Spring 5-2018

Viability of Time-Memory Trade-Offs in Large Data Sets

Kyle Harper

VIABILITY OF TIME-MEMORY TRADE-OFFS IN LARGE DATA SETS

A graduate project submitted to Dakota State University in partial fulfillment of the requirements for the degree of

Master of Science

in

Information Systems

May 1, 2018

By
Kyle Harper

Project Committee:

Dr. Christopher Olson
Dr. David Bishop
Dr. Stephen Krebsbach

PROJECT APPROVAL FORM

We certify that we have read this project and that, in our opinion, it is satisfactory in scope and quality as a project for the degree of Master of Science in Information Systems.

Student Name: Kyle Harper

Master's Project Title: Viability of Time-Memory Trade-Offs in Large Data Sets

Faculty supervisor: Chris Kloon Date: 4/26/2018

Committee member: Stephen Krebsbach Date: 4/26/2018

Committee member: David Bishop Date: 4/26/2018

ACKNOWLEDGMENT

First and foremost, my wife and family for providing a support network I could depend on.

Yann Collet and Facebook for creating and sharing two astounding compression techniques with the world (lz4 and zstd, respectively), and inspiring me to extend their newfound techniques into the areas of this project.

ABSTRACT

The main hypothesis of this paper is whether compression performance – both hardware and software – is at, approaching, or will ever reach a point where real-time compression of cached data in large data sets will be viable to improve hit ratios and overall throughput.

The problem identified is: storage access is unable to keep up with application and user demands, and cache (RAM) is too small to contain full data sets. A literature review of several existing techniques discusses how storage IO is reduced or optimized to maximize the available performance of the storage medium. However, none of the techniques discovered preclude, or are mutually exclusive with, the hypothesis proposed herein.

The methodology includes gauging three popular compressors which meet the criteria for viability: zlib, lz4, and zstd. Common storage devices are also benchmarked to establish costs for both IO and compression operations to help build charts and discover break-even points under various circumstances.

The results indicate that modern CISC processors and compressors are already approaching tradeoff viability, and that FPGA and ASIC could potentially reduce all overhead by pipelining compression – nearly eliminating the cost portion of the tradeoff, leaving mostly benefit.

DECLARATION

I hereby certify that this project constitutes my own product, that where the language of others is set forth, quotation marks so indicate, and that appropriate credit is given where I have used the language, ideas, expressions or writings of another.

I declare that the project describes original work that has not previously been presented for the award of any other degree of any institution.

Signed,

_____

Kyle Harper

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Background of the Problem

Storage devices must address a myriad of constraints in their design. The two most commonly known are capacity and performance, which is rated in Input/Output Operations per Second (IOPS) or alternatively: block accesses. However, additional constraints exist: reliability, error detection/correction, environmental tolerances, physical size and weight, and much more. These factors all contribute to the limitations of many storage devices. Despite these constraints storage devices have continued to grow in capacity and performance while remaining reliable. Unfortunately, storage performance remains a bottleneck in many systems. Storage devices simply have not increased their IOPS sufficiently to keep up with storage capacities, network speeds, and CPU processing power. Evidence of this can be found in file systems, relational databases, and any other system that requires a "large" data set (including video games!).

Despite our insatiable appetite for a digital wonderland, hardware advancements continue to provide the computational power necessary to appease us. However, not all resources have seen equal gains in capacity and performance. This is never truer than with storage access. CPU throughput, RAM capacity and speeds, and even storage capacities continue to climb, but access to that storage struggles to keep up. The struggle of accessing storage is felt most heavily in large data systems such as databases, data warehouses, storage farms (i.e. "cloud storage"), and similar technologies. However, herein a large data set will refer to any system whose primary storage (RAM) is insufficient to retain the entirety of the data set.

Statement of the problem

As mentioned, storage access is rated in IOPS or blocks per second which can be read or written to/from main memory (RAM). This performance is governed by the storage device

hardware itself. Software optimizations and strategies, such as database indexes, are forced to focus on techniques to reduce the number of block accesses to accomplish a given task, such as fetching a record.

A common method of optimization is the caching of blocks of data in memory. Once a block of data has been read in, future reads can happen in a fraction of the time because main memory is significantly faster than non-volatile storage devices (including solid-state drives). Unfortunately, main memory is a finite resource – typically orders of magnitude smaller than non-volatile storage – which precludes a system from simply caching all the data from the storage devices at once. Several techniques called Cache Replacement Strategies (or Policies) exist to help decide which buffered blocks of data (herein: buffers) are most likely to be used again and are therefore good candidates for keeping in main memory after their initial requestor is done using them.

Currently, cache replacement strategies keep buffers "raw" (uncompressed) in main memory, which is potentially limiting maximal performance. Selectively compressing buffers could offer higher cache throughput. Compression uses CPU and Memory resources, making its performance independent of storage, hence allowing a trade-off. This project aims to analyze the following: since compression performance is limited by CPU and memory resources, then the advancements in software and CPU/RAM hardware to-date (2018) will make a compressed-cache strategy increasingly feasible compared to raw-buffer caches by improving cache hit ratios and throughput.

Objectives of the project

Overall feasibility of the hypothesis is the ultimate objective and is defined as the ability to provide more pages (buffers) per time unit to an application than current raw-only cache strategies provide. Several measurements will be taken as evidence but Hit/Miss Ratio (i.e. Page Faults) will be the primary determinate variable initially. Indeterminate factors such as available CPU/Memory resources and the use of compression will affect the hit/miss ratio, as will the data set size relative to the amount of resources, naturally. The objective will focus on the degree to which compression is able to affect hit ratios.

The second major measurement for the hypothesis will be time. Simply increasing the hit ratio by compressing data only proves that data is compressible and therefore more pages

(buffers) can fit into memory. The time measurement will prove whether the sacrifice of raw-buffer space for use in the compressed-buffer space, along with the time required to compress and decompress pages dynamically, leads to overall higher levels of throughput as measured by: pages per time unit.

While not strictly an objective for viability, it should be noted that compressors are the engines and algorithms which index and encode data into compressed and decompressed form (e.g. zlib, lzma, etc.), and are therefore a major facet of this research. Modern compressors are flexible and offer drastically different performance characteristics than those of old, which will affect the results of the case study later. Understanding current performance, past performance and options of compressors is necessary to help draw conclusions on the hypothesis. Existing technologies such as video cards and ASIC/FPGA technology will also be researched to help indicate possible transitivity with dedicated compression technology.

CHAPTER 2

LITERATURE REVIEW

Flash-y Tricks: The Rise of the Solid-State Drive

Solid-state drives (SSDs) gave the hardware world a simple, powerful shot-in-the-arm solution to IOPS limitations in the early 1990s by switching from costly, sensitive DRAM SSD modules to flash-based SSDs. SSDs have improved over the years to support IOPS ratings in the tens, and even hundreds of thousands. Sequential access remains only marginally better than traditional platter-based hard drives for most SSDs but is hundreds or thousands of times faster for random IO patterns ("Solid State Drives," 2018).

Individual SSDs are still drastically slower than typical PC Dynamic RAM and massively inferior to specialty memory types such as GDDR, HBM, and HMC video RAM ("List of Interface Bit Rates," 2018).

Storage manufacturers and vendors now offer dedicated storage arrays of drives that boast more than 1,000,000 IOPS and 150 GB/sec transfer rates ("VMAX All Flash," 2018). However, these solutions separate the storage device from the server and use a network fabric to connect them, such as iSCSI or Fibre Channel. This results in an additional latency compared to local storage access. Additionally, these high-end solutions are still slower than many modern video RAM modules ("List of Interface Bit Rates," 2018).

Traditional Buffer Replacement Strategies and Solutions

Caching strategies attempt to solve the storage IO bottleneck problem by keeping copies of data in RAM inside logical buffer pools for reuse after they are initially read in by an application. These strategies employ simplistic algorithms – for time saving purposes – to decide which buffers are most likely to be used again, and which ones should be evicted when the buffer pool is full. Common strategies include: First-In/First-Out (FIFO), Last-In/Last-First-Out (LIFO), Least Recently Used (LRU), Least Frequently Used (LFU), clock-sweep,

and several others (“Cache Replacement Policies,” 2018). Almost all strategies attempt to increase cache hit ratios by trading latency for efficiency and by avoiding pitfalls regarding recency and frequency:

- Evicting a less-popular buffer over a more-popular buffer.
- Evicting a buffer that will be used again later instead of one that will be used sooner.
- Preventing pool “flushing” or “pollution” when large scans of data happen.
- Tracking and using buffer statistics to keep more efficient buffers in the pool.
- Etc.

Caching strategies make eviction choices based on predefined behavior and/or observations of data (“Cache Replacement Policies,” 2018). Their hit ratio performance is naturally improved by relying on inherent data access patterns matching ideas such as the Pareto Principle. The Pareto Principle – often called the 80-20 Rule – is a generalization that says “… roughly 80% of the effects come from 20% of the causes” (“Pareto Principle,” 2018). In a data system this would translate to 20% of the data being requested or used 80% of the time. To say it conversely: if data access for a given set is evenly distributed, then any caching strategy will be significantly hindered because recency and frequency will be equivalent among all data; specifically, optimizations to hit ratios will fall off and overall hit ratios will worsen as the proportion of the data set vs the available RAM increases, as seen below.

No single power law will perfectly describe all data access, but the 80/20 distribution is used in this report to demonstrate its effect compared to even distributions, where applicable.

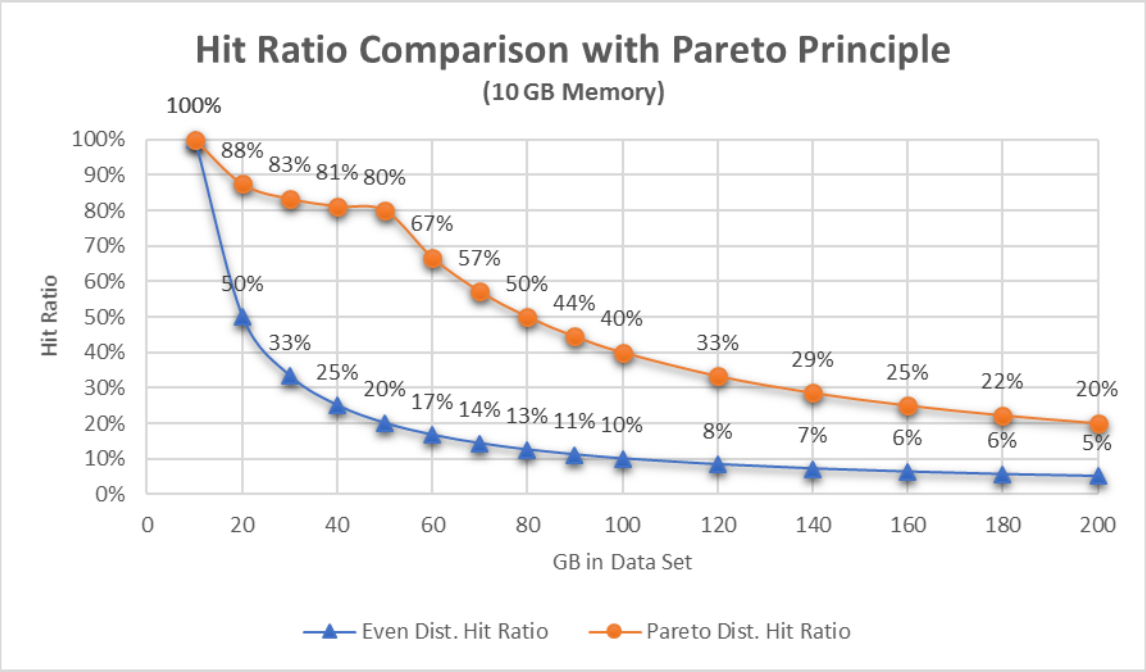


Figure 1a: Hit Ratios for Even vs Pareto Distribution (10GB RAM)

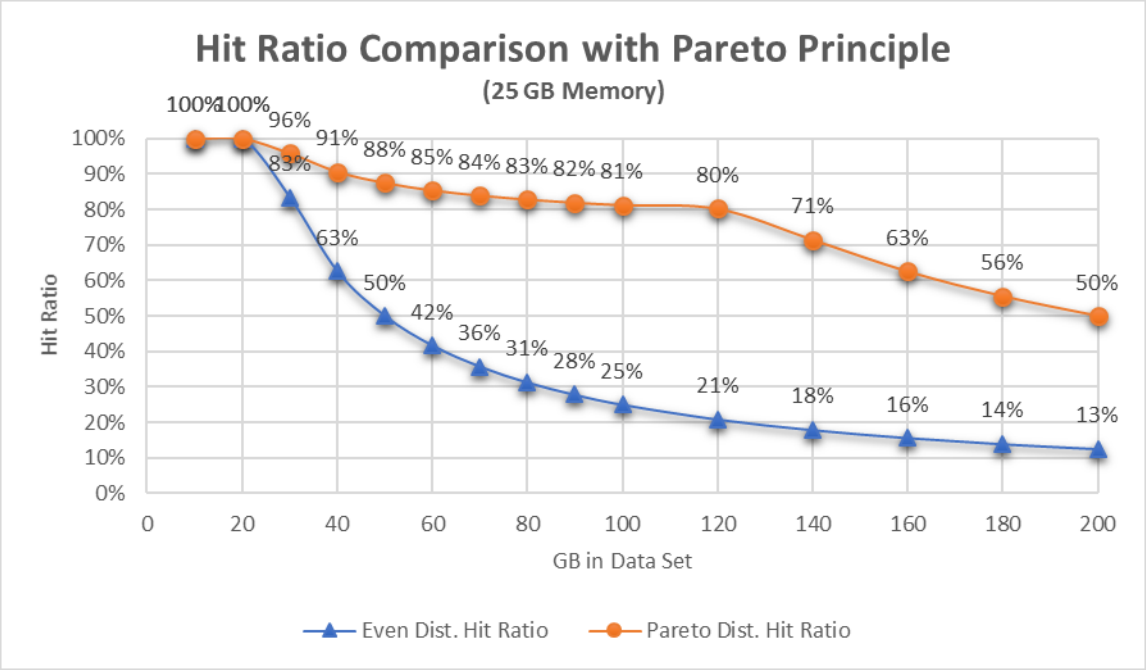


Figure 1b: Hit Ratios for Even vs Pareto Distribution (25GB RAM)

Physical Data Organization

The physical layout of data in storage devices attempts to reduce random IO patterns by putting logically sequential data into physically sequential blocks on-disk. This strategy does not reduce disk IO, rather it improves performance by allowing storage devices to operate at peak effectiveness: sequential IO. Sequential read (and write) access can be hundreds of times faster than random read/write access on a given storage device, as noted in the table below. The reasoning for this varies: traditional hard drives have seek time as the physical platters and heads line up, SSDs have switching logic and refresh intervals that have to line up, etc. Several filesystems support a concept known as "extents" which allows a generic system call for a file to result in a contiguous file on-disk (i.e.: no fragmentation) (Pomeranz, 2010).

Access Type	**Hard Drive (7200 rpm)**	**Solid-State Drive**	**RAM Drive**
Sequential Read	44,745	66,661	711,135
Random Read	188	12,249	613,215
Sequential Write	42,154	18,406	679,464
Random Write	364	11,056	554,294

Table 1: Example 4K IO Ratings for 7200 RPM HDD and NAND SSD

Table 1 demonstrates that even inexpensive NAND-based solid-state drives – like the one used in these tests – are significantly faster at sequential access patterns than random ones. However, a RAM drive is included for comparison with a device built specifically for efficient random access.
(Note: Chapter 4 "Case Study" delves into drive IOPS more fully.)

Logical Organization - Indexing

Logical organization techniques for structured data, such as relational databases and document-based datastores, reduce the number of block accesses required when attempting to find information. For example, an application might use a Skip List data structure to keep track of buffers in a pool, but when a page fault occurs it can rely on a precomputed index

stored separately on-disk for more efficient retrieval of the page. In a database system, this index is often a tree (e.g. balanced tree); a document-based datastore such as Solr (which leverages Lucene indexing) will use inverted indexes.

The advantage of these logical structures is faster access to values based on different keys or attributes. For example, a 100 MB database table with a 4K block-size would have at least 25,000 blocks (ignoring fill-ratio and overhead). A full scan of that table to find a given record with a unique attribute would require the average case of N/2 (12,500) block accesses. An index, even one with a terrible blocking-factor, could reduce that to just a handful block accesses.

The obvious disadvantage is the CPU time required to compute and maintain each index, as well as the disk storage and IO (block accesses) required to persist the index on-disk. Each time the data changes all indexes will require updating.

IBM Active Memory Expansion (AME)

IBM offers a feature on their AIX systems, starting with version 6, called Active Memory Expansion (AME). AME offers memory analysis tools and transparent compression of data and program pages in RAM. AME is the closest solution to what this project attempts to research. IBM reports that memory is broken into two pools (compressed & uncompressed) and data is moved between them as it becomes more or less popular ("Active Memory Expansion," n.d.). A presentation was released by IBM (Griffiths, 2012) which outlines the "Expansion Factor" concept: a user simply specifies a ratio of how much perceived memory they want vs how much they actually have. For example, a factor of 1.5 on a system with 10GB of RAM would result in that system using compression of pages until 15GB of data (compressed + raw) was fitted into RAM.

Unfortunately, AME is proprietary which precludes delving into the details of its operation. Other than a generic Expansion Factor, nothing seems to be tunable or configurable (e.g.: compressor, statistic weights, etc.). Regardless, the Power7+ and Power8 systems added dedicated compression circuitry to optimize the compression and decompression cycles by 90% (Cler, 2015), which implies the AME technology is working and being used. Additionally, IBM's decision to include ASIC components lends credence to ASIC's ability to narrow the feasibility gap to the proposed hypothesis later.

One important note: AME will compress both data and program pages such as stack and heap (Griffiths, 2012). The time-memory trade-off described in this paper emphasizes compression of in-memory blocks of on-disk data and therefore creates a distinction when comparing the two. For a better comparison, additional research would be needed to discover how AME analyzes and selects pages for compression, and the impact it has.

