On High-Latency Bowtie Data Streaming

Gabriel Stella, Dmitri Loguinov

Internet Research Lab (IRL)
Department of Computer Science and Engineering
Texas A&M University

December 18, 2022

Agenda

- Motivation
- Bowtie Streaming
- Optimizing Run Length
- Multi-Pass Optimization
- Experiments

Motivation

- Many applications use external-memory (EM) algorithms to process datasets larger than RAM
- This often requires *concurrent* I/O with multiple files
  - Sorting (merging/distribution)
  - MapReduce computation
  - Graph mining
  - Database join/group/aggregate queries
- Parallel I/O is challenging because large-scale storage frequently uses arrays of HDDs
  - High sequential read/write transfer speed (S_r, S_w), but large seek delays, i.e., switching between files is expensive

Computer Science, Texas A&M University

Motivation

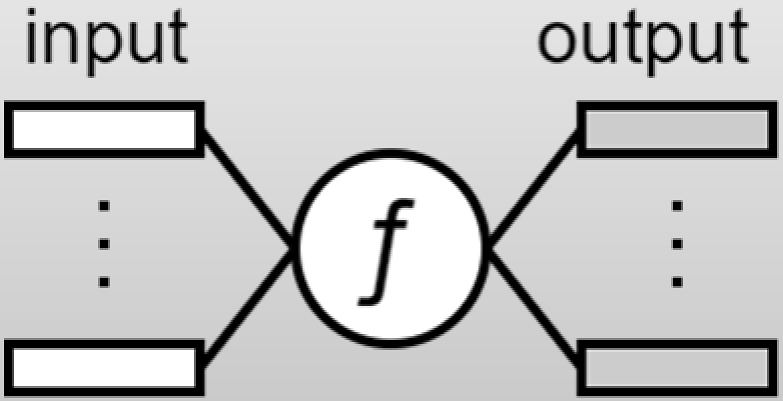
- Long history of EM research [Aggarwal88], [Vitter94], [Dementiev08], [Blelloch15], [Shatnawi15], [Arge17]
  - Does not account for seeking and not concerned with runtime
  - More recent theory [Bender19] goes to the other extreme and assumes that every I/O incurs a seek
- Instead, a realistic EM model should
  - Focus on the *runtime* of the application
  - Explicitly account for the fact that EM algorithms perform bursts of sequential I/Os interleaved with random seeks
- We fill this void by introducing a novel I/O model called *bowtie streaming* and modeling its performance

Agenda

- Introduction
- Bowtie Streaming
- Optimizing Run Length
- Multi-Pass Optimization
- Experiments

Bowtie Streaming

- Definition: a *stream* is an external data object without support for random access



- Definition: a $n \times m$ *bowtie* is an EM computation that runs a user-supplied function *f*, which reads from n input streams, each at some average rate λ_{in}, and writes to m output streams, each at some other rate λ_{out}

- A bowtie is called *high-latency* if the inter-stream seek delay δ is non-negligible compared to the time spent sequentially accessing each file

Bowtie Streaming

- Let N be the total amount of data across all streams, with fraction α coming from input, and $M < N$ the amount of memory available to the I/O scheduler
- Definition: assuming s is the total number of stream switches, the runtime of a bowtie application is

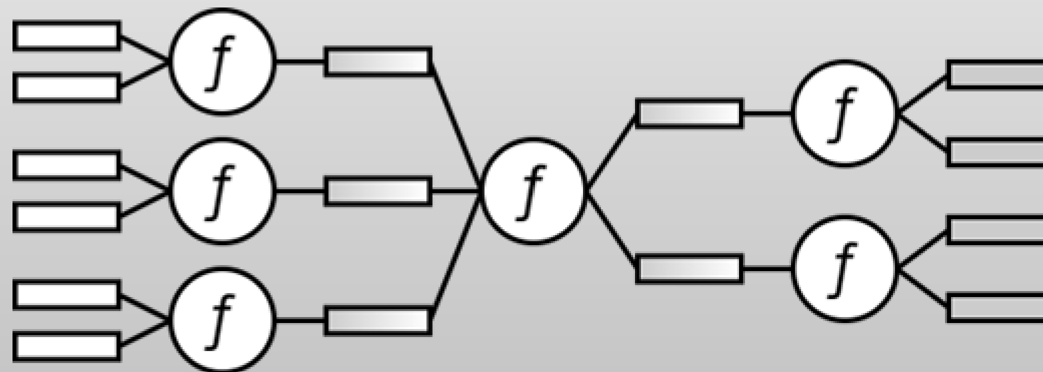
$$T = \frac{\alpha N}{S_r} + \frac{(1-\alpha)N}{S_w} + s\delta$$

