Hyperion: Stream Archival for Large Volumes and Retrospective Queries

Srinivas Gadam
Technology Architect
TEXAS, USA,
sgadam77@gmail.com

ABSTCRT:

Network monitoring systems with data archiving and retrospective queries are beneficial for various applications, including anomaly detection and security forensics. Nonetheless storing data for these kinds of systems is hard because a single link can handle hundreds of thousands of packets of data per second, and this data needs to be searched securely so that queries can be done in the past. At these data speeds, both database indexes and general-purpose file systems function badly. The Hyperion system—which allows for the online retrieval, indexing, and archiving of large data streams—is explained in this study. We implement a write-optimized stream file system to facilitate the high-speed storage of simultaneous data streams, as well as a novel approach to signature file indexes in a distributed multi-level index. Our study of Hyperion on commodity hardware includes fake data and real network trace analysis. Over a million packets per second of trace archiving can be achieved with our streaming file system, StreamFS, demonstrating its speed. The index enables queries over hours of data to be completed few seconds. The system can index and archive more packets while performing simultaneous online queries.

Keywords: Hyperion, File System, Index, Security, Network, Analysis.

I. INTRODUCTION

Reason for Motivation: Network monitoring, which involves collecting and inspecting packet headers, has grown popular for a variety of management and forensic objectives, ranging from tracing down system attackers to detecting mistakes or performance issues. Network monitoring solutions are available in two flavours. Live monitoring involves the monitoring system capturing and analysing packets in real-time. Such systems can continuously query the packet stream to detect certain situations [20], compute and update traffic statistics, and proactively detect security assaults by scanning for worm or denial of service signatures [9]. In live monitoring systems, captured packet headers and payloads are deleted after examination, regardless of the specific usage.

Nonetheless, there are numerous situations in which keeping packet headers around for a while is beneficial. Network forensics is one such example—the ability to "go back" and retrospectively examine network packet headers is extremely useful for network troubleshooting (e.g., root-cause analysis), determining how an intruder gained access to a computer system, or determining how a worm entered a specific administrative domain. These network monitoring systems need to be able to query and analyse live data, as well as store archived data. At wire speeds, these systems must capture, archive, and index data. Moreover, they must quickly retrieve old data in order to respond to backward-looking enquiries.

Currently, there are two options for designing an archiving system for data streams. A relational database can be used to archive data, or a bespoke index can be built on top of a standard file system. The structure of collected information—a header for each packet with a series of fields—naturally lends itself to a database view. Systems like GigaScope [6] and MIND [18], which have a SQL interface for querying network monitoring data, are the result of this.

The rate at which a monitoring system must receive new data must be high. A single gigabit link can generate hundreds of thousands of packet headers per second and tens of Mbyte/s of data to archive, and a single monitoring system may record from multiple links. Using conventional database systems has been made impossible by these rates. MIND, which is built on a peer-to-peer index, collects and saves only flow-level data, not raw packet headers. GigaScope is a stream database, and, like previous stream databases to date [20, 27, 1], it offers continuous queries on live streaming data; data preservation is not a design consideration in such systems. GigaScope, for example, can conduct continuous searches on data from some of the Internet's fastest links, but it relies on other methods to save results for later use.

Alternatives include storing captured packet headers in log files and creating a special-purpose index for faster querying. However, a general-purpose file system is not built to take use of the unique properties of network monitoring applications, resulting in lower system throughput than is possible. Unix-like file systems, for example, are normally designed to write little files and read large ones sequentially, whereas network monitoring and querying write very large files at fast data rates while issuing small random reads. The huge data volume in these applications and the requirement to bound worst-case performance to avoid data loss may make it desirable to optimise the system for certain access patterns instead of using a general-purpose file system.

Thus, the particular demands imposed by high-volume stream archiving indicate that neither traditional databases nor file systems are well-suited to meet their storage requirements. This highlights the need for a new storage system based on commodity hardware and specifically built to meet the demands of high-

volume stream archiving in terms of disc performance, indexing, data ageing, and query and index distribution.

Introduce in this study a unique stream archiving system, called Hyperion1, for indexing and preserving high-volume packet header streams.

PIER [15] and MIND [18] can query historical network monitoring data via a distributed network, just like Hyperion can. Both of these systems, however, rely on DHT structures that cannot handle the rapid insertion rates needed for indexing packet-level trace data. They can only index slower sources like flow-level information. The Gigascope network monitoring system has the capability to handle network monitoring streams at their maximum speed and offers a query language based on SQL. But these queries can only be used on new data streams; GigaScope does not have a way to do queries on data that has already been collected. StreamBase [28] is a versatile stream database that, similar to GigaScope, has the capability to process streaming queries at exceptionally high rates. Furthermore, StreamBase offers support for permanent tables for retrospective queries, just as Hyperion. However, these tables are B-tree- or regular hash-indexed tables, thus they have the same performance restrictions.

