Permuting Data on Random-Access Block Storage*

Risi Thonangi
Duke University
rvt@cs.duke.edu

Jun Yang
Duke University
junyang@cs.duke.edu

ABSTRACT

Permutation is a fundamental operator for array data, with applications in, for example, changing matrix layouts and reorganizing data cubes. We consider the problem of permuting large quantities of data stored on secondary storage that supports fast random block accesses, such as solid state drives and distributed key-value stores. Faster random accesses open up interesting new opportunities for permutation. While external merge sort has often been used for permutation, it is an overkill that fails to exploit the property of permutation fully and carries unnecessary overhead in storing and comparing keys. We propose faster algorithms with lower memory requirements for a large, useful class of permutations. We also tackle practical challenges that traditional permutation algorithms have not dealt with, such as exploiting random block accesses more aggressively, considering the cost asymmetry between reads and writes, and handling arbitrary data dimension sizes (as opposed to perfect powers often assumed by previous work). As a result, our algorithms are faster and more broadly applicable.

1 Introduction

Permutation is a powerful primitive. In contrast to sorting, where the output address of a record depends on the results of comparing it with other records, here we are given a function that directly returns the output address of a record given its input address, without comparing its contents with other records. Below are some examples where we use permutation to reorganize data in useful ways.

Example 1. *Suppose we have a large $p \times q$ dense matrix stored in a file in the row-major layout. To transpose it (or convert it into column-major), we use a permutation that maps input address (i, j) to output address (j, i). In terms of linearized addresses within the file (assuming indices start from 0), this permutation moves the record at input address $i \times q + j$ to output address $j \times p + i$.*

We can also convert the matrix into a blocked *layout, where the matrix is stored as a grid of submatrices, say, in row-major order, and entries within each submatrix are stored, say, also in row-major order. This layout is popular because it provides good locality of access for a wide range of matrix operations. For simplicity, suppose the matrix is $2^4 \times 2^2$. We can write an input address in the original row-major layout as a bit string $x_5x_4x_3x_2x_1x_0$, where $x_5x_4x_3x_2$ represents the row index in binary and x_1x_0 represents the column index in binary. A permutation mapping $x_5x_4x_3x_2x_1x_0$ to output address $x_5x_1x_4x_3x_2x_0$ converts the matrix into a blocked layout with a 2×2 grid of $2^3 \times 2$ submatrices.*

Example 2. *Suppose we have precomputed an aggregate for a sales data warehouse similar to TPC-H [12], where the group-by attributes include dimensions such as* order-date, part-key, supplier-state, customer-state, *etc., and the aggregated measure attributes include for example the total* quantity *of sales. The result may have been computed and stored in* (order-date, part-key, supplier-state, customer-state, ...) *order. Now we want to "resort" it in a different order* (supplier-state, part-key, customer-state, order-date, ...), *and use it to simultaneously compute other aggregates whose group-by attributes are prefixes of the new sort order. This type of reorganization is popular in computing multi-dimensional aggregates from data cubes (e.g., [1, 15, 8]).*

We can regard this type of reorganization as a permutation of records in an address space formed by the dimensional attributes. Data can in fact be compactly stored as an array, as is commonly done in MOLAP [15, 14], with one-to-one mappings between integer array indices and dimensional attribute values. Records do not need to store either array indices or dimensional attribute values, as they are implied by the record addresses.

We study the problem of permuting lots of data (that cannot all fit in available memory) on secondary, block-based storage with fast random block accesses. Traditional hard disks are block-based, but random accesses are significantly slower than sequential accesses. On the other hand, modern flash-based storage devices can support random block accesses (reads and non-overwrite writes) as fast as sequential ones. Distributed key-value stores are another example designed for fast random accesses, where we can store a block as a key-value pair. Some of the hash-based key-value stores do not even provide a range scan interface. These storage solutions are quickly becoming popular alternatives to traditional hard drives for a variety of different reasons, ranging from fast speed, low energy consumption, to ease of maintenance (by being cloud-based).

For these storage solutions, it would seem natural to exploit their good support of random accesses in implementing permutation. But doing so effectively is still tricky because, as secondary storage, these storage solutions are still inevitably block-based. A simple, brute-force strategy would be to go through the output address space in order; for each output address, we would compute what input address should map to it, and simply read the record at that input address and write it out. However, this strategy generates lots of small, random *record* reads, which are expensive on block-based storage where costs are charged by blocks.

*This work is supported by the NSF award IIS-0916027 and an Innovation Research Program Award from HP Labs.

Proceedings of the VLDB Endowment, Vol. 6, No. 9

Another strategy is sorting. We would associate each record with a key whose comparison order reflects the desired output order, and simply run the external merge sort algorithm on these key-carrying records. Sorting is strikingly effective. In fact, it has been shown [13] that *general* permutation is as expensive as sorting in the I/O model of computation (for large enough data). For many types of permutations encountered in practice, however, permutation can be cheaper than sorting when conditions are right.

We show, in this paper, that we can indeed do much better than sorting (and the brute-force strategy) for a large class of useful permutations. The key intuition is that, by analyzing the permutation function, we can figure out what subset of input records to read into memory at one time—we call them *action records*. For each set of action records, we permute them in memory and write them to desired output addresses. Each set of action records is chosen carefully such that they are clustered in both input and output address spaces; hence, reads and writes are on the granularity of blocks instead of records. To illustrate, consider a simple example.

Example 3. *Consider a permutation that changes the ordering of records within each block and the ordering of blocks in the input file, but does not move any record across block boundaries. Suppose we have only one block of memory. We can choose records in each input block to be a set of action records. We read in each input block, permute its records, and write it out to the desired output block address. This strategy is "one-pass" in that it reads and writes each block only once (albeit in random order). Moreover, it remains one-pass regardless of how large the input is.*

This feat would be difficult to accomplish with external merge sort, whose number of passes depends on the input size. With some tweaking, external merge sort can be made one-pass for special cases (e.g., almost-sorted input). However, its sequential access pattern makes changing block ordering hard, and it lacks a general framework for exploiting the properties of a given permutation.

Translating this intuition into a general, practical algorithm is not as simple. Some permutations require multiple passes to implement efficiently, with careful choices of action records for each pass. The realities of "imperfect" data dimension sizes and the performance characteristics of modern random-access block storage also make it challenging to achieve optimal performance. Specifically, we make the following technical contributions:

- We handle *address-digit permutations* (*ADP*), which can be defined as functions permuting digits in a mixed-radix representation of the address space. ADP naturally captures all example permutations discussed earlier, as well as others such as bit-reversal (popular for Fast Fourier Transforms) and z-order (popular as a space-filling curve). We design efficient algorithms for performing ADP on random-access block storage, characterize the space of execution strategies, and develop techniques to search for optimal strategies that minimize cost.
- Many types of modern random-access block storage exhibit interesting asymmetries between read and write costs, and between non-overwrite and in-place write costs. For example, writes on flash-based storage tend to be more expensive than reads in terms of time, energy, as well as wear effect; in-place writes (and hence partial block writes) are even more expensive because they require first clearing a large "erasure block." Writes on distributed key-value stores, and especially in-place writes, also tend to be more expensive than reads because of replication and consistency. Our approach is fully cognizant of these performance characteristics. Our algorithms use "nice" I/O patterns that avoid partial block writes as much as possible. We build into our algorithms a mechanism called *filtered reads* to explore the trade-off between reads and writes—by performing more reads, we can reduce the number of passes and hence writes. Our optimization techniques consider this trade-off in finding execution strategies that minimize the overall cost.
- We show that our approach significantly outperforms external merge sort on several fronts. For typical block sizes, we only need a small amount of memory to enable efficient one-pass permutation of arbitrarily large input. If multiple passes are required, the number of passes depends on the inherent difficulty of the permutation, not on the input size; thus, the cost scales linearly with input size. In contrast, sorting will take more than one pass in most practical cases, and its cost is superlinear in the input size. Furthermore, sorting requires key comparisons and operates on a less efficient data representation, where each record must carry a key for comparison. For Example 2, the key includes all dimension attribute values. Our approach, however, does not require such a key to be stored, as the address of a record in the input already encodes where it would be in the output. This saving can be substantial with many dimensions, and it translates to proportionally smaller intermediate result sizes and lower cost per pass for our approach.
- While there has been some work on permuting data on secondary storage, our approach is practically more general and more efficient for random-access block storage. Previous approaches focus on hard drives, while ours achieves better performance by exploiting random block accesses more aggressively and considering the cost asymmetry between reads and writes. Furthermore, previous approaches typically assume that input, memory, and block sizes are all conveniently perfect powers of 2, and consider permutations defined using address bit permutations. For such approaches to work with an input address space consisting of d dimensions of arbitrary sizes, padding is required and may blow up the data size (and hence execution cost) by a factor exponential in d. We choose to tackle general mixed-radix address spaces. While doing so requires addressing a number of nontrivial technical challenges, the resulting solutions are more efficient and have broader practical applicability.

After covering the preliminaries in Section 2, we will begin by considering a simple case in Section 3, where addresses can be represented using a single radix, and block and memory sizes are perfect powers of this radix. We will then consider the general, mixed-radix case in Section 4. In Sections 3 and 4, starting with a very simple algorithm, we will present a series of improvements and generalizations, each building on the previous. Section 5 presents our experimental evaluation. We discuss future extensions and related work in Section 6, before concluding in Section 7.

2 Preliminaries

We consider a class of permutations which we call *address-digit permutations* (*ADP*). It is related to the so-called *BPC permutations* [4], which we further discuss in Section 6.

Consider the (0-based) address of a record in a file in the mixed-radix representation with n digits. The n-dimensional space of such addresses is specified by integer radices $\mathfrak{r} = (r_{n-1}, \ldots, r_1, r_0)$, where $r_i \geq 2$ for all $i \in [0, n)$. The lowest-order (least significant) digit is at position 0. We use $\langle x_{n-1}, \ldots, x_1, x_0 \rangle_{\mathfrak{r}}$, where $0 \leq x_i < r_i$, to denote the (linearized) record address

$$\sum_{i=0}^{n-1} \left(x_i \prod_{j<i} r_j \right) \tag{1}$$

within the file. We omit subscript $\mathfrak{r}$ when the context is clear.

An ADP is a permutation that can be defined as a permutation

of digits in the mixed-radix representation of an address. More precisely, it is defined by a permutation π of the digit positions $0, 1, \ldots, n-1$, which maps a digit at position i to position $\pi(i)$. This ADP moves the record at input address $\langle x_{n-1}, \ldots, x_1, x_0 \rangle_{\mathfrak{r}}$ to output address
$\pi\left(\langle x_{n-1}, \ldots, x_1, x_0 \rangle_{\mathfrak{r}}\right) = \langle x_{\pi^{-1}(n-1)}, \ldots, x_{\pi^{-1}(1)}, x_{\pi^{-1}(0)} \rangle_{\pi(\mathfrak{r})}$, where $\pi(\mathfrak{r}) = (r_{\pi^{-1}(n-1)}, \ldots, r_{\pi^{-1}(0)})$ denotes the resulting radices for output record addresses.[1] As an example, consider π_1 in Figure 1 of a 6-digit input address space. It maps position 0 to 3, 1 to 1, etc. Effectively, π_1 moves a record from input address $\langle x_5, x_4, x_3, x_2, x_1, x_0 \rangle$ to output address $\langle x_4, x_3, x_0, x_2, x_1, x_5 \rangle$. Figure 1 also shows how the output is further permuted by another ADP π_2. Although not shown here, the digits in general can have different radices, which permutations carry along.

Records are stored compactly in a file consisting of a sequence of *blocks*. We assume that all records have a fixed size, and no records span multiple blocks. Let B denote the block size, as measured by the number of records. We also assume that the data is *dense* in the input address space; i.e., there is a record at every address $\langle x_{n-1}, \ldots, x_0 \rangle$, where $x_i \in [0, r_i)$ for each i. The total number of records is therefore $N = \prod_{i=0}^{n-1} r_i$. For example, Figure 2 illustrates an input file with a 6-bit address space, as well as the output file obtained by applying π_1 in Figure 1 (ignore for now the illustration of algorithmic steps between them).