- It is often more useful to view performance in terms of the *average sequential run length* $L = N/s$
- Definition: the *throughput* of a bowtie application is

$$\lambda = \left[\frac{\alpha}{S_r} + \frac{1-\alpha}{S_w} + \frac{\delta}{L}\right]^{-1}$$

Bowtie Streaming

- The optimal solution to an $n \times m$ bowtie may require a decomposition into smaller bowties
  - A 6x4 case may be split into three 2x1 merge bowties, followed by a 3x2 interconnect, and then two 1x2 distribution bowties



  - But can we do better and under what conditions?

- Objective: assuming a d-pass bowtie with rate λ_i in level i, maximize the overall throughput

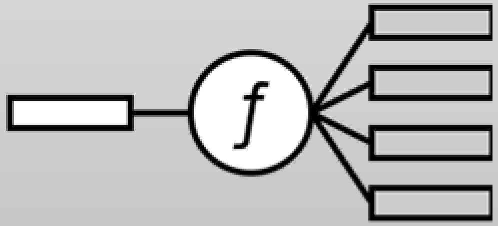
$$\left(\sum_{i=0}^{d-1} \frac{1}{\lambda_i}\right)^{-1}$$

Agenda

- Introduction
- Bow Streaming
- Optimizing Run Length
- Multi-Pass Optimization
- Experiments

Optimizing Run Length

- The first step is to optimize single-pass bowtie performance, which translates into maximizing L
- Discussion here focuses on one particular scenario; see the paper for the other four
  - In *distribute-from-file*, data from a single input stream is split into m destinations



- Most existing methods (Spark, Hadoop, STXXL, [Vitter94]) perform I/O *on demand*, i.e., without buffering ahead, minimizing seeks, or taking into account memory size M

Optimizing Run Length

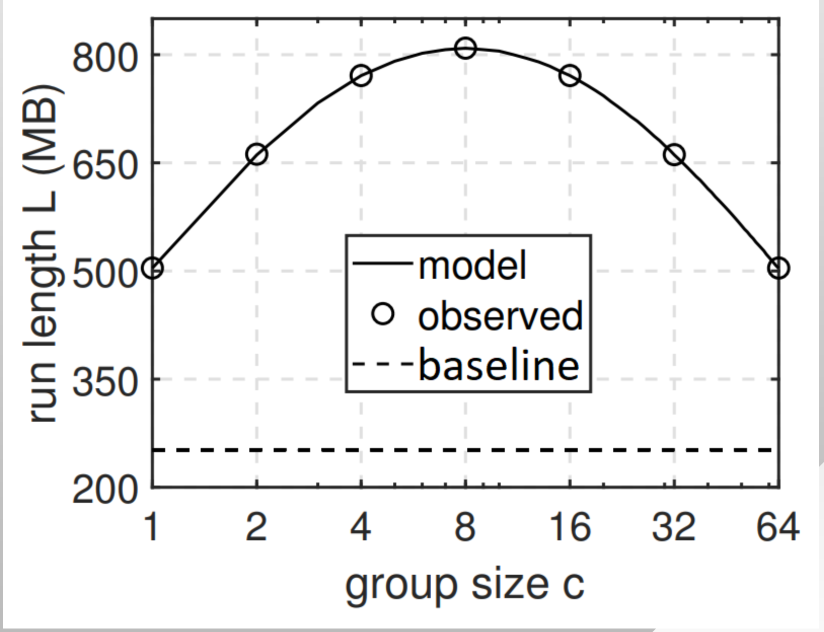
- Baseline memory-aware approach:
  - Split memory in half between input and output
  - Read $M/2$ from input, distribute the data, empty all buckets
- Theorem: the baseline algorithm yields $L = M/(m+1)$
  - But we can do even better with a new formalism
- Definition: the buffer of pending data for each stream i is called a *bucket* and its size at time t is $X_i(t)$, where $\sum X_i(t) \le M$
- Definition: a *bucket game* is an in-memory scheduler that decides which buffer(s) to empty when the memory is exhausted (i.e., $\sum X_i(t) = M$)