Several systems, like the Endace DAG [10], have been created to collect and store packet monitoring data at wire-speed. However, these systems are only meant to be used for offline analysis of data and don't offer any ways to index or even access the data. CoMo [16] focuses on high-speed monitoring and storage, including capabilities for both real-time streaming and historical inquiries. However, despite the fact that it contains a storage component, it does not incorporate any system for indexing, which restricts its usefulness for querying big monitor traces. The log structure of StreamFS is influenced by the original Berkeley log-structured file system [22] and the WORM file system from the V System [5]. Numerous studies have been conducted in the past to facilitate streaming reads and writes for multimedia file systems (e.g. [30]); however, the sequential-write random-read nature of our issue necessitates solutions that are substantially distinct.

To learn more about Bloom filters and the signature file index used in this system, read a study by Faloutsos [11] and Bloom's original article [3]. This body of work (e.g., [23]) as well as other domains, including sensor networks [14, 7], have both described multi-level and multi-resolution indices.

Three components make up Hyperion:

- StreamFS, a stream file system optimized for sequential immutable streaming writes;
- A multi-level index based on signature files, previously used by text search engines to maintain high update rates;
- A distributed index layer that distributes coarse-grain summaries of locally archived data to other nodes, enabling distributed querying.

We have deployed Hyperion on open-source Linux servers, and we have utilised our prototype to carry out an extensive experimental assessment with actual network traffic. In our tests, the worst-case StreamFS throughput for streaming writes is 80% of the average disc speed, which is about 50% greater than the best general-purpose Linux file system. In addition, StreamFS is demonstrated to be capable of handling a workload similar to streaming a million packet headers per second to disc while responding to simulated read requests. Conversely, our multi-level index can handle data rates exceeding 200K packets/sec and offers interactive query responses, allowing it to search through an hour's worth of trace data in a matter of seconds. Lastly, we look at the overhead associated with expanding a Hyperion system to tens of monitors and use an actual case study to illustrate the advantages of our distributed storage solution.

II. RESEARCH DESIGN

Two Difficulties in Design

1. Many issues need to be taken into consideration when designing a high-volume archiving and indexing system for data streams.
2. Archive several, high-throughput streams.

A single densely loaded gigabit link can readily produce monitor data at a rate of 20Mbyte/sec2; a single system may need to monitor several such links, resulting in a pace significantly exceeding this. Simply storing this data as it arrives may be a challenge, as a commodity hardware-based system of this size must rely on disc storage; while the peak speed of such a system is adequate, the worst-case speed is significantly lower than what is necessary. Utilizing the features of contemporary discs and disc arrays, along with the sequential append-only nature of archival writes, is required to reach the required rates.

- Maintain indexes on archived data
- Extensive searches via archived data

Would be prohibitively expensive, thus an index is required to handle most queries. Updates to this index must be stored at wire line speed over time, as packets are captured and archived. Consequently, they must support particularly efficient updating mechanisms. This high update rate (e.g., 220K pkts/sec in the case above) rules out several index structures, such as a B-tree index over the entire stream, which would necessitate one or more disc operations for each insertion. Unlike storage performance requirements, which must be met, to avoid data loss, retrieval performance is less crucial; still, we aim to make it efficient enough for interactive use. A extremely selective query, which returns relatively few data entries, is designed to search an hour of indexed data in 10 seconds.

Storage capacity is limited in compared to arriving data, which is virtually limitless if the system operates long enough. Reclamation and reuse of storage are so required. Data ageing strategies that erase the oldest or least useful data to make room for new data are required, and data should be removed from the index as it ages.

A typical monitoring system will have several monitoring nodes, each monitoring one or more network links. For example, in network forensics, it can occasionally be required to query data stored at several nodes in order to track events (like worms) as they propagate across a network as in Figure 1. This distributed querying necessitates some form of coordination between monitoring nodes, which entails a trade-off between the distribution of data and queries. If too much data or index information is disseminated across monitoring nodes, it may limit the system's overall scale as the number of nodes grows; if requests must be flooded to all monitors, query performance will not scale utilize standard hardware. The use of commodity processors and storage limits the processing and storage bandwidths available at each monitoring node, therefore the system must optimize resource consumption to scale to large data volumes.

