A

Achieving Low Latency Transactions for Geo-replicated Storage with Blotter

Henrique Moniz[1], João Leitão[2], Ricardo J. Dias[3], Johannes Gehrke[4], Nuno Preguiça[5], and Rodrigo Rodrigues[6]
[1]Google and NOVA LINCS, New York, USA
[2]DI/FCT/Universidade NOVA de Lisboa and NOVA LINCS, Lisbon, Portugal
[3]SUSE Linux GmbH and NOVA LINCS, Lisbon, Portugal
[4]Microsoft, Seattle, USA
[5]NOVA LINCS and DI, FCT, Universidade NOVA de Lisboa, Caparica, Portugal
[6]INESC-ID, IST (University of Lisbon), Lisbon, Portugal

Synonyms

Blotter design and protocols; Geo-replicated transactions; Non-monotonic Snapshot Isolation in Geo-replicated Systems

Definitions

Blotter is a protocol for executing transactions in geo-replicated storage systems with non-monotonic snapshot isolation semantics. A geo-replicated storage system is composed by a set of nodes running in multiple data centers located in different geographical locations. The nodes in each data center replicate either all or a subset of the data items in the database, leading to a full replication or partial replication approach. Blotter was primarily designed for full replication scenarios but can also be used in partial replication scenarios. Under non-monotonic snapshot isolation semantics, a transaction reads from a snapshot that reflects all the writes from a set of transactions that includes, at least, all locally committed transactions and remote transactions known when the transaction starts. Two concurrent transactions conflict if their write set intersects, i.e., if there is a data item written by both. In such case, one of the transactions will abort.

Overview

Many Internet services are backed by geo-replicated storage systems in order to keep data close to the end user. This decision is supported by studies showing the negative impact of latency on user engagement and, by extension, revenue (Hoff 2009). While many of these systems rely on weak consistency for better performance and availability (DeCandia et al. 2007), there is also a class of applications that require support for strong consistency and transactions. For instance, many applications within Google are operating on top of Megastore (Baker et al. 2011), a system that provides ACID semantics within the same shard, instead of Bigtable (Chang et al.

S. Sakr, A. Y. Zomaya (eds.), *Encyclopedia of Big Data Technologies*,
https://doi.org/10.1007/978-3-319-77525-8

2008), which provides better performance but weaker semantics. This trend also motivated the development of Spanner, which provides general serializable transactions (Corbett et al. 2012) and sparked other recent efforts in the area of strongly consistent geo-replication (Sovran et al. 2011; Saeida Ardekani et al. 2013a; Lloyd et al. 2013; Zhang et al. 2013; Kraska et al. 2013; Mahmoud et al. 2013).

In this chapter, we present Blotter, a transactional geo-replicated storage system, whose goal is to cut the latency penalty for ACID transactions in geo-replicated systems, by leveraging a recent isolation proposal called Non-Monotonic Snapshot Isolation (NMSI) (Saeida Ardekani et al. 2013a). We focus on the Blotter algorithms and discuss how they achieve: (1) at most one round-trip across data centers (assuming a fault-free run and that clients are proxies in the same data center as one of the replicas) and (2) read operations that are always served by the local data center.

To achieve these goals, Blotter combines a novel *concurrency control* algorithm that executes at the data center level, with a carefully configured Paxos-based (Lamport 1998) replicated state machine that replicates the execution of the concurrency control algorithm across data centers. Both of these components exploit several characteristics of NMSI to reduce the amount of coordination between replicas. In particular, the concurrency control algorithm leverages the fact that NMSI does not require a total order on the start and commit times of transactions. Such an ordering would require either synchronized clocks, which are difficult to implement, even using expensive hardware (Corbett et al. 2012), or synchronization between replicas that do not hold the objects accessed by a transaction (Saeida Ardekani et al. 2013b), which hinders scalability. In addition, NMSI allows us to use separate (concurrent) Paxos-based state machines for different objects, on which we geo-replicate the *commit* operation of the concurrency control protocol.

Compared to a previously proposed NMSI system called Jessy (Saeida Ardekani et al. 2013a), instead of assuming partial replication, we target full replication, which is a common deployment scenario (Baker et al. 2011; Shute et al. 2013; Bronson et al. 2013). Our layering of Paxos on top of a concurrency control algorithm is akin to the Replicated Commit system, which layers Paxos on top of two-phase Locking (Mahmoud et al. 2013). However, by leveraging NMSI, we execute reads exclusively locally and run parallel instances of Paxos for different objects, instead of having a single instance per shard. Furthermore, when a client is either colocated with the Paxos leader or when that leader is in the closest data center to the client, Blotter can commit transactions within a single round-trip to the *closest* data center.

We have implemented Blotter as an extension to the well-known geo-replicated storage system Cassandra (Lakshman and Malik 2010). Our evaluation shows that, despite adding a small overhead in a single data center, Blotter performs much better than both Jessy and the protocols used by Spanner and also outperforms in many metrics a replication protocol that ensures snapshot isolation (Elnikety et al. 2005). This shows that Blotter can be a valid choice when several replicas are separated by high latency links, performance is critical, and the semantic differences between NMSI and snapshot isolation are acceptable by the application.

Key Research Findings

The research conducted to design and implement Blotter led to the following key contributions and findings:

1. A simple yet precise definition of non-monotonic snapshot isolation, a recent isolation model that aims at maximizing parallel execution of transactions in geo-replicated systems.
2. The design of Blotter, which combines a novel concurrency control protocol for a single data-center and its extension toward an arbitrary number of data centers by leveraging a carefully designed state machine replication

solution leveraging a variant of the Paxos algorithm.
3. The demonstration that it is possible to achieve strong consistency in geo-replicated storage systems within a modest latency envelop in fault-free runs.

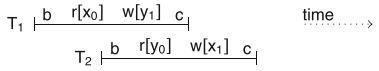
Non-monotonic Snapshot Isolation

We start by providing a specification of our target isolation level, NMSI, and discussing the advantages and drawbacks of this choice.

Snapshot Isolation Revisited

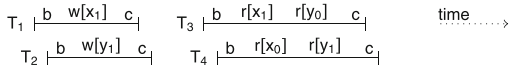
NMSI is an evolution of snapshot isolation (SI). Under SI, a transaction (logically) executes in a database snapshot taken at the transaction begin time, reflecting the writes of all transactions that committed before that instant. Reads and writes execute against this snapshot and, at commit time, a transaction can commit if there are no write-write conflicts with concurrent transactions. (In this context, two transactions are concurrent if the intervals between their begin and commit times overlap.)

Snapshot isolation exhibits the write-skew anomaly, where two concurrent transactions T_1 and T_2 start by reading x_0 and y_0, respectively, and later write y_1 and x_1, as exemplified in the following figure. This execution is admissible under snapshot isolation, as there is no write-write conflict, and each transaction has read from a database snapshot. However, this execution is not serializable.



Specification of NMSI

NMSI weakens the SI specification in two ways. First, the snapshots against which transactions execute do not have to reflect the writes of a monotonically growing set of transactions. In other words, it is possible to observe what is called a "long fork" anomaly, where there can exist two concurrent transactions t_a and t_b that commit, writing to different objects, and two other transactions that start subsequently, where one sees the effects of t_a but not t_b, and the other sees the effects of t_b but not t_a. The next figure exemplifies an execution that is admissible under NMSI but not under SI, since under SI both T_3 and T_4 would see the effects of both T_1 and T_2 because they started after the commit of T_1 and T_2.

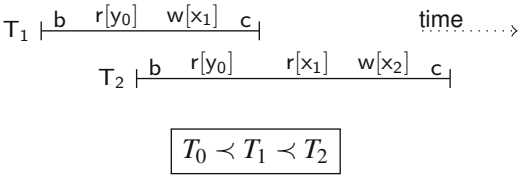


Second, instead of forcing the snapshot to reflect a subset of the transactions that committed at the transaction begin time, NMSI gives the implementation the flexibility to reflect a more convenient set of transactions in the snapshot, possibly including transactions that committed *after* the transaction began. This property, also enabled by serializability, is called *forward freshness* (Saeida Ardekani et al. 2013a).

Definition 1 (Non-Mon. Snapshot Isol. (NMSI)) An implementation of a transactional system obeys NMSI if, for any trace of the system execution, there exists a partial order $\prec$ among transactions that obeys the following rules, for any pair of transactions t_i and t_j in the trace:

1. if t_j reads a value for object x written by t_i then $t_i \prec t_j \wedge \nexists t_k$ writing to $x : t_i \prec t_k \prec t_j$
2. if t_i and t_j write to the same object x then either $t_i \prec t_j$ or $t_j \prec t_i$.

The example in Fig. 1 obeys NMSI but not SI, as the depicted partial order meets Definition 1.



Achieving Low Latency Transactions for Geo-replicated Storage with Blotter, Fig. 1 Example execution obeying NMSI but not SI. This assumes the existence of a transaction T_0 that writes the initial values for x and y

The Benefits of NMSI in Replicated Settings

NMSI weakens the specification of SI in a way that can be leveraged for improving performance in replicated settings.

The possibility of having "long forks" allows, in a replicated setting, for a single (local) replica to make a decision concerning what data the snapshot should read. This is because it is not necessary to enforce a serialization between all transaction begin and commit operations, although it is still necessary to check for write-write conflicts,

In the case of "forward freshness," this allows for a transaction to read (in most cases) the most recent version of a data object at a given replica, independently of the instant when the transaction began. This not only avoids the bookkeeping associated with tracking transaction start times but also avoids a conflict with transactions that might have committed after the transaction began.

Impact of NMSI for Applications

We analyze in turn the impact of "forward freshness" and "long forks" for application developers. Forward freshness allows a transaction t_a to observe the effects of another transaction t_b that committed after t_a began (in real time). In this case, the programmer must decide whether this is a violation of the intended application semantics, which is analogous to decide if serializability or strict serializability is the most adequate isolation level for a given application. Long forks allow two transactions to execute against different branches of a forked database state, provided there are no write-write conflicts. In practice, the main implication of this fact is that the updates made by users may not become instantly visible across all replicas. For example, this could cause two users of a social network to each think that they were the first to post a new promotion on their own wall, since they do not see each other's posts immediately (Sovran et al. 2011). Again, the programmer must reason whether this is admissible. In this case, a mitigating factor is that this anomaly does not cause the consistency of the database to break. (This is in contrast with the "write-skew" anomaly, which is present in both SI and NMSI.) Furthermore, in the particular case of our protocol, the occurrence of anomalies is very rare: for a "long fork" to occur, two transactions must commit in two different data centers, form a quorum with a third data center, and both complete before hearing from the other.

Finally, NMSI allows consecutive transactions from the same client to observe a state that reflects a set of transactions that does not grow monotonically (when consecutive transactions switch between two different branches of a long fork). However, in our algorithms, this is an unlikely occurrence, since it requires that a client connects to different data centers in a very short time span.

