Stormy: An Elastic and Highly Available Streaming Service in the Cloud

Simon Loesing Martin Hentschel Tim Kraska* Donald Kossmann

Systems Group, ETH Zurich
{firstname.lastname}@inf.ethz.ch

** University of California, Berkeley*
kraska@cs.berkeley.edu

ABSTRACT

In recent years, new highly scalable storage systems have significantly contributed to the success of Cloud Computing. Systems like Dynamo or Bigtable have underpinned their ability to handle tremendous amounts of data and scale to a very large number of nodes. Although these systems are designed the store data, the fundamental architectural properties and the techniques used (e.g., request routing, replication and load balancing) can also be applied to data streaming systems. In this paper, we present Stormy, a distributed stream processing service for continuous data processing. Stormy is based on proven techniques from existing Cloud storage systems that are adapted to efficiently execute streaming workloads. The primary design focus lies in providing a scalable, elastic, and fault-tolerant framework for continuous data processing, while at the same time optimizing resource utilization and increasing cost efficiency. Stormy is able to process any kind of streaming workloads, thus, covering a wide range of use cases ranging from real-time data analytics to long-term data aggregation jobs.

1. INTRODUCTION

Stormy is a distributed multi-tenant streaming service designed to run on Cloud infrastructure. Like traditional data stream processing engines (SPEs) [1, 7, 20, 25], it executes continuous queries against ongoing streams of incoming data and eventually forwards the results to a designated target.

Stormy uses a synthesis of well-known techniques from Cloud storage systems [10, 15, 16] to achieve scalability and availability. This includes distributed hash tables (DHT) [4], which are a widespread approach for efficient data distribution, reliable routing, and flexible load balancing. Stormy introduces a novel idea to use DHTs for streaming workloads. Instead of mapping keys to data, we map events to queries. Thus, the DHT can be used to distribute queries across all nodes, and route events from query to query according to the query graph. To achieve high availability, Stormy uses replication. Queries are replicated on several nodes, and events are concurrently executed on all replicas. Stormy's architecture is by design decentralized, there is no single point-of-failure. All in all, Stormy combines standard techniques of Cloud Computing with stream processing to improve elasticity, scalability, and fault tolerance compared to existing solutions.

The spectrum of use cases for streaming applications is huge. It ranges from small personal applications (e.g., aggregating Twitter or RSS feeds) to large business applications (e.g., traffic control or monitoring of manufacturing processes). Research efforts on stream processing have so far mainly focused on distributing and parallelizing large and complex streaming applications. In contrast, Stormy explores new paths as it is specifically built to host a large number of small to medium size streaming applications and be used by many users at the same time. Thereby, Stormy enables the concept of Streaming as a Service. One could imagine a public streaming service where users can add new streams on demand; that is, register and share queries, and instantly run their stream for as long as they want. Stormy makes this possible.

The key contributions of this paper are:

- The design of a Cloud streaming service, which allows deploying many stream applications and is designed towards scalability and elasticity.
- A protocol that routes data streams through a distributed system, and ensures fault tolerance and consistency.
- A report of our first experiences combining data stream management systems with the new technological trend of Cloud Computing.

This paper is organized as follows. Section 2 presents the requirements to run a streaming service in the Cloud. Section 3 details the data and programming model. Section 4 provides an overview of the system architecture. Section 5 describes different operation and failure scenarios. Section 6 summarizes the current progress of Stormy. Section 7 reviews related work. Finally, Section 8 concludes this paper.

2. STREAMING AS A SERVICE

A streaming service in the Cloud has different requirements than traditional SPEs. In contrast to traditional SPEs, a Cloud streaming service must support multi-tenancy; that is, accommodate many customers with different workloads. The service also needs to tolerate failures, which

DanaC 2012, March 30, 2012, Berlin, Germany

are common in Cloud environments. In addition, elasticity, availability, and consistency are equally important. In the following we describe all of these requirements in detail.

Multi-tenancy, Scalability, Elasticity. A streaming service in the Cloud should accommodate many different customers at the same time. Thus, it needs to scale in two dimensions: (1) the number of queries and (2) the number of streams. First, the system should be able to execute more and more queries (by many different customers), by increasing the number of nodes in the system. Second, the system should also support more and more streams (for even a single customer), by also increasing the number of nodes. Ideally, the system processes any number of queries and streams the more nodes are added to the system, and thus can support an arbitrary amount of customers. Equally important, the system should shut down under-utilized nodes at any time to optimize its own operation costs; a property called elasticity.

