ANVIL: An In-Storage Accelerator for Name–Value Data Stores

Ryan Wong
Univ. of Illinois Urbana-Champaign
Urbana, IL, USA
Sandia National Laboratories
Albuquerque, NM, USA
ryanw13@illinois.edu

Nikita Kim
Carnegie Mellon Univ.
Pittsburgh, PA, USA
nikitaki@andrew.cmu.edu

Aniket Das
Univ. of Illinois Urbana-Champaign
Urbana, IL, USA
akdas3@illinois.edu

Kevin Higgs
Univ. of Illinois Urbana-Champaign
Urbana, IL, USA
kmhiggs2@illinois.edu

Engin Ipek
Micron
San Diego, CA, USA
eipek@micron.com

Sapan Agarwal
Sandia National Laboratories
Livermore, CA, USA
sagarwa@sandia.gov

Saugata Ghose
Univ. of Illinois Urbana-Champaign
Urbana, IL, USA
ghose@illinois.edu

Ben Feinberg
Sandia National Laboratories
Albuquerque, NM, USA
bfeinbe@sandia.gov

Abstract

Name–value pairs (NVPs) are a widely-used abstraction to organize data in millions of applications. At a high level, an NVP associates a name (e.g., array index, key, hash) with each value in a collection of data. Specific NVP data store formats can vary widely, ranging from simple arrays/dictionaries and lookup tables to key–value stores and data mining workloads. Despite their importance, existing optimizations for NVPs are limited to only a single data store format, as the broad definition of NVPs allows for significant heterogeneity in encoding and implementation.

We propose ANVIL, the first end-to-end system that allows programmers to broadly accelerate most formats of NVPs. With a conventional solid-state drive (SSD), large-scale NVP lookups can saturate both external and internal SSD bandwidth, as every NVP in the data store needs to be sent back to the host CPU to check for a matching name. ANVIL makes use of in-storage processing to avoid reading out any data for names that do not match, by performing name match checks directly inside the SSD's NAND flash chips. We demonstrate that ANVIL can substantially reduce disk I/O, reduce metadata overheads, and provide speedups of 4.0×, 25×, and 14.6% over a conventional SSD, for three different NVP workloads (database transactions, analytics, and graph processing).

CCS Concepts

• **Computer systems organization** → **Architectures**; • **Hardware** → **Memory and dense storage**; • **Information systems** → *Record storage alternatives.*

Keywords

name–value pairs, in-storage processing, solid-state drives, key–value stores, processing-using-memory, hash indexing

ACM Reference Format:
Ryan Wong, Nikita Kim, Aniket Das, Kevin Higgs, Engin Ipek, Sapan Agarwal, Saugata Ghose, and Ben Feinberg. 2025. ANVIL: An In-Storage Accelerator for Name–Value Data Stores. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture (ISCA '25), June 21–25, 2025, Tokyo, Japan.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3695053.3731000

1 Introduction

The *name–value pair* (NVP) abstraction is one of the most widely used approaches to organize data in modern programs. Broadly defined, an NVP data store is any collection of data that associates a name (e.g., an array index, a key, a hash index) with each value in the collection, allowing programs to use the associated name to find a value. The concept of NVP can be found in millions of applications in both hardware (e.g., content-addressable memories [52, 53], lookup tables [45]) and software (e.g., array/dictionary data structures [115], data mining [3], key–value store databases [33, 44], web addresses [151], and data formats [16, 40, 41]). As the amount of data generated and consumed by modern applications continues to grow exponentially [9, 28, 35, 38, 39, 62, 117, 119], there has been an increased reliance on sophisticated NVP-based data management systems as the backbone of these applications.

While NVPs have many uses, a common thread across the vast majority of NVP-based applications is the need to retrieve NVPs from the data store whose names match a user-requested query. Although this association-based approach is powerful, the ambiguous definition of names introduces a non-trivial overhead for modern hardware systems. Memory subsystems do not inherently map names to data addresses, and the typical method of accessing an NVP requires some form of indirection. While the most basic forms of NVP data stores (e.g., contiguous data arrays) are able to use pointer arithmetic to keep this indirection overhead low, even nominally sophisticated NVPs, such as a Python dictionary, incur significant overheads due to this indirection. Typically, to retrieve the value associated with a specific name, the system must perform a *lookup* operation, where it traverses through a data structure of *all* names until it finds the specified name, and then retrieves the value by either dereferencing a pointer or reading a different field of the data structure, depending on the specific NVP format. Lookup operations scale linearly with the number of names stored in the NVP data store. Therefore, NVP systems attempt to accelerate the lookup process using additional metadata, allowing for faster translations from the input name to the associated value. Given the rapid rate of data growth, the lookup operation can consume large amounts of memory and storage bandwidth, which remain major bottlenecks in modern systems [31, 47, 103, 107, 130].

The increased reliance on NVP data stores, combined with bandwidth bottlenecks, has led to significant innovation over the last several decades on improving NVP lookup performance. On the software side, databases led much of this innovation (e.g., optimized indexes for SQL tables [82, 165]), while recent works have explored optimized data organizations for specific application domains or NVP data store formats [1, 36, 42, 61, 65, 70]. On the hardware side, researchers have developed application-specific or data-structure-specific accelerators for certain types of NVPs [51, 64, 75, 152]. Unfortunately, these solutions cannot be used by any NVP data store, as the broad and ambiguous definition of NVPs allows for a wide range of heterogeneity in encoding and implementation. *Our goal* in this work is to develop an accelerator that can be compatible with most formats of NVP data stores, while alleviating the bandwidth pressure incurred by NVP lookups in modern systems.

To that end, we propose ANVIL (***A**ccelerating **N**ame–**V**alues With **I**n-Storage **L**ookups*). ANVIL provides an end-to-end system for programmers to easily make use of *in-storage processing* for rapid, energy-efficient NVP lookups. As we characterize bandwidth bottlenecks for NVP data stores, we make two observations. First, given the growing rate of data, it is becoming increasingly hard to keep entire NVP data stores resident in main memory. Second, while *solid-state drives* (SSDs) provide enough capacity to hold most of the large NVP data stores that modern applications use, conventional systems incur bandwidth bottlenecks both at the SSD's front-end interface with the host (e.g., CPU), *and* at the SSD's back-end interface with its underlying NAND flash chips. To address both of these issues, ANVIL employs NAND-flash-compatible *processing-using-memory* (PUM), a technique where NAND flash cells can perform bulk bitwise operations without the need for external processors (e.g., [46, 56–59, 110, 132, 135, 140, 141, 157, 158]). Recent works have proposed basic bulk in-storage primitives, which treat the NAND flash chips as either programmable logic arrays [22, 46, 110] or content-addressable memories [140, 141]. While these solutions provide essential building blocks for an in-SSD NVP accelerator, they require significant manual effort to develop an accelerator for a specific NVP data store format.

ANVIL builds its end-to-end system atop these basic primitives. At its core, ANVIL makes use of a dual data representation, where names from an NVP are stored in a transposed format within a NAND flash block *as well as* in a conventional data format. This dual format allows ANVIL to employ bulk in-storage primitives to perform fast parallel searches. These searches retrieve all entries corresponding to the desired name for a lookup (using the transposed names), while still enabling non-lookup operations that require reading or modifying the entire pair (using the conventional data format). ANVIL implements this dual representation automatically, allowing programmers to avoid the burden of explicit data mapping. We provide an NVMe 2.0 compliant interface for user applications to use ANVIL, including commands that enable (1) a mechanism to collect and retrieve search results for NVP lookups, (2) the ability to coherently modify the stored NVP data stores to insert data updates, and (3) concurrent support for I/O operations and for NVP lookups. Importantly, ANVIL requires no modifications to the NAND flash arrays, and requires only lightweight changes to the array peripheral circuitry and the SSD firmware, making it highly compatible with existing SSD architectures.

While ANVIL can provide performance and data movement improvements across many applications, we focus on three examples of NVP data stores to demonstrate how ANVIL can optimize various approaches to large-scale data processing. We evaluate the benefits of ANVIL over a system with a conventional SSD for these use cases. For transactional databases, ANVIL's associative search can mitigate the performance penalties of accessing the disk, with a 4.0× speedup. For database analytics, ANVIL provides a speedup of 25× for a collection of analytical queries. For graph analytics, ANVIL applies associative search on a sparse, irregular data structure, providing speedups for larger-than-memory datasets of 14.6%.

We make the following contributions in this work:

- We introduce ANVIL, the first end-to-end system and NVMe-compatible interface for in-SSD name–value pair acceleration.
- We modify the SSD firmware to perform NVP lookup operations alongside conventional I/O requests, and demonstrate how to effectively manage the (meta)data required to support in-SSD NVP computation.
- We show how ANVIL can reliably be implemented with only lightweight changes to peripheral circuitry and SSD firmware.

2 Background

In this section, we briefly discuss the underlying technologies that ANVIL builds on top of.

2.1 NAND Flash SSD Organization

Modern NAND-flash-memory-based SSDs are typically divided into a front end and a back end. The front end handles (1) interfacing to the host; (2) managing the *flash translation layer* (FTL), the firmware layer responsible for mapping the host's logical block address for a piece of data to an SSD-internal physical address; and (3) dispatching I/O requests to the back-end memory subsystem, which contains the NAND flash memory chips. Due to the complexity of managing these requests and the FTL, the front end typically executes on a dedicated microcontroller and dedicated DRAM.

The back end is divided into multiple *channels*, which can execute independent I/O operations in parallel. Additional parallelism is attained by connecting multiple NAND flash memory chip packages to each channel (known as *way pipelining* or *package interleaving*). Furthermore, within each chip package, different *dies* can support interleaved commands (*die interleaving*). Dies are the smallest superstructure that the front end can issue an independent command to. Each die contains one or more *planes*, which may operate in parallel when specific multi-plane operations are issued to the die. Finally, each plane is composed of multiple *blocks* of NAND flash memory.

A NAND flash block consists of one or more arrays that hold multiple rows of NAND flash cells, where a row connects multiple cells together via its *wordline*, as seen in Figure 1. The wordlines (WL_j) of a block are connected together via shared *bitlines* (BL_k), which connect a vertical column of NAND flash memory cells to a shared peripheral circuit that can perform I/O. Each row contains one or more *pages* of data, and only a single page per block can be accessed at a time. Read and write operations are performed at a page granularity, while erase operations are performed on an entire block at once. To take advantage of internal parallelism, pages from different channels that share the same address offsets can be connected into a larger virtual structure called a *superpage* that spans multiple blocks, with the blocks collectively referred to as a *superblock*.

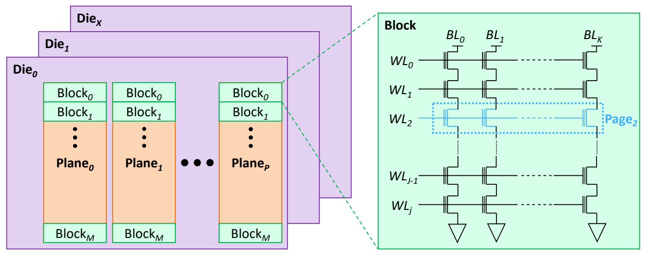


Figure 1: NAND flash chip organization (left), with simplified view of a NAND flash block (right).

A NAND flash cell is a transistor with a *floating gate* that can hold electrons. As more electrons are injected into the floating gate, the threshold voltage (V_{th}) required to turn on the transistor increases, with this threshold voltage corresponding to the cell's *state* (i.e., its stored data value). In a *single-level cell* (SLC), the cell can store two states (0 or 1) that correspond to two discrete *windows* of V_{th} levels. To increase density and reduce the cost per bit, NAND flash manufacturers often store multiple bits in a single cell. For example, a *multi-level cell* (MLC) can store four states (00, 01, 10, 11) to represent two bits, while *triple-level cells* (TLC) and *quad-level cells* (QLC) can store three and four bits, respectively.

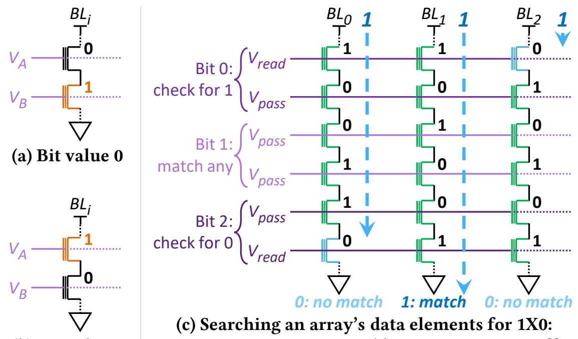
During a read, a voltage is applied to the top of the bitline and ground/string select lines for the NAND flash block containing the data to be read. The state of each cell in a row is read based on whether the transistor turns on and allows current to flow through it. To read stored data from an n-bit cell, we need to apply up to n read reference voltages to the wordline being read, meaning that the read latency increases for each additional bit that is stored in the cell. All other wordlines in the block are driven to a pass-through voltage (V_{pass}), which is sufficiently high enough to turn on the unread transistors regardless of their stored state (i.e., V_{pass} > high V_{th}) and let the bitline voltage pass through. To reduce read latencies, many systems use an SLC cache (i.e., a region of cells that hold only a single bit) to improve performance [161].

To perform a write, V_{pass} is applied to all the wordlines except for the page to be written, which is driven to a programming voltage ($V_{program}$). $V_{program}$ is much higher than V_{read} and V_{pass}, which results in electrons being injected into the floating gate. Once electrons have been injected, it is not possible to remove them until an erase operation is performed. Therefore, prior to storing data, the cell must be erased by applying a large negative voltage (V_{erase}), which frees the electrons and causes V_{th} of the cell to be set low. Multi-bit cells use more complex (and slower) methods to perform programming, such as two-step programming [20, 113] or foggy–fine programming [18, 19].

2.2 Searching Within a NAND Flash Block

Prior work [140] has proposed a NAND-flash-based primitive called in-memory search (IMS). IMS is based on the concept of a *content-addressable memory* (CAM) [13, 76, 144], where an array of data elements can be indexed directly using part of the stored data value (as opposed to conventional address-based indexing in memory and table structures). In general, a CAM executes a lookup (i.e., a *query*) in the form of a parallel search across all data elements (e.g., names) in an array. A *ternary content-addressable memory* (TCAM) extends CAMs to support a wild-card bit (i.e., a don't care bit; represented as X), which will match *any* bit value. Specialized TCAM circuits have been constructed using SRAM-like cells [63, 109] or emerging *non-volatile memories* (NVMs) [7, 34, 43, 98, 99, 104, 118, 156, 162, 164].

