort Indy and thus optimizes for energy efficiency. It employs a two-pass external merge sort algorithm. Internal sort utilizes a tag-based hybrid bucket-merge sort comparable to Psort. The design relies on parallel sorting by two physical cores and leverages from low latency provided by SSDs at random read access.

Configuration	**Joule Sort Indy (10^8)**	**Joule Sort Indy (10^9)**
$nodes$	1	1
$cores$	2	2
$disks$	4	4
$cost$	3500 USD	3500 USD
Metrics		
$year$	2010	2010
$performance$	2.3 kJoules	25.1 kJoules
$data$	10 GB	100 GB
$time$	72 s	691 s
$throughput$	138.9 MB/s	144.7 MB/s

Table 4 - EcoSort fact sheet

Also a small-scale system, EcoSort runs on a single physical machine. The CPU is a two core Intel Atom 330 and is backed by 4 GB of main memory and 4 Super Talent 256GB MLC SSDs. The system relies on Debian Linux kernel 2.6.30 and uses RAID-0 XFS for file storage. EcoSort is built using the GCC 4.4. tool chain.

2.5.5 NOW-Sort

NOW-Sort (18) was developed over a decade ago, competed in the original 100MB Datamation benchmark and was re-used later in winning systems in the Minute Sort Benchmark. (19) It represents the first sort implementation based on a cluster of commodity hardware machines that competed successfully against mainframe setups. The hardware provided multiple processors and

disks per machine, which has become a typical environment today. For distributed external sorting the system relies on a two-pass algorithm, first distributing data across nodes and generating sorted runs, then merging runs locally to obtain the final result. Distribution is performed by separating input tuples based on a key prefix and sending them to target nodes in batches. The Receiver combines a series of batches up to its memory limit and generates a run that is written to disk. Internal sorting is realized as hybrid bucket-sort radix-sort that is optimized using cache-aligned bucket and buffer sizes and tag-based sorting. Application I/O relies on disk striping and partial pipelining; read and write operations on disk and network are interleaved while sorting is performed synchronously before writing. Though, this prevents maximum throughput in the first pass, it allows large runs to be generated which in turn reduces disk and processing overheads in the second pass.

A look on the hardware employed by NOW-Sort is more of archival character as the initial records were set in 1997 with respective hardware. The cluster uses 64 nodes, each equipped with two (fast-narrow SCSI) disks and 64MB of main memory. Networking is realized by 160 MB/s Myrinet Cards connected by 26 switches in a 3-ary tree topology.

2.5.6 Summary and Comparison

Papers presenting these systems identify efficient sort implementations and interleaved I/O as core challenges in design and development. Network bandwidth is not perceived as limiting factor, for minor exceptions (19), as network interconnections outperform disk I/O per cluster node for most systems.

Systems that rely on multiple nodes to run the benchmark need to distribute data after it has been read off disks initially. This is approached by analyzing a portion of each tuple's key and storing the tuple in the according intermediate buffer. At some point data is transmitted across the network in batches and processed at the target node.

Most implementations presented above rely on run formation in the first pass and merge sort in the second pass. On a large scale this resembles a hybrid bucket-sort of data across nodes or disks, internal sorting of buckets at each

node and a final local merge sort. Data is being sorted trice: for distribution, for run formation and for merging.

Internal sort implementations rely on merge-sort or bucket-sort based approaches and hybridization. Though, merge-sort has a theoretical boundary of O(n log n) operations compared to O(n) bucket-sort, most concrete merge-based implementations show competitive benchmark performance. The use of key-pointer tags instead of full size tuples shows superior performance for large input size and can be improved further by introducing cache-aligned buffers.

In terms of secondary storage access these systems use a two pass merge sort approach. In the first pass data is read off disk sequentially and generated runs are persisted in the same way. The second pass performs merge sort, reading blocks from existing runs in parallel and writing a sequential output file. Sequential access provides maximum throughput for commodity disks, seek times induced by random access during the second read phase lead to a drop in performance. The number of seeks is minimized by balancing the number of files with an appropriate input block size. Most systems report a specific number of files they can sustain for merging before performance drops sharply. EcoSort leverages heavily from SSDs and flash memory, as seeking does not apply to solid state drives. DEMSort uses a slightly different approach to distribution of data across nodes by using estimates for partitioning in a first pass and incurring a re-distribution step before merge sorting local runs. Hadoop uses a distributed file system to store input and output data, but do not provide data on the number of actual read and write operations. The estimated throughput values per disk may indicate that this number exceeds the two-pass minimum. (See Section 1)

A comparison of systems in terms of performance was the initial motivation of sort benchmark. Over time, different metrics were added as it became clear that systems are designed with a different focus in mind. The first additions were minute sort and penny sort, emphasizing low startup and shutdown overheads and cost efficiency. Later, Joule sort was added tackling energy consumption of benchmark systems.

In an economic sense, cost-efficiency is probably the most important metric (ignoring aspects of programmer and management efficiency). Penny sort addresses this directly, but enterprise class systems deal with a larger scale of data than current penny sort systems. Large scale in turn is addressed by Gray sort, but it is hard to compare system in terms of hardware and development cost as relevant submissions do not provide information on hardware cost. In (8) an attempt is made to estimate cost-efficiency for recent submissions to sort benchmark.

The following comparison of system performance and cost-efficiency is inspired by (8) and adds data from recent submissions to sort benchmark. The total throughput of a system during its benchmark run (in MB/s) is divided up per node, CPU core and disk. Although, systems are developed for different benchmark categories performance and cost consideration are made in every case.

Year	Name	Category	Nodes	Cores	Disks	Data (MB)	Time (s)
2009	DEMSort	Gray Indy	195	1.560	780	100.003.000	10.628
2009	Hadoop	Gray Daytona	3.452	27.616	13.808	100.000.000	10.380
2009	psort	Penny Indy	1	1	5	248.000	2.211
2010	EcoSort	Joule Indy	1	2	4	10.000	77

Table 5 - Comparison of benchmark systems

There is a large difference in scale for the amount of data, the run time and the amount of hardware. Hence, any comparison of per node performance can only be regarded as rough approximation. Still, throughput numbers and estimations of cost-efficiency show a drastic advantage for smaller systems that cannot be explained by sole costs of additional networking hardware.

Year	Name	Throughput (MB/s)	TP/Node (MB/s)	TP/Core (MB/s)	TP/Disk (MB/s)	Cost Eff. (Bytes/s/$)
2009	DEMSort	9409,1	48,3	6,0	12,1	1e4,1
2009	Hadoop	9633,9	2,8	0,3	0,7	1e3,2
2009	psort	112,2	112,2	112,2	22,4	1e5,5
2010	EcoSort	129,9	129,9	64,9	32,5	1e4,6

Table 6 - Comparison of system throughput and cost-efficiency

Estimations show there is a factor of more than 100 in cost-efficiency when comparing cost-efficiency of Hadoop, a large-scale application, with psort, the best-performing penny sort system. Also, a significant difference in throughput per hardware component can be found. The estimated gap in cost-efficiency is reflected in throughput numbers per node, core and disk to some extent.

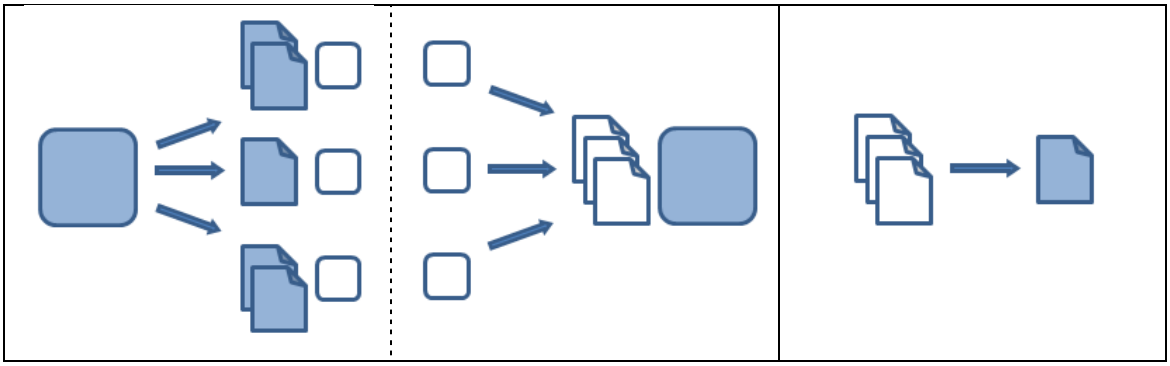
The numbers suggest that there is a potential for improvements to resource-efficiency in large-scale applications. Throughput numbers per component are smaller for large-scale systems DEMSort and Hadoop and indicate issues that do not exist in psort or EcoSort. This might be due to the requirements of networking hardware or low efficiency in algorithms and disk I/O.

3 Tritonsort Architecture

The architectural overview introduces the design of the Tritonsort prototype. Individual steps of the processing pipeline are described and details of pipeline configuration for the Gray Sort and Minute Sort benchmark are provided. In addition hardware and operating system environment of the test bed are presented.

3.1 Pipeline architecture

The Tritonsort prototype competes in large-scale sort benchmarks, hence it performs memory-external sort distributed across a number of individual machines. The processing pipeline is inspired by the Map-Reduce approach, with additional emphasis put on the management of intermediate data. The implementation of external sort is based on distribution sort. The mapping step splits up input data in sufficiently small “buckets”, so each bucket fits into memory for internal sorting by the reduce step.



Map	Store	Reduce
1a Read tuples from disk	**2a** Receive incoming data	**3a** Read presorted data
1b Determine destination	**2b** Cache in memory	**3b** Sort tuples
1c Send to target node	**2c** Store on disk	**3c** Store on disk

Figure 4 - Tritonsort Architecture

The system highly relies on concurrent computation and I/O and is designed as pipeline with consecutive worker stages. Unlike the approach taken by pure MapReduce, a pipeline may have an arbitrary worker (Mapper/Reducer) chain.

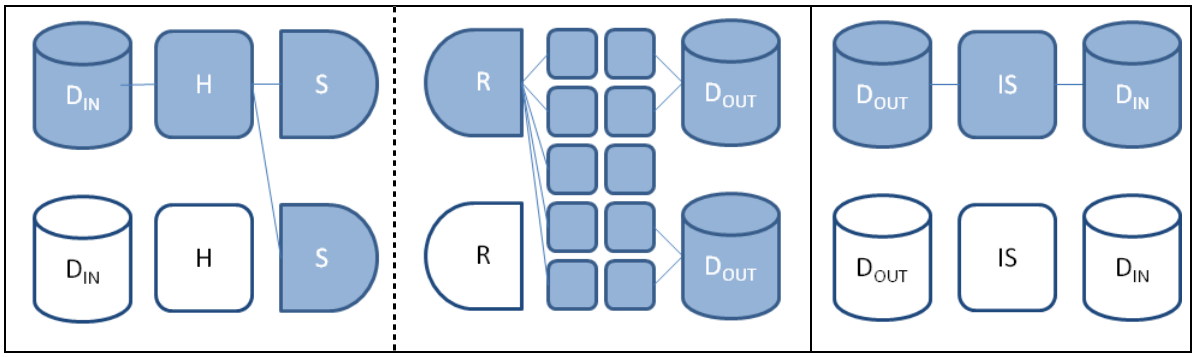
This simplifies process optimization, e.g. by reconfiguration of an intermediate writer stage. Also, this provides increased extensibility as additional stages, e.g. continuous input sampling, can be added to an existing pipeline almost transparently. A trade-off is tight coupling between components if the system is not designed carefully.

The pipeline starts by mapping tuples to target nodes. Input data is read from disk, the target location of each tuple is determined and it is sent to the target location. The destination node collects incoming tuples in their according buckets and stores them on a distinct set of disks. When all tuples have been distributed, each node locally passes through the buckets, sorts the contents and writes the final results back to the input disks.

There currently exist two pipeline configurations capable of performing the Sort Benchmark challenges “Gray Sort Indy” and “Minute Sort Indy”. The Map stage is identical, Store and Reduce stages are adapted to work efficiently for two-pass and one-pass sorting respectively.

3.2 Gray Sort Configuration

The Gray Sort benchmark requires a dataset of at least 100TB, leading to multi-pass sort algorithms and long benchmark runtimes. The system has to sustain high disk and network throughput over multiple hours and guarantee stability.



Map	Store	Reduce
1a Read tuples from disk	**2a** Receive incoming tuples	**3a** Read intermediate data
1b Hash tuple key	**2b** Cache in Buckets	**3b** Sort tuples internally
1c Send to target node	**2c** Store on disk	**3c** Store on disk

Figure 5 - Gray Sort Configuration

For the Gray sort configuration a two-pass distribution sort - internal sort approach is chosen. The map stage distributes data to a global set of buckets across the network whereas the store stage continuously writes bucket contents to the disk. The reduce pass reads individual buckets and sorts them internally. Any I/O operations are interlaced; disks are divided into even partitions for input and output. On the test bed hardware 8 readers and 8 writers are used for a total of 16 disks per node.

The map stage utilizes a hash function to determine each tuple's destination bucket and transmits tuples to the according node in batches using a uniform partitioning function. The receiver collects these into bucket buffers and the writer repeatedly flushes buffers to disk. During the reduce stage bucket files are read from disk, passed to a sorter stage and finally handed to the writer again.

3.3 Minute Sort Configuration

Minute Sort focuses at performance within a timeframe of 60 seconds and therefore favors systems with maximum amounts of parallelism and low startup and shutdown overheads.

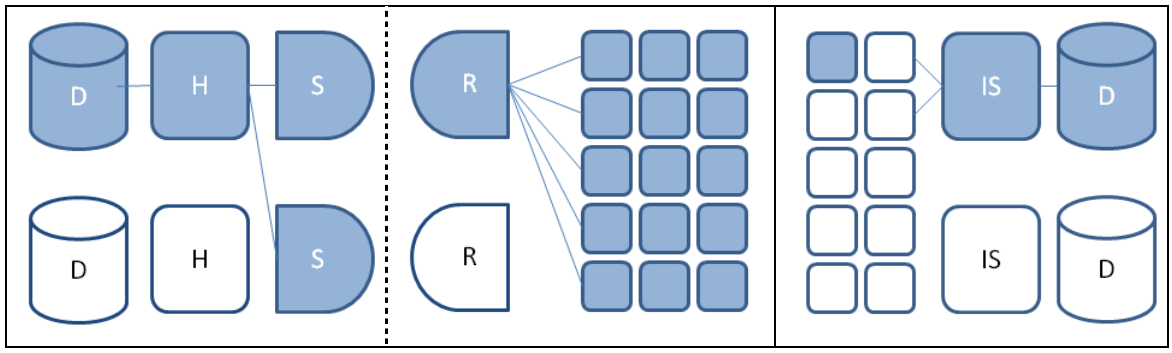


Figure 6 - Minute Sort Configuration

Minute sort is configured as distribution sort - internal sort without intermediate file creation using a synchronous I/O pipeline. The pipeline layout is similar to Gray sort, but the intermediate file writer and reader stages are removed and

buckets are processed directly by the internal sort stage. Input data is read from all disks on all nodes in parallel, distributed, sorted and written back to disk concurrently.

There are several design decisions that show to have a major impact on the efficiency of the prototype. Computational efficiency is achieved mainly by using a single pass hash function and a linear-time sort algorithm and by keeping in-memory copies to the minimum. Memory efficiency is owed mainly to buffer pools and dynamic size tuning instead of plain double buffering between worker stages. Certain trade-offs are made that impact memory usage, e.g. the tag-based radix sort algorithm uses additional overhead memory compared to Quicksort. In case of tag-based sorting this allows additional optimization, however. Finally, I/O bandwidth is optimized by keeping network performance steady by batch transmission of data and reducing disk seek times by writing data to buckets in long sequences. While the number of seeks can be reduced a fundamental relationship between intermediate and output data prevents purely sequential access when using the distribution-based approach to external sorting.

3.4 Test bed

The test bed consists of 52 HP ProLiant DL380 G6 machines interconnected by 10Gbps Ethernet via a 52-port Cisco Nexus 5020 switch. Every server is equipped with two Intel Xeon E5520 processors for a total of 8 physical CPUs (16 logical CPUs counting Hyper-threads) running at 2.26 GHz. An individual machine holds 24GB of ECC RAM and two hard disk controllers with 8 500GB SATA hard drives attached each. In total the test bed provides 1.248 TB of main memory and 416 TB of secondary storage. Networking is enabled by 10Gbps Myricom cards and secondary 1Gbps connections, both running unmodified Ethernet.

Figure 7 - Test bed hardware

The test bed provides a Debian Linux Kernel 2.6.32 environment. Tritonsort is based on the GNU GCC 4.3 build chain and facilities provided by the C++ Boost library. File access is handled by the Ext4 file system and networking is realized using the built-in Sockets interface.

4 Challenges and Approach

The following section presents motivation for and challenges in creating a resource-efficient system for Sort Benchmark. A concept for quantifying efficiency of software components and the integral system is described in order to provide an in-depth evaluation of the approach taken by Tritonsort.

4.1 Motivation

Sort benchmark at large-scale traditionally does not factor in the amount of hardware used. In the struggle to surpass previous records, the scale of computing clusters used in benchmarks grew into hundreds and thousands, sacrificing efficiency for scalability at all costs. Recent systems in the Gray and Minute Sort benchmarks do not utilize major quantities of hardware capacity. The scale of hardware used leads to a series of issues in network bandwidth, fault-tolerance and other areas that need to be dealt with. This leads to a further decrease in efficiency and requires even larger hardware assets. The result is an overhead in infrastructure and maintenance cost and added engineering effort to address the high complexity that could be avoided altogether.

Assuming software could use existing hardware more efficiently in the first place, excessive scaling was not necessary. Some systems (see Section 1) waste more than 90 percent of available disk bandwidth and CPU time. If these idle resources can be exploited by increasing system efficiency, the same task could be performed on a way smaller set of hardware.

The “Tritonsort” case-study aims at building a resource-efficient system that leverages form advantages of a compact set of commodity hardware. The Tritonsort prototype performs superior in Gray Sort Indy and Minute Sort Indy and runs on a significantly smaller set of commodity hardware than state-of-the-art systems.

At the current state performance of Tritonsort is limited by two tasks within the processing chain: intermediate data storage and internal sorting. In the two-pass sorting approach taken by Tritonsort intermediate data storage slows

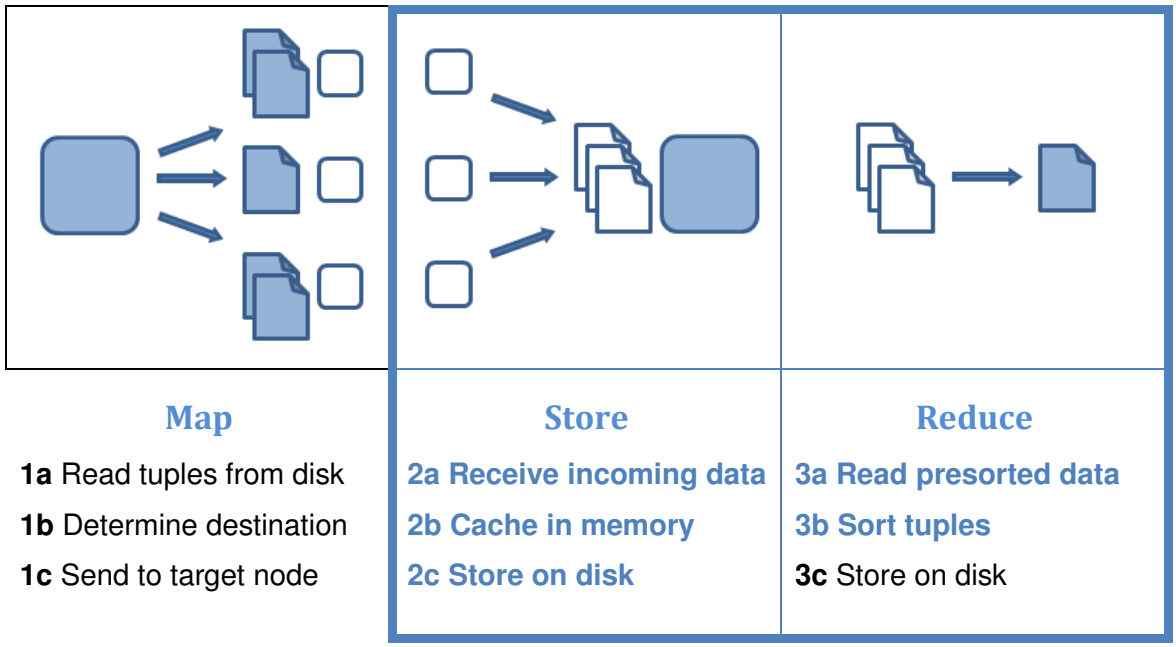
down the primary pass while internal sorting limits throughput of the second pass. Assuming both bottlenecks can be resolved without adding more hardware resources, system efficiency could be increased.

Hence, this paper focuses on these two aspects of in Tritonsort's subsystems for disk I/O and internal sorting. Both elements require careful design, as they have fundamental impact on overall system performance and efficiency. The realization of resource efficient components is performed as a four step process: first, metrics and ways to measure "performance" and "efficiency" are defined. Second, existing systems are compared in terms of architecture and efficiency using installed metrics. Third, suitable designs for disk I/O and internal sort subsystems are created. Fourth, the design is implemented and evaluated by comparison to state-of-the-art systems by benchmark results and metrics.

The core contributions of this paper are development of software components for intermediate data storage and internal sorting and an in-depth analysis of resulting resource efficiency of Tritonsort and state-of-the-art systems in large-scale Sort Benchmark.

4.2 Challenges

Tritonsort aims at achieving resource-efficiency. This requires balanced use of available hardware capabilities in each stage of the pipeline. Also, interaction and parallelism between stages need to be accounted for.



Map	Store	Reduce
1a Read tuples from disk	**2a Receive incoming data**	**3a Read presorted data**
1b Determine destination	**2b Cache in memory**	**3b Sort tuples**
1c Send to target node	**2c Store on disk**	**3c** Store on disk

Table 7 - Challenges overview

4.2.1 Map

The map stage reads input data from disk and distributes tuples to all participating nodes. It is active in parallel with the Store stage. Main challenges are computational and memory efficiency.

1a Access to the input data should be possible at maximum disk performance. If there is backpressure from other pipeline stages, performance must not be decreased further.

1b A tuple's destination bucket (and node) is determined before sending. This process must be faster than disk and network transfer to avoid bottlenecking. Map and Store stage are active in parallel, so memory consumption of the former may reduce cache (2b) and write performance (2c) of the latter. Also, the distribution mechanism must ensure that the contents of each bucket fit into memory for internal sorting (3b) in the reduce stage.