Availability. A streaming service in the Cloud should be always available, even if some of its nodes are unavailable. That is, it should always be possible to push new events to the service, even if some nodes are down. This, in turn, requires that events can be pushed to any node of the streaming service and not only to a specific entry point. In particular, there should be no single point of failure in the system which, if temporarily unavailable, would force the system to be blocked.

Strong Consistency. In this work we focus on strong consistency. In detail, strong consistency demands the following three requirements. (1) All events within the service are totally ordered. (2) No accepted events are lost inside the service. This also requires that state data, for example windows of streaming computations, is kept persistent. State data cannot be lost in case of node failures. And (3) output events are monotonic, which means that an output event can never lie in front of a previously output event.

No Administration. A streaming service in the Cloud is required to have as little administrative overhead as possible. Ideally, there should be no need for administration at all. It would be unacceptable, and obviously expensive, if the system would need human assistance in case of problems. Therefore, the system is required to automatically scale and load balance itself if the workload changes; as well as to automatically fail over when single nodes in the system drop out. Automatic adjustment ties in with elasticity in that the system can be optimized to always operate at the lowest possible cost. No unnecessary nodes need to be active, which only increases the cost of the streaming service in the Cloud.

3. DATA AND PROGRAMMING MODEL

In this section, we present the system design of Stormy and explain under which assumptions the above requirements are implemented.

3.1 Data Model and Assumptions

A streaming application usually consists of three components: (1) external data streams that feed the system with data; (2) queries that process the data; and (3) external output destinations, to which the system pushes results. These components a connected in a directed acyclic graph (DAG), or query graph, in which the vertices are the queries that process the data, and the edges are internal data streams that connect two or more queries. In Stormy, every component can be separately registered, and the system incrementally builds the query graph of the streaming application.

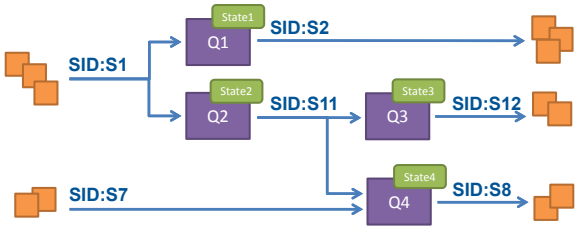

Figure 1: A sample stream query graph in Stormy.

Queries of streaming application are generally continuous and stateful. Once a query is registered, it starts processing events and only stops when the system terminates or the query is deregistered from the system. Queries typically maintain state such as aggregates of windows or local variables. Query state is kept on the same node that executes the query. In Stormy, we assume that a query can be completely executed on one node. In other words, the load of a query never exceeds the capacity of a single node—thus, there is an upper limit of the number of incoming events of a stream. Stormy (as well as other systems [11, 19]) is currently not able to deal with streams for which the incoming event rate exceeds the capacity of a node.

The distributed streaming system has to take care of the correctly routing of data inside the DAG from query to query. For this reason, every data stream is identified by a unique stream identifier (SID). The moment an external data stream is added to the system it gets assigned an SID. When a query is registered, it subscribes to an incoming SID it receives data from and creates a new outgoing SID, at which results will be made available. A query has exactly one output stream, but can receive data from several input streams. To collect all the processed results, an application outside the system can subscribe to the outgoing SID of the last query. Every time a data source, query, or output destination is registered, Stormy maps the incoming and outgoing SIDs and thus builds an internal representation of the query graph.

External data streams push events into the system. The stream of events is usually endless, and there are no restrictions on the rate or the pattern in which the events are emitted. Events have the following properties:

- When entering the system, every event gets assigned a unique identifier, an event id. Given this event id, it is possible to detect and eliminate duplicate events inside the system.
- An event that propagates through the execution graph always belongs to a single stream. The current SID, is encoded in each event to ensure that the system can identify the next processing step of that event.
- Time is explicitly encoded inside events. The order of events from a particular data source can be detected by comparing the timestamps of the events.