Similar to a TCAM, an IMS primitive can perform a parallel search with wild-card bits across all searchable data elements in a single NAND flash block. IMS requires data to be transposed compared to the conventional layout: instead of storing the bits that belong to the same data element on the same wordline (i.e., horizontally), IMS stores the bits of the data element on the same *bitline* (i.e., vertically). In this transposed layout, a single bit of data is represented by *two adjacent NAND flash cells* on the same bitline. These two cells, which we call a *TCAM cell pair*, can represent a bit value 0 (Figure 2a), a bit value 1 (Figure 2b), or a bit value X (representing a bit that will match either a 0 or a 1; i.e., an ignorable bit). Each cell continues to use the same V_{th} states as conventional SLC cells.



(a) Bit value 0

(b) Bit value 1

(c) Searching an array's data elements for 1X0: green transistors are on, blue transistors are off

Figure 2: Associative search in a NAND flash block.

When invoked, IMS applies read reference voltages, corresponding to the search data (i.e., the *key*), to the NAND flash cell gates, as shown in Figure 2c. Each block column represents a different data element. Across Bit 0 (i.e., the first two rows) of each data element, we search for a 1 by applying V_{read} to the first row and V_{pass} to the second row. Bit 1 shows how to match either a 0 or a 1, and Bit 2 shows a search for a 0. At the same time, IMS applies a bit value 1 at the top of each bitline, identical to how reads are performed. If all bits of a data element match the search key, all transistors along the bitline turn on, and the 1 propagates to the output (i.e., the bottom) of the bitline to indicate a match. If *any* bit does not match, one of that bit's transistors will be off (e.g., orange transistors in Figure 2c), and the 1 *stops* propagating, causing the bitline to output 0. IMS returns a *match vector* holding the block's bitline outputs.

3 Motivation: Name–Value Pairs With IMS

Prior works on in-flash PUM processing promise the potential to reduce data movement. Despite their promise, these techniques (e.g., IMS [140], ParaBit [46], Flash-Cosmos [110]) suffer from critical shortcomings when applied to name–value pair systems.

While IMS [140] has the potential to unlock significant in-storage parallelism, we find that fundamental drawbacks in its design leave it unable to outperform CPU-side data scanning for NVP systems. Unfortunately, as we show below, while IMS should deliver significant speedups for NVP lookups, it actually incurs high serialization latencies that often cause it to perform *worse* than a conventional data scan (i.e., sending each name back to the CPU).

Figure 3 shows ideal (i.e., contention-free) execution times for an NVP access in a contemporary SSD (see Section 7 for methodology), using both conventional SSD reads (*Conv*) and IMS, for three different NVP sizes. Our example is simple: the names in the NVP structure are unique, and we are searching for only a single match. We observe from the figure that *IMS is faster than Conv only when the number of NVP tuples exceeds a certain count.* This is because of a fundamental weakness in IMS: it expects *all* of the NVP data to be stored vertically in the SSD. The vertical format allows IMS to search many names in parallel, but forces a matching *n*-bit value to take *n* SSD reads. Even for a value as small as 16 B, this requires 128 SSD reads, which must be serialized due to the NAND flash block structure, resulting in a latency of 2.88 ms. In that same time, an SSD could transfer over 4 million 20 B tuples back to the CPU.

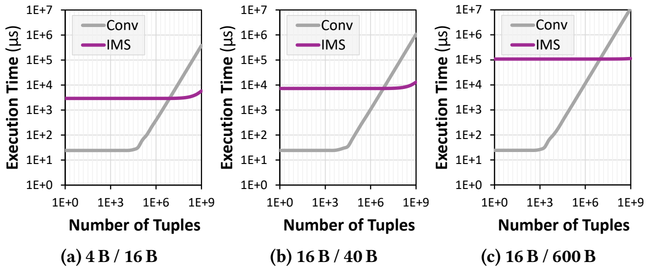


Figure 3: Search time for different name/value sizes.

Figure 4 shows the impact of value size on speedup over Conv. We examine two different systems: (1) the previously proposed IMS [140]; and (2) Dual, a *hypothetical, ideal* in-storage accelerator that we conceptualize to demonstrate the potential benefits of using vertical layouts to accelerate name searches *and* horizontal data layouts to minimize value retrieval latency. In this experiment, we assume a fixed size of 8.4 M tuples in the NVP data structure. We observe that *the serialization bottleneck of IMS worsens significantly as the value width increases.* The IMS work [140] focuses on value-oriented search, where each value is only 48 bits in length, and a similarity search is conducted to count how many value matches were in each group. As a result, they do not need to read out values from the NAND flash memory. However, most real-world uses of NVPs associate short names with long values, and search the shorter names in order to reduce indexing/lookup times. For example, the typical name size in the CUSTOMER table of the popular TPC-C database benchmark is 4–16 B, while the average value size is approximately 600 B. Our hypothetical *Dual* configuration delivers speedups even as the value size continues to increase.

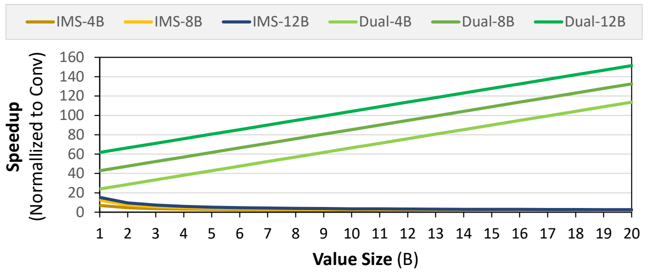


Figure 4: IMS/Dual speedup with different name sizes (indicated in the legend) and value sizes (along the x-axis).

The high serialization cost of value readouts is present in other in-NAND-flash processing architectures. Both ParaBit [46] and Flash-Cosmos [110], state-of-the-art in-NAND-flash approaches for bulk bitwise operations, use the same vertical data format as IMS. While this works well for the computation that these two mechanisms perform inside NAND flash chips, any data readout for host CPU processing suffers from the same bit-by-bit serial readout bottleneck.

Overall, we identify three critical issues to enabling a dual-layout SSD organization: (1) supporting conventional I/O requests alongside NVP lookups, (2) transparently linking vertical and horizontal data regions in a way that optimizes NVP lookups without introducing programmer complexity, and (3) ensuring that updates between vertical and horizontal data regions remain coherent.

4 An SSD for Accelerating Name–Value Pairs

To realize the potential of Dual (Section 3), we propose ANVIL. ANVIL addresses all three critical issues from Section 3. (1) ANVIL logically divides the SSD into two regions: *search regions*, which contain name entries stored in the *vertical bitline* format; and *data regions*, which contain name–value pairs stored in the *conventional horizontal* page format. (2) ANVIL uses an SSD-memory resident *link table* structure to associate names in the search region to their corresponding entries in the data region. The link table ensures that our NVMe-compatible *Lookup* commands can transparently leverage ANVIL with minimal programmer effort. (3) ANVIL keeps search regions, data regions, and the link table coherent with each other whenever an NVP is updated. As we will demonstrate, ANVIL can improve the performance of a variety of NVP data stores, reducing data movement and I/O traffic with only minimal hardware modifications to the array periphery and the introduction of a *SRCH* chip command that performs IMS.

4.1 High-Level Overview

Figure 5 shows the front-end interface for ANVIL. ANVIL aims to eliminate two types of data movement required by conventional reads: CPU–FE (front end) and FE–BE (back end). Applications interact with ANVIL through new NVMe commands that we introduce. An application uses one of these commands to allocate a new search region. Our modified FTL performs block-level allocation for the search region, and the name data is written to the NAND flash chips in a vertical manner (i.e., the bits of a word are written to multiple TCAM cell pairs along the same bitline, with each bitline representing one name entry). ANVIL provides additional commands that can update an NVP, and can append new names to a search region. These regions, and the names that they contain, are efficiently tracked in the firmware.

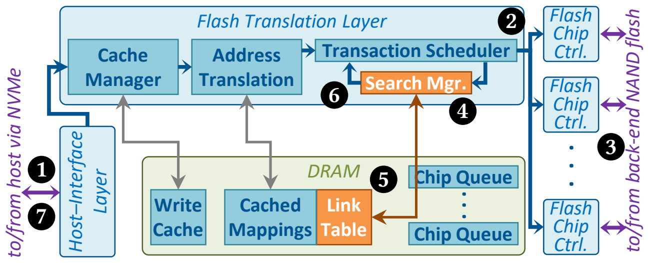


Figure 5: ANVIL front end (new modules shown in orange).

To perform a *ternary* search (i.e., a search that for each bit either looks for a matching value or treats it as a don't care) over one or more search regions, the application issues a *Lookup* NVMe command to the firmware (❶ in Figure 5). The firmware schedules *SRCH* chip commands for the NAND flash chips in the back end (❷). *SRCH* uses per-wordline read reference voltages to represent each bit of the search name, and passes these voltages to the NAND flash blocks that contain the search regions requested by the command (❸). The NAND flash blocks, using modified peripheral circuitry to support per-wordline voltages, then perform IMS (Section 2.2). The output of each bitline indicates whether the word stored along the bitline is a match. Combined with block-level parallelism, a single *SRCH* chip command can search over tens of thousands of name entries (i.e., data elements) simultaneously. The list of matches is returned to an in-firmware search manager (❹), which uses metadata stored in a *link table* to decode where the specific data lies (❺). The firmware then schedules and issues read requests for only the matching data (❻), and the matching data is returned to the host (❼).

4.2 Name–Value Pairs With ANVIL

We demonstrate how NVPs map onto ANVIL using a simple dictionary (see Section 6 for more complex data stores). Figure 6a shows a table associating area codes (name) to an anchor city (value), where one tuple takes up approximately one data page in the SSD. Optimally, our four example tuples are stored in four consecutive data pages. Let us consider an example where an application wants to look up the anchor city corresponding to area code 805.

A system with a conventional SSD can either (1) scan each tuple until a match is found (1a–1d in Figure 6a), incurring data movement overheads for each page; or (2) use an optimized NVP structure, such as a hash table (❷), to quickly identify a subset of potential locations where the matching data *may* reside. Although the optimized NVP structure typically leads to faster access times, hash collisions (i.e., aliasing) may require additional processing. For

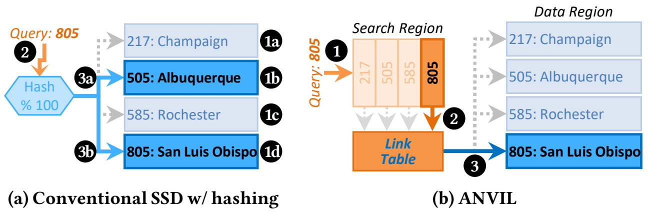


Figure 6: Looking up an NVP in a dictionary.

example, given the hash function $N\%100$, both the page containing Albuquerque (3a) *and* the page containing San Luis Obispo (3b) would be retrieved, requiring the host to resolve the collision using additional instructions and cache accesses.

Figure 6b shows how ANVIL can accelerate this NVP retrieval. First, we store each tuple in the same data format as a conventional SSD, which we denote as the data region. We allocate a *separate* NAND flash block to serve as the search region, which stores the *name* data vertically along the bitlines (i.e., one name per bitline). A new entry is then allocated in the link table, linking the name entries in the search region to the start of the value entries in the data region. When an area code lookup occurs, ANVIL issues *SRCH* commands to the search region (❶ in Figure 6b), which returns a match bitvector (❷), and uses this bitvector to return only the matching page from the data region (❸) to the host.

This example demonstrates three main points about ANVIL. First, *only* the name data is replicated across the two regions, reducing the storage overhead. Second, the full name–value pair is stored in the conventional data format along pages (wordlines). Therefore, conventional I/O operations can continue to interact with the data region as they would in a conventional SSD. Third, ANVIL can mitigate the overheads of NVP structures (e.g., hash collisions) during the lookup phase, without the need for post-processing, reducing CPU–FE data movement.

4.3 Managing Searchable Data in Firmware

As our example from Section 4.2 shows, ANVIL maintains the ability to perform baseline SSD functionality as well as our new *Lookup* functionality by splitting the SSD into two types of regions, *data* and *search*. Data regions behave exactly the same as data currently does in a conventional SSD, relying on conventional read, programming, and erase operations. Search regions can be allocated only through a new command that we introduce (Section 4.4), where the size of the region is based on the number of name entries and the name entry length. For a typical 128–512-row NAND flash block [18], ANVIL can store name entries as large as 63–255 bits, reserving the last bit to represent whether each name entry is valid.[1]

ANVIL does not limit the length of a name entry to the number of TCAM cell pairs along a bitline in a block ($\frac{\text{wordlines per block}}{2}$, which we define as the *native name size*). In cases where we need longer name entries, a search region can be extended across multiple blocks, with each name entry split across the blocks. While this may require additional *SRCH* operations, as well as additional blocks, the resulting match vectors from each block can be ANDed together to form a final match vector prior to decode and accessing the data

[1]We can store and look up shorter names, by setting the search name bits to X for the bits in the entry not used by the name.

region. Alternatively, for large tables, the number of name entries in a search region may exceed the number of columns in a single block (page size) and the match vectors must be concatenated together.

Linking Search Regions to Data Regions. Each search region is connected to a data region, and the mapping between the regions is maintained in a software-controlled *link table.* For each name entry in the search region, the data region contains a corresponding *value entry.* Both name and value entries have a fixed length, allowing the link table to store only a single base physical address per block in the data region (along with a pointer to a firmware buffer for updated values; see Section 4.4). The ANVIL firmware can add an offset to the base address to look up specific name entries corresponding to matching value entries.

The contents of the linked value entry are application dependent. For example, if an application directly wants the value of the matching name, its corresponding value entry contains a non-transposed replica of the name.[2] If we want to implement a key–value store (KVS), which stores tuples of keys and values, each name entry in the search region would correspond to a key, and its corresponding value entry in the data region would hold the value (and, if desired, a non-transposed copy of the key).

Support for Block-Level Allocation. ANVIL requires modifications to the flash translation layer (FTL) to implement *Lookup.* Most importantly, rather than using page-level allocation, search regions (but not data regions) use block-level allocation since each bit of a search name within a search block must be allocated contiguously. Despite performing block-level allocation, ANVIL uses existing page-level programming operations to write data to the block, handling transposition transparently to the programmer.

In a conventional SSD, superblocks are formed from a collection of blocks usually from different chips with the same block offset. Accordingly, prior work has proposed superblock FTL designs [67]. ANVIL is amenable to this type of allocation scheme, as it enables the system to search over an entire superblock in parallel. Although block-level allocation may result in more internal fragmentation inside the search regions, conventional SSD garbage collection operates at a block level, and we thus expect this fragmentation to have minimal impact. We quantify internal fragmentation in Section 8.4.

4.4 ANVIL Command Interface

To use ANVIL in programs, we propose an NVMe 2.0 compliant command set specification [105]. The proposed commands are similar to the KVS command set in NVMe 2.0 [106], and take advantage of the ability to add vendor-specific functionality to the interface.