1c All tuples need to be transferred over the network. The small size of tuples and the distribution of key values may reduce network throughput when transmitted individually.

4.2.2 Store

The store stage receives tuples incoming from the network and stores them in buckets on intermediate disks. It is active in parallel with the Map stage. Main challenges are memory and disk I/O efficiency.

2a Data arriving from the network is stored in the according bucket. Data is incoming from multiple sources concurrently, so consistency needs to be ensured by synchronization. These measures should not affect network and write performance negatively, however.

2b A caching layer is introduced as purely random access to different locations on disk reduces throughput (3a). Tuples are expected to arrive at uniform rate for all buckets co-located on a single node, but each bucket should be accessible sequentially in the end. The cache needs to use available memory efficiently, must support synchronization measures (2a) and must be capable to deal with disk I/O underperforming network I/O.

2c Data stored in the cache needs to be written to disk continuously. To maximize write performance, random access to disk must be minimized. Still, the contents of each bucket should be stored in a single continuous file on disk to maximize read performance (3a) for the reduce stage.

4.2.3 Reduce

The reduce stage reads and sorts intermediate data. It becomes active on each node locally when incoming data transfer is completed. Main challenges are computational and memory efficiency.

3a Intermediate data should be accessed at maximum disk performance. Depending on the layout of bucket data on disk this might involve different access patterns.

3b Contents of each bucket are sorted in memory. Sorting a single bucket need to be faster than reading and writing its contents to avoid bottlenecking. Also, the implementation's memory efficiency determines the maximum bucket size. This indirectly affects mapping (1b), write performance (2b) and read (3a) performance as it determines the overall number of buckets needed in the system to store the total amount of data.

3c Each bucket is written to disk again. Data is expected to be written sequentially and well-performing. Efficient memory usage could improve the maximum bucket size for sorting (3b).

4.3 Contributions of the paper

This paper specifically focuses on the aspects of computational and disk I/O efficiency. Computational efficiency is relevant for internal sorting (3b) in the reduce step. Memory efficiency of caching (2b) helps increasing I/O performance (2c) and determines the number and maximum size of buckets for internal sorting. (3b) Synchronization measures should neither block Receivers (2a) nor Writers (2c) significantly. The efficiency of disk I/O is especially important when storing intermediate data (2c) as it involves non-sequential access patterns and might affect the read performance (3a) of the Reduce stage. Hence, contributions to the Tritonsort prototype can be divided into two parts, disk I/O and internal sorting.

In addition the paper provides a concept for evaluating resource efficiency of large-scale systems in the context of Sort Benchmark and compares state-of-the-art systems holistically with respect to efficiency in disk I/O and computation.

4.4 Evaluation

The prototype is evaluated in the 100TB Gray sort Indy and the Minute sort Indy category. The results are analyzed further using different metrics, such as average throughput per minute, and compared to state-of-the-art systems.

4.4.1 Measuring performance

Overall system performance is quantified by sort benchmark standards for Gray sort Indy and Minute sort Indy. Gray sort compares systems based on data sorted per minute metric (in TB/min). The number is obtained by dividing the total amount of input data by the total time required for processing (first node starting until last node completing). The metric uses fixed input data size and measures a variable amount of time. Minute sort uses the amount of input data sorted as metric (in TB). A system must be able to complete sorting a set of input data in less than 60 seconds in average for 15 consecutive benchmark runs.

For both benchmarks the amount of input data can be predetermined exactly by generating a fixed number of records across nodes and disks. Time measurements are performed on a single head node that sends a notification message to all nodes on startup and waits until all nodes report their task completed. The total time indicated by the head node is pessimistic as it includes network delays in addition to the runtime. This overestimation is at scale of milliseconds however, and suitable for timing benchmark runs at minute or hour scale.

Subsystems are benchmarked using smaller single-node benchmark setups. The performance of disk access and internal sorting is quantified using a fixed input size and measuring time passed to complete the operation (in MB/s). Test runs are repeated multiple times to obtain average values that factor out random fluctuations.

4.4.2 Measuring Resource efficiency

The quantification of efficiency is more complex than performance measures. A first look at sort benchmark provides two measures of efficiency: cost, addressed by penny sort and energy, addressed by Joule sort.

The latter, Joule sort is suited to estimate energy efficiency of systems. The number of records sorted per joule consumed power is indicative for a system's energy efficiency, ignoring consumption of additional infrastructure. For industrial applications, energy consumption directly translates into costs for power supply and cooling, but these costs are only one aspect of economic consideration. For example, the use of SSDs saves power compared to disk drives, but requires a significantly higher initial investment.

Penny sort directly focuses on hardware costs, but its definition inherently results in systems of small scale. The comparison of large-scale systems in terms of cost is hampered by the unavailability of accurate information. It is possible to estimate costs as system hardware is known and they are built from commodity hardware, but the resulting numbers are rough and valid for comparison of orders of magnitude only.

Anderson and Tucek (8) indicate that there are additional aspects of system efficiency. They include the former measures, cost and energy, and add computational, memory, storage, I/O, programmer and management efficiency. In the context of sort benchmarks some of them are suited for comparison, while others are difficult to address or quantify. The authors use MB/s per node, MB/s per core, MB/s per disk and byte/s per dollar metrics to compare system efficiency. Their results show a gap in efficiency between small and large scale applications, but underestimate the difference. For example, the average read/write bandwidth per disk is not fully representative for I/O efficiency as a system may saturate disk bandwidth but access data on disk more often than necessary for external sort. For example, DEMSort employs another disk access phase in addition to two pass sorting. (22)

The default metrics of Sort Benchmark, "amount of data sorted" (Minute sort Indy) and "total throughput per minute" (Gray Sort Indy) are applied to attain quantitative results for the integral system. Resource-efficiency in terms of computational expense, memory and I/O throughput is evaluated by comparison to existing systems with comparable hardware. Proposed efficiency metrics from Anderson and Tucek, "throughput per node", "throughput per

core", "throughput per disk" and cost-based "throughput per dollar" are used, although the latter can be based on estimations only.

For evaluation of resource efficiency this paper suggests a relative measure of system properties. This allows comparison of real-world resource efficiency of two systems without relying on performance numbers under optimal conditions from manufacturer whitepapers. At the same time an in-depth comparison at the level of different hardware labels and product revisions is unsuitable until exact and complete information about cluster hardware and cost is available for large-scale systems in Sort Benchmark. Thus, the disadvantage of this approach is an undifferentiated perspective on hardware and the impossibility of determining resource efficiency on an absolute scale.

In this paper a quantitative comparison between systems is based on the following variables and metrics. Also, the evaluation of subcomponents and alternative implementations thereof makes use of these measures.

1. **Hardware**

Hardware metrics quantify basic properties of the hardware used by a system. This includes the number physical machines, the total number of CPUs and the total number of hard drives. Estimated costs focus on the initial investment for cluster hardware without maintenance.

- **Cluster nodes**
  $nodes$: Number of physical machines in the cluster
- **CPU cores**
  $cores$: Number of physical CPU cores in the cluster, not counting HyperThreads.
- **Hard drives**
  $disks$: Number of physical hard drives in the cluster, independently of operating system or file system view and RAID configuration. The expression "disk" is used as a synonym when referring to Solid State drives.
- **Estimated hardware costs** (in USD)
  $cost$: Approximation in orders of magnitude of total costs of hardware

components derived from (8) by mapping “bytes/s/$” to $\frac{throughput}{cost}$. If a paper provides exact information on hardware costs, this data is used instead.

2. Benchmark

Benchmark metrics are derived from Sort Benchmark and provide information about benchmark type and scale and a system’s specific benchmark runtime and performance.

- **Gray Sort performance** (in TB/min)
  $performance_{Gray}$: Gray Sort benchmark performance as used by Sort Benchmark.
- **Minute Sort performance** (in GB)
  $performance_{Minute}$: Minute Sort benchmark performance as used by Sort Benchmark.
- **Input data** (in MB, GB or TB)
  $data$: The total amount of input data processed during a benchmark run. Sort Benchmark uses a ratio of 10% for key information and 90% for payload in generated input data.
- **Runtime** (in seconds)
  $time$: Total runtime of a system per benchmark.
- **Throughput** (in TB/min or MB/s)
  $throughput$: System performance independently of specific benchmark metric.

$$throughput = \frac{data}{time}$$

3. Relative Throughput

Relative throughput is used to quantify system performance on component level without relying on an absolute baseline. These metrics correlate benchmark performance and hardware properties via throughput numbers.

- **Throughput per node** (in MB/s)

  $throughput_{node}$: Benchmark throughput per physical machine.

$$throughput_{node} = \frac{throughput}{nodes}$$

- **Throughput per core** (in MB/s)

  $throughput_{core}$: Benchmark throughput per physical CPU core.

$$throughput_{core} = \frac{throughput}{cores}$$

- **Throughput per disk** (in MB/s)

  $throughput_{disk}$: Benchmark throughput per physical hard drive. This metric divides benchmark performance to hard drives, the actual amount of data transferred from/to hard drive interfaces is a multitude of this value in general.

$$throughput_{disk} = \frac{throughput}{disks}$$

- **Throughput per cost** (in bytes/s/$)

  $throughput_{cost}$: Benchmark throughput per USD hardware costs. The conversion of MB/s to bytes/s in throughput is performed for ease of visualization and reference to literature.

$$throughput_{cost} = \frac{throughput}{cost}$$

4. Relative Hardware Scale

Relative hardware scale relates two systems in terms of hardware components. These numbers are mainly used to normalize numbers for relative resource efficiency. Estimated costs incorporate this information holistically, and hence, the calculation of cost efficiency is not affected by these ratios.

- **Relative cluster nodes**

  $ratio_{nodes}$: Number of nodes in system A compared to B.

$$ratio_{nodes,A,B} = \frac{nodes_A}{nodes_B}$$

- **Relative CPU cores**

  $ratio_{cores}$: Number of CPU cores in system A compared to B.

$$ratio_{cores,A,B} = \frac{cores_A}{cores_B}$$

- **Relative hard drives**

  $ratio_{disks}$: Number of hard drives in System A compared to B.

$$ratio_{disks,A,B} = \frac{disks_A}{disks_B}$$

5. Relative Resource Efficiency

Relative resource efficiency is used to compare efficiency of two different systems. Each metric is based on the ratio between systems' throughput for a specific component and is normalized by the relative amount of hardware. An exception is Relative Cost Efficiency which relates cost efficiency numbers that are already normalized by their dependence on holistic cost estimations.

- **Computational Efficiency**

  $efficiency_{cores}$: Normalized throughput per CPU core of system A compared to B.

$$efficiency_{cores,A,B} = \frac{throughput_{node,A}}{throughput_{node,B}} * \frac{1}{ratio_{nodes,A,B}}$$

- **Disk I/O Efficiency**

  $efficiency_{disks}$: Normalized throughput per hard drive of system A compared to B.

$$efficiency_{disks,A,B} = \frac{throughput_{disk,A}}{throughput_{disk,B}} * \frac{1}{ratio_{disks,A,B}}$$

- **Cost Efficiency**

  $efficiency_{cost}$: Benchmark throughput relative to system cost of system A compared to system B.

$$efficiency_{cost,A,B} = \frac{\frac{throughput_A}{cost_A}}{\frac{throughput_B}{cost_B}} = \frac{throughput_A}{cost_A} * \frac{cost_B}{throughput_B}$$

6. **Metrics used for evaluation**

The evaluation of the Tritonsort prototype is performed using benchmark performance, computational efficiency, disk I/O efficiency and cost efficiency. Subcomponents for internal sorting and disk I/O are evaluated using the throughput metric.

- **Benchmark performance** ($performance$)
- **Computational Efficiency** ($efficiency_{cores}$)
- **Disk I/O Efficiency** ($efficiency_{disks}$)
- **Cost Efficiency** ($efficiency_{cost}$)
- **Throughput** ($throughput$)

4.5 Architecting for Efficiency

Resource efficiency in computing is a broad topic, especially for benchmark applications. With the target benchmarks “Gray Sort” and “Minute Sort” in mind the thesis focuses on computational and I/O efficiency.

Computational efficiency is addressed mainly by the choice of internal sort algorithm. Optimization of the actual implementation is based on the findings of (11)(18)(26) and focuses at inexpensive CPU instructions and memory access. Additionally, the interaction between operating system and the application is taken into account to increase overall application efficiency, e.g. by avoiding redundant memory allocation.

Efficiency at the disk I/O interface is achieved by using straight two-pass algorithms for the “Gray” configuration and one-pass for “Minute”. This ensures data on disk to be accessed the least amount possible (15) and is in line with the findings of (18) regarding Minute sort. Also, disk access and manipulation of data in memory are interleaved to avoid idle times at the I/O interface. Although, Tritonsort relies on the operating system for disk access, it provides an application specific implementation of write caching to circumvent issues caused by sustained rate parallel file access.

Computational and I/O efficiency also depends on memory efficiency for buffering and caching. In terms of memory efficiency two mayor issues are

tackled. First, sort algorithms performing in linear time are not capable of operating in-place in general. That may effectively halve the potential size of each data partition generated by the map stage and leads to an increased number of partitions. This in turn impacts I/O performance as additional files are co-located on each physical disk. To avoid this effect, the implementation leverages from tag-based sorting and performs in-place permutation of the actual input data. Secondly, the write cache is required to make optimal use of available memory to achieve high I/O performance. This includes using light-weight metadata structures and preventing memory fragmentation.

4.6 Design Constraints

The design of intermediate data storage and internal sorting components is constrained by three factors. These are benchmark rules, operating environment and Tritonsort's pipeline architecture.

The most fundamental design constraint is imposed by the dataset size required for the Gray Sort benchmark. 100TB of data do not fit into main memory of state-of -the-art systems, and therefore, require memory external sort to be performed. (With exceptions that come close with regard to main memory (23)) Also, this implies a runtime of multiple hours accompanied by high utilization of I/O interfaces.

The benchmark categories "Gray Sort Indy" and "Minute Sort Indy" both use the same format of input data. The data consists of a number of fixed-length binary records with a fixed key and payload portion. The length restriction simplifies implementation of buffers, iterators and sort algorithms. For example, offsets can be calculated using simple indexing and records can be swapped in-place by the sort implementation. Also, the distribution of key values can be assumed uniform.

Additional rules require that data is never compressed when passing through network or disk interfaces. Also, when using replication the number of replica for input files has to match the number for output files. This has two implications for component design. First, disk I/O performance can be optimized by minimizing hardware and OS overheads, e.g. seeking or cache lookups. Secondly, internal

sorting needs to move the full amount of data around memory, making the memory bus a potential bottleneck.

The hardware environment consists of a homogenous series of rich nodes. Hence, a distinction between specialized types of nodes is not necessary. Also, the relatively small number of nodes allows a flat network topology with all nodes connected to a central switch. A first estimation of I/O interface bandwidth suggests that disk access should become the main bottleneck at a maximum of 800 MB/s reading and writing per node. The 10Gbit Ethernet is capable of transferring a maximum of 1250 MB/s full-duplex, and hence, should not affect throughput negatively.

Tritonsort's pipeline architecture requires each stage to be designed as a number of threaded "workers" that receive one unit of work at a time. A worker may process or store that unit and pass a modified unit to the following stage of the pipeline. Workers queue up work units before processing and queue operations rely on locking for consistency. This design favors components to operate on batches of data, in order to keep overheads to a minimum.

4.7 Limitations

The following section addresses the limitations of this paper. Resource-efficiency is a major topic in, however, this is limited to technological aspects. Economical aspects are approached from a hardware investment point of view, ignoring issues longing from programmer and management efficiency. This also extends to considerations about fault-tolerance and failure-redundancy that are not required at prototype scale, but are obligatory in real-world deployments.

Programmer and Management efficiency is not addressed as it seems hard to define and capture. Time requirements for preparing and running processing tasks may be nullified by automated scripts, development times of pipeline stages and optimization highly depend on task complexity and skill of the programmer, and so on.

The paper mostly ignores bandwidth and latency changes in networking when scaling the number of disks per node or the number of nodes in the cluster by significant amounts. The current hardware test bed provides a 10Gbps

connection between nodes that can easily handle the throughput generated from eight input disks - 8x100 MB/s disk input stream versus 1280 MB/s network bandwidth. When additional disks are added network bandwidth for individual links will become a bottleneck, while an increase in the number of nodes will require additional switches or port multiplier hardware that limit throughput between certain partitions of the cluster.

One of the main arguments supporting the use of pure MapReduce is trivial tolerance in the presence of failures by replication. In case of Tritonsort failures can manifest themselves as soft read errors and disks failures as well as nodes and whole racks going down due to network or power issues. While disk problems can be handled by RAID with moderate performance trade-offs any failures at the scale of nodes, racks or switches can only be handled by replication across multiple machines and locations. The aspect of fault-tolerance is not covered by this paper and any solution based on redundancy will likely lead to a decrease in efficiency numbers. However, (8) argue that the low efficiency of existing systems is a major source of critical failures due to comparably task runtimes and additional orders of magnitude in the number of hardware components used.

5 Contributions

The following section describes work contributed to the code base of the Tritonsort prototype. The structure focuses on two aspects of Tritonsort's distribution-sort based pipeline: storage of intermediate data on disk and internal sorting.

Storage of intermediate data balances write and read performance of (2c) and (3a) to achieve maximum average performance. This is achieved by caching data pending for write efficiently (2b). The implementation of internal sort addresses (3c) and balances memory overheads and computational cost.

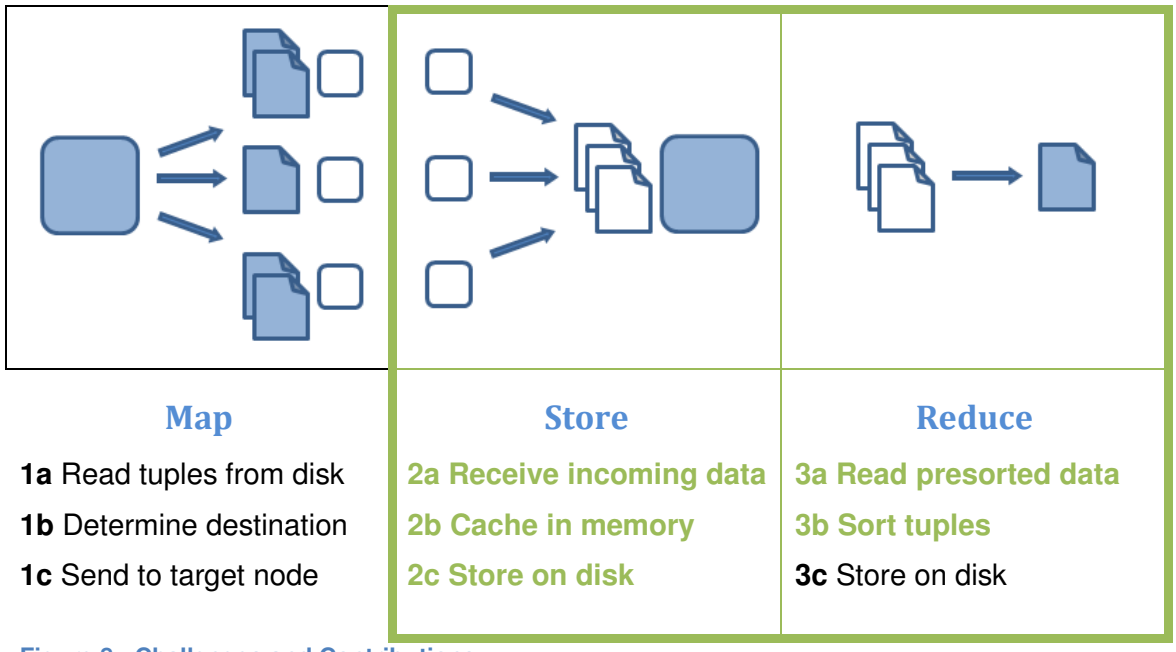


Figure 8 - Challenges and Contributions

The section first presents the disk I/O layer which represents a core component of the Store stage. Interleaving of network and disk I/O, write caching and file management are presented is this part. The second part takes a closer look on the sort algorithm and the implementation of tag-based sorting and in-place permutation of input data.

5.1 Data persistence

The data persistence section addresses challenges in the Store stage (2b) and (2c) by introducing a disk I/O layer with application specific write caching. This also affects read performance in the Reduce stage of the pipeline (3a).

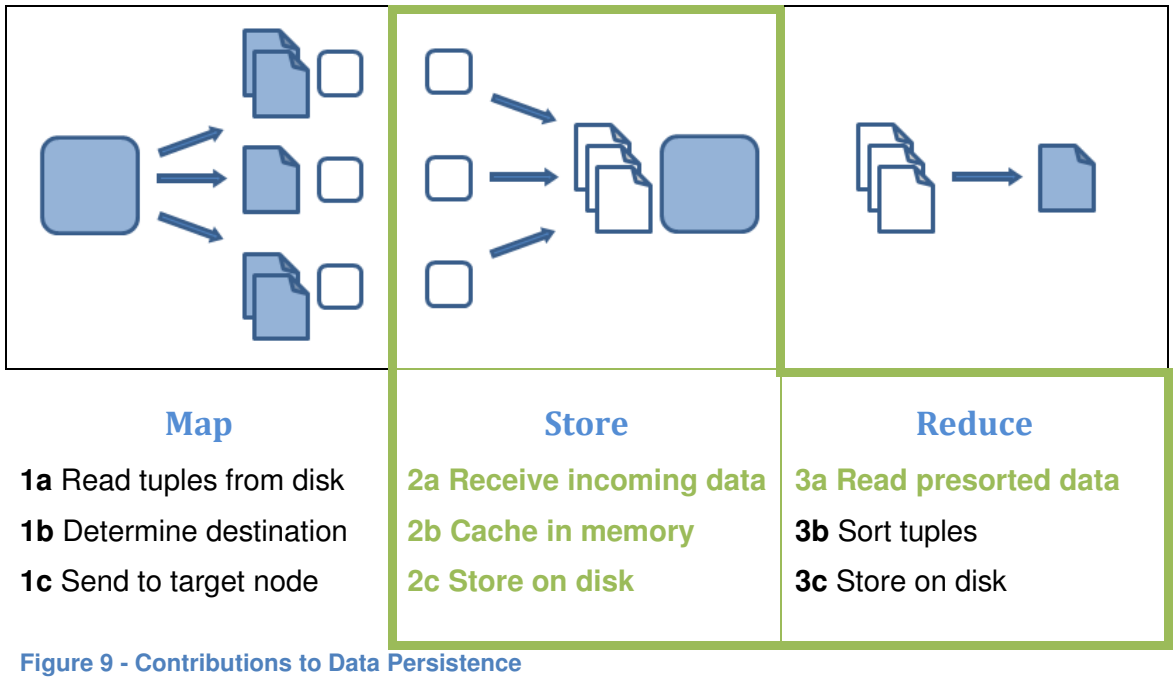


Figure 9 - Contributions to Data Persistence

The dedicated Store stage of the pipeline collects incoming data from the network via Receivers. These worker threads receive sets of tuples. Each batch of tuples is preceded by metadata, determining the destination bucket and batch size. The receiver mutually locks the indicated bucket, appends tuples to the bucket buffer and releases access again. As multiple buckets are linked to each physical disk, a Writer per disk continuously processes buckets and frees up buffer memory again.