Figure 1 illustrates a sample execution graph that consists of two input streams (identified by SID S1 and S7), four queries (Q1–Q4), and three output destinations. If, for example, an event enters the system via stream S1, the systems forwards it to the subscribed queries. In this case, the

event is sent to the nodes responsible for Q1 and Q2. After being processed, each event gets attached the outgoing SID of the query. For Q1, the event is attached with SID S2 and the system forwards the event to the corresponding output destination. For Q2, the event is attached with SID S11 and forwarded to the nodes responsible for Q3 and Q4. Further processing continues the same way.

Stormy does not allow fixed boundaries on latency. That is, events do not have to be processed within a given time interval. Still, latency is an optimization goal. The streaming service tries to process events as fast as possible.

3.2 Programming Model

The API of Stormy consists of only four functions:

- **registerStream(description) → SID:** Registers an external data stream and returns the generated internal SID for this stream.
- **registerQuery(name, query, input SIDs) → SID:** Adds a query with the parameters name, query, and a set of one or more input SIDs. It returns the SID of the query output stream.
- **registerOutput(SID, target):** Registers an output destination that receives the data from the given SID. The target parameter contains the host address and port of the output location.
- **pushEvents(SID, events):** Pushes a set of one or more events with the given input stream SID to the system.

Multiple customers can register streams, queries, and output destinations; either independently of each other, or they may even share queries or output destinations. Any function, which registers a stream, query, or output destination, also has a corresponding de-registration function. The de-registration functions are omitted here for brevity.

To support all requirements of Section 2, the architecture of Stormy consists of the following three mechanisms: (1) query distribution to achieve scalability and elasticity, (2) replication to gain fault tolerance and availability, and (3) leader election to ensure strong consistency.

3.3 Query Distribution

The system needs a mechanism to distribute queries among the available nodes. To do so, Stormy uses consistent hashing [14], a technique employed in many distributed storage systems. In consistent hashing, the output range of a hash function can be treated as a fixed circular space, or "ring". (The largest value wraps over to the smallest value.). Each node in the system gets assigned a random value that defines its position on this ring. If a new item is inserted in the system, the hash value of its key is calculated and the item is assigned to the first node reached by following the ring clock-wise starting at the location of the item's hash value. In other words, every node becomes responsible for the key space between its own position and the position of its predecessor.

The mapping of ranges to nodes is maintained using a distributed hashtable. In a DHT, every node knows the mapping of keys to nodes and can therefore forward an incoming request to the responsible node. If the DHT changes (e.g., to accommodate a new node that is added to the system), the new version of the mapping is propagated through the system using a gossip protocol. A gossip protocol distributes new information to all nodes in the system, typically with some time delay. Therefore, it might happen that a node with outdated mapping information forwards the request to the wrong node. However, the request will eventually arrive at the correct node as the new mapping information is gossiped through the system.

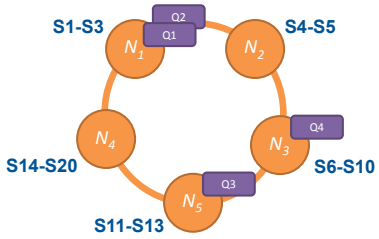

Figure 2: A Stormy ring with five nodes.

Stormy uses consistent hashing for two reasons: (1) to assign queries to nodes; and (2) to route events to nodes, and, thus, to the responsible queries. To assign a query to a node, we use the input stream SID as parameter. Every node has a range of SIDs assigned, and all the queries with an input stream SID in the range of a node will be placed on that node. To route events to nodes, we use the SID as well. As discussed, every event is attached with an SID. Figure 2 illustrates how queries are assigned to nodes. Figure 2 shows a Stormy ring consisting of five nodes, where every node is responsible for a specific range of SIDs (e.g., Node N1 is responsible for all values in the range S1 to S3). To place the four queries of Figure 1 on the nodes of this ring, we use the incoming SID of these queries. Queries Q1 and Q2 with SID S1 is placed on node N1. Query Q3 with SID S11 is place on node N3. Query Q4 is a special case because it has two input stream ids, S7 and S11. In such a situation, we select the first input stream id, SID 7, as the reference SID to determine on which node Q4 will be located, here node N3. Furthermore, we maintain a system-wide list of *aliases*, which additionally map the second output stream S11 to the node of Q4. Note that such a special mapping is only required if a query has several input stream identifies belonging to different nodes.

