Scalable Storage Support for Data Stream Processing

Zoe Sebepou and Kostas Magoutis
Institute of Computer Science (ICS)
Foundation for Research and Technology – Hellas (FORTH)
N. Plastira 100, Heraklion, GR-70013, Greece
{sebepou,magoutis}@ics.forth.gr

Abstract— **Continuous data stream processing systems have offered limited support for data persistence in the past, for three main reasons: First, online, real-time queries examine current streaming data and (under the assumption of no server failures) do not require access to past data; second, stable storage devices are commonly thought to be constraining system throughput and response times when compared to main memory, and are thus kept off the common path; finally, the use of scalable storage solutions which would be required to sustain high data streaming rates have not been thoroughly investigated in the past. Our work advances the state of the art by providing data streaming systems with a scalable path to persistent storage. This path has low impact in the performance properties of a scalable streaming system and allows two fundamental enhancements to their capabilities: First, it allows stream persistence for reference/archival purposes (in other words, queries can now be applied on past data on-demand); second, fault tolerance is achievable by checkpointing and stream replay schemes that are not constrained by the size of main memory.**

Keywords; scalable storage systems; data streaming; fault-tolerance.

I. INTRODUCTION

Data sources that continuously produce data are abundant in today's information-driven society, ranging from civilian and military surveillance devices [1], environmental sensors, mobile telephony base stations, network devices, stock market price monitors, and credit-card points of sale. A number of research projects [5][6][7][11][12] have investigated ways to achieve high-performance, online data stream processing. Some of these projects have led to business ventures [2][3][4] underlining the significant demand for this technology in today's society.

Modern streaming systems are designed for complex event processing (CEP) expressed in special stream-oriented query languages [5][6][7]. The data operated on are typically streams of records often referred to as *tuples*. In most common applications tuples are live— that is, recently produced— and associated with a monotonically increasing timestamp. In such applications, tuple processing is also associated with timeliness guarantees— that is, a response or trigger must be produced within a short time bound from the time a tuple is produced by the data source. However new types of applications such as fault-tolerant streaming (for example, for processing financial transactions), offline data-warehouse style processing of streaming data, and verification of compliance or violation of service-level agreements (SLAs), have created interest to persisting streams for archiving/reference, replay, and/or post-facto introspection.

Sources of continuously-produced information today are growing in both number and data rates produced. As one example, consider a mobile operator that needs to process an increasingly growing rate of call-detail-records per second as its operations expand in both geographical coverage (number of base stations) and client base (number of calls initiated per unit time). Such trends call for scalable streaming platforms that can keep up with data growth both in terms of capacity and I/O throughput. Scalable storage systems for such platforms should be designed, configured, and tuned for the specific characteristics of continuous data streaming.

From the storage system perspective, the persistence of continuous data streams exhibits two main distinctive characteristics. First, streaming workloads consist largely of sequential writes/appends, with sequential reads taking place mostly during recovery or in retrospective queries. Whereas the read use-cases are certainly highly valuable, they are not expected to be the norm in actual deployments. Second, streaming workloads feature multiple concurrent writers on exclusively-owned or shared storage objects. The former case is expected to be prevalent in actual deployments and we therefore focus on it in this study.

Continuous data streaming workloads present a number of challenges for the storage system designer. The storage system should be optimized for both latency and throughput since scalable data stream processing systems must handle both heavy stream traffic *and* produce results to queries within given time bounds, typically tens to a few hundreds of milliseconds. This means that the storage system should be designed to exhibit a large degree of parallelism for scalability and also to minimize the eventuality of high-delay events on the common path. Another important challenge is to guarantee data stability under different failure assumptions.

In this paper we focus on scalable storage support for data stream processing systems for the types of applications described above. Our contributions are summarized here:

1. Design of a high-performance stream processing engine I/O architecture that allows simultaneous persistence and communication of live and past (retrieved from storage) data streams.

2. Implementation of the architecture in the context of an open-source streaming middleware (Borealis [7]).

3. Evaluation of the scalability of the architecture using an open-source scalable storage (PVFS2 [18]).

We find that our design and implementation of the persistence path enables rich functionality with low impact on the performance properties of the stream processing system. In summary, our results show that the throughput of the persistence path when implemented with appropriate tuning over PVFS2 scales with increasing load. In addition, it adds a reasonable delay (about 20ms) per processing element in a well-provisioned system.

