Apache Flink™: Streaming Dataflows as a Basis for Universal Data Analytics

Authors

Abstract

Apache Flink is an open source system for processing streaming and batch data. Flink is built on the philosophy that many classes of data processing applications, including real-time analytics, continuous data pipelines, historic data processing (batch), and iterative algorithms (machine learning, graph analysis) can be expressed and executed as pipelined fault tolerant dataflows. In this paper, we present Flinks architecture and expand on how a (seemingly diverse) set of use cases can be unified under the same execution model.

1 Introduction

Data stream processing (complex event processing / continuous queries) and static data processing (batch processing, OLAP) were traditionally considered as two very different types of applications. They were programmed using different programming models and APIs, and were executed by different systems (for example, dedicated streaming systems such as Apache Storm, IBM Infosphere Streams, Microsoft Streaminsight, or Streambase versus relational databases or Apache Hadoop execution engines). Traditionally, batch data analysis made up for the lion share of the use cases, data sizes, and market, while streaming data analysis existed mostly in specialized applications.

It is becoming more and more apparent, however, that a huge number of today's large scale data processing use cases handle data that is, in reality, continuously produced over time. These continuous streams of data come for example from web logs, application logs, sensors, or as changes to application state in databases (transaction log records). Rather than treating the streams as streams, today's setups ignore the continuous and timely nature of the data production. Instead, data records are (often artificially) batched into static data sets (for example hourly/daily/monthly chunks) and then processed in a time-agnostic fashion. Data collection tools, workflow managers, and schedulers orchestrate the creation and processing of batches, in what is actually a continuous data processing pipeline. Architectural patterns like the "lambda architecture" [] combine batch and stream processing systems to implement multiple paths of computation: A streaming fast path for timely approximate results, and a batch offline path for late accurate results.

Apache Flink follows a paradigm that embraces data stream processing as the unifying model for real-time analysis, continuous streams, and batch processing, both in the programming model and in the execution engine. In combination with durable message queues that allow quasi arbitrary replay of data streams (like Apache Kafka [3] or Amazon Kinesis[1]), stream processing programs make no difference between processing the latest events

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

[1] https://aws.amazon.com/kinesis/

in real-time, continuously aggregating data periodically in large windows, or processing terabytes of historical data: these types of computations simply start their processing at different points in the durable stream, and maintain different forms of state during the computation. Through a highly flexible windowing mechanism, Flink programs can compute both early and approximate, as well as delayed and accurate results in the same operation, obviating the need to combine different systems for the two use cases. Flink supports different notions of time (event time, ingestion time, processing time) in order to give programmer high flexibility in defining how events should be correlated.

At the same time, Flink acknowledges that there is, and will be, a need for dedicated batch processing (dealing with static data sets). Complex queries over static data are still a good match for a batch processing abstraction. Furthermore, batch processing is still needed both for legacy implementations of streaming use cases, and for analysis applications where no efficient algorithms are yet known that perform this kind of processing on streaming data. Batch programs are special cases of streaming programs, where the stream is finite, and order/time of records does not matter (all records implicitly belong to one all-encompassing window). However, to support batch use cases with a competitive ease and performance, Flink has a specialized API for processing static data sets, uses specialized data structures and algorithms for the batch versions of operators like join or grouping, and uses dedicated scheduling strategies. The result is that Flink presents itself as a full-fledged and efficient batch processor on top of a streaming runtime, including libraries for graph analysis and machine learning. Flink is a top-level project of the Apache Software Foundation that is developed and supported by a large and lively community (of more than 140 open source contributors at the time of this writing []), and is used in production in several companies.

The contributions of this paper are the following:

- We make the case for a unified architecture of stream and batch data processing, including specific optimizations that are only relevant for static data sets.
- We show how to make these dataflows fault tolerant via an asynchronous snapshotting mechanism (Section 4).
- We discuss how we can build a full-fledged stream analytics system with a flexible windowing mechanism (Section 5), as well as a full-fledged batch processor (Section 6) on top of these dataflows, by showing how streaming, batch, iterative, and interactive analytics can be represented as streaming dataflows.

2 System Architecture

In this section we lay out the architecture of Flink as a software stack, and as a distributed system. While Flink's stack of APIs continues to grow, we can distinguish four main layers: deployment, core, APIs, and libraries.