Blotter Protocols

We now describe the Blotter protocols, introducing first the concurrency protocol algorithm in a single data center scenario, and then how this protocol can be replicated to multiple data centers. A more detailed explanation of the protocols including their evaluation can be found elsewhere (Moniz et al. 2017).

System Model

Blotter is designed to run on top of any distributed storage system with nodes spread across one or multiple data centers. We assume that each data object is replicated at all data centers. Within each data center, data objects are replicated and partitioned across several nodes. We make no restrictions on how this intra-data center replication and partitioning takes place. We assume that nodes may fail by crashing and recover from such faults. When a node crashes, it loses its volatile state, but all data that was written to stable storage is accessible after recovery.

We use an asynchronous system model, i.e., we do not assume any known bounds on computation and communication delays. We do not prescribe a fixed bound on the number of faulty nodes within each data center – this depends on the intra-data center replication protocol used.

Blotter Architecture

The client library of Blotter exposes an API with the expected operations: `begin` a new

transaction, `read` an object given its identifier, `write` an object given its identifier and new value, and `commit` a transaction, which either returns commit or abort.

The set of protocols that comprise Blotter are organized into three different components:

Blotter intra-data center replication. At the lowest level, we run an intra-data center replication protocol to mask the unreliability of individual machines within each data center. This level must provide the protocols above it with the vision of a single logical copy (per data center) of each data object and associated metadata, which remains available despite individual node crashes. We do not prescribe a specific protocol for this layer, since any of the existing protocols that meet this specification can be used.

Blotter Concurrency Control. These are the protocols that ensure transaction atomicity and NMSI isolation in a single data center and, at the same time, are extensible to multiple data centers by serializing a single protocol step.

Inter-data Center Replication. This completes the protocol stack by replicating a subset of the steps of the concurrency control protocol across data centers. It implements state machine replication (Schneider 1990; Lamport 1978) by judiciously applying Paxos (Lamport 1998) to the concurrency control protocol to avoid unnecessary coordination across data centers.

Single Data Center Protocol

The single data center concurrency control module consists of the following three components: the client library and the transaction managers (TM), which are non-replicated components that act as a front end providing the system interface and implementing the client side of the transaction processing protocol, respectively; and the data managers (DM), which are the replicated components that manage the information associated with data objects.

Client Library. This provides the interface of Blotter, namely, *begin*, *read*, *write*, and *commit*. The *begin* and *write* operations are local to the client. *Read* operations are relayed to the TM, who returns the values and metadata for the objects that were read. The written values are buffered by the client library and only sent to the TM at *commit* time, together with the accumulated metadata for the objects that were read. This metadata is used to set the versions that running transactions must access, as explained next.

Transaction Manager (TM). The TM handles the two operations received from the clients: *read* and *commit*. For *read*s, it merely relays the request and reply to or from the data manager (DM) responsible for the object being read. Upon receiving a *commit* request, the TM acts as a coordinator of a two-phase commit (2PC) protocol to enforce the all-or-nothing atomicity property. The first phase sends a *dm-prewrite-request*, with the newly written values, to all DMs storing written objects. Each DM verifies if the write complies with NMSI. If none of the DMs identifies a violation, the TM sends the DMs a *dm-write* message containing the metadata with snapshot information aggregated from all replies on the first phase; otherwise, it sends a *dm-abort*.

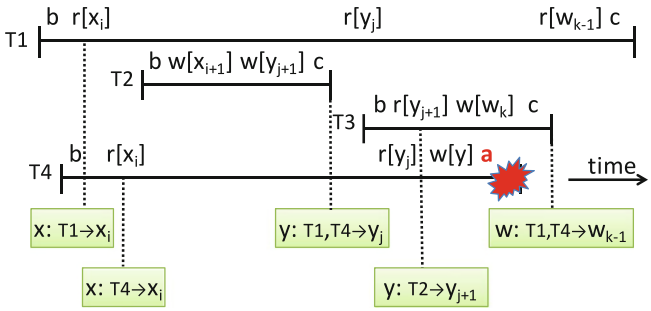
Data Manager (DM). The core of the concurrency control logic is implemented by the DM, which maintains the data and meta-data necessary to provide NMSI. Algorithm 1 presents the handlers for the three types of requests. Now, we explain how these functions enforce NMSI rules presented in our definition.

Partial order $\prec$. We use a multi-version protocol, i.e., the system maintains a list of versions for each object. This list is indexed by an integer version number, which is incremented every time a new version of the object is created (e.g., for a given object x, overwriting x_0 creates version x_1, and so on). In a multi-versioned storage, the $\prec$ relation can be defined by the version number that transactions access, namely, if t_i writes x_m and t_j writes x_n, then $t_i \prec t_j \Leftrightarrow m < n$; and if t_i writes x_m and t_j reads x_n, then $t_i \prec t_j \Leftrightarrow m \leq n$.

Algorithm 1: Single data center DM protocols

```
   // read operation
1  upon ⟨ dm-read, T, x ⟩ from TM do
2      processRead ⟨ T, x, TM ⟩;
   // prewrite operation
3  upon ⟨ dm-prewrite, T, x, value ⟩ from TM do
4      if x.prewrite ≠ ⊥ then
           // another prewrite is pending
5          x.pending ← x.pending ∪ {(T, x, value, TM)};
6      else
7          processPrewrite ⟨ T, x, value, TM ⟩;
   // write operation
8  upon ⟨ dm-write, T, x, agg-startd-before ⟩ from TM do
9      for each T' in agg-startd-before do
10         if T' not in x.snapshot then
11             x.snapshot[T'] ← x.last;
12     x.last ← x.last + 1;
13     x.value[x.last] ← x.nextvalue;
14     finishWrite ⟨ T, x, TM ⟩;
   // abort operation
15 upon ⟨ dm-abort, T, x ⟩ from TM do
16     finishWrite ⟨ T, x, TM ⟩;
   // process read operation
17 processRead ⟨ T, x, TM ⟩
18     if T ∉ x.snapshot then
19         if x.prewrite ≠ ⊥ then
20             x.buffered ← x.buffered ∪ {(T, TM)};
21             return
22         else
23             x.snapshot[T] ← x.last;
24     version ← x.snapshot[T];
25     value ← x.value[version];
26     send ⟨ read-response, T, value, {T'|x.snapshot[T'] < version} ⟩ to TM;
   // process dm-prewrite request
27 processPrewrite ⟨ T, x, value, TM ⟩
28     if x.snapshot[T] ≠ ⊥ ∧ x.snapshot[T] < x.last then
           // there is a write-write conflict
29         send ⟨ prewrite-response, reject, ⊥ ⟩ to TM;
30     else
31         x.prewrite ← T;
32         x.nextvalue ← value;
33         send ⟨ prewrite-response, accept, {T'|T' ∈ x.snapshot} ⟩ to TM;
   // clean prewrite information and serve buffered reads and pending prewrites
34 finishWrite ⟨ T, x, TM⟩
35     if x.prewrite = T then
36         x.nextvalue ← ⊥; x.prewrite ← ⊥;
37     for each (T, TM) in x.buffered do
38         processRead ⟨ T, x, TM ⟩;
39     if x.pending ≠ ⊥ then
40         (T, x, value, TM) ← removeFirst(x.pending);
41         processPrewrite ⟨ T, x, value, TM ⟩;
```

NMSI rule number 1. Rule number 1 of the definition of NMSI says that, for object x, transaction t must read the value written by the "latest" transaction that updated x (according to $\prec$). To illustrate this, consider the example run in Fig. 2. When a transaction $T1$ issues its first read operation, it can read the most recently committed version of the object, say x_i written by $T0$ (leading to $T0 \prec T$). If, subsequently, some other transaction $T2$ writes x_{i+1} ($T0 \prec T2$), then the protocol must prevent $T1$ from either reading or overwriting the values written by $T2$. Otherwise, we would have $T0 \prec T2 \prec T1$ and $T1$ should have read the value for object x written by $T2$ (i.e., x_{i+1}) instead of that written by $T0$ (i.e., x_i). Next, we detail how this is achieved first for reads, then writes, and then how to enforce the rule transitively.



Achieving Low Latency Transactions for Geo-replicated Storage with Blotter, Fig. 2 Example run

Reading the latest preceding version. The key to enforcing this requirement is to maintain state associated with each object, stating the version a running transaction must read, in case such a restriction exists. In the previous example, if $T2$ writes x_{i+1}, this state records that $T1$ must read x_i.

To achieve this, our algorithm maintains a per-object dictionary data structure ($x.snapshot$), mapping the identifier of a transaction t to a particular version of x that t either must read or has read from. Figure 2 depicts the changes to this data structure in the shaded boxes at the bottom of the figure. When the DM processes a read of x for t (function `processRead`), if the dictionary has no information for t, the most recent version is read, and this information is stored in the dictionary. Otherwise, the specified version is returned.

In the previous example, $T1$ must record the version it reads in the *x.snapshot* variable. Subsequently, when the commit of $T2$ overwrites that version of x, we are establishing that $T2 \not\prec T1$. As such, if $T2$ writes to another object y, creating y_{j+1}, then it must also force $T1$ to read the preceding version y_j. To do this, when transaction $T2$ commits, for every transaction t that read (or must read) an older version of object x (i.e., the transactions with entries in the dictionary of x), the protocol will store in the dictionary of every other object y written by $T2$ that t must read the previous version of y, unless an even older version is already prescribed (`dm-write` handler). In this particular example, $y.snapshot$ would record that $T1$ and $T4$ must read version y_j, since, at commit time, $x.snapshot$ indicates that these transactions read x_i.

Preventing illegal overwrites. In the previous example, we must also guarantee that $T1$ does not overwrite any value written by $T2$. To enforce this, it suffices to verify, at the time of the commit of transaction t, for every object written by t, if T should read its most recent version. If this is the case, then the transaction can commit, since no version will be incorrectly overwritten; otherwise, it must abort (function `processPrewrite`). In the example, $T4$ aborts, since $y.snapshot$ records that $T4$ must read y_j and a more recent version exists (y_{j+1}). Allowing $T4$ to commit and overwrite y_{j+1} would lead to $T2 \prec T4$. This breaks rule number 1 of NMSI, since it would have required $T1$ to read x_{i+1} written by $T2$, which did not occur.

Applying the rules transitively. Finally, for enforcing rule number 1 of the definition of NMSI in a transitive manner, it is also necessary to guarantee the following: if $T2$ writes x_{i+1} and y_{j+1}, and subsequently another transaction $T3$ reads y_{j+1} and writes w_k, then the protocol must also prevent $T1$ from reading or overwriting the values written by $T3$, otherwise we would have $T0 \prec T2 \prec T3 \prec T1$, and thus $T1$ should also have read x_{i+1}.