Our experience with implementing persistence of concurrent streams efficiently over a scalable filesystem such as PVFS2 provides the following observations:

- Synchronous filesystem metadata operations should be taken off the critical path, and if necessary replaced by asynchronous operations or removed altogether. Local filesystem metadata and/or self-identifying stream records can be leveraged to achieve consistency after a failure.
- The latency of disk-synchronous (stable) writes can be reduced by increasing parallelism (stripe width) in file I/O operations.
- Concurrent streams multiplexed on parallel filesystem servers produce I/O patterns that are not handled well by most general-purpose local filesystems. A log-structured or extent-based local filesystem is best suited for such patterns.

The remainder of this paper is structured as follows: First, we relate our work to previous research on stream processing systems. Then we discuss our persistence architecture, highlighting the key factors expected to affect performance, and describe the storage systems used in this study. Finally we describe our experimental testbed and present our results on the performance properties of our system.

II. Related work

Traditional data streaming systems have offered limited support for stream persistence in the past. Systems such as Aurora/Borealis [5] and the Stanford STREAM Data Manager [7] have considered the need for persisting streams but have mainly focused on simple database interfaces (for example, to implement *connection points* in Aurora/Borealis [6]) or algorithmic studies of disk buffering policies [8].

Previous work on streaming data persistence was also motivated by applications such as network monitoring for the collection of network statistics, troubleshooting, and forensics. Hyperion [11] is one such system that proposes a log-based filesystem (LFS)-derived solution called StreamFS, which optimizes archival, indexing, and online retrieval of multiple data streams. While related in spirit and motivation to our work, to the best of our knowledge the Hyperion approach has not been studied in a continuous-query streaming environment and has not been extended over scalable storage platforms.

High-availability (HA) solutions that adapt process-pairs approaches to data streaming have been proposed in the past. In the work of Hwang *et al.* [9] upstream nodes retain tuples in memory buffers until receiving an explicit acknowledgment from downstream nodes that they have checkpointed the associated state and transferred it to a passive or active backup. This approach may result in tuple loss if upstream nodes run out of memory buffers while the recovery of failed downstream nodes takes longer than expected. It also results in underutilization of the overall system memory since part of it is dedicated to holding duplicate checkpoint state.

More recent research work on stream fault-tolerance such as Kwon *et al.* [10], have experimented with scalable storage solutions but have not examined their scalability properties in the context and detail of this paper. Kwon *et al.* proposed a rollback-recovery scheme using asynchronous checkpointing (implemented in Borealis [5]) with checkpoints stored in a distributed and replicated filesystem (their implementation uses HDFS [19]). Their work is related to ours in its focus on persisting operator state on scalable, fault-tolerant storage. Their specific focus however is on scheduling periodic concurrent checkpoints, whereas our focus is on the more general case of continuously streaming tuples (which includes checkpoints as a special case).

Another recent related project is that of Hilley and Ramachandran [12], which focuses on a programming abstraction that integrates transport, manipulation, and storage of streaming data. Their work is related to ours in its goal for seamless integration of different data paths at the programming-abstraction level through a common API. However, their system does not focus or optimize for storage scalability and has not been thoroughly investigated as such.

Streaming-type workloads with large files that are mostly appended to and read sequentially are also typical of Web search and Map-Reduce [14] type processing. The importance of such workloads has led to the design of systems such as the Google File System (GFS) [15]. GFS has been optimized for high throughput but makes no provision for reducing latency (for example writing a 64MB chunk to a GFS chunkserver can take about a second on a 1Gbps network). GFS is thus not immediately suitable for continuous stream processing applications.

Finally, multimedia servers [17] are similar to data stream storage systems in that they handle sequential access patterns to the underlying data. Research on multimedia servers mostly during the 90's has put emphasis on quality of service (QoS), disk scheduling, and optimizations of the read data path. Continuous stream processing applications exhibit a significant concurrent write activity and thus present a different set of challenges for the storage system.