Flink's Runtime and APIS. The core of Flink is the distributed dataflow engine, which executes dataflow programs called Job Graphs. A Job Graph is a DAG of operators and connections between operators. There are two "core" APIs in Flink: the DataSet API for processing finite data sets (often referred to as "batch processing"), and the DataStream API for processing potentially unbounded data streams (often referred to as "stream processing"). Flink's core runtime engine can be seen as a streaming dataflow engine, and both the DataSet and DataStream APIs create programs (Job Graphs) accepted by this engine. On top of the core APIs, Flink bundles "domain-specific" libraries and APIs that generate DataSet and DataStream API programs. Currently these are the following: i) Gelly, an API and library for processing graphs, ii) FlinkML, an API and library for composing Machine Learning pipelines and iii) Table, an API similar in spirit to Microsoft's LINQ. A Flink cluster comprises of three types of processes: the client, the JobManager, and the TaskManager. The client takes the program code (written in a mixture of Flink's APIs), transforms it to a Job Graph, and submits it to the JobManager. The compilation process involves a type extraction and checking phase that generates serializers and comparators for all used types. DataSet programs, also go through a cost-based query optimization phase, similar to the physical optimizations performed by relational query optimizers (more details in Section X).

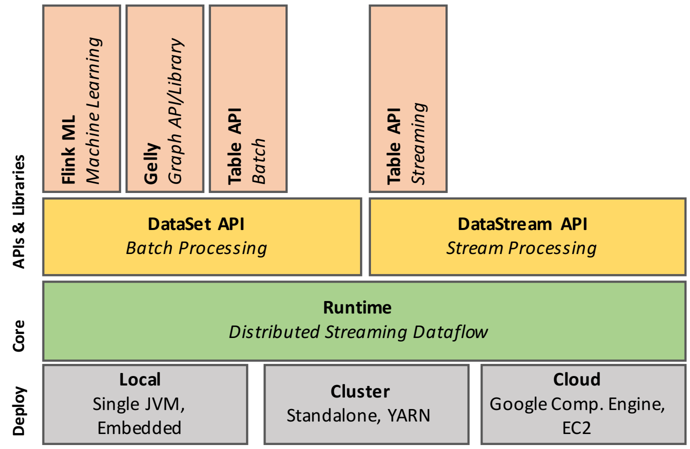

Figure 1: The Flink software stack.

The JobManager is Flink's master node. It coordinates all message-passing during job execution by sending heartbeats to the TaskManagers, receiving statistics, controls the tasks' lifecycle, coordinates the fault tolerance machanisms. The actual data processing takes place in the TaskManagers, or worker processes that they execute several tasks in multiple threads, and maintain data structures shared by all tasks (e.g., buffer pools) executed by a TM. TMs communicate directly with each other using a multiplexed TCP connection per TM pair.

3 Execution model

The Job Graph. Flink's execution model is based on the Job Graph, a directed acyclic graph (DAG) that consists of nodes and edges. There are two classes of nodes: (stateful) operators, and (logical) intermediate results (IRs). For example, the graph below consists of five operators (circles), and three intermediate results. Operators abstract computation (e.g., transformations, joins, etc), state (e.g., a persistent counter), as well as data sources (e.g. reading data from a file system, a socket, a message queue, etc.), and data sinks. Operators produce intermediate results, as well as updates to state. An intermediate result is a logical handle (pointer) to the data that is produced by one operator. An intermediate result can be consumed by one or more operators. Intermediate results are logical in the sense that the data they point to may or may not be materialized on disk. When the JobGraph is scheduled in a cluster for execution, it is parallelized to form an ExecutionGraph that consists of tasks (parallel instances of operators), and intermediate result partitions (IRPs).

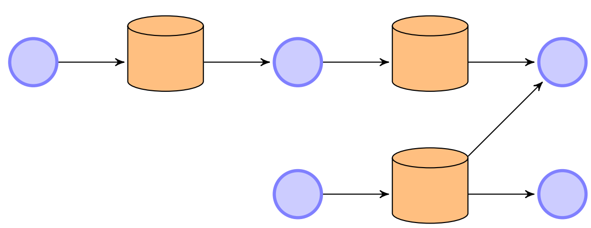

Figure 2: The JobGraph is the logical view of a Flink job.