To achieve this, when transaction $T3$ (which read y_{j+1}) commits, for every transaction t that must read version y_l with $l < j + 1$ (i.e., the transactions that had entries in the dictionary of y when t_o read y), the protocol will store in the dictionary of every other object w written by $T3$ that t must read the previous version of w (if an older version is not already specified). In the example, since the state for $y.snapshot$ after $T2$ commits specifies that $T1$ must read version y_j, then, when $T3$ commits, $w.snapshot$ is updated

to state that $T1$ and $T4$ must read version w_{k-1}. This is implemented by collecting dependencies in read and pre-write functions and use these information to update the snapshot information for all objects in the write function.

NMSI rule number 2. Rule number 2 of the NMSI definition says that any pair of transactions that write the same object x must have a relative order, i.e., either $t_i \prec t_j$ or $t_j \prec t_i$. This order is defined by the version number of x created by each transaction.

Therefore, it remains to ensure that this is a partial order (i.e., no cycles). A cycle could appear if two or more transactions concurrently committed a chain of objects in a different order, e.g., if t_m wrote both x_i and y_{j+1} and t_n wrote both x_{i+1} and y_j. To prevent this, it suffices to use a two-phase commit protocol where, for each object, a single accepted prepare can be outstanding at any time.

Geo-Replication

Blotter implements geo-replication, with each object replicated in all data centers, using Paxos-based state machine replication (Lamport 1998; Schneider 1990). In this model, all replicas execute a set of client-issued commands according to a total order, thus following the same sequence of states and producing the same sequence of responses. We view each data center as a state machine replica. The state is composed by the database (i.e., all data objects and associated metadata), and the state machine commands are the `tm-read` and the `tm-commit` of the TM-DM interface.

Despite being correct, this approach has three drawbacks, which we address in detail in a separate paper (Moniz et al. 2017).

First, read operations in our concurrency control protocol are state machine commands that mutate the state, thus requiring an expensive consensus round. To address this, we leverage the fact that the NMSI properties allow for removing this information from the state machine, since the modified state only needs to be used locally.

Second, the total order of the state machine precludes the concurrent execution of two commits, even for transactions that do not conflict. In this case, we leverage the fact that the partial order required by the NMSI definition can be built by serializing the `dm-prewrite` operation on a per-object basis, instead of serializing `tm-commits` across all objects. As such, instead of having one large state machine whose state is defined by the entire database, we can have one state machine per object, with the state being the object (including its metadata), and supporting only the `dm-prewrite` operation.

Finally, each Paxos-based state machine command requires several cross-data center message delays (depending on the variant of Paxos used). We adjusted the configuration of Paxos to reduce the cross data center steps to a single round-trip (from the client of the protocol, i.e., the TM) for update transactions, by using Multi-Paxos (Lamport 1998) and configuring Paxos to only tolerate one unplanned outage of a data center. In fact, this assumption is common in existing deployed systems (Ananthanarayanan et al. 2013; Corbett et al. 2012). (Planned outages are handled by reconfiguring the Paxos membership Lamport et al. 2010.)

Examples of Applications

Blotter can be leveraged to design multiple large-scale geo-distributed applications, and in particular web applications whose semantics are compatible with the guarantees (and anomalies) associated with NMSI.

In particular, we have explored how to implement a set of the main operations of a micro-blogging platform, such as Twitter.

In our implementation of a simplistic version of Twitter, we model the timeline (sometimes called wall in the context of social network applications) as a data object in the (geo-replicated) data storage service. Furthermore, we associate with each user a set of followers and a set of followees. Each of these sets is modeled as an independent data object in the data storage.

We have supported three different user interactions. We selected these operations by relying on the model presented in Zhang et al. (2013):

- *Post-tweet* appends a tweet to the timeline of a user and its followers, which results in a transaction with many reads and writes, since one has to read the followers of the user posting the tweet and then write to the timeline of both the user and each corresponding follower.
- *Follow-user* appends new information in the set of followers to the profiles of the follower and the followee, which results in a transaction with two reads and two writes.
- *Read-timeline* reads the wall of the user issuing the operation, resulting in a single read operation.

In this case, the long fork anomaly can only result in a user observing (slightly) stale data, and perhaps even updating its own state based on that data, e.g., a user that decides to follow another can miss a few entries of that new user in his own timeline. Note that reading stale data can be addressed by waiting for the information about transactions to be propagated across all data centers in the deployment.

We also implemented the RUBiS benchmark, which models an auction site similar to eBay, on top of Blotter.

We ported the benchmark from using a relational database as the storage back end to using a key-value store. Each row of the relational database is stored with a key formed by the name of the table and the value of the primary key. We additionally store data for supporting efficient queries (namely, indexes and foreign keys).

In this benchmark, users issue operations such as selling, browsing, bidding, or buying items and consulting a personal profile that lists outstanding auctions and bids. All of these operations were implemented as Blotter transactions. In this application, the long fork anomaly that is allowed by NMSI has a few implications. In particular, when browsing the items that are available to be sold (or in auction), users can observe stale data, in the form of missing a few objects that are relevant for their queries.

Surprisingly, since bidding and buying an item involves writing to a single object in the data store in our adaptation, the long fork anomaly does not affect these semantics, since all transactions that write on this object are totally ordered by the Paxos leader that is responsible for mediating the access to it.

The experimental results that evaluate the performance of these implementations can be in a separate paper (Moniz et al. 2017).

Future Directions of Research

In this chapter, we have presented the design of Blotter, a novel system architecture and set of protocols that exploits the benefits of NMSI to improve the performance of geo-replicated transactional systems.

Blotter demonstrates the practical benefits that can be achieved by relaxing the well-known and widely used SI model. To better frame the applications that can benefit from the Blotter design, we have presented a simple yet precise specification of NMSI and discussed how this specification differs in practice from the SI specification.

An interesting direction for future research in this context is to explore automatic mechanisms to check if an existing application designed for the SI model could safely be adapted to operate under NMSI. This could be achieved by exploiting application state invariants, and modeling transactions through their preconditions and post-conditions.

Acknowledgements Computing resources for this work were provided by an AWS in Education Research Grant. The research of R. Rodrigues is funded by the European Research Council (ERC-2012-StG-307732) and by FCT (UID/CEC/50021/2013). This work was partially supported by NOVA LINCS (UID/CEC/04516/2013) and EU H2020 LightKone project (732505). This chapter is derived from Moniz et al. (2017).

References

Ananthanarayanan R, Basker V, Das S, Gupta A, Jiang H, Qiu T, Reznichenko A, Ryabkov D, Singh M, Venkataraman S (2013) Photon: fault- tolerant and scalable joining of continuous data streams. In: SIGMOD'13: proceeding of 2013 international conference on management of data, pp 577–588

Baker J, Bond C, Corbett JC, Furman J, Khorlin A, Larson J, Leon JM, Li Y, Lloyd A, Yushprakh V (2011) Megastore: providing scalable, highly available storage for interactive services. In: Proceeding of the conference on innovative data system research (CIDR), pp 223–234. http://www.cidrdb.org/cidr2011/Papers/CIDR11_Paper32.pdf

Bronson et al N (2013) Tao: facebook's distributed data store for the social graph. In: Proceeding of the 2013 USENIX annual technical conference, pp 49–60

Chang F, Dean J, Ghemawat S, Hsieh WC, Wallach DA, Burrows M, Chandra T, Fikes A, Gruber RE (2008) Bigtable: a distributed storage system for structured data. ACM Trans Comput Syst 26(2):4:1–4:26. http://doi.acm.org/10.1145/1365815.1365816

Corbett et al JC (2012) Spanner: Google's globally-distributed database. In: Proceeding of the 10th USENIX conference on operating systems design and implementation, OSDI'12, pp 251–264. http://dl.acm.org/citation.cfm?id=2387880.2387905

DeCandia et al G (2007) Dynamo: Amazon's highly available key-value store. In: Proceeding of the 21st ACM symposium on operating systems principles, pp 205–220. http://doi.acm.org/10.1145/1294261.1294281

Elnikety S, Zwaenepoel W, Pedone F (2005) Database replication using generalized snapshot isolation. In: Proceedings of the 24th IEEE symposium on reliable distributed systems, SRDS'05. IEEE Computer Society, Washington, DC, pp 73–84. https://doi.org/10.1109/RELDIS.2005.14

Hoff T (2009) Latency is everywhere and it costs you sales – how to crush it. Post at the high scalability blog. http://tinyurl.com/5g8mp2

Kraska T, Pang G, Franklin MJ, Madden S, Fekete A (2013) MDCC: multi-data center consistency. In: Proceeding of the 8th ACM European conference on computer systems, EuroSys'13, pp 113–126. http://doi.acm.org/10.1145/2465351.2465363

Lakshman A, Malik P (2010) Cassandra: a decentralized structured storage system. SIGOPS Oper Syst Rev 44(2):35–40. http://doi.acm.org/10.1145/1773912.1773922

Lamport L (1978) Time, clocks, and the ordering of events in a distributed system. Commun ACM 21(7):558–565. http://doi.acm.org/10.1145/359545.359563

Lamport L (1998) The part-time parliament. ACM Trans Comput Syst 16(2):133–169. http://doi.acm.org/10.1145/279227.279229

Lamport L, Malkhi D, Zhou L (2010) Reconfiguring a state machine. ACM SIGACT News 41(1):63–73

Lloyd W, Freedman MJ, Kaminsky M, Andersen DG (2013) Stronger semantics for low-latency geo-replicated storage. In: Proceeding of the 10th USENIX conference on networked systems design and implementation, NSDI'13, pp 313–328. http://dl.acm.org/citation.cfm?id=2482626.2482657

Mahmoud H, Nawab F, Pucher A, Agrawal D, El Abbadi A (2013) Low-latency multi-datacenter databases using replicated commit. Proc VLDB Endow 6(9):661–672. http://dl.acm.org/citation.cfm?id=2536360.2536366

Moniz H, Leitão J, Dias RJ, Gehrke J, Preguiça N, Rodrigues R (2017) Blotter: low latency transactions for geo-replicated storage. In: Proceedings of the 26th international conference on World Wide Web, International World Wide Web conferences steering committee, WWW '17, Perth, pp 263–272. https://doi.org/10.1145/3038912.3052603

Saeida Ardekani M, Sutra P, Shapiro M (2013a) Non-monotonic snapshot isolation: scalable and strong consistency for geo-replicated transactional systems. In: Proceeding of the 32nd IEEE symposium on reliable distributed systems (SRDS 2013), pp 163–172. https://doi.org/10.1109/SRDS.2013.25

Saeida Ardekani M, Sutra P, Shapiro M, Preguiça N (2013b) On the scalability of snapshot isolation. In: Euro-Par 2013 parallel processing. LNCS, vol 8097. Springer, pp 369–381. https://doi.org/10.1007/978-3-642-40047-6_39