Allocate / Deallocate / Append. Since search regions are managed separately from data regions, they must be explicitly allocated and managed. The *Allocate* command creates a search region based on the name entry size, and links it to the specified data region by storing the data region's starting address and value entry size, as discussed in Section 4.3. The command can optionally provide data for the search region by providing a pointer to the host memory that stores value entries. The *Deallocate* command frees a search region by marking all blocks for erase.

The *Append* command is used to add names of the same size to an existing search region, along with their corresponding value entries to the data region. The firmware stores the new name entry (along with its corresponding value entry) in a software buffer. Once there are enough name–value pairs in the software buffer to fill an entire block, one new block each is appended to the search region and the data region, and the name and its value entry are transferred out of the buffer and written to the drive. The firmware appends the link table with the new mapping.

Importantly, *Allocate*/*Append* require that the order of name entries is the same as the order of value entries in the linked data region. This must be managed by the host application.

Simple Lookup / Lookup / Lookup Continue. Once name entries and corresponding value entries are written, the host can issue *Lookup* commands to the SSD. For simplicity, we describe the *Simple Lookup* command, which contains a fixed-length search name (up to 127 bits), the address of the search region, and a pointer to a host buffer where return data can be stored. Alternatively, if the name size exceeds 127 bits, the host may issue a *Lookup* command, which uses a data pointer to communicate the search name to the SSD. Both the *Lookup* and *Simple Lookup* commands can also communicate additional operations to complete with the resultant search data, such as logical AND and OR reductions between shorter names. The firmware issues one or more chip-level *SRCH* commands to the selected chips to perform bulk parallel search using IMS, where each command invocation returns a match vector from the selected blocks. The firmware uses each match vector, along with a base address from the link table, to calculate the addresses for value entries in the data region that correspond to matching name entries. The firmware then issues read commands to these addresses, and writes the returned data to the host buffer.

Note that the *Lookup* NVMe command does not know how many tuples will match, and therefore may not allocate sufficient host buffer space to store all of the returned data entries. To address this, a flag is added to the completion queue entry, notifying the host that the buffer was inadequately sized, and that more matches remain. Upon receiving this signal, the host may issue a *Lookup Continue* to the same search region address with a new host buffer to complete the data transfer issued by a prior *Lookup.*

Delete. When the *Delete* command is invoked to remove a name entry and its corresponding value entry, the firmware first searches for the name entry in the search region. The firmware then invalidates all matching name entries, by using normal chip-level page commands to read and update the entries' valid bits for each block containing a match. This invalidation is written *in place*, since it involves only raising V_{th} of one cell per match to change a 1 to a 0.

Updating an existing name entry (or its associated value entry) involves first calling *Delete* to remove the old entries, and then calling *Append.* While such updates are costly, we find that they are infrequent (or non-existent) for many target applications, which tend to use relatively stable datasets.

To mitigate the cost of writes (and updates), we store data in a firmware-managed buffer, which stores data in the SSD's on-board

[2]By replicating the name in a conventional wordline-oriented data region, we can read an n-bit name's value out in a single cycle, instead of performing n reads (one for each bit) to extract the name from the entry in the transposed search region.

DRAM until a full NAND flash block can be written. As DRAM-based write buffers are volatile (i.e., they lose data in the event of a power loss), we must ensure the durability of these updates with a backup. ANVIL can leverage conventional logging techniques (e.g., write-ahead logging [50, 100, 101, 120]) that are often used to guarantee persistence for write buffer data in the event of a failure, by maintaining a short log of pending writes in the SSD.

5 ANVIL Optimizations

Building upon our basic version of ANVIL, we propose a series of optimizations to (1) improve the reliability of the *Lookup* command and (2) reduce several data movement overheads.

5.1 Enhancing ANVIL's Reliability

While ANVIL's in-storage lookups optimize FE–BE data movement by not sending most data to the SSD firmware, this also prevents ANVIL from making use of the in-firmware error-correcting techniques that ensure correct reads [18, 60, 97, 127]. Due to aggressive scaling, raw reads from NAND flash chips are subject to three main sources of bit errors [18]: read disturb, write disturb, and retention errors. For ANVIL, these uncorrected bit errors can result in either false positives or false negatives during lookups. Therefore, we propose two mechanisms that can help mitigate lookup errors.

A false positive occurs when the *SRCH* chip command incorrectly returns a non-matching NVP, because cells along the NVP's bitline that should have stayed off instead turned *on*. False positives can occur due to retention errors, which reduce the V_{th} of a cell over time, causing the cell to incorrectly turn on. False positives do not create a correctness issue for ANVIL, as post-processing (e.g., having the host re-check the names returned by the *Lookup* command) can identify and eliminate them. In contrast, false negative occurs when *SRCH* incorrectly fails to return a matching NVP, because cells along the NVP's bitline that should have been on incorrectly stayed *off*. Read and write disturb can increase the V_{th} of a cell over time, causing the cell to fail to turn on. A false negative *cannot* be corrected using post-processing, as no data retrieved from the back-end NAND flash would indicate the error.

One benefit of ANVIL is that the *SRCH* chip command reduces the likelihood of read disturb compared to conventional NAND flash reads. Prior work has shown that the probability of read disturb errors decreases exponentially as the voltage applied to a NAND flash cell is reduced [19]. While a NAND flash read applies V_{pass} to all but one cell on the bitline, the *SRCH* chip command typically applies V_{pass} to only half of the cells, instead using the much lower V_{read} for the remaining cells. In addition to this increased resilience, we employ two techniques that successfully mitigate remaining false negatives by counteracting V_{th} increases.

Enhanced SLC-Mode Programming. For the search region only, we employ the enhanced SLC-mode programming (ESP) technique from Flash-Cosmos [110]. Modern NAND flash memory stores multiple bits per cell, causing the tail distributions of V_{th} to overlap between adjacent cell states after error-induced V_{th} shifts. First, ESP treats NAND flash cells as SLC (Figure 7a), thereby reducing the likelihood of overlap. Second, ESP increases the V_{th} margin between bit value 0 (high V_{th}) and bit value 1 (low V_{th}), providing more headroom for both retention voltage drops and read/write

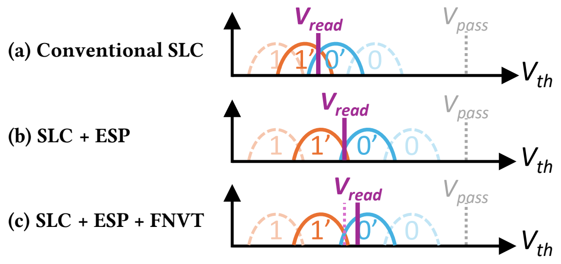


Figure 7: SLC with ESP and FNVT. Primes indicate the state distributions after they shift due to errors.

disturbs (Figure 7b). In addition, prior work has observed that the error rate reduces by two orders of magnitude [60] when going from MLC to SLC, a behavior that is complementary to ESP.

False Negative Voltage Tuning. Given that ANVIL is able to correct false positives but not false negatives in post-processing, we bias the *SRCH* chip command to reduce false negatives at the cost of introducing more false positives. We do this using a simple technique that we call *false negative voltage tuning* (FNVT). With FNVT, we make use of simple yet accurate firmware models of NAND flash errors (e.g., [91]) to predict shifts in V_{th}, and increase V_{read} based on the model prediction to counteract the V_{th} increases that would otherwise generate false negatives (Figure 7c).

We demonstrate why FNVT can eliminate correctness issues using Figure 8, which shows a collection of allergens (name), foods (value), and a query requesting foods that contain peanuts, a common allergen. Figure 8b shows the results using ANVIL with only SLC devices (i.e., no ESP and no FNVT). When ANVIL executes the query, it returns two entries: PB&J (A_1), a correctly identified food; and brownies (A_3), a false positive incorrectly identified as having a peanut allergen. While the CPU can iterate over the returned entries and throw out brownies, it failed to receive two foods with peanut allergens from ANVIL: blondies (A_0) and brittle (A_4). These two entries are considered false negatives, and because ANVIL failed to return them to the CPU, the query is now incorrect.

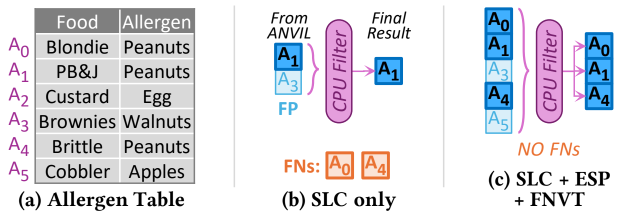


Figure 8: Query for food allergens with ANVIL.

Figure 8c shows the same query when ESP and FNVT are applied. By biasing towards false positives, ANVIL returns *all* foods containing peanuts (A_0, A_1, A_4), as well as additional foods not containing peanuts (A_3, A_5). Although foods that were true negatives in SLC-only mode have now become false positives (e.g., cobbler; A_5), these false positives can still be filtered out via post-processing, meaning that the query still returns the correct data. Using SLC, ESP, and FNVT, ANVIL can ensure a reliable search.

Reliability Analysis. To evaluate ANVIL's reliability, we use anonymized NAND flash data from prior work [91] and develop a

Gaussian sampling-based model. The data was collected from *real* state-of-the-art SSDs, and includes detailed, high-precision voltage distribution information that allows us to discern how SLC-mode devices would behave in commercial SSDs. The data has been widely used by prior works on NAND flash reliability [90, 110, 111].

We examine reliability as ANVIL searches across 33 M entries (each with a 12 B name and 128 B value), for an SSD at 2.5 k, 5 k, 10 k, and 20 k P/E cycles. We characterize this using a typical 192 x 4 kB array, with four modes: (1) S: traditional SLC, (2) S+E: SLC with ESP, (3) S+F: SLC with FNVT, and (4) S+E+F: SLC with ESP + FNVT. Figure 9 shows the error counts for the four modes. At 2.5 k P/E cycles, traditional SLC has a false negative rate of 3.5×10^{-4} (11411 out of 33 M), and a false positive rate of 2.2×10^{-5} (725 out of 33 M). At low P/E counts, when we apply S+E, the error rate is reduced to zero; However, at 20 k P/E cycles, we observe that S+E can no longer eliminate all errors, with a false negative rate of 7.0×10^{-7} (23 entries out of 33 M) and a false positive rate of 6.1×10^{-8} (2 out of 33 M). In contrast, S+F can completely eliminate false negatives at all P/E cycle counts at the cost of substantially increasing the rate of false positives (1.5×10^{-2} for 2.5k P/E cycles). When we combine FNVT on top of ESP (S+E+F), no false negatives manifest once again, *and* the false positive rate decreases to only 1.22×10^{-6} (40 out of 33 M). S+E+F results in a performance overhead of only 0.2% (firmware and data movement) to verify the values, compared to error-free search. We conclude that while ESP may be sufficient for devices early on, FNVT can effectively work with ESP to eliminate false negatives throughout the device's lifetime with minimal overheads due to false positives.

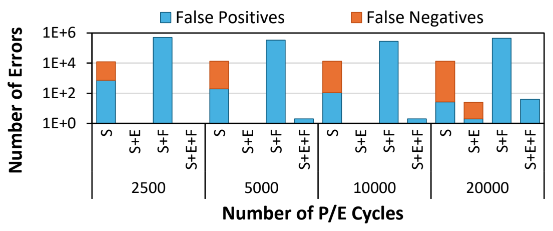


Figure 9: Manifested errors during *Lookup* over 33 M entries.

5.2 Reducing ANVIL's Data Movement

Supporting Early (Conditional) Termination. Once a search has been executed on the NAND flash blocks, the data must be read back by the microcontroller. However, this data is a match vector, which must be decoded prior to executing the secondary read. Because we are searching for a particular piece of data among a large dataset, we expect the majority of the match vectors to return no matches. Therefore, storing a match vector of all 0s would waste the limited in-SSD DRAM capacity. Accordingly, we propose a mechanism to support early (conditional) termination. At each flash channel controller, we add a small circuit to quickly decode the match vector as it is burst (i.e., read) from the back end. If a data burst is all zeroes, we increment a counter and discard the data burst. If there is a match, then we tag the burst with the value stored in the counter, which is used to later decode the corresponding value.

Write Inversion. Modern NAND flash supports inverse reads, where the data read from the NAND flash device is the inverse of the value stored in the cells [110]. While initially designed for reads, this functionality can be used as part of the program–verify operation to accelerate writes [81]. We can use write inversion to reduce the amount of command data transmitted to the NAND flash chips. As an option for optimization, we can choose to restrict our primitive from Section 2.2 to store only bit values 0 and 1 (i.e., we no longer store X values in the SSD, but can still use X as a don't care in a search value), the two wordlines coupled to form TCAM cell pairs in ANVIL are the inverse of each other. Once the program operation for a wordline has been completed using a program–verify operation, a subsequent row address can be supplied without additional write data. If the program operation is then executed, the inverse data will then be written to the new wordline. Write inversion can reduce FE–BE data movement between the firmware processor and the NAND flash arrays by approximately 2× during programming.

Data Result Compaction. A *Lookup* NVMe command may return multiple matches. However, the corresponding matches may not be contiguous in the data region. Therefore, a search operation with *N* matches would return *N* logical blocks (i.e., pages) of data to the host, resulting in wasteful CPU–FE data movement. Thus, if the maximum data entry size is less than the size of a host logical block, we compact multiple data entries into a few host blocks, which are returned to the host. To enable this optimization, it is necessary to know the size of the corresponding data entries in the data region. This information is provided to ANVIL as part of the *Allocate* command, and is stored in the link table structure.

Duplicate Page Reads. Names in an NVP data structure may not be unique, causing a lookup for a given name to return multiple matches. In many cases (e.g., graphs analytics), these entries may be spatially stored next to one another (e.g., edges with the same source). This effect can cause ANVIL to retrieve the same data page multiple times, resulting in substantial FE–BE movement. To mitigate this effect, we include 64 B of metadata per *SRCH*, which stores the most recently fetched data page; thus, when a match vector is decoded, a given data page will only be retrieved once. This optimization can both reduce the number of duplicate page reads as well as mitigate overheads of false positives. We quantify this effect in Section 8.4.

6 Accelerating Name–Value Applications

ANVIL can be used across a wide range of NVP-based applications. In this section, we demonstrate how ANVIL can accelerate complex NVP structures in two large-scale applications.

6.1 Relational Databases

A general relational database can be formalized as an NVP by creating tuples out of structured data that associates a queried attribute (name) with the record (value). Figure 10 shows an example of two tables (adapted from Aho et al. [6]): one with records of faculty members, and another with records of students. A potential query may request the **faculty in Group 3**. For this query, the Group column contains the names, and the record is the associated value.

