On High-Latency Bowtie Data Streaming

Gabriel Stella
Texas A&M University

Dmitri Loguinov
Texas A&M University

***Abstract*—In this paper, we consider applications that read sequential data from n input files and write the result into m output files, which encompasses many types of external-memory sorting, database join/group queries, and MapReduce computation. We call this I/O model *bowtie streaming* and develop novel algorithms for modeling its throughput, maximizing sequential run lengths, and obtaining optimal multi-pass split/merge factors under non-trivial stream-switching (i.e., seek) delay. Based on these developments, we build a platform called Tuxedo for general bowtie computation and show that it is able to perform external-memory sorting with a million times fewer attempted seeks than Hadoop and two orders of magnitude fewer than highly optimized external-memory frameworks STXXL and nsort.**

I. Introduction

Modern society relies on enormous clusters to deliver such vital services as web search, social-network analysis, and machine learning. Their datasets often exceed RAM size by orders of magnitude and require external-memory algorithms that are both scalable and efficient. This becomes especially important in an I/O model we call *bowtie streaming*, where the application concurrently interacts with multiple files on both input and output, but processes each of them *sequentially* (i.e., without seeking within the file). A prime example would be merging n sorted sequences into one output; however, bowtie streaming includes many other applications – MapReduce computation [7], [17], external-memory sorting [19], graph mining [16], [26], join/group/aggregate queries in databases, and stream analytics [8], [9], [10], [11].

With the per-byte cost of magnetic drives (HDDs) still significantly lower than that of SSDs and the number of write cycles much higher, large storage arrays of spinning disks are common today. Furthermore, emerging laser-assisted heat recording technology (HAMR/HDMR), which is predicted to deliver 100 TB in a 3.5" form factor in the next decade [25], [30], [31], [33], promises to maintain this advantage in the future [14]. Therefore, understanding performance of streaming applications in the context of HDDs has important worldwide implications. To appreciate the issues involved, consider a disk volume that achieves high sequential throughput within individual files (e.g., 10 or 20 GB/s with a 36-disk RAID), but incurs a non-negligible latency δ for switching between the streams. Because concurrent access to files on such volumes involves a *conflict*, i.e., competition for shared resources, applications that do not take into account δ and other disk parameters often deliver highly disappointing performance.

This category of methods includes many software systems for processing bulk data in external memory, such as Hadoop [7], Spark [8], Stratosphere [3], Apache Beam [5], Facebook Cassandra [27], NoSQL databases [28], and the C++ library STXXL [19]. A fundamental issue that plagues existing solutions is that they often rely on principles that stem from a long line of work in the field of external-memory algorithms [1], [10], [12], [13], [19], [36], [37], [38], *where I/O cost is decoupled from application runtime*. While this may be fine for slow computation, we are interested in high-performance frameworks that can sustain multi-GB/s rates during data streaming and whose bottleneck lies specifically in I/O.

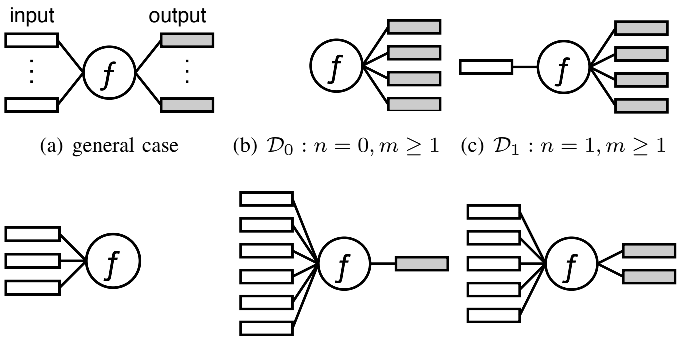
To this end, we study theory and algorithms for neutralizing the negative effect of file-switching, with the goal to create a faster stream-processing platform over high-speed arrays of HDDs. For these scenarios, we propose an accurate model of I/O-related runtime and design novel algorithms for constructing a sequence of alternating I/O operations that minimizes seeking. We also consider parameter selection for multi-pass merge/distribution, an integral part of bowtie streaming, which in the past 30 years has had multiple ad-hoc recommendations [18], [36], [38], but no solution that is provably optimal in practice. Finally, we utilize the developed techniques and models to build an external-memory bowtie-streaming system called Tuxedo that delivers over a million times fewer seeks than Hadoop/Spark [7], [8] and $200\times$ fewer than STXXL [18] in sorting tasks.

II. Bowtie Streaming

A. Definitions

Big-data computation commonly relies on streaming semantics to deal with the ever-growing scale of input [2], [4], [6], [7], [8], [9], [15], [16], [17], [22], [26]. Define a *stream* to be a data object over which the application cannot acquire random access, i.e., the only supported operations are open/close and read/write from the current device pointer. While streaming may operate over various physical devices and abstract data types, including networks and inter-thread shared queues, our examples throughout the paper focus on the storage subsystem of the host. In these cases, we use the terms *stream* and *file* interchangeably.

We consider an application with a user-supplied processing function f that merges $n \geq 0$ input streams and distributes the result to $m \geq 0$ output streams, where $n + m > 0$. To allow asymmetric I/O, define $\alpha \in [0, 1]$ to be the fraction of traffic that comes from input. Further suppose N is the number of bytes in all streams and T is the runtime of the application. Then, letting $\lambda = N/T$ be the *throughput* of the system, it follows that function f consumes from each input file at rate $\alpha\lambda/n$ and produces into each output stream at rate $(1-\alpha)\lambda/m$. To keep this scenario interesting, we assume that I/O is the main bottleneck, i.e., function f can keep up with



(a) general case (b) $\mathcal{D}_0 : n = 0, m \geq 1$ (c) $\mathcal{D}_1 : n = 1, m \geq 1$

(d) $\mathcal{M}_0 : n \geq 1, m = 0$ (e) $\mathcal{M}_1 : n \geq 1, m = 1$ (f) $n, m \geq 1$

Fig. 1. Taxonomy of bowtie streaming.

individual streams, and the amount of memory M allocated to buffering is smaller than N. We call this I/O model, illustrated in Fig. 1(a), a *bowtie*.

Bowtie streaming encompasses five individual categories, which are presented in Fig. (b)-(f) using constraints on n, m. We call the first two *distribution*, the next two *merge*, and the last one *interconnect*. In $\mathcal{D}_0$ and $\mathcal{M}_0$, the omitted half of the pipeline is either generated/consumed in RAM or delivered from/to a non-disk device (e.g., network). These cases are particularly important in cluster computing and distributed systems. Interestingly, their analysis and optimal I/O strategies differ from those of $\mathcal{D}_1$ and $\mathcal{M}_1$. Also note that an $n \times m$ interconnect can be implemented using an $\mathcal{M}_1$ merge into an intermediate object, followed by a $\mathcal{D}_1$ distribution; however, this two-stage approach generally performs worse than methods that directly use $n+m$ streams in a single stage, which explains why it belongs in a separate category.

I/O traffic of a bowtie application can be viewed as bursts of sequential operations on individual streams, interspersed by *seeks*, i.e., inter-stream jumps with some average delay $\delta(n, m) > 0$. For HDD-based file systems, this is a compound metric that includes not only such standard parameters as inter-track seek time, rotational delays, and head-settling time, but also RAID-controller read-ahead traffic, hard-drive cache logic, out-of-order command sequencing, and OS overhead. If the seek delay is non-negligible compared to the average time spent sequentially interacting with a stream, we call the bowtie *high-latency*. This is our main focus here.