The fundamental advantage of consistent hashing is flexibility with regard to load balancing and departure or arrival of new nodes. If a node becomes overloaded, it can hand over a part of its range to one of its immediate neighbors. If the *overall* load of the system is getting too high, a new node needs to be added to the system. This is decided by the leader in the ring that brings up a new node and places it on the ring next to the most overloaded nodes. By using consistent hashing, the new node only needs to communicate with its neighbors to receive parts of their queries and query state. All other nodes are not affected, which enables the system to make further progress even in overload situations. Thus, structural changes in Stormy affect only at most two nodes, which allow Stormy to scale out to dozens or even hundreds of nodes. Section 4.3 elaborates on the details of load balancing and node arrival in Stormy.

3.4 Replication

To achieve high availability, Stormy uses replication. Every query is replicated on several nodes, and a replication protocol takes care that every incoming event is executed by

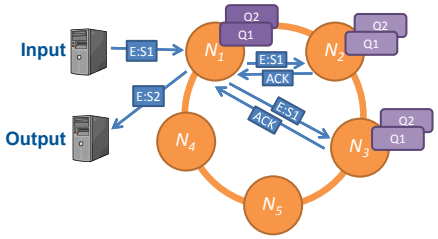

Figure 3: Event execution with replication factor 3.

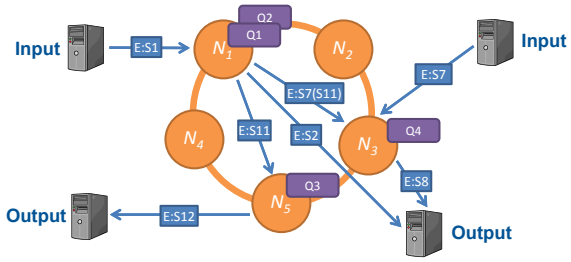

Figure 4: Stream execution in Stormy.

every replica. We use successor list replication, as proposed in Chord [24], to create replicas on the $r-1$ direct successors of the responsible ("master") node. r is the configurable replication factor, which determines how many copies of a query exist. (By default $r = 3$.) The master node and its replicas form a replication group. Figure 3 shows the replication group for queries Q1 and Q2. Node N1 is the master node, and the two successor nodes, N2 and N3, are the respective replicas.

To ensure consistent processing inside a replication group, we guarantee that events are processed in total order and that no event is lost. The replication of events depends on the master node, which acts as coordinator to replicate events and always maintains the current query state. For brevity, we can only sketch the replication protocol used in Stormy: An incoming event is routed to the master node as defined by the DHT. As first step, the master uses the event id and timestamp to validate that is has already received (and processed) all previous events. Second, the master sends the event to all replica nodes and locally executes the event's respective query. As soon as a replica receives an event, the replica runs the same validation process as the master. If the event is valid, the replica executes it and sends back a confirmation to the master. (As an optimization, sending and acknowledging bulks of events is favorable.) In case the validation fails (e.g., the replica has missed some events), the replica requests a state update from the master. Once the replica has been provided with the current state, the execution can proceed. The master waits until it has received a majority quorum of acknowledgments, meaning that at least half of the replicas have successfully executed the event. Then the master updates its state and puts the results on the output stream. In case the quorum majority is not reached after a specified timeout, the execution has failed and an error is issued to the process that as initially submitted the event. Failure scenarios are discussed further in Section 4.2. Figure 3 illustrates this replication protocol for an event E. Node N1 is the master to replicate the event, wait for acknowledgements, and output the result.

High availability has its price. The replicated execution generates additional processing overhead and requires more resources. If consistency or availability is less critical, it makes sense to deactivate replication; that is, setting the replication factor to $r = 1$. Nonetheless, for most applications the cost of setting $r > 1$ is acceptable to reduce the risk of disruption in the stream processing.

3.5 Leader Election

Until now we assumed that the master node is always the node to which the query is assigned in the DHT. This is true as long as no nodes are added or deleted from the system at runtime, and no node failures occur. In a dynamic system however, nodes can join or leave the system, and queries might be relocated in a load-balancing step. As a result, the master role can move from one node to another. To maintain a consistent state, Stormy uses a leader election process, which guarantees that exactly one master node exists for each replication group at any time.