There are two main challenges when maximizing the throughput at the Store stage of the pipeline: maximizing writer throughput and eliminating receiver stalls. The writer has to deal with random disk access patterns as multiple bucket files are co-located on each physical disk. In order keep seek-related overheads low, the size of sequential writes has to be maximized. The write size depends on the number of co-located files and the amount of bucket data available in memory. Receiver stalls show up due to mutual exclusion, either due to exclusion between multiple receivers or due to exclusion between writers

and receivers. In general, mutual exclusion is necessary to prevent data corruption during concurrent access.

In the first part of this section the approach to maximizing sequential write size is developed. This leads to a solution that can be extended to address issues related to pipeline stalls which are discussed in the second part.

5.1.1 Caching in external distribution sort

The system architecture builds on memory external distribution sort that introduces a fundamental mismatch between access patterns to data during the map stage and the reduce stage. The following section describes its cause and two approaches to handle it.

For ease of representation, the example below focuses on the activity of a single disk. The behavior is comparable for multiple disks and machines performing parallel distribution sort. Although, portions of data are sent and received over the network and multiple input files are read in parallel, the store and reduce stage behave similarly. Data destined for the same bucket is always stored on the same physical drive by Tritonsort.

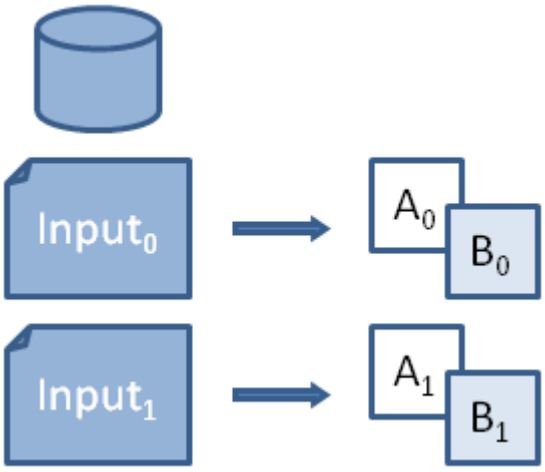
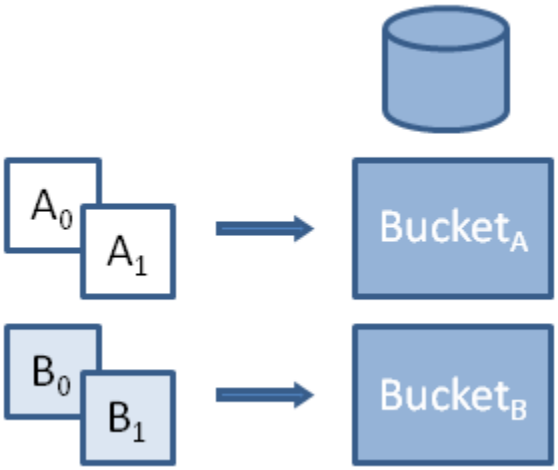


Figure 10 - Processing input data

Assuming uniformly distributed key values in the input data, any input file contains a certain portion of tuples for every single target bucket. In the map stage input files are processed serially, and hence, produces data destined for every single bucket within the system at about an even rate.



The reduce stage in contrast processes buckets serially and requires all data of a single bucket to be present at once. The concatenation of sorted buckets is guaranteed to be sorted globally only, if each tuple within the bucket's key range is taken into account during internal sort.

Figure 11 - Processing intermediate data

The store stage of Tritonsort's pipeline is in charge of collecting data from the mapper for buckets in parallel and delivering complete buckets serially to the reducer. This has two implications: first, map and store stage must process all input data before the reduce stage can be started. Second, data persistence involves random disk access at some point as the total amount of data does not fit into main memory. The first issue is addressed by the pipeline controller and is not of further interest. The second poses the question whether random access should be performed during writing or reading.

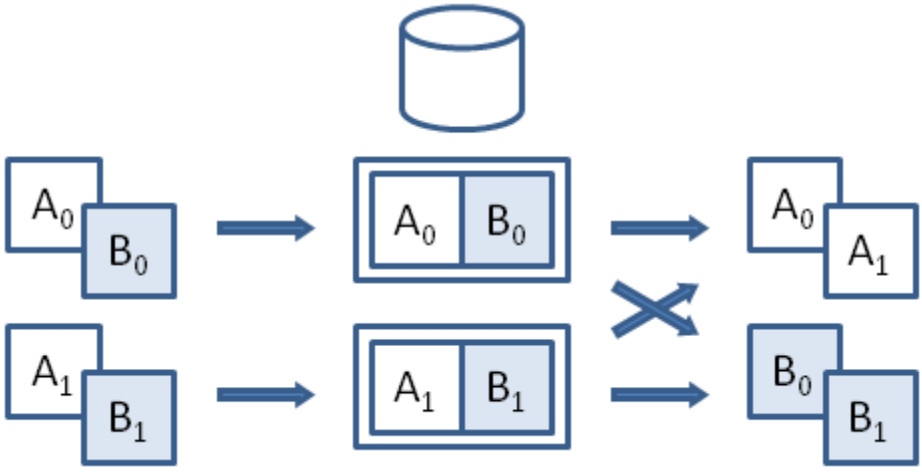


Figure 12 - Accessing intermediate data with random read

In case of random read access, data from the mapper is stored in intermediate files that represent partially sorted runs. This resembles the behavior of merge sort approaches, with the tuples being sorted by a portion of their key value only

(depending on mapper configuration). When the reducer accesses data of a single bucket, the according chunk in each file has to be accessed.

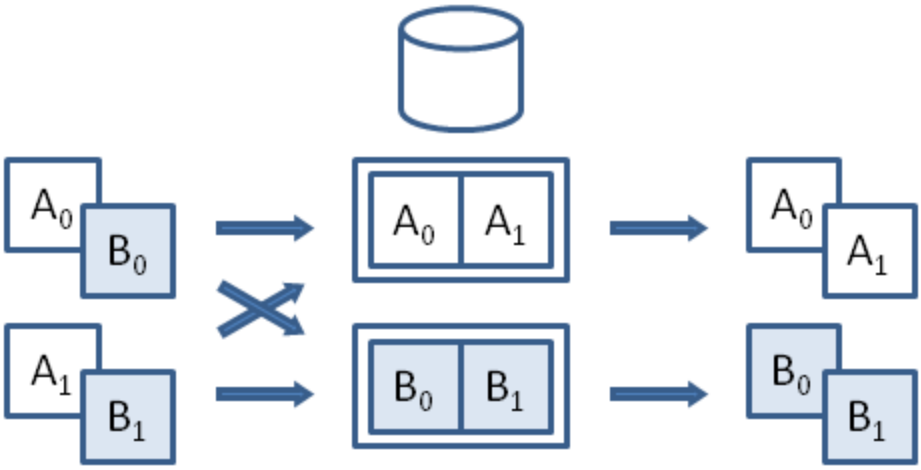


Figure 13 - Accessing intermediate data with random write

The alternate approach is random write access. Each bucket is represented by an intermediate file and incoming data is appended to the according file. The reducer accesses a single logical file at a time when processing buckets. Compared to the random read approach this allows additional control over physical placement of data through file-system options, e.g. by pre-allocating file space. Also, implementation of a custom read-ahead policy is difficult and requires modification of kernel and file-system while application-specific write caching can be implemented with relative ease.

In either way however, non-sequential data access is required what causes disks to introduce seeks in order to access required locations. In terms of throughput the time spent seeking is lost as it is neither used for writing or reading data. The common way to work around this issue is buffering, which is inherently limited by the amount of memory and disk cache.

Tritonsort employs random write access to re-order intermediate data. Caching is mainly performed in main memory as disk controllers and disks typically hold less than a second worth of data incoming from the network. Write-caching is usually handled by the operating system and file system, but did not perform well in an a-priori experiment. This did not come unexpectedly due to the

developers of NOW-Sort reporting similar issues for comparable workloads years ago.

Hence, tuning the efficiency of write buffering in Tritonsort is the central task when optimizing the Store stage. Data received by a node is almost uniformly distributed across all buckets while individual buckets are processed serially by writers repeatedly appending data to its corresponding file on disk. While a bucket is being processed by a writer, additional data might be received which leads to a potential mutual exclusion issue. So, in order to achieve optimal throughput the size of each sequential write has to be maximized while blocking times for receivers and writers have to be minimal.

5.1.2 Caching Buckets

The first question that arises when dealing with disk access is whether to rely on default file system behavior or to handle caching oneself. The figure below shows disk write performance for typical workload generated by Tritonsort's processing pipeline. Three implementations of the writer stage are compared - a primitive writer relying on buffered I/O, a writer using unbuffered I/O and manual double buffering and the final design using unbuffered I/O and dynamic buffer sizes.

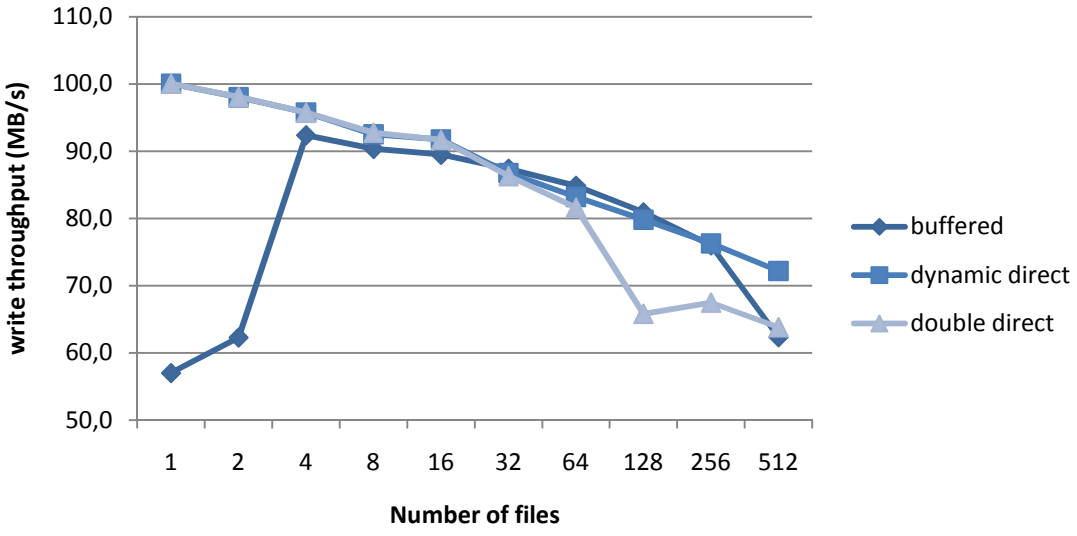


Figure 14 - Impact of caching on write performance

The file system still handles file-space allocation, but most operating system and file system caches are circumvented using Linux' Direct I/O interface.

Buffered I/O and double buffering show comparable performance characteristics, with unbuffered I/O having the advantage. The writer using dynamic buffer sizes for buckets is slower for small numbers, but gains a significant advantage as the number of files increases with growing amounts of input data. Hence, manual caching brings a notable boost in write performance.

A first approach to manual caching is the use of a single buffer per bucket for collecting incoming data. Whenever a receiver gets data from the network, it is appended to a buckets buffer. When the buffer is full it is handed to the writer and the receiver waits until space is available. The writer appends the buffer to a file, clears the buffer and returns it for reuse.

This solution is simple, but has a major disadvantage. Every time a writer processes a bucket, any receiver accessing that bucket blocks and a potential stall is induced into the pipeline. If receivers block long enough, sending nodes may become blocked too any slow down the rest of the system. With a single buffer concurrent modification of data in a bucket could lead to data corruption, thus requiring mutal exclusion among receivers as well as between receivers and writer. Receivers write small amounts of data to many different buckets in rapid succession while writers block a relatively long time on a single bucket when writing its contents to disk. This leads to an almost synchronous pipeline behavior between receivers and writers with reduced performance.

The figure below shows part of a visual representation of worker activity for receivers and writers for a benchmark run using the above approach. Each row represents the time line for a single worker thread. Colored areas indicate activity while blank areas show inactivity.

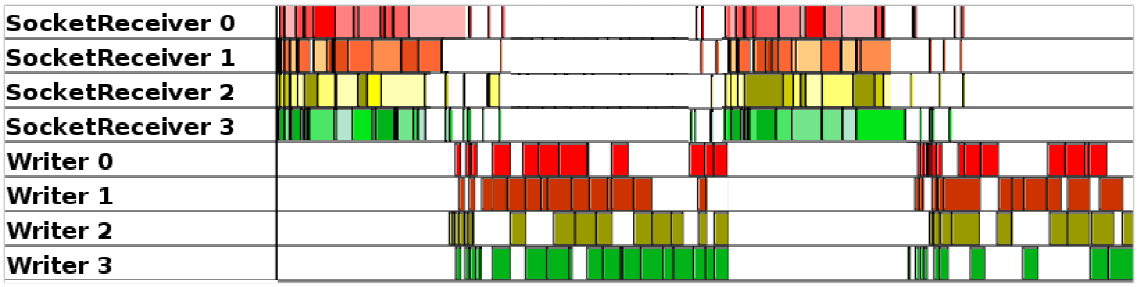


Figure 15 - Writer and Receiver stall

A receiver fills up a buffer and hands it to the writer. The writer starts processing and exclusively locks the buffer. During that period of time almost all receivers will encounter data destined for that bucket and block too. When the writer finishes, receivers start filling in data again until another buffer gets full. However, the writer cannot be active during this period of time either, as it's work queue is empty. The consequence is a drop in receiver and writer performance.

The impact of mutual exclusion can be reduced by adding a dedicated spare buffer that can be swapped in by the writer when a buffer starts being processed. This interleaves disk and network I/O by allowing receivers to continue collect data even though the buffer originally underlying the bucket became full during the process.

Taking a closer look on the behavior of receivers and writers at runtime shows that buckets fill up at almost the same rate. Even though a receiver can now continue writing to a bucket when it has been passed to the writer, it may block on another bucket while the writer is still busy processing the first one. As buckets fill up at the same rate that pattern repeats for every bucket causing repeated pipeline stalls on the receiver side. Thus, the spare buffer reduces blocking to a certain extent, but does not prevent it entirely.

A possible solution to eliminate receiver blocking is the introduction of a spare buffer per bucket. Though, a receiver might still block on a bucket when both buffers are filled up and pending a write, the pipeline basically runs at the speed of the writer stage. The figure below represents worker activity over time again, with the notable difference of interleaved receiver and writer activity.

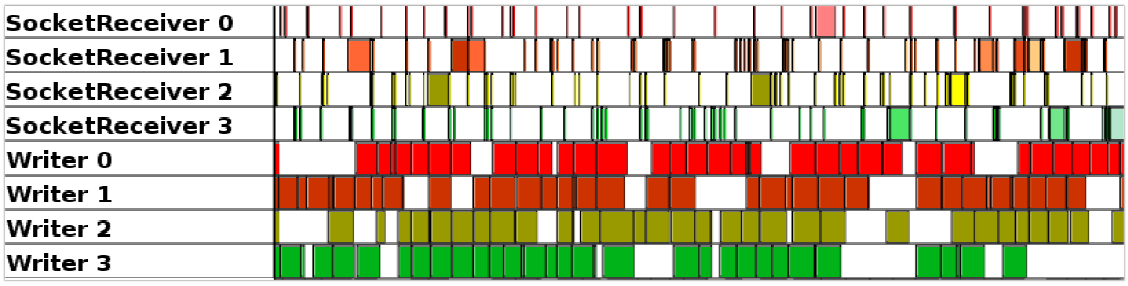


Figure 16 - Writer and Receiver decoupled

The use of double-buffering conflicts with the initial motivation of maximizing write size, however, as it halves the amount of data available per sequential write.

When investigating the fill and drain rates of buffers associated to a single bucket it shows that half the available memory is not actually occupied by tuple data during runtime. All buffers held by the receivers fill continuously over the period of time that is required by the writer to transfer the contents of an equally large set of buffers to disk. Buffers are freed up by the writer and returned to their according buckets. In the Gray sort configuration, the newly freed memory is sufficient to collect a large portion of incoming data for a single bucket. Even though, most parts of the memory region are unused for a period of time, they cannot be reused for a different bucket that requires additional storage space to prevent stalling.

Sequential write size can be increased by a factor of two in this context using more fine-grained buffer management. Instead of allocating fixed amounts of memory per bucket, memory could be shared across all buckets related to a single writer. A single memory pool allocates a series of small buffers up to a given limit and provides these on demand. A bucket can request additional buffer space when a receiver fills in data, and analogously, the bucket returns buffer space to the pool when it has been processed by a writer. Although, the bucket internally encapsulates any interaction with the buffer pool it still increases complexity at the writer side. The transfer of a single bucket may involve data being collected from multiple different locations in memory, but data can be written to disk sequentially.

Resizing bucket buffers dynamically helps memory efficiency too. When a writer processes a single bucket it frees up multiple smaller chunks of memory. As the total amount of memory available to buckets is shared throughout the pool, heavily populated buckets can expand using one of these chunks while empty buckets do not drain the pool unnecessarily. This allows different buckets to effectively use different amounts of memory and dynamically adapt to changes in demand.

The use of a memory pool allows tight control of memory allocated and prevents losses due to memory fragmentation. Also, multiple writer stages are able of increasing concurrency of pool operations by using a separate pool for their corresponding set of buckets. Loss due to memory fragmentation is prevented by using a fixed buffer chunk size.

There is a trade-off for increased management activity, however, that mainly affects computational efficiency. Firstly, mutual exclusion is required at the level of memory pool operations. Multiple buckets are written to and consumed from at the same time by receivers and writers, potentially causing different resize operations to overlap. Secondly, the number of pool operations is inversely related to the buffer chunk size. When the buffer size is halved, the number of acquisition and return operations doubles. Also, the number of operations required by receivers and writers goes up with decreasing buffer size. Tritonsort typically uses a chunk size between 1MB and 4MB for bucket buffers, which represents a well-performing trade-off between flexibility and overheads.

5.1.3 Writing Buckets

Another substantial change in the pipeline architecture becomes necessary when transitioning from a single, continuous buffer to dynamic re-allocation model. Activity of the writer stage can no longer be triggered by receivers passing buffers to process as there is not any fixed limit to bucket size that indicates necessity of a write. The question arises, how to determine the necessity of a write and, in case multiple buckets require processing, how to determine the order they are handled in.

One solution addressing the missing size limit of buckets is the introduction of a user-define threshold value. When a receiver surpasses the limit while collecting data the according writer is notified and buckets are written to disk in the order they are enqueued. A receiver may continue writing data to the bucket even while it is pending a write as long as there is sufficient memory provided by the memory pool.

The fixed threshold has two main drawbacks, though. First, the threshold has to be tuned manually and determining the optimum involves several difficulties. If the threshold is too small, the average write size goes down, degrading overall

throughput. If the threshold is too large, buffer chunks may be used up before buckets are enqueued to writers either degrading performance by blocking several receivers or in worst case causing a deadlock due to buckets not reaching the threshold at all. Second, the use of a fixed threshold value prevents dynamic changes in writer behavior when fluctuations occur in the amount of incoming data. A temporary decrease might delay buckets being enqueued eventually bringing writers to a stop even. When this is followed by a proportional increase the queue size at writer grows to higher than average levels. This in turn may cause receivers to block due to the memory pool running out of buffer chunks resulting in an over-proportional drop in inbound data transfer rate. The resulting behavior is unstable at runtime and repeatedly causes pipeline stalls at receiver and writer side.

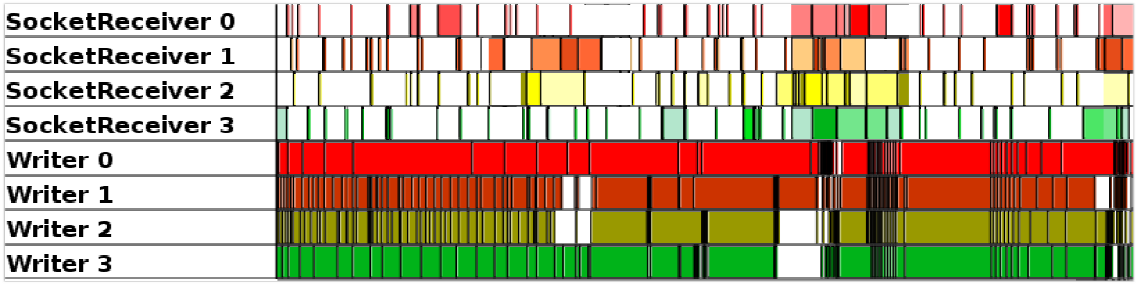


Figure 17 - Writer using fixed thresholds

Decoupling receiver and writer activity gets rid of the necessity for a threshold that triggers writer activity. Receivers read data off the network and store it into buckets as before, but do not longer care for notifying writers. Writers in turn become pro-active in terms of selecting buckets and writing contents to disk. This removes the push-based connection between both pipeline stages, eliminating direct communication. Writers become active in the system at the same time with receivers and continuously choose and process buckets. The writer stage is shut down when receivers finished work and there is not any pending data left in buckets. A new aspect introduced by this approach is the policy used for bucket selection. The policy impacts sequential write size as well as the ability of the writer stage to adapt to dynamically changing loads.

A simple policy selects buckets in a pre-determined order for processing. This ensures fairness of write scheduling and the size of sequential writes is constant, as is throughput. This can be realized using a pre-permuted ordering,

e.g. round-robin. Alternatively, buckets can be selected on-demand every time a writer gets ready for processing. By selecting the bucket containing the highest amount of data, the policy may optimize directly for sequential write size.