Application-Specific Technologies: Video Gaming

Video games sometimes encounter a similar problem to other large data systems: there is more texture and asset data to load into the video card's memory than is available. When a video card runs out of memory it must evict older texture/asset data and pull in new data from system RAM or worse: the disk! This is another example of a cache replacement.

The difference in this example is that video cards use asset data so intensely they are hindered simply by waiting for system memory. A video card's own memory (e.g.: GDDR or HBM) is designed to operate at speeds several times faster than standard PC DDR memory. A "cache miss" for asset or texture data can drag performance (in this case, frame rate) down considerably.

Video cards often support games running at 60, 100, or even 144 frames per second, giving the video card 17, 10, or 7 milliseconds, respectively, to transfer and compute all the graphical data required by a scene. A delay of a single millisecond to swap texture/asset data for each frame would be simply devastating to its operating performance. To reduce the chances of having to replace cached data (similar to a page fault in generic caching) some video card manufacturers and game developers employ asset compression algorithms such as Adaptive Scalable Texture Compression (ASTC) (Chait, n.d.).

ASTC and other texture compressors are admittedly lossy: a compressed texture cannot be decompressed into its original form (it remains original on-disk of course). In video gaming this is acceptable because the loss of quality can be controlled and minimized, reducing the perceived impact to the user (Chait, n.d.). Regardless of its lossy nature, this is another example of compression being used to keep more data in-memory to avoid invoking replacement logic from a slower data storage medium (system memory or disk).

Filesystem Compression

Storage devices have physical sections for storing data in predefined chunks, often 512 bytes or 4KB. Filesystems create logical blocks of these chunks in integral powers of 2: 4KB, 8KB, 16KB, etc. Some filesystems, such as ZFS, are able to transparently compress data that is written to or read from the device in an effort to reduce the number of blocks required for a given piece of data ("ZFS," 2018). For example, a PostgreSQL database with an 8KB page of easily compressible data could be transparently compressed to 2-3KB and therefore stored inside a single block in the filesystem instead of two blocks. ZFS uses light-weight algorithms such as lzjb and lz4 to reduce the CPU overhead – and subsequent latency – of the transparent compression process ("ZFS," 2018).

The described technique is not purely theoretical: the PostgreSQL gurus at CitusData demonstrated the ability to use a compressed filesystem to relieve IO pressure with ZFS, reducing disk usage and improving query performance (Steinbach, 2013). Oracle released a blog article in 2009 claiming similar findings with Oracle DB and ZFS with transparent compression enabled; albeit weakly described ("ZFS Compression – A Win-Win," 2009). Chittenden recommends always using ZFS compression with PostgreSQL: citing an average compression ratio of 2.8:1 across several petabytes of data from multiple environments and sources (Chittenden, 2017).

CHAPTER 3

SYSTEM DESIGN (RESEARCH METHODOLOGY)

Introductory Note(s)

Data units are often defined inconsistently and interchanged which can be misleading. For example, “kilobyte” can be defined as 1024 (2^10) or 1000 (10^3). This project will deal with small quantities of data at times and being off by even a few bytes can skew results. We will follow the International System of Units (SI) definitions and always work in powers of 2. References to “kilobyte”, “KB”, “KiB”, and similar variations will mean “kibibyte”, or simply 1024 bytes.

Data Set

For consistency and practicality, the selected data set for testing must address real-world formats and patterns. Several data corpora exist for testing lossless compression, including the Calgary corpus (circa 1989), the two Canterbury corpora (circa 1997), and the Silesia corpus. The Silesia corpus was created most recently by researchers at the Silesian University of Technology in Poland and is intended to solve unfair balances in the previous corpora: too-small file sizes, emphasizing English text over other languages, lack of database or medical record data, et cetera (Deorowicz, n.d.).

Data collection for this project will use the Silesia corpus. Additionally, index data (typically btrees) will be taken from a PostgreSQL database, since most large data sets employ indexing and the Silesia corpus does not contain any. The database is a publicly available backup of the Mouse Genome Database (MGD) and was selected for its size and unique column-values to avoid repetition which would artificially inflate compression performance. Two of the indexes are numeric in nature, the last is text based, as described in the following table.

Index	Uniqueness	Avg Size	Total Size	% of Corpus
Integer field	9% (11:1)	8 bytes	17768 KB	6.2%
Timestamp field	86% (1.2:1)	8 bytes	17768 KB	6.2%
Text field	85% (1.2:1)	48 bytes	55624 KB	18.8%
Total			91160 KB	31.2%

Table 2. Supplemental Index Data to Silesia Corpus.

Compressor Selection

Dozens of compressors exist, but only three will be researched: deflate (via zlib), Zstandard, and lz4. Several compressors, such as lzma, are heavy-weight regarding CPU-time and RAM requirements and are therefore impractical. Other compressors, such as Google's brotli and lzop are reasonable options, but superfluous for this testing.

Deflate will be implemented via (and referred to as) zlib, which most people also know as "gzip". It was selected for its ubiquity in fast-paced environments, including web-page compression. It is an older compression library but is still very popular and memory efficient; generally, under 1 MB for compression or decompression ("Quick Benchmark," 2016).

Zstandard (herein: zstd) is a modern compressor developed by Facebook employees Yann Collet and Chip Turner after Yann's creation of lz4. It was selected for its inclusion and emphasis on data compression optimizations to-date (Collet, 2016). Zstd meets or exceeds the compression metrics of zlib while operating at 2 to 5 times the speed thanks to leveraging ALU optimizations, avoiding pipeline flushing with reduced (or eliminated) branching, and a newer Huffman decoder (Collet, 2016). These optimizations will contrast zstd with zlib to answer whether compression software is getting smarter and faster, and if so, to what extent. Also, despite zstd's support for larger window sizes, it continues to use only a few MB of memory for compression.

The final selected compressor is lz4. It was selected for comparison due to its emphasis on speed: possibly reaching the limits of RAM performance on a multi-CPU/core system like the one this project uses. LZ4's exact memory usage is a compile-time setting (LZ4_MEMORY_USAGE) which defaults to 16KB which falls well below the threshold for

significance regarding memory usage (source: lz4 source code, lib/lz4.h). LZ4's reduced memory footprint for dictionary encoding provides much of its speed due to more (if not all) of the lookup table fitting into a CPU's L1 (very fast) cache. Additionally, the exceptionally small memory footprint makes the LZ4 a good candidate for analysis on ASIC and FPGA chips later in this report.

Storage Device Performance Metrics

Storage devices often offer performance ratings from the manufacturer which are "synthetic" in nature; they are not achievable in real-world workloads. Benchmarking the data rates (bytes/sec) and operation rates (IOPS) across multiple device types in a consistent manner will help establish baseline performance throughput. All benchmarks will be taken on a dedicated system using the Flexible I/O Tester (herein: FIO) (Axboe, n.d.). Emphasis will be placed on benchmarking and comparing a standard platter-based hard drive and an SSD.

A note about high(er)-performance devices. Enterprise-level devices like SAN shelves and the Samsung PM1725a boast random-read 4KB IOPS values exceeding 1,000,000 ("Introducing the Samsung PM1725a…," 2017). These are synthetic tests that push the limit of the hardware to understand where its physical boundaries are under optimal conditions (e.g.: ideal concurrency / thread count). They will not be comparable to the FIO test results, which include overhead from system calls to the operating system and filesystem overhead. Taking these device ratings at face-value would mean they are capable of sub microsecond 4K random-read IO. This value does not match up with their published Quality of Service ratings of 95 microseconds for a 4KB random-read ("Introducing the Samsung PM1725a…," 2017) or the Dell VMax published latency of 350 microseconds ("VMax All Flash," n.d.). Rather than speculate on their comparable performance, this will remain "additional research" if/when such hardware is available for benchmarking in a controlled environment.

Compression Characteristics and Performance Metrics

Compression algorithms and techniques evolved to address several data formats and resource requirements. A thorough benchmark of key compression performance characteristics across several compressors will measure whether they are capable of operating

at a level that exceeds storage IO performance. Compression is measured differently in the industry, since the idea of a "block" does not mean the same thing. Compression will be measured in two ways: compression ratio and data rate.

Compression ratios represent the number of raw bytes (uncompressed data) compared to compressed bytes. Compression ratio is sometimes called "efficiency" or "compressibility", which are misnomers because as compression gets tighter (read: better) these values goes down, which is illogical. Therefore, this document will always refer to the effectiveness as: compression ratio. Multiple compressors will compress the data corpus to build a table of compression ratios. The table below will be used to compare the extrapolated effect on the hypothesis's first metric (hit ratio) in hypothetical buffer pools with splits between raw and compressed spaces. Below are some examples of how a compression ratio will look:

- 20,000 raw bytes compressed to 5,000 bytes ➔ ratio of 4:1
- 20,000 raw bytes compressed to 2,000 bytes ➔ ratio of 10:1
- Et cetera

Data rate is the number of bytes per second a compressor can achieve using a given set of hardware. This metric includes an important factor that sheer compression ratios do not and is also a requirement for the second metric of the hypothesis: time. Data rates for both compression and decompression will be measured on a dedicated system, as they are wildly different for most compressors (decompression is much faster than compression, typically). These values will be used for the second metric of the hypothesis, overall throughput which serves to help determine feasibility, by discovering the time required to achieve the compression targets compared to eschewing them and always page faulting.

A note about memory usage: Memory use is an additional consideration for "expensive" compressors (e.g.: lzma) since the memory they require would take away total memory in a given system, which could have otherwise been used for raw buffers. For this report, compressor memory requirements will be ignored unless they exceed 10MB per thread/process. The selected compressors (zlib, zstd, and lz4) all fall under that mark.

A custom program will be written in C to gather measurements. Since hardware will affect performance, a dedicated system will be built and used for consistency and fairness. Several compression benchmarking programs exist, but none will break the data corpus into

fixed-size blocks the way a cache would be constructed (e.g.: 4KB, 8KB, 16KB, etc.). Additionally, parallelization of the workload must be finely controlled, which is not possible in most benchmarking tools.

Ultimately, if compressors cannot out-pace storage IO in key metrics, then the hypothesis postulated is not currently viable. However, if such a case exists we will continue to research historical compression performance trends to conclude the remainder of the question implied by the hypothesis: is a compressed cache strategy ever likely to be viable? (See: ASIC/FPGA).

ASIC and FPGA Extrapolation and Transitivity

Chapter 2 outlined technologies boosted by the switch from a general-purpose processor to a RISC, FPGA, or ASIC processor. Given the overall simplicity of the selected compressor algorithms, FPGAs and ASICs could offload the key component of the proposed technique (IBM's AME technology already uses this approach). FPGA and ASIC performance will be approximated and used to extrapolate the presumed transitive effects on the proposed system.

Cache Replacement Strategy Effectiveness

All cache replacement strategies attempt to be clairvoyant with victim selection: balancing recency and frequency. However, their inner workings for selection are outside the scope of this research. The use of compression certainly opens the opportunity for more sophisticated statistics for victim selection due to the compression metrics mentioned above: ratio and data rate (faster and tighter == better buffer for retention). But again, this research will presume equal replacement strategy logic for traditional pools and compressed pools.

Since victim selection algorithms (e.g. LRU or Clock) will remain equivalent, a full-blown program loading and moving buffers in a manner described herein is unnecessary. Results will instead be broken into pieces and a probabilistic approach will be taken to analyzing throughput. In short, each step in the selection logic will be evaluated separately where possible and individualized costs for raw vs compressed pools will be compared to determine throughput.

The following flowcharts demonstrate traditional buffer replacement compared to the logic proposed in this project from the viewpoint of cache replacement.

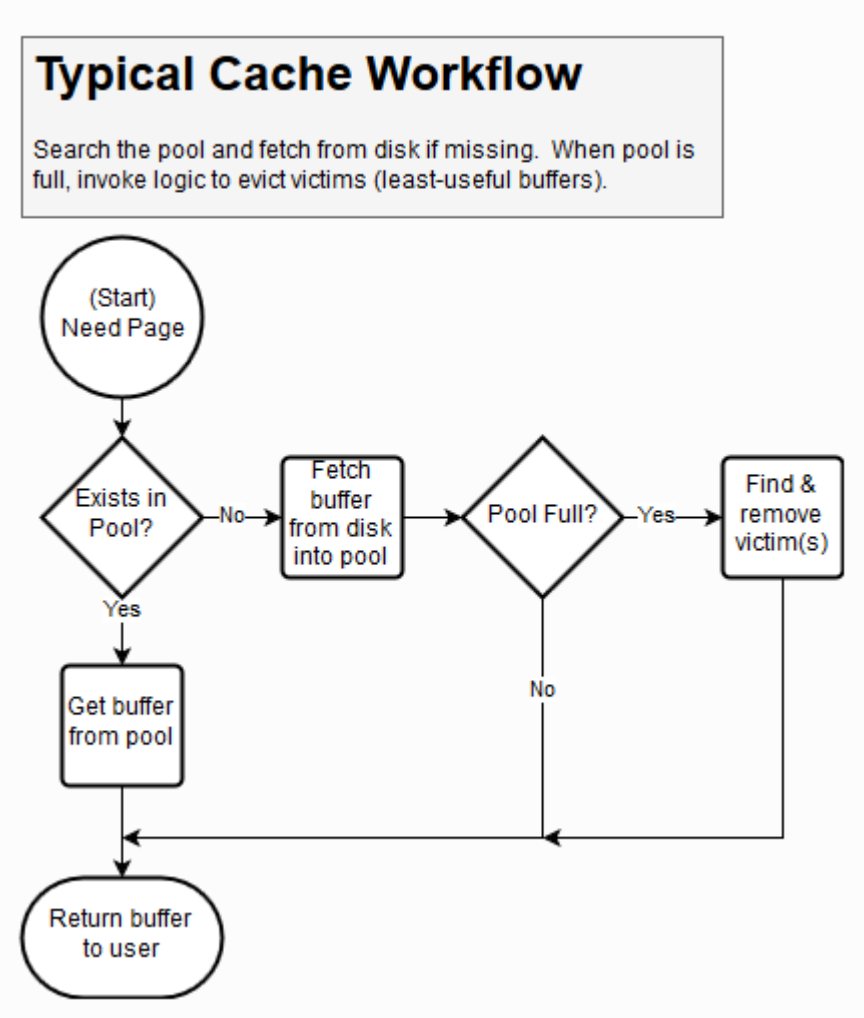


Figure 2a: Typical Cache Fetching and Replacement Flowchart

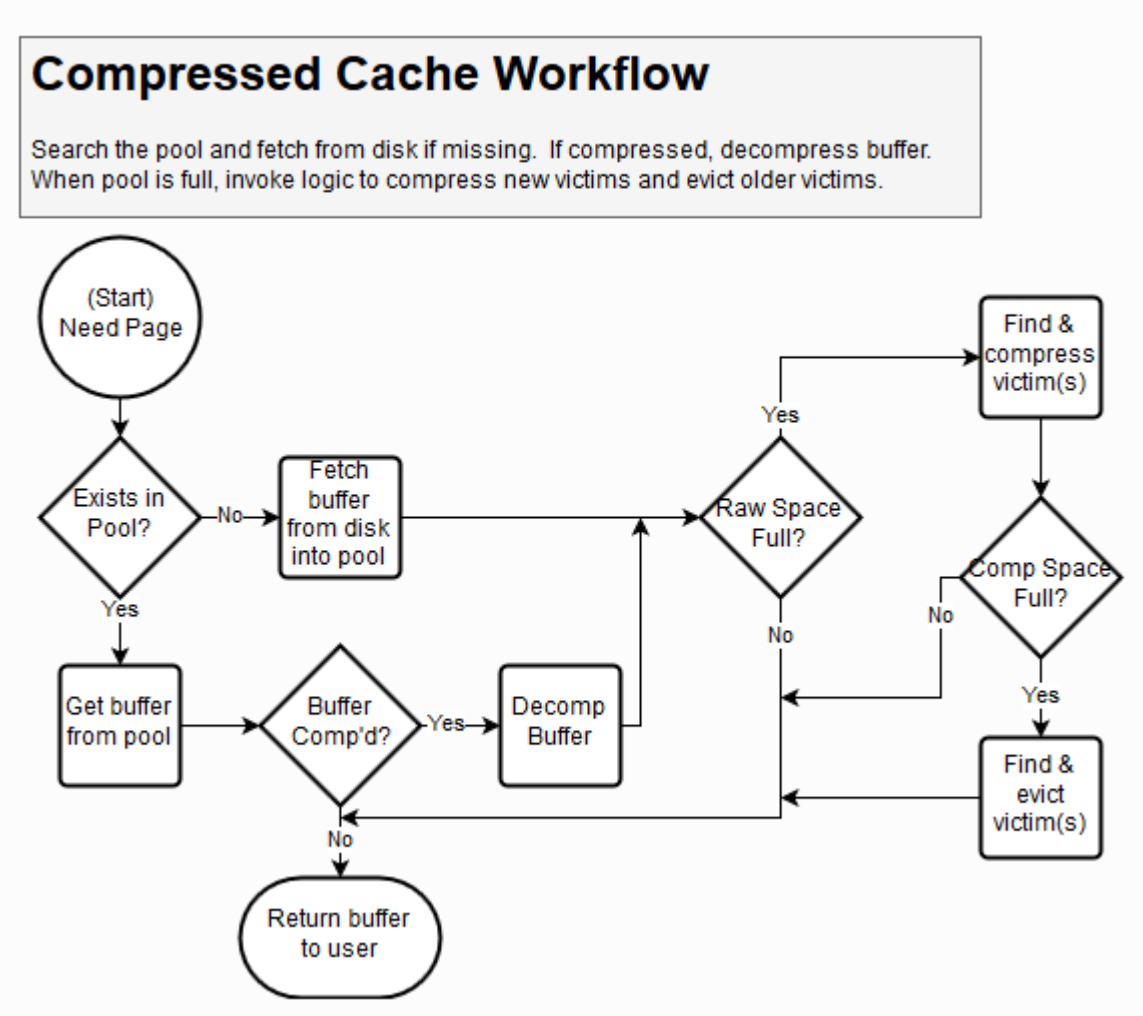


Figure 2b: Compressed Cache Fetching and Replacement Flowchart

The flowchats above outline additional logic to account for when attempting a compressed cache workflow, in addition to the cost of compressing and decompressing data. The following sections will describe each area of cost, whether it is considered significant, and why or why not.