If the master changes and the old master is still alive (e.g., after a load-balancing step), a master-handover procedure is initiated by the leader election process. The old master stops processing, queues incoming events, and send its state to the new master. The new master takes over the state, acknowledges the master handover, and waits for the old master to forward all queued events. If the handover process fails, the old master keeps the master role and processes the queued events. If the old master is no longer reachable, the leader election process assigns the master role to a replica that is in the same state as the old master node.

To prevent a single point of failure, the leader election service runs on several nodes (by default 3). To consistently assign master nodes, leader election is based on a consensus protocol, such as Paxos [17].

4. SCENARIOS

This section illustrates how Stormy behaves under different scenarios. We present how the query graph in Figure 1 is executed under normal conditions, but also discuss how the system reacts to failures and behaves during load balancing.

4.1 Normal State Scenarios

Normal state operation in Stormy is straightforward. Figure 4 illustrates how the query graph of Figure 1 is executed. We assume that all of the three, the input stream, queries, and output destinations, have been registered. Thus, the system is ready to receive events. In this example, the input events are directly sent to the correct node, which is a feature provided by the Stormy client library. However, as every node maintains a copy of the routing table, new events could be sent to any node in the system and would be directly forwarded to the correct node. The moment all queries are registered, events can be executed and routed through the system as specified by the query graph. Once the processing is complete, the final results are pushed to the output destinations and are made available for further external processing.

4.2 Failure Scenarios

As discussed, failures can happen at any time. The following scenarios explain the three most common failure cases.

Transient Node Failures. Node failures can be detected by message timeouts or heartbeats. If the delivery of

an event to a node times out, the sender node assumes the receiver to be dead. However, events are still sent to the receiver for a period of several heartbeats, in the hope that it might become available again. Every node maintains a list of assumed-to-be-dead and (known-to-be-) dead nodes. This list is synchronized with all other nodes using the gossip protocol. A node becomes available again if a message is successfully delivered. Each node will resend previously timed-out messages. If a node is assumed to be dead for a period of several heartbeats, it is finally declared (known-to-be-) dead.

Permanent Node Failures. A permanent node failure will lead to nodes that are known to be dead (or simply dead). A dead node is removed from the ring, and the leader election service is automatically activated to elect a new node as master. In the meanwhile, events are buffered until a new master is elected. The new master will become responsible for the data range of the failed node. The next successor nodes on the ring might become new replicas, depending on the replication factor. New replicas have to request the queries and query state information from the master. As soon as the data has been copied, routing information is updated and the new replicas take part in the processing of queries. Note that from the point on the node is declared dead until the recovery of all replication groups the involved nodes do not process any events.

Lost Events. Events are sent via messages from one node to the other, and are prone to message loss or corruption. In order to be able to recognize message losses, every event is transmitted with a unique identifier and timestamp. This allows Stormy to uniquely identify the position of an event in the data stream, and verify that all previous events have been processed in correct order. If a master node recognizes a missing event, it first waits for a fixed interval because the message might just be delayed. After that interval, an error is signaled to the sender and the event will be resubmitted. (Duplicate events can be detected by the event identifier.)

4.3 Performance Scenarios

As discussed, overload situations are critical and might affect latency. To cope with overload situations, Stormy uses two techniques, load balancing and cloud bursting.

Load Balancing. Load balancing is a technique to redistribute the load across nodes. The decision to balance load is made locally by a node, based on its current utilization. Each node continuously measures its resource utilization of CPU, memory, network consumption, and disk space. These parameters form a utilization factor, which is disseminated via the gossip protocol to all other nodes in the system. In regular intervals (by default every 2 minutes), a node compares its utilization factor to those of its immediate neighbors. If the load difference is above a specified threshold, the node initiates a load-balancing step; that is, the node transfers part of its data range to its less loaded neighbor. Section 3.5 explains how this process is executed consistently. As every node makes the decision to balance load locally, several load-balancing steps can happen in parallel, which allows the whole system to react efficiently to load variations.