There are three design principles for Hyperion.

The challenges described in the preceding section give rise to three fundamental concepts that will guide or system design.

A. Addressing enquiries, not accessing information

A versatile file system facilitates fundamental processes such as reading and writing. Nevertheless, the nature of monitoring applications ensures that data is typically accessed through queries. For example, in relation to Hyperion, these queries would be predicates that identify values for specific packet header fields, such as the source and destination address. Therefore, a stream archiving system must facilitate data retrieval using queries rather than direct reading of

unorganised data. Effective query support necessitates the upkeep of an index, especially one that is optimised for high update rates.

B. Utilise consecutive, unchangeable writing operations to gain an advantage.

Stream archiving produces continuous sequential writes to the underlying storage system; these writes are often immutable because data is not updated once archived. The system should utilise data placement strategies that take advantage of these I/O properties in order to minimise disc search overheads and enhance system throughput.

C. Store data locally, offer concise summaries worldwide.

There is an inherent conflict between the necessity to scale, which prioritises local archiving and indexing to prevent network writes, and the necessity to prevent overflow in order to respond to distributed queries, which prioritises the sharing of information across nodes. We "resolve" this problem by suggesting an architecture in which data archiving and indexing are done locally and a coarse-grained summary of the index is sent between nodes to allow for distributed querying without flooding.

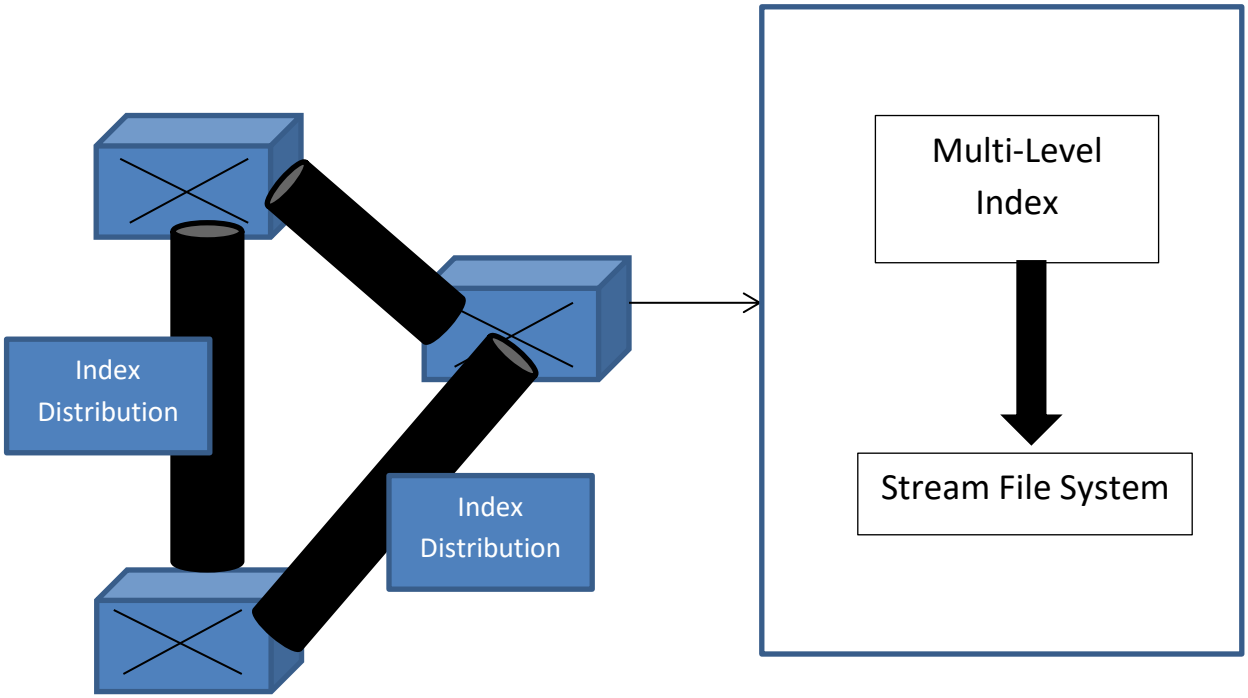

Figure 1. Hyperion network monitoring system

Taking these ideas into consideration, we created Hyperion, a stream archiving system with three main parts:

- An intricately streamlined stream directory for large-scale archiving and retrospective querying;
- A multi-level index structure built for high update rates with respectable lookup performance; and
- A distributed index layer that disperses a coarse-grain summary of the local indices to facilitate distributed queries

III. Hyperion Stream File System

The Hyperion storage system must meet the following requirements: it should be capable of storing several high-speed traffic streams without any data loss, allow for reuse of storage on a full disc, and support concurrent read activity without compromising write performance. The primary obstacle to fulfilling these requirements is the inconsistency in the performance of standard disc and array storage. While it is very simple to construct storage systems that can handle the necessary workload under ideal conditions, the performance can degrade significantly, up to a thousand times slower, under unfavourable conditions.