Note that there is no need to store record addresses explicitly for permutation because they are implied by record positions within the file. For instance, as discussed, the data cube in Example 2 need not store any dimension attribute values for its records.

We briefly discuss how to handle variable-size records and non-dense input data in Section 6.

Additional Notation Let $S \subseteq [0, n)$ denote a set of digit positions in (the mixed-radix representation of) an address. A *setting* θ_S of S assigns each position $i \in S$ a value $\theta_S(i) \in [0, r_i)$. A setting θ_S is *complete* if $S = [0, n)$. Note that a specific record address is a complete setting. A setting θ_S is *partial* if $S \subsetneq [0, n)$. When writing a partial setting θ_S, we use $\perp$ for digits whose positions are not in S. We write x^k for a sequence of k digits with the same value x. For example, $\langle 1, 1, 0, 1, \perp^2 \rangle$ denotes a partial setting that sets the four highest-order digits (positions 5 through 2) to 1, 1, 0, and 1, while leaving the two lowest-order digits unset.

We say that two settings θ_{S_1} and θ_{S_2} are *compatible* if for every position $i \in S_1 \cap S_2$, $\theta_{S_1}(i) = \theta_{S_2}(i)$.

Given a setting θ_S, let $\Omega_{S'}(\theta_S)$, where $S' \subseteq S$, denote the setting for S' that is compatible with θ_S. Think of this operator as "project." For example, $\Omega_{[2,4)}\left(\langle 1, 1, 0, 1, \perp^2 \rangle\right) = \langle \perp^2, 0, 1, \perp^2 \rangle$.

Let $\mho_{S'}(\theta_S)$, where $S \subseteq S'$, denote the set of settings for S' that are compatible with θ_S. Think of this operator as "enumerate." As a special case, $\mho_{[0,n)}(\theta_S)$ denotes the set of all record addresses (i.e., complete settings) compatible with θ_S. As another special case, $\mho_S(\langle \perp^n \rangle)$ denotes the set of all possible settings for S; we use $\mho_S$ as a shorthand for it. For example, if all radices are 2, $\mho_{[0,6)}$ returns all bit strings of length 6. Finally, we let $\mho_\varnothing = \{\langle \perp^n \rangle\}$.

Given an ADP π and a set S of digit positions in the input address space, we use $\pi(S)$ to denote the set $\{\pi(i) \mid i \in S\}$, the resulting positions of the digits in S in the output address space. Given a setting θ_S, we use $\pi(\theta_S)$ to denote the result of applying π to θ_S, which is a setting of $\pi(S)$ in the output record address that assigns each digit position $j \in \pi(S)$ the value $\theta_S(\pi^{-1}(j))$.

[1] We overload π so that in addition to as a function acting on digit positions, we can also interpret it as a function on addresses or a function on radices.

Algorithm 1: $\mathsf{ADP}^{\mathsf{basic}}_{\mathsf{pass}}(R, \pi)$: basic single-pass, single-radix ADP.

Input: R: a file of r^n records; π: an ADP where $\delta(\pi) \leq m - b$.
Output: a file containing permuted records.

$A \leftarrow [0, b) \cup F(\pi)$;
$\overline{A} \leftarrow [0, n) \setminus A$;
$\pi_{\text{mem}} \leftarrow g_2 \circ \pi \circ g_1$, where:
  $g_1 \leftarrow i \mapsto (i+1)$-th smallest element of A, for $0 \leq i < |A|$, and
  $g_2 \leftarrow i \mapsto$ number of elements in $\pi(A)$ smaller than i;
foreach $\theta_{\overline{A}} \in \mho_{\overline{A}}$ **do** *// each possible setting of* $\overline{A}$
  clear memory;
  foreach $\theta_{[b,n)} \in \mho_{[b,n)}\left(\theta_{\overline{A}}\right)$ in asc. order **do**
    read input block with id $\theta_{[b,n)}$ and append it to memory;
  permute records in memory by π_{mem};
  foreach $\vartheta_{[b,n)} \in \mho_{[b,n)}\left(\pi(\theta_{\overline{A}})\right)$ in asc. order **do**
    write the next memory block as output block with id $\vartheta_{[b,n)}$;

3 Single-Radix ADP with Perfect-Power Block and Memory Sizes

Let us start simple. Assume that all radices in an n-digit input address space are the same (r), so the total number of input records is $N = r^n$. Assume further that the block size is $B = r^b$, a perfect power of the radix; and that the size of available memory (also measured in number of records) is $M = r^m$, where $b \leq m \leq n$.

We need some new terminology to help with exposition. We call the b lowest-order digits (at positions $b-1, \ldots, 1, 0$) *in-digits* (for in-block digits); they specify the address of a record within the block storing it. We call the remaining higher-order digits (at positions $n-1, n-2, \ldots, b$) *out-digits* (to contrast with in-digits). A setting of all out-digits identifies a block within the file; we call this setting the *block id*. For example, in Figure 2, $r = 2$ and $n = 6$, so addresses are 6-bit strings; $b = 3$, so a block can be identified by the $n - b = 3$ highest-order bits (out-digits) while the remaining $b = 3$ bits (in-digits) vary among records within the block; $m = 4$, so the memory holds $2^m = 16$ records or $2^{m-b} = 2$ blocks.

Given an ADP π, we are especially interested in those out-digits in the input address space that are turned by π into in-digits in the output address space. We call these the *entering* digits of π. Let $F(\pi) = \{i \in [b, n) \mid \pi(i) < b\}$ denote their digit positions in the input address space ("F" is for "entering *F*rom positions"). Let $\delta(\pi) = |F(\pi)|$ denote the number of entering digits. For example, for π_1 in Figure 1, there is only one entering digit 5 (position within the input address space). Thus, $\delta(\pi_1) = 1$. On the other hand, if we consider the composition of two ADPs in Figure 1, $\pi_2 \circ \pi_1$ (i.e., π_1 followed by π_2), then $F(\pi_2 \circ \pi_1) = \{4, 5\}$, and $\delta(\pi_2 \circ \pi_1) = 2$.

As we shall see, given an ADP π, $\delta(\pi)$ gives a measure of how "difficult" π is. Example 3 has discussed the case when $\delta(\pi) = 0$; i.e., π keeps all out-digits out and in-digits in (though π may change the ordering among out-digits and the ordering among in-digits). In this case, we can do π in one pass with only one block of memory.

3.1 Basic Single-Radix ADP

Basic One-Pass Single-Radix ADP With more than one block of memory, we can handle a more difficult ADP π with $\delta(\pi) > 0$ in one pass. More precisely, we show how, using memory of size r^m (or, equivalently, r^{m-b} blocks), we can perform in one pass an ADP π where the number of entering digits $\delta(\pi) \leq m - b$.

On a high level, $\mathsf{ADP}^{\mathsf{basic}}_{\mathsf{pass}}$ (shown as Algorithm 1) works in iterations. Each iteration processes a set of "action records": it first reads $r^{\delta(\pi)}$ blocks containing these action records into memory (they fit because $\delta(\pi) \leq m-b$), permutes these records in memory, and writes them out as $r^{\delta(\pi)}$ output blocks.

More specifically, $\mathsf{ADP}^{\mathsf{basic}}_{\mathsf{pass}}$ selects and processes action records for each iteration as follows. It partitions the input digit positions

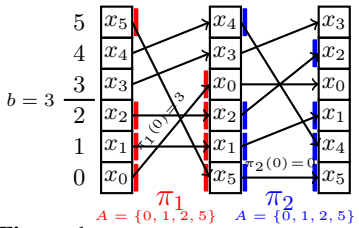


Figure 1: Illustration of ADPs π_1, π_2, and their composition $\pi_2 \circ \pi_1$, in a 6-digit address space. Action digits chosen by $\mathsf{ADP}^{\mathsf{basic}}_{\mathsf{pass}}$ for π_1 and π_2 (assuming $b = 3$) are shown—action digit positions in the input address space are marked with strips to their right; in the output address space they are marked to their left.

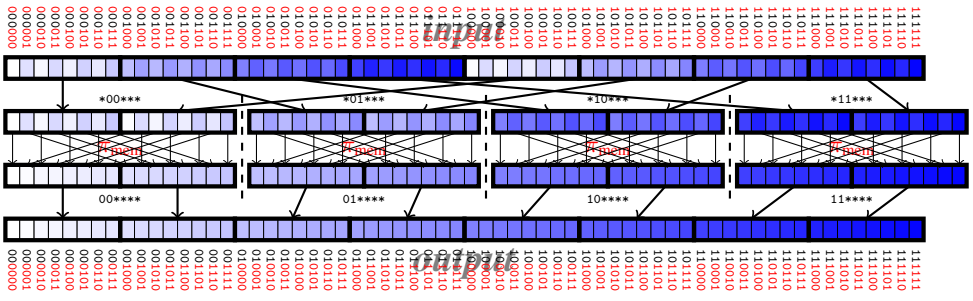


Figure 2: $\mathsf{ADP}^{\mathsf{basic}}_{\mathsf{pass}}$ performing π_1 in Figure 1; $r = 2$, $n = 6$, $b = 3$, and $m = 4$. Records are color-coded by their output ordering. Block boundaries are shown with thick lines. Action digits in input and output addresses are shown in red. There are a total of 4 iterations. Contents of the memory during these iterations are shown with the two middle rows; dashed lines separate the iterations.

into two groups:

- *Action* digits, A, which include in-block digit positions $[0, b)$ and entering digit positions $F(\pi)$. For example, Figure 1 shows the actions digits for π_1 and π_2.
- *Non-action* digits, $\overline{A}$, which include all remaining digits.

Each iteration of $\mathsf{ADP}^{\mathsf{basic}}_{\mathsf{pass}}$ considers a particular setting $\theta_{\overline{A}}$ of $\overline{A}$ (Line 4). Figure 2, which shows how $\mathsf{ADP}^{\mathsf{basic}}_{\mathsf{pass}}$ works for π_1 in Figure 1, serves as an example accompanying the description below:

- The action records are those whose input addresses are compatible with $\theta_{\overline{A}}$. There are exactly $r^{|A|}$ of them, and they occupy exactly $r^{|A|-b} = r^{\delta(\pi)}$ full input blocks. $\mathsf{ADP}^{\mathsf{basic}}_{\mathsf{pass}}$ reads all these blocks into memory (Lines 6–7).
  For example, in Figure 2, where $\overline{A} = \{3, 4\}$, the first iteration reads two blocks (memory size) of action records whose input addresses are compatible with $\langle \perp, 0, 0, \perp^3 \rangle$.
- Initially, the ordering of action records in memory is consistent with their ordering in the input file. $\mathsf{ADP}^{\mathsf{basic}}_{\mathsf{pass}}$ permutes them in memory, in place, such that their ordering becomes consistent with the desired output ordering. This in-memory permutation, π_{mem}, is itself an ADP over the memory address space. It is invariant across iterations, and is precomputed by Line 3.
  Figure 2 illustrates how this π_{mem} is applied to action records in memory consistently across iterations.
- Now, we go through memory blocks in order, and write them to appropriate locations in the output file (Lines 9–10). The output blocks have ids that compatible with the setting for $\overline{A}$ (or more precisely, $\pi(\theta_{\overline{A}})$)) in the current iteration of the outer loop.
  For example, in the first iteration shown in Figure 2, $\theta_{\overline{A}} = \langle \perp, 0, 0, \perp^3 \rangle$, so $\pi(\theta_{\overline{A}})) = \langle 0, 0, \perp, \perp^3 \rangle$, which gives us the two output blocks whose ids' two highest-order bits are 0s.

There are a total of $r^{|\overline{A}|}$ iterations. For example, in Figure 2, it takes four iterations to apply π_1.