Data Transfer. The unit of data transfer in the Flink runtime is a buffer. Buffers contain one or more records, and a record can span multiple buffers. Buffers are requested and relinquished from local buffer pools, shared among operators that live in the same task manager. An IRP is simply a collection of buffers. Work in Flink progresses (i.e., records flow through the pipeline) as long as there are buffers available, essentially implementing distributed blocking queues (the logical streams) with bounded capacity (the amount of memory available to the buffer pools, which can be configured by the user). This mechanism, in addition to implementing record network transfers doubles down as a natural way to backpressure the flow in the case of slow operators (including external systems that consume data).

We mentioned that the intermediate results are logical handles to the data, rather than the data itself. Internally, intermediate results are abstract classes with many implementations. These implementations can perform pipelined data exchange, and blocking data exchange.

Pipelined Data Exchange. Pipelining (also called intra-operator parallelism), means that a producing and a consuming operator make progress at the same time, without the consumer waiting for the producer to finish. Pipelining is required in streaming systems, and is also used in batch systems to reduce latency. Flink implements pipelining by implementing intermediate results which activate network transfers between the producer task and consumer task, as soon as their first buffer is available. Flink allows the configuration of its buffers' size and timeout. A buffer is sent to its consumer task as soon it is filled, or as soon as a timeout is reached. Hence, by setting the buffer size and/or timeout to lower values, one can reduce latency, while achieving the opposite effect (i.e., high throughput) by using higher values.

Blocking Data Exchange. Sometimes it is desirable to break a job into stages, scheduling and executing each stage individually (e.g., to enable interactive processing, and better staging of resources in large batch jobs). To do that, the system has to materialize intermediate results (in memory or disk). Flink implements blocking data exchange via an intermediate result that signals its availability only when all the buffers from the producer have been materialized. The cached buffers can double down as a materialized reusable intermediate result.

4 Fault Tolerance via Asynchronous Snapshots

Fault tolerance in Flink is achieved by taking a snapshot of the execution graph at regular intervals. When failure occurs the state of the execution is restored from the latest snapshot. Upon recovery, in order to guarantee exactly-once processing semantics, the records (events) consumed since the dryad snapshot are reprocessed in the same order at each respective source. Normally this possible by using persistent message queueing systems, e.g. Apache Kafka. [3]

Flink's core mechanism for distributed snapshots is called "Asynchronous Barrier Snapshotting" (ABS). [14] It creates a global snapshot by collecting the state of each task in the execution graph in an asynchronous manner. It does so by superimposing snapshotting using injected markers, called "barriers", in the input data stream. Similar to Chandy-Lamport distributed snapshots [16], the barriers in ABS dictate the records that should be fully processed before each respected global snapshot, however, no records in transit are included in a snapshot, keeping the persisted state at a minimum. ABS achieves this by a special "aligning" phase which ensures that all records from all upstream tasks preceding broadcasted barriers have been fully consumed before processing the data stream further. Recovery from failure reverts the execution graph to the most recent snapshot and is fully consistent by respecting the causal dependency of records. Furthermore, ABS [14] supports snapshotting of cyclic graphs. The operation of ABS is fully decoupled from the backend used for persisting states, thus, allowing multiple different backend implementations to be integrated. The following figure illustrates the snapshot process.

5 Stream Analytics on Top of Dataflows

Flink's DataStream API implements a full stream analytics framework on top of Flink's runtime, including the mechanisms to manage time (including out-of-order event processing), defining windows, and maintaining and

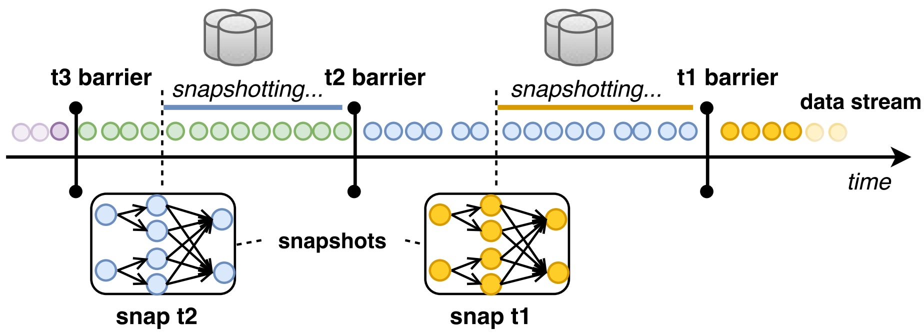

Figure 3: Asynchronous Barrier Snapshotting.

updating user-defined state. The streaming API is based on the notion of a DataStream, a (possibly unbounded) immutable collection of elements of a given type. Since the Flink runtime already has pipelined data transfers, continuous stateful operators, and a fault tolerance mechanism for consistent state updates, overlaying a stream processor on top of it essentially boils down to implementing a windowing system and a state interface.The programming model of Flink's streaming API, builds on two basic abstract data types, namely data streams and window streams. Each of the two supports specific operations since they exhibit different properties. For example, windows support transformations that are only possible on bounded collections such as joins, groupings and aggregations. Other transformations, such as (flat)map are can be applied incrementally on data streams.