Note: the following areas need additional research to confirm their accuracy. The information is based on generalized impressions and evidence, sometimes without quanitfiable values due to varying hardware and data.

Branch Statements (if…then, cmp…jmp)

A compressed cached strategy requires additional logic for each cache hit to determine if the buffer is compressed. If so, then another conditional is required to determine if the

decompressed size will exceed the pool limit. While if…then statements impede compiler and CPU optimization techniques, which ultimately affect pipeline performance via branch mispredictions, they will be considered insignificant. This is not to say they are free, and further research might need done to better quanitfy the impact. However, this author suspects the cost would be on the order of a few nanoseconds on a modern (2018) processor. Furthermore, branch mispredictions (the truly costly part of an if…else) will likely remain low since the system will know the amount of the pool given to compressed buffers, along with the overall compression ratio being achieved. Additionally, the majority of decompressions will not result in a full pool if victims are chosen in bulk (as described later in Chapter 4).

Search Time

The potential to increase the cache hit ratio happens because more buffers are packed into the available memory. While modern searching on ordered data structures is significantly cheaper than scanning sequentially (i.e.: linear search), it is still not free. Data structures which allow binary searching (including probablistic ones such as skip lists) minimize the impact of searching a large data set. Specifically, they are O(log n) average case which is graphed below:

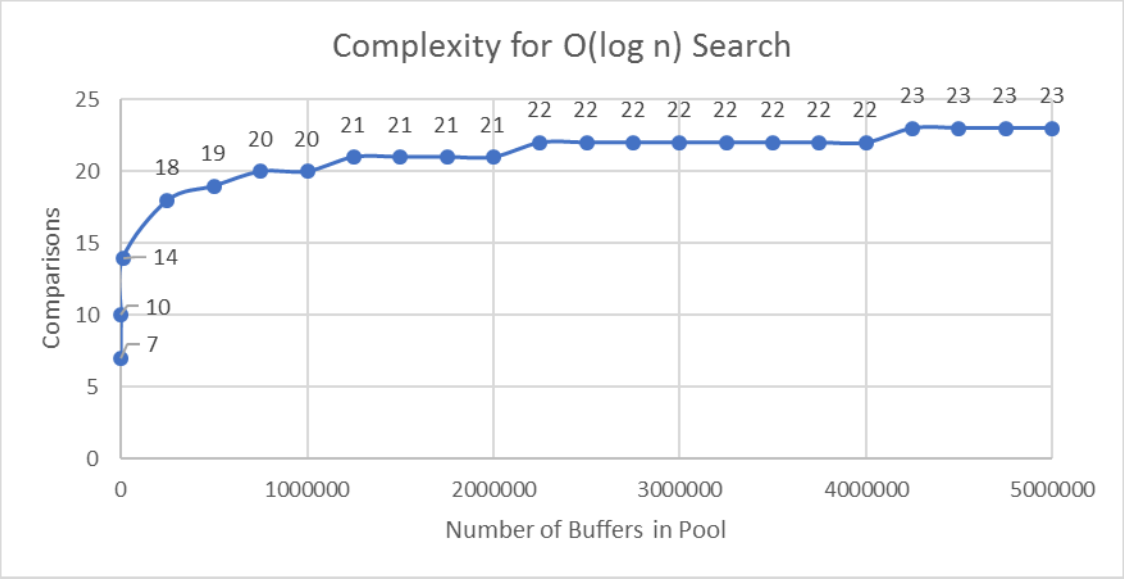


Figure 3: Comparison/Complexity for Searching Larger Pool Counts

For small pools the increase in buffer count will have significant effect. However, the effect diminishes; for example, doubling from 1,000,000 entries (8GB with 8KB buffers) to 2,000,000 entries (16GB) only changes complexity from 20 comparisons to 21. Therefore, this research will consider the additional time insignificant.

Caching Strategy

The actual caching strategy for victim selection (e.g.: LRU, Clock, ARC, et cetera) is of no consequence, nor is a particular strategy compulsory. It is possible, even likely that the two sets of buffers – raw and compressed – would use separate strategies since the compressed space is already low-recency. However, it is not mandatory and whichever strategy was selected for a raw-only cache could also be used with the compressed cache.

An area of research outside the scope of this document would be: what additional statistics could be captured and leveraged to make more intelligent victim choices with a compressed cache strategy.

Cache Hit

Cache hits are not technically free, everything requires instructions. However, this cost is not only low (see Search Time above), but also uniform for identical Cache Replacement Strategies. Therefore, cache hits will be considered 0 µsec cost.

Stolen Time

The tradeoff analyzed assumes the CPUs are dedicated to relieving IO pressure on the disk. No costs will be incurred for concepts along the line of: “well if you steal CPU from the DB service then the system runs slower…”

CHAPTER 4

CASE STUDY (RESULTS AND DISCUSSION)

Introduction

The literature review gave several techniques used by modern systems to reduce the number of calls to the disk, thereby maximizing the available IOPS of a given storage device. The following sections will measure storage devices, compressors, and RAM to draw conclusions on whether further reductions in disk access can be had by means of compression without suffering overall throughput reductions due to the overhead in the time-memory trade-off.

Hit/Miss ratio will be a simple matter of compressing the selected corpus of data with various block sizes (matching common database and filesystem choices) and graphing the following functions:

- Raw Only (Control `T_Bytes`):
  - `f(T_Bytes) = (T_Bytes / B_Size) / N_Bufs`
- Raw + Compressed Split (Control `C_%`):
  - `g(C_%) = ((T_Bytes * (1 - C_%) + (T_Bytes * C_% * C_Ratio)) / B_Size / N_Bufs`

Where:

- T_Bytes is the Total Bytes available for the system to use for caching.
- B_Size is the size of a Buffer (aka Page of data).
- N_Bufs is the number of buffers in the data set.
- C_% is the percentage of memory to use to hold compressed buffers.
- C_Ratio is the overall compression ratio (e.g.: 3:1).

The improvement to Hit/Miss ratio will then be joined with data gathered by benchmarking the steps involved in cache selection (lookup time, comp/decomp operations, storage IO, etc.) to determine throughput, which is the primary indicator for the viability of the hypothesis set forth.

Disk Metrics

The following tables and figures will show drive performance on a dedicated test system (Appendix A). Several test configurations are listed below, but the two important ones are Mode and Sequential Ratio. Real-world workloads must contend with read-write patterns but adding the overhead of writing merely complicates the testing and does not impact what has been set out to be analyzed. Therefore, the storage devices will measure IOPS in read-only mode. Furthermore, the Sequential Ratio represents how many reads will be sequential blocks. For many data sets (table scans, documents, etc.) this could be high, whereas b-tree indexes might be quite small. Further analysis would be needed to pick an accurate percentage; however, PostgreSQL offers some values in its query planning documentation which indicate that, due to the cache holding most of the random-read data, sequential data might be upwards of 90% of page fault reads. Ergo, we will use 90% sequential reads ("Query Planning," n.d.).

Test configuration:

- Duration: 5 minutes per run, 3 runs each block size
- Block sizes: 4, 8, 16, 32, 64, and 128KB
- Program: FIO (direct mode, queue depth 4, libaio)
- File Size: 70% of Device (84 GB for SSD, 700GB for HDD).
- Mode: Random Read Only
- Sequential Ratio: 9:1 (90%)

	IOPS				**Bandwidth (MB/sec)**		
Block Size	HDD	SSD	RAM		HDD	SSD	RAM
4 KB	551	9380	697351		2	36	2723
8 KB	550	8642	513610		4	67	4012
16 KB	537	7800	332678		8	121	5197
32 KB	489	5899	197465		15	184	6170
64 KB	460	4437	108430		28	277	6776
128 KB	390	2651	57073		48	331	7133

Table 3: IOPS and Bandwidth Relative to Block Size

The table above demonstrates some expected results: the general purpose SSD is 5-20x faster than a typical 7200 RPM HDD device, and RAM is 10-100x faster than the SSD. RAM performance with larger blocks appears to drop off sharply, but this is due to the FIO program running in-memory and stealing bandwidth from the RAM Disk, which is expected behavior.

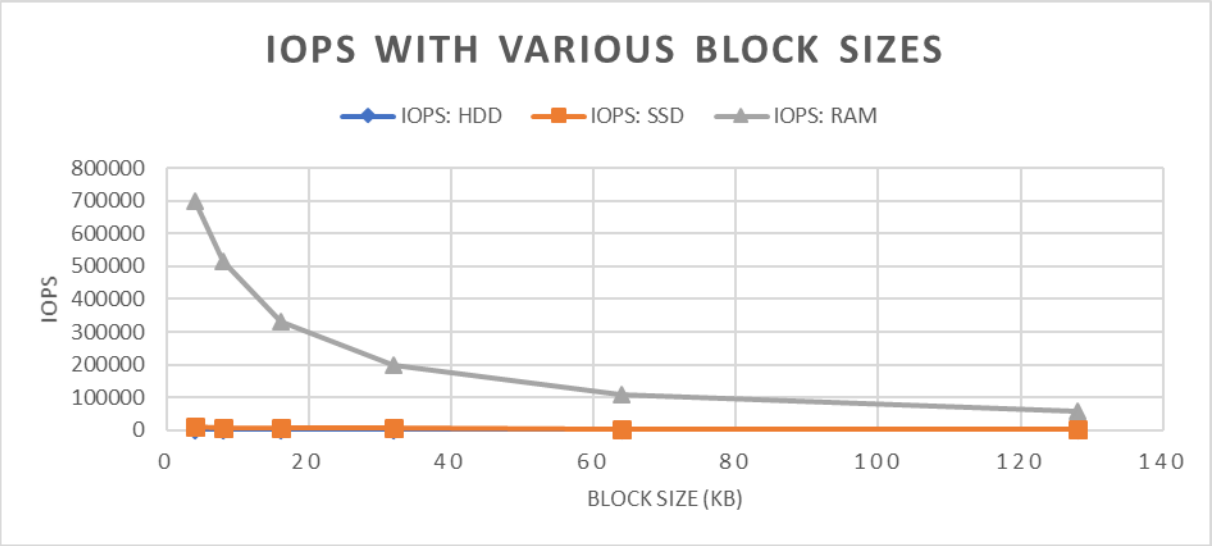


Figure 4a: IOPS with Various Block Sizes

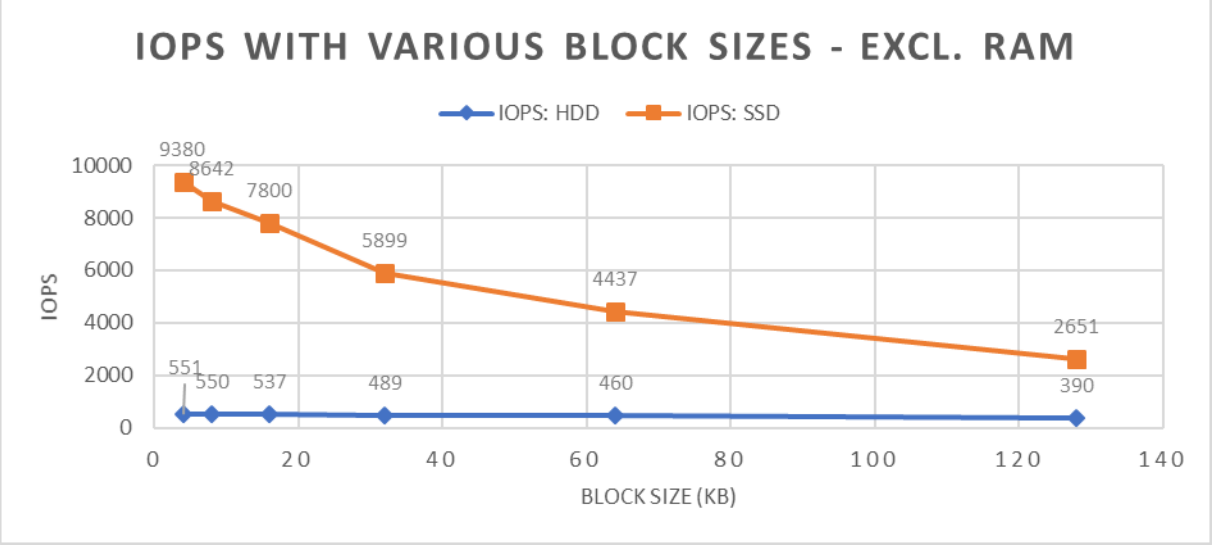


Figure 4b: IOPS with Various Block Sizes – Excluding RAM

While the 4KB block size offered the highest IOPS, most databases store data in larger page sizes: 8KB for PostgreSQL and Oracle, 16KB for InnoDB (mysql). The larger page sizes result in much higher bandwidth for only a minor hit to IOPS, as shown in the figure

below. The results show IOPS ratings of approximately 8,000 to 9,000, which translates to 110 – 125 microseconds per IO.

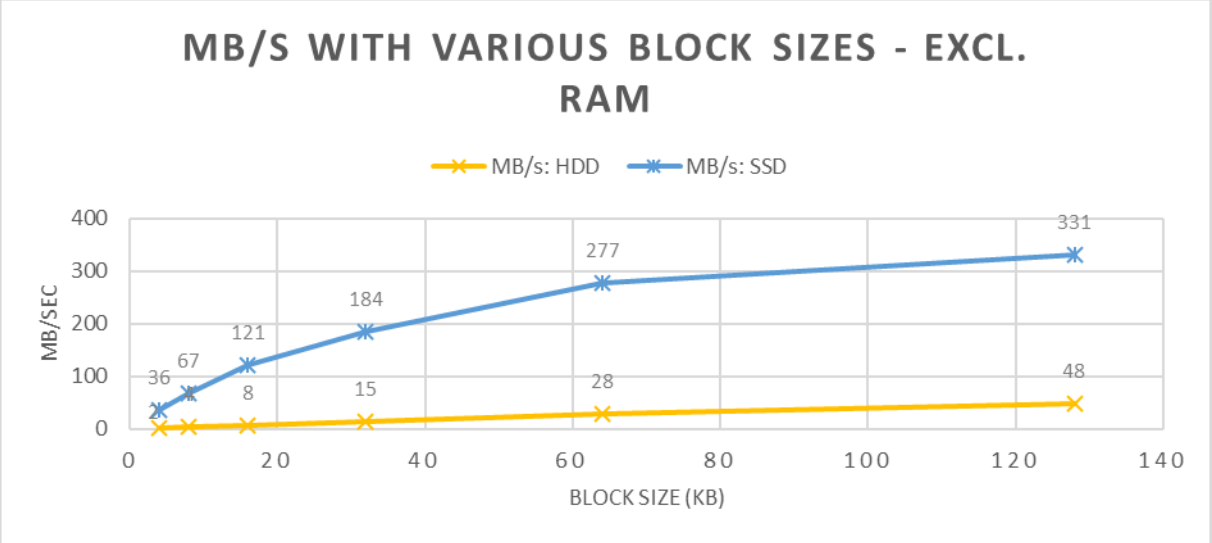


Figure 5: MB/Sec With Various Block Sizes – Excluding RAM

The performance tradeoff of IOPS and MB/Sec begins to fade around 8-16KB. Furthermore, since several large DB vendors use 8 or 16KB, that will be the de facto size for the remainder of this report. In this scenario the test system requires **110 microseconds** per I/O, using an SSD with a page size of 8KB, and will be the cost associated with a page fault for this report. This metric is one of the most important, as it **establishes the "time to beat"** to satisfy higher throughput. It also happens to align with externally-generated benchmarks (Boner, n.d.).

Compression Metrics

Before delving into the analysis of the two objectives (hit ratio and throughput) the compression ratio and the data rate of each compressor are required. Compression ratio is not variable when the same options are used, but data rate is. The test system used can be found in Appendix A.

Compression Ratio

Data compression varies by source size, to an extent: several small files will compress poorly compared to a single larger file containing the same data, typically. The following

tables and figures will show compression ratios for the prescribed compressors on the data corpus from chapter 3. The selected page size for this report is 8KB and follow-up tables will compare compression ratios for that block size as well.