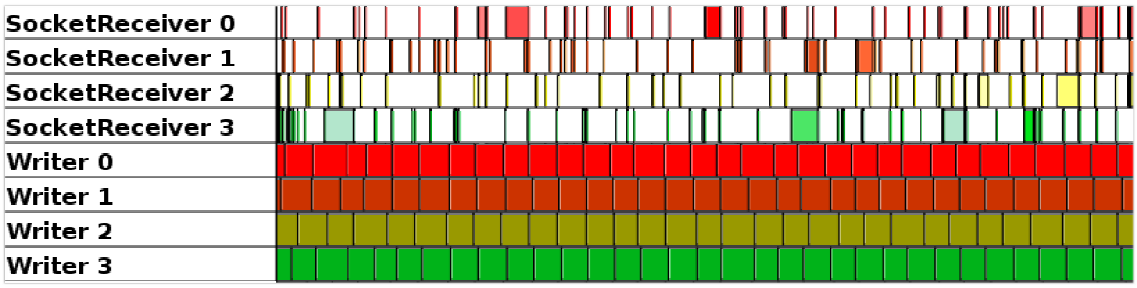


Figure 18 - Writer using demand-based scheduling

The implementation used by Tritonsort employs a demand-based selection pattern. Write performance benchmarks based on the dynamic buffer approach comparing round-robin and on-demand selection show a slight advantage for the demand-based approach. However, for large amounts of data as found in a 100TB Gray Sort run a significant difference couldn't be determined.

5.1.4 Conclusion

Disk access is a core factor of system performance and efficiency as the overall pipeline is mostly I/O bound. Although, commodity file systems limit control over physical file layout and write order, a number of optimization techniques are available.

First, inference between network I/O and disk I/O can be avoided by decoupling activity in the Writer stage from the Receiver stage. This allows both types of I/O to be overlapped. Secondly, write performance for high numbers of co-located files can be improved by implementing an application-specific write cache-layer. This adds complexity to the design, but gains stable and predictable disk throughput.

5.2 Internal sort

The internal sort implementation addresses the main challenge of the Reduce stage (3b) by providing a well-performing algorithm. The maximum buffer size supported by the reduce stage significantly affects I/O in (2c) too, as it implicitly determines the number of intermediate files.

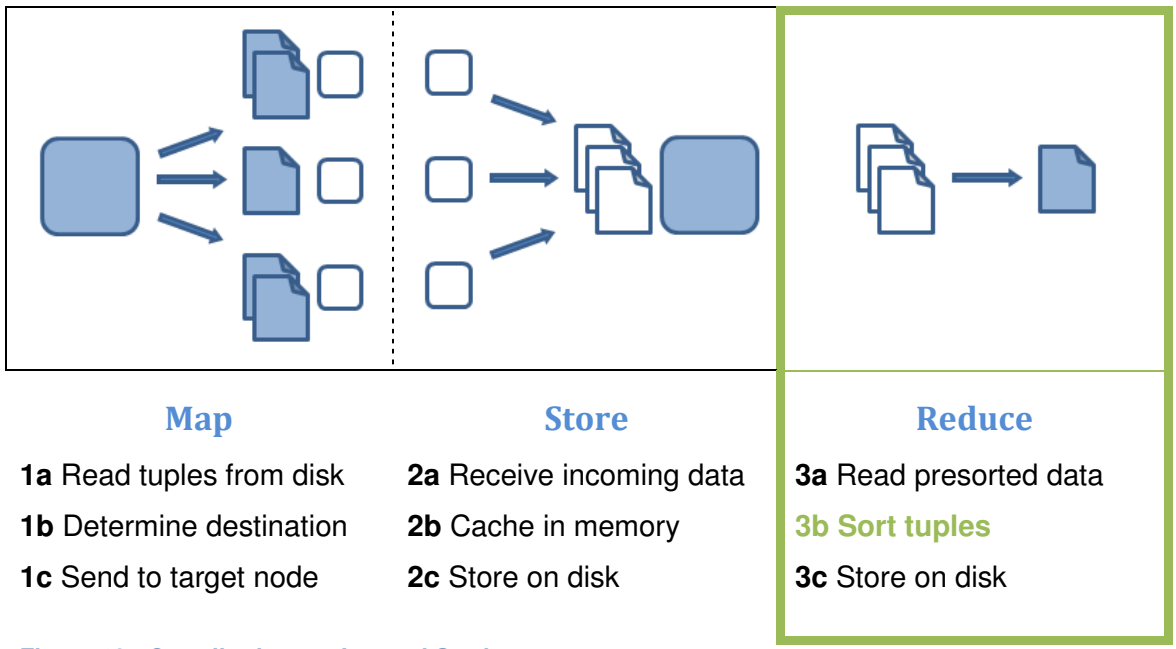


Figure 19 - Contributions to Internal Sorting

Internal sort is performed in the Reduce stage of the processing pipeline. The Gray Sort configuration reads stored buckets from disk in and sorts them. The Minute sort configuration applies internal sort directly to buckets stored in memory. In both cases internal sort is required to be faster than disk I/O to avoid bottlenecking.

Tritonsort uses a linear time Radix sort that operates on key tags instead of full tuples. The use of 16-bytes key & pointer tags reduces runtime compared to algorithms moving full 100-bytes tuples. Also, Radix sort is modified to permute input buffers in-place in order to spare memory. The tag sort approach was first implemented by (11) and applied to radix sort by (18). In contrast to NOW-sort the implementation used by Tritonsort does not limit the algorithm's internal buffer size. This allows sorting large buffers without introducing a merge step, but trades this for decreased cache efficiency.

The introduction of the specialized sort algorithm is necessary in the first place as the Quicksort implementation does not provide sufficient performance. The

figure below shows sort performance on the test bed hardware for a single physical processor operating on different buffer sizes.

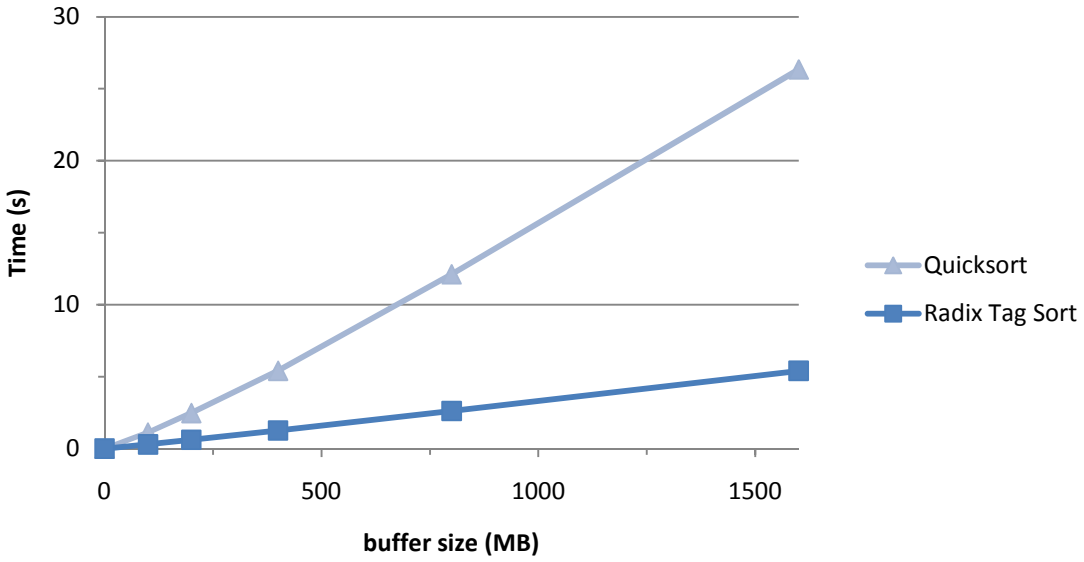


Figure 20 - Internal Sort runtime

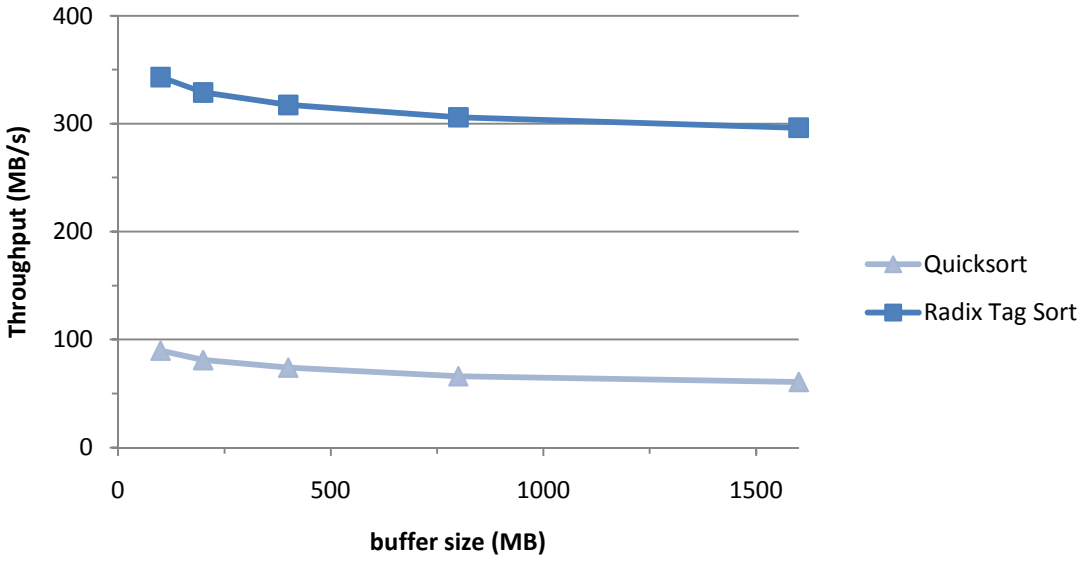


Figure 21 - Internal Sort throughput

In the Gray configuration, the test bed's 8 physical processors need to deliver output for 8 disks in parallel. Each bucket holds about 800 MB of data, what translates to 66,1 MB/s per core using Quicksort. Assuming an average throughput of 80 MB/s per disk, the system would lose 15-20 percent of potential I/O throughput. Using the Minute configuration the drop becomes significantly higher as all 16 disks are used for output in parallel. Even though,

Quicksort performance goes up as buffer size decreases to 100 MB, sorting would provide 700 MB/s and underutilize the potential I/O throughput of 1280 MB/s. Radix Tag Sort provides 4 times higher throughput per core than Quicksort, and hence, is capable of saturating available I/O bandwidth.

On a high level, the sort stage operates in three phases: tag extraction, radix sort and permutation. The first step performs a linear scan through the input data and constructs tags containing each tuples' keys and memory offsets. The tag buffer is then handed over to a conventional radix sort using an 8-bit radix or 256 buckets, respectively. Each round in radix sort consists of a counting step for building a histogram of key values, a step reallocating buffer space to buckets and the actual distribution step of tags to buckets. After 10 rounds each key-bit has been taken into account and the concatenation of buckets yields the final ordering of tags. The memory offsets stored within the tags are then used to create the lookup table required for permuting the input buffer in-place.

5.2.1 In-place permutation

In-place permutation is performed using a lookup table and an additional buffer capable of holding a single tuple. A lookup table maps destination offsets to source offsets with "get from" semantics. The algorithm uses the table to re-order items and solve dependency cycles.

The figure below illustrates in-place reordering of an input buffer of size 3, containing tuples "C", "B" and "A". In the example field 0 requires contents of field 2 to be in order. The numbers 0 to 2 represent indices in the input array and "buffer" denotes the external space used for moving and swapping elements. Finally, the "from" row represents the contents of the lookup table, mapping its value as source offset and its index as destination for move operations.

1.

	0	1	2	buffer
Tuple	C	B	A	-
from	2	1	0	-

The process starts by checking the first entry in the lookup table at offset 0. This returns "2", indicating that the contents of field 2 need to be moved to field 0.

2.

	0	1	2	buffer
Tuple	-	B	A	C
from	2	1	0	-

The algorithm then moves the contents of field 0 to the external buffer.

3.

	0	1	2	buffer
Tuple	A	B	-	C
from	0	1	0	-

Contents from field 2 are moved to field 0 and the table entry is marked done by setting source index equal destination index.

4.

	0	1	2	buffer
Tuple	A	B	C	-
from	0	1	2	-

Steps 1 to 3 repeat for field 2 and consecutive entries until the cycle is resolved. At completion of each cycle the contents of the external buffer are moved to the free region.

5.

	0	1	2	buffer
Tuple	A	B	C	-
from	0	1	2	-

In order to correctly reorder buffers that contain multiple cycles any remaining table entries are scanned and processed analogously until all entries are marked done.

Figure 22 - Radix Sort in-memory reordering

The algorithm has linear run-time as each field is moved once at maximum. Also, the detection of multiple cycles can be performed by linearly scanning elements in the lookup table once from front to back and resolving cycles as they are encountered.

5.2.2 Memory requirements

The demand for memory is dominated by the input buffer size, followed by space requirements for tags, buckets and lookup table. Meta data of buckets and histogram information do not depend on the input size. The following analysis of memory requirements is described using the big-O notation.

The input buffer requires O(n) space holding full tuples. Tag extraction occupies another O(m) with m >= 0.1n supposing the use of “Indy” binary records with 10-byte key and 90-byte payload. Also, tags need to hold an additional reference to their source tuple and platform specific padding increases memory usage. The test environment used for benchmark runs showed an overhead of

m = 0.16n for tags. The re-distribution step of radix sort requires two distinct sets of buffers, one to read from and another one to write to. Each set of buckets must be capable of holding the entire number of tags, resulting in a total overhead for tags and buckets of O(2m). The lookup table for in-place permutation does not introduce additional memory requirements due to reuse of space allocated to the unused set of buckets. In total the implementation of tag-based radix sort requires O(n + 2m) memory.

The size of the input buffer cannot be reduced beyond O(n) without introducing another I/O pass, and hence, tag-based radix sort is able to generate a run roughly at size of available main memory. The memory required by tag buffers could be reduced further by using tags that hold a portion of the actual tuple key, e.g. 4 instead of 10 bytes. This is sufficient to perform 4 rounds of radix sort before the input buffer needs to be re-scanned in order to update tag keys for a set of consecutive rounds. Assuming a 4-byte key portion and a 4-byte tuple reference, tag buckets could be transformed to use m' = 0.08n and effectively halve the overhead to O(m' = m/2). This approach trades and additional pass of random access to data in the input for reduced amounts of memory being copied during individual radix rounds. Another option is in-place reordering of tags, in analogy to tuple reordering. Though, this requires an additional buffer to hold the lookup table the overhead is reduced to O(m + m').

Memory overheads can be reduced in practice by amortizing them across multiple input buffers processed by a pipeline stage. Assuming that a single sorter processes a buffer faster than a single reader can access disks then the set of sorters can be smaller than the set of readers without degrading pipeline performance over sufficiently long runtimes. In case of Tritonsort, micro-benchmarks measure a maximum throughput of 100 MB/s per reader and 250 MB/s per sorter, so 4 sorters are sufficient to process data incoming from 8 readers.

5.2.3 Conclusion

The computationally most expensive task performed in the pipeline is internal sorting. Hashing in the map stage and caching in the store stage do create some overheads, but they are comparably small. As large amounts of data are

sorted the use of a linear-time algorithm has an intrinsic advantage compared to comparison-based approaches. Due to tag-based sorting performance and memory efficiency can be improved, although, this adds the necessity for separate permutation of input data.

6 Evaluation and Discussion

The Evaluation section is organized in four parts. First, internal sort and the disk I/O layer are tested and compared quantitatively to alternative approaches. Then, the integral system is compared to state-of-the-art systems in the context of Sort Benchmarks regarding performance and resource-efficiency.

6.1 Internal sort

Internal sort is implemented as Radix sort and operates on tags instead of full tuples. Radix sort guarantees linear run times, but increases memory consumption compared to standard Quicksort. The memory usage in turn is addressed by relying on tags while sorting and by reordering the input buffer in-place. The use of tags also speeds up sorting as the amount of data moved between (Radix-)buckets is decreased.

The benchmark is run on a single node in Tritonsort's test bed. The setup used for comparison looks as follows: input data is generated once for all test runs. For each buffer size, 5 consecutive sort runs are performed in memory using either sort algorithm. The run time is taken each time for a single buffer getting sorted by a single-threaded implementation of the algorithm. After the process completes average times are calculated and checksum and order of tuples are verified.

Input data is stored in a single 1600 MB file. It contains 16.000.000 tuples with a fixed length of 100 bytes. The first 10 bytes are considered the key value, the key values are distributed uniformly between 0 and 2^{80}-1. Before each sort run, a buffer in main memory is filled with a portion of these tuples from the beginning of the file to an offset depending on the maximum buffer size.

The benchmark is performed for 100MB, 200MB, 400MB, 800MB, and 1600MB buffer size. These values are chosen as they represent likely values for bucket files produced by Tritonsort's distribution sort pass depending on configuration. The figures below present the sort time and throughput relative to buffer size.

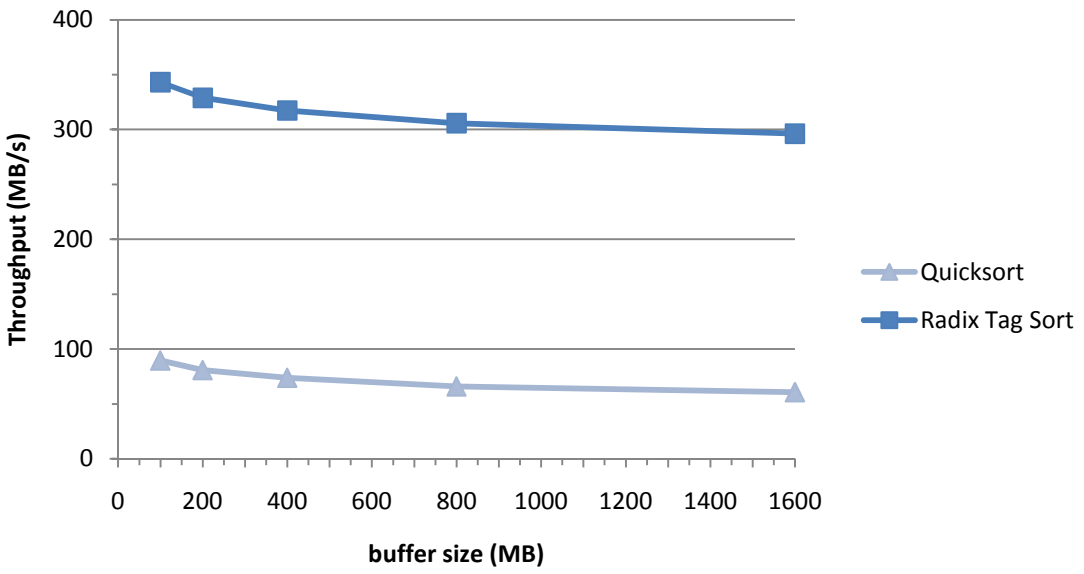


Figure 23- Internal Sort throughput

Input size (tuples)	*Input size (MB)*	*Radix Sort (MB/s)*	*Quicksort (MB/s)*
1.000.000	100	343,1	89,6
2.000.000	200	328,9	81,0
4.000.000	400	317,3	73,9
8.000.000	800	305,8	66,1
16.000.000	1600	296,2	60,8

Table 8 - Internal Sort throughput

The experiment shows an advantage for Radix Tag sort by a factor of 3.8 for 100MB buffers that continually increases up to 4.9 for 1600MB buffers. For a typical benchmark run of “Gray Sort” Tritonsort uses buffers of about 800MB. In this scenario, Radix Tag sort provides 4.6 times the performance delivered by Quicksort.

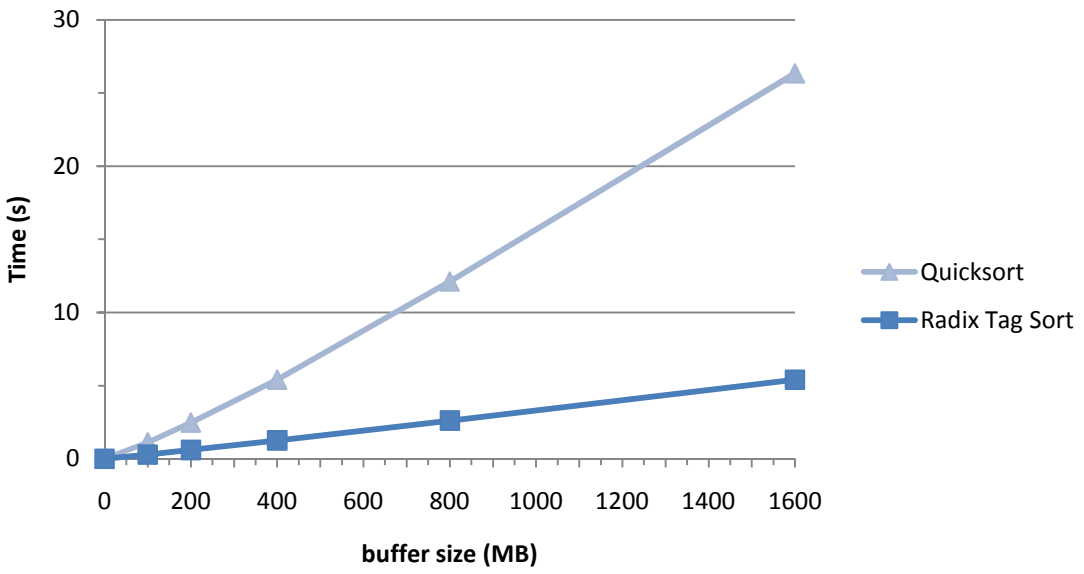


Figure 24 - Internal sort time

Input size (tuples)	*Input size (MB)*	*Radix Sort (s)*	*Quicksort (s)*
1.000.000	100	0,29	1,12
2.000.000	200	0,61	2,47
4.000.000	400	1,26	5,41
8.000.000	800	2,62	12,11
16.000.000	1600	5,40	26,33

Table 9 - Internal Sort time

From a runtime perspective Radix Tag Sort scales almost linearly with input size. Quicksort shows non-linearity from 200 MB buffer size upward. Also, the runtime graph directly reflects the observations made by throughput measurements. Radix Tag Sort performs the same amount of work in one fourth to one fifth of the time required by the Quicksort implementation.

6.1.1 Discussion

The relative speed increase of Radix Sort of a factor of four compared to Quicksort is a necessary improvement to performance. Without this, the processing pipeline is bottlenecked by internal sorting instead of disk I/O which should be the limiting factor throughput the pipeline for the purpose of scalability. The performance gain is bought by an increase in memory usage per sorter instance however. For the Sort Benchmark scenario this is the central

disadvantage compared to Quicksort, and hence, most development effort was put into reducing the overhead. Per sorter instance Radix Tag Sort uses about 30 percent additional memory over Quicksort. Applied to the context of parallel processing the advantage in performance allows using less parallel instances at the same time to process comparable amounts of data. In Tritonsort, typically 3-4 Radix Sort instance are used compared to 8 Quicksort instances. For a system's perspective this lowers memory overhead for sorting again to 10-15 percent.