B. Throughput

Assume some scheduling algorithm $\mathcal{A}$ that regulates file interleaving during bowtie streaming. To assess performance of the system, define (S_r, S_w) to be the sequential read/write speed of the streams and let (s_r, s_w) be the number of read/write stream switches executed by $\mathcal{A}$. It then follows that the runtime of the application consists of sequential scanning through αN input bytes, sequential writing of $(1-\alpha)N$ output bytes, and $(s_r + s_w)$ seeks, i.e.,

$$T(n, m) = \frac{\alpha N}{S_r} + \frac{(1-\alpha)N}{S_w} + (s_r + s_w)\delta(n, m). \qquad (1)$$

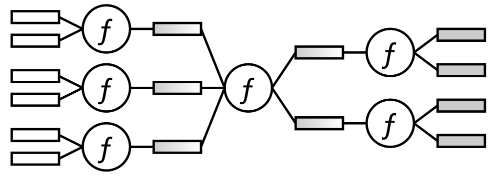


Fig. 2. Multi-pass stream bowtie.

Note that $\alpha = 0$ covers $\mathcal{D}_0$, $\alpha = 1$ produces $\mathcal{M}_0$, and the remaining cases $\alpha \in (0, 1)$ fall under the interconnect (which includes $\mathcal{M}_1, \mathcal{D}_1$). Define $L(n, m) = N/(s_r + s_w)$ to be the *average sequential run length* achieved by the bowtie scheduler. Recalling that throughput $\lambda(n, m) = N/T(n, m)$, we get

$$\lambda(n, m) = \left[\frac{\alpha}{S_r} + \frac{1-\alpha}{S_w} + \frac{\delta(n, m)}{L(n, m)}\right]^{-1}. \qquad (2)$$

A simple objective of an application may be to optimize (2), which translates into maximizing $L(n, m)$; however, the problem has additional angles. An important consequence of non-negligible seek delays and finite M is that throughput $\lambda(n, m)$ dramatically drops as the number of files increases beyond some threshold. It is therefore often possible to achieve faster rates by performing multiple passes of merge/distribution [19], [32]. This is shown schematically in Fig. 2, where a 6×4 initial problem is replaced by three binary $\mathcal{M}_1$ merges on input, a 3×2 interconnect, and two binary $\mathcal{D}_1$ distributions. The question then becomes how to select the optimal sets of merge factors $\mathbf{n} = (n_1, n_2, \ldots)$ and distribution factors $\mathbf{m} = (m_1, m_2, ...)$ to achieve the highest throughput $\lambda(n, m)$, which we study in the remainder of the paper.

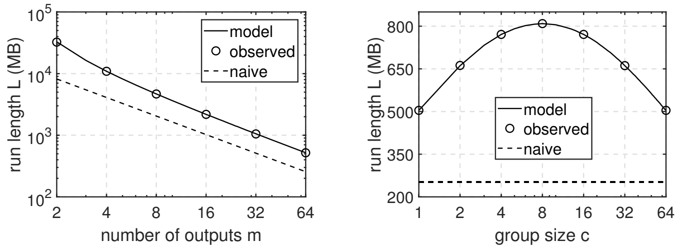
III. Distribution and Merging

We begin analysis with pure distribution and merge, following the order in Fig. 1(b)-1(e). These four scenarios not only serve as building blocks for the more complex bowties later in the paper, but also reflect special cases of interest for which having simple formulas is beneficial.

A. Distribute from Memory

For $\mathcal{D}_0$ in Fig. 1(b), a naive solution would be to never share RAM across files by providing each stream with a dedicated M/m memory. This trivially yields runs of size M/m; however, we can more than double this result. Suppose the system evolves in discrete steps $t = 1, 2, \ldots$ and the available memory M is split into m output buckets, each holding $X_i(t)$ bytes of buffered data for stream i, where $\sum_{i=0}^{m-1} X_i(t) = M$. Define $x_i(t) = E[X_i(t)]$ to be the expected size of the i-th bucket, into which the bowtie deposits new items with probability $1/m$.

When the memory is exhausted, our first Algorithm $\mathcal{A}_1$ chooses the largest bucket $W(t) = \mathrm{argmax}_i X_i(t)$ and starts writing it to disk. As the I/O progresses, blocks of memory released from bucket $W(t)$ allow additional data to arrive from function f and top off the buckets, including $W(t)$.



(a) $\mathcal{D}_0(m = 8)$

(b) $\mathcal{D}_1(m = 16)$

Fig. 3. Steady-state characteristics of distribution bowties ($M = 16$ GB).

This continues until bucket $W(t)$ becomes empty, which also implies that the memory is maxed out using the remaining $m-1$ buckets. This stops the current run for stream $W(t)$, the time is advanced to $t+1$, and the process repeats.

Let $\mathbf{y}(t)$ be a vector of expected bucket occupancies $\mathbf{x}(t) = (x_0(t), \ldots, x_{m-1}(t))$ sorted in descending order.

Theorem 1. *Algorithm $\mathcal{A}_1$ converges $\mathbf{y}(t)$ in $\mathcal{D}_0$ to a stationary state $\mathbf{y}^*$, where*

$$y_i^* = \frac{2M}{m}\Big(1 - \frac{i}{m-1}\Big), \quad i \in [0, m-1]. \tag{3}$$

Because $\mathbf{y}^*$ is an attractive fixed point of the system, its run length can be easily computed from (3).

Theorem 2. *The steady-state sequential run length of Algorithm $\mathcal{A}_1$ solving the $\mathcal{D}_0$ bowtie is*

$$L(0, m) = \frac{2M}{m-1}. \tag{4}$$

Fig. 3(a) shows that (4) matches simulations quite well. It also plots the run length of the naive allocator, which is worse than (4) by a factor of $2m/(m-1)$. This ratio begins at 4 and converges to 2 as $m \to \infty$.

B. Distribute from File

For scenario $\mathcal{D}_1$ in Fig. 1(c), new data cannot be added to output buckets without first switching to the reader, which incurs a seek that we did not have before. A naive solution is to split memory equally between input/output streams, reading $M/2$ bytes into a buffer, processing them with f into m buckets, and then saving each of the buffers to disk. This results in M bytes exchanged using $m+1$ seeks, which trivially yields $L = M/(m+1)$. However, we can again do much better.

When the memory is full, Algorithm $\mathcal{A}_2$ selects a group of $c \geq 1$ largest buckets, where m/c is an integer, writes them to disk, and then reloads enough data from the input file to replenish the departed chunk, running function f concurrently with read traffic. By overlapping input I/O and computation, the reader side of the bowtie requires only negligible buffering (e.g., two blocks). Hence the entire memory M is still shared across m output buckets. Since f is no slower than the sequential disk rate, the I/O never goes idle during execution of the system. Also note that each step t now contains $c+1$ seeks, out of which c are for writing and one is for reading.

Define $\tilde{y}_i(t)$ to be the size of the i-th largest group (i.e., sum of its bucket occupancies) at the start of step t when the memory is full.

Theorem 3. *Algorithm $\mathcal{A}_2$ converges $\tilde{\mathbf{y}}(t)$ in $\mathcal{D}_1$ to a stationary state $\tilde{\mathbf{y}}^*$, where*

$$\tilde{y}_i^* = \frac{2M}{m/c+1}\Big(1 - \frac{ic}{m}\Big), \quad i \in [0, m/c-1]. \tag{5}$$

Given this equilibrium, the average run length follows.

Theorem 4. *The steady-state sequential run length of Algorithm $\mathcal{A}_2$ solving the $\mathcal{D}_1$ bowtie is*

$$L(1, m) = \frac{2M}{(1-\alpha)(m/c+1)(c+1)}. \tag{6}$$

We can now examine the various strategies for selecting c. For this discussion, we use $\alpha = 1/2$ commonly found in sorting applications, i.e., no keys are removed from or added to the streams by f. Two obvious approaches would be to empty either one bucket (i.e., $c = 1$) or all buckets ($c = m$) when the memory is full, both of which produce $L(1, m) = 2M/(m+1)$. However, can we do better?