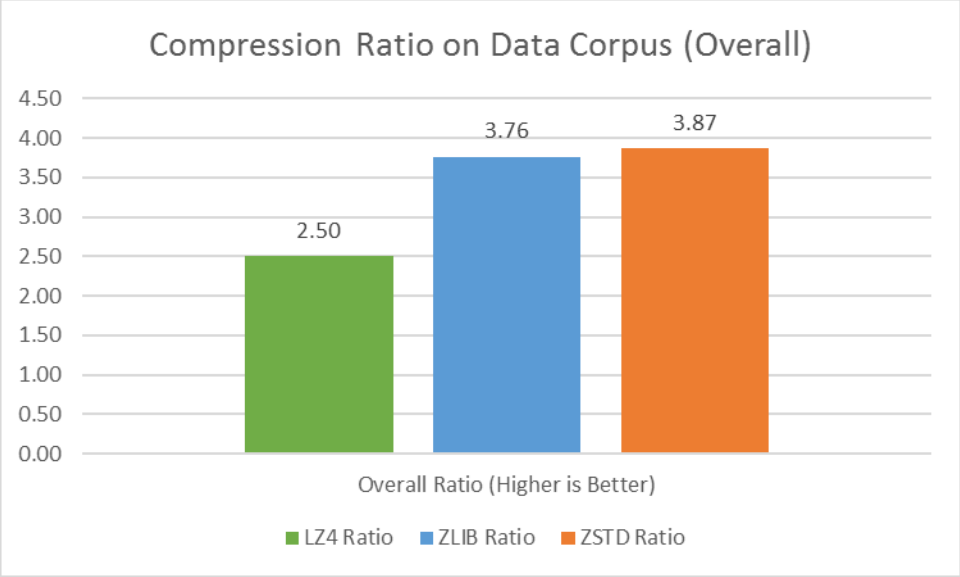
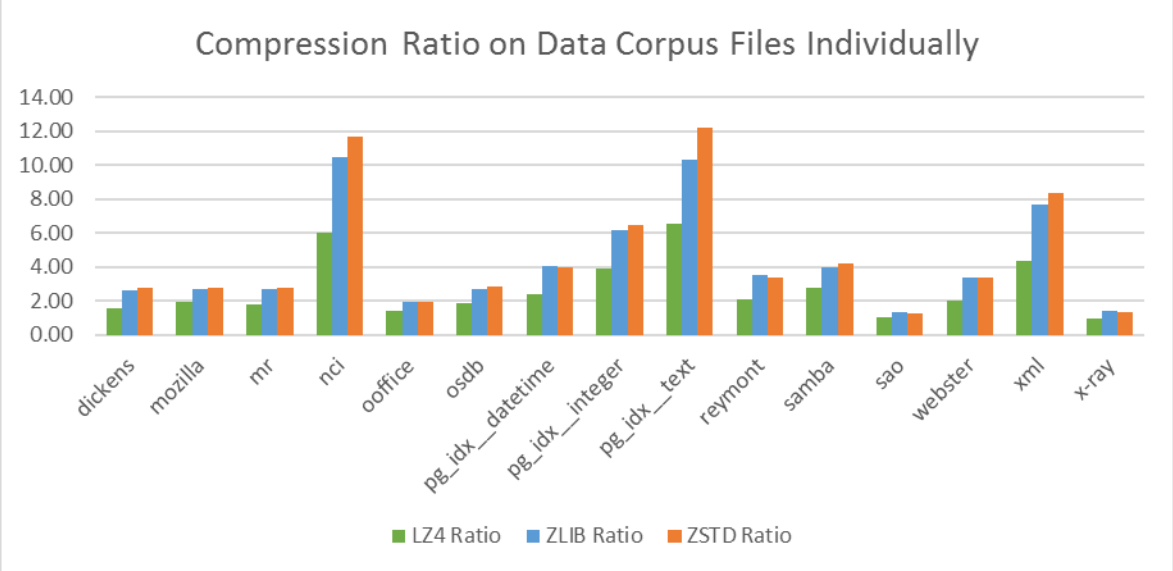


Figure 6a: Compression Ratio on Data Corpus (Overall)



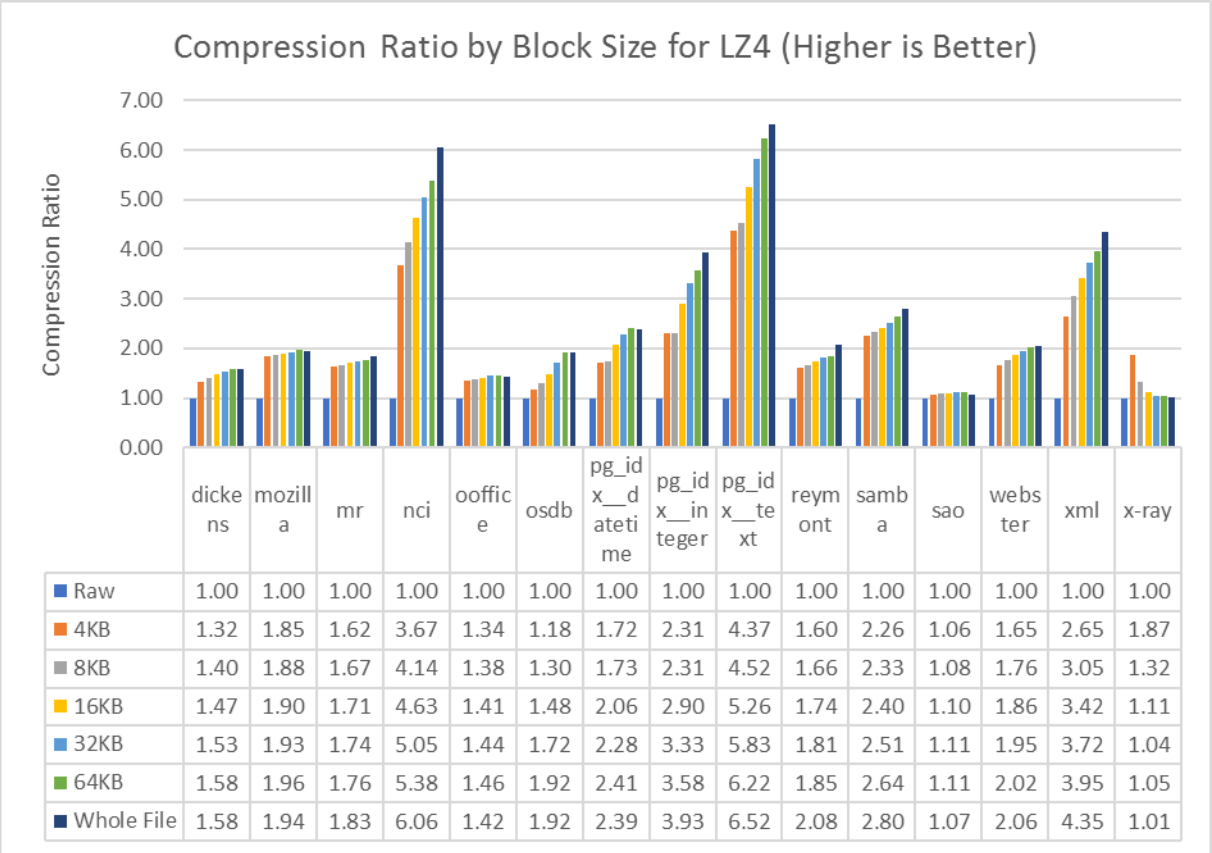
LZ4 Ratio	1.58	1.94	1.83	6.06	1.42	1.92	2.39	3.93	6.52	2.08	2.80	1.07	2.06	4.35	1.01
ZLIB Ratio	2.63	2.69	2.70	10.49	1.99	2.70	4.08	6.14	10.32	3.56	3.96	1.36	3.40	7.72	1.40
ZSTD Ratio	2.76	2.76	2.80	11.69	1.96	2.87	4.03	6.52	12.23	3.39	4.25	1.30	3.39	8.35	1.37

Figure 6b: Compression Ratio on Data Corpus Files Individually

The three compressors achieved overall compression ratios of 2.50, 3.76, and 3.87 for lz4, zlib, and zstd, respectively. As anticipated from the literature review in chapter 3, zstd's compression ratio is comparable to zlib in most cases. The resulting compression ratios range from 1.01 (almost no compression) to 12.23 (very good compression). Some observations about the ratios and compressor performance:

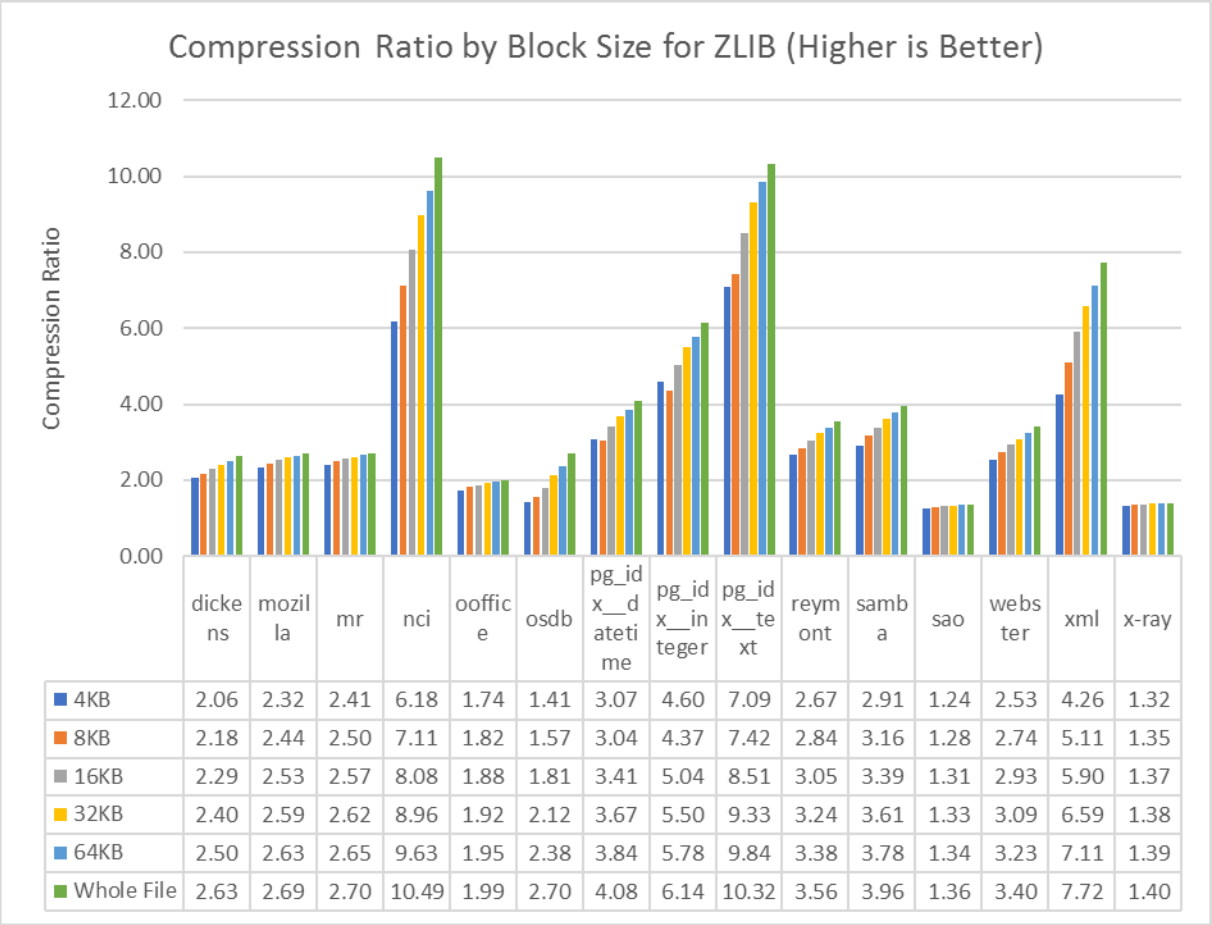
- Highly random data from the corpus, such as x-rays and the sao star catalog, were difficult to compress and achieved lower compression ratios; ranging from 1.01 to 1.4.
- LZ4 was consistently behind the other compressors, even on highly compressible data, which was expected due to its speed-centric design.
- Datetime index data was significantly less compressible (2-4x) compared to a similar integer index (4-6x), despite both being 8-byte numeric data types internally.
- Pattern-based data such as text-heavy indexes and data (e.g.: the NCI database) compress very well. This is good since text/varchar data is often larger than other data types.

Compression was performed on individual files, which should generally be the most ideal setup for a compressor. Compression ratios will typically fall when the data is broken into smaller pieces and compressed individually, due to less statistical (or dictionary) data being available for better pattern matching and symbol selection. The following tables and figures will now look at compression ratios when the data is broken into smaller pages.



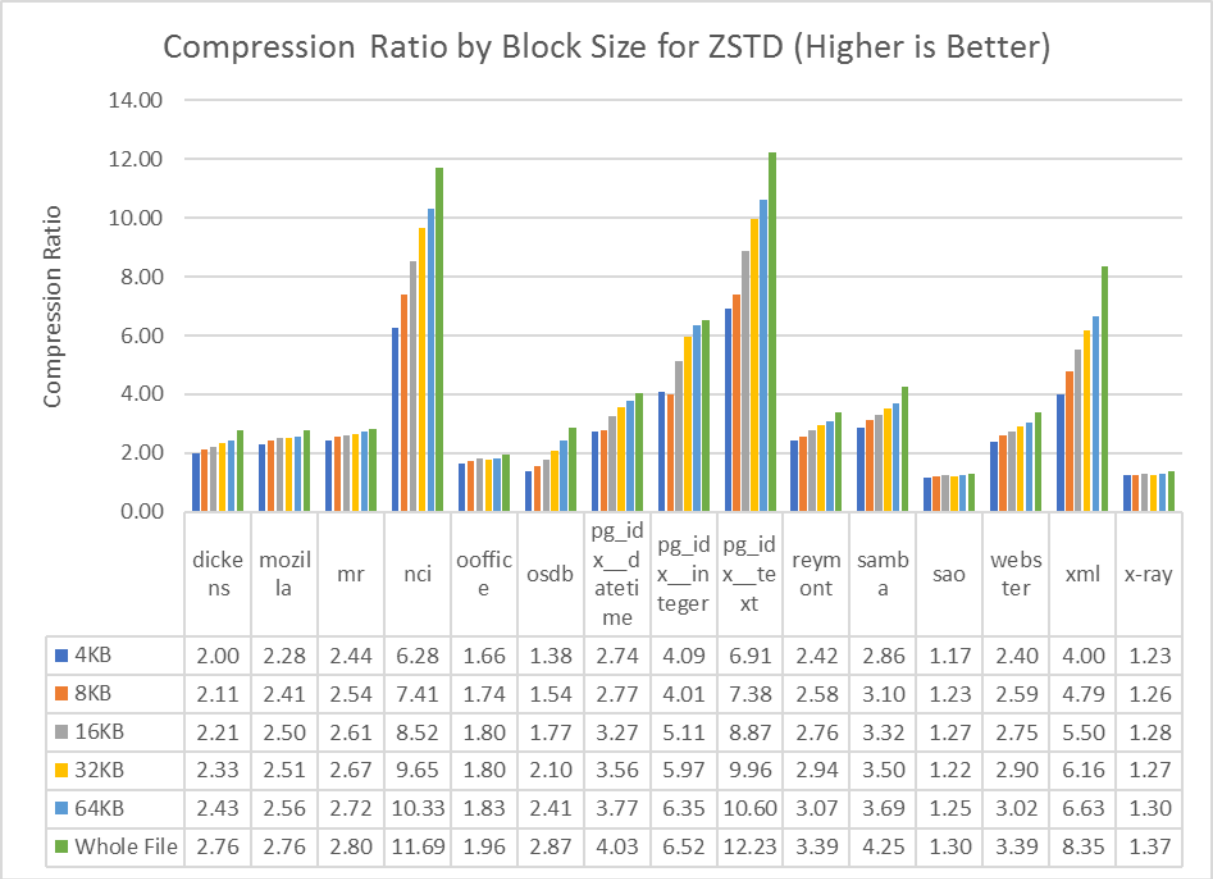
	dickens	mozilla	mr	nci	ooffice	osdb	pg_idx__datetime	pg_idx__integer	pg_idx__text	reymont	samba	sao	webster	xml	x-ray
Raw	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
4KB	1.32	1.85	1.62	3.67	1.34	1.18	1.72	2.31	4.37	1.60	2.26	1.06	1.65	2.65	1.87
8KB	1.40	1.88	1.67	4.14	1.38	1.30	1.73	2.31	4.52	1.66	2.33	1.08	1.76	3.05	1.32
16KB	1.47	1.90	1.71	4.63	1.41	1.48	2.06	2.90	5.26	1.74	2.40	1.10	1.86	3.42	1.11
32KB	1.53	1.93	1.74	5.05	1.44	1.72	2.28	3.33	5.83	1.81	2.51	1.11	1.95	3.72	1.04
64KB	1.58	1.96	1.76	5.38	1.46	1.92	2.41	3.58	6.22	1.85	2.64	1.11	2.02	3.95	1.05
Whole File	1.58	1.94	1.83	6.06	1.42	1.92	2.39	3.93	6.52	2.08	2.80	1.07	2.06	4.35	1.01

Figure 7a: LZ4 Compression Ratio by Block Size



	dickens	mozilla	mr	nci	ooffice	osdb	pg_idx__datetime	pg_idx__integer	pg_idx__text	reymont	samba	sao	webster	xml	x-ray
4KB	2.06	2.32	2.41	6.18	1.74	1.41	3.07	4.60	7.09	2.67	2.91	1.24	2.53	4.26	1.32
8KB	2.18	2.44	2.50	7.11	1.82	1.57	3.04	4.37	7.42	2.84	3.16	1.28	2.74	5.11	1.35
16KB	2.29	2.53	2.57	8.08	1.88	1.81	3.41	5.04	8.51	3.05	3.39	1.31	2.93	5.90	1.37
32KB	2.40	2.59	2.62	8.96	1.92	2.12	3.67	5.50	9.33	3.24	3.61	1.33	3.09	6.59	1.38
64KB	2.50	2.63	2.65	9.63	1.95	2.38	3.84	5.78	9.84	3.38	3.78	1.34	3.23	7.11	1.39
Whole File	2.63	2.69	2.70	10.49	1.99	2.70	4.08	6.14	10.32	3.56	3.96	1.36	3.40	7.72	1.40

Figure 7b: ZLIB Compression Ratio by Block Size



	dickens	mozilla	mr	nci	ooffice	osdb	pg_idx__datetime	pg_idx__integer	pg_idx__text	reymont	samba	sao	webster	xml	x-ray
4KB	2.00	2.28	2.44	6.28	1.66	1.38	2.74	4.09	6.91	2.42	2.86	1.17	2.40	4.00	1.23
8KB	2.11	2.41	2.54	7.41	1.74	1.54	2.77	4.01	7.38	2.58	3.10	1.23	2.59	4.79	1.26
16KB	2.21	2.50	2.61	8.52	1.80	1.77	3.27	5.11	8.87	2.76	3.32	1.27	2.75	5.50	1.28
32KB	2.33	2.51	2.67	9.65	1.80	2.10	3.56	5.97	9.96	2.94	3.50	1.22	2.90	6.16	1.27
64KB	2.43	2.56	2.72	10.33	1.83	2.41	3.77	6.35	10.60	3.07	3.69	1.25	3.02	6.63	1.30
Whole File	2.76	2.76	2.80	11.69	1.96	2.87	4.03	6.52	12.23	3.39	4.25	1.30	3.39	8.35	1.37