Cloud Bursting. If the overall load of the system is getting too high, a new node has to ba added to the system. We call this mechanism cloud bursting. The decision to cloud-burst is made by only one node in the system, the elected cloud bursting leader. This leader initiates the cloud bursting procedure, which brings up and adds a new node to the system. This new node takes a random position on the ring, takes over parts of the data range from its neighbors, and finally updates the DHT. This mechanism also works the other way round. If the overall load of the system is too low, the cloud bursting leader decides to drop a node from the system. Similar to adding a new node, we first move the queries and their state to the node that takes over the range, update the DHT, and terminate the old node.

5. IMPLEMENTATION

We implemented Stormy on top of Cloudy [15], a modular Cloud storage system that is easily extendable with further functionality. Cloudy offers a DHT to distribute and replicate events across the nodes of the system, and therefore already provides scalability and elasticity. On top of Cloudy, we added the MXQuery stream processing engine [21]. MXQuery allows to execute any XQuery query with additional streaming constructs. We implemented a new processing protocol that can buffer and validate events and take care of new features like master handover. Furthermore, we added Apache Zookeeper [2] to ensure consistent leader election. (Currently, Zookeeper runs outside of the system and is demonstrably not a bottleneck of Stormy.) Stormy is still in active development. Future work includes evaluating Stormy's performance using the popular Linear Road benchmark [3].

6. RELATED WORK

Similar to Stormy, distributed stream processing engines (SPEs) like Borealis [1], Medusa [8], or TelegraphCQ [7] also distribute stream processing across several nodes. These systems allocate queries to individual nodes to scale with the number of queries. Although basic functions to move queries between nodes are provided, these systems are not designed to run in the dynamic environment of the Cloud. However, there have been first attempts to support more dynamism in distributed SPEs. For example, follow-up papers on Borealis try to improve fault tolerance by exploring trade-offs between consistency and availability [5], and to minimize latency using a load-distribution algorithm [27]. However, to the best of our knowledge none of these systems support all the listed requirements in Section 2 as Stormy does. A further approach in SPEs to improve availability is load shedding [26]. Load shedding intentionally drops single events to handle overload situations. The price of load shedding is loss of result quality. Stormy, on the other hand, is designed to guarantee total ordering of events with best possible result quality. To handle overload situations, Stormy implements load balancing; queries are moved between nodes to uniformly distribute load in the system. Research of distributed SPEs has also focussed on intra-query parallelism. Systems like StreamCloud [12] or Flux [23] introduce mechanisms to parallelize single queries. Implementing intra-query parallelism in Stormy is considered future work.

The replication and distribution mechanism in Stormy has been adopted from Cloud storage systems, such as Amazon Dynamo [10] or Apache Cassandra [16]. Dynamo is a highly-available Cloud storage system, usually called key-value store, which introduced replication and distribution mechanisms to enable availability and scalability in storage

systems composed of many machines. We build on these mechanisms to ensure the same properties in a distributed streaming system. Chubby [6] is a distributed locking service introduced by Google. Chubby replicates state among multiple services to ensure consistency and fault tolerance. Stormy adopts this mechanism for its leader election algorithm, only one node is can be the master at any time.

Recently, real-time analytics systems have gained focus of interest by many researchers and businesses. Real-time analytics systems aim to speed up batch-oriented analytical programs, which are typically expressed in MapReduce [9] or Dryad [13]. Thereby, they process new input data immediately, similar to how Stormy processes new events immediately. Examples of recent real-time analytics systems include Google Percolator [22], Twitter Storm [18], Naiad [19], and Limmat [11]. In contrast to Stormy, real-time analytics systems typically offer simple programming models, close to the original MapReduce programming model. Stormy, however, offers the richer programming model of data stream applications, including the notion of windows and window operators. Furthermore, Stormy strictly orders events that are forwarded inside the system. Real-time analytics systems usually have relaxed ordering constraints [22, 19, 11].

7. CONCLUSION

In summary, we presented Stormy, a distributed streaming system that runs on Cloud infrastructure. We highlighted important requirements that must be met by a streaming service in the Cloud. These requirements include support for multi-tenancy, scalability, availability, strong consistency, and no administration overhead (or human intervention). Stormy meets these requirements by combining data stream processing with well-known techniques from key-value stores and distributed lock services. In particular, Stormy partitions and replicates continuous query processing across nodes in the system, which allows scaling with the number of nodes and being resilient to node failures. Furthermore, Stormy automatically adjusts itself to the current workload, thereby adding and removing nodes to optimize operation costs of the system. Thus, Stormy is a first implementation of a multi-tenant streaming service in the Cloud.