III. Persistence Architecture

Data flow in traditional stream processing systems takes place by communicating tuples produced and consumed by operators over data paths called *streams* [5][7]. Each operator is associated with a distinct pair of input and output queues. Operators are deployed within stream processing engines (SPE) on different nodes. An SPE is akin to a virtual machine specialized to providing runtime support for the execution of stream operators [6]. The data path between operators is commonly implemented through communication between memory buffers over the network. However, in systems where data loss is not acceptable, SPEs should be able to persist tuples so that they can be later retrieved from arbitrary points in time.

A key concept in our scalable stream persistence architecture is that of a *persistent stream object* (PSO), a persistent image of the state of a queue Q over time T ($t_{\text{most recent}} - t_{\text{least recent}}$). A queue (and thus a PSO) is associated with a schema that describes the structure of its tuples– for example, a telephone call-detail record (CDR). PSOs are characterized by logical contiguity. Their physical representation may take different forms, such as a single storage object or a set of storage objects, and may be hosted in different types of containers, such as a RAID array or a distributed filesystem. A PSO is identified through an assigned ID that reflects the context of the associated queue Q: its schema, the operator it connects to, the position of this operator within the overall operator graph, and the description of the graph.

A. SPE I/O architecture

Each SPE serves multiple streams connecting pairs of queues, and may therefore be persisting on multiple PSOs. The data path between operator queues and PSOs extends the typical SPE structure such as found in the Aurora/Borealis system [5][6]. Incoming tuples from a stream *S* are shepherded by a thread (referred to as the Enqueue thread) to be enqueued into the SPE for processing by operator(s) that use *S* as one of their inputs. The tuples produced by the operators are placed on an output stream and dequeued by a separate thread (referred to as the Dequeue thread) and grouped into Stream Events (SEs). SEs are serialized objects grouping several tuples for the purpose of efficient communication. Our detailed description of the SPE I/O data path starts with the case of failure-free operation:

Failure-free operation: SEs created by the dequeue thread are first serialized. If the streams they are associated with are set for persistence, the SEs enter the persist-event list, otherwise they move directly onto the forward-event list. A write operation to storage is initiated using an asynchronous API [13]. This write is typically stable; that is, the write is not complete unless the I/O has been flushed to disk— however the architecture can accommodate different semantics and tradeoffs. The asynchronous I/O operations are handled by a state machine in an event loop. For parallelism, we maintain a configurable window of *N* concurrently outstanding I/Os. Once a completion of a write I/O is posted by the storage system we first update a per-stream index, and then move the persisted event data structure to the forward-event list. Subsequently a network send operation is initiated. The SE remains there until successfully sent out over the network.

Operation under failure: When a downstream SPE node fails, all streams connected to queues on that node disconnect and no outgoing network communication takes place on those streams until reconnection (other streams however are not affected). SEs produced by local operators are still persisted as described during failure-free operation. However, in this case as soon as such SEs are stable they are deleted from memory. Other SEs belonging to still-connected streams proceed to the forward-event list as described during failure-free operation.

PSO indexing: As PSOs grow by appending serialized SEs to them, our I/O architecture maintains per-PSO indices mapping a given tuple identifier (a timestamp) into a serialized SE within a PSO. In the case of a filesystem implementation, the index points to a file offset where the SE containing the tuple requested is located. In our current prototype the PSO index is implemented using an Oracle Berkeley DB database.

B. Storage system used in this study

In this work we use the PVFS2 [18] clustered parallel filesystem. PVFS2 stripes files over a cluster of storage servers offering parallel I/O paths for scalable I/O performance. PVFS2 decouples metadata from data accesses, an approach first popularized by the NASD project [16]. One or more metadata servers are responsible for informing clients of the location of data in the storage servers but are not involved in the actual I/O operation. PVFS2 does not offer client-side caching, a feature that fits well workloads such as streaming with little or no data re-use.

In Figure 1 we depict a (one of possibly several) SPE performing I/O asynchronously. Possible metadata updates during I/O are handled by the metadata server. Each storage (data) node maintains parts of several PSOs, each part identified by a pair (X, A) that stands for "corresponding to SPE X stored on data node A". Data layout to storage servers in PVFS2 is done in a round-robin manner independent of their actual load. Despite the simplicity of this policy it should perform reasonably well for sequential stream accesses.