This brings up another point, namely synthetic benchmark results. The presented sorter benchmark operates in a completely isolated environment. When running in a pipeline configuration multiple sorters are operating in parallel and share physical resources. In this case cache effects and competition reduce performance. A direct comparison of Quicksort and Radix Sort performance in a full pipeline run is difficult. Practically, sustained rate throughput can only be approximated by varying the number of parallel sorter instances and measuring system performance as runtime is usually constrained by disk I/O. If the system slows down, internal sorting is certainly the bottleneck. This approach leads to the numbers of 4 Radix Sort instances or 8 Quicksort instances provided above, although the 8 parallel Quicksort instances still do not outperform disk I/O sufficiently to deal with variations in bandwidth.

Taking a closer look on the implementation of Radix Tag Sort, there is a solid base implementation of the general algorithm. Up to this point optimization removed obvious slowdowns by timing individual passages of the implementation during benchmarking. This includes unnecessary memory reallocation and redundant arithmetic. However, typical aspects of cache-awareness and Assembler-level optimization are ignored even though they provide another perspective for improvement. Several cache-based improvements to Radix sort implementations are available in literature, e.g. a two stage bucket-radix sort as employed by NOW-Sort. This is left for further work as bottlenecks in the processing pipeline are already shifted towards disk and network I/O when using the current state of the Radix Tag Sort implementation.

More some more into the area of further work, two potential changes to the system can be identified that may pose a challenging task for re-architecting the internal sort implementation. First, the current design relies on single threaded processing per input buffer. If disk I/O relies on distributed file systems or RAID in the future, buffer size may increase substantially and create a bottleneck. Second, the introduction of variable-length keys could decrease performance when realizing compatibility via zero-padding or comparable approaches.

Overall, the development of the Radix Tag Sort-based component successfully resolves the bottleneck of internal sorting on the disk-heavy cluster nodes in Tritonsort's test bed. Also, the solution provides headroom in case further increases in I/O bandwidth.

6.2 Disk access

The disk I/O layer depends on the file system to handle raw disk control and file space allocation. Write Caching and buffering are handled manually by Tritonsort in order to increase throughput while writing in the Store stage and reading in the Reduce stage of the pipeline.

Tritonsort uses a distribution sort approach for memory external sort too and creates a number of intermediate files; one file per bucket. Multiple files are co-located on each physical disk and data is received for all buckets in parallel. When continually storing incoming data on disk an overhead is introduced as disks need to seek the appropriate physical positions before writing. The overhead can be reduced when (file-)system buffers increase the size of each sequential access.

To evaluate the performance of different approaches a benchmark is set up that mimics distribution sort behavior by writing to files in parallel first and reading sequentially afterwards. First, a fixed amount of random input data is generated and stored in buckets in parallel at equal rate. Meanwhile the implementation of bucket buffers and Writers stores data on disk autonomously. After input generation completes, the benchmark waits for Writers to completely persist any remaining data and flush file system caches via synching. In a second step the files are read sequentially one by one, using a single fixed buffer.

The benchmark is run on Tritonsort's test bed hardware using a single disk. Time is taken from the start of input generation to completion of synching and from the start of reading to the end of access to the last file. The benchmark is repeated for different numbers of buckets (and intermediate files) and the resulting numbers are averaged from benchmark runs on 4 different nodes.

The disk I/O layer in Tritonsort uses buckets that dynamically share a common memory pool on-demand. The design and implementation is complex compared to "intuitive" solutions relying on manual double buffering or default the default file system behavior.

The "buffered" implementation does not allocate memory for buckets manually. Instead, each incoming chunk of data is handed to the file system. The files system and operating system may freely dedicate a large portion of main memory to file caches.

The "double direct" implementation manually allocates two fixed-size buffers to each bucket. One buffer is used to receive data into, the other one is written to disk. On completion, buffers are swapped out. Buckets are written to disk in round-robin order and data is transferred invoking Linux direct I/O interface.

The "dynamic direct" implementation uses a shared memory pool for all buckets co-located on a physical disk. (Hence, a single pool is used in the benchmark) Buckets are written to disk based on demand - every time the Writer becomes available, the contents of the largest bucket are written to disk. The implementation relies on direct I/O too.

Although direct I/O circumvents some system caches, the file system still handles disk space allocation. Also, buffers on hard drives and drives controllers are active and perform own cache optimizations. The total amount of input data per run is set to a multiple of the amount of main memory to avoid additional cache effects when reading data in the second pass.

6.2.1 Write performance

The figure below illustrates benchmark performance during the write phase. The number of buckets is plotted on the horizontal axis, write performance is shown

on the vertical axis. The performance axis is offset at 40 MB/s for improved visibility.

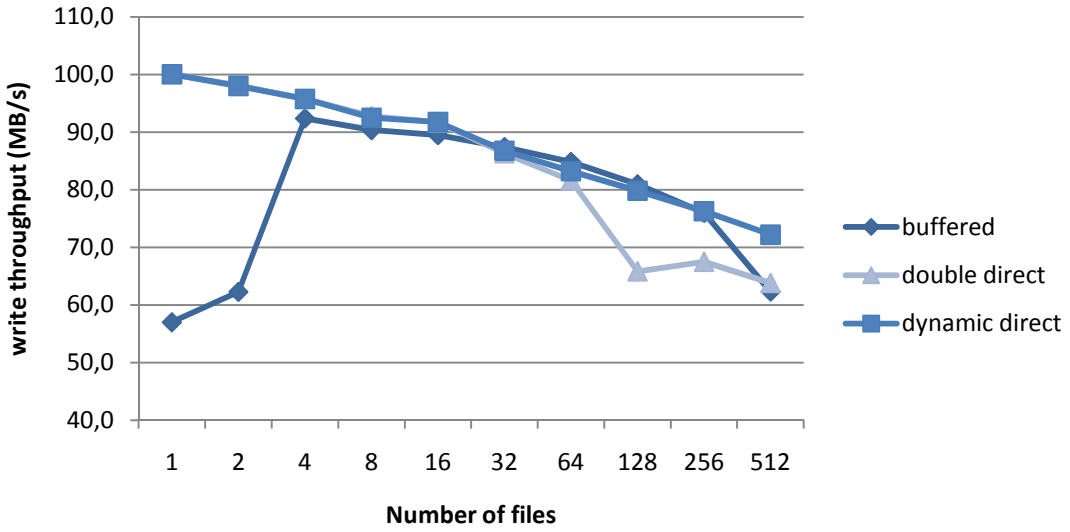


Figure 25 - Intermediate data write performance

Number of files	Buffered (MB/s)	double direct (MB/s)	dynamic direct (MB/s)
1	57,0	100,0	100,0
2	62,3	98,1	98,0
4	92,4	95,7	95,7
8	90,4	92,8	92,5
16	89,5	91,7	91,7
32	87,4	86,3	86,7
64	84,8	81,6	83,2
128	80,9	65,8	79,8
256	76,0	67,5	76,3
512	62,3	63,8	72,2

Table 10 - Intermediate data write performance

For small numbers of files, double-buffering and dynamic buffers perform equally, the file system-based approach surprisingly underperforms. In the range from 4 to 64 co-located files, all three implementations provide almost identical performance. The double buffering solution starts losing performance for 128 and more files. Dynamic buffering and the file-system solution continue at comparable rate for up to 256 files, when the dynamic buffer implementation gains an advantage again.

When performing a full 100TB sort run as required by the large-scale Sort Benchmark “Gray Sort” category, Tritonsort typically generates between 280

and 320 intermediate files depending on the number of participating nodes. For this application and larger benchmark instances, the dynamic buffer approach delivers optimal performance.

Overall, dynamic bucket buffers show constant results with predictable behavior when scaling in the number of files. The approach performs well for both, small and large numbers of files compared to the other two solutions.

6.2.2 Read performance

The figure below shows benchmark performance during the read phase. Again, the number of buckets is plotted on the horizontal axis, write performance is plotted on the vertical axis. The performance axis is offset at 40 MB/s.

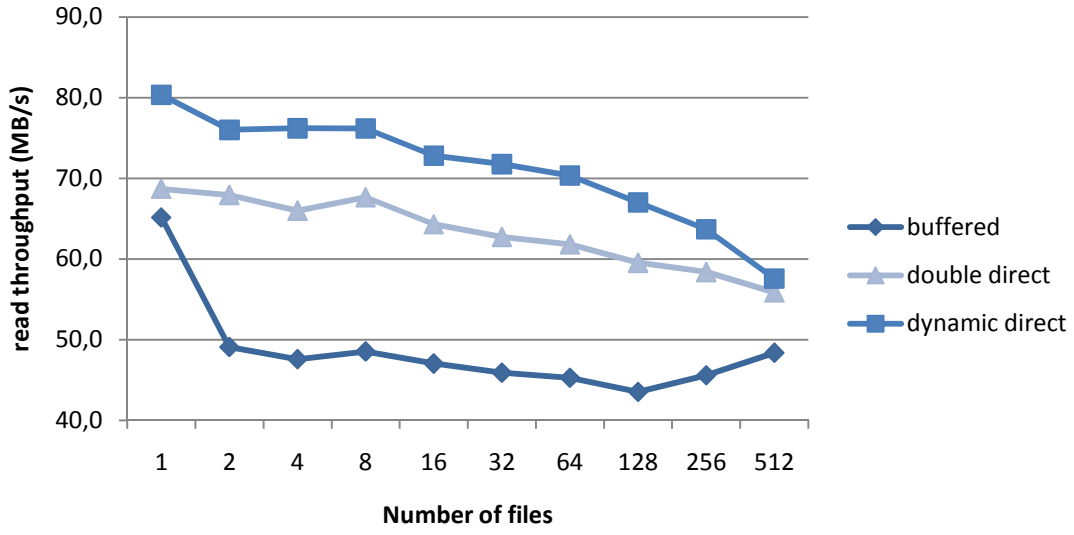


Figure 26 - Intermediate data read performance

Number of files	Buffered (MB/s)	double direct (MB/s)	dynamic direct (MB/s)
1	65,2	68,7	54,9
2	49,1	67,9	52,8
4	47,6	66,0	46,4
8	48,5	67,6	39,9
16	47,0	64,3	43,0
32	45,9	62,7	80,8
64	45,3	61,8	76,6
128	43,5	59,5	71,3
256	45,6	58,4	67,7
512	48,4	55,9	63,9

Table 11 - Intermediate data read performance

The read performance shows a difference for small numbers of intermediate files. With an increasing number of files, differences converge. Files generated by the dynamic bucket implementation show slightly better performance, but this likely depends on slight differences in physical file layout as there is not any difference in the implementation of the benchmark reader.

6.2.3 Discussion

Especially in relation with disk I/O the setup of a fully deterministic benchmark environment has shown to be difficult. Multiple benchmark runs deliver somewhat similar results, but produce unexpected spikes too. These seem to depend on many different variables such as system uptime, aging of the file system, background processes, etc. It was tried to recreate the exact environment for every benchmark run by reformatting disks, rebooting the system and starting tests at similar uptime timestamps. Still, these fluctuations could not be avoided completely, and hence, the numbers presented hereby are average values obtained from multiple runs with spike values being corrected manually.

Nevertheless, for the 100TB Gray Sort scenario the dynamic buffer approach shows a factor of 1.15 better write performance than double buffering and practically equal performance to the default buffered write behavior. For the consecutive read phase dynamic buffering performs a factor of 1.15 better than double buffering and 1.48 better than default file system behavior.

The close match in performance between dynamic buffering and default file system behavior for most benchmark cases is remarkable. The file system does an excellent job at maximizing write throughput for parallel disk access, although this comes at a heavy toll during reading data back later. For the default implementation the drop in performance during reading may arrive from file fragmentation, file-cache lookups or configuration issues, but this has not been investigated yet and is left for further investigation.

The I/O subsystem of Tritonsort currently still represents the bottleneck for increased performance, so ongoing optimization in this area is necessary. Disk I/O performs well when comparing Tritonsort to large-scale benchmark systems,

but there is potential for improvement as shown by small-scale single systems such as psort.

Also a potential drawback is the high complexity of the application-specific write cache implementation. The evaluation experiment for intermediate data access shows that a simple implementation based on default file system behavior performs comparably well for a number of cases during writing. However, corner cases exist for small and increasingly higher numbers of output files. If these can be resolved and read performance be increased by an improved configuration (or implementation) of the file system, this part of Tritonsort's pipeline could be re-architected and simplified.

Overall, the dynamic buffering approach represents the best alternative available to Tritonsort at the current state. It provides an advantage of 15/15 percent for read- and write-performance compared to double buffering and 50/- percent compared to default file system behavior. In comparison to state-of-the-art systems of comparable scale, Tritonsort also achieves highest throughput per disk using the presented approach.

6.3 System performance

In the following section Tritonsort's performance is evaluated using the metrics of Sort Benchmark, resource-efficiency is quantified and compared based on hardware requirements and cost. Values obtained for state-of-the-art systems are estimations derived from the respective publication and valid for purposes of comparison only. Cost estimations are taken from (8).

For purposes of estimating efficiency, Gray sort results are more valuable than Minute sort results. Minute sort measures burst performance and favors low startup and shutdown times. The short runtime makes it difficult to derive useful results by metric application.

Tritonsort competes in the Sort Benchmark Challenge, and hence, can be compared directly to a number of systems. In case of large-scale benchmarking (at or above 100TB of data), available results are relatively scarce. Also, a number of benchmark results were published without being submitted to Sort Benchmark officially. (27)(23)

The figure below gives a quantitative overview about hardware used in different systems. For ease of representation a logarithmic scale is used.

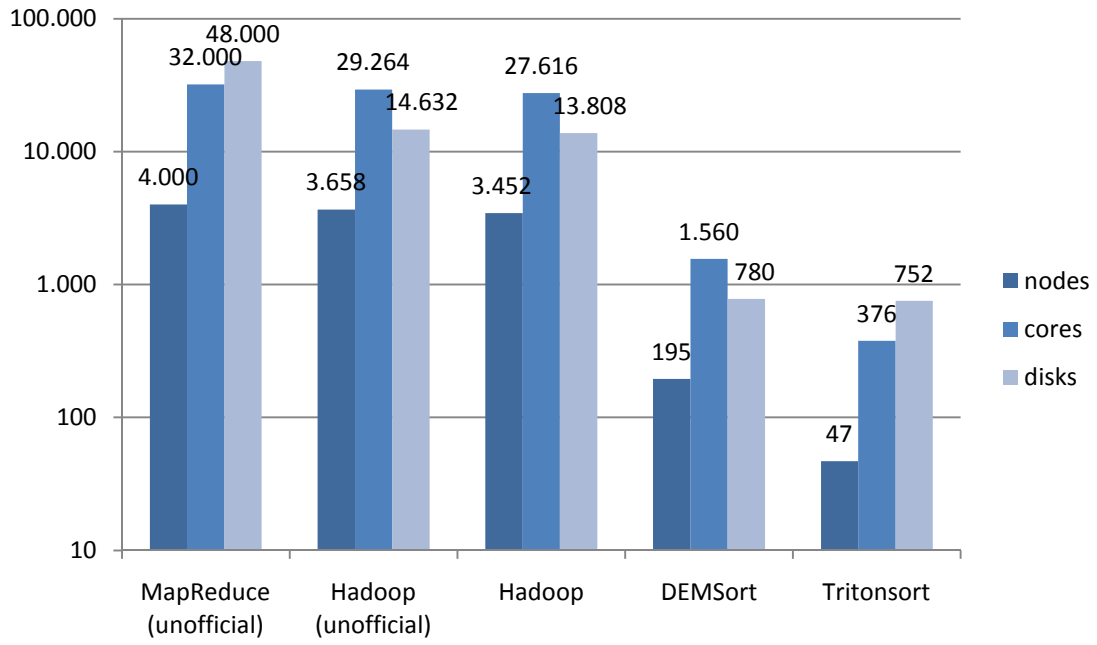


Figure 27 - Large-scale Systems in Benchmark

The scale of hardware assets varies massively. This should not come surprisingly as Hadoop and MapReduce are driven by industry companies, whereas DEMSort and Tritonsort are developed in an academic environment. All clusters are optimized to perform DISC tasks, except for DEMSort which operates in a HPC cluster environment.

Year	Name	Category	Nodes	Cores	Disks	Data (TB)	Time (s)
2008	MapReduce	(unofficial)	4.000	32.000[1]	48.000	1.000	21.720
2009	Hadoop (PB)	(unofficial)	3.658	29.264	14.632	1.000	58.500
2009	Hadoop	Gray Daytona	3.452	27.616	13.808	100	10.380
2009	DEMSort	Gray Indy	195	1.560	780	100	10.628
2010	Tritonsort	Gray Indy	47	376	752	100	10.318

Table 12 - Large-scale Systems in Benchmark

A selection of five systems is used for comparison. The table above gives an overview of system name, hardware components and the amount of data and time required for performing parallel external sort as used by sort benchmarks. Performance

Performance is the central interest of sort benchmarks, and hence, is used for a first comparison and as basis for considerations of efficiency.

The next figure presents relative system throughput per component. It is obtained by dividing the total amount of input data sorted by the amount of time required. A first look on resource-efficiency becomes possible by breaking down throughput numbers to component level and comparing them to each other, although these numbers do not factor in potential peak performance of a component. Since the systems' hardware relies on comparably potent components, values can still be used as indication for hardware efficiency. For ease of representation a logarithmic scale is used.

[1] Number of CPU cores estimated based on (8)

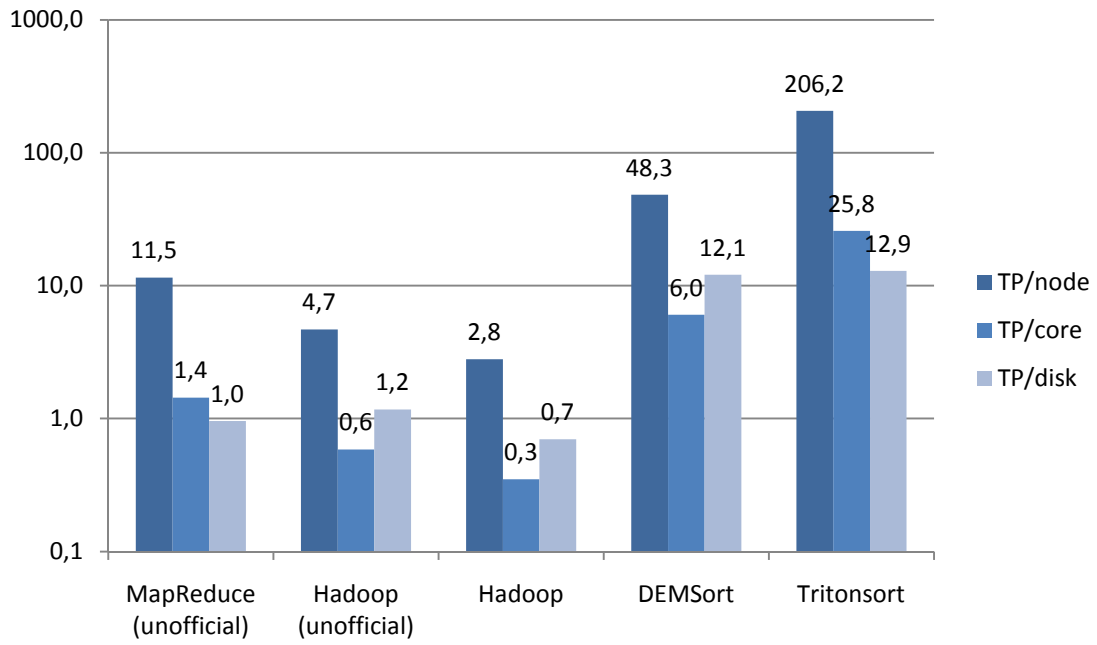


Figure 28 - System throughput per component

Name	*Throughput (MB/s)*	*TP/Node (MB/s)*	*TP/Core (MB/s)*	*TP/Disk (MB/s)*	*Cost Eff. (Bytes/s/$)*
MapReduce	46040,5	11,5	1,4	1,0	1e3,5
Hadoop (PB)	17094,0	4,7	0,6	1,2	1e3,2
Hadoop	9633,9	2,8	0,3	0,7	1e3,2
DEMSort	9409,1	48,3	6,0	12,1	1e4,1
Tritonsort	9691,8	206,2	25,8	12,9	1e4,4

Table 13 - Relative throughput per component

Performance-wise Tritonsort outperforms state-of-the-art systems in Sort Benchmark “Gray Sort” rankings. Compared to industry scale clusters overall throughput seems low, however, a substantial advantage in efficiency can be found.

6.3.1 Efficiency

It is difficult to quantify efficiency in absolute numbers as system hardware is different and hardware costs and performance change over time. The hardware components used in clusters by DEMSort and Hadoop are comparable to Tritonsort’s test bed as far as it can be determined from documentation. Information about hardware used for Google’s MapReduce run is scarce.

However, the clusters use commodity hardware and hence can be assumed of roughly equal potency per disk and CPU core. Therefore, these numbers are valid for comparing orders of magnitude only.

In order to provide a baseline for a holistic efficiency comparison, numbers for throughput per component and cost efficiency are fixed to 100 percent for the values of DEMSort, winner of the first Gray Sort Indy Challenge in 2009. A higher percentage indicates higher throughput, whereas lower values indicate less throughput in a specific aspect.

Name	*Throughput (MB/s)*	*TP/Node (%)*	*TP/Core (%)*	*TP/Disk (%)*	*Cost Eff. (%)*
MapReduce	46040,5	23,9	23,9	8,0	25,1
Hadoop (PB)	17094,0	9,7	9,7	9,7	12,6
Hadoop	9633,9	5,8	5,8	5,8	12,6
DEMSort	9409,1	100,0	100,0	100,0	100,0
Tritonsort	9691,8	427,4	427,4	106,8	199,5

Table 14 - Relative resource efficiency

The numbers suggest that Tritonsort performs well with respect to computational and disk I/O efficiency. This seems reasonable as it is mirrored by improved estimated cost efficiency too.

It can be observed that DEMSort shows comparable performance per disk, but at the same time shows only a quarter of throughput per core. This divergence could be explained by hardware coming from traditional HPC applications rather than disk-heavy benchmarks.

The numbers found for Hadoop provide an interesting insight, as about the same configuration was used in a 100TB and a 1PB benchmark run. Total throughput and throughput per component increase when datasets get larger. Even though, throughput for a 1PB run is about double the value of a 100TB run, throughput per disk is down by a factor of ten compared to Tritonsort.

MapReduce's total throughput on a 1PB data set outranges all other systems. However, as noted before, larger datasets increase efficiency. Throughput per component is comparable to Hadoop, although the disk-heavy hardware setup

improves results per core. Still, relative numbers per component are significantly lower than Tritonsort.

6.4 Benchmark-specific comparison

The following section provides an in-depth comparison of Tritonsort's performance to Sort Benchmark's top-performing systems Hadoop and DEMSort in the Gray Sort and the Minute Sort benchmark. The analysis addresses quantitative benchmark results and qualitative aspects of system architecture.