Overall, $\mathsf{ADP}^{\mathsf{basic}}_{\mathsf{pass}}$ reads the entire input once in a specific order, and writes the output in a specific order. Thus, the I/O cost of $\mathsf{ADP}^{\mathsf{basic}}_{\mathsf{pass}}$, if we simply count the total number of block accesses, is exactly $2N/B$. Intuitively, $\mathsf{ADP}^{\mathsf{basic}}_{\mathsf{pass}}$ permutes digits within A (which includes in-digits in either input or output address space) in memory in each of its iterations; it permutes out-digits simply using random block I/Os.

Basic Multi-Pass Single-Radix ADP $\mathsf{ADP}^{\mathsf{basic}}_{\mathsf{pass}}$ allows up to $m - b$ out-digits in the input address to move to in-digit positions in the output address. Given an ADP π, if $\delta(\pi) > m - b$, we cannot complete π using a single invocation of $\mathsf{ADP}^{\mathsf{basic}}_{\mathsf{pass}}$. In this case, we adopt a simple multi-pass strategy: we use $\lceil \frac{\delta(\pi)}{m-b} \rceil$ passes, each of which calls $\mathsf{ADP}^{\mathsf{basic}}_{\mathsf{pass}}$ to move up to $m - b$ out-digits to in-digit positions. As a shorthand, we say that a pass *admits* an out-digit in the input address if the pass moves this digit to an in-digit position in the output address.

For example, consider the ADP $\pi = \pi_2 \circ \pi_1$ in Figure 1. Suppose $m = 4$ and $b = 3$. Since $m - b < \delta(\pi) = 2$ (both 4 and 5 are entering input digit positions), we cannot perform π in one pass of $\mathsf{ADP}^{\mathsf{basic}}_{\mathsf{pass}}$. Instead, we would use two passes, as illustrated by Figure 1: the first pass performs π_1 (to admit x_5), and the second pass perform π_2 (to admit x_4).

Discussion The basic approach outlined in this subsection already offers compelling advantages over sorting. The cost of multi-pass ADP, measured by the total number of block I/Os, is $\lceil \frac{\delta(\pi)}{m-b} \rceil \cdot \frac{2N}{B}$. The number of passes, $\lceil \frac{\delta(\pi)}{m-b} \rceil$, is independent of the input size. Note that $\delta(\pi) \leq b$ because the number of entering digits cannot exceed the number of in-digits. Therefore, as long as $m \geq 2b$, we are guaranteed a one-pass solution. For example, assuming a typical block size of 8kB and a fairly small record size of 4B, we need only $(\frac{8\text{kB}}{4\text{B}})^2 \cdot 4\text{B} = 16\text{MB}$ of memory to guarantee a one-pass solution for arbitrarily large input and arbitrary ADP. Bigger records would lower the memory requirement further. In contrast, one-pass sorting is generally impossible once the input exceeds memory size, and the cost of multi-pass sorting is superlinear in the input size.

These advantages are possible because of $\mathsf{ADP}^{\mathsf{basic}}_{\mathsf{pass}}$'s aggressive use of random block I/Os. To make our approach more practical, however, we still need to consider additional performance characteristics of modern random-access block storage, as well as arbitrary radices that do not "fit" memory and block sizes; both issues can significantly impact efficiency and applicability.

3.2 Improved Single-Radix ADP

We consider characteristics of modern random-access block storage to improve the basic approach of Section 3.1 in several ways. First, we can make I/Os "nicer"—instead of accessing random individual blocks, we strive to make larger requests spanning multiple consecutive blocks, and turn writes into appends. While they do not change the number of block I/Os in the case we consider in this section (single-radix ADP with perfect-power block and memory sizes), nicer I/Os can be more efficient if per-request overhead is high. Furthermore, they will become crucial for bounding the number of block I/Os in the general case we consider in Section 4, where I/O requests are no longer aligned with block boundaries.

The second idea, already mentioned in Section 1, is the use of "filtered reads" to account for the cost asymmetry between reads and writes. Intuitively, at the expense of reading some unnecessary data, filtered reads let us admit more digits in each pass, thereby decreasing the required number of passes (and hence writes).

In this subsection we show how to realize these improvements. We first present, in Section 3.2.1, an improved one-pass algorithm, which follows a given "plan" specifying what and how to permute.

The plan choice affects performance (e.g., admitting more digits beyond $m - b$ will incur higher read costs). Thus, in Section 3.2.2, we discuss how to choose the optimal strategy for performing a given ADP, which may involve multiple passes and choosing a plan for each pass to minimize overall cost.

3.2.1 Improved One-Pass Single-Radix ADP

$\mathsf{ADP}^{\mathsf{sr}}_{\mathsf{pass}}$ (shown as Algorithm 2) is the improved version of $\mathsf{ADP}^{\mathsf{basic}}_{\mathsf{pass}}$. We describe the improvements in terms of the ideas outlined at the beginning of this subsection; the corresponding modifications to $\mathsf{ADP}^{\mathsf{basic}}_{\mathsf{pass}}$ are color-coded in Algorithm 2.

Before starting, we note a high-level change: $\mathsf{ADP}^{\mathsf{sr}}_{\mathsf{pass}}$ takes two extra input arguments (A, β), which define the plan to be followed in execution. A is the set of action digits, while β specifies the size of input *segments*, which we will explain shortly. We will discuss how to choose (A, β) intelligently in Section 3.2.2.

Making Reads Bigger $\mathsf{ADP}^{\mathsf{basic}}_{\mathsf{pass}}$ reads individual blocks. With careful choices of action records, however, we may be able to make fewer, bigger read requests in the unit of *input segments*. In general, an input segment contains all records that share a common input address prefix (i.e., higher-order digit values) and, therefore, lie consecutively in the input file. Plan parameter $\beta \geq b$ specifies the size of each input segment to be r^β. The r^β records in a segment lie in $r^{\beta-b}$ consecutive input blocks. Line 9 reads each input segment.

Making Writes Bigger We can do the same for writes. Instead of writing individual blocks, we can write one *output flush* at a time. Analogous to an input segment, an output flush contains all output records that share the same output address prefix and, therefore, lie consecutively in the output file. Given an ADP π and the set of action digits A, we determine the output flush size as follows:

$$\phi(\pi, A) = \max\{j \mid [0, j) \subseteq \pi(A)\}. \quad (2)$$

In other words, we look for the longest output address suffix (i.e., lowest-order digits) consisting entirely of action digits. Let $\varphi = \phi(\pi, A)$ denote the thus computed *flush (digit) position* (Line 1). The size of each output flush is r^φ, and all r^φ in this flush lie in $r^{\varphi-b}$ consecutive output blocks. Line 14 writes each output flush.

For example, consider π_1 in Figure 1. Examining the destination positions of action digits in the output of π_1, we see that the longest suffix containing only action digits has length 4, so $\varphi = 4$. Thus, we can write $2^{\varphi-b} = 2$ blocks (which happen to be all of memory) as a single output flush. Figure 2 clearly illustrates this possibility.

Why do we compute φ by Eq. (2)? Given π and A, we want output flushes to be as large as possible. Recall that all action records in memory at the same time share the same setting for non-action digits, but a flush contains records whose output addresses have different settings for digit positions in $[0, \varphi)$. Therefore, the range $[0, \varphi)$ cannot contain any non-action digits; Eq. (2) computes the largest possible φ under this constraint.

Turning Writes into Appends In addition to making writes bigger, we can in fact ensure that the output file can be written as a number of *partitions*, such that we only need to *append* flushes to these partitions. To reduce the number of partitions that need to be kept open simultaneously for appends, we produce partitions in *partition groups*. With each memory full of action records, we append a flush to each of the partitions in the same partition group. After completing a group of partitions in this round-robin fashion, we move on to the next group. Bigger partitions and fewer partitions per group are nicer.

Like a flush, a partition consists of all records sharing a common output address prefix. Given π and A, we determine the output partition size as follows:

$$\rho(\pi, A) = \max\{j \leq n \mid [\phi(\pi, A), j) \cap \pi(A) = \varnothing\}. \quad (3)$$

Intuitively, in the output address space, starting from the flush position φ and heading toward higher-order positions, we skip over any non-action digits and stop at the first action digit we encounter (or n, if no action digit exists in $[\varphi, n)$). Let $\varrho = \rho(\pi, A)$ denote the thus computed *partition (digit) position* (Line 1). The size of each partition is r^ϱ, and there are a total of $r^{n-\varrho}$ partitions. We will justify Eq. (3) shortly.

For example, consider π_1 in Figure 1. Examining the destination positions of action digits in the output of π_1, we see that no action digit exists in $[\varphi, n) = [4, 6)$, so $\varrho = 6$, meaning that the entire output file is one partition. Next, consider π_2. In the output address space of π_2, we see $\varphi = 3$. We skip position 3 as it is not an action digit, and stop at position 4, which is. Therefore, $\varrho = 4$, meaning that there are $2^{n-\varrho} = 4$ partitions of size $2^\varrho = 16$ each.

Note that we can consider a setting for the output digit positions $[\varrho, n)$ as a *partition number*, which is a 0-based sequence number that identifies a partition within the output file. Each partition contains $r^{\varrho-\varphi}$ flushes. We can consider a setting for the output digit positions $[\varphi, \varrho)$ as a *flush number*, which a 0-based sequence number that identifies a flush within a partition.

To implement group-at-a-time, append-only flushing, we need to pick an appropriate order for bringing in sets of action records. The outermost loop of $\mathsf{ADP}^{\mathsf{basic}}_{\mathsf{pass}}$ simply steps through all possible settings for $\overline{A}$, leaving the ordering unspecified. In $\mathsf{ADP}^{\mathsf{sr}}_{\mathsf{pass}}$, we break this loop down into a nest of two. Note that we can divide non-action digits (more precisely, their positions in the output address space) into two sets: non-action partition number digits, i.e., $\pi(\overline{A}) \cap [\varrho, n)$; and the remaining non-action digits, which are exactly the flush number digits $[\varphi, \varrho)$ (since no digit in $\overline{A}$ enters $[0, \varphi)$ by the definition of φ).

- The outer loop of $\mathsf{ADP}^{\mathsf{sr}}_{\mathsf{pass}}$ (Line 4) iterates over the partition groups by enumerating every possible setting $\vartheta_{\pi(\overline{A})\cap[\varrho,n)}$ for non-action partition number digits. Each group thus consists of partitions whose numbers are compatible with this setting. This definition is natural, because in the outer loops, we do not have settings for action digits.
- Then, for each partition group, the inner loop (Line 5) enumerates all settings for $\overline{A}$ compatible with $\vartheta_{\pi(\overline{A})\cap[\varrho,n)}$ in order, which basically involves stepping through flush numbers and extending $\vartheta_{\pi(\overline{A})\cap[\varrho,n)}$ with them to form settings for $\overline{A}$.

In every iteration of the inner loop on Line 5, the output loop (Line 13) writes $r^{|\pi(A)\setminus[0,\varphi)|}$ flushes. Each flush is effectively appended to its respective partition in the current partition group being processed. It is straightforward to confirm that the number of partitions in each group is also $r^{|\pi(A)\setminus[0,\varphi)|}$.

Having seen how the algorithm operates, we can now justify our calculation of partition size in Eq. (3). Consider the output address space. Flush number digits, i.e., $[\varphi, \varrho)$, should not come from action digits, because we need to step through flush numbers on Line 5 but cannot control settings for actions digits. Under this constraint, Eq. (3) computes the largest possible ϱ, in order to maximize the partition size.

Permuting More Using Filtered Reads Recall that $\mathsf{ADP}^{\mathsf{basic}}_{\mathsf{pass}}$ can admit only $m - b$ digits, so an ADP π with $\delta(\pi) > m - b$ cannot be performed in one pass. We now show how to make $\mathsf{ADP}^{\mathsf{sr}}_{\mathsf{pass}}$ perform π in one pass. In this pass, $\mathsf{ADP}^{\mathsf{sr}}_{\mathsf{pass}}$ writes the output exactly once, as before. However, $\mathsf{ADP}^{\mathsf{sr}}_{\mathsf{pass}}$ can use "filtered reads"—when it reads from an input segment, it may keep only a filtered subset of the records in memory as action records. The net effect is more reads, but greater flexibility that enables us to permute more.

