Succinct

Re-architecting data representation, storage and retrieval

Anurag Khandelwal, Rachit Agarwal, Ion Stoica

- A new initiative at AMPLab, with a particular focus on systems for storing and retrieving data for queries
- General theme at AMPLab:
  - Low latency via pushing data into memory
  - Spark, Spark Streaming, Shark
- More challenging for systems serving queries
  - Larger volumes of data (input data, secondary indexes)
  - Low latency is a necessity, not a performance parameter

Traditional Approaches

How to push more data into memory?

- Use more memory
  - Short term solution
  - Data growth has outpaced memory advances
- Use traditional compression techniques
  - Reduce the data size significantly
  - However, even simple data access requires complete or partial data decompression/scan; complex queries may decompress/scan a large fraction of the data.

Succinct

Designing techniques and building systems that allow operating directly on a compressed representation.

Our first goal is to build a system that allows efficient random access and search directly on a compressed representation. This gives us a data store with:

- API a step closer to that of traditional RDBMS
- Scalability of NoSQL stores

Example

UserID	UserInformation	HasBeenTo	Likes
12444	..“San Francisco”..	..“England”..	..“Skydiving”..
⋮	⋮	⋮	⋮
12462	..“San Francisco”..	..“England”..	..“Skydiving”..

List Users that Live in “San Francisco” **and** Have been to “England” **and** Like “Skydiving”

- **Current Approaches:**
  - Keep original data
  - Maintain large secondary indexes (100s of TB, potentially compressed)
- **Our Approach:** A Succinct representation that:
  - Is **value searchable**: allows arbitrary string searches on values (Functionality)
  - Does not require **any** indexes (Scalability). In fact, resulting data structure size **smaller than the input file size**!

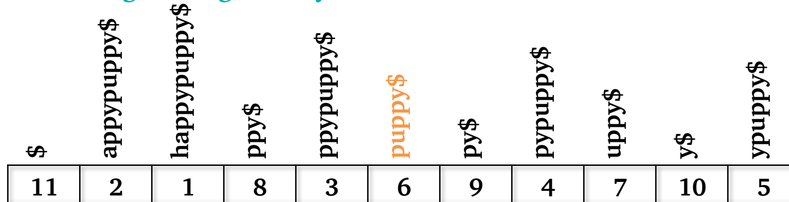
Technique Overview

Example String: happypuppy$

Step 1: Extract the set of suffixes from input data.

Step 2: Sort suffixes lexicographically.

Step 3: Construct an array, with i^{th} index containing location of ith sorted suffix. Allows searching arbitrary substrings using binary search.

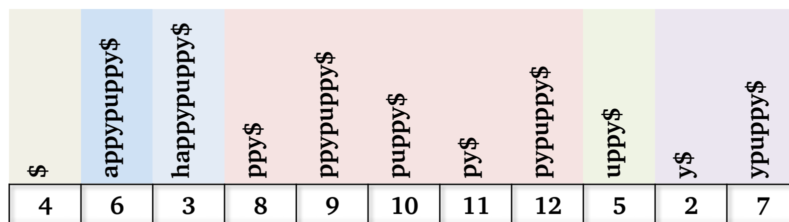


Problem: Each entry takes $\log n$ bits; total size is $n \log n$.

Step 4: Solution: store sampled values only.

Problem: How to lookup unsampled values?

Step 5: Construct an array, with i^{th} entry storing the index of the successor suffix of ith sorted suffix. Allows looking up unsampled values by following next-hops.

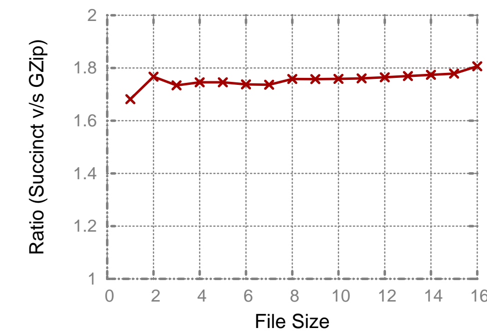


Step 6: Increasing integer sequences are compressible! Most of the innovation done here — get contiguous integer sequences, leading to good compressibility.

Further extensions give:

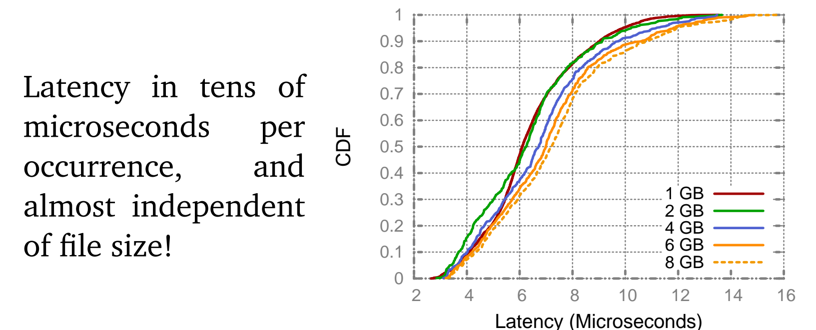
- Wildcard searches
- Lexicographic range queries
- Random access on compressed tables

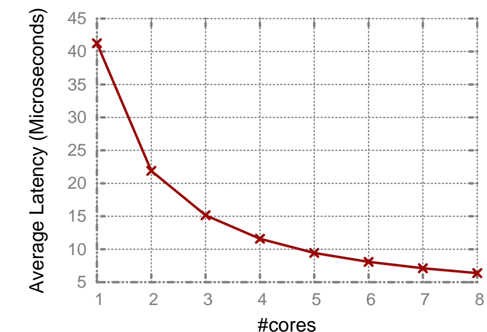
Preliminary Benchmarks



Memory footprint close to GZip-compressed file size! (Including every bit stored!)

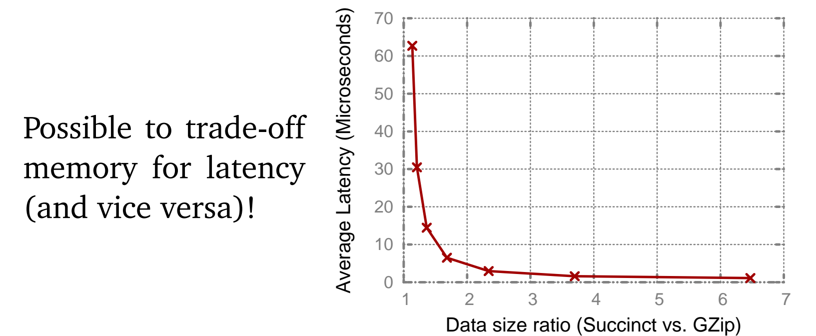
Latency in tens of microseconds per occurrence, and almost independent of file size!





Latency scales almost linearly with number of cores!

Possible to trade-off memory for latency (and vice versa)!



Research Challenges

- Fine-grain updates challenging
- Distributed query implementation requires touching multiple machines
- Network latency quickly becomes a bottleneck
- Not yet explored: fault tolerance and consistency issues

www.succinct.cs.berkeley.edu

{anuragk, ragarwal, istoica}@eecs.berkeley.edu