Theorem 5. *Algorithm $\mathcal{A}_2$ achieves optimal performance with $c = \sqrt{m}$, where the best run length is given by*

$$L(1, m) = \frac{2M}{(1-\alpha)(\sqrt{m}+1)^2}. \tag{7}$$

Compared to the more straightforward methods with $c = 1$ or $c = m$, (7) increases $L(1, m)$ by $1.36\times$ for $m = 16$, $1.6\times$ for $m = 64$, and $1.78\times$ for $m = 256$, eventually converging to 2 as $m \to \infty$. This implies that an application using the optimal c can achieve the same throughput as the obvious methods using up to $2\times$ less RAM. An example is shown in Fig. 3(b) for $\alpha = 1/2$, where simulations follow (6) precisely, the optimal $c = 8$ agrees with Theorem 5, and the naive strategy is $3.2\times$ worse than (7).

C. Merge to Memory

We now deal with merge $\mathcal{M}_0$ in Fig. 1(d), which has many similarities to $\mathcal{D}_0$; however, there are important differences as well. Suppose each step t of Algorithm $\mathcal{A}_3$ begins when some bucket j reaches the empty state, after which it must be reloaded for computation f to continue. The algorithm performs a seek to the appropriate file and proceeds to stream input data into bucket j, concurrently with f draining the items across all streams, until another bucket becomes exhausted. By this time, a large chunk of RAM has been freed, which we use to refill bucket j until memory is maxed out again. When this happens, time advances to step $t+1$ and the process repeats.

What is interesting about this system is that it is sensitive to the initial state of the buckets. A naive approach preloads them to equal size M/n; however, this causes them to reach the empty state at roughly the same time, which in turn leads to one bucket taking over the entire RAM, followed by a deadlock. Instead, suppose Algorithm $\mathcal{A}_3$ starts the i-th largest bucket at some size z_i, where $\sum_{i=0}^{n-1} z_i = M$.

Theorem 6. *The optimal initial bucket occupancy for Algorithm $\mathcal{A}_3$ solving $\mathcal{M}_0$ is*

$$z_i^* = \frac{2M}{n}\Big(1 - \frac{i}{n-1}\Big), \quad i \in [0, n-1]. \tag{8}$$

This leads to the following result.

Theorem 7. *The steady-state sequential run length of Algorithm $\mathcal{A}_3$ under the optimal initial state is*

$$L(n,0) = \frac{2M}{n-1}. \tag{9}$$

Not surprisingly, $\mathcal{M}_0$ has the same stationary vector as $\mathcal{D}_0$ (except m is replaced by n); however, the main difference is that this equilibrium does not occur naturally, i.e., the system does not converge to it unless the initial state is properly selected. This distribution of bucket sizes must also be maintained during the run so that small deviations from the ideal staircase pattern in (8) does not cause divergence to suboptimal (or even deadlock) states.

D. Merge to File

For scenario $\mathcal{M}_1$ in Fig. 1(e), consider Algorithm $\mathcal{A}_4$ that breaks n streams into groups of size c and preloads each of them to some initial size $\tilde{\mathbf{z}}$, i.e., each bucket in group i is filled to occupancy $\tilde{z}_i/c$ and $\sum_{i=0}^{n/c-1} \tilde{z}_i = M$. The algorithm then runs function f concurrently with sequential writes to output until some bucket becomes exhausted. It then reloads the c smallest buckets, filling the memory back to M and splitting the available space equally among the reloaded group.

Theorem 8. *The optimal initial state for group i in Algorithm $\mathcal{A}_4$ working on $\mathcal{M}_1$ is*

$$\tilde{z}_i^* = \frac{2M}{n/c+1}\Big(1 - \frac{ic}{n}\Big), \quad i \in [0, n/c-1]. \tag{10}$$

The next result follows from the proof of Theorem 5.

Theorem 9. *Algorithm $\mathcal{A}_4$ with an optimal preload is optimized by $c = \sqrt{n}$, which results in*

$$L(n,1) = \frac{2M}{\alpha(\sqrt{n}+1)^2}. \tag{11}$$

Our final observation in this section is that α has an asymmetric impact on (7) and (11), which actually makes sense. As $\alpha \to 0$, the majority of I/O comes from the write component, which helps increase the run length in (11) for $\mathcal{M}_1$. At the same time, small values of α have a detrimental effect on the run length in (7) for $\mathcal{D}_1$, where long sequential reads become shorter. As $\alpha \to 1$, the read/write roles are reversed and the formulas behave the opposite.

IV. Interconnect

We now deal with the interconnect scenario in Fig. 1(f). For this section, we aim to investigate bowties that operate by directly reading $n \geq 1$ input files and writing into $m \geq 1$ using a single pass over the data.

Our main observation here is that we can stitch the reader part of optimal merge algorithms with the writer part of distribution, both developed in the previous section, to solve the interconnect. Out of the four combinations, only three are valid – $(\mathcal{M}_1 + \mathcal{D}_1)$, $(\mathcal{M}_0 + \mathcal{D}_1)$, and $(\mathcal{M}_1 + \mathcal{D}_0)$, whereas $(\mathcal{M}_0 + \mathcal{D}_0)$ is impossible since it requires overlapping f with concurrent streaming on both input and output. Our next topic is to understand which of these methods is better. To avoid clutter during comparison, we fix $\alpha = 1/2$ and generalize the best method towards the end of this section.

A. Non-Overlapping

Our first approach is Algorithm $\mathcal{A}_5$, which starts by allocating n buckets for reading and m buckets for writing, where the total size of the former is M_r bytes, that of the latter is M_w, and $M_r + M_w = M$. Note that explicit separation between read/write components is necessary to support multi-file I/O on both sides of function f. On the reader side, Algorithm $\mathcal{A}_5$ plays the $\mathcal{M}_1$ game using the corresponding elements of Algorithm $\mathcal{A}_4$ and on the writer side, it runs the $\mathcal{D}_1$ game using the appropriate parts of Algorithm $\mathcal{A}_2$.

In more detail, we partition input/output buckets into groups of size c_r and c_w, respectively. Each time step t begins with some input bucket j reaching the empty state. Under ideal conditions to be determined below, this coincides with output space M_w being exhausted. Thus, the largest group of output buckets is offloaded to disk, using c_w seeks, which frees up enough room for f to continue. Then, the group to which bucket j belongs is reloaded from input, saturating M_r to maximum utilization, which requires another c_r seeks.

Unlike prior techniques developed in this paper, Algorithm $\mathcal{A}_5$ does not overlap computation with I/O. As a result, the disk will experience idle periods when f is shuffling items between input and output buckets in memory.

Theorem 10. *The steady-state sequential run length of Algorithm $\mathcal{A}_5$ under optimal memory partitioning (M_r, M_w) is*

$$L(n,m) = \frac{4M}{(c_r + c_w)(n/c_r + m/c_w + 2)}. \tag{12}$$

B. Overlapping

We now consider Algorithm $\mathcal{A}_6$ that combines either $(\mathcal{M}_0 + \mathcal{D}_1)$ or $(\mathcal{M}_1 + \mathcal{D}_0)$, depending on which one is faster for a particular pair of (n, m). It overlaps I/O and computation on one of the two sides, which not only achieves a larger run length, but also reduces the runtime by never allowing the disk to become idle. This overcomes both limitations of $\mathcal{A}_5$.