A plan (A, β) with filtered reads is one where the range $[0, \beta)$ of input address digit positions contain some non-action digits; i.e.,

Algorithm 2: $\mathsf{ADP}^{\mathsf{sr}}_{\mathsf{pass}}(R, \pi, A, \beta)$: improved single-pass, single-radix ADP.

Input: R: a file of r^n records; π: an ADP (where $\delta(\pi) > m - b$ is possible); $A \subseteq [0, n)$: input digit positions identifying the action digits to be used by the algorithm; $\beta \in (0, n)$: an input digit position specifying the input segment size to be used by the algorithm.
Output: a file containing permuted records.

1 $\varphi \leftarrow \phi(\pi, A)$; $\varrho \leftarrow \rho(\pi, A)$; *// using Eq. (2) and (3)*
2 $\overline{A} \leftarrow [0, n) \setminus A$;
3 $\pi_{\text{mem}} \leftarrow g_2 \circ \pi \circ g_1$, where:
  $g_1 \leftarrow i \mapsto (i+1)$-th smallest element of A, for $0 \le i < |A|$, and
  $g_2 \leftarrow i \mapsto$ number of elements in $\pi(A)$ smaller than i;
4 **foreach** $\vartheta_{\pi(\overline{A})\cap[\varrho,n)} \in \mathfrak{V}_{\pi(\overline{A})\cap[\varrho,n)}$ **do** *// each partition group*
5 **foreach** $\vartheta_{\pi(\overline{A})} \in \mathfrak{V}_{\pi(\overline{A})}\left(\vartheta_{\pi(\overline{A})\cap[\varrho,n)}\right)$ in asc. order **do** *// each flush #*
6 $\theta_{\overline{A}} \leftarrow \pi^{-1}(\vartheta_{\pi(\overline{A})})$;
7 clear memory;
8 **foreach** $\theta_{[\beta,n)} \in \mathfrak{V}_{[\beta,n)}\left(\Omega_{\overline{A}\cap[\beta,n)}(\theta_{\overline{A}})\right)$ in asc. order **do**
9 $E \leftarrow$ read the segment of input records with addresses compatible with $\theta_{[\beta,n)}$;
10 **foreach** $e \in E$ with input address compatible with $\theta_{\overline{A}}$, in order, **do**
11 append e to memory;
12 permute records in memory by π_{mem};
13 **foreach** $\vartheta_{[\varphi,n)} \in \mathfrak{V}_{[\varphi,n)}\left(\vartheta_{\pi(\overline{A})}\right)$ in asc. order **do**
14 write, as one flush, the next r^{φ} records in memory to output addresses compatible with $\vartheta_{[\varphi,n)}$;

$[0, \beta) \cap A \neq \varnothing$. Let $A^+ = [0, \beta) \setminus A \subseteq \overline{A}$ denote the set of non-action digits in $[0, \beta)$. In $\mathsf{ADP}^{\mathsf{basic}}_{\mathsf{pass}}$ we had $[0, b) \subseteq A$, which can be considered as a special case where $A^+ = \varnothing$.

We just need some small changes to enable filtered reads. For each possible setting $\theta_{\overline{A}}$ of $\overline{A}$, our goal is still to read in all records with compatible addresses as action records. In the loop starting on Line 8, we still read input by segments. Note that we need to project $\theta_{\overline{A}}$ by $\Omega_{\overline{A}\cap[\beta,n)}(\cdot)$ (Line 8), because if $A^+ \neq \varnothing$, $\overline{A} \not\subseteq [\beta, n)$.

More importantly, whenever we read an input segment, we get records with all possible settings for $[0, \beta)$. But we only want a subset as action records—namely, those with addresses compatible with $\theta_{\overline{A}}$, or, more precisely, addresses with the same setting as $\theta_{\overline{A}}$ for digits in A^+. Therefore, we only keep one in every $r^{|A^+|}$ records; others are discarded. We thus call $r^{|A^+|}$ the *waste factor*. This filtering step is reflected by the loop on Line 10 (which we actually implement together with Line 9 in a streaming fashion).

Note that for each $\theta_{\overline{A}}$, $\mathsf{ADP}^{\mathsf{sr}}_{\mathsf{pass}}$ executes Line 9 $r^{|A|-\beta+|A^+|}$ times, because there are exactly $|A^+|$ digits in $[0, \beta)$ that are not in A. Thus, $\mathsf{ADP}^{\mathsf{sr}}_{\mathsf{pass}}$ performs a total of $r^{|\overline{A}|} r^{|A|-\beta+|A^+|} = r^{n-\beta+|A^+|}$ segment reads. The total number of input segments is $r^{n-\beta}$, meaning that input is read $r^{|A^+|}$ times—consistent with the waste factor.

As a concrete example, Figure 3 shows how, using filtered reads, $\mathsf{ADP}^{\mathsf{sr}}_{\mathsf{pass}}$ is able to handle $\pi_2 \circ \pi_1$ from Figure 1 in one pass, which was not possible for $\mathsf{ADP}^{\mathsf{basic}}_{\mathsf{pass}}$. Here, $A^+ = \{2\}$, so the waste factor is 2. Therefore, we end up reading each input block twice (and throwing away half of the data). However, doing so nets us a one-pass strategy that reads all records twice and writes them once, which is strictly better than a two-pass strategy that would read them twice and write them twice.

Requirement and Cost Analysis $\mathsf{ADP}^{\mathsf{sr}}_{\mathsf{pass}}$ assumes that its given plan (A, β) meets the following requirements: $|A| \le m$ (i.e., action records fit in memory); $\beta \ge b$ (i.e., input segments are no smaller than blocks); $\phi(\pi, A) \ge b$ (i.e., output flushes are no smaller than blocks). In Section 3.2.2, we ensure that we only generate plans meeting these requirements.

Because we assume both block and memory sizes to be perfect powers of the single radix, all input segments and output flushes align with block boundaries. Thus, we use a simple but effective

Algorithm 3: $\mathsf{Optimize}^{\mathsf{sr}}_{\mathsf{pass}}(\pi)$: find the best one-pass plan for a single-radix ADP, which may involve filtered reads.

Input: π: an ADP.
Output: a plan (A, β) with the lowest cost.

1 $A \leftarrow [0, b) \cup F(\pi)$; $\beta \leftarrow b$;
2 **while** $|A| < m$ **do** *// make bigger output flushes with all available memory*
3 $A \leftarrow A \cup \{\pi^{-1}(\min([0, n) \setminus \pi(A)))\}$;
4 **while** $|A| > m$ **do** *// ensure action records fit in memory*
5 $A \leftarrow A \setminus \{\pi^{-1}(\rho(\pi, A)$ **if** $\rho(\pi, A) < n$ **else** $\max(\pi(A)))\}$;
6 $\beta \leftarrow \max\{i \in [b, n) \mid [b, i) \subseteq A\}$; *// make bigger input segments*
7 **return** (A, β);

cost model counting block I/Os. The number of blocks written by $\mathsf{ADP}^{\mathsf{sr}}_{\mathsf{pass}}$ is $\frac{N}{B}$. The number of blocks read is $\frac{N}{B} r^{|A^+|}$, where $r^{|A^+|}$ is the waste factor. To account for the asymmetry between read and write costs, we model the total cost as a linear combination of numbers of blocks read and written, where parameter $\alpha \ge 0$ represents the cost of reads relative to writes. Recalling that $A^+ = [0, \beta) \setminus A$, the cost of $\mathsf{ADP}^{\mathsf{sr}}_{\mathsf{pass}}$ given plan (A, β) is

$$\text{cost}(A, \beta) = \frac{N}{B}\left(\alpha 2^{|[0,\beta)\setminus A|} + 1\right). \tag{4}$$

3.2.2 *Improved Single-Radix ADP with Optimization*

Given an ADP π as well as the amount of memory $M = r^m$, we need to a find a strategy for performing π with the lowest cost. Although we can always compete π in one pass using $\mathsf{ADP}^{\mathsf{sr}}_{\mathsf{pass}}$ with filtered reads, the best strategy may involve multiple passes instead. Before showing how to find such overall best strategies, however, we discuss how to choose the best single-pass plan.

Choosing Plan for One-Pass Single-Radix ADP Given π, we choose the best one-pass plan (A, β) using $\mathsf{Optimize}^{\mathsf{sr}}_{\mathsf{pass}}$ (Algorithm 3). It starts with a "baseline" plan with $A = [0, b) \cup F(\pi)$ and $\beta = b$, the same choice as $\mathsf{ADP}^{\mathsf{basic}}_{\mathsf{pass}}$. There are two cases.

The first case is when $\delta(\pi) \le m - b$. Here, the baseline plan already meets the memory and segment/flush size requirements without using filtered reads. It is already optimal for the cost model in Eq. (4). However, if $\delta(\pi) < m - b$, there is memory to accommodate more digits in A. We choose to add those digits that immediately follow $[0, \phi(\pi, A))$ in the output address space (Line 3). Doing so makes flushes bigger (even though this improvement is not modeled by Eq. (4)).

The second case is when $\delta(\pi) > m - b$. The baseline choice of A exceeds the memory requirement, so we must remove $\delta(\pi) - (m - b)$ digits from A. But which ones? We cannot remove any digit in $F(\pi)$ from A—that would introduce a non-action digit in positions $[0, b)$ in the output address space, making the output flush smaller than a block. Therefore, we can only remove from A input address digit positions in $[0, b)$. The removed digits become A^+, so $|A^+| = \delta(\pi) - m + b$. Which digits we remove from A does not affect the cost computed by Eq. (4). However, we prefer removing the action digit at the current partition position $\rho(\pi, A)$ in the output address space (Line 5), which makes partitions bigger by increasing $\rho(\pi, A)$. If $|A|$ is still too big when the entire input is one partition (i.e., $\rho(\pi, A) = n$), we remove the higher-order action digits in the output address space, keeping $\rho(\pi, A) = n$.

Finally, we increase β as much as possible without affecting A or A^+ (Line 6). In other words, we make input segments bigger without affecting the cost in Eq. (4).

The analysis above leads us directly to the following lemma:

Lemma 1. *Give an ADP and the available memory size, $\mathsf{Optimize}^{\mathsf{sr}}_{\mathsf{pass}}$ finds an optimal one-pass plan for $\mathsf{ADP}^{\mathsf{sr}}_{\mathsf{pass}}$ (possibly with filtered reads) under the cost model of Eq. (4). For an ADP π with $\delta(\pi) = \delta$, the cost of this optimal plan is*

$$\text{cost}(\delta) = (n - b)\left(\alpha 2^{\min\{0, \delta-(m-b)\}} + 1\right).$$

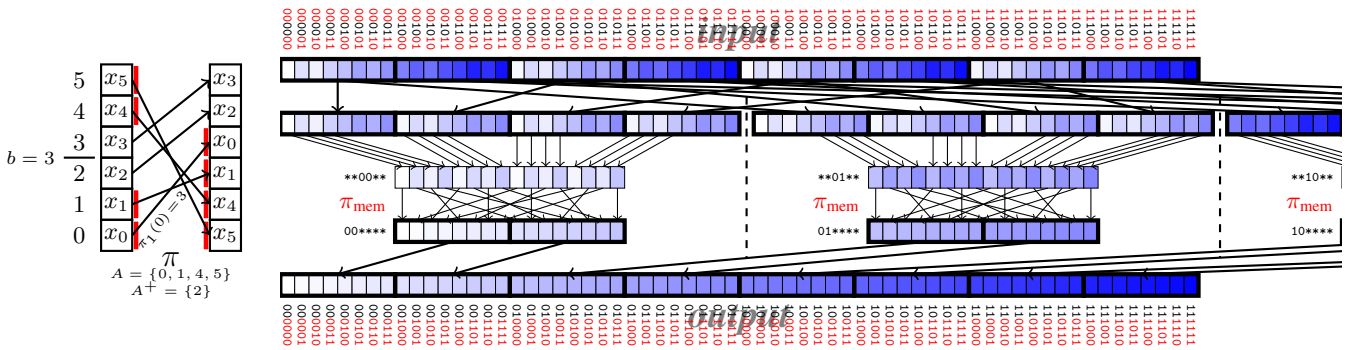


Figure 3: $\mathsf{ADP}^{\mathsf{sr}}_{\mathsf{pass}}$ performing $\pi = \pi_2 \circ \pi_1$ from Figure 1 in one pass using filtered reads; $r = 2$, $n = 6$, $b = 3$, and $m = 4$ (same as Figure 2). There are a total of 4 iterations separated by dashed lines; the last two are clipped. The second row shows the input blocks read by each iteration; keep in mind that they are simply streamed in and filtered to produce the memory contents shown on the third row.