The API is designed with simplicity in mind, while at the same time providing powerful tools to deal with time and uncertainty. Here is, for example, a word count program on a simple time-based window:

Listing 1: WindowWordCount Scala implementation

```
object WindowWordCount {
  def main(args: Array[String]) {

    val env = StreamExecutionEnvironment.getExecutionEnvironment
    val text = env.socketTextStream("localhost", 9999)

    val counts = text
    .flatMap {line =>
          line.toLowerCase.split("\\W+")
             .filter { w => w.nonEmpty }}
    .map {w => (w, 1) }
    .keyBy(0)
    .timeWindow(Time.of(5, TimeUnit.SECONDS))
    .sum(1)

    counts.print
    env.execute("Window Stream WordCount")
  }
}
```

Flink distinguishes between two notions of time:

Event time is the time that an event actually happened (e.g., the time that a sensor emitted a signal, or the time that a person tapped on their smartphone).

Processing time is the wall-clock time of the machine that is processing the data.

In distributed systems, there is an arbitrary lag between event-time and processing-time [10]. This may mean arbitrary delays for getting an answer based on event-time semantics. To avoid arbitrary delays these systems regularly insert special events called "low watermarks" that include a time attribute t indicating that all events lower than t have already entered the system. The watermarks aid the execution engine to process events in the correct event order.

Flink programs that are based on processing time, rely on the machine clocks, and hence a less reliable notion of time, but exhibit the best latency. Programs that are based on event time provide the most reliable semantics, but may exhibit latency due to event time-processing time lag. Flink includes a third notion of time as a special case of event time called ingestion time, which is the time that events enter the system. This provides a lower latency than event-time semantics, and avoids the arbitrary processing time semantics.

Streaming Windows. A window defines a logical group of records that are processed together. Flink's windowing system largely follows the Dataflow model [10] proposed by Google. A window definition consists of three building blocks: a window assigner, optionally a trigger, and optionally an evictor. The assigner predefines the logical groups to which each record belongs, e.g. count-based or time-based periodic windows. The trigger defines when the window operation is performed on each group. The evictor defines which records to keep in each group. The following example defines windows that operate in event time of 6 seconds that slide every 2 seconds (Assigner). The window results are computed once the watermark passes the end of the window (Trigger). Flink also offer short hands for commonly used functionalities.

Listing 2: Window API functionalities

```
//general window API
stream
  .window(SlidingTimeWindows.of(Time.seconds(6), Time.seconds(2))
  .trigger(TimeTrigger.create())

//equivalent short hand for common use case
stream.timeWindow(Time.seconds(6), Time.seconds(2))
```

A global window creates a single logical group. The following example defines a global window (assigner) that invokes the operation on every 1000 events (trigger) while keeping the last 100 elements (evictor).

Listing 3: Global Window usage

```
stream
  .window(GlobalWindow.create)
  .trigger(Count.of(1000))
  .evict(Count.of(100))
```

Note that streams are already partitioned on a key before windowing, so the above is a local operator that does not require coordination between machines. This mechanism can be used to implement a wide variety of windowing functionality [10], and Flink comes bundled with syntactic sugar for the most common window definitions (e.g., time- and count-based windows).

Simple programs in Flink's DataStream API looks like functional, side effect-free programs consisting of transformations on unbounded immutable collections. We incorporate mutable state in the API by providing interfaces to users to register any local variable within a transformation with the system's checkpointed mechanism and freely use this variable in their code.