Figure 7c: ZSTD Compression Raito by Block Size

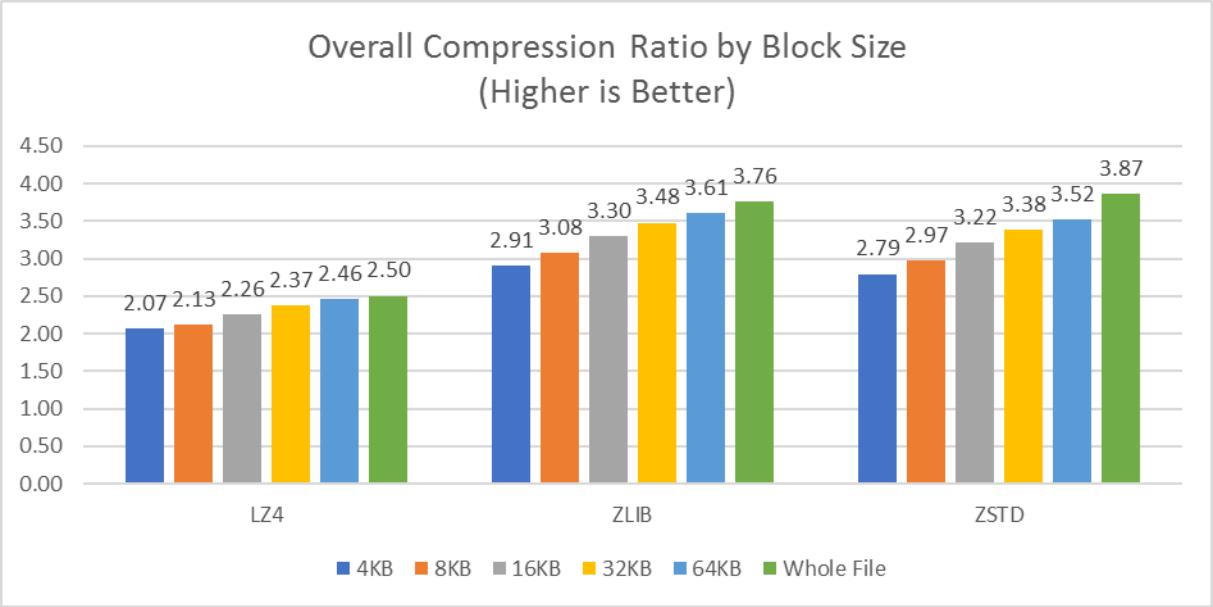


Figure 7d: Overall Compression Ratio by Block Size

Each compressor was able to achieve good overall compression ratios for most block sizes and data sources. One exception is LZ4 which struggled to achieve 2:1 ratios on 10 of the 15 sources. ZStandard was able to achieve comparable ratios with zlib while executing in significantly less time (more on that later). Therefore, **zlib will be removed from the rest of this analysis** since zstd is superior in data rate while not being inferior in compression ratio (example below). Objective #1 Analysis below will study the effect on hit ratio.

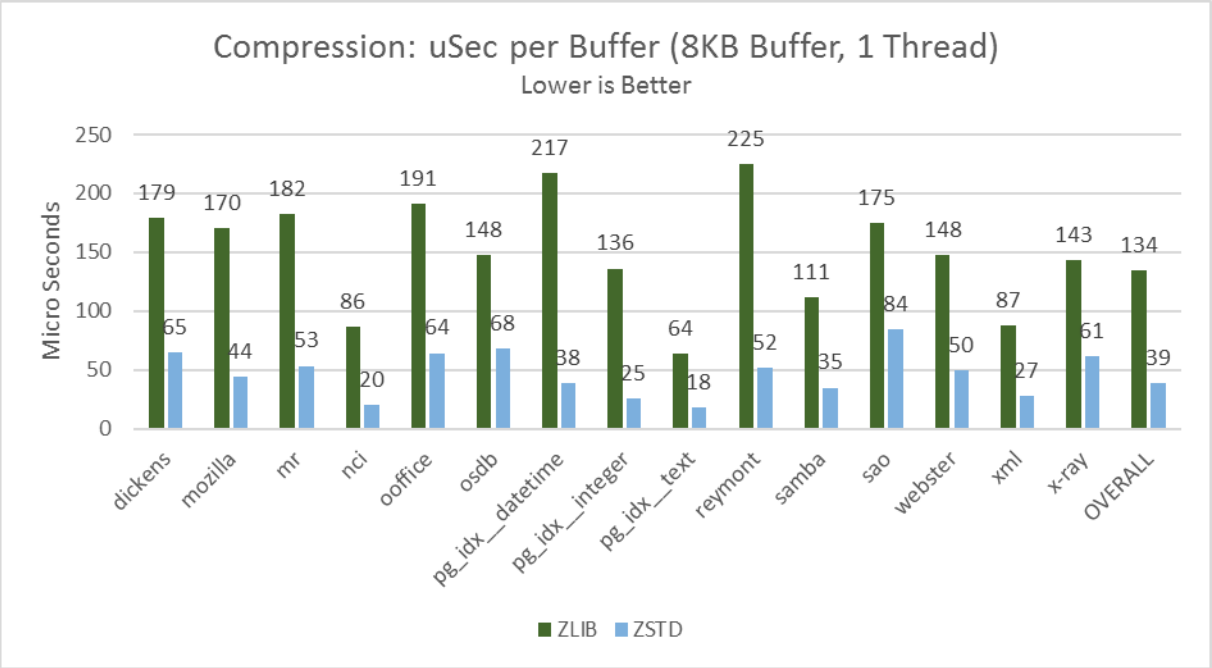


Figure 8a: ZSTD vs ZLIB Compression Data Rate (ZSTD Far Superior)

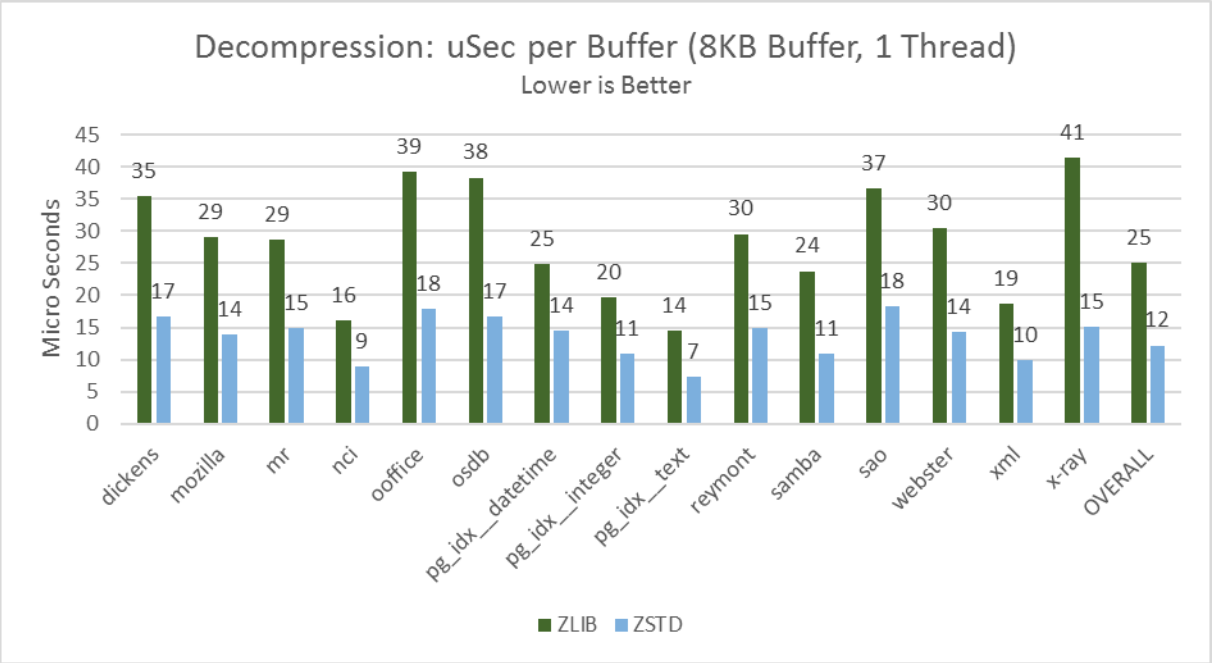


Figure 8b: ZSTD vs ZLIB Decompression Data Rate (ZSTD Far Superior)

With 8KB blocks, the compression **ratios of 2.13 and 2.97** for lz4 and zstd, respectively, will be **used for our objective analysis**.

Data Rate

The following tables and figures demonstrate the data rates associated with compression. Unlike compression ratio, the data rate for compression changes based on several factors: processor type, memory type and speed, parallelization, and more. The test system (found in Appendix A) uses 2.4GHz memory and an Intel i7 7700K processor capable of handling 8 threads concurrently; the following tests examine both single and multi-threaded approaches. Additionally, the data rate for compression and decompression vary, with decompression usually being much faster, and will be examined separately.

Note: Fluctuations in performance due to cache warming, CPU throttling from thermal regulation, and miscellaneous background processes will try to be minimized via pre-warming logic in the application prior to each of several samples.

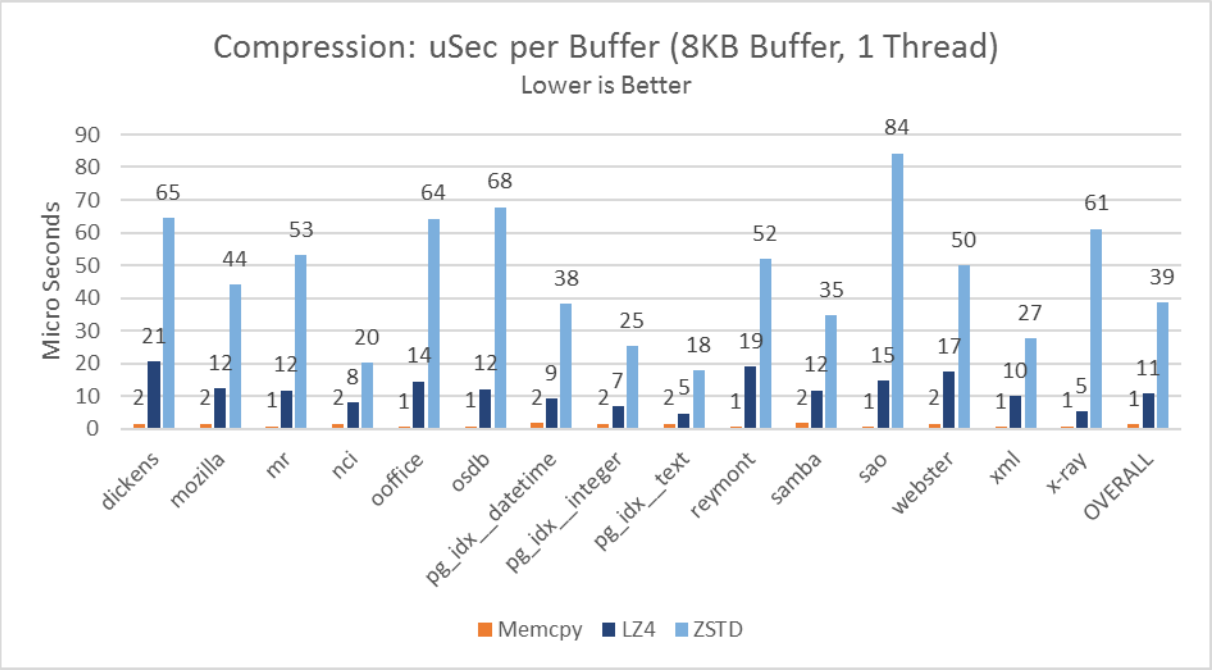


Figure 9a: Compression Data Rate for 8KB Buffers using 1 Thread

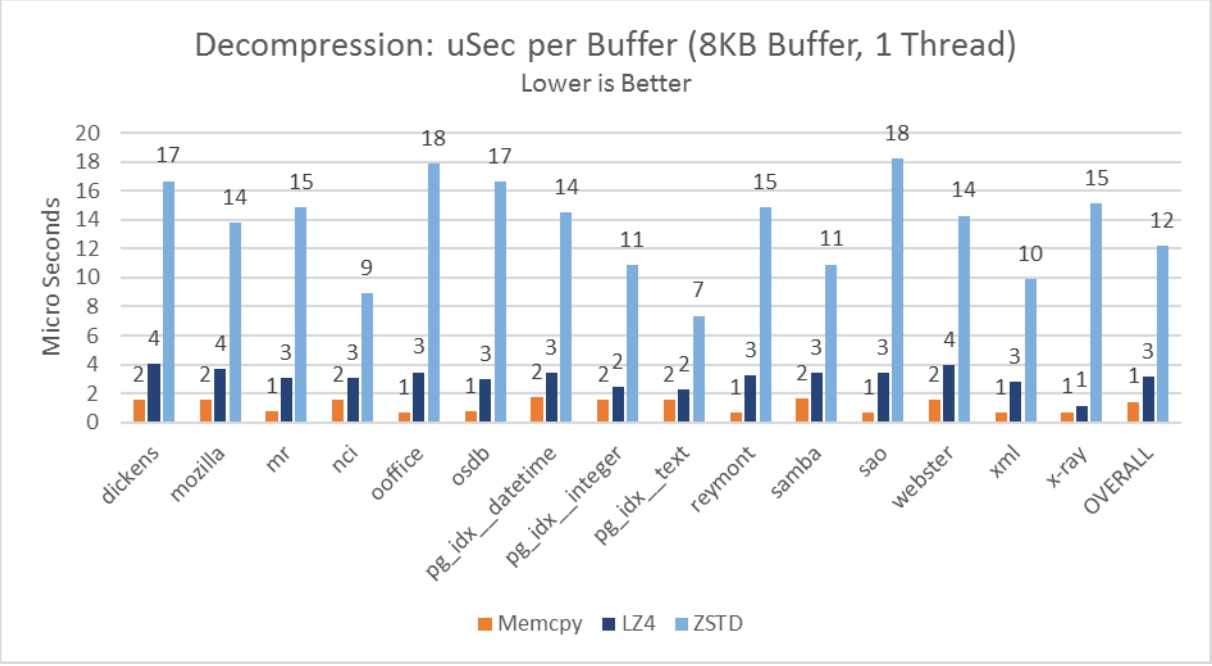


Figure 9b: Decompression Data Rate for 8KB Buffers using 1 Thread

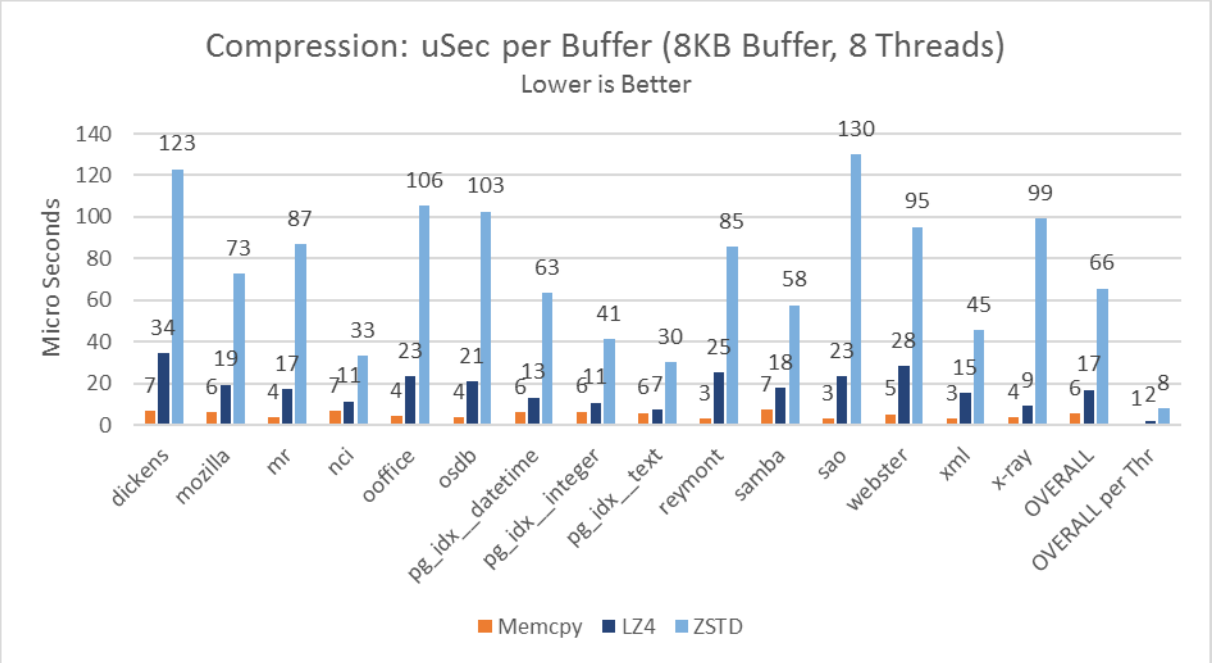


Figure 9c: Compression Data Rate for 8KB Buffers using 8 Threads

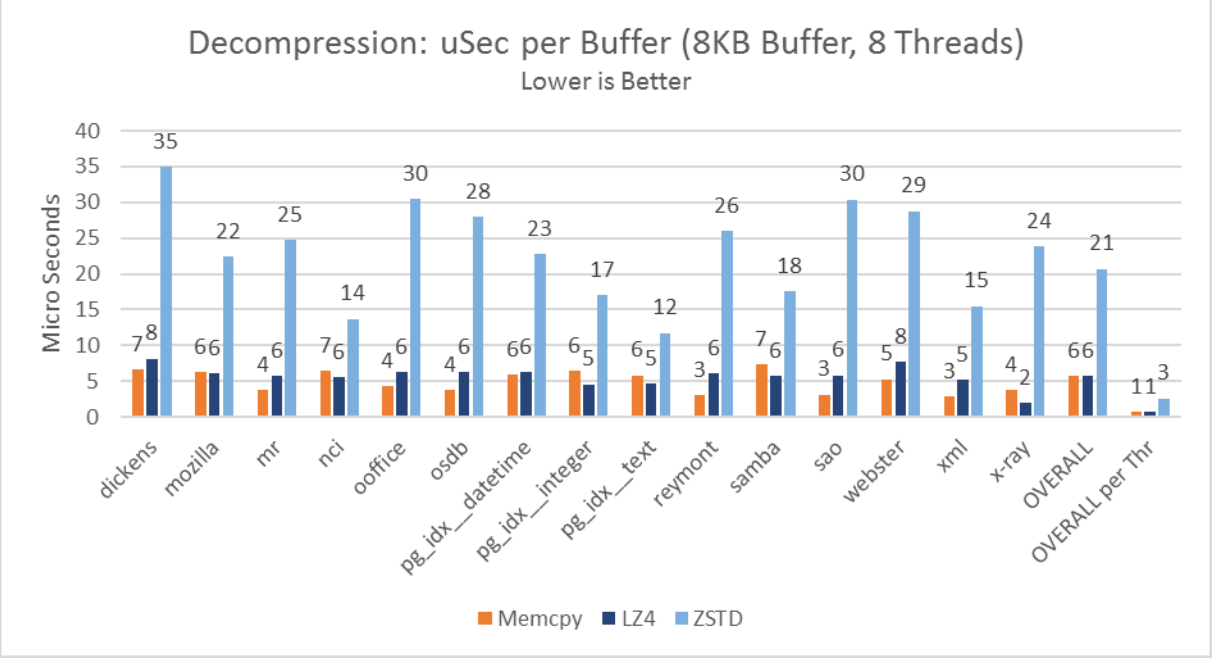


Figure 9d: Decompression Data Rate for 8KB Buffers using 8 Threads

For ZSTD: Compression speeds averaged 39 microseconds of CPU-time per 8KB buffer overall when a single thread was used. Since the mode was single-threaded, 39

microseconds also represents the amount of real-time required. When multithreading was enabled – to leverage each of the 8 processing threads on the processor – the CPU cache and memory contention pushed the average CPU-time per buffer to 66 microseconds. However, the parallel nature reduced the real-time requirement of buffers compressing in bulk down to just over 8 microseconds.