We next focus on four important factors affecting performance.

Metadata updates. In addition to I/O operations, a write operation may trigger a metadata update as a result of a change in the file size and/or the time of last modification or access. A metadata update may not be needed if overwriting a pre-existing file and maintenance of the last-modified or last-accessed timestamps has been disabled. However in cases where metadata updates are necessary, there are two implementation choices: The first is to synchronously flush the metadata update to disk ensuring durability. The second choice is to enable write-back caching at the metadata server, which should reduce durability but improve performance.

Data stream workloads make the latter a viable alternative with the use of self-identifying tuples and when each data node runs a local filesystem that maintains its own metadata about local files. In such case, recovery of the file size or the time of last modification or access is possible by examination of the file contents and recovery of local filesystem metadata in storage nodes. For this reason we can safely use asynchronous metadata updates in our experiments.

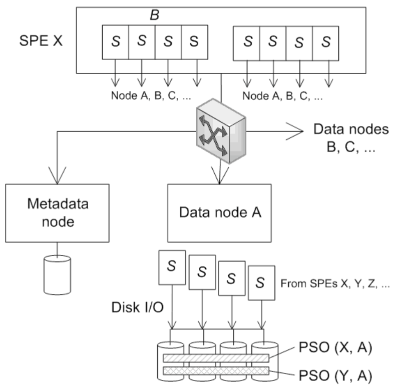

Figure 1: Model of stream persistence over PVFS2.

Data stability: For fault-tolerance, an SPE requires that tuples are stably written to disk prior to forwarding them on the network. Stable writes involve the disks on each I/O operation and are thus expected to affect response time. Throughput is expected to be affected to the degree of available parallelism (*e.g.*, number of servers, number of writing threads per server, and number of disks per server involved) to avoid blocking on any such operation.

Local filesystem: Concurrent writes from multiple SPEs interfere at storage servers. Within each storage server, write requests targeting different local files may reduce storage node efficiency if the local filesystem cannot handle such patterns very well. Log-based filesystems (LFS) or extent-based filesystems are particularly suitable for such patterns. In many cases though, distributed file systems use general-purpose filesystems that are not optimized for the concurrent sequential writes to large files typical of data streaming workloads.

Block size (*S*). Using a large *S* improves the efficiency of data transfers, which is one of the reasons GFS uses a 64MB chunk. However a large *S* is not advantageous in a latency-sensitive environment due to the high transfer time. In our experimental evaluation we use a moderate *S* value of 256KB.

IV. EXPERIMENTAL TESTBED

Our experimental setup consists of a 16-node cluster of dual-CPU AMD 244 servers with 2GB DRAM running Linux 2.6.18 and connected through a 1Gbps Ethernet switch using Jumbo (9000 byte) frames. In this cluster we deployed the Borealis software release (as of summer 2008) and PVFS2 version 2.8.1. For selected experiments requiring more powerful servers we used dual quad-core (total of 8 cores) AMD Opteron 2354 with 4GB of DRAM. Unless explicitly mentioned otherwise we use the dual-CPU machines in our experiments.

In our experimental setup half of the PVFS2 nodes were configured as file servers—one of them doubling as metadata server— and the rest as clients. Each node of each filesystem was provisioned with a dedicated logical volume comprising four 40GB partitions of SATA disks in a RAID-0 configuration with a 64KB stripe unit. PVFS2 was setup to stripe files using a 256KB unit, a value that was chosen to equal a full RAID stripe. The total capacity of the parallel filesystem in the 8-server setup was about 1.1TB. Each file server node used an underlying Linux filesystem of type ext3 or xfs where noted. By default, PVFS2 uses a write-back data cache on its servers. In our experiments we modified the default policy to use synchronous writes (unless explicitly mentioned otherwise).

V. RESULTS

In this section we report results comparing Borealis performance with and without persistence over PVFS2. We use performance without the persistence path as a baseline.

A. Baseline performance: Single node

Table 1 describes streaming throughput in a Borealis setup with a single SPE, hosting a filter operator connected to a data source (sender) and a data sink (receiver) node. The filter operator inspects a fixed-size (integer) field in each tuple and forwards it along the operator's single output stream. We chose a simple operator setup to focus on I/O performance rather than computational behavior of the streaming system. Throughput is in MB/sec and the tuple size varies from 256 bytes to 4KB.