6 Batch analytics on top of dataflows

A bounded data set is a special case of an unbounded data stream. Thus, a streaming program that inserts all of its input data in a window can form a "batch" program and "batch analytics", or "batch processing" should be fully covered by Flink's features that we presented above. However, i) the syntax, i.e., the API for batch computation can be simplified (not need for window definitions, simpler joins and loops), ii) we can simplify the fault tolerance mechanisms and iii) we can apply query optimization borrowing ideas from MPP database systems. For these reasons, Flink treats specially batch computations, by implementing the above optimizations. For enabling batch computations on top of it's streaming runtime, Flink embeds blocking versions of its operators (sorts, joins, etc) within its runtime. These operators simply block until they have received all of their input. Moreover, Flink currently disables the asynchronous snapshotting mechanism for batch programs, and simply uses backtracking based recovery as first described in Dryad. [20]

With these established, we look that the batch-specific optimizations mentioned above: query optimization, and query processing on paged (managed) memory.

Query Optimization. Flink's optimizer builds on techniques from parallel database systems such as plan equivalences, cost models and interesting properties. However, the arbitrary UDF-heavy DAGs that make up Flink's dataflow programs, do not allow a traditional optimizer to employ database techniques out of the box [blackboxes], since the operators hide their semantics from the optimizer. For the same reason, cardinality and cost estimation methods are equally difficult to employ. Flink's optimizer employs a number of novel methods for overcoming these issues [19, 11, 18] for which we provide a short overview below. Flink's runtime supports various execution strategies including repartition/broadcast data transfer, as well as sort-based grouping and sort- and hash-based join implementations. Flink's optimizer enumerates different physical plans based on the concept of interesting properties propagation [24], using a cost-based approach to choose among multiple physical plans. The cost includes network/disk I/O and CPU cost. To overcome the cardinality estimation issues that were mentioned earlier, Flink's optimizer uses hints that are provided by the programmer.

Memory Management. Building on database technology, Flink, instead of storing objects in the JVM's heap, serializes objects into a memory segments. These memory segments resemble database blocks into which, Java objects representing the tuples that go through the runtime are serialized. Operations such as sorting, and joining, operate as much as possible on the binary data, keeping the de/serialization overhead at a minimum and partially spilling data to disk when needed. To handle arbitrary objects, Flink uses type inference, and custom serialization mechanisms. By keeping the data processing on binary representation and off-heap, Flink manages to reduce the garbage collection overhead, and implement cache-efficient and robust algorithms that scale gracefully in under memory constraints.

Native Iterations on top of Dataflows. The final aspect of Flink on which we focus, is how to implement loops on top of the Flink's dataflow engine. Some approaches execute iterations by submitting new jobs for each iteration or by adding additional nodes to a running DAG [13, 23], hiding from the engine that it is executing an iterative program. The approach, implemented in Naiad [22] adds feedback edges in the dataflow graph, supporting graphs with cycles, that allow for nested iterations. A third approach was to design specialized engines around iterative processing along (e.g., Apache Giraph, and GraphLab) allowing to reduce the number of computations in each iteration. [21]

Flink follows an approach that maintains the DAG-based runtime, but allows for special "head" and "tail" tasks to signify the beginning and end of iteration, that exchange data via shared memory. This approach simulates a cyclic dataflow within a DAG engine making Flink competitive with specialized graph engines [], while outperforming the driver-based approach []. Flink supports delta iterations [], which exploit sparse computational dependencies, and are used, as the basis for Gelly, Flink's Graph API. Finally, Flink offers Bulk Synchronous Parallel (BSP) iterations in its DataSet API, and asynchronous iterations in its DataStream API.

7 Related work

There is, by now, a wealth of engines for distributed batch and stream analytical processing. We categorise these systems below.

Batch Processing. Apache Hadoop [2] is the most popular open source system for large-scale data analysis that is based on the MapReduce paradigm [17]. Dryad [20], introduced embedded user-defined functions in general DAG-based dataflows and was used by SCOPE [24] which added a language and an SQL optimizer on top of it. Apache Tez [7] can be seen as an open source implementation of the ideas proposed in Dryad. MPP databases [], and recent open-source implementations [1] [Impala] restrict their API to SQL variants. Very similar to Flink, Apache Spark [5] is a very popular data processing framework that implements a DAG-based processing framework, provides an SQL optimizer, performs driver-based iterations, and treats stream computations as micro-batches. In contrast, Flink is the only system that i) supports an optimizer that can optimize DAG programs beyond SQL queries, ii) is able to perform very efficient iterative processing natively, iii) includes stream processing as a first-class citizen, enabling more complex use cases than micro-batches.