Optimizing Run Length

- Note that the bucket game assumes negligible buffering on the reader side (e.g., two blocks)
- The objective is to design a scheduler that achieves the largest L
  - If e_i is the set of buckets emptied during step i, each bucket game is described by some vector $q = (e_1, e_2, \ldots)$
  - Selection of optimal q for general cases is complicated, but is tractable for certain scenarios of interest
- Emptying the single largest bucket seems like a reasonable solution, but we consider a more general problem that removes the $c \geq 1$ largest buckets
  - A simulation is available at gabrielrstella.com/buckets.php

Optimizing Run Length

- Theorem: the $1 \times m$ bucket-game system of recurrences converges to a unique steady state whose run length is optimized by $c = \sqrt{m}$

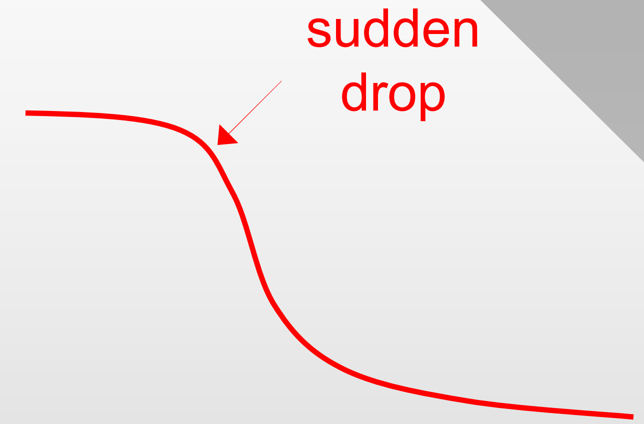


- Theorem: The optimal run length for the $1 \times m$ bowtie is $L = 4M/(\sqrt{m}+1)^2$
  - This is almost 4x better than baseline
  - With $m = 64$ files and memory size $M = 16$ GB, the baseline gets $L = 252$ MB, $c = 1$ yields 500 MB, while the optimal approach with $c = 8$ reaches $L = 809$ MB

Agenda

- Introduction
- Bowtie Streaming
- Optimizing Run Length
- Multi-Pass Optimization
- Experiments

Multi-Pass Optimization



- Throughput $\lambda(m)$ begins relatively flat for small fan-out factors, but then exhibits a sharp decline after some threshold
  - To increase performance at large $n + m$, multiple passes over the data are usually beneficial

- Idea: use dynamic programming to find the optimal set of intermediate bowties that minimizes the total runtime

- Alg 1: find optimal $n \times 1$ and $1 \times m$ bowties under arbitrary λ functions

- Alg 2: determine the best interconnect $i \times j$ that creates the optimal ($n \rightarrow i \times j \rightarrow m$) multi-pass bowtie

Multi-Pass Optimization

- Example: for an $1 \times m$ bowtie, the algorithm finds a list of split factors $(m_1, \ldots, m_d)$, where $\prod_{i=0}^{d-1} m_i = m$, such that the total throughput $[\sum_i 1/\lambda(m_i)]^{-1}$ is maximized
  - The single-pass solution runs $\lambda(m) \sim 1/m$ as $m \to \infty$, while the multi-pass has much better scalability $\lambda(m) \sim 1/\log(m)$
- Consider a 1×8000 bowtie outputting 64 TB using $M = 8$ GB on a 24-HDD RAID system with sequential I/O speed $S_r = S_w = 4$ GB/s and seek delay $\delta = 10$ ms
  - Prior work often suggests one pass, which runs @ 208 MB/s
  - Binary splits, another alternative that appears in related work, require 13 passes, which gives 273 MB/s
  - The optimal split vector (90, 89), however, pushes 1353 MB/s