For $(\mathcal{M}_0 + \mathcal{D}_1)$, Algorithm $\mathcal{A}_{6r}$ uses $c_r = 1$ on input and $c_w \geq 1$ on output. At every time t, there is a current bucket j from which the reader is streaming data into the input of f. The remaining $n-1$ files are prebuffered in their respective buckets. Once another bucket $i \neq j$ gets empty, the following sequence takes place: a) file j continues producing data until its bucket is refilled to maximum size (i.e., M_r runs out of space), which requires no additional seeks; b) the largest group of c_w output buckets is saved to disk, which requires c_w seeks; and c) bucket i becomes the current bucket, from which we begin streaming into f in the next iteration, which requires

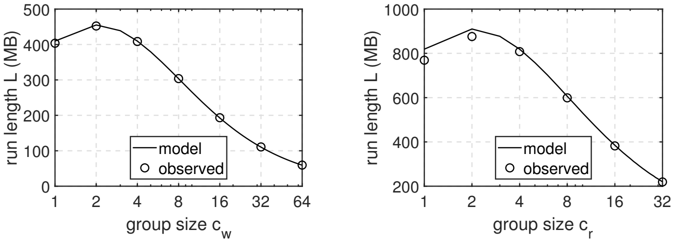


Fig. 4. Run length in $n \times m$ interconnect bowties ($M = 16$ GB).

one seek. For $(\mathcal{M}_1 + \mathcal{D}_0)$, Algorithm $\mathcal{A}_{6w}$ works similarly, except $c_r \geq 1$ and $c_w = 1$.

Theorem 11. *Both version of Algorithm $\mathcal{A}_6$ achieve*

$$L(n,m) = \frac{4M}{(c_w + c_r)(n/c_r + m/c_w)}. \quad (13)$$

Note that (13) is strictly larger than (12) for the same values of (c_r, c_w). Combining this with the fact that Algorithm $\mathcal{A}_5$ runs into disk stalls, while $\mathcal{A}_6$ does not, leads to the following.

Corollary 1. *Under their respectively optimal parameters (c_r, c_w), Algorithm $\mathcal{A}_6$ yields smaller runtimes than $\mathcal{A}_5$.*

To obtain the best run length $L(n,m)$, we next derive the optimal point of (13).

Theorem 12. *Algorithm $\mathcal{A}_{6r}$ wins over $\mathcal{A}_{6w}$ when $m \geq n$, in which case $c_w = \sqrt{m/n}$ is optimal; otherwise, $\mathcal{A}_{6w}$ wins and $c_r = \sqrt{n/m}$ is optimal. In both cases,*

$$L(n,m) = \frac{4M}{(\sqrt{n} + \sqrt{m})^2}. \quad (14)$$

For example, a 3×100 interconnect in (14) achieves a decent $L(n,m) \approx M/34$. Compared to performing a full 3×1 merge $\mathcal{M}_1$ into a file, followed by a 1×100 distribution $\mathcal{D}_1$, which runs with a similar $L(n,m) \approx M/30$, Algorithm $\mathcal{A}_6$ saves two full passes over the data. Fig. 4 shows that model (13) matches the actual run length in our system and that the optimal (c_r, c_w) are in agreement with the prediction in Theorem 12. Additionally, the figure demonstrates that incorrect selection of these parameters may yield a drastic reduction in performance, e.g., from 450 MB to 70 MB in Fig. 4(a).

C. Generalization

We now address $\alpha \neq 1/2$. Let two functions be *reader-writer symmetric* if one can be converted into the other by swapping every occurrence of n with m, α with $1 - \alpha$, and c_r with c_w.

Theorem 13. *When $m > 1 + \alpha(n-1)/(1-\alpha)$, the optimal choice is Algorithm $\mathcal{A}_{6r}$, where group size*

$$c_w = \sqrt{\frac{(1-\alpha)m}{(1-\alpha) + \alpha(n-1)}} \quad (15)$$

achieves the best run length

$$L(n,m) = \frac{2M}{(\sqrt{(1-\alpha)m} + \sqrt{(1-\alpha) + \alpha(n-1)})^2}; \quad (16)$$

otherwise, Algorithm $\mathcal{A}_{6w}$ is better, where the optimal group size c_r and the corresponding run length are reader-writer symmetric to (15) *and* (16).

When $\alpha = 1/2$, the result in (16) reduces to (14). Furthermore, Algorithm $\mathcal{A}_6$ subsumes our earlier methods for all cases with $n, m \geq 1$, i.e., its optimal run length (16) matches that of $\mathcal{A}_2$ in (7) when $n = 1$ and its reader-symmetric version achieves the same performance as $\mathcal{A}_4$ in (11) when $m = 1$. It still cannot handle $n = 0$ or $m = 0$ due to the assumption that streams exist on both sides; however, for such cases Algorithms $\mathcal{A}_1, \mathcal{A}_3$ offer good solutions to $\mathcal{D}_0, \mathcal{M}_0$ bowties.

V. Multi-Pass Optimization

We now arrive at the issue of choosing the best multi-pass factors $\mathbf{n} = (n_1, \ldots, n_r)$ and $\mathbf{m} = (m_1, \ldots, m_w)$ that guarantee peak performance. In prior literature, this problem has been considered only for $\mathcal{M}_1/\mathcal{D}_1$ and only in a limited context, i.e., all split factors were assumed to be equal to some constant k and there was no data expansion/shrinkage between the levels. Even then, the results are often conflicting. For example, Vitter in one paper [36, p. 123] uses $k = \sqrt{M/B}/\ln^2(M/B)$, where B is the I/O block size. This result produces binary merging under common conditions (e.g., $M = 2 - 16$ GB and $B = 1$ MB). The companion book [38, p. 32], however, specifies $k = \min(n, M/B)$, which results in single-pass merging for all $n \leq M/B$. An identical approach is mentioned in [12] and a recent paper [23, p. 3]. Finally, STXXL [18] computes the optimal k as $n^{1/\gamma}$, where $\gamma = \lceil \log_{M/B} n \rceil$ is the number of merge passes. This yields $k = n$ for $n \leq M/B$ and $k = \sqrt{n}$ for $n \in (M/B, (M/B)^2]$.

A. Merge and Distribution

Before attempting bowties that contain interconnects with $n, m \geq 2$, we first have to solve the multi-pass $\mathcal{M}_1, \mathcal{D}_1$ problems. Since merge and distribution are symmetric, description focuses on merge $\mathcal{M}_1$. For the multi-pass solution to make sense, the final outcome needs to be invariant to the selection of vector $\mathbf{n}$. In other words, a 4-way merge, followed by a 3-way merge, produces the same amount of data as a single 12-way merge. It is then convenient to introduce a function $p(k)$ that specifies the amount of data remaining in k files after an (n/k)-way merge.

This formulation provides the necessary properties (i.e., optimal substructure, overlapping subproblems) to allow dynamic programming in Listing 1 to optimize the runtime of a multi-way merge. Let `time[k]` keep track of the best runtime achievable by a $1 \times k$ merge bowtie and `path[k]` be the corresponding vector of merge factors $\mathbf{n}$. Then, for every $j \in [2, n]$, we iterate over all $i \in [2, j]$ to examine if the optimal solution for a $1 \times j$ merge can be constructed using a k-way merge followed by an i-way, where $k = \lceil j/i \rceil$ (Lines 3-6). During this step, the system reads $p(n/k)$ bytes and writes

Listing 1: Optimal Multi-Pass Merge Factors

```
1  Func PrepareMultipassMergeTable(n)
2      time[1] = 0; path[1] = ∅    ▷ zero cost for n = 1
3      for (j = 2; j ≤ n; j++) do
4          time[ j ] = ∞    ▷ best time initially unknown
5          for (i = 2; i ≤ j; i++) do
6              k = ⌈ j / i ⌉    ▷ k-way merge, followed by i-way
7              e = p(n/k) / S_r + p(n/j) / S_w + δ(p(n/k) + p(n/j)) / L(i,1)
8              t = time[k] + e    ▷ time for a j-way merge
9              if t < time[j] then
10                 time[j] = t; path[j] = path[k] ∪ { i }
11     return (time, path)
```

Listing 2: Optimal Multi-Pass Bowtie

```
1  Func OptimalBowtie(n, m)
2      mT = PrepareMultipassMergeTable(n)
3      dT = PrepareMultipassDistributionTable(m)
4      best = ∞    ▷ unknown runtime yet
5      for (i = 1; i ≤ n; i++) do
6          for (j = 1; j ≤ m; j++) do
7              t = p(i) / S_r + q(j) / S_w + δ(p(i) + q(j)) / L(i, j)
8              t += mT.time[n / i] + dT.time[j]    ▷ add M_1, D_1 times
9              if t < best then
10                 best = t; path = (i, j)
11     return (best, path)
```

$p(n/j)$, which leads to the runtime of the i-way merge in Line 7. Combining this with `time[k]` in Line 8 to optimally perform a k-way merge, we find the total runtime for j. If this beats the current best estimate, we update both `time[j]` and `path[j]` in Line 10.