6.4.1 Gray Sort

The Gray Sort benchmark evaluates sustained system performance for long-running tasks. Also, the minimum requirement of 100TB input data indirectly enforces distributed hardware architecture and scalability of the software framework that coordinates individual nodes.

The benchmark uses the "average TB sorted per minute" (TB/min) metric to quantify system performance taking into account the total amount of input data and total runtime. As of 2010 the typical runtime for this benchmark lies between two and three hours what makes high average performance of system components the most important factor of success. Overheads caused by administrative tasks such as startup and shutdown delays do not severely impact overall performance as they even out over a long runtime.

1. Comparison to DEMSort

DEMSort is a merge-based sort implementation and uses hybrid memory architecture, shared-memory in the run formation stage and partitioned access to data in the redistribution and merge stages. Hence, there are substantial differences to Tritonsort's distribution-based approach.

Sorting takes place in three steps: run formation, redistribution and local merge. During run formation batches of input data are distributed transparently across nodes and sorted internally. The redistribution stage determines exact partitioning elements in the presorted runs and moves non-matching tuples between nodes until data is distributed evenly. The final merge stage takes place local to each node, reading and merging intermediate data to a final result.

Distribution of input data to target nodes is separated into a speculative first and an exact second pass during run formation and redistribution. Tritonsort uses a single speculative approach in the map stage. For both systems speculative distribution is based on the guarantee that key values in the input data are distributed uniformly. For DEMSort processing of node local data is performed during the run formation and merge stage, whereas Tritonsort performs node local sort during the store and reduce stage.

The decision to use a redistribution stage in DEMSort seems necessary from an architectural point of view ranging from the constraint of near in-place permutation of input data imposed by the lack of sufficient storage space for separate output files. This introduces an additional I/O pass at disk and network interfaces however. Although, the overhead is relatively small, it surpasses the minimum of 4 I/O passes for memory external sort. Another potential slowdown is caused by non-interleaved operation of network I/O and sorting during the run formation stage.

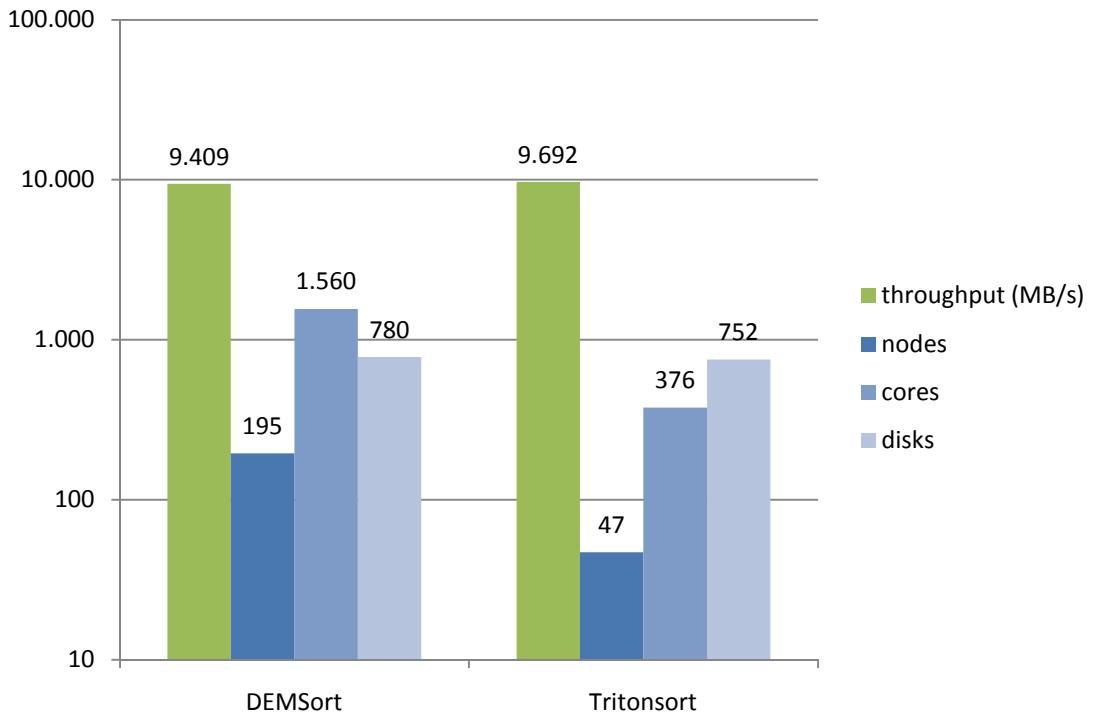


Figure 29 - Gray Sort - DEMSort vs Tritonsort - hardware performance

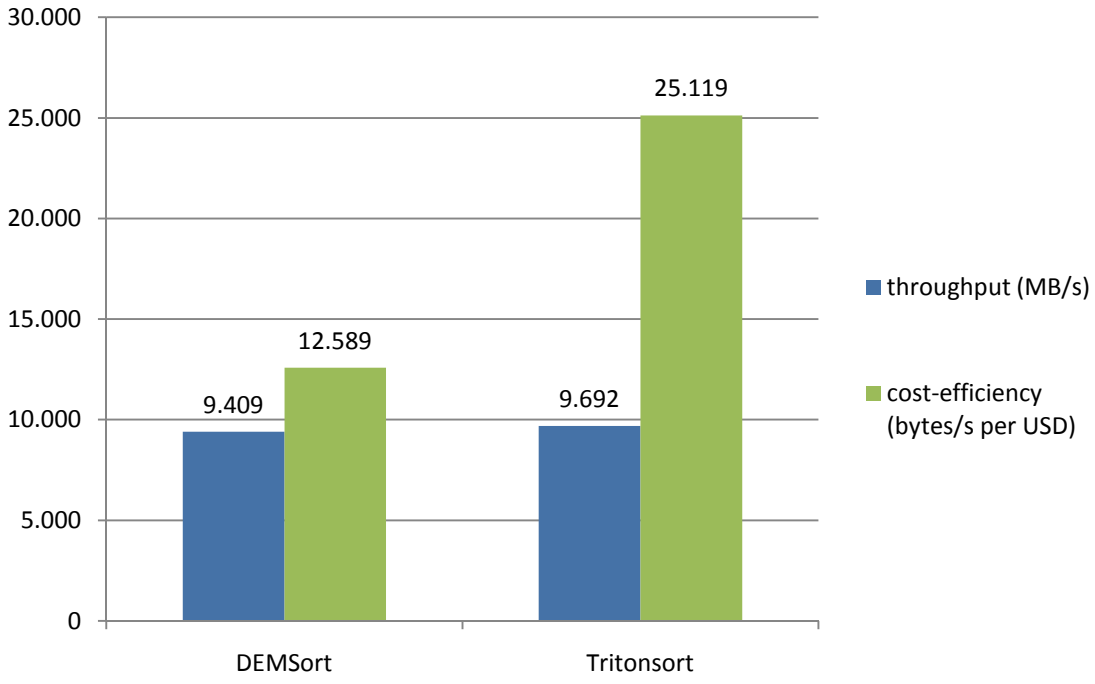


Figure 30 - Gray Sort - DEMSort vs Tritonsort - Cost-Efficiency

Hardware	**Tritonsort**	**DEMSort**
$nodes$	47	195
$cores$	376	1560
$disks$	752	780
$cost$	$10^{5.6}\ USD$	$10^{5.9}\ USD$
Benchmark		
$performance_{Gray}$	$0.582\ TB/min$	$0.565\ TB/min$
$data$	$100\ TB$	$100\ TB$
$time$	$10318\ s$	$10628\ s$
$throughput$	$9691.8\ MB/s$	$9409.1\ MB/s$
Relative Throughput		
$throughput_{node}$	$206.21\ MB/s$	$48.25\ MB/s$
$throughput_{core}$	$25.78\ MB/s$	$6.03\ MB/s$
$throughput_{disk}$	$12.89\ MB/s$	$12.06\ MB/s$
$throughput_{cost}$	$10^{4.4}\ bytes/s/\$$	$10^{4.1}\ bytes/s/\$$
Relative Scale		
$ratio_{nodes,T,D}$	0.241	
$ratio_{cores,T,D}$	0.241	
$ratio_{disks,T,D}$	0.964	
Resource Efficiency		
$efficiency_{cores,T,D}$	4.275	
$efficiency_{disks,T,D}$	1.069	
$efficiency_{cost,T,D}$	1.995	

Table 15 - Gray Sort - Evaluation Tritonsort vs DEMSort

From a hardware point of view DEMSort's cluster is about four times the size of Tritonsort's test bed, although the number of hard disks is almost equal. In detail, the ratio of nodes is 4.15, ratio of CPU cores is 4.15 and the ratio of hard drives is 1.04. DEMSort achieves 0.565 TB/min compared to Tritonsort's 0.582, a ratio of 0.971. Regarding cost-efficiency DEMSort reaches $10^{4.1}$ bytes/s/$ in contrast to Tritonsort providing $10^{4.4}$ bytes/s/$. Therefore, Tritonsort achieves about 4 times higher computational efficiency, comparable disk I/O efficiency and 2 times estimated cost-efficiency of DEMSort in Gray Sort.

2. Comparison to Hadoop

Hadoop is based on the Map-Reduce paradigm and relies on HDFS, a distributed file system for persistence of data. The overall sort process is very similar to the approach taken by Tritonsort: the Map stage reads and distributes input data to target nodes that store data locally. The reduce stage then reads the intermediate data and sorts it internally before writing the result back to the distributed file system.

Hadoop contributes to the Gray Sort Daytona benchmark ranking whereas Tritonsort performs in Gray Sort Indy. Benchmark scale, input data and rules are similar for the most part. There are two additional requirements for Daytona, however. First, entries must have the ability to perform general purpose sorting without assuming a predetermined uniform distribution of key values in the input. Hadoop addresses this by sampling a portion of input data at startup and distributing this information to the cluster before the actual sorting takes place. Secondly, input data must not be destroyed during processing, and hence, an out-of-place sort algorithm is enforced. This does not impact Hadoop in a notable way as Hadoop tasks do not delete input data per default anyway.

In addition, there are two aspects which may impact efficiency. First, Hadoop uses a replication factor of 2 for large datasets and employs a speculative scheduler possibly executing number of subtasks multiple times. Secondly, Hadoop runs in a Java Virtual Machine environment. Although, a number of optimizations were made to file transfers in the VM environment there is some additional overhead compared to Tritonsort which directly accesses operating system functionality.

In terms of hardware the Hadoop cluster larger than Tritonsort’s test bed by a factor of 70 in nodes in the Gray Sort configuration.

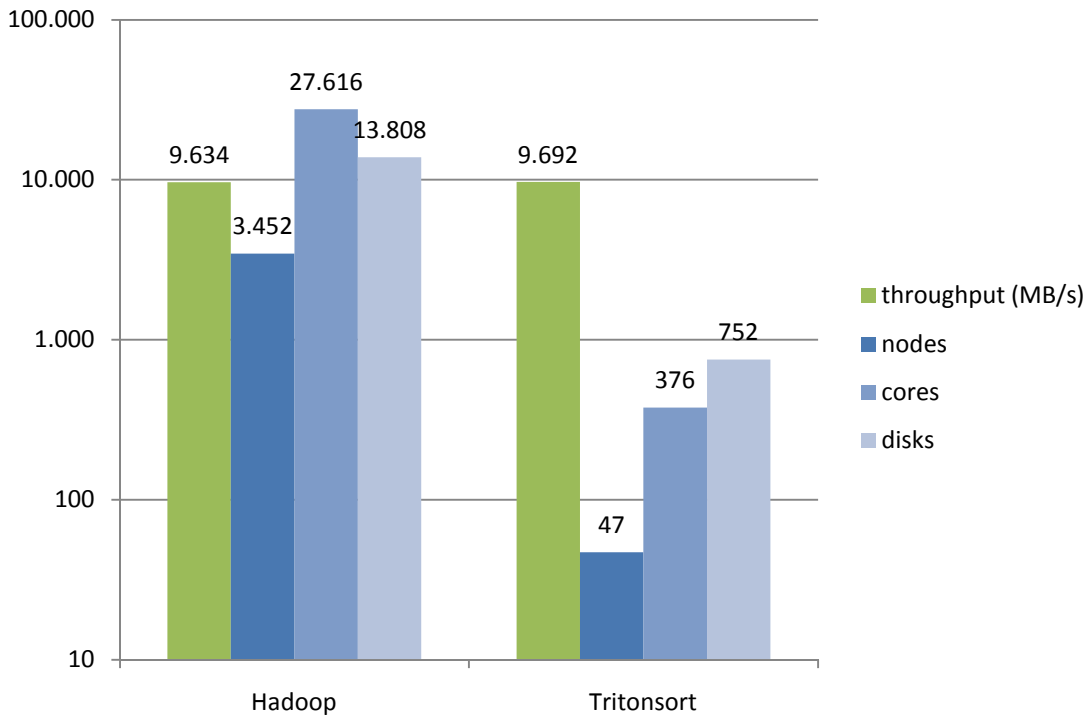


Figure 31 - Gray Sort - Hadoop vs Tritonsort - hardware performance

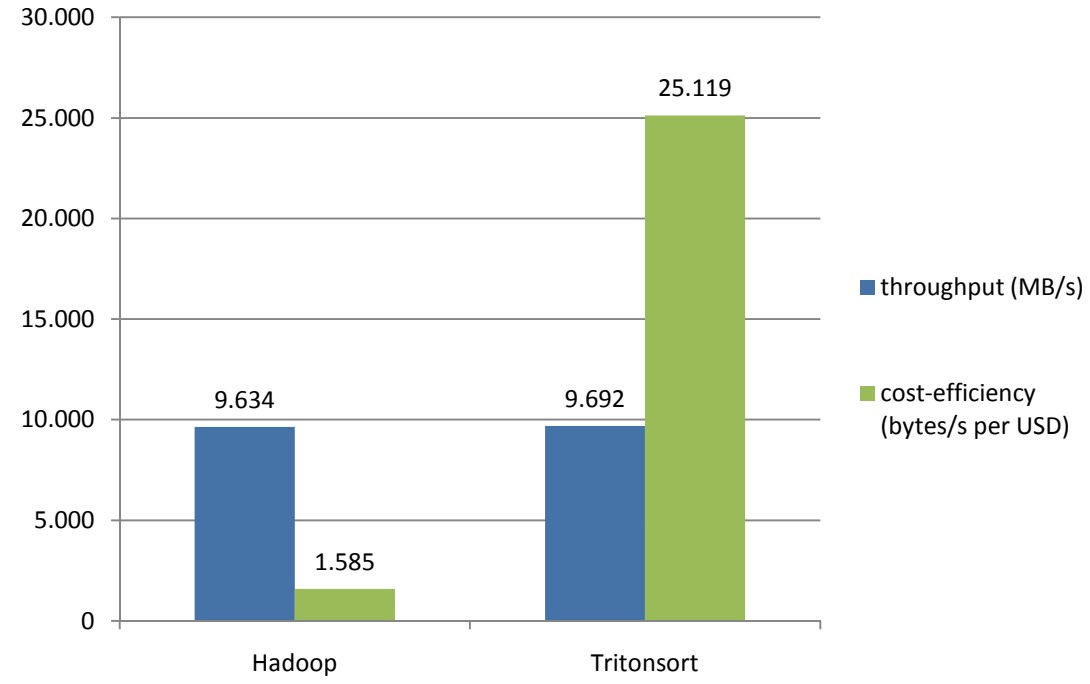


Figure 32 - Gray Sort - Hadoop vs Tritonsort - Cost-Efficiency

Hardware	Tritonsort	Hadoop
$nodes$	47	3452
$cores$	376	27616
$disks$	752	13808
$cost$	$10^{5.6}\ USD$	$10^{6.8}\ USD$
Benchmark		
$performance_{Gray}$	$0.582\ TB/min$	$0.578\ TB/min$
$data$	$100\ TB$	$100\ TB$
$time$	$10318\ s$	$10380\ s$
$throughput$	$9691.8\ MB/s$	$9633.9\ MB/s$
Relative Throughput		
$throughput_{node}$	$206.21\ MB/s$	$2.79\ MB/s$
$throughput_{core}$	$25.78\ MB/s$	$0.35\ MB/s$
$throughput_{disk}$	$12.89\ MB/s$	$0.70\ MB/s$
$throughput_{cost}$	$10^{4.4}\ bytes/s/\$$	$10^{3.2}\ bytes/s/\$$
Relative Scale		
$ratio_{nodes,T,H}$	0.0136	
$ratio_{cores,T,H}$	0.0136	
$ratio_{disks,T,H}$	0.0545	
Resource Efficiency		
$efficiency_{cores,T,H}$	73.66	
$efficiency_{disks,T,H}$	18.41	
$efficiency_{cost,T,H}$	15.85	

Table 16 - Gray Sort - Evaluation Tritonsort vs Hadoop

For Gray Sort, the ratio of hardware components Hadoop versus Tritonsort is 73.34 for nodes, 73.34 for CPU cores and 18.36 for hard drives. The benchmark performance for Hadoop equals 0,578 TB/min compared to 0.582 TB/min for Tritonsort, a ratio of 0.994 in performance. In terms of cost Hadoop delivers $10^{3.2}$ bytes/s/$ compared to Tritonsort's $10^{4.4}$ bytes/s/$. Hence, Tritonsort provides about 73 times computational efficiency, 18 times disk I/O efficiency and 16 times estimated cost efficiency of Hadoop in Gray Sort.

6.4.2 Minute Sort

The Minute Sort Benchmark measures the amount of input data processed by a system when running for 60 seconds or less total. The benchmark uses the “GB sorted” metric to quantify system performance. Runtime is measured in “wall time” at a singular head node from the moment the first node starts until the last node completing its task. A system is required to perform 15 consecutive runs for the same amount of input data and the average runtime of these runs is considered for validity.

Compared to Gray Sort the amount of input data is relatively small and possibly allows systems to perform memory internal sort. This reduces the theoretical minimum number of I/O passes from 4 to 2, thus decreasing the amount of time spent for disk access, but disallowing interleaved I/O and sorting at the same time. The short runtime also increases the impact of startup and shutdown procedures, as their overhead counts towards the overall timing.

1. **Comparison to DEMSort**

DEMSort operates within the Minute Sort Indy category and uses a comparable approach as the Gray Sort configuration. Although not stated explicitly by the developers, the increase in throughput suggests that intermediate file creation is spared for Minute sort. This guarantees that data on disk is accessed exactly twice, once for reading and once for writing. Assuming that the redistribution stage is present in this configuration, network I/O still involves some overhead. In contrast to Gray Sort synchronous pipelining of network transfer and sorting does not affect Minute sort negatively.

The cluster configuration of DEMSort is the same as for Gray Sort. Tritonsort uses a higher number of nodes for Minute Sort than in the Gray Sort configuration, and hence, has a slight advantage over DEMSort in the total number of hard drives. The number of nodes and CPU cores still is a multitude, however.

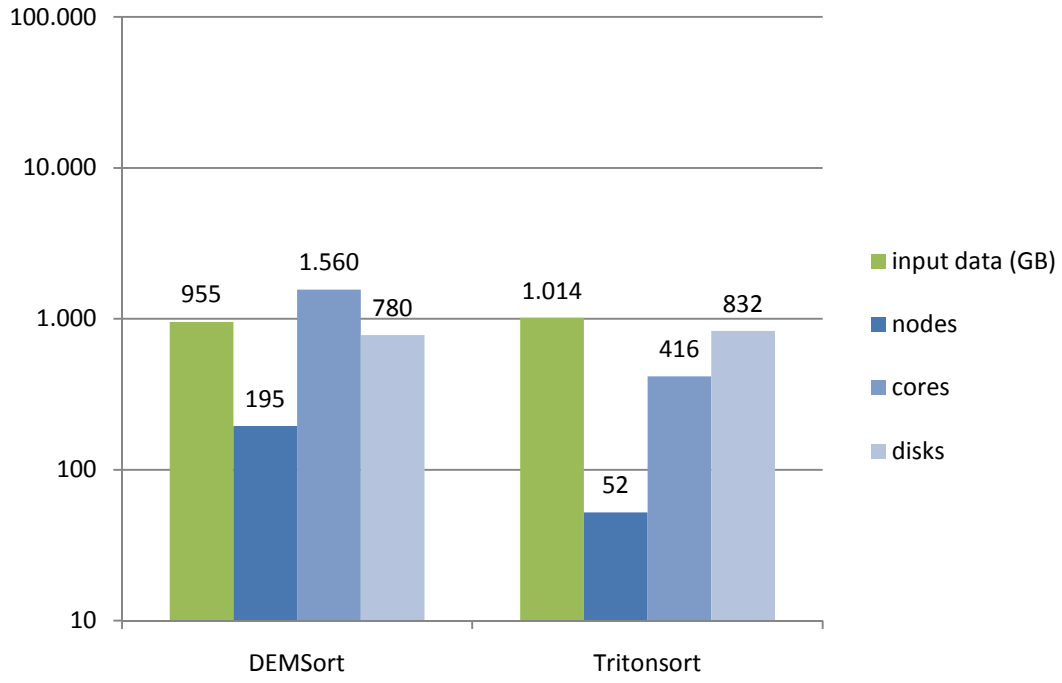


Figure 33 - Minute Sort - DEMSort vs Tritonsort - hardware performance

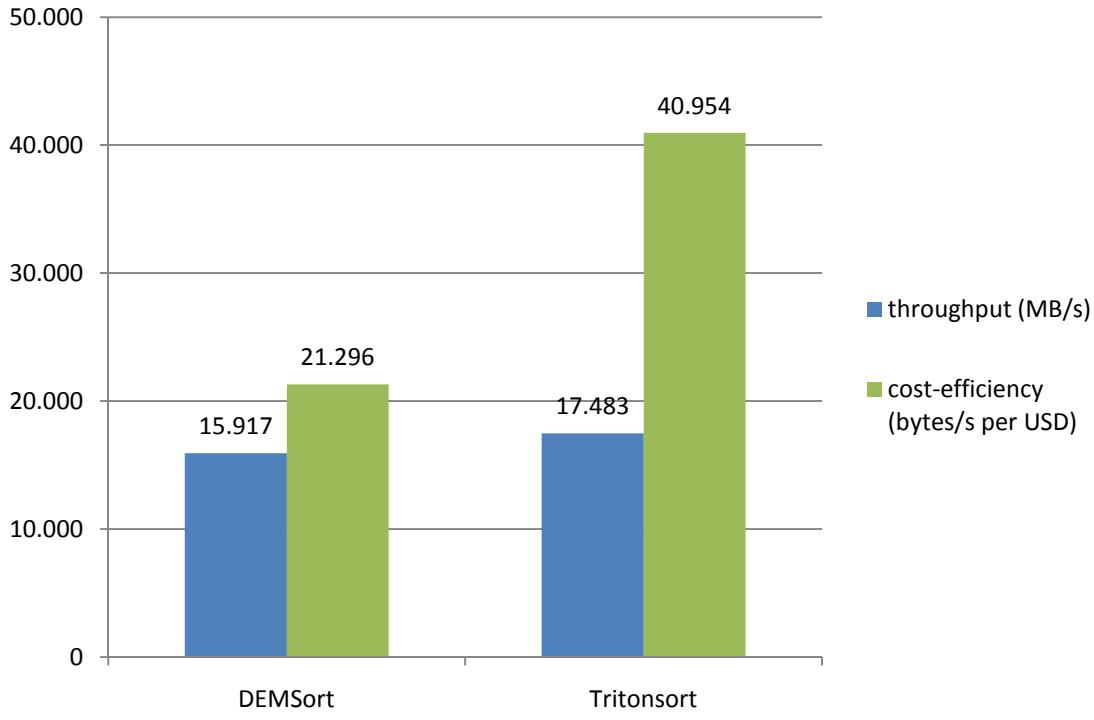


Figure 34 - Minute Sort - DEMSort vs Tritonsort - Cost-Efficiency

Hardware	Tritonsort	DEMSort
$nodes$	52	195
$cores$	416	1560
$disks$	832	780
$cost$	$10^{5.6}\ USD$	$10^{5.9}\ USD$
Benchmark		
$performance_{Minute}$	$1014\ GB$	$955\ GB$
$data$	$1014\ GB$	$955\ GB$
$time$	$58\ s$	$60\ s$
$throughput$	$17482.8\ MB/s$	$15916.7\ MB/s$
Relative Throughput		
$throughput_{node}$	$336.21\ MB/s$	$81.62\ MB/s$
$throughput_{core}$	$42.03\ MB/s$	$10.20\ MB/s$
$throughput_{disk}$	$21.01\ MB/s$	$20.41\ MB/s$
$throughput_{cost}$	$10^{4.6}\ bytes/s/\$$	$10^{4.3}\ bytes/s/\$$
Relative Scale		
$ratio_{nodes,T,H}$	0.267	
$ratio_{cores,T,H}$	0.267	
$ratio_{disks,T,H}$	1.067	
Resource Efficiency		
$efficiency_{cores,T,H}$	4.12	
$efficiency_{disks,T,H}$	1.03	
$efficiency_{cost,T,H}$	1.92	

Table 17 - Minute Sort - Evaluation Tritonsort vs DEMSort

In Minute Sort, cluster hardware of DEMSort and Tritonsort show a ratio of 3.75 for nodes, 3.75 for CPU cores and 0.938 for hard drives. In benchmark ratings, DEMSort achieves 0.955 TB compared to 1.01 TB for Tritonsort, a ratio of 0.942. Throughput is 15917 MB/s versus 18107 MB/s and in terms of cost-efficiency results in $10^{4.3}$ bytes/s/$ for DEMSort and $10^{4.6}$ bytes/s/$ for Tritonsort. In total Tritonsort reaches 4 times computational efficiency, equal disk I/O efficiency and 2 times cost-efficiency.