In this part, we will first investigate building this storage system on top of a general-purpose file system. After examining the performance of many standard file systems on stream writes generated by our application, we introduce StreamFS, an application-specific file system for stream storage. In order to address these concerns, we first construct a stream storage system in greater depth. A general-purpose file system stores files, while a stream storage system stores streams. The following streams are:

- ✓ **Recycled:** When the storage system reaches its maximum capacity, fresh data can be written successfully, but old data is discarded or overwritten in a circular buffer manner. Unlike a general-purpose file system, which loses new data and retains old data, this operates differently.
- ✓ **Immutable:** while an application can add new data to a stream, it cannot edit data that has already been written.
- ✓ **Record-oriented:** features like timestamps are linked to specific ranges within a stream instead of the stream itself. In StreamFS, data can be written in records that have intact bounds when retrieved.

The Hyperion stream file system, StreamFS, uses LFS's log-structured write allocation; as seen in Figure 2, all writes occur at the write frontier, which advances as data is written. LFS requires a garbage collector, or segment cleaner, to eliminate fragmentation that happens when files are deleted; however,

StreamFS does not require this and never transfers data in normal operation. This addresses the main limitation of log-structured file systems by utilising the StreamFS storage reservation system and leveraging the characteristics of stream data.

The simplest approach to accomplish this would be to overwrite all data as the write frontier advances, resulting in a single age-based expiration policy for all streams. This strategy fails to account for variations in both data transfer speed and the length of time data needs to be stored. In contrast, StreamFS offers a storage guarantee to each stream, ensuring that no records from a stream will be reclaimed or overwritten while the stream size (i.e., the number of retained records) is less than this guarantee. On the other hand, if the size of a stream exceeds its promise, only the most recent data is safeguarded, and any earlier records are deemed excessive. The total of guarantees must be less than the size of the storage system minus a fraction; we refer to the ratio of guarantees to volume size as volume utilisation.4 Similar to other file systems, the level of utilisation significantly impacts performance.