We now show an example to illustrate performance benefits of multi-pass optimizations that run over real I/O curves of the physical device rather than hardcoding $k = 2$ or $k = n$ as suggested in prior work. Consider a 24-disk Areca RAID array that merges a 64-TB workload across $n = 8,000$ files using $M = 8$ GB. Under the optimal $\mathcal{M}_0$ run length $2M/(n-1)$, usage of $k = 2$ [36] requires 13 passes and delivers an end-to-end throughput of only $\lambda(n,m) = 273$ MB/s. The more popular choice of $k = n$ [12], [18], [38] is even worse, finishing a single pass @ 208 MB/s. Instead, our system constructs an optimal two-pass merge tree with $n_1 = 90$ and $n_2 = 89$ whose overall throughput is 1,353 MB/s. This is $5 - 6.5\times$ faster than the prevalent solutions in the literature.

For distribution $\mathcal{D}_1$, the optimization algorithm is similar, except function p is replaced by its output-equivalent q, n by m, $L(i,1)$ with $L(1,i)$, and `time[k]` is changed to represent the minimum delay to split k initial files into m.

B. Interconnect

To tackle general bowties that may include an interconnect, our system uses the algorithm in Listing 2. The first step (Lines 2-3) is to prepare optimal merge-distribution tables, which constructs one-dimensional vectors of optimal runtime using Listing 1. We then iterate over all possible (i,j), performing a n/i-way multi-pass merge from n to i files, an $i \times j$ interconnect, and an m/j-way multi-pass distribution from j files to m (Lines 5-10).

The overall CPU cost of this algorithm is $\Theta(n^2 + m^2)$, which is quite reasonable for the typical values encountered in practice (i.e., up to a few thousand files). When higher speed is desirable, the tables could be filled at exponentially increasing intervals so that a quick approximation can be obtained in $\Theta(\log^2 n + \log^2 m)$ time. Then a more detailed search can be performed in the vicinity of the discovered solution, if needed.

VI. I/O Comparison Platform

A. Methodology

We combine the algorithms and models developed so far into a high-performance bowtie streaming system we call *Tuxedo*. It provides an open-source platform [35] for performing I/O-intensive computation using the various algorithms and techniques introduced earlier, including measurement of disk parameters, accurate prediction of throughput $\lambda(n,m)$, maximization of sequential run lengths $L(n,m)$, and optimal calculation of multi-pass/interconnect factors. Tuxedo can accommodate any suitable user function f, which we illustrate by building an external-memory sorter that uses our multi-pass scenario $\mathcal{D}_1$ to split the input stream into m files, each of which fits in RAM, perform a sort on the resulting chunks using Vortex [24], and output them back to disk. Our target application performs a sort on uniform 64-bit integers.

We compare Tuxedo to several existing implementations – two tremendously popular big-data processing engines Hadoop [7] and Spark [8]; a highly optimized C++ library STXXL for external-memory data structures and algorithms [18]; and a repeated winner of the sort benchmark [34], now a commercial product, called nsort [29]. While the sort competition [34] has seen a number of impressive records, many of them utilize existing programs (i.e., nsort, Hadoop, Spark, STXXL), while others provide no publicly available executables/code, which makes comparison with them either redundant or impossible.

Since most frameworks in our test bottleneck on the CPU sort, there needs to be a systematic way to compare the quality of their I/O schedulers. By focusing on file-access patterns, rather than just the runtime, we eliminate interference from the in-memory sort/merge and from various inefficiencies in the pipeline (e.g., data movement). Ideally, the measurement platform would log all I/O calls performed by the sort and reenact them later in a standardized way. In some situations, it may be possible to manually modify the source code of each program to accomplish such recording; however, this is generally tedious and error-prone (e.g., the STXXL codebase is almost 100,000 lines of C++, Hadoop has over 3.3M). In other cases, this is simply impossible (e.g., nsort is closed-source). To overcome these issues, we develop a novel set of software tools that can record, analyze, and consistently replay I/O traces of any process in Windows.

This toolkit, explained in more detail below, includes three components: 1) *process-tracing system*, whose purpose is to intercept and record calls to file APIs; 2) *log-conversion system*, which merges, cleans, and translates the logs into a format that permits easier analysis; and 3) *replay system*, which reads converted log files, issues its I/O requests in the most efficient way, and generates a performance report.

B. Tracing

To log all I/O requests of a given process, we insert *detours* (i.e., inline hooks) into file APIs [20], [21]. This is done by modifying the assembly code of the target function to perform a jump to a user-specified monitor. The overwritten bytes in the original function are stored in a separate buffer and merged with special *trampoline* code that allows the API to be called directly (i.e., bypassing the hook). After this, all interaction of the process with files is redirected to our custom handler, which logs the relevant parameters and then passes the call to the kernel through the trampoline. On the way back, the handler records the error codes provided by the OS, discards unsuccessful I/O attempts, and adjusts I/O size to the value reported by the API. Furthermore, to avoid infinite looping when writing into the log file from the handler, the hook ignores I/O requests that come from itself.

In order for this technique to work, the handler must execute in the virtual space of the sorting framework. This task, known as *DLL injection*, is accomplished by creating a remote thread in the target process and instructing it to load our hooking DLLs. To prevent the sorter from performing I/O before detours are installed, our platform starts each measured process in a suspended state and waits for the remote thread to finish before resuming the application. To account for cases when the studied framework (e.g., Hadoop) spawns additional processes, which may perform their own I/O, our code also hooks process-creation APIs. This allows the measurement platform to automatically inject a copy of itself into each child process, no matter how many are created.

Building a robust hooking library has additional challenges, a few of which we describe here. One example would be calls to *ReadFile* and *WriteFile* with network sockets, which is frequently done by Hadoop. Since there is no simple way to differentiate between file and network descriptors in user space, the platform must maintain a list of handles previously issued by *CreateFile* and remember which of them are still active. The second example is dealing with Windows overlapped (asynchronous) I/O, where requests are issued through one API and results (i.e., errors, transfer sizes) are collected through a different API, possibly at a later time and by an unrelated thread. Another interesting case involves jumping within files using a mixture of absolute and relative offsets. For example, seeking from the end of the file requires current knowledge of file size, which changes over time and/or may be initially unknown. To deal with these uncertainties, the hook library must log the new file pointer carried in the response from seek APIs; otherwise, subsequent I/Os cannot be properly ordered for replay.