We can map the database to ANVIL's dual storage representation as follows. First, complete copies of each database table are stored in the data region. Second, for any table column that could be used

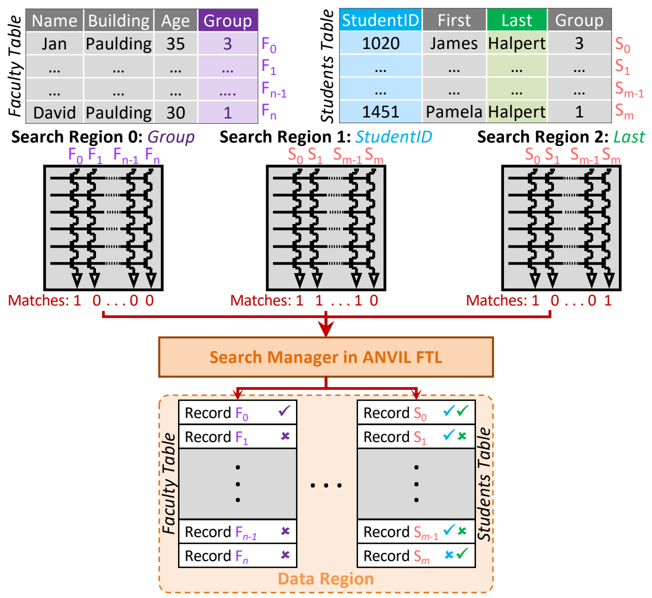


Figure 10: Search/data region mapping of database tables.

as an index, we store a copy of the column in a search region, and link that search region to the table's data region using ANVIL's link table. Transactional queries (OLTP; online transactional processing) can leverage ANVIL's accelerated *Lookup* commands to retrieve matching records, while analytical queries (OLAP; online analytical processing) can use either the search regions or the data regions to examine many records at once.

Optimizing for Multiple Indexes. Some queries may use multiple attributes (i.e., more than one index). ANVIL can support this behavior in one of two ways. First, a user can allocate *separate* search regions for each column that can be used as an index. Each search region requires a separate entry in the link table, but all regions link to the same data region that contains the complete database table.

Linking multiple names to the same data element requires an additional link table entry for each search region (name). A *Lookup* command encodes which search region it targets, and thus uses the link table entry corresponding to the desired region. When the *Delete* command is invoked, *all* search regions that use attributes (names) corresponding to the NVP (tuple) must be updated. A *Delete* executes a search and retrieves a match vector, which is then used for invalidation. For example, in Figure 10, to delete the record for StudentID 1020, ANVIL first executes SRCH on Search Region 1 (StudentID), which returns a match vector. Because search regions columns are laid out in the same order as data region tuples, both Search Regions 1 and 2 (which are both linked to the Students table) will have the same ordering. Therefore, ANVIL reuses the match vector from Search Region 1 to send an invalidation to Search Region 2, eliminating the need for costly scan-based searching during deletion.

Second, a user can take advantage of our fused-name optimization that allows a tuple to fuse multiple attributes into a single name (key), which can be stored along a single bitline. For example, both **StudentID** and **Last** can be stored in a single bitline. When one of the attributes is not used as part of the query (e.g., StudentID), ANVIL can simply mask off those bits using "don't cares". If the query uses both attributes (e.g., StudentID **AND** LastName), ANVIL can efficiently retrieve the NVP using a single *Lookup* NVMe command. Unlike the case where a data region has multiple search regions, our fused-name optimization *does not* require additional link table entries.

Data Region Storage. Unlike transactional queries, which return one or a few records, OLAP queries typically return many records at once. As a result, the performance of an NVP data store for OLAP depends highly on key properties of the distribution of matching records across the database table(s). Given that these properties are data and workload dependent, we sweep two parameters to study ANVIL's behavior across a wide range of potential workloads: *selectivity* is the fraction of records in the database that match a query, and *locality* captures how likely the records are to share a page. For example, from Figure 11, a query that retrieves all **Halpert** records would return two NVPs, corresponding to a selectivity of 25% (2 out of 8 tuples). Because the two Halperts are on different SSD pages, the search manager must execute two reads, resulting in a locality of 0%. A locality of 100% would mean that all matching records can be retrieved using a minimum number of reads.

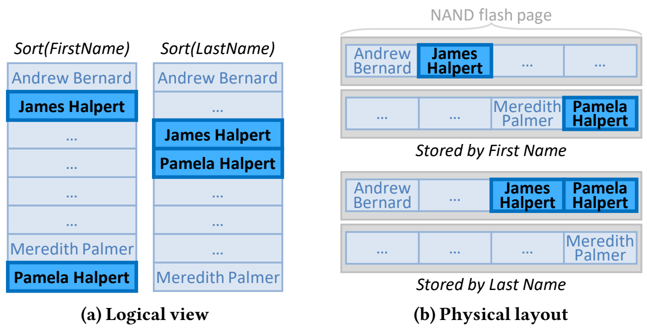


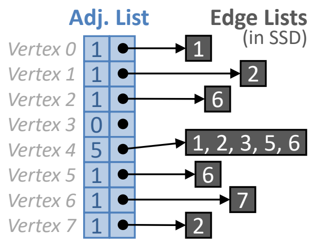
Figure 11: Example of selectivity and locality.

6.2 Graph Processing

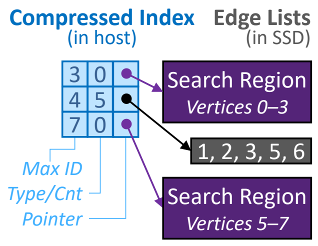
Graph processing is employed today across a wide variety of domains, ranging from social media networks [159], to roads and geographical data [84], to the connectivity of the Internet [149]. A graph processing framework [114, 167] typically initiates analytics by preprocessing the graph from the dataset's generic input format (e.g., an edge array [48, 121]) into a framework-specific format (along with applying other optimizations, e.g., [92, 142, 166, 168]). For large networks, graph frameworks use metadata structures such as in-memory indexes that allow the system to quickly locate the edges of interest (which are stored in the SSD).

While many different metadata structures exist for the index [168], we focus on *adjacency lists*, which form the basis of many graph analytics data structures. For each vertex, the adjacency list stores a count of the number of outgoing edges, and a pointer to the first edge belonging to the vertex in the edge list. For the sake of simplicity, we describe how the index can be created for out edges; however, the process can be repeated for in edges. Figure 12a shows an example adjacency list for eight vertices.

Notably, as graph datasets grow in size, the size of the index grows, requiring additional memory to store both the index and



(a) Conventional adjacency list


Compressed Index (in host)
Edge Lists (in SSD)
Search Region Vertices 0–3
1, 2, 3, 5, 6
Search Region Vertices 5–7
Max ID
Type/Cnt
Pointer

(b) ANVIL index

Figure 12: Comparison of graph index structures.

edges. Once the index exceeds the capacity of memory, systems that process large graphs experience severe performance degradation [66], requiring the system to migrate both index and graph data between memory and SSD. ANVIL can optimize SSD I/O for large graphs by eliminating the conventional adjacency list and reducing the multi-step edge retrieval process into a single *Lookup* NVMe command.

A naïve ANVIL graph format could simply forgo an in-memory index, and instead perform bulk parallel search directly on the edge array to find corresponding edges based on either source or destination vertices. This, however, is highly inefficient, because while ANVIL's *Lookup* NVMe command can search millions of edge entries simultaneously (accounting for maximum parallelism), large graphs can contain *billions or even trillions* of edges. This would require ANVIL to perform thousands of back-to-back *Lookup* NVMe commands, and could stall unrelated I/O operations by tying up all of the SSD's channels. Instead, ANVIL uses a compressed host-side in-memory structure to reduce metadata overheads.

Optimizing the Index. Graphs often follow a power law distribution [17, 23, 108, 167], where few vertices have a high degree (i.e., count of connected edges), while many vertices have a single-digit degree. If we were to allocate a dedicated search region to each vertex, most of the regions would be highly underutilized. For the common case where the out-degree of the vertex is less then the number of bitlines in the block, a *SRCH* chip command will still check all of the empty bitlines in the block, wasting both energy and area. We propose two optimizations to reduce the underutilization, resulting in the compressed index shown in Figure 12b.

First, multiple vertices with a small out-degree, and with consecutive IDs, can be compressed into a single search region. Our index stores a single entry for the search region, and stores the highest vertex ID in the Max ID column, along with a search region pointer. To access the correct search region for a vertex, the graph framework performs a binary search over the sorted Max ID field. For example, in Figure 12b, vertices 0–3 are compressed into a single search region, and vertices 5–7 are compressed into a second search region. If the framework wants to locate the region for vertex 6, it uses the entry with a Max ID of 7, since 6 falls between 4 and 7.

Second, certain vertices may contain substantially more edges than the number of bitlines per block. A *Lookup* for such a vertex could tie up multiple levels of internal SSD parallelism, and may return matches to most of its edges, resulting in little to no reduction in read count. For such vertices, we do not store their edges in a search region. Instead, we store an edge list in the data region, and keep a direct pointer to the edge list in the index (along with an edge count). We set the edge count to 0 for search region entries, to distinguish it from an edge list entry.[3] Figure 12b shows how vertex 4 has a direct pointer to its edge list in our index.

7 Methodology

To model ANVIL, we implement the *Lookup* NVMe command inside SimpleSSD [49], and observe that the execution time is 2.24× the execution time for a conventional page read. Due to the requirements of examining large datasets [146], we abstract SimpleSSD into a high-fidelity analytical model (taking inspiration from SimpleSSD and NANDFlashSim [68]), which captures the behaviors of NVMe overheads, FTL, SSD-DRAM, CPU–FE and FE–BE data movement, as well as the highly parallel back-end NAND flash. For example, the latency of the *Lookup* NVMe command (Section 4.1) includes the NVMe protocol (❶ in Figure 5), translation (❷), block-level *SRCH* to the search region (❸), match vector retrieval and decode (❹ & ❺), physical access(es) to the data region (❻), FE–BE data movement, and CPU–FE data movement (❼). Our model reports the execution time of a *Lookup* NVMe command as 2.59× the execution time of a base read request, thus adversely affecting ANVIL.

We model two different SSDs (Table 1). SSD-A, which uses a configuration consistent with modern SSDs, and SSD-B representative of an IMS-contemporary, 96-layer NAND flash. Our *Baseline* system uses a host CPU that issues standard read/write I/O requests to an SSD. The NVMe initialization overhead is set to 4 µs, based on prior work [80, 87, 136].

Table 1: 3D NAND flash configuration.

	SSD-A [110]	**SSD-B** [150]
Internal Parallelism	Channel×Dies	Channel×Dies
Channel x Package x Dies	8×1×8	4×1×4
Planes x Blocks x Pages	2×2,048×196	2×2,048×96
Page Size	16 kB	16 kB
DRAM Access Time (64 B)	11 ns	11 ns
CPU–FE Bandwidth	8.0 GB/s	2.4 GB/s
FE–BE Bandwidth	1.2 GB/s per ch.	1.2 GB/s per ch.
Read Latency	22.5 µs	60.0 µs
Search (SRCH) Latency	25.0 µs	66.6 µs
Write Lat. (SLC/MLC/TLC)	200 µs/500 µs/700 µs	200 µs/500 µs/700 µs
Search Parameters	128 k names	128 k names
Native Name Size	97 bit	47 bit

We conservatively assume that the data in the data region for both the baseline and ANVIL resides in SLC portions of the SSD. This makes the *SRCH* and read latencies comparable; in contrast, if the data resides in MLC, the read latency would be significantly larger than that of the search operation. New ANVIL-specific commands incur the overheads of the the respective operations. Our model makes two conservative assumptions that adversely affect ANVIL. First, we assume that the NAND flash access time for search is ~10% higher than a read operation and reflects the increased read latency observed in Flash-Cosmos [110]. Second, multi-block *SRCH* commands reserve all internal parallelism (e.g., channels/dies) needed by the *SRCH* command, even when the resulting match is a single data entry from just one block. Neither of these

[3]This is inspired by prior work on sparse data structures [143], including those for graph analytics, and may be applicable for other applications (e.g., sparse linear algebra).

impacts read performance in the baseline. Each NVP workload uses a workload-specific name length, ranging from 3–24 B.

OLTP. We use the TPC-C database [137] to evaluate OLTP workloads. We generate a trace of 1 M transactions for an OLTP workload using the DBx1000 DBMS [163] with a scale factor of 100. In line with state-of-the-art database table optimizations, our baseline implementation uses hash indexes that are stored in the host DRAM, eliminating the need for a scan operator.

OLAP. We examine TPC-H [139], and populate the database using DBGEN [138]. With a scale factor of 100, the resulting database has a size of 115 GB. We evaluate two analytic queries examined in prior work [51, 152], which are modified versions of TPC-H queries that scan one 78 GB table from the database.

Graphs. We use a vertex access traversal trace for the SSSP algorithm implementation [95], using a collection of synthetic and real-world graphs (Table 2). These graphs are highly sparse, with less than 1% connectivity. We examine four configurations: (1) *IM*, where Baseline uses an in-memory index points to each edge list's SSD address; (2) *OOM* (out of memory), where Baseline stores both the edge list and full index on disk; (3) *ANVIL-U*, where ANVIL uses an index without optimizations; and (4) *ANVIL-O*, where ANVIL employs an optimized index (Section 6.2). IM and OOM use state-of-the-art high-performance indexing optimizations for multicore graph processing [95]. We include all DRAM access times for the index. Unless otherwise stated, graph applications are normalized to OOM.

Table 2: Evaluated graphs.

Graph	Nodes	Edges	Graph	Nodes	Edges
Patents [83]	3.7M	16.5M	Orkut [160]	3M	117M
Road-CA [84]	1.9M	2.7M	Youtube [160]	1.1M	3M
Road-PA [84]	1.1M	1.5M	LiveJournal [10]	4.8M	69M
Road-TX [84]	1.3M	1.9M	Kron25	33.5M	1B
Twitter [159]	17M	1.5B	Mag240 [145]	121.7M	1.3B

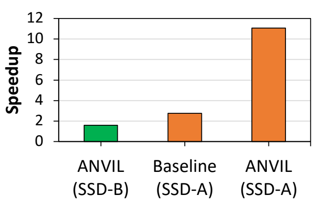
8 Evaluation

We analyze the performance and trade-offs of ANVIL using a collection of NVP applications. We begin by evaluating each of our applications from Section 6 in detail. We then study several system-level impacts of ANVIL.

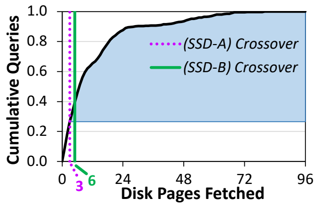
8.1 Database OLTP

Figure 13a shows the speedup achieved by ANVIL for the TPC-C workload, compared to Baseline using configuration SSD-B. TPC-C is representative of the impact of hash collisions on NVP data stores: although we would like to retrieve a matching NVP in a single I/O operation, the collision causes the system to retrieve multiple NVPs, requiring manual post-processing. Even with this additional overhead, ANVIL achieves a 4.0× and 1.6× speedup over Baseline with SSD-A and with SSD-B, respectively.