8. ACKNOWLEDGEMENTS

We would like to thank Stephan Merkli, Flavio Pfaffhauser, and Raman Mittal for their valuable work on the Cloudy storage system and early contributions to Stormy.

9. REFERENCES

[1] D. J. Abadi, Y. Ahmad, M. Balazinska, M. Cherniack, J.-H. Hwang, W. Lindner, A. S. Maskey, E. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. The design of the Borealis stream processing engine. In *CIDR '05*, pages 277–289, 2005.

[2] Apache Zookeeper. http://zookeeper.apache.org/.

[3] A. Arasu, M. Cherniack, E. Galvez, D. Maier, A. S. Maskey, E. Ryvkina, M. Stonebraker, and R. Tibbetts. Linear Road: A stream data management benchmark. In *VLDB '04*, pages 480–491, 2004.

[4] H. Balakrishnan, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Looking up data in P2P systems. *CACM*, 46:43–48, February 2003.

[5] M. Balazinska, H. Balakrishnan, S. R. Madden, and M. Stonebraker. Fault-tolerance in the Borealis distributed stream processing system. *ACM Transactions on Database Systems*, 33:1–44, March 2008.

[6] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. In *OSDI '06*, pages 335–350, 2006.

[7] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, F. Reiss, and M. A. Shah. TelegraphCQ: Continuous dataflow processing. In *SIGMOD '03*, page 668, 2003.

[8] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Çetintemel, Y. Xing, and S. Zdonik. Scalable distributed stream processing. In *CIDR '03*, pages 257–268, 2003.

[9] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI '04*, pages 137–150, 2004.

[10] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *SOSP '07*, pages 205–220, 2007.

[11] M. Grinev, M. Grineva, M. Hentschel, and D. Kossmann. Analytics for the real-time web. *PVLDB*, 4(12):1391–1394, September 2011.

[12] V. Gulisano, R. Jimenez-Peris, M. Patino-Martinez, and P. Valduriez. StreamCloud: A large scale data streaming system. In *ICDCS '10*, pages 126–137, 2010.

[13] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *EuroSys '07*, pages 59–72, 2007.

[14] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In *STOC '97*, pages 654–663, 1997.

[15] D. Kossmann, T. Kraska, S. Loesing, S. Merkli, R. Mittal, and F. Pfaffhauser. Cloudy: A modular cloud storage system. *PVLDB*, 3:1533–1536, September 2010.

[16] A. Lakshman and P. Malik. Cassandra: A decentralized structured storage system. *SIGOPS Operating Systems Review*, 44:35–40, 2010.

[17] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16:133–169, May 1998.

[18] N. Marz. A Storm is coming. http://engineering.twitter.com/2011/08/storm-is-coming-more-details-and-plans.html, August 2011.

[19] F. McSherry, R. Isaacs, M. Isard, and D. G. Murray. Naiad: The animating spirit of rivers and streams. In *SOSP '11*, 2011.

[20] Microsoft SQL Server StreamInsight. http://www.microsoft.com/sqlserver/en/us/solutions-technologies/business-intelligence/complex-event-processing.aspx.

[21] MXQuery: A lightweight, full-featured XQuery engine. http://mxquery.org/.

[22] D. Peng and F. Dabek. Large-scale incremental processing using distributed transactions and notifications. In *OSDI '10*, pages 251–264, 2010.

[23] M. Shah, J. Hellerstein, S. Chandrasekaran, and M. Franklin. Flux: An adaptive partitioning operator for continuous query systems. In *ICDE '03*, pages 25–36, 2003.

[24] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM '01*, pages 149–160, 2001.

[25] StreamBase Systems, Inc. http://www.streambase.com/.

[26] N. Tatbul, U. Çetintemel, and S. Zdonik. Staying FIT: efficient load shedding techniques for distributed stream processing. In *VLDB '07*, pages 159–170, 2007.

[27] Y. Xing, S. Zdonik, and J.-H. Hwang. Dynamic load distribution in the Borealis stream processor. In *ICDE '05*, pages 791–802, 2005.