	Batch size			
Tuple size	1	16	32	64
256	5 MB/sec	15 MB/sec	13 MB/sec	14 MB/sec
1024	14 MB/sec	38 MB/sec	35 MB/sec	34 MB/sec
4096	31 MB/sec	63 MB/sec	52 MB/sec	45 MB/sec

Table 1 Baseline measurements with dual-core nodes.

An important Borealis parameter is the *batching factor* at the source, which is the amount of tuples it injects into Borealis with each network I/O. The batching factor refers to the source node and does not constrain the Borealis servers into how they group tuples in performing I/O operations. We experimented with batching factors of 1, 16, 32 and 64 tuples. Borealis servers use an adaptive grouping factor, which is inversely proportional to tuple size.

In Table 1 we observe that Borealis performance improves with increasing tuple size due to gradually reduced per-tuple CPU overhead. The figure also indicates that tuple batching helps performance to a certain degree. For a batching factor of 16 and tuple size 4KB we observe a throughput of 63MB/sec which exceeds no-batching performance by a factor of two. This represents the highest throughput rate we have achieved using the dual-core server setup. All CPUs are fully saturated in all measurements taken in this benchmark.

B. *Storage performance: Single node*

We next evaluate the performance of the single SPE node of the previous setup with the persistence path enabled. We use two storage configurations: A single PVFS2 server performing synchronous writes (Pvfs2-sync) and one that does not (Pvfs2-async). We also evaluate a configuration (Str_Code) that includes the persistence path but omits calls into the PVFS2 client. Finally, we compare the above to baseline Borealis measurements.

We use a batching factor of 16 to target the best performing configuration of the previous experiment. We have empirically determined that Borealis performs I/O (network or storage) at steady-state using fixed-size blocks of 184 tuples for the tuple sizes we have considered. The Borealis I/O size is thus 64KB, 184KB, and 736KB for tuple sizes 256 bytes, 1KB, and 4KB, respectively.

In Figure 2 we observe that the differences between configurations become progressively more pronounced with increasing tuple size as the system becomes more I/O-bound. At 4KB tuple size, we observe that Pvfs2-async performance is about half that of non-persisting Borealis due to splitting the outgoing link bandwidth between persistence and regular communication traffic.

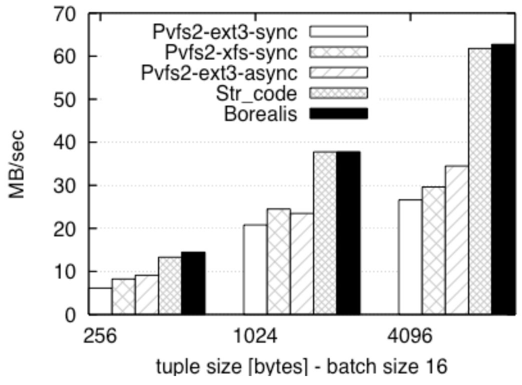

Figure 2 Storage performance with a single Borealis node.

Synchronous (stable) writes reduce PVFS2 performance by about 8MB/s or 15% compared to asynchronous writes. The majority of this overhead seems to be coming from the file/storage client itself as the Str_Code configuration indicates. The CPU of the SPE node is fully saturated in all cases in this experiment. Finally, measurements of I/O response time in the Pvfs2-sync setup report an average of 20ms.

To evaluate the impact of a faster SPE node we repeat this experiment for the Pvfs2-sync, and baseline configurations with 4KB tuple size and batching factor 16 on the 8-core server. In Table 2 we observe that all configurations benefit from running the SPE on a faster CPU. However, PVFS2 seems to receive a proportionally higher benefit improving its performance by a factor of two. We also observe that the PVFS2 configuration achieves more than half the throughput of the baseline configuration despite moving every tuple twice over the outgoing link. The reason for this is the additional parallelism within the PVFS2 client which can use the spare CPU cores (which the single network worker thread itself cannot use) to utilize a higher fraction of the outgoing link bandwidth.

	Borealis	PVFS2
2-core	63 MB/sec	28 MB/sec
8-core	75 MB/sec	50 MB/sec

Table 2 Baseline storage measurements with 8-core nodes.