Overall Optimization of Single-Radix ADP Given π, $\mathsf{ADP}^{\mathsf{sr}}$ (Algorithm 4) finds the optimal (possibly multi-pass) strategy and executes it by invoking $\mathsf{ADP}^{\mathsf{sr}}_{\mathsf{pass}}$ with an appropriate plan for each pass. To optimize, we first establish an upper bound on the optimal number of passes to search through. Then, for each particular number of passes, we consider only the strategy that balances the numbers of digits admitted in each pass as much as possible.

For example, suppose $b = 16$, $m = 19$, and $\delta(\pi) = 14$. Without filtered reads, each pass can admit $m - b = 3$ digits. We will need at most 5 passes. Among the four-pass strategies, $\mathsf{ADP}^{\mathsf{sr}}$ will consider the one that admits $(4, 4, 3, 3)$ digits in each of the four passes. Less balanced strategies, e.g., $(3, 3, 3, 5)$ and $(4, 4, 4, 2)$, will not be considered.

The following theorem establishes that this optimization method ensures the optimality of $\mathsf{ADP}^{\mathsf{sr}}$ under reasonable assumptions about the cost function. Indeed, the function $\text{cost}(\delta)$ in Lemma 1 meets these assumptions. (Because of space constraints, we leave all proofs in the technical report version of this paper [11].)

Theorem 1. *Suppose* $\text{cost}(\delta)$ *is the cost to perform an ADP* π *with* $\delta(\pi) = \delta$ *in one pass. If* $\text{cost}(\delta)$ *is convex in* δ *and flat for* $\delta \in [1, m - b]$*, then* $\mathsf{ADP}^{\mathsf{sr}}$ *finds the optimal number of passes and the optimal number of digits to admit in each of the passes.*

Strategies considered by $\mathsf{ADP}^{\mathsf{sr}}$ are strict improvement and superset of the those considered by our basic approach in Section 3.1. Therefore, we inherit all advantages of the basic approach over sorting discussed at the end of Section 3.1: having a number of passes independent of input size and a running time linear in it, as well as being able to do one-pass permutation with very small memory requirement. $\mathsf{ADP}^{\mathsf{sr}}$ also goes further: it has nicer I/O patterns, and it can exploit the cost asymmetry between reads and writes with filtered reads. Interestingly, if this asymmetry is so large (i.e., $\alpha \to 0$) that reads are practically free, $\mathsf{ADP}^{\mathsf{sr}}$ will choose to complete any ADP in one pass, using essentially an improved version of the brute-force strategy discussed in Section 1.

Algorithm 4: $\mathsf{ADP}^{\mathsf{sr}}(R, \pi)$: improved multi-pass, single-radix ADP.

Input: R: a file of r^n records; π: an ADP.
Output: a file containing permuted records.

$\Delta_{\text{opt}} \leftarrow \bot$; $C_{\text{opt}} \leftarrow \infty$;
for $q = \lceil \frac{\delta(\pi)}{m-b} \rceil$ to 1 step -1 **do** *// search for the optimal number of passes*
  // compute the list Δ *of the number of out-digits to move*
  // to in-digit positions in each of the q *passes:*
  $\Delta \leftarrow \varnothing$; $C \leftarrow 0$; $q_1 \leftarrow \delta(\pi) \bmod q$;
  for $p = 1$ to q_1 **do**
    $\delta_p \leftarrow (\delta(\pi) - q_1)/q + 1$; Δ.append(δ_p); $C \leftarrow C + \text{cost}(\delta_p)$;
  for $p = q_1 + 1$ to q **do**
    $\delta_p \leftarrow (\delta(\pi) - q_1)/q$; Δ.append(δ_p); $C \leftarrow C + \text{cost}(\delta_p)$;
  if $C \le C_{\text{opt}}$ **then**
    $\Delta_{\text{opt}} \leftarrow \Delta$; $C_{\text{opt}} \leftarrow C$;
for $p = 1$ to $|\Delta_{\text{opt}}|$ **do** *// in each pass, move* δ_p *out-digits to in-digit positions*
  $\delta_p \leftarrow$ the p-th element of Δ_{opt};
  $D_{\text{enter}} \leftarrow \pi^{-1}\big($the δ_p smallest elements of $\pi(F(\pi))\big)$;
  $D_{\text{exit}} \leftarrow \pi^{-1}\big($the δ_p smallest elements of $T(\pi))\big)$;
  $L \leftarrow [0, b) \cup D_{\text{enter}} \setminus D_{\text{exit}}$ ordered by $i \mapsto \pi(i)$;
  L.append$([b, n) \setminus D_{\text{exit}} \cup D_{\text{enter}}$ ordered by $i \mapsto \pi(i))$;
  $\pi_p \leftarrow i \mapsto$ (0-based) position of i in L, for $0 \le i < n$;
  $(A, \beta) \leftarrow \mathsf{Optimize}^{\mathsf{sr}}_{\mathsf{pass}}(\pi_p)$;
  $R \leftarrow \mathsf{ADP}^{\mathsf{sr}}_{\mathsf{pass}}(R, \pi_p, A, \beta)$;
  $\pi \leftarrow \pi_p^{-1} \circ \pi$; *// compute the "remaining" permutation*
return R;

4 General ADP

We now consider the general case where the input address space has mixed radices $\mathfrak{r} = (r_{n-1}, \ldots, r_0)$, and block size B and memory size M may not be perfect powers of a single radix. Recall from Section 2 that we assume data are dense in the original address space, so the number of input records is $N = \prod_{i=0}^{n-1} r_i$. However, as we will soon see, it is often necessary to "decompose" the original address space in order to perform a permutation, and data may become no longer dense in an intermediate address space—i.e., some seemingly valid addresses may not have records associated with them. One of the challenges we tackle in this section is how to keep records compactly stored in files without physically padding these conceptual holes in the address space.

We first present, in Section 4.1, a general one-pass algorithm, which builds on $\mathsf{ADP}^{\mathsf{sr}}_{\mathsf{pass}}$. We then motivate the need for address space decomposition and present our solution in Section 4.2. Finally, in Section 4.3, we discuss how to optimize a general ADP, which may require a multi-pass strategy.

4.1 One-Pass ADP

$\mathsf{ADP}_{\mathsf{pass}}$ is our general one-pass ADP algorithm. It uses the same ideas and follows the same flow as $\mathsf{ADP}^{\mathsf{sr}}_{\mathsf{pass}}$ (Section 3.2.1). We process one memory full of action records at a time: they are read in from input segments, filtered if needed, permuted in memory, and then written out as output flushes. We go through the sets of action records in a way such that output is produced in groups of partitions: for each group, we append flushes to the partitions in this group in a round-robin fashion. For additional explanation and examples, see Section 3.2.1. Because of space constraints, we leave the pseudo code of $\mathsf{ADP}_{\mathsf{pass}}$ in [11]. Here, we focus on the differences from $\mathsf{ADP}^{\mathsf{sr}}_{\mathsf{pass}}$.

First, $\mathsf{ADP}_{\mathsf{pass}}$ takes two additional input arguments, $\mathfrak{T}$ and ϖ, beyond π, A, and β. $(\mathfrak{T}, \varpi)$ specifies the input address space, which in general can be a decomposed one. $(\mathfrak{T}, \varpi \circ \pi^{-1})$ speci-

fies the output address space after applying ADP π. We will explain these notations further in Section 4.2; for now, it suffices to remember what they present respectively.

To deal with the generality of mixed-radix, decomposed address spaces, we also need a few helper functions (see [11] for a complete list). The most important one, $\mathrm{maxSize}^{\mathfrak{T},\varpi}(S)$, returns the maximum number of records whose input addresses are compatible with a setting of digits not in S. Computing these functions is simple assuming dense data—e.g., $\mathrm{maxSize}^{\mathfrak{T},\varpi}(S)$ would be the product of the radices of the digits in S. In the general case, however, calculation becomes non-trivial; we will show how to do it in Section 4.2, when we discuss decomposed address spaces in depth.

The key difference between $\mathsf{ADP}_{\mathsf{pass}}$ and the less general $\mathsf{ADP}^{\mathsf{sr}}_{\mathsf{pass}}$ is that input segments and output flushes may no longer align with block boundaries. Since the underlying storage is block-based and partial block writes are especially expensive, we take some additional steps to ensure practical efficiency. To avoid partial block writes, we reserve a block of memory as an output buffer for each partition in a partition group. The number of buffer blocks (i.e., number of partitions in a group) is calculated using the helper function $\mathrm{maxSize}(\cdot)$. Each output flush first goes through the buffer block for the corresponding partition; an actual append happens whenever the buffer block fills up (or the partition is complete). These buffer blocks avoid partial block writes that would have been incurred by flushes that do not begin or end on block boundaries. These buffer blocks also allow us to use flushes that are smaller than blocks, which is more flexible than $\mathsf{ADP}^{\mathsf{sr}}_{\mathsf{pass}}$. Of course, memory required by the buffer blocks must be accounted for and balanced against memory used for action records, which we consider next and in Section 4.3.

Requirement and Cost Analysis Suppose $\mathsf{ADP}_{\mathsf{pass}}$ is invoked with a plan $(\mathfrak{T}, \varpi, \pi, A, \beta)$. Its memory consumption can be calculated below, as the sum of the number of action records (the first term) and the total size of buffer blocks (the second term):

$$\mathrm{mem}(\mathfrak{T}, \varpi, \pi, A, \beta) = \mathrm{maxSize}^{\mathfrak{T},\varpi}(A) + B \cdot \mathrm{maxSize}^{\mathfrak{T},\varpi}\big(\pi^{-1}([\rho(\pi, A), n)) \cap A\big). \quad (5)$$

$\mathsf{ADP}_{\mathsf{pass}}$ assumes that its plan meets the following requirements: $\mathrm{mem}(\mathfrak{T}, \varpi, \pi, A, \beta) \le M$ (memory requirement); $\beta \ge b$ (i.e., input segments are no smaller than blocks); $\rho(\pi, A) \ge b$ (i.e., output partitions are no smaller than blocks—a weaker requirement than $\mathsf{ADP}^{\mathsf{sr}}_{\mathsf{pass}}$, thanks to output buffer blocks). In Section 4.3, we ensure that we only generate plans meeting these requirements.

As with $\mathsf{ADP}^{\mathsf{sr}}_{\mathsf{pass}}$, we model the cost of $\mathsf{ADP}_{\mathsf{pass}}$ as follows, where C_r is the number of block reads, C_w is the number of block writes, and $\alpha \ge 0$ represents the cost of reads relative to writes:

$$\mathrm{cost}(\mathfrak{T}, \varpi, \pi, A, \beta) = C_\mathrm{r} + \alpha C_\mathrm{w}. \quad (6)$$

However, calculating C_r and C_w is significantly more involved than the less general case of $\mathsf{ADP}^{\mathsf{sr}}_{\mathsf{pass}}$.