Schneider FB (1990) Implementing fault-tolerant services using the state machine approach: a tutorial. ACM Comput Surv 22(4):299–319. http://doi.acm.org/10.1145/98163.98167

Shute J, Vingralek R, Samwel B, Handy B, Whipkey C, Rollins E, Oancea M, Littlefield K, Menestrina D, Ellner S, Cieslewicz J, Rae I, Stancescu T, Apte H (2013) F1: a distributed SQL database that scales. Proc VLDB Endow 6(11):1068–1079. https://doi.org/10.14778/2536222.2536232

Sovran Y, Power R, Aguilera MK, Li J (2011) Transactional storage for geo-replicated systems. In: Proceeding of the 23rd ACM symposium on operating systems principles, SOSP'11, pp 385–400. http://doi.acm.org/10.1145/2043556.2043592

Zhang Y, Power R, Zhou S, Sovran Y, Aguilera M, Li J (2013) Transaction chains: achieving serializability with low latency in geo-distributed storage systems. In: Proceedings of the 24th ACM symposium on operating systems principles, SOSP, pp 276–291. http://doi.acm.org/10.1145/2517349.2522729

ACID Properties in MMOGs

▸ Transactions in Massively Multiplayer Online Games

Active Disk

▸ Active Storage

Active Storage

Ilia Petrov[1], Tobias Vinçon[1], Andreas Koch[2], Julian Oppermann[2], Sergey Hardock[3], and Christian Riegger[1]
[1]Data Management Lab, Reutlingen University, Reutlingen, Germany
[2]Embedded Systems and Applications Group, TU Darmstadt, Darmstadt, Germany
[3]Databases and Distributed Systems Group, TU Darmstadt, Darmstadt, Germany

Synonyms

Active disk; Intelligent disk; Near-Data Processing; Programmable Memory/Processing In Memory (PIM)

Overview

In brief, *Active Storage* refers to an architectural hardware and software paradigm, based on co-location storage and compute units. Ideally, it will allow to execute application-defined data- or compute-intensive operations in situ, i.e., within (or close to) the physical data storage. Thus *Active Storage* seeks to minimize expensive data movement, improving performance, scalability, and resource efficiency. The effective use of *Active Storage* mandates new architectures, algorithms, interfaces, and development toolchains.

Over the last decade, we are witnessing a clear trend toward the fusion of the compute-intensive and the data-intensive paradigms on architectural, system, and application level. On the one hand, large computational tasks (e.g., simulations) tend to feed growing amounts of data into their complex computational models; on the other hand, database applications execute computationally intensive ML and analytics-style workloads on increasingly large data sets. Both result in massive *data transfers* across the memory hierarchy, which block the CPU, causing unnecessary CPU waits and thus impair performance, scalability, and resource efficiency. The root cause for this phenomenon lies in the generally low data locality as well as in traditional architectures and algorithms, which operate on the *data-to-code* principle. It requires data and program code to be transferred to the processing elements to be executed. Although *data-to-code* simplifies development and system architectures, it is inherently bounded by the *von Neumann bottleneck*.

These trends are impacted by the following recent developments: (a) *Moore's law* is said to be cooling down for different types of semiconductor elements, and *Dennard scaling* is coming to an end. The latter postulates that performance per watt grows at approximately the rate mandated by Moore's law. (Besides the scalability of cache coherence protocols, *Dennard scaling* is among the frequently quoted reasons as to why modern many-core CPUs do not have the 128 cores that would otherwise be technically possible by now – see also Muramatsu et al. (2004) and Hardavellas et al. (2011)) As a result compute performance improvements cannot be based on the expectation of increasing clock frequencies and therefore mandate changes in the hardware and software architectures. (b) Modern systems can offer much *higher levels of parallelism*, yet scalability and the effective use of parallelism are limited by the programming models as well as by amount and type of data transfers. (c) *Access gap* and Memory Wall storage (DRAM, Flash, HDD) is getting larger and cheaper; however access latencies decrease at much lower rates. This trend also contributes to slow data transfers and to blocking processing at the CPU. (d) Modern data sets are large in volume (machine data, scientific data, text) and are growing fast (Szalay and Gray 2006). (e) Modern workloads (hybrid/HTAP or analytics-based such as OLAP or ML) tend to have low data locality and incur large scans (sometimes iterative) that result in massive data transfers.

In essence, due to system architectures and processing principles, current workloads require transferring growing volumes of large data through the virtual memory hierarchy, from the physical storage location to the

processing elements, which limits performance and scalability and worsens resource and energy efficiency.

Nowadays, three important technological developments open an opportunity to counter these drawbacks. Firstly, hardware manufactures are able to *fabricate combinations of storage and compute elements at reasonable costs* and package them within the same device. Secondly, the fact that this trend covers virtually all levels of the memory hierarchy: (a) CPU and caches, (b) memory and compute, (c) storage and compute, (d) accelerators – specialized CPUs and storage, and eventually (e) network and compute. Thirdly, as magnetic/mechanical storage is being replaced with semiconductor nonvolatile technologies (Flash, Non-Volatile Memories– NVM), another key trend emerges: the device internal bandwidth, parallelism, and access latencies are significantly better than the external ones (device-to-host). This is due to various reasons: interfaces, interconnect, physical design, and architectures.

Active Storage is a concept that targets the execution data processing operations (fully or partially) in situ: within the compute elements on the respective level of the storage hierarchy, close to where data is physically stored and transfer the results back, without moving the raw data. The underlying assumptions are: (a) the result size is much smaller than the raw data, hence less frequent and smaller-sized data transfers; or (b) the in situ computation is faster and more efficient than on the host, thus higher performance and scalability or better efficiency. Related concepts are *in situ processing*, *In-Storage Processing*, *smart storage*, and *Near-Data Processing* (Balasubramonian et al. 2014). The *Active Storage* paradigms have profound impact on multiple aspects:

1. *Interfaces*: hardware and software interfaces need to be extended, and new abstractions need to be introduced. This includes device and storage interfaces and operating systems and I/O abstractions: operations and conditions, records/objects vs. blocks, atomic primitives, and transactional support.
2. *Heterogeneity* in terms of storage and computer hardware interfaces is to be addressed.
3. *Toolchain*: extensive tool support is necessary to utilize Active Storage: compilers, hardware generators, debugging and monitoring tools, and advisors.
4. *Placement* of data and computation across the hierarchy is crucial to efficiency.
5. *Workload adaptivity* is a major goal as static assignments lower the placement and collocation effects.

These challenges already attract research focus, as todays accelerators exhibit Active Storage alike characteristics in a simplistic manner, i.e., GPUs and FPGAs are defined by a considerably higher level of internal parallelism and bandwidth in contrast to their connection to the host system. Specialized computation already uses hardware programmable with high-level synthesis toolchains like TaPaSCo (Korinth et al. 2015) and co-location with storage elements and often necessitate for shared virtual memory. Classical research questions, e.g., about a dynamic workload distribution, data dependence, and flexible data placement are approached by Fan et al. (2016), Hsieh et al. (2016), and Chen and Chen (2012), respectively. The significant potential arising with Near-Data Processing is investigated under perfect conditions by Kotra et al. (2017) stating performance boosts of about 75%.

Key Research Findings

Storage

The concept of *Active Storage* is not new. Historically it is deeply rooted in the concept of *database machines* (DeWitt and Gray 1992; Boral and DeWitt 1983) developed in the 1970s and 1980s. Boral and DeWitt (1983) discusses approaches such as processor-per-track or processor-per-head as an early attempt to combine storage and simple computing elements to accelerate data processing. Existing I/O bandwidth and parallelism are claimed to be the limiting factor to justify parallel DBMS. While this conclusion is not surprising given the characteristics of magnetic/mechanical storage

combined with Amdahl's balanced systems law, it is revised with modern technologies. Modern semiconductor storage technologies (NVM, Flash) are offering high raw bandwidth and levels of parallelism. Boral and DeWitt (1983) also raises the issue of temporal locality in database applications, which has been questioned back then and is considered to be low in modern workloads, causing unnecessary data transfers. Near-Data Processing and Active Storage present an opportunity to address it.

The concept of *Active Disk* emerged toward the end of the 1990s and early 2000s. It is most prominently represented by systems such as Active Disk (Acharya et al. 1998), IDISK (Keeton et al. 1998), and Active Storage/Disk (Riedel et al. 1998). While database machines attempted to execute fixed primitive access operations, *Active Disk* targets executing application-specific code on the drive. Active Storage/Disk (Riedel et al. 1998) relies on processor-per-disk architecture. It yields significant performance benefits for I/O-bound scans in terms of bandwidth, parallelism, and reduction of data transfers. IDISK (Keeton et al. 1998) assumed a higher complexity of data processing operations compared to Riedel et al. (1998) and targeted mainly analytical workloads and business intelligence and DSS systems. Active Disk (Acharya et al. 1998) targets an architecture based on on-device processors and pushdown of custom data processing operations. Acharya et al. (1998) focusses on programming models and explores a streaming programming model, expressing data-intensive operations as so-called disklets, which are pushed down and executed on the disk processor.

An extension of the above ideas (Sivathanu et al. 2005) investigates executing operations on the RAID controller. Yet, classical RAID technologies rely on general-purpose CPUs that operate well with slow mechanical HDDs, are easily overloaded, and turn into a bottleneck with modern storage technologies (Petrov et al. 2010).

Although in the Active Disk, concept increases the scope and applicability, it is equally impacted by bandwidth limitations and high manufacturing costs. Nowadays two trends have important impact. On the one hand, semiconductor storage technologies (NVM, Flash) offer significantly higher bandwidths, lower latencies, and levels of parallelism. On the other hand, hardware vendors are able to fabricate economically combinations of storage and compute units and package them on storage devices. Both combined the result in new generation of Active Storage devices.

Smart SSDs (Do et al. 2013) or multi-stream SSDs aim to achieve better data processing performance by utilizing on device resources and pushing down data processing operations close to the data. Programming models such as *SSDlets* are being proposed. One trend is *In-Storage Processing* (Jo et al. 2016; Kim et al. 2016) that presents significant performance increase on embedded CPUs for standard DBMS operators. Combinations of storage and GPGPUs demonstrate an increase of up to 40x (Cho et al. 2013a). IBEX (Woods et al. 2013, 2014) is a system demonstrating operator pushdown on FPGA-based storage.