To understand why ANVIL can improve performance, we examine how many disk pages are fetched by the workload. Figure 13b shows cumulative distribution function of the number of disk pages fetched. We also plot the *crossover point*, which corresponds to the minimum number of disk pages a query must fetch for ANVIL to perform better than Baseline. We observe that for ANVIL, the crossover



(a) Speedup



(b) Breakdown of disk pages fetched

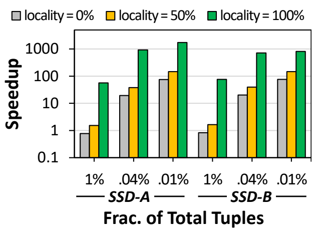
Figure 13: Speedup relative to Baseline with SSD-B. Blue region shows when ANVIL improves performance.

point for SSD-A is at three pages, meaning it can improve performance on 73.5% of the total queries performed by TPC-C. Similarly, for SSD-B, the crossover point is six pages (corresponding to 54.1% of total TPC-C queries). Because SSD-B has fewer wordlines per block, the native name size is smaller. Therefore, additional *SRCH* chip commands are needed to locate the correct name entries, increasing the search time and corresponding crossover point. Despite these overheads, ANVIL can still improve performance for OLTP.

8.2 Database OLAP

Our TPC-H OLAP workload is representative of NVP data stores that do not have an index, and must scan across every name to locate *all* relevant NVPs. On average, ANVIL speeds up the scan operator by 159× and 76× for Queries 1 and 2, respectively. These improvements are due to ANVIL's ability to quickly identify and retrieve only those data pages with queried data.

Figure 14 show the two evaluated OLAP queries compared to the scan operator on SSD-A, as we sweep various selectivity and locality values. In our sweep, we include a selectivity of 0.04% and a locality of 0% as these are the default properties observed in our synthesized database. From the figure, we make four observations. First, ANVIL speedups over the baseline range from 0.73× (1% selectivity, 0% locality) to 1568.0× (0.01% selectivity, 100% locality), with an average speedup of 117.9× across the sweep. Second, ANVIL's benefits increase as the selectivity (i.e., the fraction of total tuples that match a given query) decreases. Third, at 1% selectivity and 0% locality, ANVIL observes a slowdown compared to performing a scan using Baseline. In this workload, each data page stores approximately 117 records. With a 1% selectivity, and no locality, this means that every matching NVP is on a different SSD page. Since every page contains useful data, ANVIL cannot reduce I/O compared to a scan operator, even though it adds to the execution time by performing its *SRCH* chip commands. Fourth, ANVIL's benefits are smaller for Query 2 than for Query 1, as Query 2 contains additional constraints (predicates) in the query. However, ANVIL still preserves much of the



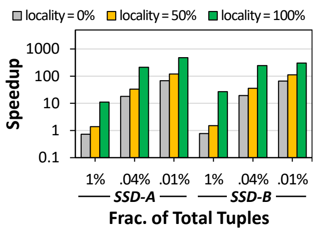


Figure 14: Speedup for Query 1 (left) and Query 2 (right) with ANVIL vs. scan using Baseline (y-axis is log scale).

speedup by taking advantage of its fused name optimization to reduce the impact of these additional constraints.

We conclude that ANVIL can substantially improve the performance of analytics, and that its benefits increase with match sparsity and with worsening locality (two common effects as the dataset size increases).

8.3 Graph

Main Memory Benefits. On average, ANVIL reduces in-memory overheads by 47.5%, compared to a baseline index with a 4 B pointer and 4 B of metadata (e.g., vertex weight) per entry. Figure 15 shows the reduction in memory footprint. From the figure, we observe that Kron25 with ANVIL-O consumes more memory than ANVIL-U. This is due to the sparse data structure requiring additional pointer nodes to manage the high-degree vertices. Note that because the in-memory index uses a per-block search region, SSD-A and SSD-B observe the same reduction in main-memory index overheads.

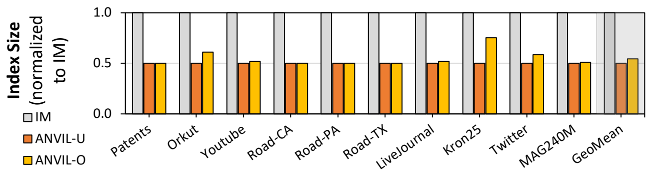


Figure 15: Graph index overhead vs. IM.

Performance. Figure 16 shows the execution time for SSSP vertex traversals on our four graph configurations using SSD-A, normalized to OOM. We make four observations from the figure. First, OOM incurs a 99% overhead over IM, averaged across all graph networks, indicating the storage access cost of large graphs in conventional systems. Second, ANVIL-U performs 11.0% better than OOM on average, as it avoids data movement costs between the CPU and the SSD. Third, while ANVIL-U improves performance for most datasets, we do see performance degradation for Kron25. This is due to the overheads incurred from moving and decoding the match vector for vertices with a high degree, which we find often in Kron25. Fourth, because ANVIL-O uses our optimized data structure, it can further improve the performance for large graphs with vertices with high degrees, with an average improvement of 14.6% and 3.6% over OOM and ANVIL-U, respectively. Without the optimized data structure, a *Lookup* command will find that high-degree vertices will be the source node for many edges. This will result in many match vectors of all ones, which will consume significant FE–BE bandwidth. By leveraging the optimized data structure, the edge lists for high-degree vertices can be directly accessed without the need for a *Lookup*. For Kron25, ANVIL-O results in a 21.3% speedup compared to ANVIL-U, showing that our optimization overcomes the performance overheads of ANVIL for high-degree vertices.

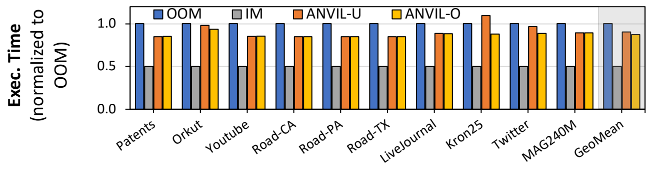


Figure 16: Execution time for SSSP (SSD-A) vs. OOM.

Internal Parallelism. From Figure 17, we can observe that when applying ANVIL to SSD-B, a slowdown occurs, for two related reasons. First, SSD-B has a native name size of 48 B, which is less than the 64 B search name needed by this workload. As a result, ANVIL must perform two *SRCH* chip commands for every set of names (i.e., vertices). Second, SSD-B has limited internal bandwidth compared to SSD-A. Therefore, the extra *SRCH* chip commands increase contention for available internal bandwidth. To verify this, we create a hypothetical SSD-C using the same parameters of SSD-B, *except* we increase the number of available channels to 20. We observe that for SSD-C, ANVIL-O can improve performance by 22.5% compared to OOM. By increasing the number of available channels, more in flight operations can proceed. Notably, increasing the number of channels does *not* increase the number of blocks required by the search. From this, we conclude that ANVIL can better utilize increasing internal SSD bandwidth, even when CPU–FE I/O (e.g., NVMe bandwidth) is limited.

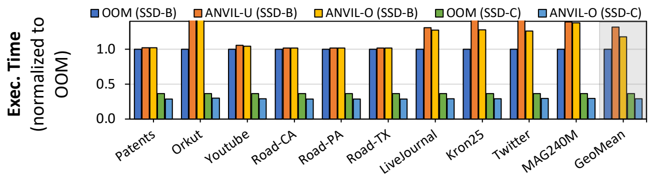


Figure 17: Execution time for SSSP (SSD-B and SSD-C) vs. OOM. IM observes the same trends as Figure 16.

8.4 System-Level Impacts

Memory and Storage Overheads. ANVIL incurs two types of storage overhead compared to a Baseline SSD. First, total drive capacity is reduced, as some blocks are used to implement the search regions. For SSD-A, we find that for TPC-C, TPC-H, and kron25, 23 (< 0.01%), 4578 (1.7%), and 8200 (3.1%) blocks (percent of total blocks) are required, respectively. Second, the link table, which resides in the SSD's DRAM, must be allocated. We find that the evaluated workloads require 2.5 kB (< 0.00% of available DRAM), 0.2 MB (< 0.01%), and 66 MB (3.2%) for TPC-C, TPC-H, and kron25, respectively. SSD-B has a native name size that is 2× smaller than SSD-A. Due to the reduced name size, two blocks must be logically linked to accommodate the longer name size, thereby increasing the storage overhead by 2× to 46 (0.07%), 9156 (14.0%), and 16400 (25.0%) blocks for TPC-C, TPC-H, and Kron25, respectively.

Within the search regions, fragmentation may occur due to two main reasons. First, when the length of a name is less than the native name size, the remaining bits along the bitline (see Section 2.1) are left unused, which contributes to the majority of fragmentation. Second, partially filled blocks (e.g., fewer names than total number of bitlines per block) leave remaining bitlines empty. We observe that for TPC-C, TPC-H, and kron25, 1.5%, 41.9%, and 38% of NAND flash cells are unused, respectively, which is already accounted for in our block overhead analysis. Our fused-name optimization (Section 6.1) can reduce fragmentation by packing multiple names (attributes) into a single bitline. Data regions have identical fragmentation to Baseline, as they operate using conventional horizontal data stores and continue to use the same table format.

Energy. We use the number of activated blocks as a proxy for energy. Figure 18 shows the number of block activations performed by ANVIL, normalized to OOM, for both SSD-A and SSD-B. Recall that typically, a *SRCH* chip command uses V_{read} for half of the wordlines (as compared to only one wordline for a read). Since V_{read} is less than V_{pass}, we expect that *SRCH* should consume less energy than a read. We observe that the greatest reduction in block activations is for the database queries, with a reduction of 93.7%. For graphs, we observe that the activation count remains the same as OOM. To retrieve the node data, the baseline must first retrieve the index page followed by the node data; in contrast, ANVIL executes a *SRCH* chip command and subsequently reads the node data.

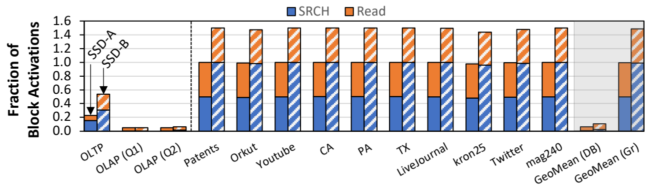


Figure 18: Block activations normalized to Baseline.

Impact on Data Movement. Figure 19 shows CPU–FE and FE–BE data movement for both SSD-A and SSD-B, normalized to Baseline with their respective SSD. From the figure, we make three observations. First, we observe an average data movement reduction of 58.6%. The greatest benefits are for the database workloads (92.3% and 97.4% reduction in FE–BE and CPU–FE data movement, respectively), due to ANVIL's ability to directly access only the relevant records and, thus, data pages, reducing the need for unnecessary data movement. By contrast, the indexes in IM and OOM already can access only the specific in-SSD vertex data, reducing ANVIL's benefits. Second, we observe greater reductions in CPU–FE data movement compared to FE–BE. This is due to the internal data movement required for the match vectors. Third, SSD-A has the potential to reduce data movement more than SSD-B. Recall from Section 4.3 that the native name size is the maximum length of a single searchable name entry in *one SRCH* chip command. Because SSD-B has a shorter native name size, the match outputs of multiple *SRCH* commands need to be logically ANDed together.

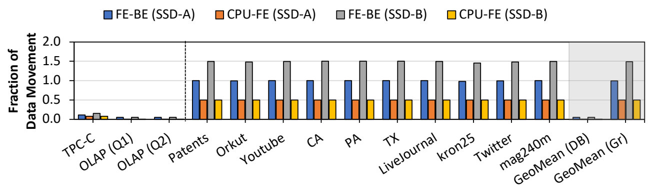


Figure 19: Data movement normalized to Baseline.

We examine the impact of our duplicate page read optimization using the graph analytics workload. When ANVIL executes a *SRCH* chip command, a single vertex may return multiple matches, which all correspond to the same location in the data region. Without our optimization, a vertex with N edges would require N reads to the data region. However, by tracking the most-recently fetched page, physically adjacent matches in the search region will not trigger duplicate page reads. Using our optimization on the graph analytics workload, we observe that data movement can be reduced by 82×.

Firmware Overhead. We profile the firmware overheads for ANVIL using gem5 [14, 88] with a dual-core in-order 4-stage Arm CPU operating at 800 MHz, which is representative of a low-end CPU used for firmware on a modern SSD. The overhead is dependent on the selectivity of a query (a higher fraction of matches incurs additional overhead). To determine a high-end overhead estimate, we measure the firmware using an average query selectivity of 1 match out of every 32 records (3.1%; which is 940× the selectivity of TPC-C). For a single-name *Lookup*, the firmware has a performance overhead of 0.9%, which increases to 2.1% for a multi-name *Lookup*. When selectivity is halved, the overhead reduces to 0.4% and 1.7%, respectively. Note that while we conservatively assume that match vectors from different *Lookup* commands are being decoded continuously by ANVIL; real-world workloads demonstrate periods of downtime or low utilization [8, 12, 89, 148], which would significantly lower the firmware overheads.

9 Discussion

Performance Impact of Workload Properties. While ANVIL can deliver significant improvements over a conventional system for many workloads, there are several workload properties that determine whether (and how much) ANVIL can provide these improvements. Key factors include (1) the size of the dataset, (2) the ratio of matches to total data (selectivity), (3) the size of the name or value data, (4) the location of the value entries in the data region (locality), and (5) the rate of updates to reads or *Lookups*. While such a multivariate space is difficult to explore exhaustively, we aim to demonstrate many diverse points across this space with our workload choices in Section 8, along with selected trends along several of these dimensions.

To provide some additional insight, we explore how the rate of updates impacts performance, using the TPC-C CUSTOMER table. To control the update rate, we devise a workload where users log into a warehouse (shopping) system (*Lookup*), and some of these users *may* place an order that modifies the database table (*Update*). Figure 20 shows the speedup of ANVIL over Baseline under different update ratios (i.e., the fraction of total queries that update a record). For our workload, a 50% update ratio implies that 50% of users that log into a system place an order. We observe from the figure that when more than 22% of the workload queries are updates, then non-ANVIL systems may perform better. However, this does not eliminate the potential usage for ANVIL. For workloads that may have regions of highly concentrated writes, followed by periods of read-heavy behavior, data can be migrated into ANVIL for the read-heavy periods to benefit from accelerated *Lookup*. Note that we use an OLTP example to illustrate the update ratio impact, as OLAP and many graph workloads tend to be highly read intensive [102, 129, 131].