First, $C_\mathrm{r} = (\frac{N}{B} + U_\mathrm{r}) \times \mathrm{maxSize}^{\mathfrak{T},\varpi}([0, \beta) \setminus A)$. Here, U_r denotes the number of input segment boundaries that do not coincide with block boundaries.[2] The multiplier $\mathrm{maxSize}^{\mathfrak{T},\varpi}([0, \beta) \setminus A)$ is the waste factor of filtered reads (1 if reads are not filtered). In the general case, where input segment boundaries generally do not coincide with block boundaries, we use the number of input segments, $\mathrm{maxSize}^{\mathfrak{T},\varpi}([\beta, n))$, for U_r.

Next, $C_\mathrm{w} = \frac{N}{B} + U_\mathrm{w}$. Here U_w denotes the number of output partition boundaries that do not coincide with block boundaries. In the general case, where input segment boundaries generally do not coincide with block boundaries, we use the number of output partitions, $\mathrm{maxSize}^{\mathfrak{T},\varpi}\big(\pi^{-1}([\rho(\pi, A), n))\big)$, for U_w. Recall that unaligned flush boundaries do not incur additional block writes because of output buffer blocks.

[2] Intuitively, segment and block boundaries divide the input file into parts. Each part incurs a block read, and there are a total of $\frac{N}{B} + U_\mathrm{r}$ parts.

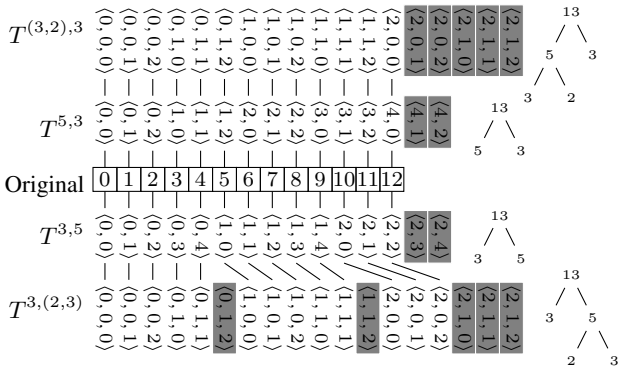


Figure 4: Different (imperfect) decompositions of the same single-digit address space with radix 13. The superscripts of T indicate radices of the decomposed address spaces. $T^{3,(2,3)}$ is obtained by further decomposing the lower-order digit of $T^{3,5}$; $T^{(3,2),3}$ is obtained by further decomposing the higher-order digit of $T^{5,3}$. Invalid addresses are shaded.

4.2 Address Decomposition

With arbitrary radices and block and memory sizes, there are now situations where we cannot pick a valid set of action digits A to perform an ADP (even with filtered reads). For example, consider transposing a big $p \times q$ matrix, where $p > M$ and $q > M$. If we model the address space using two digits (row and column numbers), including any digit in A at all would exceed the memory requirement of $\mathsf{ADP}_{\mathsf{pass}}$, because every radix is bigger than M.

Conceptually, the solution is straightforward—we simply *decompose* a digit into multiple "smaller" components such that the product of the radices of these components equals the radix of the original digit. Then, the task of permuting bigger digits can be accomplished by permuting smaller components. In the case of matrix transpose, for example, if p and q are perfect powers of 2, we can decompose them into $\log_2 p$ and $\log_2 q$ bits, respectively.

In practice, however, we might not be able to find such perfect decompositions. For example, p and q may not be perfect powers; worse, both can be prime numbers. In such cases, we have to deal with *imperfect* decompositions, where the original radix does not equal the product of component radices. Imperfect decompositions complicate address manipulations because data may not be dense in the decomposed address spaces; Eq. (1) no longer works. For example, consider imperfect decompositions in Figure 4. We see that there are (shaded) "gaps" in the decomposed address spaces. For instance, address $\langle 0, 1, 2\rangle$ in $T^{3,(2,3)}$ has no record associated with it, even though all digit values are less than their respective radices; however, the same address $\langle 0, 1, 2\rangle$ is valid in $T^{(3,2),3}$, a decomposition with the same radices as $T^{3,(2,3)}$ but nonetheless different. Eq. (1) happens to work for $T^{(3,2),3}$, but not for $T^{3,(2,3)}$.

One way to cope with imperfect decompositions is padding—we simply fill gaps in the decomposed address space with dummy records, essentially blowing up the original address space. For example, a 1025×513 matrix could be padded and handled as a 2048×1024 matrix if we want a decomposed address space consisting of bits. However, dummy records take space in files, making each pass more expensive; injecting dummy records into the original input and removing them from the final output also incur extra cost. Worse, the space overhead of padding is exponential in the number of digits—given an n-digit original address space, if all digits need to be decomposed and if, for each digit, valid addresses

make up for a fraction $f < 1$ of the address space decomposed from this digit, then only f^n fraction of the addresses in the overall decomposed address space are valid. Therefore, we need a better approach for making ADP practical in the general case.

In this subsection, we present an approach that keeps records stored compactly at all times, even if the address space is decomposed imperfectly and data become no longer dense. The notion of address decomposition requires careful formulation—as Figure 4 shows, two decomposed address spaces ($T^{(3,2),3}$ and $T^{3,(2,3)}$) can come from the same original space, have the same list of radices, but require very different address mappings. In the following, we first study what it means to decompose a single digit, and then discuss how to decompose and permute address spaces in general.

4.2.1 Decomposing a Single Digit

A *decomposition of a digit* with radix r into two digits with radices $r'' > 1$ and $r' > 1$ is *valid* if $r''r' \geq r$, $(r'' - 1)r' < r$, and $r''(r' - 1) < r$. Under this decomposition, value x (radix r) is mapped to a pair of values x'' (radix r'') and x' (radix r'), where $x'' = \lfloor x/r' \rfloor$ and $x' = x \bmod r'$.

A *recursive decomposition* of an *original digit* with radix r into a sequence of *final digits* is a node-labeled binary tree T, where:

- Each node $u \in T$ represents a digit, and is labeled by $\mathrm{rad}(u)$, its radix. The root of T, denoted $\mathrm{root}(T)$, represents the original digit. The sequence of leaves in T, denoted $\mathrm{Leaves}(T)$, represents the sequence of final digits, from the highest order to the lowest.
- A node $u \in T$ having left child u'' and right child u' represents the decomposition of the digit corresponding to u into those corresponding to u'' and u'. We call u'' (u') the *higher-order* (*lower-order*, resp.) *child* of u.

Indeed, Figure 4 shows the decompositions as trees, whose structures allow us to distinguish them.

We begin with a crucial observation characterizing the valid settings in a decomposed address space.

Lemma 2. *Consider an η-digit address space with radices $(r_{\eta-1}, \ldots, r_0)$ obtained from a single digit with radix r through decomposition T. Let $\theta_{[p,\eta)}$ denote a valid setting for digits at positions $[p, \eta)$ $(1 \leq p \leq \eta)$. The set X of valid addresses that are compatible with $\theta_{[p,\eta)}$ can be partitioned into subsets according to their values of the digit at position $p-1$. Let $X_\lambda \subseteq X$ denote the subset of addresses with value λ for digit position $p-1$. Let Z_0 denote the set of valid addresses in T compatible with $\langle 0^{\eta-p+1}, \perp^{p-1} \rangle$. Then: **1)** The set of possible values for λ is $[0, t)$ for some $t \in (0, r_{p-1}]$; i.e., there are no "holes." **2)** $X_0, X_1, \ldots, X_{t-2}$ are identical to Z_0 modulo their values for digit positions $[p-1, \eta)$. **3)** X_{t-1} is a subset of Z_0 modulo digit positions $[p-1, \eta)$; i.e., for every address in X_{t-1}, replacing its values for digit positions $[p-1, \eta)$ by 0's yields an address in Z_0.*

Lemma 2 reveals a small but important difference from the special case where the entire decomposition T is perfect (i.e., every binary decomposition in T is perfect). With a perfect T, the set of valid values for digit position $p-1$ is always $[0, r_{p-1})$; however, for general T, the set of valid values at digit position $p-1$ depends on the prefix setting $\theta_{[p,\eta)}$. For example, for $T^{3,(2,3)}$ in Figure 4, given prefix $\langle 0, 0, \perp \rangle$, the lowest-order digit can be 0, 1, or 2; however, given prefix $\langle 1, 1, \perp \rangle$, it can only be 0 or 1.

Lemma 2 prompts us to introduce a useful quantity when working with decomposed address spaces. Given a prefix setting $\theta_{[p,\eta)} = \langle x_{\eta-1}, x_{\eta-2}, \ldots, x_p, \perp^p \rangle$ for digit positions $[p, \eta)$, let $\kappa^T\big(\theta_{[p,\eta)}\big)$ denote the number of valid settings for $[0, \eta)$ compatible with $\theta_{[p,\eta)}$. For example, for $T^{3,(2,3)}$ in Figure 4, $\kappa^{T^{3,(2,3)}}(\langle 0, 0, \perp \rangle) = 3$ and $\kappa^{T^{3,(2,3)}}(\langle 1, \perp, \perp \rangle) = 5$. For $T^{(3,2),3}$, $\kappa^{T^{(3,2),3}}(\langle 1, \perp, \perp \rangle) = 6$.

We can calculate $\kappa^T(\cdot)$ in time linear in the number of digits. Because of space constraints, we leave details to [11].

Lemma 3. *Given $\theta_{[p,\eta)}$, $\kappa^T\big(\theta_{[p,\eta)}\big)$ can be computed in $O(|T|)$.*

4.2.2 Decomposing and Permuting an Address Space

A *decomposition* $\mathfrak{T}$ of an m-digit *original address space* is a sequence of node-labeled binary trees $(T_{m-1}, \ldots, T_1, T_0)$, where each T_i ($i \in [0, m)$) is a recursive decomposition of the digit at position i in the original input address. A *permuted decomposed address space* is defined by a pair $(\mathfrak{T}, \varpi)$, where $\mathfrak{T}$ specifies the decomposition from the original address space and ϖ maps a digit position $i \in [0, n)$, where $n = \sum_{i=0}^{m} |\mathrm{Leaves}(T_i)|$, to a leaf (of one of the trees) in $\mathfrak{T}$. As notational shorthands, let $\varpi^{\text{tree}}(i)$ denote the tree $T \in \mathfrak{T}$ where $\varpi(i) \in T$, and let $\varpi^{\text{lpos}}(i)$ denote number of leaves following $\varpi(i)$ in $\varpi^{\text{tree}}(i)$.

We say that a permuted decomposed address space $(\mathfrak{T}, \varpi)$ is *consistent* if ϖ preserves the ordering of leaves within each tree in $\mathfrak{T}$; i.e., for all $k > k'$ where $\varpi^{\text{tree}}(k) = \varpi^{\text{tree}}(k')$, $\varpi^{\text{lpos}}(k) > \varpi^{\text{lpos}}(k')$. We consider only consistent permuted decomposed address spaces (our optimization procedure in Section 4.3 ensures that it is the case). In the following, unless otherwise noted, all permuted decomposed address spaces are consistent.

Consider an n-digit permuted decomposed address space $(\mathfrak{T}, \varpi)$. We call a subset P of digit positions $[0, n)$ a *component-wise prefix* (or *suffix*) if for every $T \in \mathfrak{T}$, those digits in P that correspond to the leaves of T, if any, form a prefix (or suffix, resp.) of $\mathrm{Leaves}(T)$. Suppose θ_P is a component-wise prefix (or suffix) setting. Let $\Omega_T(\theta_P)$, where $T \in \mathfrak{T}$, denote a setting for the address space consisting of only $\mathrm{Leaves}(T)$ as digits, obtained by removing from θ_P all digit i where $\varpi^{\text{tree}}(i) \neq T$. Clearly, $\Omega_T(\theta_P)$ will be a prefix (or suffix, resp.) setting.

We now extend the definition of $\kappa^{\cdot}(\cdot)$ to the permuted decomposed address space $(\mathfrak{T}, \varpi)$. Suppose θ_P is a component-wise prefix setting for this address space. Let $\kappa^{\mathfrak{T},\varpi}(\theta_P)$ denote the number of valid addresses compatible with θ_P. This quantity can be readily computed in a component-wise fashion, as the product of number of possible suffix settings in the decomposition of each of the original digits (recall that these numbers can be computed by Lemma 3).