Do et al. (2013) is one of the first works to explore offloading parts of data processing on Smart SSDs, indicating potential of significant performance improvements (up to 2.7x) and energy savings (up to 3x). Do et al. (2013) defines a new session-based communication protocol (DBMS-SmartSSD) comprising three operations: OPEN, CLOSE, and GET. In addition they define a set of APIs for on-device functionality: Command API, Thread API, Data API, and Memory API. It does not only enable pushdown but also workload-dependent, cooperative processing. In addition, Do et al. Do et al. (2013) identify two research questions: (i) How can *Active Storage* handle the problem of on-device processing at the presence of a more recent version of the data in the buffer? (ii) What is the efficiency of operation pushdown at the presence of large main memories? The latter becomes obvious in the context of large data sets (Big Data) and computationally intensive operations.

Similarly, Seshadri et al. (2014) propose and describe a user-programmable SSD called Willow, which allows the users to augment the storage device with the application-specific logic.

In order to provide the new functionality of the SSD for a certain application, three subsystems must be appropriately modified to support a new set of RPC commands: the application, the kernel driver, and the operating system running on each storage processing unit inside the Flash SSD.

The initial ideas of Do et al. (2013) have been recently extended in Kim et al. (2016), Jo et al. (2016), and Samsung (2015). Kim et al. (2016) demonstrate between 5x and 47x performance improvement for scans and joins. Jo et al. (2016) describe a similar approach for In-Storage Computing based on the Samsung PM1725 SSD with ISC option (Samsung 2015) integrated in MariaDB. Other approaches (Cho et al. 2013b; Tiwari et al. 2013). Tiwari et al. (2013) stress the importance of in situ processing.

Woods et al. (2014, 2013) demonstrate with Ibex an intelligent storage engine for commodity-relational databases. By off-loading complex queries operators, they tackle the bandwidth bottlenecks arising when moving large amounts of data from storage to processing nodes. In addition, the energy consumption is reduced due to the usage of FPGAs rather than general-purpose processors. Ibex supports aggregation (GROUP By), projection, and selection. Najafi et al. (2013) and Sadoghi et al. (2012) explore approaches for flexible query processing on FPGAs.

JAFAR is an *Active Storage* approach for column stores by Xi et al. (2015) and Babarinsa and Idreos (2015). JAFAR is based on MonetDB and aims at reducing data transfers, hence pushing size reducing DB operators such as selections; joins are not considered. Xi et al. (2015) stress the importance of on-chip accelerators but do not consider *Active Storage* and accelerators for complex computations in situ.

Memory: Processing-In-Memory (PIM)

Manufacturing costs of DRAM decrease, and memory volume increases steadily, while access latencies improve at a significantly lower rate yielding the so-called Memory Wall (Wulf and McKee 1995). On the one hand, technologies such as Hybrid Memory Cube (HMC) attempt to address this issue by locating processing units close to memory and by utilizing novel interfaces alike. On the other hand, new types of memory are introduced, characterized by an even higher density and therefore larger volumes, shorter latencies, and higher bandwidth and internal parallelism, as well as non-volatile persistence behavior.

Balasubramonian (2016) discusses in his article the features that can be meaningfully added to memory devices. Not only do these features execute parts of an application, but they may also take care of auxiliary operations that maintain high efficiency, reliability, and security. Research combining memory technologies with the Active Storage concept in general, often referred to as Processing-In-Memory (PIM), is very versatile. In the late 1990 (Patterson et al. 1997), proposed IRAM as a first attempt to address the Memory Wall, by unifying processing logic and DRAM, starting with general research question of the computer science like communication, interfaces, cache coherence, or address schemes. Hall et al. (1999) purpose combining their Data-IntensiVe Architecture (DIVA) PIM memories with external host processors and defining a PIM-to-PIM interconnect; Vermij et al. (2017) present an extension to the CPU architecture to enable NDP capabilities close to the main memory by introducing a new component attached to the system bus responsible for the communication; Boroumand et al. (2017) propose a new hardware cache coherence mechanism designed specifically for PIM; Picorel et al. (2017) show that the historically important flexibility to map any virtual page to any page frame is unnecessary regarding NDP and introduce Distributed Inverted Page Table (DIPTA) as an alternative near-memory structure.

Studying the upcomming new interface HMC, Azarkhish et al. (2017) analyzed its support for NDP in a modular and flexible fashion. The authors propose a fully backward compatible extension to the standard HMC called smart memory cube and design a high-bandwidth, low-latency, and AXI(4)-compatible logic base interconnect, featuring a novel address scrambling mechanism for the reduction in

vault/bank conflicts. A completely different approach to tackle Active Storage in today's memory is presented in Gao et al. (2016b). It introduces DRAF, an architecture for bit-level reconfigurable logic that uses DRAM subarrays to implement dense lookup tables because FPGAs introduce significant area and power overheads, making it difficult to use them in datacenter servers. Leaving the existing sequential programming models in touch by extending the instruction set architecture, Ahn et al. (2015) proposes new PIM-enabled instructions. Firstly, the proposed instruction set is interoperable with existing programming models, cache coherence protocols, and virtual memory mechanisms. Secondly, the instructions can be executed either in-memory or on the processors depending on the data locality. A conceptual near-memory acceleration architecture is presented by Kim et al. (2017b) claiming the need for adopting a high-level synthesis approach. In Lim and Park (2017), kernel operations that can greatly improve with PIM are analyzed resulting in the necessity of three categories of processing engines for NDP logic – in-order core, a coercegrain reconfigurable processor (CGRA), and dedicated hardware.

Proposing Caribou, an intelligent distributed storage layer, István et al. (2017) target NDP on DRAM/NVRAM storage over the network through a simple key-value store interface. Utilizing FPGAs, each storage node provides high-bandwidth NDP capabilities and fault tolerance through replication by Zookeeper's atomic broadcasts.

The application of the Active Storage concept on memories besides data management is often based on analytical scenarios or neural networks but comprises a variety of different approaches. Gao et al. (2016a) develop hardware and software for an NDP architecture for in-memory analytic frameworks, including MapReduce, graph processing, and deep neural networks. One year later, Gao et al. (2017) presents the hardware architecture and software scheduling and partitioning techniques for TETRIS, a scalable neural network accelerator using 3D memory. For a similar use case, (Chi et al. 2016) proposes PRIME, providing microarchitecture and circuit designs for a ReRAM-based PIM architecture enabling morphable functions with insignificant area overhead. A compiler-based allocation strategy approach for PIM architectures is proposed by Memoultion Wang et al. (2017). Focusing on convolutional neural networks, it offers thread-level parallelism that can fully exploit the computational power-embedded processors. Another hardware/software co-design for data analytics is presented by the Mondrian Data Engine (Drumond et al. 2017). It focuses on sequential access patterns to enable simple hardware that access memory in streams. A standardization of NDP architecture, in order for PIM stacks to be used for different GPU architectures is proposed by Kim et al. (2017a). Their approach intend to allow data to be spread across multiple memory stacks as is the norm in high-performance systems.

Active Network

Having only the slight variation that data is not persisted at any time but rather streamed through, active networks are another very widespread application of the Active Storage concept. Powerful processing elements near the network adapters or often integrated to the network controller itself as a System-on-Chip (SoC) is not solely responsible for the conventional protocol interpretation anymore but also take over further tasks like security verifications and scheduling of in-transit services and data processing or simply improving the network performance.

Tennenhouse and Wetherall (1996) first introduced the term Active Network as an approach for performing sophisticated computation within the network. By injecting customized program features into the nodes of the network, it is possible to execute these at each traversed network router/switch. Continuing the research of Tennenhouse and Wetherall (1996) and Sykora and Koutny (2010) present an Active Network node called Smart Active Node (SAN). Thereby, they focus on its ability to translate data flow transparently between IP network and active network to further improve performance of IP applications.

Often Active Network is also referred as software-defined network (SDN) and comprises

already an advanced state of research. Especially the area around security comprises progressive research in authentication and authorization, access control, threats, and DoS attacks as summarized by Ahmad et al. (2015). But also the utilization of RDMA-capable network became a trend since the demand on higher bandwidth arose with the introduction of dedicated GPUs in the computation. Ren et al. (2017) propose iRDMA, an RDMA-based parameter server architecture optimized for high-performance network environment supporting both GPU- and CPU-based training.

Cross-References

- Big Data and Exascale Computing
- Computer Architecture for Big Data
- Emerging Hardware Technologies
- Energy Implications of Big Data

References

Acharya A, Uysal M, Saltz J (1998) Active disks: Programming model, algorithms and evaluation. In: Proceedings of the eighth international conference on architectural support for programming languages and operating systems, ASPLOS VIII, pp 81–91

Ahmad I, Namal S, Ylianttila M, Gurtov A (2015) Security in software defined networks: a survey. IEEE Commun Surv Tutorials 17(4):2317–2346

Ahn J, Yoo S, Mutlu O, Choi K (2015) PIM-enabled instructions: a low-overhead, locality-aware processing-in-memory architecture. In: Proceeding of 42nd annual international symposium on computer architecture (ISCA'15), pp 336–348

Azarkhish E, Pfister C, Rossi D, Loi I, Benini L (2017) Logic-base interconnect design for near memory computing in the smart memory cube. IEEE Trans Very Large Scale Integr VLSI Syst 25:210–223

Babarinsa OO, Idreos S (2015) Jafar: near-data processing for databases. In: SIGMOD

Balasubramonian R (2016) Making the case for feature-rich memory systems: the march toward specialized systems. IEEE Solid-State Circuits Mag 8(2):57–65

Balasubramonian R, Chang J, Manning T, Moreno JH, Murphy R, Nair R, Swanson S (2014) Near-data processing: insights from a micro-46 workshop. IEEE Micro 34(4):36–42

Boral H, DeWitt DJ (1983) Database machines: an idea whose time has passed? A critique of the future of database machines. In: Leilich H-O, Missikoff M (eds) Database machines. Springer, Berlin/Heidelberg, pp 166–187

Boroumand A, Ghose S, Patel M, Hassan H, Lucia B, Hsieh K, Malladi KT, Zheng H, Mutlu O (2017) LazyPIM: an efficient cache coherence mechanism for processing-in-memory. IEEE Comput Archit Lett 16(1):46–50

Chen C, Chen Y (2012) Dynamic active storage for high performance I/O. In: 2012 41st international conference on Parallel Processing. IEEE, pp 379–388

Chi P, Li S, Xu C, Zhang T, Zhao J, Liu Y, Wang Y, Xie Y (2016) PRIME: a novel processing-in-memory architecture for neural network computation in ReRAM-based main memory. In: Proceeding of 2016 43rd international symposium on computer architecture (ISCA 2016), pp 27–39

Cho BY, Jeong WS, Oh D, Ro WW (2013a) Xsd: accelerating mapreduce by harnessing the GPU inside an SSD. In: WoNDP: 1st workshop on near-data processing in conjunction with IEEE MICRO-46

Cho S, Park C, Oh H, Kim S, Yi Y, Ganger GR (2013b) Active disk meets flash: a case for intelligent SSDs. In: Proceeding of ICS, pp 91–102