C. *Streaming and Storage Scalability*

In this experiment we vary the number of Borealis SPE nodes and the number of PVFS2 storage servers performing synchronous/stable writes. Each Borealis SPE node is hosting a filter operator (similar to previous experiments) and all operators are connected (input to output) via a single stream.

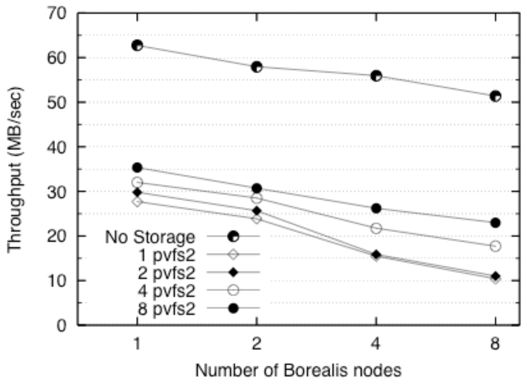

Figure 3 Scalability of Borealis over PVFS2.

Figure 3 depicts streaming performance with and without persistence when the Borealis and PVFS2 nodes vary from one to eight. We use a tuple size of 4KB and batch size of 16. An initial observation in the case of non-persisting Borealis nodes is that increasing their number from 1 to 8 results in a throughput drop from 62MB/s to about 52MB/s. We attribute this non-storage-related drop to imbalances in resource usage across nodes over the course of experiments in the chain of SPEs.

For a single Borealis SPE we observe that (in accordance with Table 1) throughput is less that half of what is achievable without persistence and improves to exactly half with larger PVFS2 setups. The reason behind the improvement is better parallelization/overlap of the stable write operations. Increasing the load (number of Borealis SPE nodes) we observe a gradual drop in throughput that is partly attributable to the Borealis issues described earlier and partly (where the drop is steeper) to overload in smaller PVFS2 setups. In the case of 8 Borealis SPEs on a single

PVFS2 server we observe a streaming throughput of about 10 MB/s or a PVFS2 server write-throughput of about 80 MB/s. Given that the server is not limited by either its network link or its CPU and the Borealis SPE nodes are not CPU-bound, we believe that in this case we are limited by the efficiency of the PVFS2 server local filesystem under many concurrent writers.

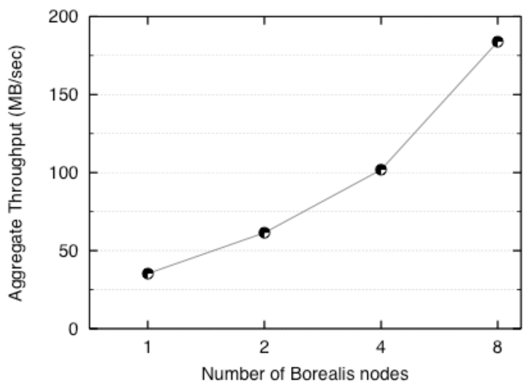

Figure 4 Aggregate throughput of Borealis over PVFS2.

Figure 4 provides a view of the aggregate throughput (calculated as the number of Borealis SPE nodes multiplied by SPE throughput towards the PVFS2 system) with increasing Borealis nodes. The figure shows that performance scales with increasing load. The minimum number of PVFS2 nodes needed to achieve the observed aggregate throughput with 1, 2, 4, 8 Borealis nodes is 8.

VI. Discussion and Conclusions

In this paper we have shown that stream persistence can be achieved in a scalable manner using a fully asynchronous event-driven SPE I/O architecture layered over an appropriately tuned parallel filesystem. This persistence path comes with an associated overhead: It requires the use of CPU and network bandwidth (as tuples to be communicated downstream are duplicated on the storage path). However, we have shown that the CPU overhead need not be high and it can be assigned to spare cores in multi-core CPUs; in addition, the network bandwidth may be needed mostly when there is spare bandwidth available, for example when downstream operators are overloaded (and thus cannot communicate at full speed) or have failed.