For LZ4: Compression speeds averaged 11 microseconds of CPU-time per 8KB buffer overall when a single thread was used. However, LZ4's minimalist design really shone through: not only was it substantially faster than the other compressors, its small memory footprint kept more data in L1 and L2 caches (theoretically, see: https://github.com/lz4/lz4/issues/489) which reduced cache and memory contention under concurrency. With all 8 processing threads running the CPU-time increased to 17 microseconds; a mere 2 microseconds of real-time when compressing in bulk.

Decompression was 2-3x faster for all compressors, with LZ4 approaching near real-time (i.e.: limited by memory throughput). Zstd required significantly more time in single-threaded mode, comparatively, but its absolute value was still only 12 microseconds. When given the advantage of multi-threading Zstd decompressed at 3 microseconds per buffer.

This research will use the multi-threaded values for compression and decompression for the following reason. Some cache replacement strategies are efficient at removing a single victim buffer (e.g.: LIFO, FIFO, and Random Replacement), while others require scanning or tracking data (e.g.: ARC, Clock, LRU, et cetera). Given the relatively small size of the selected page for many data systems (8KB) relative to the data set size of many large data systems (gigabytes, terabytes, or even petabytes), the remainder of this research will assume that the cost of invoking eviction logic warrants removing multiple victims at a time. For example, when a buffer pool is full it will not invoke the overhead to remove a single buffer to make room for the faulted page; instead it will minimize the overhead by evicting a fixed amount (x buffers or y KB worth), a percentage of the total pool size, or until a time limit is reached. These behaviors are similar to many garbage collectors, including Java's CMS collector ("Concurrent Mark Sweep Collector," n.d.) and the Microsoft CLR (VB, C#, .Net) collector (Petrusha, 2017).

Objective #1 Analysis: Hit Ratio

Using the overall compression ratios for the selected 8KB block size (2.13 for lz4 and 2.97 for zstd), the graph below shows the first objective: compression's effect on hit ratios. The graph includes an even distribution approach and a Pareto (80-20) distribution pattern. Note: this is no longer graphing the function of the data set size like in chapter 2; instead the data set is fixed at 200GB and RAM for caching is fixed at 15GB for demonstration purposes.

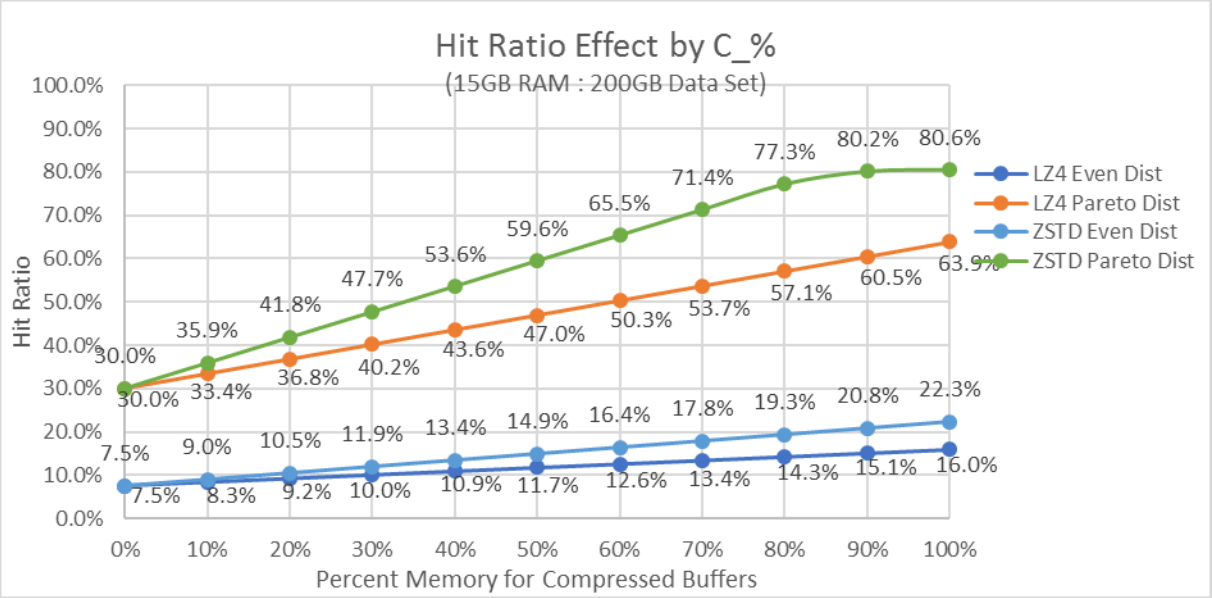


Figure 10a: Hit Ratio by Percent of Memory Given to Comp Space (200 GB set)

With such a simple function both compressors gave the expected results, and their effects compound with the Pareto Principle distribution of buffer access, allowing zstd (and zlib) to achieve an ~80% cache hit ratio by dedicating the majority of memory to compressed buffers. This stands in sharp contrast to the mere 30% hit ratio when compression is not used in this configuration.

Note: a larger data set would change the absolute values on the graph, of course; the selected data set and memory values were for demonstration, but the relative effect of compression on hit ratio would remain constant for any given data set size. Specifically, it will always approach the same ratios as the compressors achieved (2.13 and 2.97):

- LZ4 Even Dist: 16.0% / 7.5% ➔ 2.13
- LZ4 Pareto Dist: 63.9% / 30% ➔ 2.13

- ZSTD Even Dist: 22.3% / 7.5% ➔ 2.97
- ZSTD Pareto Dist: 80.6% / 30% ➔ 2.68*

  * Hit a skewing point due to the 80/20 distribution? Need further research.

Now the same graph, using a 500 GB data set size:

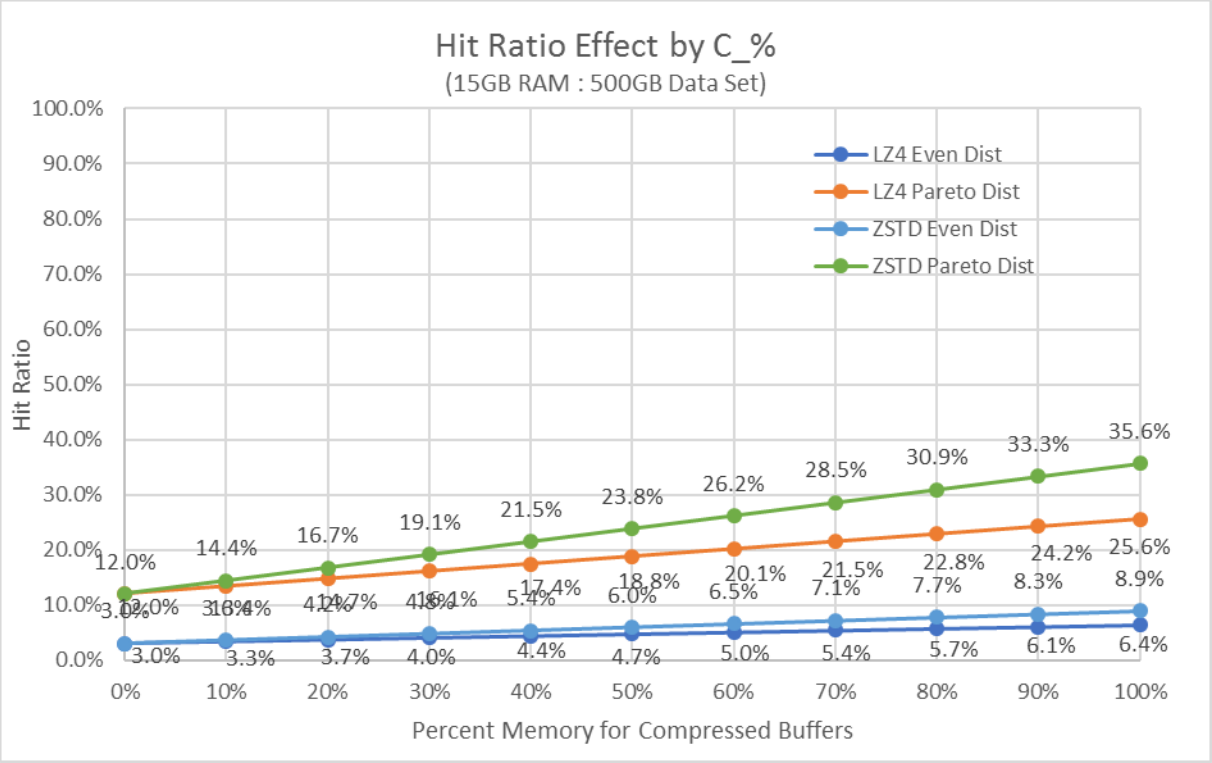


Figure 10b: Hit Ratio by Percent of Memory Given to Comp Space (500GB set)

- LZ4 Even Dist: 6.4% / 3% ➔ 2.13
- LZ4 Pareto Dist: 25.6% / 12% ➔ 2.13
- ZSTD Even Dist: 8.9% / 3% ➔ 2.97
- ZSTD Pareto Dist: 35.6% / 12% ➔ 2.97

Objective #2 Analysis: Pool Effectiveness and Throughput (the Trade-Off)

As mentioned in Chapter 1, increasing hit ratio is merely a reflection of the known-fact that data is compressible. Having discovered the hit ratios from Objective #1 above, this section will incorporate the costs for disk and compression operations to theorize the effect on throughput in various circumstances.

The disk metrics, compression metrics, and Object #1 above produced the following table of values on the prescribed test system. Please refer to Chapter 3 for additional costs being excluded from this research.

Cost Item	Disk Drive (SSD)	LZ4	ZSTD
Disk Read	110 µsec		
Comp Ratio		2.13	2.97
8KB Comp Time		2 µsec	8 µsec
8KB Decomp Time		1 µsec	3 µsec

Table 4: Operation Costs and Key Values

The cost for a page fault with a raw buffer pool would simply be one Disk Read, which is 110 µsec. Calculating the cost of a page fault in a compressed cache strategy is slightly different: one Disk Read at 110 µsec + one buffer compression (on average) which would be 2 µsec or 8 µsec for LZ4 or ZSTD, respectively. The compression time is considered wasted because, on average, when a page fault occurs the raw buffer victim is compressed and moved to the compressed space and an existing compressed buffer must be evicted entirely from the compressed space to make room for the new one. Since the removed buffer was compressed but never used again, that time is considered wasted. Below is a simple diagram demonstrating the waste.

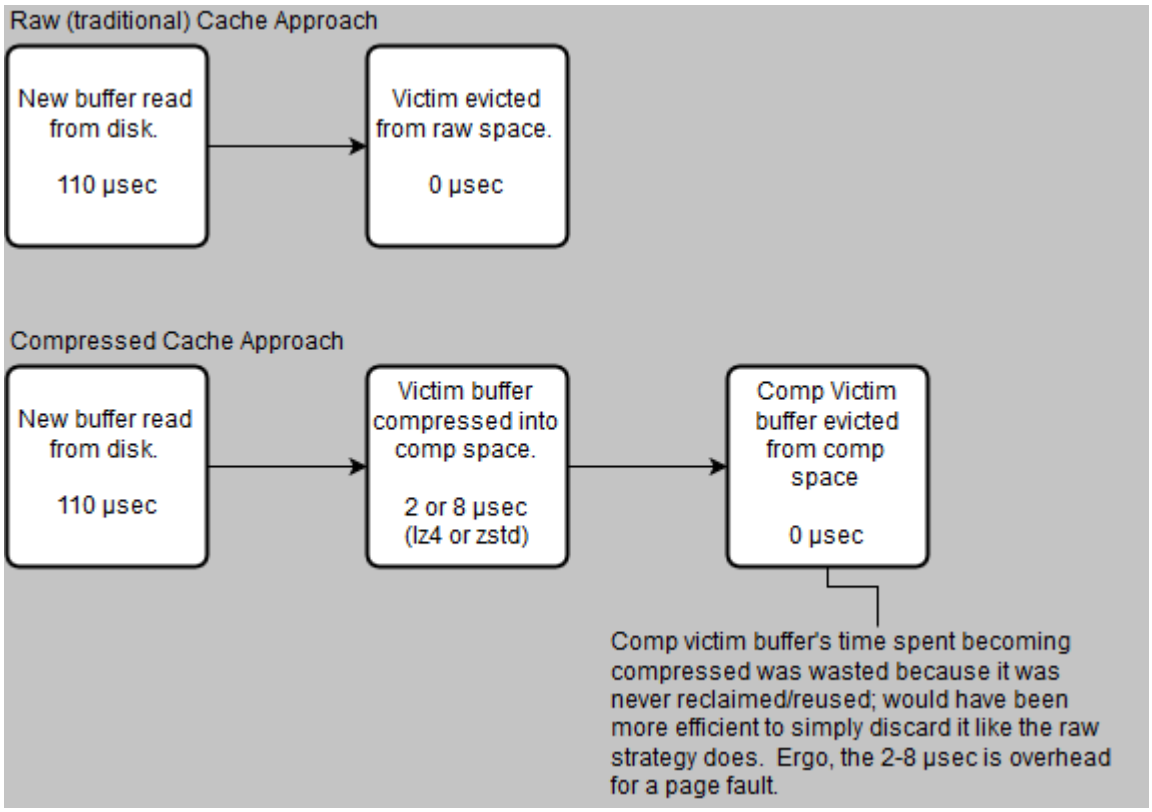


Figure 11: Raw vs Compressed Cache Cost for a Page Fault (w/Full Pool)

The final value needed for comparison is the cost to restore a buffer from the compressed space to the raw space, since applications cannot work with data directly in compressed form. This cost is exclusive to the compressed cache strategy, as reflected in the generalized formula below. Fortunately, the exclusions in Chapter 3 mean the only remaining value for calculating restoration cost is the 8KB compression plus decompression times from Table 4: 3 or 11 µsec for LZ4 or ZSTD. Therefore, the buffer restoration cost for this paper's purposes is simply 3 or 11 µsec.

Now that hit ratios, page fault costs, and restoration costs are known for both strategies, a generalized function can be derived and graphed for throughput of buffers per time unit. Specifically, it will graph the microseconds required per buffer in the average case.

Notes:

- `C_%` is the percentage of the pool allotted to compressed buffers.
- `%ChanceComp` is the percentage chance that a given buffer is compressed: `(PoolSz * C_% * CRatio) / ((PoolSz * (1-C_%)) + (PoolSz * C_% * CRatio))`

- `R_cost` is the cost of restoration for LZ4/Zstd which is compression cost + decompression cost.