C. Conversion

To keep the hooking DLL maximally efficient and simple, we delegate the responsibility of reconstructing the I/O-request sequence of the application to an offline package. Its high-level objective is to sort the log entries recorded from the hooked processes using their I/O timestamps, resolve aliases (i.e., multiple handles concurrently referring to the same file), track handle reuse after files are closed, properly locate the position of each I/O based on preceding seeks, and create a chronological trace of disk activity consisting of tuples (file ID, offset, I/O size, type). This system allows fairly complicated scenarios. For example, one process may issue a read of size x at offset y within a given file, followed by a seek to the end of the stream. Then, another process, which has an open handle to the same file, requests a seek to offset $x + y$ and a read of size z. Our platform detects this as a sequential I/O of length $x + z$, despite the multiple seeks in the middle.

The output from the conversion phase includes not only the sequenced I/O tuples, but also the peak size reached by each of the streams. The latter bit of information is needed for the replay component to preallocate space and ensure that file clusters are laid out contiguously on disk. This guarantees that sequential access reaches the maximum read/write speed and incurs no seeks within the file.

D. Replay

The last component of the measurement platform replays the I/O pattern of each framework using a custom *virtual file system* (VFS) that we built on top of NTFS. There are two reasons for creating it. First, sequential I/O speed at the end of hard drives is typically half of that at the start, which stems from different surface density in outer and inner tracks. When the OS is tasked with choosing free clusters on disk, it often writes some of them near the start of the file system and others near the end, i.e., without regard to performance. This leads to unpredictable speed during replay, which is undesirable. The second issue is internal file fragmentation, where the OS scatters clusters of a file into non-contiguous blocks on the volume, causing unnecessary seeks.

To eliminate interference from the OS, the VFS first moves all existing objects to the end of the volume, which is done by reading the MTF (Master File Table) of NTFS and running atomic cluster relocation on each existing file/directory. The VFS then creates a dedicated file of sufficient size to hold all data generated during future replays. If the clusters of this file are not sequential or fail to map to the start of the disk, they are reorganized in correct order as well. To prevent the OS or language wrappers/libraries (e.g., FILE, iostream) from buffering data and effectively extending RAM size M beyond the limits given to each framework, the replay uses unbuffered (i.e., direct) I/O that bypasses the OS cache. This also dramatically increases performance during streaming. In the end, the VFS enjoys maximum sequential I/O speed, physical seeks that correspond only to logical ones found in the trace, and a high level of consistency in the obtained results.

VII. Experiments

The test machine runs an 8-core Intel i7-7820X @ 4.7 GHz with 32 GB of DDR4-3000 RAM and a 24-disk (8 TB Hitachi Ultrastar) RAID-50 array powered by two Areca 1882ix PCIe 3.0 x8 controllers. Across both adapters, the volume has 176 TB of space, a 2-GB onboard cache, and a 4-GB/s sequential read/write speed. Because the merge function f does not shrink/expand the data, $\alpha = 1/2$ applies throughout this section (i.e., αN refers to both input and output size).

A. Java Frameworks

We begin our experiments with Hadoop [7] and Spark [8], which are big-data analytics frameworks commonly used for various sorting, aggregation, join, and MapReduce tasks. For large sorts, both take an enormous amount of time to finish, which makes them impossible to fully benchmark against the other methods. We therefore examine them only briefly to illustrate some of the interesting behavior collected by our trace software.

Consider Hadoop performing a sort of $\alpha N = 100$ GB of 64-bit integers using $M = 25$ GB of RAM, which should result in $n = \alpha N/M = 4$ files that need to be sorted and merged. During this process, which took 6.6 hours, Hadoop issued 2.83 TB of I/O (45% on read, 55% on write), spawned over 2K processes, called *CreateFile* over 11M times, interacted with 100K unique filenames, executed 569M calls to file APIs, and attempted 415M seeks. In theory, this many seeks @ 10 ms each should take 48 days; however, not every such attempt resulted in a physical seek. While we define seeks as non-sequential I/O requests to the OS, the RAID/HDD controllers were able to hide some of the associated latency by buffering data and performing read-ahead. As shown by the 6.6-hour runtime, the cache aided Hadoop tremendously. Eliminating the sort/merge bottlenecks, a replay of Hadoop's I/O in our platform still takes a hefty 1.5 hours, which is enough time to scan through the input 223 times.

Spark performs better, which allows it to finish the same sort with only 10 GB of RAM (i.e., $n = 10$). Over this 10.2-hour job, it performed 511 GB of total I/O, generated 34M seeks, made 20.5M calls to *CreateFile*, and interacted with 16K unique filenames. The corresponding replay finishes in 32 minutes, which is 3× faster than Hadoop, but still far from the level of performance needed for the sorts that we perform later in this section.

B. C++ Frameworks

The remaining systems in our comparison, i.e., STXXL [18], nsort [29], and our method Tuxedo, execute much faster and exhibit better scalability. Their operation can be split into two phases – 1) creation of sorted chunks, which always generates N bytes of I/O across both read/write; and 2) merge/distribution, which uses either the $\mathcal{M}_1$ or $\mathcal{D}_1$ bowtie. Because the I/O access pattern is identical between the methods during the former phase (i.e., sequential scan through two files), our analysis focuses on the latter.

Define ω to be the total I/O of the bowtie and $d = \omega/N$ to be the number of passes it performs over the data. We next examine six scenarios in Table I, where the size of input αN and RAM capacity M are both listed in GB. STXXL in the first column uses half the memory for sorting and the other half for read-ahead, which produces $n = 2\alpha N/M$ sorted chunks. Its selection of merge factor k is not concerned with runtime; instead, it aims to minimize the number of passes over the data. Assuming B is the I/O block size, this amounts to $k = n$ if $kB \leq M$ and $k = \sqrt{n}$ otherwise. In Table I, STXXL enjoys a single pass for all cases except $(2, 8192)$, where the RAM is insufficient to hold $n = 2\alpha N/M = 8192$ blocks of 2 MB each, triggering a double pass. The second column shows nsort, whose non-integer d can be explained by usage of merge factors that do not evenly divide n. As a hypothetical example, a 7-way merge on a 10-file bowtie could yield $1.7N$ bytes of I/O if chunks $1-7$ are merged into one file, which is then merged with the remaining 3.

TABLE I
BOWTIE I/O ω (GB) AND NUMBER OF MERGE/DISTRIBUTION PASSES d

M	αN	STXXL		nsort		Tuxedo	
		ω	d	ω	d	ω	d
1	8	16	1	16	1.0	16	1
2	128	256	1	257	1.0	256	1
	1,024	2,048	1	3,682	1.8	4,096	2
	8,192	32,768	2	32,754	2.0	32,768	2
8	512	1,024	1	1,025	1.0	1,024	1
	4,096	8,192	1	8,690	1.1	8,192	1
20	1,280	2,560	1	2,562	1.0	2,560	1

In the last column, Tuxedo employs Algorithm $\mathcal{A}_2$ to implement the best-achievable run length $L(n, m)$ from Theorem 5 and uses Listing 1 to select the optimal d based on the drive's non-linear I/O response curve $\lambda(n, m)$ from (2). This is in contrast to prior work, which does not attempt to optimize throughput or take into account the characteristics of the underlying file system (e.g., sequential I/O rate S_r, S_w, seek delay δ). Even though rows $(2, 1024)$ and $(8, 4096)$ both use 512 sorted runs, Tuxedo's optimization finds that the former case is better handled by a two-pass approach, while the latter can be done faster in a single pass due to larger RAM.