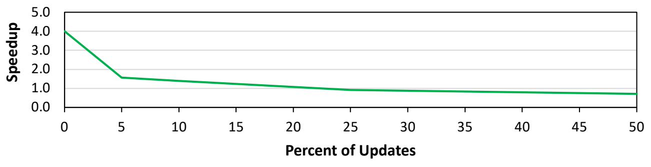


Figure 20: Speedup normalized to Baseline for update ratios.

Interacting With Main Memory. Advanced real-world NVPs leverage memory-resident structures, such as in-memory indexing structures. The performance of these structures is largely application- and hardware-specific. In our OLTP and graph analytics evaluations, we employ different types of state-of-the-art in-memory indexes for each baseline, including hash indexes and adjacency lists. These cases exemplify when we have a sufficiently large dataset that the system must go to disk to retrieve the corresponding data, and we fully allocate the host's main memory to these indexes to maximize lookup performance.

While we conservatively compare such baselines to an in-memory-index-free ANVIL, ANVIL can be optimized to work synergistically with in-memory metadata structures (e.g., indexes, buffer pools, caches). As ANVIL reduces the (random) access time of *Lookup* operations without the need for an in-memory metadata, opportunities may arise for other main-memory-based optimizations (e.g., pinning selected database tables).

Deployment in a Multi-SSD System. Although we describe ANVIL as a single-SSD system, ANVIL can be integrated into a system where data is distributed across multiple SSDs. For example, ANVIL can leverage striped and mirrored data (e.g., RAID-0/1) to enable additional parallelism for *Lookup*. In an enterprise system, where storage may be decentralized or remote, ANVIL can offer better utilization of the limited remote I/O bandwidth by reducing the amount of data transmitted between host and storage (Section 8.2). We leave full system integration for future work.

10 Related Work

Table 3 shows the comparison between ANVIL and closely related works on in-flash processing. Partial support indicates when the system supports some, but not all of a functionality or application. ANVIL's flexible nature allows it to support general NVP workloads with efficient name-or-value readouts while maintaining support for multiple different NVP optimizations (e.g., multi-name search) through the use of its dual-data representation and link table. Similarly, our proposed firmware changes allow the system to integrate ANVIL as both a storage drive and as an NVP accelerator. We discuss a wider range of prior work below.

Table 3: Prior work on in-flash processing.

IMS: In-Memory Search [140], PB: ParaBit [46], FC: Flash-Cosmos [110], AF: Ares-Flash [22], SM: Search-in-Memory [24]
○: not supported; ●: supported; ◐: partially supported

Features	**IMS**	**PB**	**FC**	**AF**	**SM**	**ANVIL**
Database Support	◐	◐	◐	◐	●	●
Graph Support	○	◐	◐	◐	○	●
General NVP Support	◐	◐	◐	◐	◐	●
Firmware Support	◐	●	●	●	●	●
Name Readout	Bit-Serl	Bit-Serl	Bit-Serl	Bit-Serl	Multi-Rd	One Rd
Value Readout	Bit-Serl	Bit-Serl	Bit-Serl	One Rd	Multi-Rd	One Rd
Multi-Name Support	○	○	○	○	●	●
Arbitrary Name/Value Size	○	○	○	○	◐	●

NVP Acceleration. A wide variety of prior works have explored improvements for specific NVP data store formats, including key–value stores [64, 69, 72], graph analytics [4, 30, 55, 79, 86, 92, 94, 121, 128], and databases [21, 75] using both traditional memories (e.g., DRAM [5, 15], NAND flash [66, 96, 134, 167]) as well as newer non-volatile memories (e.g., ReRAM [85, 133, 147], Optane [54, 125, 153]). ANVIL is compatible with many of these software systems, and can be used for larger datasets compared to in-memory solutions, while accelerating many different types of NVP data stores.

In-Flash Processing. Prior work has explored using NAND flash memory in both the digital [27, 46, 110, 126, 141] and analog [56–59] (matrix–vector multiplication, vector similarity search) PUM domains. Analog NAND-flash-based architectures require substantial array peripheral overheads (e.g., analog–digital converters) and are domain-specific solutions. Digital IFP architectures often require modifications to the the page buffer [22, 24, 25, 46], and require specific data layouts, which are incompatible NVP systems [110, 140] ANVIL leverages existing NAND flash array peripherals, without modifications to the page buffer, and provides a flexible framework that can be applied to a wide variety of applications.

Computational SSDs. Computational SSDs (i.e., smart SSDs) exploit the internal microcontroller [37, 71, 77, 78, 116, 123] or nearby FPGAs [2, 29, 122] to perform processing-near-memory. Other approaches [11, 26, 65, 73, 74, 93, 123] place additional hardware throughout the SSD hierarchy to enable in-storage computation. Some in-storage computing platforms target query processing [59, 112, 124, 154] or key–value interfaces [8, 32, 33, 61, 64, 155]. ANVIL is orthogonal to these computational SSD solutions, and can benefit from offloading certain operations to embedded compute elements.

11 Conclusion

Name–value pairs (NVPs) are a family of data stores that are used in many different ways across millions of programs to manage data. In this work, we propose ANVIL, the first accelerator that can broadly work across different NVP data store formats. ANVIL provides an end-to-end solution for NVPs, making use of parallel in-storage searches to significantly reduce the amount of disk I/O required for large-scale NVP data stores. We demonstrate that ANVIL can provide speedups of 4.0×, 25×, and 14.6% over a conventional SSD for OLTP, OLAP, and graph workloads, respectively.

Acknowledgments

This article has been authored by an employee of National Technology & Engineering Solutions of Sandia, LLC under Contract No. DE-NA0003525 with the U.S. Department of Energy (DOE). The employee owns all right, title and interest in and to the article and is solely responsible for its contents. The United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this article or allow others to do so, for United States Government purposes. The DOE will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan (https://www.energy.gov/downloads/doe-public-access-plan).

We thank Christopher H. Bennett and T. Patrick Xiao for their feedback, suggestions, and support throughout this project.

References

[1] D. J. Abadi, P. A. Boncz, and S. Harizopoulos. 2009. Column-Oriented Database Systems. *Proc. VLDB Endow.* (Aug. 2009).

[2] Advanced Micro Devices, Inc. 2021. Samsung SmartSSD. https://www.xilinx.com/applications/data-center/computational-storage/smartssd.html.

[3] R. Agrawal and R. Srikant. 1994. Fast Algorithms for Mining Association Rules. In *VLDB*.

[4] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi. 2015. A Scalable Processing-in-Memory Accelerator for Parallel Graph Processing. In *ISCA*.

[5] M. Ahn, A. Chang, D. Lee, J. Gim, J. Kim, J. Jung, O. Rebholz, V. Pham, K. Malladi, and Y. S. Ki. 2022. Enabling CXL Memory Expansion for In-Memory Database Management Systems. In *DaMoN*.

[6] A. V. Aho and J. D. Ullman. 1992. *Foundations of Computer Science*. Computer Science Press, Inc.

[7] F. Alibart, T. Sherwood, and D. B. Strukov. 2011. Hybrid CMOS/Nanodevice Circuits for High Throughput Pattern Matching Applications. In *AHS*.

[8] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. 2012. Workload Analysis of a Large-Scale Key-Value Store. In *SIGMETRICS*.

[9] B. Marr. 2018. How Much Data Do We Create Every Day? The Mind-Blowing Stats Everyone Should Read. https://www.forbes.com/sites/bernardmarr/2018/05/21/how-much-data-do-we-create-every-day-the-mind-blowing-stats-everyone-should-read/?sh=667f02260ba9.

[10] L Backstrom, D. Huttenlocher, J. Kleinberg, and X. Lan. 2006. Group Formation in Large Social Networks: Membership, Growth, and Evolution. In *KDD*.

[11] D.-H. Bae, J.-H. Kim, S.-W. Kim, H. Oh, and C. Park. 2013. Intelligent SSD: A Turbo for Big Data Mining. In *CIKM*.

[12] L. A. Barroso, U. Hölzle, and P. Ranganathan. 2019. *The Datacenter as a Computer: Designing Warehouse-Scale Machines*. Springer Nature.

[13] K. E. Batcher. 1974. STARAN Parallel Processor System Hardware. In *AFIPS*.

[14] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. 2011. The gem5 Simulator. *CAN* (Aug. 2011).

[15] A. Boroumand, S. Ghose, G. F. Oliveira, and O. Mutlu. 2022. Polynesia: Enabling High-Performance and Energy-Efficient Hybrid Transactional/Analytical Databases With Hardware/Software Co-Design. In *ICDE*.

[16] T. Bray, J. M. Paoli, C. M. Sperberg-McQueen Sperberg-McQueen, E. Maler, and F. Yergeau. 2008. Extensible Markup Language (XML) 1.0 (Fifth Edition). https://www.w3.org/TR/2008/REC-xml-20081126/.

[17] S. Brin and L. Page. 1998. The Anatomy of a Large-Scale Hypertextual Web Search Engine. *Computer Networks and ISDN Systems* (Apr. 1998).

[18] Y. Cai, S. Ghose, E. F. Haratsch, Y. Luo, and O. Mutlu. 2017. Error Characterization, Mitigation, and Recovery in Flash-Memory-Based Solid-State Drives. *Proc. IEEE* (Sep. 2017).

[19] Y. Cai, Y. Luo, S. Ghose, and O. Mutlu. 2015. Read Disturb Errors in MLC NAND Flash Memory: Characterization, Mitigation, and Recovery. In *DSN*.

[20] Y. Cai, G. Yalcin, O. Mutlu, E. F. Haratsch, O. Unsal, A. Cristal, and K. Mai. 2014. Neighbor-Cell Assisted Error Correction for MLC NAND Flash Memories. In *SIGMETRICS*.

[21] H. Caminal, Y. Chronis, T. Wu, J. M. Patel, and J. F. Martínez. 2022. Accelerating Database Analytic Query Workloads Using an Associative Processor. In *ISCA*.

[22] J. Chen, C. Gao, Y. Lu, Y. Zhang, and J. Shu. 2024. Ares-Flash: Efficient Parallel Integer Arithmetic Operations Using NAND Flash Memory. In *MICRO*.

[23] R. Chen, X. Weng, B. He, M. Yang, B. Choi, and X. Li. 2010. *On the Efficiency and Programmability of Large Graph Processing in the Cloud*. Technical Report MSR-TR-2010-44. Microsoft Research.

[24] Y.C. Chen, Y.H. Chang, and T.W. Kuo. 2024. Search-In-Memory: Reliable, Versatile, and Efficient Data Matching in SSD's NAND Flash Memory Chip for Data Indexing Acceleration. *IEEE TCAD* (Nov. 2024).

[25] Y.C. Chen, Y.H. Chang, and T.W. Kuo. 2024. Search-in-Memory (SiM): Conducting Data-Bound Computations on Flash Chip for Enhanced Efficiency. In *DATE*.

[26] B. Y Cho, W. S. Jeong, D. Oh, and W. W. Ro. 2013. XSD: Accelerating MapReduce by Harnessing the GPU Inside an SSD. In *DIMES*.

[27] W. H. Choi, P.-F. Chiu, W. Ma, G. Hemink, T. T. Hoang, M. Lueker-Boden, and Z. Bandic. 2020. An In-Flash Binary Neural Network Accelerator With SLC NAND Flash Array. In *ISCAS*.

[28] A. Chowdhery, S. Narang, J. Devlin, M. Bosma, G. Mishra, A. Roberts, P. Barham, H. W. Chung, C. Sutton, S. Gehrmann, P. Schuh, K. Shi, S. Tsvyashchenko, J. Maynez, A. Rao, P. Barnes, Y. Tay, N. Shazeer, V. Prabhakaran, E. Reif, N. Du, B. Hutchinson, R. Pope, J. Bradbury, J. Austin, M. Isard, G. Gur-Ari, P. Yin, T. Duke, A. Levskaya, S. Ghemawat, S. Dev, H. Michalewski, X. Garcia, V. Misra, K. Robinson, L. Fedus, D. Zhou, D. Ippolito, D. Luan, H. Lim, B. Zoph, A. Spiridonov, R. Sepassi, D. Dohan, S. Agrawal, M. Omernick, A. M. Dai, T. S. Pillai, M. Pellat, A. Lewkowycz, E. Moreira, R. Child, O. Polozov, K. Lee, Z. Zhou, X. Wang, B. Saeta, M. Diaz, O. Firat, M. Catasta, J. Wei, Kathy M.-H., D. Eck, J. Dean, S. Petrov, and N. Fiedel. 2023. PaLM: Scaling Language Modeling With Pathways. *JMLR* (Aug. 2023).

[29] CRZ Technology. 2019. Daisy OpenSSD. http://www.mangoboard.com/main/view.asp?idx=1061&pageNo=1&cate1=9&cate2=150&cate3=181

[30] V. Dadu, S. Liu, and T. Nowatzki. 2021. PolyGraph: Exposing the Value of Flexibility for Graph Processing Accelerators. In *ISCA*.

[31] B. Dally. 2015. Challenges for Future Computing Systems. Keynote talk at HiPEAC.

[32] B. Debnath, S. Sengupta, and J. Li. 2010. FlashStore: High Throughput Persistent Key-Value Store. *Proc. VLDB Endow.* (Sep. 2010).

[33] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. 2007. Dynamo: Amazon's Highly Available Key-Value Store. *OSR* (Oct. 2007).

[34] N. Derharcobian and C. N. Murphy. 2010. Phase-Change Memory (PCM) Based Universal Content-Addressable Memory (CAM) Configured as Binary/Ternary CAM. U.S. Patent No. 7,675,765 B2.

[35] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-Training of Deep Bidirectional Transformers for Language Understanding.

[36] C. Diaconu, C. Freedman, E. Ismert, P.-Å. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. 2013. Hekaton: SQL Server's Memory-Optimized OLTP Engine. In *SIGMOD*.

[37] J. Do, Y.-S. Kee, J. M. Patel, C. Park, K. Park, and D. J. DeWitt. 2013. Query Processing on Smart SSDs: Opportunities and Challenges. In *SIGMOD*.

[38] Domo, Inc. 2014. Data Never Sleeps 2.0. https://www.domo.com/learn/infographic/data-never-sleeps-2.

[39] Domo, Inc. 2023. Data Never Sleeps 11.0. https://www.domo.com/learn/infographic/data-never-sleeps-11.

[40] I. döt Net, T. Müller, P. Antoniou, E. Aro, and T. Smith. 2021. YAML Ain't Markup Language (YAML™) version 1.2. https://yaml.org/spec/1.2.2/.

[41] ECMA International. 2017. The JSON Data Interchange Syntax. https://ecma-international.org/publications-and-standards/standards/ecma-404/.