DeWitt D, Gray J (1992) Parallel database systems: the future of high performance database systems. Commun ACM 35(6):85–98

Do J, Kee YS, Patel JM, Park C, Park K, DeWitt DJ (2013) Query processing on smart SSDs: opportunities and challenges. In: Proceeding of SIGMOD, pp 1221–1230

Drumond M, Daglis A, Mirzadeh N, Ustiugov D, Picorel J, Falsafi B, Grot B, Pnevmatikatos D (2017) The mondrian data engine. ACM SIGARCH Comput Archit News 45(2):639–651

Fan S, He Z, Tan H (2016) An active storage system with dynamic task assignment policy. In: 2016 12th international conference on natural computation fuzzy system and knowledge discovery (ICNC-FSKD 2016), pp 1421–1427

Gao M, Ayers G, Kozyrakis C (2016a) Practical near-data processing for in-memory analytics frameworks. Parallel architecture and compilation techniques – Conference proceedings, PACT 2016-March, pp 113–124

Gao M, Delimitrou C, Niu D, Malladi KT, Zheng H, Brennan B, Kozyrakis C (2016b) DRAF: a low-power DRAM-based reconfigurable acceleration fabric. In: 2016 ACM/IEEE 43rd annual international symposium on computer architecture. IEEE, pp 506–518

Gao M, Pu J, Yang X, Horowitz M, Kozyrakis C (2017) TETRIS: scalable and efficient neural network acceleration with 3D memory. ASPLOS 51(2):751–764

Hall M, Kogge P, Koller J, Diniz P, Chame J, Draper J, LaCoss J, Granacki J, Brockman J, Srivastava A, Athas W, Freeh V, Shin J, Park J (1999) Mapping irregular applications to DIVA, a PIM-based data-intensive architecture. In: ACM/IEEE conference on supercomputing (SC 1999), p 57

Hardavellas N, Ferdman M, Falsafi B, Ailamaki A (2011) Toward dark silicon in servers. IEEE Micro 31(4):6–15

Hsieh K, Ebrahim E, Kim G, Chatterjee N, O'Connor M, Vijaykumar N, Mutlu O, Keckler SW (2016)

Transparent offloading and mapping (TOM): enabling programmer-transparent near-data processing in GPU systems. In: Proceeding of 2016 43rd international symposium on computer architecture (ISCA 2016), pp 204–216

István Z, Sidler D, Alonso G (2017) Caribou: intelligent distributed storage. Proc VLDB Endow 10(11): 1202–1213

Jo I, Bae DH, Yoon AS, Kang JU, Cho S, Lee DDG, Jeong J (2016) Yoursql: a high-performance database system leveraging in-storage computing. Proc VLDB Endow 9:924–935

Keeton K, Patterson DA, Hellerstein JM (1998) A case for intelligent disks (idisks). SIGMOD Rec 27(3):42–52

Kim G, Chatterjee N, O'Connor M, Hsieh K (2017a) Toward standardized near-data processing with unrestricted data placement for GPUs. In: Proceeding of international conference on high performance computing networking, storage and analysis (SC'17), pp 1–12

Kim NS, Chen D, Xiong J, Hwu WMW (2017b) Heterogeneous computing meets near-memory acceleration and high-level synthesis in the post-moore era. IEEE Micro 37(4):10–18

Kim S, Oh H, Park C, Cho S, Lee SW, Moon B (2016) In-storage processing of database scans and joins. Inf Sci 327(C):183–200

Korinth J, Chevallerie Ddl, Koch A (2015) An open-source tool flow for the composition of reconfigurable hardware thread pool architectures. In: Proceedings of the 2015 IEEE 23rd annual international symposium on field-programmable custom computing machines (FCCM'15). IEEE Computer Society, Washington, DC, pp 195–198

Kotra JB, Guttman D, Chidambaram Nachiappan N, Kandemir MT, Das CR (2017) Quantifying the potential benefits of on-chip near-data computing in manycore processors. In: 2017 IEEE 25th international symposium on modeling, analysis, and simulation of computer and telecommunication system, pp 198–209

Lim H, Park G (2017) Triple engine processor (TEP): a heterogeneous near-memory processor for diverse kernel operations. ACM Ref ACM Trans Arch Code Optim Artic 14(4):1–25

Muramatsu B, Gierschi S, McMartin F, Weimar S, Klotz G (2004) If you build it, will they come? In: Proceeding of 2004 joint ACM/IEEE Conference on digital libraries (JCDL'04) p 396

Najafi M, Sadoghi M, Jacobsen HA (2013) Flexible query processor on FPGAs. Proc VLDB Endow 6(12):1310–1313

Patterson D, Anderson T, Cardwell N, Fromm R, Keeton K, Kozyrakis C, Thomas R, Yelick K (1997) A case for intelligent ram. IEEE Micro 17(2):34–44

Petrov I, Almeida G, Buchmann A, Ulrich G (2010) Building large storage based on flash disks. In: Proceeding of ADMS'10

Picorel J, Jevdjic D, Falsafi B (2017) Near-Memory Address Translation. In: 2017 26th international conference on Parallel architectures and compilation techniques, pp 303–317, 1612.00445

Ren Y, Wu X, Zhang L, Wang Y, Zhang W, Wang Z, Hack M, Jiang S (2017) iRDMA: efficient use of RDMA in distributed deep learning systems. In: IEEE 19th international conference on high performance computing and communications, pp 231–238

Riedel E, Gibson GA, Faloutsos C (1998) Active storage for large-scale data mining and multimedia. In: Proceedings of the 24rd international conference on very large data bases (VLDB'98), pp 62–73

Sadoghi M, Javed R, Tarafdar N, Singh H, Palaniappan R, Jacobsen HA (2012) Multi-query stream processing on FPGAs. In: 2012 IEEE 28th international conference on data engineering, pp 1229–1232

Samsung (2015) In-storage computing. http://www.flash-memorysummit.com/English/Collaterals/Proceedings/2015/20150813_S301D_Ki.pdf

Seshadri S, Gahagan M, Bhaskaran S, Bunker T, De A, Jin Y, Liu Y, Swanson S (2014) Willow: a user-programmable SSD. In: Proceeding of OSDI'14

Sivathanu M, Bairavasundaram LN, Arpaci-Dusseau AC, Arpaci-Dusseau RH (2005) Database-aware semantically-smart storage. In: Proceedings of the 4th conference on USENIX conference on file and storage technologies (FAST'05), vol 4, pp 18–18

Sykora J, Koutny T (2010) Enhancing performance of networking applications by IP tunneling through active networks. In: 9th international conference on networks (ICN 2010), pp 361–364

Szalay A, Gray J (2006) 2020 computing: science in an exponential world. Nature 440:413–414

Tennenhouse DL, Wetherall DJ (1996) Towards an active network architecture. ACM SIGCOMM Comput Commun Rev 26(2):5–17

Tiwari D, Boboila S, Vazhkudai SS, Kim Y, Ma X, Desnoyers PJ, Solihin Y (2013) Active flash: towards energy-efficient, in-situ data analytics on extreme-scale machines. In: Proceeding of FAST, pp 119–132

Vermij E, Fiorin L, Jongerius R, Hagleitner C, Lunteren JV, Bertels K (2017) An architecture for integrated near-data processors. ACM Trans Archit Code Optim 14(3):30:1–30:25

Wang Y, Zhang M, Yang J (2017) Towards memory-efficient processing-in-memory architecture for convolutional neural networks. In: Proceeding 18th ACM SIGPLAN/SIGBED conference on languages compilers, and tools for embedded systems (LCTES 2017), pp 81–90

Woods L, Teubner J, Alonso G (2013) Less watts, more performance: an intelligent storage engine for data appliances. In: Proceeding of SIGMOD, pp 1073–1076

Woods L, István Z, Alonso G (2014) Ibex: an intelligent storage engine with support for advanced sql offloading. Proc VLDB Endow 7(11):963–974

Wulf WA, McKee SA (1995) Hitting the memory wall: implications of the obvious. SIGARCH CAN 23(1):20–24

Xi SL, Babarinsa O, Athanassoulis M, Idreos S (2015) Beyond the wall: near-data processing for databases. In: Proceeding of DaMoN, pp 2:1–2:10

Ad Hoc Benchmark

▸ TPC-H

Ad Hoc Query Processing

▸ Robust Data Partitioning

Adaptive Partitioning

▸ Robust Data Partitioning

Adaptive Windowing

Ricard Gavaldà
Universitat Politècnica de Catalunya, Barcelona, Spain

Synonyms

ADWIN algorithm

Definitions

Adaptive Windowing is a technique used for the online analysis of data streams to manage changes in the distribution of the data. It uses the standard idea of sliding window over the data, but, unlike other approaches, the size of the window is not fixed and set a priori but changed dynamically as a function of the data. The window is maintained at all times to the maximum length consistent with the assumption that there is no change in the data contained in it.

Context

Many modern sources of data are best viewed as data streams: a potentially infinite sequence of data items that arrive one at a time, usually at high and uncontrollable speed. One wants to perform various analysis tasks on the stream in an online, rather than batch, fashion. Among these tasks, many consist of building models such as creating a predictor, forming clusters, or discovering frequent patterns. The source of data may evolve over time, that is, its statistical properties may vary, and often one is interested in keeping the model accurate with respect to the current distribution of the data, rather than that of the past data.

The ability of model-building methods to handle evolving data streams is one of the distinctive concerns of data stream mining, compared to batch data mining (Gama et al. 2014; Ditzler et al. 2015; Bifet et al. 2018). Most of the approaches to this problem in the literature fall into one of the following three patterns or a combination thereof.

One, the algorithm keeps a sliding window over the stream that stores a certain quantity of the most recently seen items. The algorithm then is in charge of keeping the model accurate with respect to the items in the window. *Sliding* means that every newly arrived item is added to the front of the window and that the oldest elements are dropped from the tail of the window. Dropping policies may vary. To keep a window of a constant size (denoted W hereafter), one stores the first W elements and then drops exactly one element for each one that is added.

Two, each item seen so far is associated with a weight that changes over time. The model-building algorithm takes into account the weight of each element in maintaining its model, so that elements with higher weight influence more the model behavior. One can, for example, fix a constant $\lambda < 1$ called the *decay factor* and establish that the importance of every item gets decreased (multiplied) by λ at each time step; this implies that the importance of the item that arrived t time steps ago is λ^t its initial one, that is, weights decrease exponentially fast. This policy is the basis of the EWMA (Exponentially Weighted Moving Average) estimator for the average of some statistic of the stream.