The functions are as follows:

```
f(HR%) = (R_cost   * HR% * %ChanceComp) + (miss cost * (1-HR%))   #generalize
f(HR%) = (3µsec    * HR% * %ChanceComp) + (112µsec    * (1-HR%))   #lz4
f(HR%) = (11µsec   * HR% * %ChanceComp) + (118µsec    * (1-HR%))   #zstd
```

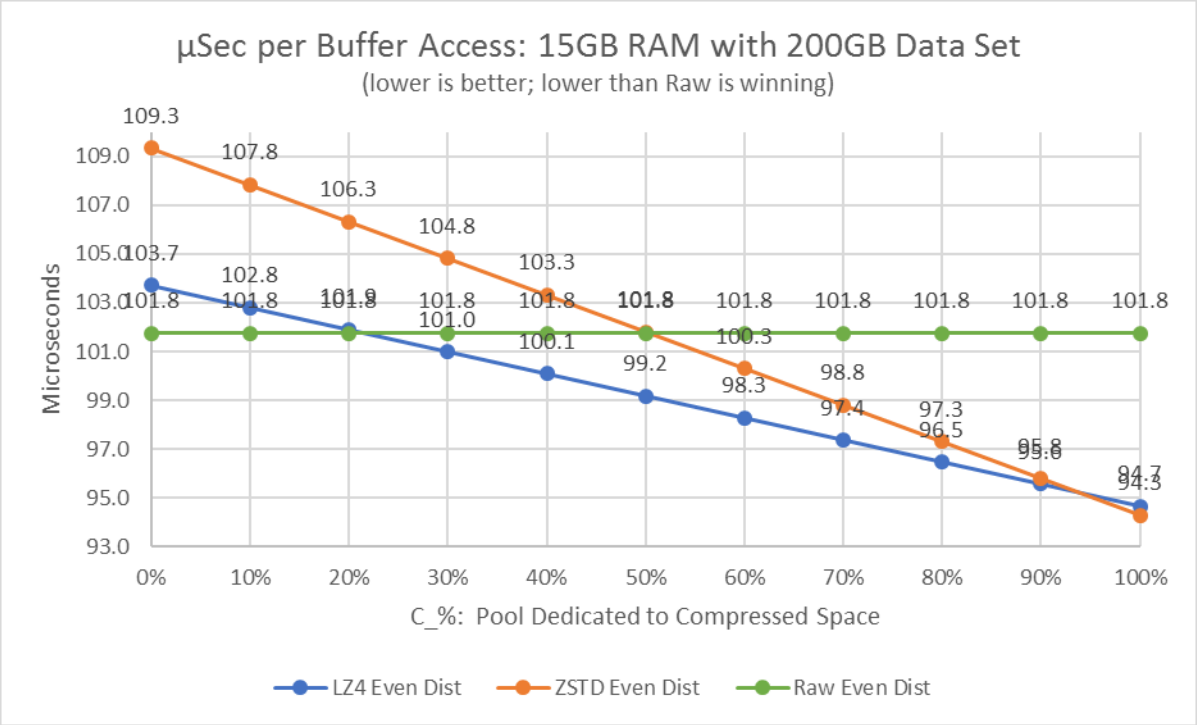


Figure 12a: µSec per Buffer: Even Dist. (15 GB RAM with 200GB Data Set)

Synposis: Low hit ratios (7.5% for Raw) kept buffer access times high at 102 µsec. LZ4 and ZSTD were able to offer higher throughput when the amount of pool space given to compressed buffers reached 20% and 50%, respectively. ZSTD was able to surpass LZ4's performance at 90+% in this scenario.

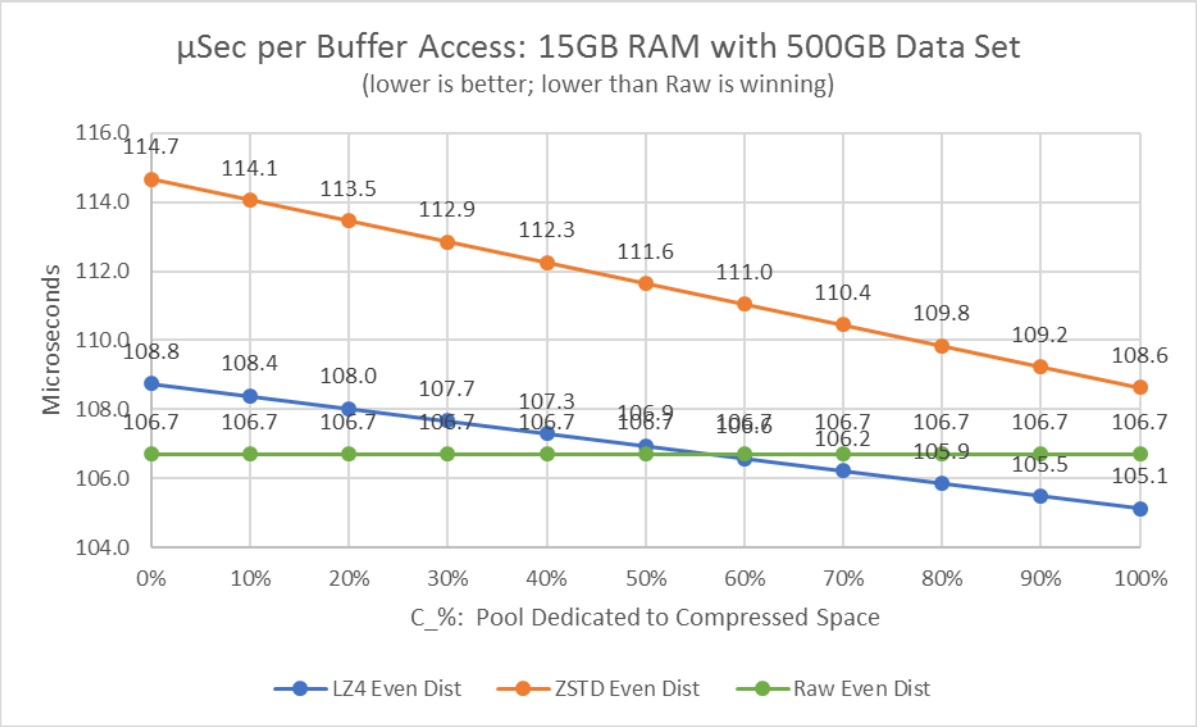


Figure 12b: µSec per Buffer: Even Dist. (15 GB RAM with 500GB Data Set)

Synopsis: The increased data set drove hit ratios down considerably (3% for Raw) which subsequently drove buffer access times up from the last Figure, to 106.7 µsec. LZ4 was eventually able to offer improved performance around 55%; ZSTD was not.

It appears that extremely low hit ratios are unsuccessful at offering higher throughput due to the fact that each cache-miss costs not only a Disk Read, but the 2 to 8 µsec penalty of compressing a buffer that likely never gets used (because of the low hit ratio). The next figures look at the same system and values, but assume a Pareto distribution.

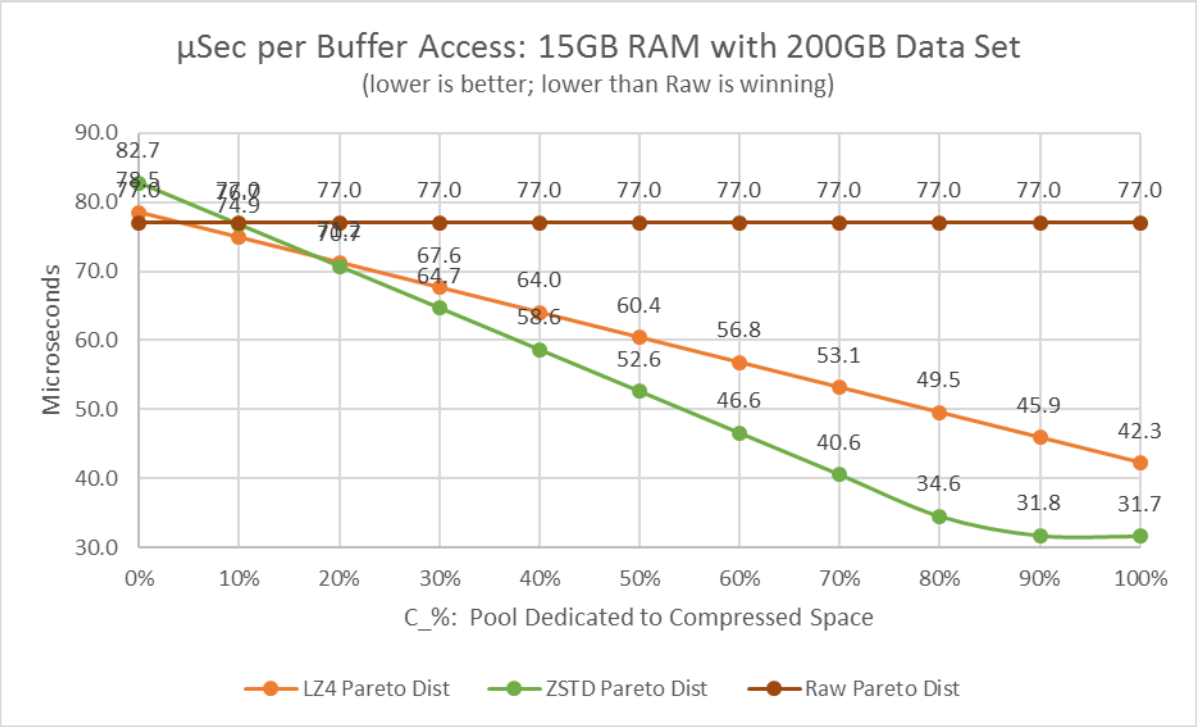


Figure 12c: µSec per Buffer: Pareto Dist. (15 GB RAM with 200GB Data Set)

Synopsis: The Pareto distribution drove hit ratios up (30% for Raw) which drove access times for the raw strategy down to 77 µsec. The higher hit ratios made the compression costs for LZ4 and ZSTD more justified by restoring a larger percentage of them instead of discarding them and paying the penalty for it. ZSTD quickly overtook LZ4 in throughput, but both ended at values significantly faster than a raw strategy.

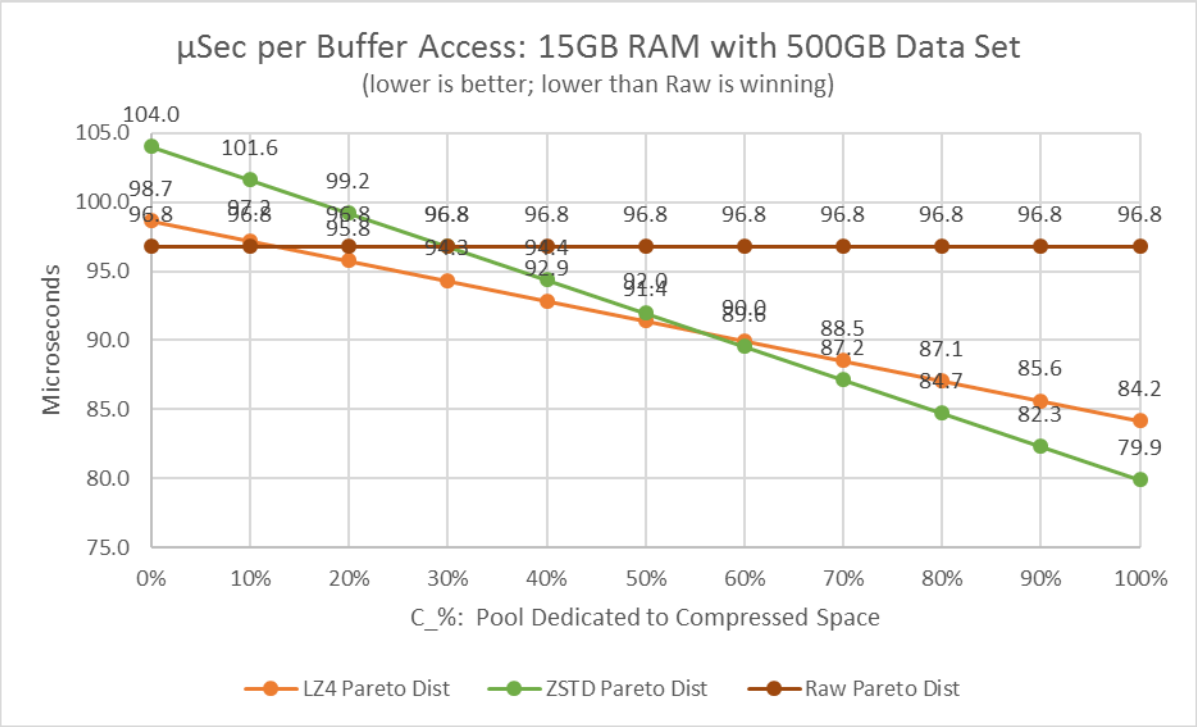


Figure 12d: µSec per Buffer: Pareto Dist. (15 GB RAM with 500GB Data Set)

Synopsis: Raw hit ratios dropped to 12%, as expected, and drove the buffer access times to 96.8 µsec. Both LZ4 and ZSTD were able to offer better performance after 20-30% assignment of C_%. However, ZSTD had a harder time surpassing LZ4 than when natural hit ratios were higher (due to the extra overhead ZSTD pays per cache miss relative to LZ4).

The Silver Bullet (Compression Time Approaching Zero)

Figures 12a-d demonstrate a few configurations where the expense of compression can be offset by the reduction of disk accesses. However, storage device performances differ – as do CPU speeds – and subsequently the cost of compression.

This paper set out to discover if, on modern "commodity" hardware, a compressed cache strategy was even theoretically viable. Some costs were ignored for assumed insignificance, but ultimately the only cost is when a buffer is compressed and then never used again (evicted).

So, if compression performance (time) is the key constraint for viability, what happens when it approaches zero?

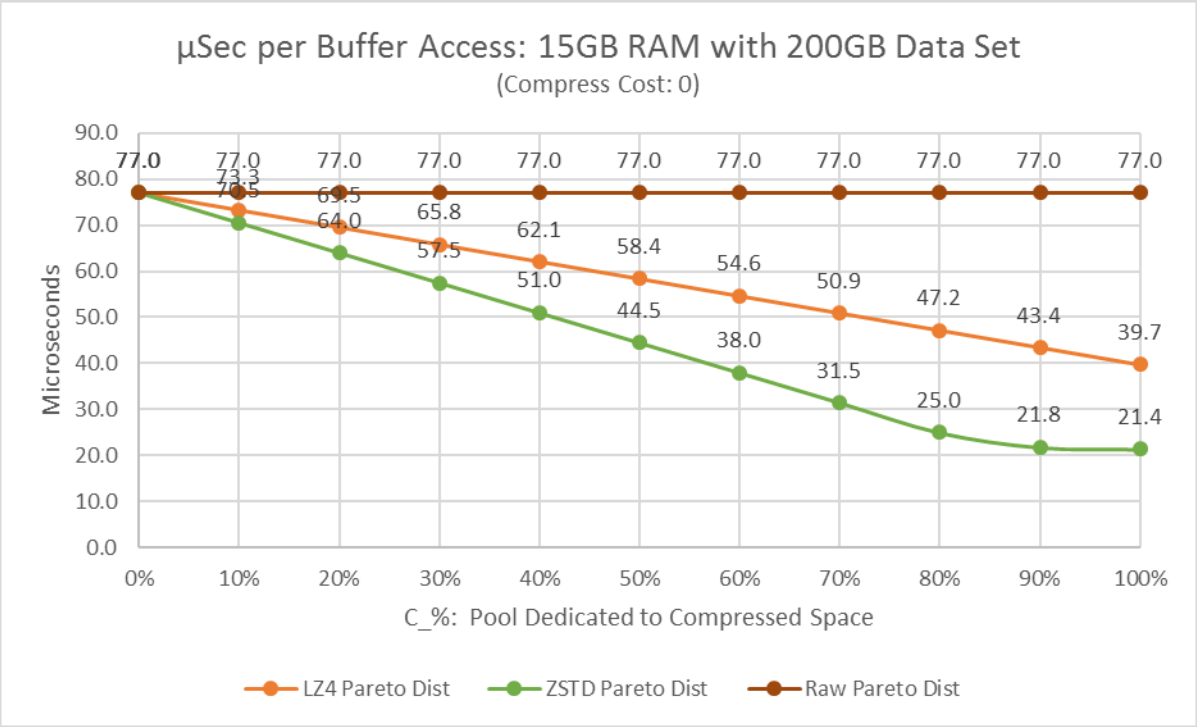


Figure 13a: µSec per Buffer: Zero Cost (15 GB RAM with 500GB Data Set)

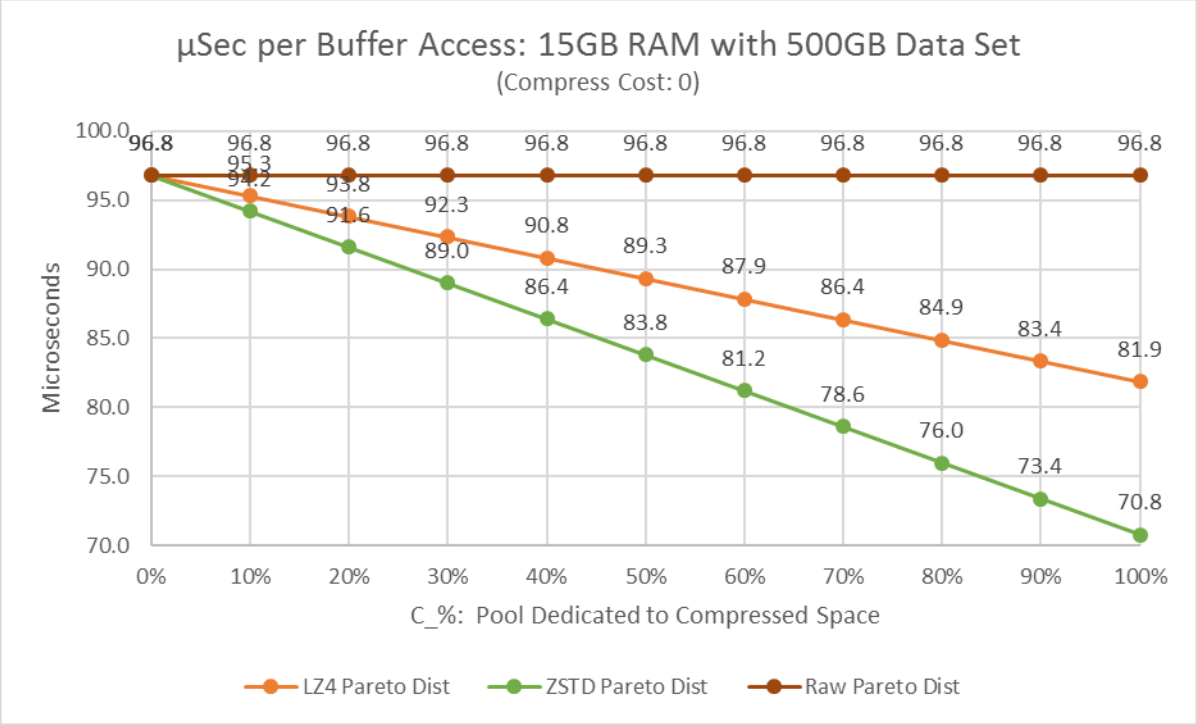


Figure 13b: µSec per Buffer: Zero Cost (15 GB RAM with 500GB Data Set)

When compression cost approaches zero, every cache facing sub-100% hit ratios would benefit immediately from compression.