[42] A. Eldawy, J. Levandoski, and P.-Å. Larson. 2014. Trekking Through Siberia: Managing Cold Data in a Memory-Optimized Database. *Proc. VLDB Endow.* (Jul. 2014).

[43] K. Eshraghian, K.-R. Cho, O. Kavehei, S.-K. Kang, D. Abbott, and S.-M. S. Kang. 2010. Memristor MOS Content Addressable Memory (MCAM): Hybrid Architecture for Future High Performance Search Engines. *TVLSI* (Aug. 2010).

[44] Facebook. 2024. RocksDB. https://github.com/facebook/rocksdb.

[45] J. D. Ferreira, G. Falcao, J. Gómez-Luna, M. Alser, L. Orosa, M. Sadrosadati, J. S. Kim, G. F. Oliveira, T. Shahroodi, A. Nori, and O. Mutlu. 2022. pLUTo: Enabling Massively Parallel Computation in DRAM via Lookup Tables. In *MICRO*.

[46] C. Gao, X. Xin, Y. Lu, Y. Zhang, J. Yang, and J. Shu. 2021. ParaBit: Processing Parallel Bitwise Operations in NAND Flash Memory Based SSDs. In *MICRO*.

[47] S. Ghose, A. Boroumand, J. S. Kim, J. Gómez-Luna, and O. Mutlu. 2019. Processing-in-Memory: A Workload-Driven Perspective. *IBM JRD* (Nov.–Dec. 2019).

[48] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. 2012. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *OSDI*.

[49] D. Gouk, M. Kwon, J. Zhang, S. Koh, W. Choi, N. S. Kim, M. Kandemir, and M. Jung. 2018. Amber: Enabling Precise Full-System Simulation With Detailed Modeling of All SSD Resources. In *MICRO*.

[50] J. N. Gray. 1978. *Notes on Data Base Operating Systems*. Springer Berlin Heidelberg.

[51] B. Gu, A. S. Yoon, D.-H. Bae, I. Jo, J. Lee, J. Yoon, J.-U. Kang, M. Kwon, C. Yoon, S. Cho, J. Jeong, and D. Chang. 2016. Biscuit: A Framework for Near-Data Processing of Big Data Workloads. In *ISCA*.

[52] Q. Guo, X. Guo, Y. Bai, and E. İpek. 2011. A Resistive TCAM Accelerator for Data-Intensive Computing. In *MICRO*.

[53] Q. Guo, X. Guo, R. Patel, E. Ipek, and E. G Friedman. 2013. AC-DIMM: Associative Computing With STT-MRAM. In *ISCA*.

[54] F. T. Hady, A. Foong, B. Veal, and D. Williams. 2017. Platform Storage Performance With 3D XPoint Technology. *Proc. IEEE* (Sep. 2017).

[55] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi. 2016. Graphicionado: A High-Performance and Energy-Efficient Accelerator for Graph Analytics. In *MICRO*.

[56] P.-K. Hsu, V. Garg, A. Lu, and S. Yu. 2024. A Heterogeneous Platform for 3D NAND-Based In-Memory Hyperdimensional Computing Engine for Genome Sequencing Applications. *TCAS-I* (Apr. 2024).

[57] P.-K. Hsu, W. Xu, T. Rosing, and S. Yu. 2023. An In-Storage Processing Architecture With 3D NAND Heterogeneous Integration for Spectra Open Modification Search. In *MEMSYS*.

[58] P.-K. Hsu and S. Yu. 2022. In-Memory 3D NAND Flash Hyperdimensional Computing Engine for Energy-Efficient SARS-CoV-2 Genome Sequencing. In *IMW*.

[59] H.-W. Hu, W.-C. Wang, Y.-H. Chang, Y.-C. Lee, B.-R. Lin, H.-M. Wang, Y.-P. Lin, Y.-M. Huang, C.-Y. Lee, T.-H. Su, C.-C. Hsieh, C.-M. Hu, Y.-T. Lai, C.-K. Chen, H.-S. Chen, H.-P. Li, T.-W. Kuo, M.-F. Chang, K.-C. Wang, C.-H. Hung, and C.-Y. Lu. 2022. ICE: An Intelligent Cognition Engine With 3D NAND-Based In-Memory Computing for Vector Similarity Search Acceleration. In *MICRO*.

[60] P. Huang, P. Subedi, X. He, S. He, and K. Zhou. 2014. FlexECC: Partially Relaxing ECC of MLC SSD for Better Cache Performance. In *USENIX ATC*.

[61] J. Im, J. Bae, C. Chung, Arvind, and S. Lee. 2020. PinK: High-Speed In-Storage Key-Value Store With Bounded Tails. In *USENIX ATC*.

[62] J.-F. Im, K. Gopalakrishna, S. Subramaniam, M. Shrivastava, A. Tumbde, X. Jiang, J. Dai, S. Lee, N. Pawar, J. Li, and R. Aringunram. 2018. Pinot: Realtime OLAP

for 530 Million Users. In *SIGMOD*.

[63] S. Jeloka, N. B. Akesh, D. Sylvester, and D. Blaauw. 2016. A 28 nm Configurable Memory (TCAM/BCAM/SRAM) Using Push-Rule 6T Bit Cell Enabling Logic-in-Memory. In *JSSC*.

[64] Y. Jin, H.-W. Tseng, Y. Papakonstantinou, and S. Swanson. 2017. KAML: A Flexible, High-Performance Key-Value SSD. In *HPCA*.

[65] S.-W. Jun, M. Liu, S. Lee, J. Hicks, J. Ankcorn, M. King, S. Xu, and Arvind. 2015. BlueDBM: An Appliance for Big Data Analytics. In *ISCA*.

[66] S.-W. Jun, A. Wright, S. Zhang, S. Xu, and Arvind. 2018. GraFBoost: Using Accelerated Flash Storage for External Graph Analytics. In *ISCA*.

[67] D. Jung, J.-U. Kang, H. Jo, J.-S. Kim, and J. Lee. 2010. Superblock FTL: A Superblock-Based Flash Translation Layer With a Hybrid Address Translation Scheme. *ACM Trans. Embed. Comput. Syst.* (Apr. 2010).

[68] M. Jung, W. Choi, S. Gao, E. H. Wilson III, D. Donofrio, J. Shalf, and M. T. Kandemir. 2016. NANDFlashSim: High-Fidelity, Microarchitecture-Aware NAND Flash Memory Simulation. *TOS* (Feb. 2016).

[69] O. Kaiyrakhmet, S. Lee, B. Nam, S. H. Noh, and Y.-R. Choi. 2019. SLM-DB: Single-Level Key-Value Store With Persistent Memory. In *FAST*.

[70] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi. 2008. H-Store: A High-Performance, Distributed Main Memory Transaction Processing System. *Proc. VLDB Endow.* (Aug. 2008).

[71] Y. Kang, Y.-S. Kee, E. L. Miller, and C. Park. 2013. Enabling Cost-Effective Data Processing With Smart SSD. In *MSST*.

[72] Y. Kang, R. Pitchumani, P. Mishra, Y.-S. Kee, F. Londono, S. Oh, J. Lee, and D. D. G. Lee. 2019. Towards Building a High-Performance, Scale-In Key-Value Storage System. In *SYSTOR*.

[73] K. Kim, R. Johnson, and I. Pandis. 2019. BionicDB: Fast and Power-Efficient OLTP on FPGA. In *EDBT*.

[74] S. Kim, H. Oh, C. Park, S. Cho, and S.-W. Lee. 2011. Fast, Energy Efficient Scan Inside Flash Memory SSDs. In *ADMS*.

[75] O. Kocberber, B. Grot, J. Picorel, B. Falsafi, K. Lim, and P. Ranganathan. 2013. Meet the Walkers Accelerating Index Traversals for In-Memory Databases. In *MICRO*.

[76] T. Kohonen. 1980. *Content-Addressable Memories*. Springer-Verlag.

[77] G. Koo, K. K. Matam, T. I, H. V. K. G. Narra, J. Li, H.-W. Tseng, S. Swanson, and M. Annavaram. 2017. Summarizer: Trading Communication With Computing Near Storage. In *MICRO*.

[78] J. Kwak, S. Lee, K. Park, J. Jeong, and Y. H. Song. 2020. Cosmos+ OpenSSD: Rapid Prototype for Flash Storage Systems. *TOS* (Jul. 2020).

[79] A. Kyrola, G. Blelloch, and C. Guestrin. 2012. GraphChi: Large-Scale Graph Computation on Just a PC. In *OSDI*.

[80] J. Lawley. 2014. *Understanding Performance of PCI Express Systems*. White Paper WP350. Xilinx, Inc.

[81] J. Lee, H.-S. Im, D.-S. Byeon, K.-H. Lee, D. H. Chae, K.-H. Lee, Y.-H. Lim, J.-D. Choi, Y.-I. Seo, J.-S. Lee, and K.-D. Suh. 2002. A 1.8V 1Gb NAND Flash Memory With 0.12μm STI Process Technology. In *ISSCC*.

[82] T. J. Lehman and M. J. Carey. 1986. A Study of Index Structures for Main Memory Database Management Systems. In *VLDB*.

[83] J. Leskovec, J. Kleinberg, and C. Faloutsos. 2005. Graphs Over Time: Densification Laws, Shrinking Diameters and Possible Explanations. In *KDD*.

[84] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney. 2009. Community Structure in Large Networks: Natural Cluster Sizes and the Absence of Large Well-Defined Clusters. *Internet Math.* (Jan. 2009).

[85] H. Li, H. Jin, L. Zheng, and X. Liao. 2020. ReSQM: Accelerating Database Operations Using ReRAM-Based Content Addressable Memory. *IEEE TCAD* (Nov. 2020).

[86] H. Liu and H. H. Huang. 2017. Graphene: Fine-Grained IO Management for Graph Computing. In *FAST*.

[87] J. Liu, A. Mamidala, A. Vishnu, and D. K. Panda. 2004. Performance Evaluation of InfiniBand With PCI Express. In *HOTI*.

[88] J. Lowe-Power, A. M. Ahmad, A. Akram, M. Alian, R. Amslinger, M. Andreozzi, A. Armejach, N. Asmussen, S. Bharadwaj, G. Black, G. Bloom, B. R. Bruce, D. R. Carvalho, J. Castrillón, L. Chen, N. Derumigny, S. Diestelhorst, W. Elsasser, M. Fariborz, A. F. Farahani, P. Fotouhi, R. Gambord, J. Gandhi, D. Gope, T. Grass, B. Hanindhito, A. Hansson, S. Haria, A. Harris, T. Hayes, A. Herrera, M. Horsnell, S. A. R. Jafri, R. Jagtap, H. Jang, R. Jeyapaul, T. M. Jones, M. Jung, S. Kannoth, H. Khaleghzadeh, Y. Kodama, T. Krishna, T. Marinelli, C. Menard, A. Mondelli, T. Mück, O. Naji, K. Nathella, H. Nguyen, N. Nikoleris, L. E. Olson, M. S. Orr, B. Pham, P. Prieto, T. Reddy, A. Roelke, M. Samani, A. Sandberg, J. Setoain, B. Shingarov, M. D. Sinclair, T. Ta, R. Thakur, G. Travaglini, M. Upton, N. Vaish, I. Vougioukas, Z. Wang, N. Wehn, C. Weis, D. A. Wood, H. Yoon, and E. F. Zulian. 2020. The gem5 Simulator: Version 20.0+. https://arxiv.org/abs/2007.03152.

[89] C. Lu, K. Ye, G. Xu, C.-Z. Xu, and T. Bai. 2017. Imbalance in the Cloud: An Analysis on Alibaba Cluster Trace. In *Big Data*.

[90] Y. Luo, S. Ghose, Y. Cai, E. F. Haratsch, and O. Mutlu. 2018. HeatWatch: Improving 3D NAND Flash Memory Device Reliability by Exploiting Self-Recovery and Temperature Awareness. In *HPCA*.

[91] Y. Luo, S. Ghose, E.F. Haratsch, and O. Mutlu. 2016. Enabling Accurate and Practical Online Flash Channel Modeling for Modern MLC NAND Flash Memory. *JSAC* (Sep. 2016).

[92] S. Maass, C. Min, S. Kashyap, W. Kang, M. Kumar, and T. Kim. 2017. Mosaic: Processing a Trillion-Edge Graph on a Single Machine. In *EuroSys*.

[93] V. S. Mailthody, Z. Qureshi, W. Liang, Z. Feng, S. G. de Gonzalo, Y. Li, H. Franke, J. Xiong, J. Huang, and W.-m. Hwu. 2019. DeepStore: In-Storage Acceleration for Intelligent Queries. In *MICRO*.

[94] G. Malewicz, M. H. Austern, A.J.C Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. 2010. Pregel: A System for Large-Scale Graph Processing. In *SIGMOD*.

[95] J. Malicevic, B. Lepers, and W. Zwaenepoel. 2017. Everything You Always Wanted to Know About Multicore Graph Processing but Were Afraid to Ask. In *USENIX ATC*.

[96] K. K. Matam, G. Koo, H. Zha, H.-W. Tseng, and M. Annavaram. 2019. GraphSSD: Graph Semantics Aware SSD. In *ISCA*.

[97] C. Matsui, C. Sun, and K. Takeuchi. 2017. Design of Hybrid SSDs With Storage Class Memory and NAND Flash Memory. *Proc. IEEE* (2017).

[98] S. Matsunaga, K. Hiyama, A. Matsumoto, S. Ikeda, H. Hasegawa, K. Miura, J. Hayakawa, T. Endoh, H. Ohno, and T. Hanyu. 2009. Standby-Power-Free Compact Ternary Content-Addressable Memory Cell Chip Using Magnetic Tunnel Junction Devices. *APEX* (Feb. 2009).

[99] S. Matsunaga, A. Katsumata, M. Natsui, S. Fukami, T. Endoh, H. Ohno, and T. Hanyu. 2011. Fully Parallel 6T-2MTJ Nonvolatile TCAM With Single-Transistor-Based Self Match-Line Discharge Control. In *VLSIC*.

[100] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. 1992. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *TODS* (Mar. 1992).

[101] C. Mohan and F. Levine. 1992. ARIES/IM: An Efficient and High Concurrency Index Management Method Using Write-Ahead Logging. In *SIGMOD*.

[102] L. Nai, Y. Xia, I. G. Tanase, H. Kim, and C.-Y. Lin. 2015. GraphBIG: Understanding Graph Computing in the Context of Industrial Solutions. In *SC*.