Three, the model builder monitors the stream with a *change detection* algorithm that raises a

flag when it finds evidence that the distribution of the stream items has changed. When this happens, the model builder revises the current model or discards the model and builds a new one with fresh data. Usually, change detection algorithms monitor one statistic or a small number of statistics of the stream, so they will not detect every possible kind of change, which is in general computationally unfeasible. Two easy-to-implement change detection algorithms for streams of real values are the CUSUM (Cumulative Sum) and Page-Hinkley methods (Basseville and Nikiforov 1993; Gama et al. 2014). Roughly speaking, both methods monitor the average of the items in the stream; when the recent average differs from the historical average by some threshold related to the standard deviation, they declare change.

Methods such as EWMA, CUSUM, and Page-Hinkley store a constant amount of real values, while window-based methods require, if implemented naively, memory linear in W. On the other hand, keeping a window provides more information usable by the model builder, namely, the instances themselves.

A disadvantage of all three methods as described is that they require the user to provide parameters containing assumptions about the magnitude or frequency of changes. Fixing a window size to have size W means that the user expects the last W items to be relevant, so that there is little change within them, but that items older than W are suspect of being irrelevant. Fixing a parameter λ in the EWMA estimator to a value close to 1 indicates that change is expected to be rare or slow, while a value closer to 0 suggests that change may be frequent or abrupt. A similar assumption can be found in the choice of parameters for the CUSUM and Page-Hinkley tests.

In general, these methods face the trade-off between reaction time and variance in the data: The user would like them to react quickly to changes (which happens with, e.g., smaller values of W and λ) but also have a low number of false positives when no change occurs (which is achieved with larger values of W and λ). In the case of sliding windows, in general one wants to have larger values of W when no change occurs, because models built from more data tend to be more accurate, but smaller values of W when the data is changing, so that the model ignores obsolete data. These trade-offs are investigated by Kuncheva and Žliobaitė (2009).

Methods

Adaptive windowing schemes use sliding windows whose size increases or decreases in response to the change observed in the data. They intend to free the user from having to guess expected rates of change in the data, which may lead to poor performance if the guess is incorrect or if the rate of change is different at different moments. Three methods are reviewed in this section, particularly the ADWIN method.

The Drift Detection Method (DDM) proposed by Gama et al. (2004) applies to the construction of two-class predictive models. It is based on the theoretical and practical observation that the empirical error of a predictive model should decrease or remain stable as the model is built with more and more data from a stationary distribution, assuming one controls overfitting. Therefore, when the empirical error instead increases, this is evidence that the distribution in the data has changed.

More precisely, let p_t denote the error rate of the predictor at time t, and $s_t = \sqrt{p_t(1-p_t)/t}$ its standard deviation. DDM stores the smallest value $p_{\min}$ of the error rates observed up to time t, and the standard deviation $s_{\min}$ at that point. Then at time t:

- If $p_t + s_t \geq p_{\min} + 2 \cdot s_{\min}$, DDM declares a warning. It starts storing examples in anticipation of a possible declaration of change.
- If $p_t + s_t \geq p_{\min} + 3 \cdot s_{\min}$, DDM declares a change. The current predictor is discarded and a new one is built using the stored examples. The values for $p_{\min}$ and $s_{\min}$ are reset as well.

This approach is generic and fast enough for the use in the streaming setting, but it has the drawback that it may be too slow in responding to changes. Indeed, since p_t is computed on the

basis of all examples since the last change, it may take many observations after the change to make p_t significantly larger than $p_{\min}$. Also, for slow change, the number of examples retained in memory may become large.

An evolution of this method that uses EWMA to estimate the errors is presented and thoroughly analyzed by Ross et al. (2012).

The OLIN method due to Last (2002) and Cohen et al. (2008) also adjusts dynamically the size of the sliding window used to update a predictive model, in order to adapt it to the rate of change in nonstationary data streams. OLIN uses the statistical significance of the difference between the training and the validation accuracy of the current model as an indicator of data stability. Higher stability means that the window can be enlarged to use more data to build a predictor, and lower stability implies shrinking the window to discard stale data. Although described for one specific type of predictor ("Information Networks") in Last (2002) and Cohen et al. (2008), the technique should apply many other types.

The ADWIN (ADaptive WINdowing) algorithm is due to Bifet and Gavaldà (2007) and Bifet (2010). Its purpose is to be a self-contained module that can be used in the design of data stream algorithms (for prediction or classification, but also for other tasks) to detect and manage change in a well-specified way. In particular, it wants to resolve the trade-off between fast reaction to change and reduced false alarms without relying on the user guessing an ad hoc parameter. Intuitively, the ADWIN algorithm resolves this trade-off by checking change at many scales simultaneously or trying many sliding window sizes simultaneously. It should be used when the scale of change rate is unknown, and this is problematic enough to compensate a moderate increase in computational effort.

More precisely, ADWIN maintains a sliding window of real numbers that are derived from the data stream. For example, elements in the window could be W bits indicating whether the current predictive model was correct on the last W stream items; the window then can be used to estimate the current error rate of the predictor. In a clustering task, it could instead keep track of the fraction of outliers or cluster quality measures, and in a frequent pattern mining task, the number of frequent patterns that appear in the window. Significant variation inside the window of any of these measures indicates distribution change in the stream. ADWIN is parameterized by a confidence parameter $\delta \in (0, 1)$ and a statistical test $T(W_0, W_1, \delta)$; here W_0 and W_1 are two windows, and T decides whether they are likely to come from the same distribution. A good test should satisfy the following criteria:

- If W_0 and W_1 were generated from the same distribution (no change), then with probability at least $1 - \delta$ the test says "no change."
- If W_0 and W_1 were generated from two different distributions whose average differs by more than some quantity $\epsilon(W_0, W_1, \delta)$, then with probability at least $1 - \delta$ the test says "change."

When there has been change in the average but its magnitude is less than $\epsilon(W_0, W_1, \delta)$, no claims can be made on the validity of the test's answer. Observe that in reasonable tests, ϵ decreases as the sizes of W_0 and W_1 increase, that is, as the test sees larger samples. ADWIN applies the test to a number of partitions of its sliding window into two parts, W_0 containing the oldest elements and W_1 containing the newer ones. Whenever $T(W_0, W_1, \delta)$ returns "change", ADWIN drops W_0, so the sliding window becomes W_1. In this way, at all times, the window is kept of the maximum length such that there is no proof of change within it. In times without change, the window can keep growing indefinitely (up to a maximum size, if desired).

In order to be efficient in time and memory, ADWIN represents its sliding window in a compact way, using the Exponential Histogram data structure due to Datar et al. (2002). This structure maintains a summary of a window by means of a chain of buckets. Older bits are summarized and compressed in coarser buckets with less resolution. A window of length W is stored in only $O(k \log W)$ buckets, each using a constant amount of memory words, and yet

the histogram returns an approximation of the average of the window values that is correct up to a factor of $1/k$. ADWIN does not check all partitions of its window into pairs (W_0, W_1), but only those at bucket boundaries. Therefore, it performs $O(k \log W)$ tests on the sliding window for each stream item. The standard implementation of ADWIN uses $k = 5$ and may add the rule that checks are only performed only every t number of items for efficiency – at the price of a delay of up to t time steps in detecting a change.

In Bifet and Gavaldà (2007) and Bifet (2010), a test based on the so-called Hoeffding bound is proposed, which can be rigorously proved to satisfy the conditions above for a "good test." Based on this, rigorous guarantees on the false positive rate and false negative rate of ADWIN are proved in Bifet and Gavaldà (2007) and Bifet (2010). However, this test is quite conservative and will be slow to detect change. In practice, tests based on the normal approximation of a binomial distribution should be used, obtaining faster reaction time for a desired false positive rate.

Algorithms for mining data streams will probably store their own sliding window of examples to revise/rebuild their models. One or several instances of ADWIN can be used to inform the algorithm of the occurrence of change and the optimal window size it should use. The time and memory overhead is moderate (logarithmic in the size of the window) and often negligible compared with the cost of the main algorithm itself.

Several change detection methods for streams were evaluated by Gama et al. (2009). The conclusions were that Page-Hinkley and ADWIN were the most appropriate. Page-Hinkley exhibited a high rate of false alarms, and ADWIN used more resources, as expected.

Examples of Application

ADWIN has been applied in the design of streaming algorithms and in applications that need to deal with nonstationary streams.

At the level of algorithm design, it was used by Bifet and Gavaldà (2009) to give a more adaptive version of the well-known CVFDT algorithm for building decision trees from streams due to Hulten et al. (2001); ADWIN improves it by replacing hard-coded constants in CVFDT for the sizes of sliding windows and the duration of testing and training phases with data-adaptive conditions. A similar approach was used by Bifet et al. (2010b) for regression trees using perceptrons at the leaves. In Bifet et al. (2009a,b, 2010a, 2012), ADWIN was used in the context of ensemble classifiers to detect when a member of the ensemble is underperforming and needs to be replaced. In the context of pattern mining, ADWIN was used to detect change and maintain the appropriate sliding window size in algorithms that extract frequent graphs and frequent trees from streams of graphs and XML trees (Bifet et al. 2011b; Bifet and Gavaldà 2011). In the context of process mining, ADWIN is used by Carmona and Gavaldà (2012) to propose a mechanism that helps in detecting changes in the process, localize and characterize the change once it has occurred, and unravel process evolution.

ADWIN is a very generic, domain-independent mechanism that can be plugged into a large variety of applications. Some examples include the following:

- Bakker et al. (2011) in an application to detect stress situations in the data from wearable sensors
- Bifet et al. (2011a) in application to detect sentiment change in Twitter streaming data
- Pechenizkiy et al. (2009) as the basic detection mechanism in a system to control the stability and efficiency of industrial fuel boilers
- Talavera et al. (2015) in an application to segmentation of video streams

Future Research

A main research problem continues to be the efficient detection of change in multidimensional data. Algorithms as described above (OLIN,

DDM, and ADWIN) deal, strictly speaking, with the detection of change in a unidimensional stream of real values; it is assumed that this stream of real values, derived from the real stream, will change significantly when there is significant change in the multidimensional stream.

Several instances of these detectors can be created to monitor different parts of the data space or to monitor different summaries or projections thereof, as in, e.g., Carmona and Gavaldà (2012). However, there is no guarantee that all real changes will be detected in this way. Papapetrou et al. (2015) and Muthukrishnan et al. (2007) among others have proposed efficient schemes to directly monitor change in change in multidimensional data. However, the problem in its full generality is difficult to scale to high dimensions and arbitrary change, and research in more efficient mechanisms usable in streaming scenarios is highly desirable.