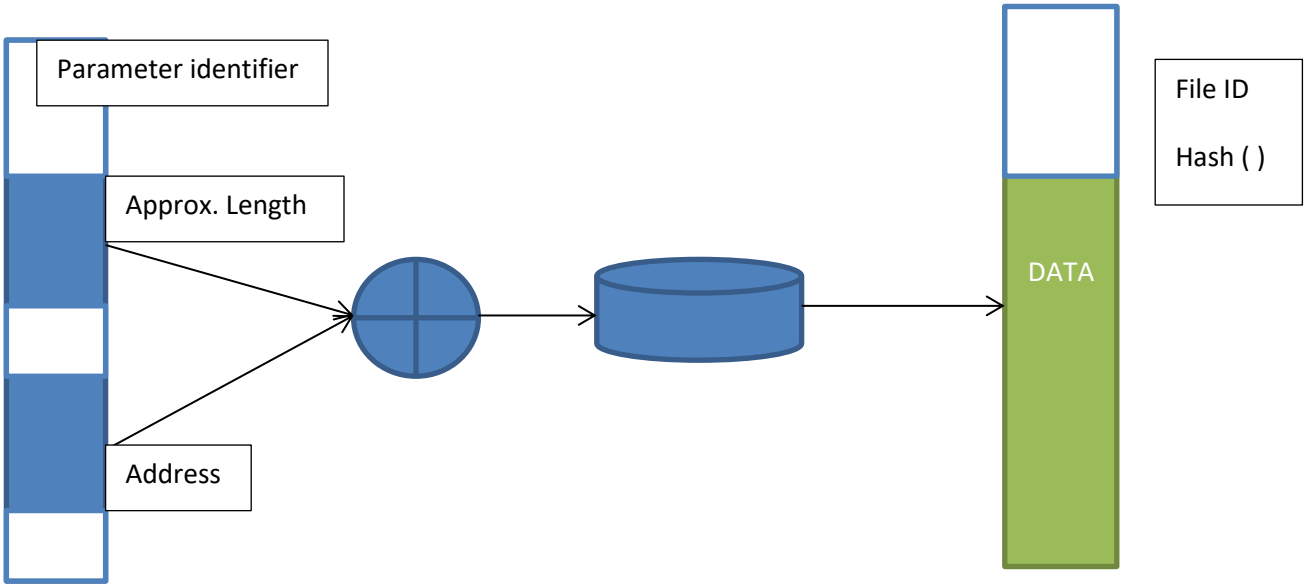

Figure 2. Read Operation

StreamFS circumvents the need for a segment cleaner by storing data in small fixed-length blocks, typically 1MB in size. Each block is dedicated to storing data from a single stream. As the writing border progresses, it is essential to ascertain whether the next block is redundant. If so, it is simply overwritten, as shown in Figure 4, and if not, it is skipped and expires later; no data copying or cleaning is required in any scenario. This provides a flexible storage allocation system that allows for both storage reserve and best-effort utilization of remaining storage. Simulation results have demonstrated that this "cleaning"

strategy performs exceptionally well, with essentially no throughput degradation for utilizations of 70% or less and no more than a 15% loss in throughput at 90% utilization [8].

3.1. Stream FS Organization

The record header utilized for self-certifying reads is a StreamFS on-disk data structure. The following lists these structures along with their fields and uses:

Each variable-length record created by the application corresponds to both an on-disk record and a record header. The header includes the aforementioned validation elements, as well as timestamp and length information.

- **Block:** Several records from the same stream are merged into a single, fixed-length block that has a default length of 1Mbyte. The block header identifies the stream to which the block belongs and records the block's boundaries.
- **Every Nth block** (default 256) serves as a block map, displaying the corresponding stream and in-stream sequence number for the previous N−1 blocks. This map is utilized for write allocation, to ascertain whether a block is included in a stream's assured allocation and should be bypassed, or if it can be overwritten.
- **File system root:** The root contains the stream directory, metadata for each stream (head and tail pointers, size, parameters), and a description of the devices that comprise the file system.

3.2. Indexing History

An Hyperion monitor must maintain an index that not only allows for efficient retrospective searches, but can also be produced quickly. Disk performance greatly limits the index's options; while avoiding random disk operations is a goal in any database, many fields must be indexed in records coming at rates of more than 100,000 per second per link. Hyperion depends on index structures that can be computed live and then stored immutably in order to scale to these rates. A stream is divided into intervals by Hyperion, which then computes one or more signatures [13] for each interval. It is possible to verify the signatures to see if a record with a specific key exists in the related data interval. Unlike a standard B-tree structure, a signature just tells if a record matching a specific key exists; it does not specify where in the interval that record is found. As a result, the complete period must be retrieved and examined to determine the outcome. However, if the key is not provided, the full interval may be skipped.

No stream-wide index is kept; instead, signature indices are calculated on an interval-by-interval basis. This resolves the issue of deleting keys from the index when they become outdated because the signature linked to a data interval also aged out.

A. Indices of Multi-level Signatures

Hyperion employs a multi-level signature index, the structure of which is seen in Figure 7. A signature index, the most well-known of which is the Bloom Filter [3], generates a compact signature for one or more records that can be used to determine whether a specific key exists in the linked records. (This is in contrast to a B-tree or traditional hash table, where the structure offers a map from a key to the position where the relevant record is kept.) To find records containing a specific key, we first retrieve and test the signatures; if any signatures match, the relevant records are retrieved and searched. Signature functions are often inaccurate and may provide false positives, indicating a match when there is none. When scanning the actual data records, this will be rectified; however, the signature function is unable to generate false negatives, as this would result in the omission of records. Search efficiency for these structures is a trade-off between signature compactness, which minimizes the quantity of data retrieved when scanning the index, and false positive rate, which causes unneeded data records to be retrieved and deleted.

B. Managing Search and Rank Inquiries

While signature indices are highly effective, similar to hash indices, their utility is limited to exact-match queries. Specifically, several query types—like range and rank (top-K) queries—that are helpful in network monitoring applications are not handled by them effectively. Hyperion can also use other functions to create indexes. These include interval bitmaps [26] and aggregation routines.

Interval bitmaps are a type of bitmap indices [4]. The domain of a variable is divided into b intervals, and a b-bit signature is formed by setting the one bit corresponding to the interval holding the variable's value. After these signatures are superimposed, a summary that indicates whether a value within a specific range is present in the collection of summarized records can be obtained.

Aggregate functions, such as min and max, can also be used as indexes; in this case, the aggregate is generated over a segment of data and saved as its signature. A query for $x<X0$ can employ aggregate minima to skip segments of data where no value would match, whereas a query for x with $COUNT(x)>N0$ can use an index specifying the top K values [17] in each segment and their counts.