Nothing is cost free, but advances in field-programmable gate arrays (FPGAs) and application-specific integrated circuitry (ASIC) offer a potential solution to the computational costs of something as simple as LZ4 or ZSTD compression. Note: IBM already integrated a technique similar to this in their Power7+ systems (Cler, 2015), as discussed in Chapter 2.

FPGAs offer more than fifty million gates for customized applications and are power efficient enough to grab the attention of Microsoft to use them in their datacenters ("Field-programmable…," 2018). Lee et al. published a journal article in 2017 that specifically benchmarked LZ4 on an FPGA. While their data corpus is unknown, Lee et al. found that programming an FPGA with only 392,000 gates they were able to create a pipelined LZ4 hardware compressor with 8 cores on 32KB blocks delivering 4Gbit/s total throughput for only 75mW of power (Lee, 2017).

Lee's measurement of 4Gbit/s can be rewritten as 500,000 KB/s or 62,500 blocks/sec of this project's 8KB pages. That would translate to a compression time of 1/62500 sec, simplified to 16 µsec. This value is not far off from our test system which also used 8 threads and required over 90 watts of power compared to the FPGA's 75mW; that is 99.9% less power and a lot less heat. The FPGA used 392K gates at 75 MHz (Lee, 2017) while current (2018) FPGAs have multi-million gate counts and can operate at hundreds of megahertz. Whether these values would translate to linear gains in Lee's LZ4 throughput would require additional research.

ASIC devices boast of even greater performance – approaching 80Gbps – compressing with zlib ("AHA – Data Compression," n.d.). Assuming these speeds could work with 8KB buffers as described herein, the 80Gbps performance would mean sub-microsecond buffer compression time (800 ns). Additionally, the AHA compression device uses zlib which has been shown to be several times slower than LZ4 (additional research needed to see if compression speed difference remain relatively consistent between CPUs and FPGA/ASIC devices). However, zlib (and zstd) offer higher compression ratios… Ergo, if ASIC can already achieve sub-microsecond compression time with zlib or zstd compression then the world is well on its way to reaching: Zero.

CHAPTER 5

CONCLUSIONS

Overall

The project achieved what was set out to be done, and the results and discoveries are promising. Compression has always been an interesting topic; hopefully this research benefits others.

Hypothesis & Viability

As for results, the compression ratios and speeds among the data corpus were great to see. LZ4 and Zstd prove that encoding logic is getting smarter in both metrics. The advent and growth of FPGA and ASIC could mean real-time, high-throughput compression devices for "commodity" hardware in the industry someday (e.g. attached to the Northbridge). Based on this research, and the research/implementations by IBM and others, this is a promising field of study for optimizing large data systems.

It would be wonderful if caching would simply go away because all storage mediums would be fast enough to not need it. Maybe PCM (phase-change memory) or some other next-generation storage technology will get us there. However, looking back at the history of computers, every time the world pushes the envelope it wants it all faster and faster… which means system memory might simply change from DRAM to HMC, HBM, or some other next-generation format, and data caching will remain a dominant strategy.

Next Steps

Two major next steps are observed: proving throughput on integrated FPGA/ASIC and building an actual benchmarking tool for a full cache replacement system.

First, FPGA and ASIC need to leave domain-specific contexts and prove their integration as a pipelined compression co-processor in standard PC hardware (e.g.: x86-

based). This is necessary for two reasons: performance and heat. FPGA and ASIC have the greatest chance at delivering those “near zero” compression times mentioned above, all while relieving the general-purpose CPU so it can do its normal job (e.g.: process queries). The heat issue is simply that: heat. Datacenters are already inundated with heating problems; FPGA and ASIC avoid most of that when compared to the wattage requirements of a general-purpose CPU.

Second, a full cache replacement system needs to be built into a benchmarking application. It’s possible to take something like the PostgreSQL source code and implant the logic… but for benchmarking (especially with different buffer sizes) it’s probably a better idea to simply build a dummy program that will generically cache anything you tell it to.

Note: this was attempted starting in 2015 as a personal learning project. Please note, the code is not great, as it was a learning experiment in several areas. However, you’re free to check out “tyche" on this author's github page: https://github.com/KyleJHarper/

REFERENCES

Pareto Principle. (2018, Mar 11). Retrieved from https://en.wikipedia.org/wiki/Pareto_principle

Cache Replacement Policies. (2018, Mar 4). Retrieved from https://en.wikipedia.org/wiki/Cache_replacement_policies

Pomeranz, H. (2010, Dec 20). *Understanding EXT4*. Retrieved from https://digital-forensics.sans.org/blog/2010/12/20/digital-forensics-understanding-ext4-part-1-extents

Active Memory Expansion. (n.d.). Retrieved from https://www.ibm.com/support/knowledgecenter/en/ssw_aix_71/com.ibm.aix.performance/intro_ame_process.htm

Griffiths, N. (2012). *Active Memory Expansion for AIX 6 & 7*. Retrieved from http://sixe.es/blog/wp-content/8_Active_Memory_Expansion.pdf

Cler, C. (2015, Oct). *Working with Active Memory Expansion*. Retrieved from http://ibmsystemsmag.com/aix/administrator/lpar/ame-intro/

Chait, D. (n.d.). *Using ASTC Texture Compression for Game Assets*. Retrieved from https://developer.nvidia.com/astc-texture-compression-for-game-assets

Solid-State Drive. (2018, Mar 20). Retrieved from https://en.wikipedia.org/wiki/Solid-state_drive

VMax All Flash. (n.d.). Retrieved from https://www.dellemc.com/en-us/storage/vmax-all-flash.htm

List of Interface Bit Rates. (2018, Mar 5). Retrieved from https://en.wikipedia.org/wiki/List_of_interface_bit_rates

Axboe, J. (n.d.). *Flexible I/O Tester*. Retrieved from https://github.com/axboe/fio

Deorowicz, S. (n.d.). *Silesia Compression Corpus*. Retrieved from http://sun.aei.polsl.pl/~sdeor/index.php?page=silesia

Mouse Genome Informatics. (n.d.). Retrieved from http://www.informatics.jax.org/

Quick Benchmark: GZip vs BZip2 vs LZMA vs XZ vs LZ4 vs LZO. (2016, Oct 9). Retrieved from: https://catchchallenger.first-world.info/wiki/Quick_Benchmark:_Gzip_vs_Bzip2_vs_LZMA_vs_XZ_vs_LZ4_vs_LZO#Memory_requirements_on_decompression

Collet, Y. and Turner, C. (2016, Aug 31). *Smaller and faster data compression with Zstandard*. Retrieved from: https://code.facebook.com/posts/1658392934479273/smaller-and-faster-data-compression-with-zstandard/

ZFS. (2018, Mar 27). Retrieved from https://en.wikipedia.org/wiki/ZFS

Ext4 Disk Layout. (2018, Mar 19). Retrieved from https://ext4.wiki.kernel.org/index.php/Ext4_Disk_Layout

Steinbach, C. (2013, Apr 30). *Running PostgreSQL on Compression-enabled ZFS*. Retrieved from https://www.citusdata.com/blog/2013/04/30/zfs-compression/

ZFS Compression – A Win-Win. (2009, Apr 28). Retrieved from https://blogs.oracle.com/solaris/zfs-compression-a-win-win-v2

Chittenden, S. (2017, Mar 4). *PostgreSQL + ZFS Best Practices*. Retrieved from https://www.slideshare.net/SeanChittenden/postgresql-zfs-best-practices

Query Planning. (n.d.). Retrieved from https://www.postgresql.org/docs/9.5/static/runtime-config-query.html

Introducing the Samsung PM1725a NVMe SSD. (2017, Nov 02). Retrieved from http://www.samsung.com/semiconductor/insights/tech-leadership/brochure-samsung-pm1725a-nvme-ssd/

Boner, J. (n.d.). *Latency Numbers Every Programmer Should Know*. Retrieved from https://gist.github.com/jboner/2841832

Concurrent Mark Sweep Collector. (n.d.). Retrieved from https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/cms.html

Petrusha, R. (2017, Mar 30). *Garbage Collection*. Retrieved from https://docs.microsoft.com/en-us/dotnet/standard/garbage-collection/index

Field-programmable gate array. (2018, Mar 19). Retrieved from https://en.wikipedia.org/wiki/Field-programmable_gate_array

Lee, S. (2017, Apr 18). *Design of Hardware Accelerator for LZ4*. Retrieved from https://www.jstage.jst.go.jp/article/elex/advpub/0/advpub_14.20170399/_pdf

AHA Data Compression. (n.d.). Retrieved from http://www.aha.com/data-compression/

APPENDICES

APPENDIX A: BENCHMARK SYSTEM

Summary

All benchmarks were obtained on a dedicated computer system to help reduce deviations. The following system specs were used.

- Motherboard: GIGABYTE GA-B250M-DS3H
- CPU: Intel i7-7700K (BX80677I77700K)
- RAM: 16 GB DDR 2400 MHz (CMK16GX4M2A2400C16R)
- HDD: Western Digital 7200 RPM (WD10EZEX)
- SSD: SanDisk (SDSSDA-120G-G26)
- Operating System: Ubuntu Desktop 16.04 LTS
- File Systems: EXT4 (4KB Block), except RAM-Disk which is tmpfs of course

APPENDIX B: COMPRESSION TESTING SUITE

The code used for the compression testing can be downloaded from the links below. It was built on 64-bit Linux (Ubuntu 16.04 Desktop LTS) using gcc and make with standard C-libs. All packages/libraries come directly from the Ubuntu repository for 16.04.

Main Program

https://github.com/KyleJHarper/masters_project
gcc --version: gcc (Ubuntu 5.4.0-6ubuntu1~16.04.9) 5.4.0 20160609
make --version: GNU Make 4.1 (Built for x86_64-pc-linux-gnu)

Targets (Make)

- build: full build with all gcc optimizations (-O3)
- quick (or fast): quick debug build (-pg and -O0)

Compressors

The masters_project contains copies of the lz4, zlib, and zstd source code rather than using sub-modules or other references. Links to the direct authors' source code is as follows:

- LZ4: https://github.com/lz4/lz4 (tag/version: 1.7.5)
- ZLIB: https://github.com/madler/zlib (tag/version: 1.2.8)
- ZSTD: https://github.com/facebook/zstd (tag/version: 1.1.2)

APPENDIX C: FIO TEST FILES

The FIO program used for benchmarking the HDD, SSD, and RAM-Disk is included below. You can replace the bs=###k with any block-size you wish (4k, 8k, etc.). You must also adjust the ‘directory’ and ‘filename’ settings to make sense on your test system.

```
[global]
bs=128k
ioengine=libaio
iodepth=4
size=700g
direct=1
runtime=300
time_based=1
directory=/mnt/hdd
filename=hdd.test.file

[rand-read]
rw=randread:9
stonewall
```

APPENDIX D: DATA AND SCRIPTS

Most data can be found in the accompanying spreadsheet (a copy will be available in the 'docs' folder of the masters_project github repository from Appendix B). However, for collation the following scripts were useful in executing test programs and collating data.

Running Compression Test Suite

```
#!/bin/bash
for t in 1 4 8 ; do     # The number of threads to use
  for i in 1 2 3 ; do   # The number of times to run the suite
    bin/masters_project ~/Desktop/masters_project/data/ ${t} | tee
~/Desktop/mp_output/${t}t_${i}
  done
done
```

Transform Test Suite Data for Excel

```
#!/bin/bash
sheet='8-9_data'   # Name of sheet
kb=4               # First block size
b=46               # B-column value (file size)
row=87             # First row to work with
tmp_row=0          # Temp for row incrementing
cols=(K L M N)     # Excel column names
last_row=161       # Last row to work with

while [ ${row} -lt ${last_row} ] ; do
  echo -n "='${sheet}'!A${row},='${sheet}'!B${row},"
  for col in "${cols[@]}" ; do
    kb=4
    tmp_row=${row}
    for junk in 1 2 3 4 5 ; do
      echo -n "='${sheet}'!${col}${tmp_row} / (B${b} / ${kb}),"
      let "kb *= 2"
      let "tmp_row++"
    done
    echo -n ","
  done
  let "b++"
  let "row += 5"
  echo
done
echo
```

Build Overall Test Suite Summaries

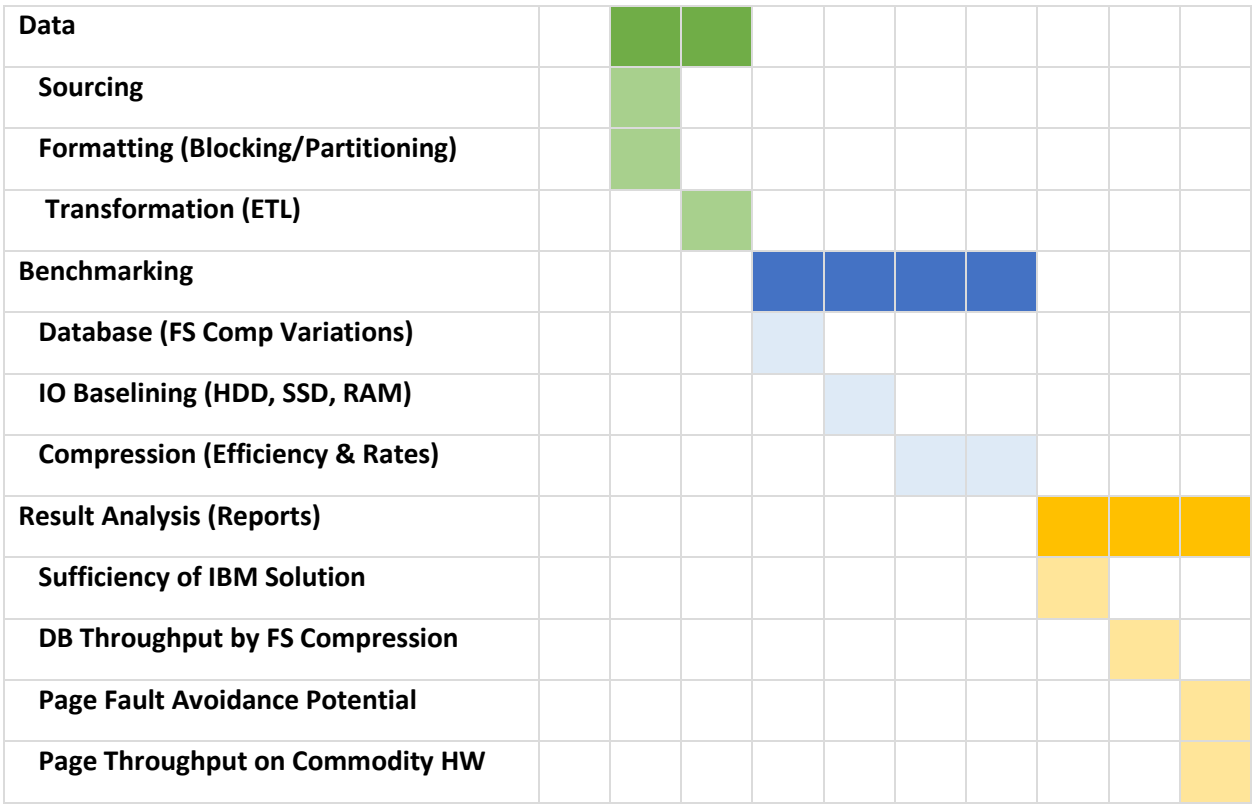
```
#!/bin/bash
row=87             # First row to work with.
row_orig=${row}    # Keep track of original first row for looping.
tmp_row=0          # Temp var for row incrementing.
bs=4               # Block size.
offset=5           # How much to offset per loop.
sheet='8-9_data'   # Sheet name (Excel).

for col in F G H I ; do
  for bs in 4 8 16 32 64 ; do
    tmp_row=${row}
    echo -n "=SUM("
    for junk in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 ; do
      echo -n "'${sheet}'!${col}${tmp_row}"
      [ ${junk} -lt 15 ] && echo -n ","
      let "tmp_row += ${offset}"
    done
    echo -n ") / (B19 / ${bs})"
    [ ${bs} -lt 64 ] && echo -n "|"
    let "row++"
  done
  row=${row_orig}
  echo
done

echo
```

APPENDIX E: WBS AND GANTT – FOR POSTERITY

The following list and table are for academic posterity.

1. Project
    1.1. History & Current Status (by week 1)
        1.1.1. Existing IBM (AME) Technology
        1.1.2. Compressors
        1.1.3. Hardware (Growth in Primary Resource Areas)
    1.2. Data (by week 3)
        1.2.1. Sourcing
        1.2.2. Formatting (Blocking / Partitioning)
        1.2.3. Transformation
    1.3. Benchmarking (by week 7)
        1.3.1. Database (Filesystem Compression Variations)
        1.3.2. IO Baselining (HDD, SSD, and RAM)
        1.3.3. Compression (Efficiency and Rates)
    1.4. Results Analysis (Reports) (by week 10)
        1.4.1. Sufficiency of Existing IBM Solution toward Hypothesis
        1.4.2. Database Throughput Change from Filesystem Compression
        1.4.3. Page Fault Avoidance Potential
        1.4.4. Page Throughput on Commodity Hardware

Week / Task	1	2	3	4	5	6	7	8	9	10
History & Current Status										
Existing IBM (AME) Technology										
Compressors										
Hardware (Resource Area Growth)										


Data
Sourcing
Formatting (Blocking/Partitioning)
Transformation (ETL)
Benchmarking
Database (FS Comp Variations)
IO Baselining (HDD, SSD, RAM)
Compression (Efficiency & Rates)
Result Analysis (Reports)
Sufficiency of IBM Solution
DB Throughput by FS Compression
Page Fault Avoidance Potential
Page Throughput on Commodity HW