Stream Processing. There is a wealth of work on streaming systems, that includes commercial systems like StreamBase, Microsoft StreamInsight, and IBM InfoSphere Streams. Many of these systems are based on research in the database community in projects such as Telegraph [], Aurora/Borealis [8], Stanford STREAM [12], Trill [15] and IBM System S [15]. Most of the above systems are either i) academic prototypes, ii) closed-source commercial products, iii) do not scale the computation horizontally on clusters of commodity servers. Newer open source streaming systems that scale horizontally, such as Apache Storm [6] and Apache Samza [4], provide low level APIs and offer only at-least-once and at-most-once guarantees. MillWheel [9], a closed-source system at Google provides exactly-once guarantees with low latency and is used by Google Dataflow [10] to implement out-of-order stream analytics. To the best of our knowledge, Flink is the only open-source project that i) offers high level APIs and processing libraries, ii) provides state management with exactly-once guarantees and iii) achieves high throughput and low latency, serving both batch and true streaming computations equally well.

8 Conclusion

In this paper we presented Apache Flink, a platform that implements one universal dataflow engine designed to perform both stream and batch analytics. Flink's dataflow engine treats operator state and logical intermediate results as first class citizens, and is used by the batch and a data stream APIs. The streaming API that is built on top of Flink's streaming dataflow engine, provides the means to keep recoverable state, and to partition, transform, discretize and aggregate a data stream. While batch computations are, in theory, a special case of a streaming computations, Flink treats them specially, by optimizing their execution with a novel query optimizer, and by implementing blocking operators that gracefully spill to disk in the absence of memory.

References

[1] Apache Drill project. `https://drill.apache.org/`.

[2] Apache Hadoop project. `https://hadoop.apache.org/`.

[3] Apache Kafka project. `http://kafka.apache.org/`.

[4] Apache Samza project. `http://samza.apache.org/`.

[5] Apache Spark project. `http://spark.apache.org/`.

[6] Apache Storm project. `http://storm.apache.org/`.

[7] Apache Tez project. `https://tez.apache.org/`.

[8] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, et al. The design of the borealis stream processing engine. In *CIDR*, volume 5, pages 277–289, 2005.

[9] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle. Millwheel: fault-tolerant stream processing at internet scale. *Proceedings of the VLDB Endowment*, 6(11):1033–1044, 2013.

[10] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, et al. The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proceedings of the VLDB Endowment*, 8(12):1792–1803, 2015.

[11] A. Alexandrov, R. Bergmann, S. Ewen, J.-C. Freytag, F. Hueske, A. Heise, O. Kao, M. Leich, U. Leser, V. Markl, F. Naumann, M. Peters, A. Rheinlaender, M. J. Sax, S. Schelter, M. Hoeger, K. Tzoumas, and D. Warneke. The stratosphere platform for big data analytics. *VLDB Journal*, 2014.

[12] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R. Motwani, U. Srivastava, and J. Widom. Stream: The stanford data stream management system. *Book chapter*, 2004.

[13] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. HaLoop: Efficient Iterative Data Processing on Large Clusters. *PVLDB*, 2010.

[14] P. Carbone, G. Fóra, S. Ewen, S. Haridi, and K. Tzoumas. Lightweight asynchronous snapshots for distributed dataflows. *arXiv preprint arXiv:1506.08603*, 2015.

[15] B. Chandramouli, J. Goldstein, M. Barnett, R. DeLine, D. Fisher, J. C. Platt, J. F. Terwilliger, and J. Wernsing. Trill: a high-performance incremental query processor for diverse analytics. *Proceedings of the VLDB Endowment*, 8(4):401–412, 2014.

[16] K. M. Chandy and L. Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Transactions on Computer Systems (TOCS)*, 3(1):63–75, 1985.

[17] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 2008.

[18] S. Ewen, K. Tzoumas, M. Kaufmann, and V. Markl. Spinning Fast Iterative Data Flows. *PVLDB*, 2012.

[19] F. Hueske, M. Peters, M. J. Sax, A. Rheinländer, R. Bergmann, A. Krettek, and K. Tzoumas. Opening the Black Boxes in Data Flow Optimization. *PVLDB*, 5(11), 2012.

[20] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 59–72. ACM, 2007.

[21] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed graphlab: a framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment*, 5(8):716–727, 2012.

[22] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 439–455. ACM, 2013.

[23] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster Computing with Working Sets. In E. M. Nahum and D. Xu, editors, *HotCloud*. USENIX, 2010.

[24] J. Zhou, P.-A. Larson, and R. Chaiken. Incorporating partitioning and parallel plans into the scope optimizer. In *ICDE*, 2010.