The benefit of our approach is summarized in Table II, which records the number of read/write seeks $s_r + s_w$ during the bowtie phase. The first row (i.e., $N = 8$ GB input with $M = 1$ GB of RAM) is visualized in Figs. 5(a)-(c) that plot the offset of each I/O request (y-axis) in the order it was issued by each framework (x-axis). Because the block scheduler of STXXL in (a) is not concerned with run length, it performs quite a barrage of seeks (i.e., 66× more than our platform) across the sorted chunks. The situation is similar with nsort in (b), except it uses even smaller blocks and sends occasional requests out-of-order, which interrupts contiguous I/O almost three times as often. This results in 178× more seeks than Tuxedo. In (c), our method maintains the optimal data-access pattern of Algorithm $\mathcal{A}_2$ and achieves the lowest number of inter-stream jumps possible in this situation. Its stationary distribution of bucket sizes, compared to Theorem 3, shows a good match in part (d) of the figure. The remaining cases of Table II exhibit even more impressive gains, where Tuxedo beats STXXL by 249× in last row and nsort by 537×.

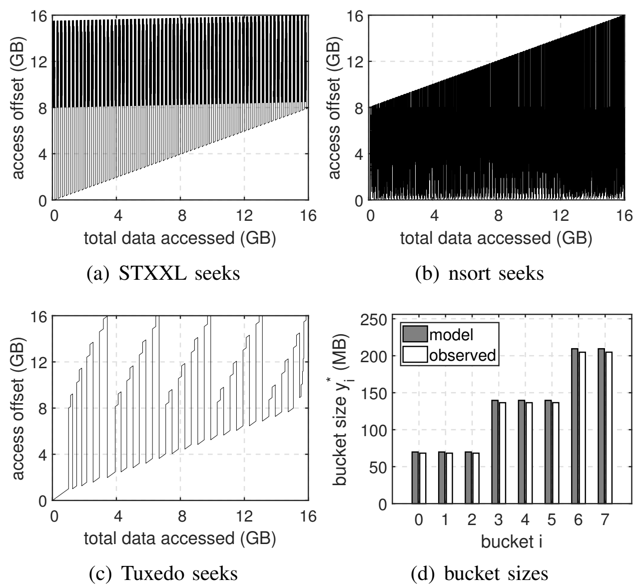
The relationship between run length $L(n, m)$ and bowtie throughput $\lambda(n, m)$ is generally given by (2); however,

TABLE II
NUMBER OF SEEKS DURING THE BOWTIE

M	αN	STXXL	nsort	Tuxedo
1	8	4,165	11,232	63
2	128	66,126	139,772	2,675
	1,024	575,718	2,181,102	18,263
	8,192	12,501,188	21,972,942	341,059
8	512	262,679	598,086	2,648
	4,096	2,102,617	5,040,780	152,382
20	1,280	655,884	1,416,011	2,639

TABLE III
BOWTIE REPLAY RATE $\lambda(n, m)$ (MB/S)

M	αN	STXXL	nsort	Tuxedo
1	8	599	207	2,962
2	128	381	213	2,114
	1,024	367	112	1,350
	8,192	187	86	1,010
8	512	382	198	2,881
	4,096	355	177	1,891
20	1,280	372	188	3,297



(a) STXXL seeks (b) nsort seeks (c) Tuxedo seeks (d) bucket sizes

Fig. 5. Captured bowtie seek patterns and stationary distribution of bucket occupancy in Tuxedo (8 GB input, 1 GB RAM).

there is a caveat to this model. Specifically, it assumes that $(n + m)L(n, m)$ is sufficiently large to prevent buffering/interference from other components of the system (i.e., RAID/HDD caches). For distribution $\mathcal{D}_1$, this translates into $(m + 1)L(1, m) \geq C$, where C is the hardware cache size. Inverting this formula using (7) and $\alpha = 1/2$, we have that Tuxedo cannot benefit from the hardware cache as long as $M \geq C(\sqrt{m} + 1)^2/4(m + 1)$. In our setup with $C = 2$ GB, function (2) is accurate for all m as long as $M \geq 1$ GB.

On the other hand, the existing frameworks execute with much smaller $L(n, m)$, which sometimes allows the cache to artificially increase their run lengths and achieve better performance than suggested by (2). This is demonstrated by the replay speed in Table III, where as before $\lambda(n, m)$ is defined as the total size of input and output (i.e., N) divided by the runtime of the merge/distribution. We examine STXXL first. Even though it runs with $L(n, m) \approx 4$ MB in most cases, the resulting merge speed varies quite a bit. It gets lucky in the first row, where the RAID controllers manage to buffer $n = 16$ streams very effectively, hiding a good portion of the seeks and achieving throughput close to 600 MB/s. However, the remaining cases are much slower, commonly in the 330-380 MB/s range.

In the next column is nsort, which also merges much faster than would be expected considering its volume of seeks; however, it does not enjoy the same boost as STXXL. It is the only existing method in our test that uses overlapped I/O, where the framework issues many parallel requests (i.e., without waiting for completion of earlier I/Os). If such requests all refer to sequential scans, they can increase performance. On the other hand, overlapping on scattered I/O, as nsort does, apparently may serve the opposite role (e.g., interruption of ongoing reads, cancelation of hardware read-ahead), depending on the logic inside the RAID cards and HDDs.

In the last column of the table, our framework delivers rates that are up to $9\times$ faster than STXXL and $17\times$ better than nsort. As expected from the models derived earlier in the paper, Tuxedo becomes faster as RAM size increases, which can be seen by comparing $N = 128$ GB, 512 GB, and 1.25 TB cases, each requiring 64 sorted chunks. The speed gradually increases $2114 \to 2881 \to 3297$ MB/s, with eventual convergence to the full sequential disk rate as $M \to \infty$. On the contrary, STXXL and nsort fail to take advantage of additional resources and deliver approximately the same throughput regardless of M. More importantly, our high-performance platform is built on a suite of theoretical models that allow closed-form optimization of $\lambda(n, m)$ under all input sizes and future hardware configurations. This is in contrast to prior methods, which are heavily dependent on presence of external buffering and whose performance is unpredictably shaped by controller read-ahead/eviction/overlapping algorithms, as well as hidden relationships between input size αN, memory size M, fan-out factor k, cache size C, and disk parameters.

Furthermore, as faster hard drives and PCIe 5.0 devices reach I/O rates in excess of 50 GB/s, the relative cost of seeks will go up substantially, making Tuxedo's advantage proportionally higher. This can be seen in (2), where $S_r, S_w \to \infty$ reduces throughput to a linear function of $L(n, m)$, which in turn is solely determined by the number of seeks. In these cases, the performance margin will resemble that in Table II, i.e., up to two orders of magnitude better for Tuxedo.

C. Sort Results

To study the developed platform in an actual application, we now focus on the full *sort speed*, which we define as the input size αN divided by the time needed to produce the output. Assuming τ is the delay incurred by the creation of sorted runs and recalling that $T(n, m)$ is the bowtie runtime, this translates into $\alpha N/(\tau + T(n, m))$. Compared to the previous

TABLE IV
SORT RATE (MB/S)

M	αN	STXXL	nsort	Tuxedo
1	8	57	56	561
2	128	56	69	554
	1,024	51	50	434
	8,192	39	32	343
8	512	56	55	650
	4,096	55	73	528
20	1,280	55	55	688

subsection, where rate $\lambda(n,m)$ was defined as $N/T(n,m)$, the numbers here will be at least $2-4\times$ lower than in Table III, even without considering bottlenecks in the CPU.

STXXL in Table IV crawls through the various cases at fairly constant rates, typically around 50-60 MB/s. One notable exception is the 8-TB file, where the speed drops to 39 MB/s due to the usage of $d=2$ passes. A similar scenario plays out for nsort, which hovers around 55-75 MB/s in single-pass scenarios and roughly half of that in double-pass. Tuxedo in the last column increases this performance by $10\times$, tackling the 8-GB sort in 14.6 seconds, the 1.25-TB file in 31.8 minutes, and the 4-TB stream in 2.2 hours. A full replay of the last scenario completes only 19 minutes quicker, which confirms that Tuxedo's in-memory sorter/distributor are fast enough to keep up with I/O at all stages of the sort. This exemplifies the type of high-performance computation f that Tuxedo and the remainder of the algorithms in this paper were designed for.