In our evaluation we have considered scenarios of several SPE nodes persisting tuples at full speed. In a practical scenario however, persistence may be required only at selected tuple *entry points*, which are edge servers that inject tuples into SPE nodes. Entry points are I/O-intensive and thus better candidates than SPEs for serving as persistence gateways. Although we exhibited our persistence path implementation in the context of the Borealis system, our SPE I/O architecture is more general and can be applied to other data stream processing systems.

VII. Acknowledgments

We thankfully acknowledge the support of the European FP7-ICT program through the STREAM project (STREP 216181).

References

[1] Christopher Drew, *"Military is Awash in Data from Drones"*, New York Times, January 10, 2010.

[2] StreamBase, http://www.streambase.com

[3] IBM Press Report, *"IBM Ushers In Era Of Stream Computing"*, http://www-03.ibm.com/press/us/en/pressrelease/27508.wss

[4] Ashlee Vance, *"IBM Unveils Real-Time Software to Find Trends in Vast Data Sets"*, New York Times, May 20, 2009.

[5] U. Cetintemel, D. Abadi, Y. Ahmad, H. Balakrishnan, M. Balazinska, M. Cherniack, J. Hwang, W. Lindner, S. Madden, A. Maskey, A. Rasin, E. Ryvkina, M. Stonebraker, N. Tatbul, Y. Xing, S. Zdonik; *"The Aurora and Borealis Stream Processing Engines"*, in Data Stream Management: Processing High-Speed Data Streams, M. Garofalakis, J. Gehrke, R. Rastogi (editors), Springer-Verlag, July 2006.

[6] D. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, S. Zdonik; *"Aurora: A New Model and Architecture for Data Stream Management*, in Proceedings of the VLDB Journal (2003).

[7] A. Arasu et al; "*STREAM: The Stanford Data Stream Management System",* in Data Stream Management: Processing High-Speed Data Streams, M. Garofalakis, J. Gehrke, R. Rastogi (editors), Springer-Verlag, July 2006.

[8] R. Motwani, D. Thomas, *"Caching Queues in Memory Buffers",* in Proceedings of the 15th Annual ACM-SIAM symposium on Discrete algorithms, New Orleans, LA, 2004.

[9] J.-H. Hwang, M. Balazinka, A. Rasin, U. Cetintemel, M. Stonebraker, S. Zdonik, *"High-Availability Algorithms for Distributed Stream Processing"*, in Proceedings of 21st International Conference on Data Engineering (ICDE'05), 5-8 April *2005*, Tokyo, Japan.

[10] S. Kwon, M. Balazinka, A. Greenberg, *"Fault-tolerant stream processing using a distributed, replicated filesystem",* in Proceedings of the VLDB Endowment (1): 574-585, 2008.

[11] P. Desnoyers, P. Shenoy, *"Hyperion: High Volume Stream Archival for Retrospective Querying",* in Proceedings of USENIX Annual Technical Conference, Santa Clara, CA, 2007.

[12] D. Hilley, U. Ramachandran, *"Persistent Temporal Streams"*, in Proceedings of ACM Middleware, 2009.

[13] Kernel asynchronous I/O for Linux, http://lse.sourceforge.net/io/aio.html

[14] J. Dean, S. Ghemawat, *"Map-Reduce: Simplified Data Processing on Large Clusters"*, in Proceedings of OSDI'04: Sixth Symposium on Operating System Design and Implementation, San Francisco, CA, December, 2004.

[15] S. Ghemawat, H. Gobioff, S. T. Leung, *"The Google File System"*, in Proceedings of the 19th ACM Symposium on Operating Systems Principles, Lake George, NY, October, 2003.

[16] G. Gibson et al, *"A Cost-Effective High-Bandwidth Storage Architecture"*, in Proceedings of the 8th Conference on Archiectural Support fro Programming Languages and Operating Systems (ASPLOS), San Jose, CA, October 1998.

[17] D. Gemmell, H. Vin, D. Kandlur, P. Rangan, *"Multimedia Storage Servers: A Tutorial and Survey"*, in IEEE Computer, vol. 28, pp. 40-49, 1995.

[18] W. Ligon, R. Ross, *"Overview of the Parallel Virtual File System"*, in Proceedings of Extreme Linux Workshop (1999).

[19] Apache Hadoop Project, *"The Hadoop Distributed File System: Architecture and Design"*, http://hadoop.apache.org, 2007.