Agenda

- Introduction
- Bowtie Streaming
- Optimizing Run Length
- Multi-Pass Optimization
- Experiments

I/O Tracing

- Due to CPU bottlenecks and various OS-related side-effects (fragmentation, buffering) in most real systems, comparison of I/O performance is complex
- We develop a novel I/O measurement platform that:
  - Intercepts and records all I/O calls from a process and its children (with negligible measured performance impact)
  - Merges and converts log files into a single list of instructions
  - Replays the I/Os in a standalone, performance-optimized, and defragmented file system
- This enables not only analysis of process I/O patterns (e.g., seek counts, run length L), but also a systematic evaluation of throughput λ across the methods

Java Frameworks

- We start with Hadoop and Spark, two popular Apache data-processing frameworks (24 HDD, 160 TB RAID)

Java Framework Results Sorting 100 GB					
Framework	RAM (GB)	Sort (hr)	Replay (min)	Seeks	L (Bytes)
Hadoop	25	6.6	90	415M	517
Spark	10	10.2	32	34M	6316

- Even though the volume delivers 4 GB/s sequential speed, the replay was 2 orders of magnitude slower
  - Hadoop spawned 2K processes that executed 2.83 TB of I/O across 569M API calls, including 11M calls to CreateFile
  - Spark required 511 GB of I/O and issued 20.5M calls to CreateFile, interacting with 16K unique filenames

C++ Frameworks

- To build on the theory developed earlier, our platform Tuxedo implements optimal multi-pass I/O-scheduling for general bowties
  - We test it by constructing a sorting application for large files consisting of 64-bit uniform keys
  - The in-memory component runs an m-way depth-first-search distribution bowtie, followed by the Vortex framework [Hanel20] that sorts memory-size chunks at the leaves
- Benchmarks also include
  - STXXL: an open-source high-performance EM algorithm suite
  - nsort: popular commercial sorting software that has been used as the backbone of several large sorting systems

C++ Frameworks

Average Sequential Run Length L (MB/seek)				
RAM (GB)	Input (GB)	STXXL	nsort	Tuxedo
1	8	3.9	1.5	260
2	128	4.0	1.9	98
2	1024	3.6	1.0	115
2	8192	1.3	0.8	49
8	512	4.0	1.8	396
8	4096	4.0	1.7	55
20	1280	4.0	1.9	993

- Tuxedo achieves 2-3 orders of magnitude larger L
  - This benefit gets larger as M increases
- Note that in the three highlighted cases, Tuxedo exhibits perfect linear scaling with M

C++ Frameworks

Replay Bowtie Rate λ (MB/s)				
RAM (GB)	Input (GB)	STXXL	nsort	Tuxedo
1	8	599	207	2,962
2	128	381	213	2,114
2	1024	367	112	1,350
2	8192	187	86	1,010
8	512	382	198	2,881
8	4096	355	177	1,891
20	1280	372	188	3,297

- When looking at just the bowtie I/O scheduling, Tuxedo is up to 17x faster
  - Our performance will continue to get better when given more memory and/or faster storage hardware

C++ Frameworks

- We finish with full sort results
- When comparing sort and replay rates, the numbers here will be significantly lower for several reasons:
  - Sort rates are calculated as $\alpha N/T$ (only input is counted)
  - Sort times include both the bowtie passes *and* the run-formation phase (replays are bowtie-only)
  - STXXL and nsort are both heavily CPU-bottlenecked
- Tuxedo's low computational cost makes our full sorts only ~10% slower than the corresponding replays

C++ Frameworks

Sort Rate (MB/s)				
RAM (GB)	Input (GB)	STXXL	nsort	Tuxedo
1	8	57	56	561
2	128	56	69	554
2	1024	51	50	434
2	8192	39	32	343
8	512	56	55	650
8	4096	55	73	528
20	1280	55	55	688

- Tuxedo's 7-12x improvement over the existing systems offers an appealing big-data engine for various EM tasks (e.g., analytics, graph mining, databases)

- **Questions?**