C. Dispersed Index and Queries

So far, our discussion has concentrated on local data preservation and indexing on each node. In a typical network monitoring system, there will be several nodes, and dispersed requests must be handled to prevent query flooding. Hyperion uses a distributed index to provide an integrated view of data across all nodes, while storing the data and most index metadata locally on the node where it was generated. For performance reasons, local storage is prioritized because it uses less bandwidth than communication; archived data that may never be viewed is best stored locally.

To build this distributed index, a coarse-grained summary of the data stored at each node is required. The top level of the Hyperion multi-level index gives such a summary, which is shared by all nodes in the system. Because broadcasting the index to all other nodes will cause excessive bandwidth as the system scales, one index node is assigned to each time period [t1,t2). All nodes transmit their top-level indices to the index node during this time interval. Designating a different index node for each time interval produces a temporally scattered index. Cross-node searches are initially directed to an index node, which uses the coarse-grain index to find nodes with matching data; the query is then transferred to this subset for further processing.

IV. IMPLEMENTATION AND RESULTS

File system testing generated dummy data (zeros) and ignored data from read operations. But for most index tests, real trace data from the link between the University of Massachusetts and the public Internet [31] was used. By mixing the recorded headers—possibly modified—with fictitious data and sending the resulting packets straight to the system being tested, these trace files were replayed on a different machine.

Our initial trials create a standard against which to measure the Hyperion system's performance. Because Hyperion is an application-specific database, it is built upon an application-specific file system. We assess Hyperion's performance against current general-purpose versions of these components. We specifically test the speed of storing network traces on a traditional relational database and many traditional file systems as in table 1 ands 4.

Table. 1. Streaming write-only throughput by file system

	Throughput (10^6 Bytes / Sec)				
Time (Seconds)	XFS	NetBSD LFS	Ext3	JFS	Reiser
0	73	35	40	24	33

500	45	50	31	23	30
1000	54	45	35	22	20
1500	57	46	32	24	21
2000	48	39	38	21	19
2500	54	49	34	23	22

Table. 2. Streaming write-only throughput by file system

	Throughput (10^6 Bytes / Sec)	
Time (Seconds)	XFS	Stream FS
0	74	50
500	47	57
1000	57	55
1500	66	55
2000	68	56
2500	60	57

The tests were conducted on the main testing system, using only one data disk. Our gateway link traces were replayed across a gigabit cable to the test system. Initially, 4·108 packets, or 26GB of packet header data, were monitored for an hour in order to fill the database. Subsequently, packets were sent to the system being evaluated, using inter-arrival times derived from the original trace. However, these inter-arrival durations were adjusted to change the average arrival rate, while queries were sent simultaneously. We calculate packet loss by comparing the send count on the test system to the receive count on Hyperion and measuring CPU consumption.

Table 3 illustrates packet loss and free CPU time remaining as the packet arrival rate varies.Six Loss rates are represented on a logarithmic scale; however, the lowest points indicate that no packets were lost out of 30 or 40 million that were received. The findings indicate that Hyperion achieved a reception and indexing rate of more over 200,000 packets per second, with an insignificant amount of packet loss. Furthermore, the main constraint on resources seems to be the amount of CPU power available. This suggests that there is potential for achieving much better performance as CPU speeds increase.

Table. 3. Streaming write-only throughput by file system

	Packet Loss Rate	
Packets (Seconds)	Loss Rate	Idle CPU
0	0.000011	0.000100
100000	0.0000025	0.000055
150000	0.000022	0.000033
200000	0.001	0.000015
250000	0.1	0.000005

V. CONCLUSION

In this study, we claim that neither general-purpose file systems nor standard database index architectures can handle the specific requirements of high-volume stream archiving and indexing. StreamFS, a write-optimized stream file system, and a multi-level signature index that can manage high update rates and facilitate distributed querying make up our new stream archiving system, Hyperion, which we presented. Our prototype evaluation revealed that

(i) StreamFS can scale to write loads of over a million packets per second,

(ii) The index can support over 200K packet/s while maintaining decent query performance for interactive use, and

(iii) Our system can grow to tens of monitors.

Future work will include improving StreamFS's aging mechanisms and implementing new index types to facilitate better querying.

References:

[1]. Desnoyers, P., and Shenoy, P. Hyperion: High Volume Stream Archival for Retrospective Querying. Tech. Rep. TR46-06, University of Massachusetts, Sept. 2006.

[2]. Dreger, H., Feldmann, A., Paxson, V., and Sommer, R. Operational experiences with high-volume network intrusion detection. In Proc. 11th ACM Conf. on Computer and communications security (CCS '04) (2004), pp. 2–11.