Theorem 2. *Suppose $(\mathfrak{T}, \varpi)$ is a consistent decomposed address space and θ_P is a component-wise prefix setting in this space. Then $\kappa^{\mathfrak{T},\varpi}(\theta_P) = \prod_{T \in \mathfrak{T}} \kappa^T(\Omega_T(\theta_P))$.*

We now revisit $\mathsf{ADP}_{\mathsf{pass}}$ in Section 4.1 and fill in the missing details. The input arguments $(\mathfrak{T}, \varpi)$ together specify the permuted decomposed input address space, which may be a decomposition of the original input address or an intermediate one (if $\mathsf{ADP}_{\mathsf{pass}}$ is called by a multi-pass strategy). We can compute $\mathrm{maxSize}^{\mathfrak{T},\varpi}(S)$ as $\kappa^{\mathfrak{T},\varpi}\big(\Omega_{[0,n)\setminus S}(\langle 0^n \rangle)\big)$ (for other helper functions, see [11]).

It remains to be shown how to implement the operator $\mho_{\cdot}(\cdot)$ (used throughout $\mathsf{ADP}_{\mathsf{pass}}$) efficiently in a permuted decomposed input address space. That is, given a component-wise prefix setting, we need to enumerate all valid settings for a bigger component-wise prefix in ascending order. For this enumeration, it can be shown that we can compute, for each digit, when to carry over to the next digit, as follows:

Lemma 4. *Suppose P and $P \cup \{k\}$ $(k \notin P)$ are component-wise prefixes in $(\mathfrak{T}, \varpi)$. Given a setting θ_P, the maximum value for digit k among valid addresses compatible with θ_P is $\big\lceil \kappa^T(\Omega_T(\theta_P)) \,/\, \kappa^T\big(\Omega_T\big(\langle 0^{n-k}, \perp^k \rangle\big)\big) \big\rceil$, where $T = \varpi^{tree}(k)$.*

However, it would be inefficient to pay $O(|T|)$ time to calculate $\kappa^{\cdot}(\cdot)$ for each digit and for every setting enumerated. Instead,

we offer an efficient implementation of $\mho.(\cdot)$ that incrementally maintains relevant $\kappa(\cdot)$ results during enumeration. It achieves an amortized running time of $O(1)$ per setting enumerated. Because of space constraints, we leave the details to [11].

4.3 General ADP with Optimization

Given input data in a mixed-radix address space, we now discuss how to carry out a given ADP, with block size B and memory size M. We assume the cost model in Section 4.1. In general, an optimal strategy may require multiple passes: each pass may choose to decompose its input address space, and may employ filtered reads. The original address space may undergo multiple decompositions and permutations. Because of space constraints, we will only briefly discuss our approach; see [11] for details.

We begin with the task of finding an optimal single-pass plan for $\mathsf{ADP}_{\mathsf{pass}}$. Given $(\mathfrak{T}, \varpi)$ and π, algorithm $\mathsf{Optimize}_{\mathsf{pass}}$ finds a plan $(\mathfrak{T}_\star, \varpi_\star, \pi_\star, A, \beta)$, where $(\mathfrak{T}_\star, \varpi_\star)$ represents an address space that is possibly further decomposed from $(\mathfrak{T}, \varpi)$, and $\pi_\star$ encodes the permutation equivalent to π in $(\mathfrak{T}_\star, \varpi_\star)$. On a high level, $\mathsf{Optimize}_{\mathsf{pass}}$ searches through all viable input segment sizes between B and M. Given digit position β in $(\mathfrak{T}, \varpi)$ that determines the segment size, we decompose the digit at position $\beta - 1$ in order to consider additional choices of segment sizes. It can be shown that an optimal plan includes in its action digits A all "out-digits"[3] in $[0, \beta)$ fully (i.e., with no decomposition). Thus, we consider, as a choice of A^+, every subset of in-digits or components from their decompositions. Finally, we enumerate all choices for A given β and A^+. This task turns out to be easy because the best choices can be enumerated by filling the gaps between action digits in the output address space, a complete gap at a time, from lower-order to higher-order. $\mathsf{Optimize}_{\mathsf{pass}}$ always finds an optimal plan. See [11] for the detailed algorithm and proofs.

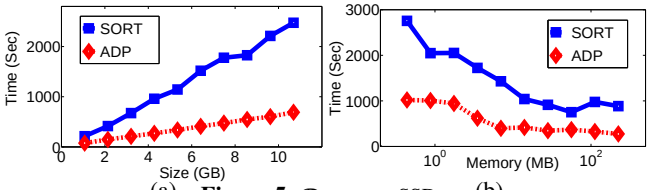
To find a multi-pass strategy, algorithm $\mathsf{Optimize}_{\mathsf{multi\text{-}pass}}$ takes a greedy approach. It chooses a plan (not involving filtered reads) for the current pass that leads to a remaining permutation with the lowest estimated "difficulty" which is measured by the memory required to complete the remaining permutation. The key to narrowing the search space for the current pass is the observation that only "in-segment" digits in the input address space and "in-flush" digits in the output address space are worth decomposing. How to ensure optimality for strategies involving multiple passes remains an interesting problem for future research. However, in most experiments we conducted, problems required one or two passes (see Section 5). Hence, to better optimize the two-pass case, we have algorithm $\mathsf{Optimize}_{\mathsf{2pass}}$ which searches through all viable input segment sizes and selects a plan based on cost.

Finally, the overall algorithm ADP (see [11]) calls the three algorithms above and chooses the least-cost plan found among them.

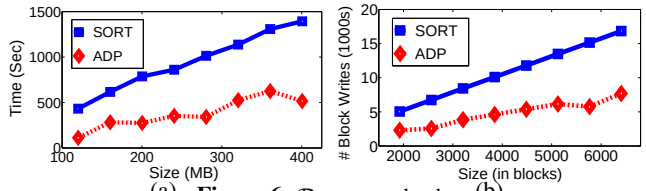
The optimization overhead of ADP is very low in practice. For all experiments of Section 5, optimization took under one second.

5 Experiments

We now evaluate ADP experimentally. Its main competitor will be a sorting-based implementation of permutation, which we refer to as SORT.[4] SORT uses the external merge sort algorithm, with the following modifications. When producing intermediate runs, SORT writes out the input address of each record as a part of this record (so that these address can be used for comparison in subsequent passes). When producing the final result, SORT does not write these addresses. During merge phases, SORT fully utilizes the available memory to read and write in units as large as possible. For in-memory sorting, SORT uses an optimized quick-sort library. We implemented both ADP and SORT in C++.



(a) **Figure 5:** $\mathcal{D}_{\text{TPC-H}}$ on SSD. (b)



(a) **Figure 6:** $\mathcal{D}_{\text{TPC-H}}$ on cloud. (b)

We conduct experiments in two hardware settings. The first one is a single workstation (Intel i7 8-core 2.8GHz CPU, 8GB RAM, Linux kernel version 3.2.0) with several storage options: an Intel X25-E 32GB SSD, an Intel 320-S 80GB SSD, and a Samsung 840 Pro 256GB SSD. To access the SSDs, we use an `ext4` file system without journaling but with direct I/O using the flag `O_DIRECT`. The block size is 4KB. Since experiments on all SSDs show similar results, we will only report results for X25-E here (see [11] for other results). In the following, "SSD" refers to this setting.[5]

The second hardware setting we consider is the "cloud." Specifically, we use an `m1.medium` machine from Amazon EC2 to permute data stored in S3. We access S3 data through the `s3ql` file system, which stores files in the unit of blocks on S3. We set the block size to 64KB.

We use primarily two datasets. The first one, $\mathcal{D}_{\text{TPC-H}}$, derived from TPC-H [12], has following base dimensions:

order-week	*part-key*	*supplier-state*	*customer-state*	*order-priority*
574	100	50	50	2

The dimension sizes are chosen to reflect reality (e.g., 574 is the number of weeks during years 1995–2005). They are not perfect powers, and they provide a good mix of small and large dimensions. When needed, we vary the dataset size by scaling *order-week* and *part-key*. The second dataset, $\mathcal{D}_{\text{v-dim}}$ is designed to study the effect of the number of dimensions: given desired input size N (in the number of records) and number of dimensions d, we select d radices (each between $\lfloor \sqrt[d]{N} \rfloor$ and $\lceil \sqrt[d]{N} \rceil$) whose product is closest to N. The record size for both $\mathcal{D}_{\text{TPC-H}}$ and $\mathcal{D}_{\text{v-dim}}$ is 8 bytes.

$\mathcal{D}_{\text{TPC-H}}$ on SSD We begin by comparing ADP and SORT with a set of experiments for $\mathcal{D}_{\text{TPC-H}}$ on SSD. Figure 5a shows the running

[3] Here, a position i is out-digit if $\text{maxSize}^{\mathfrak{T},\varpi}([0, i)) \geq B$.

[4] There are two other candidate competitors: Cormen's BPC algorithm [4], and the brute-force strategy mentioned in Section 1. As discussed in Section 6, ADP is consistently better than BPC by a factor of at least 2, so the comparison would not be interesting. The brute-force strategy incurs too many reads and is one to two orders of magnitude slower than ADP for typical problems (see [11] for details), so we do not discuss it here.

[5] Besides SSDs, we have also experimented with a traditional hard drive—a Seagate Barracuda 7200-RPM 250GB HD. While this setting is not what ADP targets, it helps illustrate, through comparison, how ADP exploits the characteristics of SSDs for efficiency. Because of ADP's random accesses, the expectation is that a pass in ADP is much slower than a pass in SORT—we indeed see this issue for some permutations. Interestingly, it turns out that for many permutations, this issue is ameliorated because ADP is able to read segments and write partitions that are much longer than a block. Overall, for the same setup in Figure 5a but with HD, ADP has comparable *average* performance as SORT (but may be better or worse for a particular permutation). For Figure 5b with HD, ADP still beats SORT when memory is limited because of fewer passes.

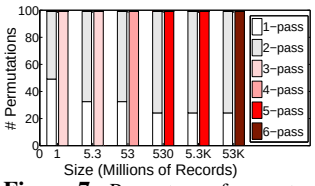


Figure 7: Percentage of permutations requiring specific numbers of passes. Darker shades mean more passes. For each input size, the left bar shows the breakdown for ADP; the right bar shows SORT.

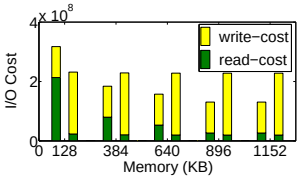


Figure 8: Best single-pass strategy with filtered reads (left bar) vs. best two-pass strategy (right bar) when memory size varies, for ADP $(2, 3, 4, 0, 6, 1, 5)$ of radices $(13, 7, 9, 35, 29, 8, 17)$.

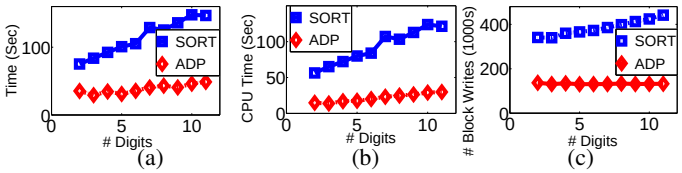


Figure 9: Effect of varying the number of dimensions d in $\mathcal{D}_{\text{v-dim}}$ on performance on SSD.

times when we vary the input size (by changing f) while keeping memory at 5% of the input size; Figure 5b varies the amount of memory while keeping the input size fixed at 4GB. Each data point for ADP shows the average running time across five random permutations (SORT is oblivious to the permutation being performed).

From Figure 5a, we see that ADP's lead over SORT improves as input size increases: for the smallest input, SORT is around 2.5 times slower than ADP, and for the largest input, it is around 3.6 times slower. Although SORT takes two passes throughout this figure, the culprit is the CPU time spent on sorting, which is still superlinear. In contrast, not only does ADP require only a single pass, it also spends no CPU time on comparisons.