2. Comparison to Hadoop

The architecture of the Hadoop Minute Sort submission is similar to Gray Sort, although task scheduling is modified and additional disk I/O is avoided as the output of mappers fits into memory completely for the reduce stage. Also, replication is disabled to increase performance of HDFS and decrease network load.

Again, Hadoop participates in the Minute Sort Daytona category and requires an additional sampling stage that increases startup times compared to systems in the Indy category. The process of sampling and distributing acquired information across the cluster potentially takes a portion of time that systems of the Indy category spend sorting instead.

For a hardware perspective the Hadoop Minute sort cluster is smaller than the Gray Sort cluster in order to optimize startup and shutdown times. Compared to Tritonsort's hardware, its difference in scale still surpasses an order of magnitude.

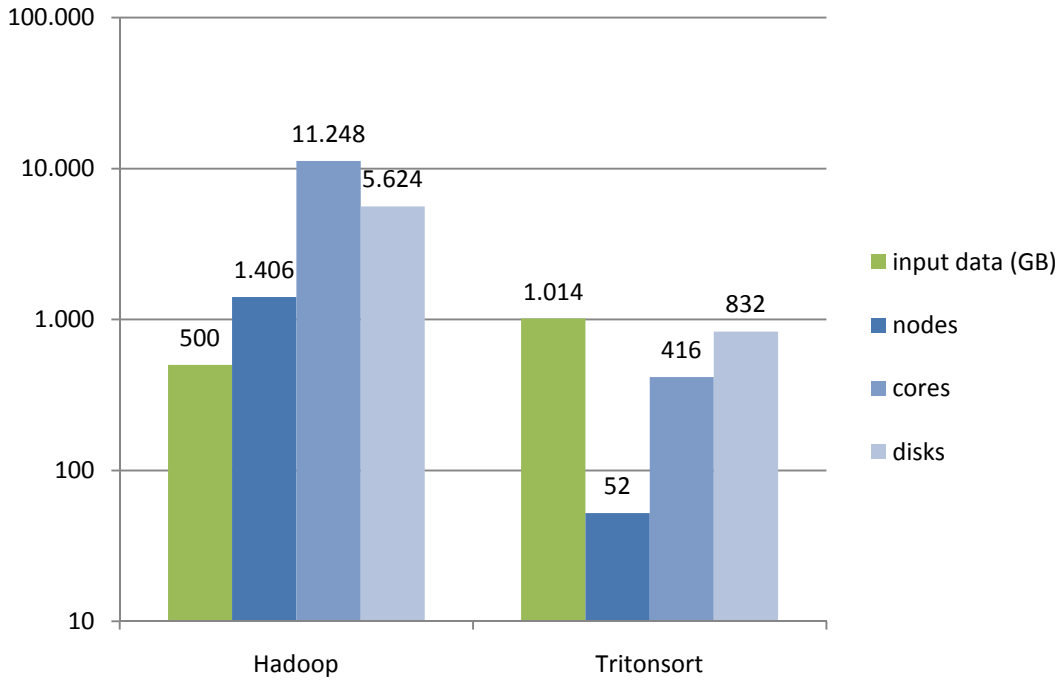


Figure 35 - Minute Sort - Hadoop vs Tritonsort - hardware performance

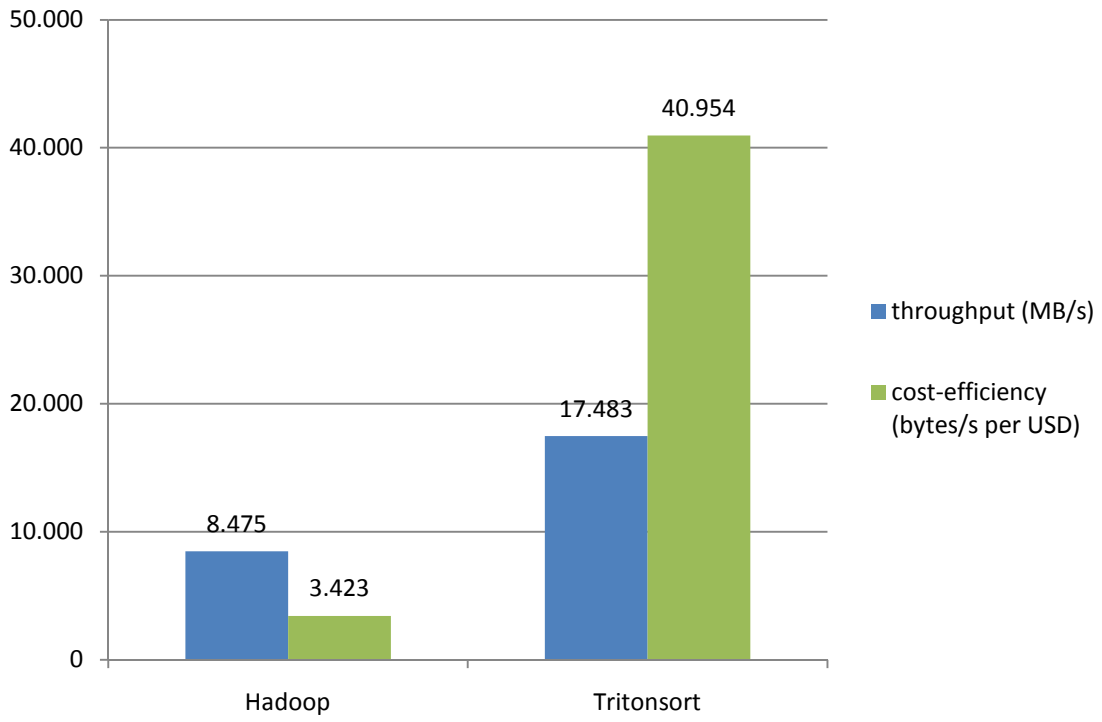


Figure 36 - Minute Sort - Hadoop vs Tritonsort - Cost-Efficiency

Hardware	**Tritonsort**	**Hadoop**
$nodes$	52	1406
$cores$	416	11248
$disks$	832	5624
$cost$	$10^{5.6}\ USD$	$10^{6.4}\ USD$
Benchmark		
$performance_{Minute}$	$1014\ GB$	$500\ GB$
$data$	$1014\ GB$	$500\ GB$
$time$	$58\ s$	$59\ s$
$throughput$	$17482.8\ MB/s$	$8474.6\ MB/s$
Relative Throughput		
$throughput_{node}$	$336.21\ MB/s$	$6.03\ MB/s$
$throughput_{core}$	$42.03\ MB/s$	$0.75\ MB/s$
$throughput_{disk}$	$21.01\ MB/s$	$1.51\ MB/s$
$throughput_{cost}$	$10^{4.6}\ bytes/s/\$$	$10^{3.5}\ bytes/s/\$$

Relative Scale	
$ratio_{nodes,T,H}$	0.0370
$ratio_{cores,T,H}$	0.0370
$ratio_{disks,T,H}$	0.1479
Resource Efficiency	
$efficiency_{cores,T,H}$	55.78
$efficiency_{disks,T,H}$	13.94
$efficiency_{cost,T,H}$	11.96

Table 18 - Minute Sort - Evaluation Tritonsort vs Hadoop

The ratio of hardware used by Hadoop compared to Tritonsort is 27.04 for nodes, 27.04 for CPU cores and 6.76 for hard drives. Benchmark performance is 0.500 TB for Hadoop and 1.01 TB for Tritonsort, a ratio of 0.493. In terms of throughput, Hadoop achieves 8475 MB/s versus 18107 MB/s. Hence, Hadoop provides cost-efficiency of $10^{3.5}$ bytes/s/$ compared to $10^{4.6}$ bytes/s/$ for Tritonsort. Overall, Tritonsort shows 56 times computational, 14 times disk I/O and 12 times cost-efficiency in Minute Sort.

6.4.3 Discussion

A broad variety of comparison results has been presented hereby with a focus on benchmark performance, computational efficiency, disk I/O efficiency and cost-efficiency. Tritonsort shows superior results in all four aspects, in every comparison provided.

On the one hand, this is a remarkable result, one the other hand the question about the correctness of values and the overall validity of the evaluation arises. Performance numbers and hardware components are derived directly from publications of Sort Benchmark results, and hence, can be regarded correct. In case of cost-efficiency, this is vague already. Hardware costs of DEMSort and Hadoop are estimated based on (8) by calculating system base-cost from the provided cost-efficiency measure “bytes/s/$” and benchmark throughput. The authors of the source paper provide numbers in powers of ten and note explicitly that their numbers are useful for purposes of comparison only. Therefore, the difference in cost efficiency between two systems in this paper can only be regarded as a guideline. As this is insufficient as sole indicator,

relative throughput per component and an overview of cluster component counts is provided is this section as well. In case provided cost estimations do not seem fitting other provided information allows a separate approximation.

Generally, it can be observed that Tritonsort uses a disk-heavy hardware setup. Compared to Hadoop and DEMSort each node holds four times the amount of hard drives. This configuration certainly helps decreasing hardware costs as the number of physical encasings and network connections goes down, although this requires computationally efficient software to process node-local data with one fourth of CPU time.

A second aspect of evaluation validity is the emphasis on resource-efficiency without comparing Tritonsort quantitatively to energy- or cost-efficient systems such as EcoSort and PSort. A major point against the comparison is the lack of scalability in these systems, although this should mainly affect networking and computation. It can be observed from smaller systems that there certainly is a factor of about 1.5 in disk I/O that should be achievable by Tritonsort in benchmark runs. Systems relying on SSD storage expand this borderline even further, but also point towards a substantial increase in hardware cost. From a computational efficiency point of view it shows that Tritonsort is not saturating its full CPU potential. Some smaller systems with slower processors and memory achieve higher throughput physical core. A portion of this headroom comes from the current bottleneck in disk I/O another portion potentially may never actually come to use in benchmark runs as it is superfluous. A third efficiency indicator, throughput per node, shows an advantage of up to a factor of two for Tritonsort compared to PSort or EcoSort and suddenly inverts the picture of resource efficiency. It can easily be explained by different hardware setups, but also points at an important fact: the definition of resource efficiency in this paper is based on the assumption of comparable hardware components and comparable scale. A comparison to a small system can lead the way for future improvements, but does not necessarily produce consistent results. Hence, an extensive comparison to small systems is not provided.

Taking a look at combined comparison results of DEMSort it can be observed that DEMSort provides comparable performance for benchmarks and disk I/O,

but stays behind in cost-efficiency by a factor of two and is improved upon computational efficiency by a factor of four. The architecture is sort-specific compared to Tritonsort's pipelining, which seems induced by the HPC hardware environment and the lack of additional storage space. Nevertheless, Tritonsort outperforms DEMSort in Gray Sort Indy and Minute Sort Indy and serves higher resource efficiency.

A look at the results for Hadoop shows a substantial advantage for Tritonsort regarding resource efficiency. The numbers are to be regarded more a guideline as they surpass an order of magnitude for Minute Sort as well as Gray Sort. It has to be noted that Hadoop processes Daytona data sets in contrast to Indy records. For the purpose of benchmark this does not affect the actual input data, however, an additional sampling stage is required upfront to determine the distribution of input keys. The sampling process adds little overhead for Gray Sort, but might impact Minute Sort results significantly. The fact however, Tritonsort delivers more than 50 times performance per core and 10 times per disk indicates a significant bottleneck in Hadoop's processing pipeline. This is unlikely to be caused by architectural differences as Hadoop and Tritonsort are both modeled after the Map-Reduce approach. It can be speculated that this is caused by network bottlenecks, inefficient disk I/O due to access from within a JVM or the broad spectrum of tasks Hadoop is designed for, although this cannot be determined from the view of this paper. In sum, Tritonsort delivers higher performance than Hadoop in Gray Sort and Minute Sort benchmarks and shows better resource efficiency by an order of magnitude.

7 Summary

The following section summarizes contributions of the thesis in detail and provides any overview over benchmark performance and evaluation results.

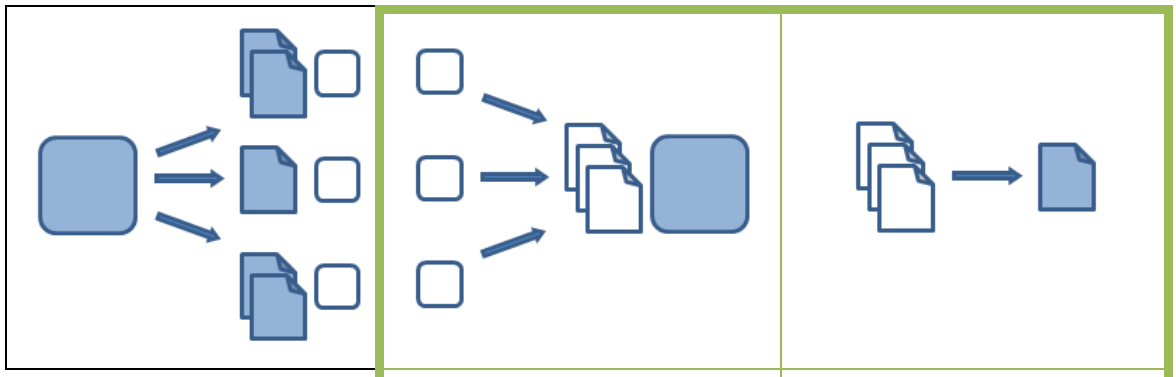
Tritonsort participated in the 2010 Sort Benchmark challenge for “Gray Sort” and “Minute Sort” and currently represents the top-performing system in both categories. In addition to performance Tritonsort also provides the highest level of hardware resource efficiency of current large-scale systems.

The research of this thesis shows that optimization of resource-utilization can yield substantial improvements to performance and cost-efficiency. This is true especially in large-scale data processing. Although, optimization requires additional engineering efforts it reduces cost and complexity at the same time as it decreases the amount of required hardware and the overall scale of a system.

The experience gained from the development of data persistence and internal sort components shows that efficiency is owed mainly to architecture and algorithms. Well-performing algorithms from literature are adapted to satisfy constraints imposed by hardware and architecture. Though, some of these modifications lower performance of theoretically optimal approaches, Tritonsort achieves solid levels of throughput in disk I/O and sorting compared to state-of-the-art systems.

7.1 Contribution Summary

The paper contributes the resource-efficient disk I/O layer and internal sort implementation to Tritonsort. The disk access layer represents an application-specific implementation of file-system buffering and improves upon the performance at the current bottleneck of the processing pipeline. Internal Sort is implemented to achieve performance necessary for processing data in the Reduce stage on-the-fly, without slowing down disk access. In addition it supports I/O performance by minimizing the number of intermediate files generated during the memory external sort.



Map	Store	Reduce
1a Read tuples from disk	**2a Receive incoming data** *(cheap synchronization)*	**3a Read presorted data** *(increased input size)*
1b Determine destination	**2b Cache in memory** *(memory-efficient buffering)*	**3b Sort tuples** *(increased performance and memory-efficiency)*
1c Send to target node	**2c Store on disk** *(sustained write throughput)*	**3c** Store on disk

Figure 37 - Contributions summary

Presentation and discussion of results is organized in separate sections for disk access in the Store stage and internal sorting in the Reduce stage.

7.1.1 Disk access

The I/O layer of Tritonsort optimizes throughput during the Store and Reduce stages of the pipeline based on three concepts. First, activity between Receivers and Writers is fully interleaved. Each bucket uses a segmented buffer for storing temporary data. Receivers are able to collect incoming data in a bucket while Writers may process the same bucket concurrently. This avoids pipeline stalls due to mutual exclusion to a large extent. Secondly, Writers are enabled to process buckets in a demand-based order. As Receiver and Writer activity is decoupled by the write cache implementation, Writers are enabled to continually perform at maximum disk performance. Thirdly, the buffers underlying each bucket share a common pool of memory, so buffer size can be adapted dynamically, based on demand. Buffer space freed up by a Writer can be allocated to multiple different buckets, which helps overcoming the systematic mismatch between input being scattered across buckets evenly in small chunks and output being performed per bucket in long sequential writes.

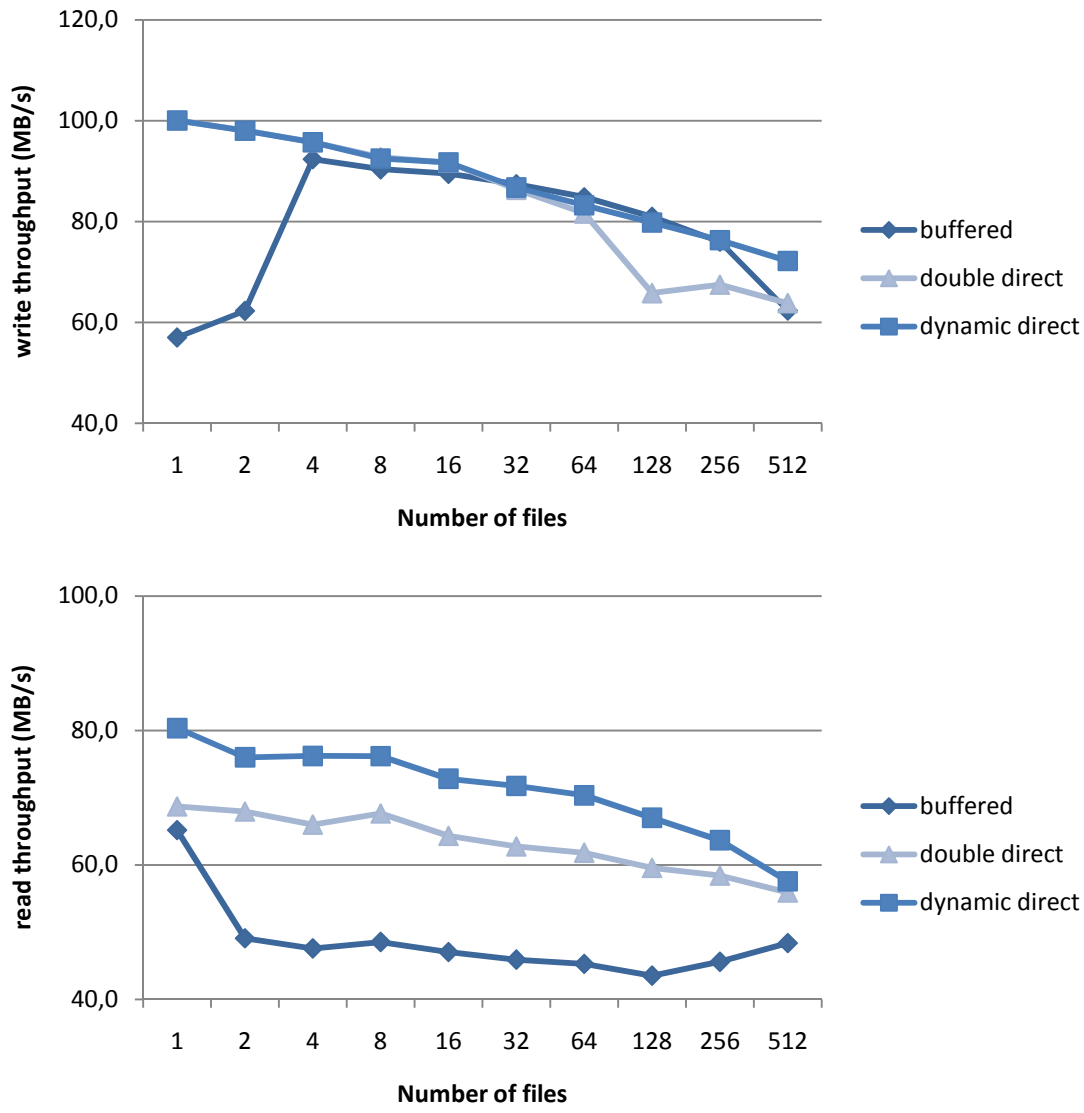


Figure 38 - Performance Disk Access

The design and implementation of the I/O layer is compared to two alternative approaches. Evaluation is based on a benchmark setup that mimics read and write patterns performed by Tritonsort in an isolated environment. The dynamic buffer implementation is compared to an implementation based on default file-system cache behavior and another based on manual double-buffering. From a qualitative point of view the dynamic buffer approach adds substantial complexity to design and implementation compared to a straight-forward implementation relying on file-system caching. Quantitatively, it is found that the dynamic buffer approach delivers highest throughput for read and write access under typical conditions as well as in corner cases, and hence, provides best I/O efficiency for the given hardware.