[3]. Endace Inc. Endace DAG4.3GE Network Monitoring Card. available at https://www.endace.com, 2006.

[4]. Faloutsos, C. Signature-Based Text Retrieval Methods: A Survey. IEEE Data Engineering Bulletin 13, 1 (1990), 25–32.

[5]. Faloutsos, C., and Chan, R. Fast Text Access Methods for Optical and Large Magnetic Disks: Designs and Performance Comparison. In VLDB '88: Proc. 14th Intl. Conf. on Very Large Data Bases (1988), pp. 280–293.

[6]. Abadi, D. J., Ahmad, Y., Balazinska, M., Çetintemel, U., Cherniack, M., Hwang, J.-H., Lindner, W., Maskey, A. S., Rasin, A., Ryvkina, E., Tatbul, N., Xing, Y., and Zdoni, S. The Design of the Borealis Stream Processing Engine. In Proc. Conf. on Innovative Data Systems Research (Jan. 2005).

[7]. Bernstein, D. Syn cookies. Published at https://cr.yp.to/syncookies.html.

[8]. Bloom, B. Space/time tradeoffs in hash coding with allowable errors. Communications of the ACM 13, 7 (July 1970), 422–426.

[9]. Chan, C.-Y., and Ioannidis, Y. E. Bitmap index design and evaluation. In Proc. ACM SIGMOD Intl. Conf. on Management of Data (June 1998), pp. 355–366.

[10]. Cheriton, D. The V Distributed System. Communications of the ACM 31, 3 (Mar. 1988), 314–333.

[11]. Cranor, C., Johnson, T., Spataschek, O., and Shkapenyuk, V. Gigascope: a stream database for network applications. In Proc. 2003 ACM SIGMOD Intl. Conf. on Management of data (2003), pp. 647–651.

[12]. Desnoyers, P., Ganesan, D., and Shenoy, P. TSAR: a two tier sensor storage architecture using interval skip graphs. In Proc. 3rd Intl. Conf on Embedded networked sensor systems (SenSys05) (2005), pp. 39–50.

[13]. Faloutsos, C., and Christodoulakis, S. Signature files: an access method for documents and its analytical performance evaluation. ACM Trans. Inf. Syst. 2, 4 (1984), 267–288.

[14]. Ganesan, D., Estrin, D., and Heidemann, J. Dimensions: Distributed multi-resolution storage and mining of networked sensors. ACM Computer Communication Review 33, 1 (January 2003), 143–148.

[15]. Huebsch, R., Chun, B., Hellerstein, J. M., Loo, B. T., Maniatis, P., Roscoe, T., Shenker, S., Stoica, I., and Yumerefendi, A. R. The Architecture of PIER: an Internet-Scale Query Processor. In Proc. Conf. on Innovative Data Systems Research (CIDR) (Jan. 2005).

[16]. Iannaccone, G., Diot, C., McAuley, D., Moore, A., Pratt, I., and Rizzo, L. The CoMo White Paper. Tech. Rep. IRC-TR-04-17, Intel Research, Sept. 2004.

[17]. Karp, R. M., Shenker, S., and Papadimitriou, C. H. A simple algorithm for finding frequent elements in streams and bags. ACM Trans. Database Syst. 28, 1 (2003), 51–55.

[18]. Li, X., Bian, F., Zhang, H., Diot, C., Govindan, R., Hong, W., and Iannaccone, G. Advanced Indexing Techniques for Wide-Area Network Monitoring. In Proc. 1st IEEE Intl. Workshop on Networking Meets Databases (NetDB) (2005).

[19]. Moore, A., Hall, J., Kreibich, C., Harris, E., and Pratt, I. Architecture of a Network Monitor. Passive & Active Measurement Workshop 2003 (PAM2003) (2003).

[20]. Motwani, R., Widom, J., Arasu, A., Babcock, B., Babu, S., Datar, M., Maku, G., Olston, C., Rosenstein, J., and Varma, R. Query processing, approximation, and resource management in a data stream management system. In Proc. Conf. on Innovative Data Systems Research (CIDR) (2003).

[21]. Sweeney, A., Doucette, D., Hu, W., Anderson, C., Nishimoto, M., and Peck, G. Scalability in the XFS File System. In USENIX Annual Technical Conference (Jan. 1996).

[22]. Tobagi, F. A., Pang, J., Baird, R., and Gang, M. Streaming RAID: a disk array management system for video files. In Proc. 1st ACM Intl. Conf. on Multimedia (1993), pp. 393–400.