From Figure 5b, we see similar advantages of ADP. For the two smallest memory sizes (460KB and 918KB), ADP uses 2 passes for four of the five randomly chosen permutations, and 1 pass for the last one, while SORT requires 4 and 3 passes respectively; on average, ADP is 2.7 and 2 times faster for these two memory sizes. For all other memory sizes tested, ADP takes only 1 pass, whereas SORT takes 2. Overall, SORT is 2 to 3.3 times slower than ADP.

$\mathcal{D}_{\text{TPC-H}}$ on Cloud We run a similar set of experiments for $\mathcal{D}_{\text{TPC-H}}$ on the cloud. Figure 6a shows the running times when we vary the input size while keeping memory at 5% of the input size. Here, the advantage of ADP is similar to what we saw for the SSD setting: for the smallest input size, SORT is around 3.8 times to 2 times slower than ADP. The running times for both algorithms is markedly slower than on the SSD setting because S3's throughput is much lower even with a block size that is 16 times larger. However, like in the SSD setting, there is little difference between random and sequential I/O for S3, so ADP shows similar gains. SORT's running time increases in an almost linear fashion as we increase the input size, whereas ADP has some zig-zags because it chose different number of passes for some permutations.

Finally, we note that running time alone may not be the correct cost metric for this setting, because Amazon S3 actually charges by the amount of data stored and requested. Therefore, to get a clearer picture, we also show, in Figure 6b, the numbers of block I/Os performed for the same set of experiments in Figure 6a. Here, we again see a clear advantage of ADP over SORT.

All Permutations on $\mathcal{D}_{\text{TPC-H}}$ As the performance of ADP depends on the given permutation, here we study the behavior of ADP for all 119 non-identity permutations for $\mathcal{D}_{\text{TPC-H}}$. Figure 7 shows the percentage of the permutations requiring a specific number of passes. We fix the memory size at 128KB and vary the input size. We see that the majority of the permutations are "easy" (requiring 1 or 2 passes) for ADP, and they do not get much harder with larger inputs. In contrast, unable to exploit easy permutations, SORT always requires a high number of passes (invariant across permutations) compared with ADP, and even more with larger inputs.

Exploiting the Read/Write Cost Asymmetry Figure 8 compares the costs of the best single-pass strategy (with filtered reads enabled) and the best two-pass strategy, as we vary available memory. Here, $\alpha = 1/16$. In this case, it turns out that the minimum memory needed for a single-pass strategy without filtered reads is around 1.5MB; thus, for all memory sizes plotted in Figure 8, single-pass strategies are only possible with filtered reads. As we decrease memory, more aggressive use of filtered reads (i.e., with bigger waste factors) are needed to stay within a single pass. Thus, we see in Figure 8 that the read portion of the cost of the single-pass strategies continues to rise as memory gets smaller. Eventually (at the smallest memory size plotted in Figure 8), two-pass strategies become favorable again. The optimization built into ADP automatically picks the best strategy in all cases.

$\mathcal{D}_{\text{v-dim}}$ on SSD In this set of experiments, we fix the input size at 512MB and memory at 5% of the input size, and vary the number of dimensions d in $\mathcal{D}_{\text{v-dim}}$. Figure 9a compares the running times of ADP and SORT on SSD. We see that the number of dimensions affects the SORT much more than ADP. ADP is around 2.1 times faster than SORT when $d = 2$ (this setting is of particular importance as it corresponds to matrix transpose, which has many applications). As d increases, this gap widens—SORT becomes progressively slower, while ADP remains efficient. When $d = 11$, ADP is around 3.1 times faster. Overall, ADP is around 2.1 to 3.4 times faster than SORT.

To further understand where the time is spent, we show in Figure 9b the CPU portion of the running times. While ADP has more complicated address calculation, SORT spends extra time in serializing/deserializing addresses as keys and comparing them. Overall, we see SORT's CPU time rising linearly with d. We believe we can further widen this gap, since our current implementation of address calculation ADP leaves more room for improvement than the already optimized libraries used by SORT.

Another advantage of ADP is that it writes smaller intermediate results, while SORT has to write keys. This advantage is clearly visible in Figure 9c, which shows the amounts of data written by ADP and SORT. As d increases, SORT writes more because keys are larger. When $d = 11$, for example, SORT needs to perform 30% more block writes than it does when $d = 2$. Larger intermediate results can also lead to more passes for SORT, although Figure 9c does not show this behavior. On the other hand, ADP's write size remains flat because ADP stores no keys; it is much lower than SORT also because ADP always takes one pass in this experiment.

6 Discussion and Related Work

Variable-Size Records We have so far assumed fixed-size records, which simplify the translation of linearized record addresses into locations within files. There are two methods to deal with variable-size records. The first method is to pad each record to its maximum size, which is simple and efficient if most records are near the maximum size. The second method is to add a level of indexing, e.g., B-tree, to enable translation of logical record addresses to physical locations. Instead of indexing each record, however, we only need to index each segment of records. Moreover, because our writes are appends in the granularity of blocks, we can index an intermediate result file efficiently like a bulk-loading procedure, thereby avoid-

ing inefficiencies of random inserts. During optimization, if record sizes are very skewed, we may use the average record size instead of the maximum size when deciding whether a set of action records can fit in memory. During execution, our algorithms can be adapted to handle unlucky situations when a large number of especially big action records overflow the available memory, but efficiency will suffer. Therefore, sorting may be more appealing in this case.

Sparse Data Section 4.2 shows how to deal with originally dense data that have *become* sparse because of address space decompositions. But what if data are already sparse in the original input address space, e.g., sparse matrices and data cubes? Analogous to the methods for handing variable-length records, there are two methods for handling sparse data—after all, one can think of a non-existent record as a record of size 0.

Related Work ADP is inspired by a class of well-known permutations called *BPC* (*bit-permute complement*). BPC is essentially a special case of ADP[6] where all radices are 2. In his dissertation [4], Cormen showed how an asymptotically optimal algorithm for BPC in the parallel disk model. Each step of Cormen's algorithm makes two passes: one to permute blocks into an intermediate order, and one to permute data within each memory worth of consecutive blocks. In comparison, our work targets storage solutions with better support for random accesses. By exploiting random reads more aggressively, we are able to complete such a step in a single pass, effectively reducing the number of I/Os by a factor of 2. We also consider optimizations such as filtered reads to further improve cost. Although our algorithms do not change the asymptotic complexity, we consider our constant factor improvement of 2 to be a very significant one, especially when it allows more problems to be solved in a single pass instead of two. As a special case of ADP (and BPC, if dimensions are powers of 2), matrix transpose has been studied extensively in the out-of-core setting [5, 6, 4, 10, 7]. Again, the target has been traditional hard drives.

Another major difference between our work and all previous work on BPC and matrix transpose is our handling of arbitrary radices and block/memory sizes. Almost all previous work conveniently assumes they are perfect powers of 2. Handling the general case requires padding, which, as we have argued in Section 1, will incur significant overhead in high dimensions. To the best of our knowledge, the only discussion of how to handle arbitrary radices efficiently is by Cormen [4] in the context of matrix transpose. He showed how, by carefully partitioning a matrix into 4 padded submatrices whose dimensions are powers of 2, the overall overhead of padding can be reduced. It is not clear how to generalize the technique to higher dimensions, where a small overhead in any two dimensions can still be blown up exponentially. We propose and evaluate a more general and robust solution.

As discussed in Example 2, one motivation for "resorting" data is computing multi-dimensional aggregates (e.g., [1, 15, 8]). This line of work mostly focused on optimizing the sharing of computation of multiple aggregates with one reorganization of data (e.g., using sorting, hashing, or traversal of a data array [15]). In general, multiple data reorganizations are still needed to compute all aggregates for a data cube—which is where our work can help in reducing the cost of the reorganizations themselves. Interestingly, Ross and Srivastava briefly mentioned [8] an optimization where sorting by dimensions (C, A, D) can benefit if data is already sorted by (C, D), because we only need to further sort (consecutive) records with the same C value. Our work can be seen as a more general framework that systematically exploits such opportunities and other less obvious ones—such as using data sorted by (C, D) to produce data sorted by (D, C) more efficiently. Recent work on cooperative sorting [3] shares a similar motivation. The main idea there is to perform multiple sorts simultaneously in order to share work; the approach is still sorting-based and targets hard drives.

Finally, interest in *radix sort* and *radix join* continues to grow, especially on modern architectures (e.g., [9, 2]). Like our work, these algorithms operate on keys in radix-based representations. However, a fundamental difference is that for sort and join, the address or presence of an output record cannot be inferred from its key alone. Therefore, unlike our algorithms for ADP, they cannot escape the additional cost factor logarithmic in the input size. Nonetheless, recent work on efficiently implementing these algorithms may give insight on how to improve the CPU cost of ADP.

[6]BPC also allows an address bit to be complemented. The analogy in general ADP would be to allow addresses to be sorted by particular digits in descending instead of ascending order. This extension is straightforward.

7 Conclusion

In this paper, we have introduced the notion of address-digit permutation (ADP), which captures a useful class of data reorganizations. We have shown how to perform ADP efficiently on modern block-based storage solutions that support random accesses better than traditional hard drives. By exploiting the characteristics of such storage, and by removing assumptions of previous work involving permutation, we have made ADP a better alternative to sorting in many practical settings.

An obvious direction of future work is more efficient implementations of ADP, especially its in-memory permutations on modern multicore architectures. Intuitively, ADP is perfectly suited to parallelization, because every record can be processed independently of all others. However, the effect of the memory hierarchy must be carefully considered. Another interesting problem for future work is how to combine the advantages of ADP and sorting. ADP works best when data are dense or uniformly sparse, while sorting handles skewed sparsity with ease. There may be a way to get the best of both approaches by judiciously choosing one of them for each subset of data or each step in an overall algorithm.

References

[1] Agarwal, Agrawal, Deshpande, Gupta, Naughton, Ramakrishnan, and Sarawagi. On the computation of multidimensional aggregates. *VLDB* 1996.

[2] Balkesen, Teubner, Alonso, and Özsu. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. *ICDE* 2013.

[3] Cao, Bramandia, Chan, and Tan. Optimized query evaluation using cooperative sorts. *ICDE* 2010.

[4] Cormen. *Virtual Memory for Data-Parallel Computing*. PhD thesis, MIT, 1993.

[5] Eklundh. A fast computer method for matrix transposing. *IEEE Transactions on Computers*, 21(7):801–803, July 1972.

[6] Kaushik, Huang, Johnson, Johnson, and Sadayappan. Efficient transposition algorithms for large matrices. *Supercomputing* 1993.

[7] Krishnamoorthy, Baumgartner, Cociorva, Lam, and Sadayappan. On efficient out-of-core matrix transposition. Technical report, Ohio State University, 2003.

[8] Ross and Srivastava. Fast computation of sparse datacubes. *VLDB* 1997.

[9] Satish, Kim, Chhugani, Nguyen, Lee, Kim, and Dubey. Fast sort on CPUs and GPUs: A case for bandwidth oblivious SIMD sort. *SIGMOD* 2010.

[10] Suh and Prasanna. An efficient algorithm for out-of-core matrix transposition. *IEEE Transactions on Computers*, 51(4):420–438, 2002.

[11] Thonangi and Yang. Permuting data on random-access block storage. Technical report, Duke University, 2013. `http://www.cs.duke.edu/dbgroup/papers/ThonangiYang-13-permute_storage.pdf`.

[12] The TPC benchmark H, 1993. `http://www.tpc.org/tpch/`.

[13] Vitter. External memory algorithms and data structures. *ACM Computing Surveys*, 33(2):209–271, 2001.

[14] Zhao. *Performance Issues of Multi-Dimensional Data Analysis*. PhD thesis, University of Wisconsin at Madison, 1998.

[15] Zhao, Deshpande, and Naughton. An array-based algorithm for simultaneous multidimensional aggregates. *SIGMOD* 1997.