VIII. CONCLUSION

We introduced the concept of *bowtie streaming*, which is a novel generalization of several common computing models in the big-data world, developed a theory for maximizing the throughput of such applications, presented new algorithms for increasing their sequential run length during interaction with multiple files, and optimized the multi-pass operation of the bowtie under fairly general constraints, including data expansion and shrinkage. Our platform Tuxedo, which implements the proposed algorithms, performs external-memory sorts $10\times$ faster than the previous efforts and delivers $2-3$ orders of magnitude fewer seeks.

REFERENCES

[1] A. Aggarwal and J. Vitter, "The Input/Output Complexity of Sorting and Related Problems," *CACM*, vol. 31, no. 9, pp. 1116–1127, Sep. 1988.
[2] T. Akidau, A. Balikov *et al.*, "MillWheel: Fault-tolerant Stream Processing at Internet Scale," *Proc. VLDB Endow.*, vol. 6, no. 11, pp. 1033–1044, Aug. 2013.
[3] A. Alexandrov, R. Bergmann *et al.*, "The Stratosphere Platform for Big Data Analytics," *VLDB Journal*, vol. 23, no. 6, pp. 939–964, Dec. 2014.
[4] D. Alves, P. Bizarro, and P. Marques, "Flood: Elastic Streaming MapReduce," in *Proc. ACM DEBS*, Jul. 2010, pp. 113–114.
[5] Apache Beam. [Online]. Available: https://beam.apache.org/.
[6] Apache Flink. [Online]. Available: https://flink.apache.org/.
[7] Apache Hadoop. [Online]. Available: http://hadoop.apache.org/.
[8] Apache Spark. [Online]. Available: https://spark.apache.org/.
[9] Apache Storm. [Online]. Available: https://storm.apache.org/.
[10] L. Arge, O. Procopiuc, and J. Vitter, "Implementing I/O-efficient Data Structures Using TPIE," in *Proc. ESA*, Sep. 2002, pp. 88–100.
[11] L. Arge, M. Rav, S. Svendsen, and J. Truelsen, "External memory pipelining made easy with TPIE," *IEEE International Conference on Big Data*, pp. 319–324, Dec. 2017.
[12] R. Barve, E. Grove, and J. Vitter, "Simple Randomized Mergesort on Parallel Disks," in *Proc. ACM SPAA*, June 1996, pp. 109–118.
[13] R. Barve and J. Vitter, "A Simple and Efficient Parallel Disk Mergesort," *Theory of Computing Systems*, vol. 35, no. 2, pp. 189–215, 2002.
[14] R. Bauer, "HDD vs SSD: What Does the Future for Storage Hold?" Mar. 2018. [Online]. Available: https://www.backblaze.com/blog/ssd-vs-hdd-future-of-storage/.
[15] F. Corcoglioniti, M. Rospocher, M. Mostarda, and M. Amadori, "Processing Billions of RDF Triples on a Single Machine Using Streaming and Sorting," in *Proc. SAC*, Apr. 2015, pp. 368–375.
[16] Y. Cui, D. Xiao, and D. Loguinov, "On Efficient External-Memory Triangle Listing," in *Proc. IEEE ICDM*, Dec. 2016, pp. 101–110.
[17] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in *Proc. USENIX OSDI*, Dec. 2004, pp. 137–150.
[18] R. Dementiev, L. Kettner, and P. Sanders, "STXXL: Standard Template Library for XXL Data Sets," *Software: Practice and Experience*, vol. 38, no. 6, pp. 589–637, May 2008.
[19] R. Dementiev and P. Sanders, "Asynchronous Parallel Disk Sorting," in *Proc. ACM SPAA*, Jun. 2003, pp. 138–148.
[20] S. Eckels, "PolyHook 2: C++17 x86/x64 Hooking Library." [Online]. Available: https://www.codeproject.com/Articles/1252212/PolyHook-2-Cplusplus17-x86-x64-Hooking-Library.
[21] S. Eckels, "PolyHook 2.0." [Online]. Available: https://github.com/stevemk14ebr/PolyHook_2_0.
[22] V. Gulisano, R. Jimenez-Peris, M. Patino-Martinez, C. Soriente, and P. Valduriez, "StreamCloud: An Elastic and Scalable Data Streaming System," *IEEE TPDS*, vol. 23, no. 12, pp. 2351–2365, Dec. 2012.
[23] T. Hagerup, "Guidesort: Simpler Optimal Deterministic Sorting for the Parallel Disk Model," Feb. 2019. [Online]. Available: https://arxiv.org/abs/1807.11328.
[24] C. Hanel, A. Arman, D. Xiao, J. Keech, and D. Loguinov, "Vortex: Extreme-Performance Memory Abstractions for Data-Intensive Streaming Applications," in *Proc. ACM ASPLOS*, June 2020, pp. 623–638.
[25] J. Hruska, "Seagate Wants to HAMR the Competition, Ship 100TB HDDs By 2025," Nov. 2018. [Online]. Available: https://www.extremetech.com/computing/280190-seagate-wants-to-hamr-the-competition-ship-100tb-hdds-by-2025.
[26] X. Hu, Y. Tao, and C. Chung, "Massive Graph Triangulation," in *Proc. ACM SIGMOD*, Jun. 2013, pp. 325–336.
[27] A. Lakshman and P. Malik, "Cassandra: A Decentralized Structured Storage System," *SIGOPS Oper. Syst. Rev.*, vol. 44, no. 2, pp. 35–40, Apr. 2010.
[28] A. Moniruzzaman and S. Hossain, "NoSQL Database: New Era of Databases for Big data Analytics – Classification, Characteristics and Comparison," *International Journal of Database Theory and Application*, vol. 6, no. 4, pp. 103–111, Aug. 2013.
[29] Nsort. [Online]. Available: http://www.ordinal.com/.
[30] A. Patrizio, "Japanese firm announces potential 80TB hard drives," Aug. 2020. [Online]. Available: https://www.networkworld.com/article/3528211/japanese-firm-announces-potential-80tb-hard-drives.html.
[31] Seagate, "Seagate MACH.2 Multi Actuator Technology Breaks Throughput Record; HAMR Reliability Tests Exceed Industry Standards," Aug. 2019. [Online]. Available: https://blog.seagate.com/enterprises/mach2-and-hamr-breakthrough-ocp/.
[32] A. Shatnawi and Y. Alzahouri, "A multi-pass algorithm for sorting extremely large data files," in *Proc. ICICS*, Apr. 2015, pp. 79–82.
[33] A. Shilov, "Seagate's Roadmap: The Path to 120 TB Hard Drives," Mar. 2021. [Online]. Available: https://www.anandtech.com/show/16544/seagates-roadmap-120-tb-hdds.
[34] Sort Benchmark. [Online]. Available: http://sortbenchmark.org/.
[35] G. Stella and D. Loguinov, "The Tuxedo Project." [Online]. Available: http://irl.cse.tamu.edu/projects/streams.
[36] J. Vitter and E. Shriver, "Algorithms for parallel memory, I: Two-level memories," *Algorithmica*, vol. 12, no. 2, pp. 110–147, Sep. 1994.
[37] J. Vitter, "External Memory Algorithms and Data Structures: Dealing with Massive Data," *ACM Computing Surveys*, vol. 33, no. 2, pp. 209–271, Jun. 2001.
[38] J. Vitter, "Algorithms and Data Structures for External Memory," *Foundations and Trends in Theoretical Computer Science*, vol. 2, no. 4, pp. 305–474, 2006.