[23]. Umass trace repository. Available at https://traces.cs.umass.edu.

[24]. Wang, W., Zhao, Y., and Bunt, R. HyLog: A High Performance Approach to Managing Disk Layout. In Proc. 3rd USENIX Conf. on File and Storage Technologies (FAST) (2004), pp. 145–158.

[25]. Ndiaye, B., Nie, X., Pathak, U., and Susairaj, M. A Quantitative Comparison between Raw Devices and File Systems for implementing Oracle Databases. https://www.oracle.com/technology/deploy/performance/WhitePapers.html, Apr. 2004.

[26]. Rosenblum, M., and Ousterhout, J. K. The design and implementation of a log-structured file system. ACM Transactions on Computer Systems 10, 1 (1992), 26–52.

[27]. Sacks-Davis, R., and Ramamohanarao, K. A two level superimposed coding scheme for partial match retrieval. Information Systems 8, 4 (1983), 273–289.

[28]. Seltzer, M. I., Smith, K. A., Balakrishnan, H., Chang, J., McMains, S., and Padmanabhan, V. N. File System Logging versus Clustering: A Performance Comparison. In USENIX Winter Technical Conference (1995), pp. 249–264.

[29]. Shriver, E., Gabber, E., Huang, L., and Stein, C. A. Storage management for web proxies. In Proc. USENIX Annual Technical Conference (2001), pp. 203–216.

[30]. Stockinger, K. Design and Implementation of Bitmap Indices for Scientific Data. In Intl. Database Engineering & Applications Symposium (July 2001).

[31]. Stonebraker, M., Cetintemel, U., and Zdonik, S. The 8 requirements of real-time stream processing. SIGMOD Record 34, 4 (2005), 42–47.

[32]. StreamBase, I. StreamBase: Real-Time, Low Latency Data Processing with a Stream Processing Engine. from https://www.streambase.com, 2006.

[33]. Sudeesh Goriparthi. Optimizing search functionality: A performance comparison between solr and elasticsearch. International Journal of Data Analytics (IJDA), 3(1), 2023, pp. 67-78.

[34]. Sudeesh Goriparthi. Tracing data lineage with generative AI: improving data transparency and compliance. International Journal of Artificial Intelligence & Machine Learning (IJAIML), 2(1), 2023, pp. 155-165.

[35]. Rahul Kalva. Leveraging Generative AI for Advanced Cybersecurity Enhancing Threat Detection and Mitigation in Healthcare Systems, European Journal of Advances in Engineering and Technology, v. 10, n. 9, p. 113-119, 2023.

[36]. Ankush Reddy Sugureddy. Enhancing data governance and privacy AI solutions for lineage and compliance with CCPA, GDPR. International Journal of Artificial Intelligence & Machine Learning (IJAIML), 2(1), 2023, pp. 166-180

[37]. Ankush Reddy Sugureddy. AI-driven solutions for robust data governance: A focus on generative ai applications. International Journal of Data Analytics (IJDA), 3(1), 2023, pp. 79-89

[38]. WADITWAR, P. The Intersection of Strategic Sourcing and Artificial Intelligence: A Paradigm Shift for Modern Organizations. Open Journal of Business and Management, v. 12, n. 6, p. 4073-4085, 2024.

[39]. S. C. Patil, B. Y. Kasula, V. A. Mohammed, K. Gupta and T. Thamaraimanalan, "Utilizing Genetic Algorithms for Detecting Congenital Heart Defects," 2024 International Conference on E-mobility, Power Control and Smart Systems (ICEMPS), Thiruvananthapuram, India, 2024, pp. 1-6, doi: 10.1109/ICEMPS60684.2024.10559358.

[40]. Rahul Kalva. Integrating DevOps and Large Language Model Operations (LLMOps) for GenAIEnabled E-commerce Innovations A Pathway to Intelligent Automation, World Journal of Advanced Research and Reviews, v. 24, n. 03, p. 879-889, 2024.

[41]. S. C. Patil, P. Takkalapally, B. Y. Kasula and J. Logeshwaran, "An improved AI-driven Data Analytics model for Modern Healthcare Environment," 2024 15th International Conference on Computing Communication and Networking Technologies (ICCCNT), Kamand, India, 2024, pp. 1-6, doi: 10.1109/ICCCNT61001.2024.10724303.

[42]. Ankush Reddy Sugureddy. Data governance excellence in the cloud leveraging GCP for enhanced lineage and security. International Journal of Data Science and Analytics (IJDSA), 2(2), 2024, pp. 8-19