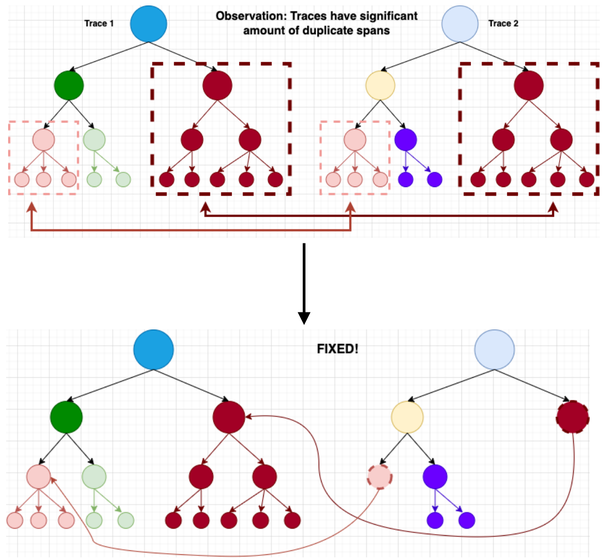
TraceZip: Compression Technique for Distributed Tracing

Introduction: Microservices architecture is gaining significant attraction due to the growing complexity of web services. Microservices use Remote Procedure Calls for communicating with each other and the history of these RPCS, called traces, is crucial for the debugging purposes. Storing the traces is impractical due to their huge volume. So the volume of the stored traces is reduced. Sampling is one widely used method for achieving this. However, reducing the data stored leads to information loss that in turn affects the debugging performance. We present TraceZip, a compression technique for traces that can save ~75% of the storage capacity.

Compression Technique:

- Traces can be represented as graphs where Microservices are represented by the nodes and the edges represent the RPC calls.
- These graphs have significant amount of common spans (within a single trace as well as across multiple traces).
- TraceZip identifies the common spans and only stores them once.



TraceZip helps in saving ~75% of the storage capacity via effective compression of the traces!

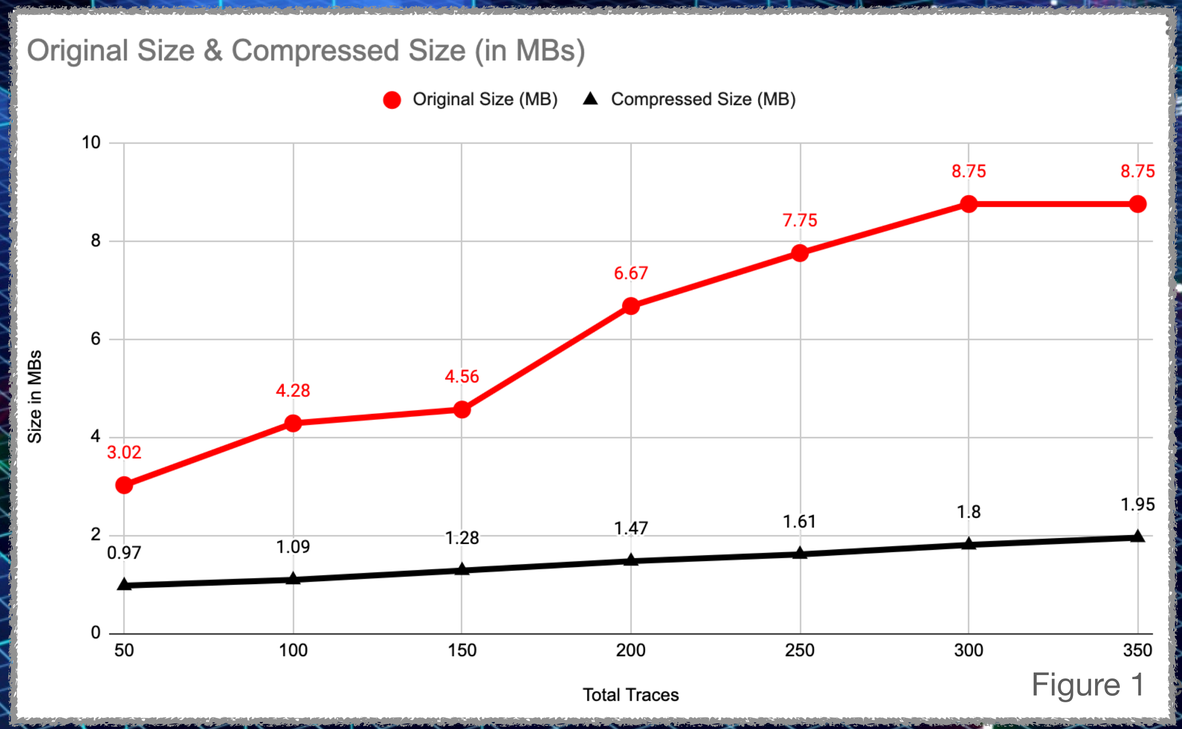
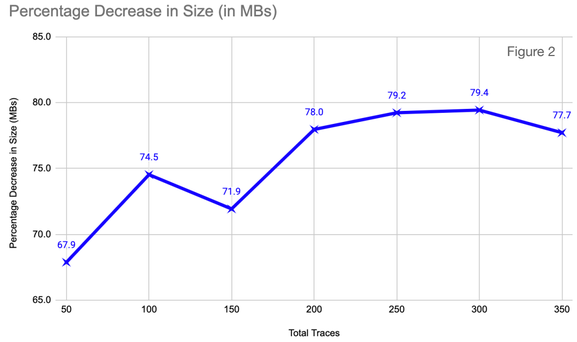


Figure 1

Analysis & Results: We setup a network of Microservices containing 6 nodes and use Jaeger for producing the traces. TraceZip compression is applied on the traces and the result is stored in an SQL database. Another SQL database is used for storing the uncompressed traces. The two databases are comparatively analyzed. Figure 1 shows the difference between the sizes of the two databases. Figure 2 shows the percentage decrease in the database sizes. It is evident that more than 75% of the storage capacity can be saved by using TraceZip compression. It effectively eliminates the need for traces sampling and in doing so, we avoid making a compromise on the debugging performance.



Future Work:

- Time analysis for compression and de-compression
- Comparison with Cassandra and GZip compression
- Timestamps compression using a base value and deltas

Related Work: Almost all of the related literature has focused on developing a smart sampling technique so that only a subset of the traces is needed to be stored. These sampling techniques can be categorized into two general classes: (i) head-based sampling and, (ii) tail-based sampling. In head-based sampling, the trace is stored or ignored before it has been completed. In tail-based sampling, a trace is first completed, and then it is sampled. Canopy, Dapper, and Jaeger are examples of the former. Lightstep and some parts of Canopy employ different flavors of the latter.
Another approach is query-based distributed tracing. This includes Pivot Tracing and Snicket. They do not store the traces as they run the queries in real-time in the microservices and only store the results of the pre-defined queries. The main disadvantage is that they limit the historical visibility into the microservices state when it comes to debugging. TraceZip can also be used as an extension of these works for improving the state visibility while maintaining the storage costs at a reasonable level.