[103] R. Nair, S. F. Antao, C. Bertolli, P. Bose, J. R. Brunheroto, T. Chen, C.-Y. Cher, C. H. A. Costa, J. Doi, C. Evangelinos, B. M. Fleischer, T. W. Fox, D. S. Gallo, L. Grinberg, J. A. Gunnels, A. C. Jacob, P. Jacob, H. M. Jacobson, T. Karkhanis, C. Kim, J. H. Moreno, J. K. O'Brien, M. Ohmacht, Y. Park, D. A. Prener, B. S. Rosenburg, K. D. Ryu, O. Sallenave, M. J. Serrano, P. D. M. Siegl, K. Sugavanam, and Z. Sura. 2015. Active Memory Cube: A Processing-in-Memory Architecture for Exascale Systems. *IBM JRD* (Mar.–May 2015).

[104] S. Narla, P. Kumar, A. F. Laguna, D. Reis, X. S. Hu, M. Niemier, and A. Naeemi. 2023. Design of a Compact Spin-Orbit-Torque-Based Ternary Content Addressable Memory. *TED* (Feb. 2023).

[105] NVM Express, Inc. 2021. *NVM Express® NVM Command Set Specification, Revision 1.0a*.

[106] NVM Express, Inc. 2022. *NVM Express® Key Value Command Set Specification, Revision 1.0b*.

[107] G. F. Oliveira, J. Gómez-Luna, L. Orosa, S. Ghose, N. Vijaykumar, I. Fernandez, M. Sadrosadati, and O. Mutlu. 2021. DAMOV: A New Methodology and Benchmark Suite for Evaluating Data Movement Bottlenecks. *IEEE Access* (2021).

[108] L. Page, S. Brin, R. Motwani, and T. Winograd. 1999. *The PageRank Citation Ranking: Bringing Order to the Web*. Technical Report 1999-66. Stanford Univ. InfoLab.

[109] K. Pagiamtzis and A. Sheikholeslami. 2006. Content-Addressable Memory (CAM) Circuits and Architectures: A Tutorial and Survey. *JSSC* (Feb. 2006).

[110] J. Park, R. Azizi, G. F. Oliveira, M. Sadrosadati, R. Nadig, D. Novo, J. Gómez-Luna, M. Kim, and O. Mutlu. 2022. Flash-Cosmos: In-Flash Bulk Bitwise Operations Using Inherent Computation Capability of NAND Flash Memory. In *MICRO*.

[111] J. Park, M. Kim, M. Chun, L. Orosa, J. Kim, and O. Mutlu. 2021. Reducing Solid-State Drive Read Latency by Optimizing Read-Retry. In *ASPLOS*.

[112] J.-H. Park, S. Choi, G. Oh, and S.-W. Lee. 2021. SaS: SSD as SQL Database System. *Proc. VLDB Endow.* (May 2021).

[113] K.-T. Park, M. Kang, D. Kim, S.-W. Hwang, B. T. Choi, Y.-T. Lee, C. Kim, and K. Kim. 2008. A Zeroing Cell-to-Cell Interference Page Architecture With Temporary LSB Storing and Parallel MSB Program Scheme for MLC NAND Flash Memories. *JSSC* (2008).

[114] R. Pearce, M. Gokhale, and N. M. Amato. 2010. Multithreaded Asynchronous Graph Traversal for In-Memory and Semi-External Memory. In *SC*.

[115] Python Software Foundation. 2024. The Python Tutorial: Dictionaries. https://docs.python.org/3/tutorial/datastructures.html#dictionaries.

[116] R. Cheerla. 2019. Computational SSDs. https://www.snia.org/sites/default/files/SDCEMEA/2019/Presentations/Computational_SSDs_Final.pdf. Talk at SDC EMEA.

[117] S. Rajbhandari, O. Ruwase, J. Rasley, S. Smith, and Y. He. 2021. ZeRO-Infinity: Breaking the GPU Memory Wall for Extreme Scale Deep Learning. In *SC*.

[118] B. Rajendran, R. W. Cheek, L. A. Lastras, M. M. Franceschini, M. J. Breitwisch, A. Schrott, J. Li, R. Montoye, L. Chang, and C. Lam. 2011. Demonstration of CAM and TCAM Using Phase Change Devices. In *IMW*.

[119] D. Reinsel, J. Gantz, and J. Rydning. 2018. *Data Age 2025: The Digitization of the World From Edge to Core*. Technical Report. IDC.

[120] K. Rothermel and C. Mohan. 1989. ARIES/NT: A Recovery Method Based on Write-Ahead Logging for Nested Transactions.. In *"VLDB"*.

[121] A. Roy, I. Mihailovic, and W. Zwaenepoel. 2013. X-Stream: Edge-Centric Graph Processing Using Streaming Partitions. In *SOSP*.
[122] S. Salamat, A. Haj Aboutalebi, B. Khaleghi, J. H. Lee, Y. S. Ki, and T. Rosing. 2021. NASCENT: Near-Storage Acceleration of Database Sort on SmartSSD. In *FPGA*.
[123] Samsung Electronics Co., Ltd. 2022. Samsung Electronics Develops Second-Generation SmartSSD Computational Storage Drive With Upgraded Processing Functionality. https://news.samsung.com/global/samsung-electronics-develops-second-generation-smartssd-computational-storage-drive-with-upgraded-processing-functionality.
[124] S. Seshadri, M. Gahagan, S. Bhaskaran, T. Bunker, A. De, Y. Jin, Y. Liu, and S. Swanson. 2014. Willow: A User-Programmable SSD. In *OSDI*.
[125] A. Shanbhag, N. Tatbul, D. Cohen, and S. Madden. 2020. Large-Scale In-Memory Analytics on Intel® Optane™ DC Persistent Memory. In *DaMoN*.
[126] W. Shim and S. Yu. 2022. GP3D: 3D NAND Based In-Memory Graph Processing Accelerator. *JETCAS* (Jun. 2022).
[127] S.-H. Shin, D.-K. Shim, J.-Y. Jeong, O.-S. Kwon, S.-Y. Yoon, M.-H. Choi, T.-Y. Kim, H.-W. Park, H.-J. Yoon, Y.-S. Song, Y.-H. Choi, S.-W. Shim, Y.-L. Ahn, K.-T. Park, J.-M. Han, K.-H. Kyung, and Y.-H. Jun. 2012. A New 3-Bit Programming Algorithm Using SLC-to-TLC Migration for 8MB/S High Performance TLC NAND Flash Memory. In *VLSIC*.
[128] J. Shun and G. E. Blelloch. 2013. Ligra: A Lightweight Graph Processing Framework for Shared Memory. In *PPoPP*.
[129] U. Sirin and A. Ailamaki. 2020. Micro-architectural Analysis of OLAP: Limitations and Opportunities. *Proc. VLDB Endow.* (Feb. 2020).
[130] R. Stoica and A. Ailamaki. 2013. Enabling Efficient OS Paging for Main-Memory OLTP Databases. In *DaMoN*.
[131] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik. 2005. C-store: A Column-oriented DBMS. In *VLDB*.
[132] C. Sun, C. Li, S. Samanta, K. Han, Z. Zheng, J. Zhang, Q. Kong, H. Xu, Z. Zhou, Y. Chen, C. Zhuo, K. Ni, X. Yin, and X. Gong. 2022. Computational Associative Memory With Amorphous Metal-Oxide Channel 3D NAND-Compatible Floating-Gate Transistors. *Advanced Electronic Materials* (Sep. 2022).
[133] Y. Sun, Y. Wang, and H. Yang. 2017. Energy-Efficient SQL Query Exploiting RRAM-Based Process-in-Memory Structure. In *NVMSA*.
[134] T. Suzuki, K. Hiwada, H. Kajihara, S. Sano, S. Nomura, and T. Shiozawa. 2021. Approaching DRAM Performance by Using Microsecond-Latency Flash Memory for Small-Sized Random Read Accesses: A New Access Method and Its Graph Applications. *Proc. VLDB Endow.* (Apr. 2021).
[135] B. S. Swaroop, A. Saxena, and S. Sahay. 2024. Satisfiability Attack-Resilient Camouflaged Multiple Multivariable Logic-in-Memory Exploiting 3D NAND Flash Array. *TCAS-I* (Feb. 2024).
[136] A. Tavakkol, J. Gómez-Luna, M. Sadrosadati, S. Ghose, and O. Mutlu. 2018. MQSim: A Framework for Enabling Realistic Studies of Modern Multi-Queue SSD Devices. In *FAST*.
[137] Transaction Processing Council. 2010. TPC-C Benchmark. https://www.tpc.org/tpcc/
[138] Transaction Processing Council. 2011. TPC-H DBGEN. https://github.com/electrum/tpch-dbgen.
[139] Transaction Processing Council. 2021. TPC-H Benchmark. https://www.tpc.org/tpch/
[140] P.-H. Tseng, F.-M. Lee, Y.-H. Lin, L.-Y. Chen, Y.-C. Li, H.-W. Hu, Y.-Y. Wang, C.-C. Hsieh, M.-H. Lee, H.-L. Lung, K.-Y. Hsieh, K.-C. Wang, and C.-Y. Lu. 2020. In-Memory-Searching Architecture Based on 3D-NAND Technology With Ultra-High Parallelism. In *IEDM*.
[141] P.-H. Tseng, F.-M. Lee, Y.-H. Lin, Y.-Y. Wang, M.-H. Lee, K.-Y. Hsieh, K.-C. Wang, and C.-Y. Lu. 2021. A Hybrid In-Memory-Searching and In-Memory-Computing Architecture for NVM Based AI Accelerator. In *VLSI*.
[142] K. Vora, G. Xu, and R. Gupta. 2016. Load the Edges You Need: A Generic I/O Optimization for Disk-Based Graph Processing. In *USENIX ATC*.
[143] R. Vuduc. 2003. *Automatic Performance Tuning of Sparse Matrix Kernels*. Ph. D. Dissertation. Univ. of California, Berkeley.
[144] J. P. Wade and C. G. Sodini. 1987. Dynamic Cross-Coupled Bit-Line Content Addressable Memory Cell for High-Density Arrays. *JSSC* (Feb. 1987).
[145] K. Wang, Z. Shen, C. Huang, C.-H. Wu, Y. Dong, and A. Kanakia. 2020. Microsoft Academic Graph: When Experts Are Not Enough. *QSS* (Feb. 2020).
[146] L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang, C. Zheng, G. Lu, K. Zhan, X. Li, and B. Qiu. 2014. BigDataBench: A Big Data Benchmark Suite From Internet Services. In *HPCA*.
[147] P. Wang, S. Li, G. Sun, X. Wang, Y. Chen, H. Li, J. Cong, N. Xiao, and T. Zhang. 2018. RC-NVM: Enabling Symmetric Row and Column Memory Accesses for In-Memory Databases. In *HPCA*.
[148] Z. Wang, P. Li, C.-J. M. Liang, F. Wu, and F. Y. Yan. 2024. Autothrottle: A Practical Bi-Level Approach to Resource Management for SLO-Targeted Microservices. In *NSDI*.
[149] Web Data Commons. 2014. Hyperlink Graphs. http://webdatacommons.org/hyperlinkgraph/.
[150] Western Digital. 2021. *WD Blue™ SN550 NVMe™ SSD*. https://www.westerndigital.com/en-ae/products/internal-drives/wd-blue-sn550-nvme-ssd?sku=WDS100T2B0C.
[151] WHATWG. 2024. URL. https://url.spec.whatwg.org/.
[152] L Woods, Z. István, and G. Alonso. 2014. Ibex—An Intelligent Storage Engine With Support for Advanced SQL Offloading. *Proc. VLDB Endow.* (Jul. 2014).
[153] K. Wu, A. Arpaci-Dusseau, R. Arpaci-Dusseau, R. Sen, and K. Park. 2019. Exploiting Intel Optane SSD for Microsoft SQL Server. In *DaMoN*.
[154] S. Xu, T. Bourgeat, T. Huang, H. Kim, S. Lee, and Arvind. 2020. AQUOMAN: An Analytic-Query Offloading Machine. In *MICRO*.
[155] S. Xu, S. Lee, S.-W. Jun, M. Liu, J. Hicks, and Arvind. 2016. BlueCache: A Scalable Distributed Flash-Based Key-Value Store. *Proc. VLDB Endow.* (Nov. 2016).
[156] W. Xu, T. Zhang, and Y. Chen. 2010. Design of Spin-Torque Transfer Magnetoresistive RAM and CAM/TCAM With High Sensing and Search Speed. *TVLSI* (Jan. 2010).
[157] H. Yang, P. Huang, R. Han, X. Liu, and J. Kang. 2023. An Ultra-High-Density and Energy-Efficient Content Addressable Memory Design Based on 3D-NAND Flash. *Sci. China Inf. Sci.* (2023).
[158] H. Z. Yang, P. Huang, R. Z. Han, Y. C. Xiang, Y. Feng, B. Gao, J. Z. Chen, L. F. Liu, X. Y. Liu, and J. F. Kang. 2020. A Novel High-Density and Low-Power Ternary Content Addressable Memory Design Based on 3D NAND Flash. In *SNW*.
[159] J. Yang and J. Leskovec. 2011. Patterns of Temporal Variation in Online Media. In *WSDM*.
[160] J. Yang and J. Leskovec. 2012. Defining and Evaluating Network Communities Based on Ground-Truth. In *KDD*.
[161] M.-C. Yang, Y.-H. Chang, C.-W. Tsao, and C.-Y. Liu. 2016. Utilization-Aware Self-Tuning Design for TLC Flash Storage Devices. *TVLSI* (Oct. 2016).
[162] X. Yin, K. Ni, D. Reis, S. Datta, M. Niemier, and X. S. Hu. 2019. An Ultra-Dense 2FeFET TCAM Design Based on a Multi-Domain FeFET Model. *TCAS-II* (Sep. 2019).
[163] X. Yu, G. Bezerra, A. Pavlo, S. Devadas, and M. Stonebraker. 2014. Staring Into the Abyss: An Evaluation of Concurrency Control With One Thousand Cores. *Proc. VLDB Endow.* (Nov. 2014).
[164] Y. Zha and J. Li. 2020. Hyper-AP: Enhancing Associative Processing Through a Full-Stack Optimization. In *ISCA*.
[165] H. Zhang, D. G. Andersen, A. Pavlo, M. Kaminsky, L. Ma, and R. Shen. 2016. Reducing the Storage Overhead of Main-Memory OLTP Databases With Hybrid Indexes. In *SIGMOD*.
[166] K. Zhang, R. Chen, and H. Chen. 2015. NUMA-Aware Graph-Structured Analytics. In *PPoPP*.
[167] D. Zheng, D. Mhembere, R. Burns, J. Vogelstein, C. E. Priebe, and A. S. Szalay. 2015. FlashGraph: Processing Billion-Node Graphs on an Array of Commodity SSDs. In *FAST*.
[168] X. Zhu, W. Han, and W. Chen. 2015. GridGraph: Large-Scale Graph Processing on a Single Machine Using 2-Level Hierarchical Partitioning. In *USENIX ATC*.