The optimization of intermediate data access also leverages from the increase in maximum bucket size by the internal sort implementation. This allows the Store stage of the pipeline to use a smaller number of intermediate files, and hence, the write throughput can be increased without negatively affecting the Reduce stage.

In the overall picture of comparison to state-of-the-art systems in large-scale Sort Benchmark, Tritonsort reaches the highest level of disk performance and I/O efficiency.

7.1.2 Internal Sort

Internal sort in the Reduce stage of Tritonsort's pipeline is implemented using a linear-time Radix sort. Combined with distribution sort performed in the first stages this allows Tritonsort to fully rely on linear-time algorithms. The actual implementation uses a number of optimizations to increase performance and memory efficiency. The first mayor optimization, tag-based sorting, extracts key values from tuples before sorting and significantly reduces the size of memory operations during the process. Also, the size of temporary buffers can be reduced and in-place reordering of the actual input data becomes possible. The in-place permutation of tuples reduces the typical memory overhead of Radix sort from 2.0 to 1.32 for this application, as a separate output buffer can be avoided. This in turn increases buffer size available per internal sort run, and therefore, decreases the number of individual bucket files required to split up data in the Map and Store stages of the pipeline.

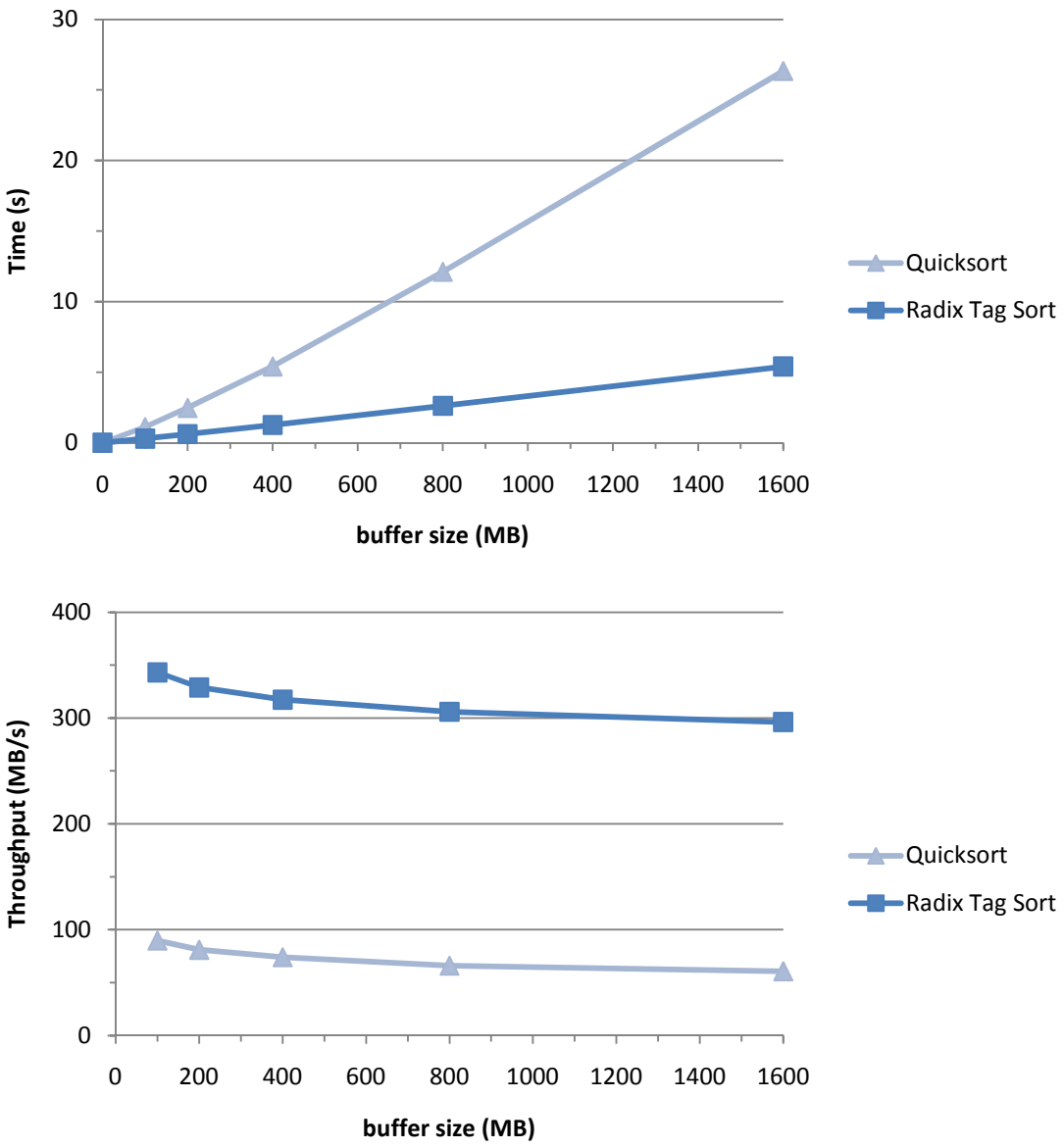


Figure 39 - Performance Internal Sort

The performance is evaluated using a simplistic internal sort benchmark. A variably sized region of memory is filled with uniformly distributed input data and sorted. The comparison is performed between a default C implementation of Quicksort and Radix Tag Sort. The Radix approach shows increased complexity in design and implementation compared to Quicksort. However, quantitative results show a significant advantage of Radix by a factor of four for typical buffer sizes between 100MB and 1600MB. From an efficiency point of view, Quicksort is to be considered better in memory efficiency, Radix shows superior computational efficiency. Given the test bed hardware, Radix sort is the algorithm of choice as it is able to handle parallel disk input on the fly, without bottlenecking.

The flexibility of the I/O subsystems allows improvements in sort memory efficiency to be translated directly into improved disk throughput. With increased buffer size for sorting the number of intermediate file decreases, lifting up I/O performance. This makes memory optimization of the implementation worthwhile and provides instant gains in I/O performance.

Throughput of internal sort is four times higher than the Quicksort alternative, the increase in sort performance is relevant to overall performance to a limited extend however. The implementation has to reach a level of throughput that allows a node's CPUs to handle parallel input provided by half its disks. Potential throughput superseding this threshold may reduce the impact of variations in input volume, but does not necessarily increase the performance of the Reduce stage. In practice however, high performance helps decreasing the number of sorter threads required in parallel which increases the relative amount of memory available per sorter and sort run.

Overall, the implementation of internal sort performs well for sustained throughput in the "Gray Sort" configuration as well as for peak performance in the "Minute Sort" configuration.

7.1.3 System Performance

As a result of the joint effort of the Tritonsort work group, the Tritonsort prototype participated in the 2010 Gray Sort Indy and Minute Sort Indy benchmark and was awarded top-performing system in both categories. Also, Tritonsort achieved the highest sort performance in Sort Benchmark's records for large-scale sorting and passed the barrier of 1 TB for Minute Sort.

The relatively small number of components in the cluster decreases complexity and costs of Tritonsort's test bed compared to systems such as Hadoop or DEMSort. For the Gray Sort benchmark, Tritonsort's 50 nodes 3 rack cluster provides comparable performance to Yahoo's 3800 nodes "Hammer" cluster.

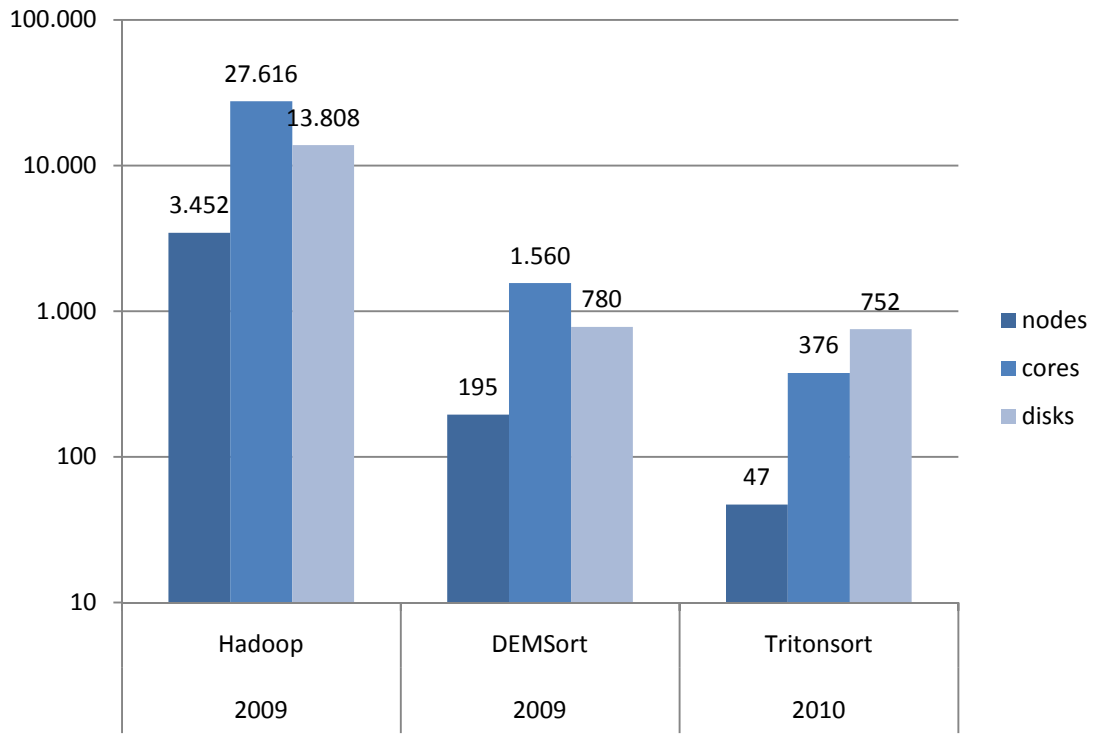


Figure 40 - Cluster hardware

In addition to less hardware than comparable systems Tritonsort achieves top performance in both large-scale benchmarks, Gray Sort and Minute Sort at the same time.

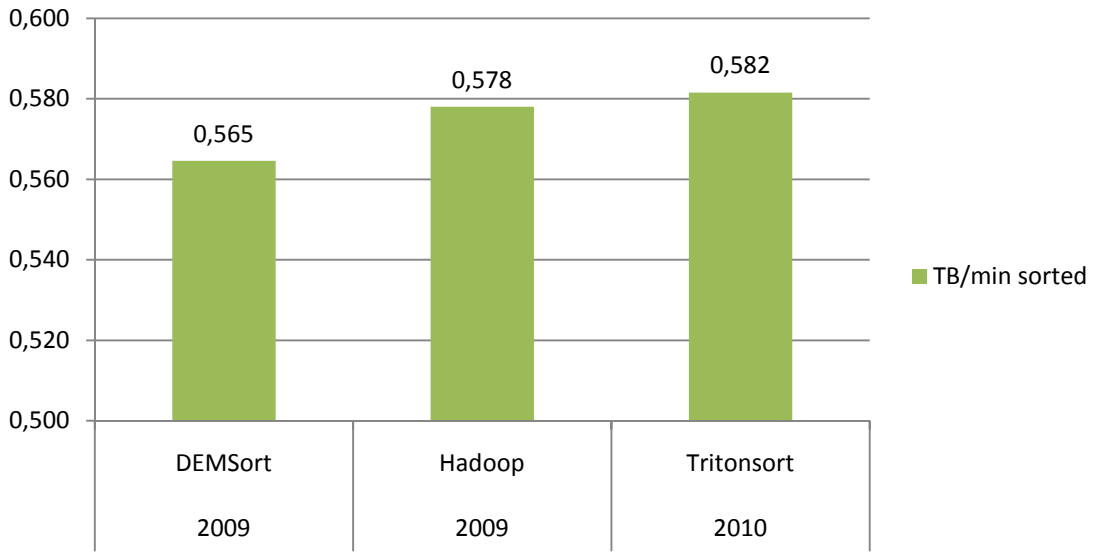


Figure 41 - Gray Sort performance

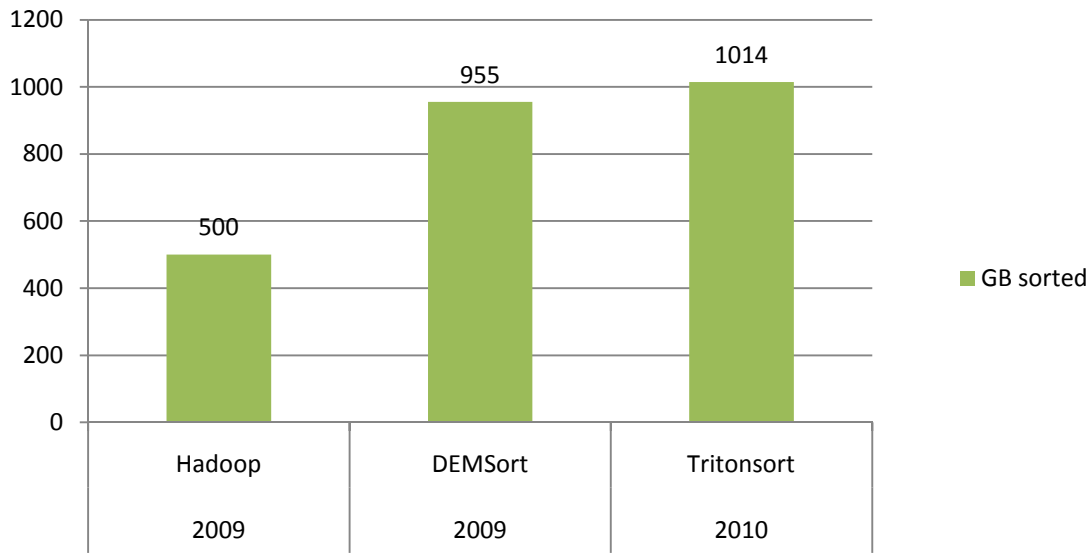


Figure 42 - Minute Sort performance

Resource efficiency decreases scale and complexity and saves money in large-scale DISC applications. Using Sort Benchmark as example it is shown there is a worthwhile potential for improvements to resource efficiency that directly lead to a smaller and more cost-effective system.

8 Future Work

The Tritonsort prototype delivers top results in benchmarking and resource-efficiency. However, as the project is work in progress there are still many perspectives for improvements and extensions to suite additional use-cases. The following section describes some of these.

In terms of hardware and cost efficiency there's potential for improvement. Optimally, Tritonsort's distributed architecture may reach levels of throughput per CPU core and disk comparable to optimized single node systems. Especially platform specific optimizations could bring improvements to computational and I/O efficiency. For example, the internal sort could be optimized to make better use of CPU caches and the I/O layer could be adapted to take disk and controller caches into account. The additional complexity introduced by networking will limit the ability to reach single-node cost efficiency at some point however.

Another spot for efficiency tweaking could be improved configuration of hardware and operating system. Tritonsort relies on OS facilities to a large extent for disk and network I/O. The development process showed that there are different possibilities to realize various features. However, performance and stability often depended on slight differences in configuration that are not obvious from documentation at first. Linux' I/O subsystem is the most notorious candidate for this kind of investigation.

A potential extension of functionality is the implementation of autonomous sampling and general purpose sorting as required by Sort Benchmark's Daytona category. A first step would be the adaption of sorting and I/O components to support variable-length Daytona datasets. In a second step, distributed input sampling could be implemented. Sampling might prove to be a significant challenge as the efficiency of the overall distribution sort algorithm highly depends on uniform load balancing.

When approaching real-world application of Tritonsort's pipelining system, facilities for fault-tolerance and failure-resistance become necessary. These should be capable of dealing with disk failures and, with growing scale, node and network issues. Disk failures could be addressed using RAID configurations, failures on larger scale require more sophisticated solutions based on replication. The introduction of RAID configurations in turn may substantially affect the disk I/O layer as read and write characteristics of RAIDs differ from independently controlled disks.

9 Acknowledgements

Alexander Pucher was supported by a scholarship of the Austrian Marshall Plan Foundation during his work at the Department of Computer Science and Engineering at the University of California San Diego.

The work with the Systems and Networking group at UC San Diego has been challenging and educating. I personally want to thank Professor Amin Vahdat and the members of the Tritonsort team Alexander Rasmussen, George Porter, Harsha V. Madhyastha, Michael Conley and Radhika N. Mysore for this great time and I want to thank Professor Stefan Biffl of Vienna University of Technology for his support during the writing of the thesis.

10 References

1. **Armbrush, M., et al.** Above the Clouds: A Berkeley View of Cloud Computing. *Reliable Adaptive Distributed Systems Laboratory.* [Online] 2 10, 2009. [Cited: 7 1, 2010.] http://radlab.cs.berkeley.edu/.

2. **Bialecki, A., et al.** Hadoop: a framework for running applications on large clusters built of commodity hardware. *Hadoop Wiki.* [Online] [Cited: 8 30, 2010.] http://wiki.apache.org/hadoop/.

3. *Mapreduce: simplified data processing on large clusters.* **Dean, J., Ghemawat, S.** Berkeley, CA, USA : USENEX Association, 2004. OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation. p. 10.

4. *Dryad: distributed data-parallel programs from sequential building blocks.* **Isard, N., et al.** Lisbon, Portugal : ACM SIGOPS, 2007. Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007. pp. 59-72.

5. **Bryant, R. E.** *Data-intensive Supercomputing: The case for DISC.* Pittsburgh, PA : School of Computer Science, Carnegie Mellon University, 2007.

6. **Dean, J.** *Designs, Lessons and Advice from Building Large Distributed Systems.* Big Sky, MT, United States : Google Inc., 2009.

7. *JouleSort: A Balanced Energy-Efficiency Benchmark.* **Rivoire, S., et al.** Beijing, China : ACM SIGMOD, 2007. 978-1-59593-686-8/07/0006.

8. *Efficiency Matters!* **Anderson, E. and Tucek, J.** Boston, MA, United States : Big Sky, MT, 2009. Proc of HotStorage.

9. **Nyberg, C. and Shah, M.** Sort Benchmark Website. [Online] June 01, 2010. www.sortbenchmark.org.

10. **Anon, et al.** *A Measure of Transaction Processing Power.* available from http://www.sortbenchmark.org : Datamation, 1985. pp. 112-118.

11. **Nyberg, C., et al.** Alphasort: A Cache-Sesitive Parallel External Sort. *VLDB Journal.* 1995, Vol. 1995, 4, pp. 603-627.

12. **Gray, J.** "A Measure of Transaction Processing" 20 Years Later. *Data Engineering, IEEE.* Special Issue on Databases for new Hardware, 2005, 28/2.

13. *Distributed Computing Economics.* **Gray, J.** 3, New York, NY, United States : ACM, 2008, Queue, Vol. 6, pp. 63-68.

14. **Knuth, D.** External Sorting. *The Art of Computer Programming.* Second Edition. Reading, MA : Addison-Wesley, 1998, Vol. 3, 5.4, pp. 248-379.

15. **Vitter, J. S.** *Foundations and Trends in Theoretical Computer Science.* Hanover, MA : Now Publishers Inc., 2008. pp. 305-474. 1551-305X.

16. *Parallel Sorting on a Shared-Nothing Architecture using Probabilistic Splitting.* **DeWitt, D. J., Naughton, J. F. and Schneider, D. A.** Miami, Florida, United States : IEEE, 1991. 1st International Conference on Parallel and Distributed Information Systems. 0-8186-2295419.

17. *Parallel sorting by over partitioning.* **Li, H. and Sevcik, K. C.** Cape May, New Jersey, United States : ACM, 1994. Proceedings of the sixth annual ACM symposium on Parallel algorithms and architectures. 0-89791-671-9.

18. *High-Performance Sorting on Networks of Workstations.* **Arpaci-Dusseau, A. C., et al.** Arizona : ACM SIGMOD, 1997. 0-89791 -911 -419710005.

19. **Zhang, X., Rivera, L. and Chien, A. A.** *HPVM Minute Sort.* s.l. : available at http://www.sortbenchmark.org, 2000.

20. **Azuma, S., et al.** *DIAPRISM Hardware Sorter.* s.l. : available at http://www.sortbenchmark.org, 2000.

21. **Wyllie, J.** *Sorting on a Cluster Attached to a Storage-Area Network.* San Jose, CA : available at http://www.sortbenchmark.org, 2005.

22. **Rahm, M., et al.** *DEMSort - Distributed External Memory Sort.* Karlsruhe, Germany : available from http://www.sortbenchmark.org, 2009.

23. **O'Malley, O. and Murthy, A. C.** *Winning a 60 Second Dash with a Yellow Elephant.* Santa Clara, CA : available from http://www.sortbenchmark.org, 2009.

24. **Bertasi, P., Bressan, M. and Peserico, E.** *psort, yet another fast stable sorting software.* Padova, Italy : available from http://www.sortbenchmark.org, 2008.

25. **Beckmann, A., et al.** *Energy-Efficient Sorting using Solid State Disks.* Karlsruhe, Germany : available from http://www.sortbenchmark.org, 2010.

26. **Lee, S., et al.** Partitioned Parallel Radix Sort. *High Performance Computing.* 1940/2000. Berlin, Heidelberg, Germany : Springer, 2000, Vol. Lecture Notes in Computer Science, pp. 160-171.

27. **Czajkowski, G.** Sorting 1 PB with MapReduce. *Google Blog.* [Online] Google, 11 21, 2008. [Cited: 05 30, 2010.] http://googleblog.blogspot.com/2008/11/sorting-1pb-with-mapreduce.html.

28. **Bertasi, P., Bressan, M. and Perserico, E.** *psort 2009.* Padova, Italy : available from http://www.sortbenchmark.org, 2009.

29. **Rasmussen, A., et al.** *TritonSort.* San Diego, CA : available from http://www.sortbenchmark.org, 2010.

30. **O'Mally, O.** *TeraByte Sort on Apache Hadoop.* Santa Clara, CA : available from http://www.sortbenchmark.org, 2008.