Cross-References

- Definition of Data Streams
- Introduction to Stream Processing Algorithms
- Management of Time

References

Bakker J, Pechenizkiy M, Sidorova N (2011) What's your current stress level? Detection of stress patterns from GSR sensor data. In: 2011 IEEE 11th international conference on data mining workshops (ICDMW), Vancouver, 11 Dec 2011, pp 573–580. https://doi.org/10.1109/ICDMW.2011.178

Basseville M, Nikiforov IV (1993) Detection of abrupt changes: theory and application. Prentice-Hall, Upper Saddle River. http://people.irisa.fr/Michele.Basseville/kniga/. Accessed 21 May 2017

Bifet A (2010) Adaptive stream mining: pattern learning and mining from evolving data streams, frontiers in artificial intelligence and applications, vol 207. IOS Press. http://www.booksonline.iospress.nl/Content/View.aspx?piid=14470

Bifet A, Gavaldà R (2007) Learning from time-changing data with adaptive windowing. In: Proceedings of the seventh SIAM international conference on data mining, 26–28 Apr 2007, Minneapolis, pp 443–448. https://doi.org/10.1137/1.9781611972771.42

Bifet A, Gavaldà R (2009) Adaptive learning from evolving data streams. In: Advances in intelligent data analysis VIII, proceedings of the 8th international symposium on intelligent data analysis, IDA 2009, Lyon, Aug 31–Sept 2 2009, pp 249–260. https://doi.org/10.1007/978-3-642-03915-7_22

Bifet A, Gavaldà R (2011) Mining frequent closed trees in evolving data streams. Intell Data Anal 15(1):29–48. https://doi.org/10.3233/IDA-2010-0454

Bifet A, Holmes G, Pfahringer B, Gavaldà R (2009a) Improving adaptive bagging methods for evolving data streams. In: Advances in machine learning, proceedings of the first Asian conference on machine learning, ACML 2009, Nanjing, 2-4 Nov 2009, pp 23–37. https://doi.org/10.1007/978-3-642-05224-8_4

Bifet A, Holmes G, Pfahringer B, Kirkby R, Gavaldà R (2009b) New ensemble methods for evolving data streams. In: Proceedings of the 15th ACM SIGKDD international conference on knowledge discovery and data mining, KDD '09. ACM, New York, pp 139–148. http://doi.acm.org/10.1145/1557019.1557041

Bifet A, Holmes G, Pfahringer B (2010a) Leveraging bagging for evolving data streams. In: European conference on machine learning and knowledge discovery in databases, proceedings, part I of the ECML PKDD 2010, Barcelona, 20–24 Sept 2010, pp 135–150. https://doi.org/10.1007/978-3-642-15880-3_15

Bifet A, Holmes G, Pfahringer B, Frank E (2010b) Fast perceptron decision tree learning from evolving data streams. In: Advances in knowledge discovery and data mining, proceedings, part II of the 14th Pacific-Asia conference, PAKDD 2010, Hyderabad, 21-24 June 2010, pp 299–310. https://doi.org/10.1007/978-3-642-13672-6_30

Bifet A, Holmes G, Pfahringer B, Gavaldà R (2011a) Detecting sentiment change in twitter streaming data. In: Proceedings of the second workshop on applications of pattern analysis, WAPA 2011, Castro Urdiales, 19-21 Oct 2011, pp 5–11. http://jmlr.csail.mit.edu/proceedings/papers/v17/bifet11a/bifet11a.pdf

Bifet A, Holmes G, Pfahringer B, Gavaldà R (2011b) Mining frequent closed graphs on evolving data streams. In: Proceedings of the 17th ACM SIGKDD international conference on knowledge discovery and data mining, KDD '11. ACM, New York, pp 591–599. http://doi.acm.org/10.1145/2020408.2020501

Bifet A, Frank E, Holmes G, Pfahringer B (2012) Ensembles of restricted hoeffding trees. ACM TIST 3(2):30:1–30:20. http://doi.acm.org/10.1145/2089094.2089106

Bifet A, Gavaldà R, Holmes G, Pfahringer B (2018) Machine learning for data streams, with practical examples in MOA. MIT Press, Cambridge. https://mitpress.mit.edu/books/machine-learning-data-streams

Carmona J, Gavaldà R (2012) Online techniques for dealing with concept drift in process mining. In: Advances in intelligent data analysis XI – proceedings of the 11th international symposium, IDA 2012, Helsinki, 25-27 Oct 2012, pp 90–102. https://doi.org/10.1007/978-3-642-34156-4_10

Cohen L, Avrahami-Bakish G, Last M, Kandel A, Kipersztok O (2008) Real-time data mining of non-stationary data streams from sensor networks. Info Fusion 9(3):344–353. https://doi.org/10.1016/j.inffus.2005.05.005

Datar M, Gionis A, Indyk P, Motwani R (2002) Maintaining stream statistics over sliding windows. SIAM J Comput 31(6):1794–1813. https://doi.org/10.1137/S0097539701398363

Ditzler G, Roveri M, Alippi C, Polikar R (2015) Learning in nonstationary environments: a survey IEEE Comp Int Mag 10(4):12–25. https://doi.org/10.1109/MCI.2015.2471196

Gama J, Medas P, Castillo G, Rodrigues PP (2004) Learning with drift detection. In: Advances in artificial intelligence – SBIA 2004, proceedings of the 17th Brazilian symposium on artificial intelligence, São Luis, Sept 29–Oct 1 2004, pp 286–295. https://doi.org/10.1007/978-3-540-28645-5_29

Gama J, Sebastião R, Rodrigues PP (2009) Issues in evaluation of stream learning algorithms. In: Proceedings of the 15th ACM SIGKDD international conference on knowledge discovery and data mining (KDD 09), Paris, June 28–July 1 2009, pp 329–338. http://doi.acm.org/10.1145/1557019.1557060

Gama J, Žliobaitė I, Bifet A, Pechenizkiy M, Bouchachia A (2014) A survey on concept drift adaptation. ACM Comput Surv 46(4):44:1–44:37. http://doi.acm.org/10.1145/2523813

Hulten G, Spencer L, Domingos PM (2001) Mining time-changing data streams. In: Proceedings of the seventh ACM SIGKDD international conference on knowledge discovery and data mining (KDD 01), San Francisco, 26–29 Aug 2001, pp 97–106. http://portal.acm.org/citation.cfm?id=502512.502529

Kuncheva LI, Žliobaitė I (2009) On the window size for classification in changing environments. Intell Data Anal 13(6):861–872. https://doi.org/10.3233/IDA-2009-0397

Last M (2002) Online classification of nonstationary data streams. Intell Data Anal 6(2):129–147. http://content.iospress.com/articles/intelligent-data-analysis/ida00083

Muthukrishnan S, van den Berg E, Wu Y (2007) Sequential change detection on data streams. In: Workshops proceedings of the 7th IEEE international conference on data mining (ICDM 2007), 28–31 Oct 2007, Omaha, pp 551–550. https://doi.org/10.1109/ICDMW.2007.89

Papapetrou O, Garofalakis MN, Deligiannakis A (2015) Sketching distributed sliding-window data streams. VLDB J 24(3):345–368. https://doi.org/10.1007/s00778-015-0380-7

Pechenizkiy M, Bakker J, Žliobaitė I, Ivannikov A, Kärkkäinen T (2009) Online mass flow prediction in CFB boilers with explicit detection of sudden concept drift. SIGKDD Explor 11(2):109–116. http://doi.acm.org/10.1145/1809400.1809423

Ross GJ, Adams NM, Tasoulis DK, Hand DJ (2012) Exponentially weighted moving average charts for detecting concept drift. Pattern Recogn Lett 33(2): 191–198. https://doi.org/10.1016/j.patrec.2011.08.019, erratum in Pattern Recogn Lett 33(16):2261 (2012)

Talavera E, Dimiccoli M, Bolaños M, Aghaei M, Radeva P (2015) R-clustering for egocentric video segmentation. In: Pattern recognition and image analysis – 7th Iberian conference, proceedings of the IbPRIA 2015, Santiago de Compostela, 17-19 June 2015, pp 327–336. https://doi.org/10.1007/978-3-319-19390-8_37

A

Advancements in YARN Resource Manager

Konstantinos Karanasos, Arun Suresh, and Chris Douglas
Microsoft, Washington, DC, USA

Synonyms

Cluster scheduling; Job scheduling; Resource management

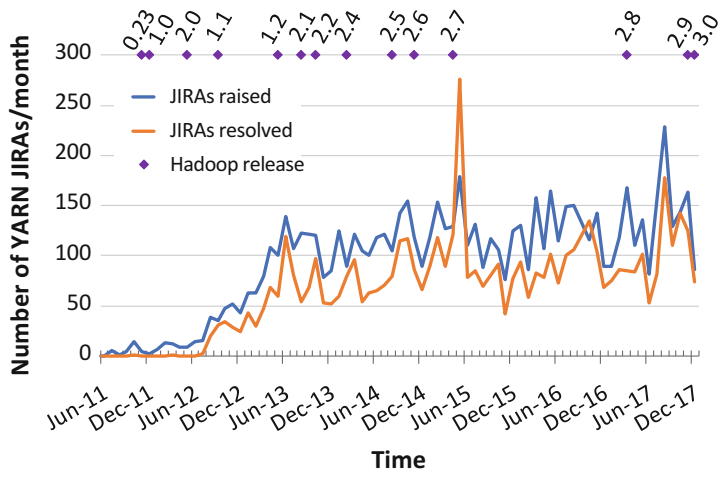
Definitions

YARN is currently one of the most popular frameworks for scheduling jobs and managing resources in shared clusters. In this entry, we focus on the new features introduced in YARN since its initial version.

Overview

Apache Hadoop (2017), one of the most widely adopted implementations of MapReduce (Dean and Ghemawat 2004), revolutionized the way that companies perform analytics over vast amounts of data. It enables parallel data processing over clusters comprised of thousands of machines while alleviating the user from implementing complex communication patterns and fault tolerance mechanisms.

With its rise in popularity, came the realization that Hadoop's resource model for MapReduce, albeit flexible, is not suitable for every application, especially those relying on low-latency



Advancements in YARN Resource Manager, Fig. 1 Timeline of Apache Hadoop releases (on top) and number of raised and resolved tickets (JIRAs) per month on YARN

or iterative computations. This motivated decoupling the cluster resource management infrastructure from specific programming models and led to the birth of YARN (Vavilapalli et al. 2013). YARN manages cluster resources and exposes a generic interface for applications to request resources. This allows several applications, including MapReduce, to be deployed on a single cluster and share the same resource management layer.

YARN is a community-driven effort that was first introduced in Apache Hadoop in November 2011, as part of the 0.23 release. Since then, the interest of the community has continued unabated. Figure 1 shows that more than 100 tickets, i.e., JIRAs (YARN JIRA 2017), related to YARN are raised every month. A steady portion of these JIRAs are resolved, which shows the continuous community engagement. In the past year alone, 160 individuals have contributed code to YARN.

Moreover, YARN has been widely deployed across hundreds of companies for production purposes, including Yahoo! (Oath), Microsoft, Twitter, LinkedIn, Hortonworks, Cloudera, eBay, and Alibaba.

Since YARN's inception, we observe the following trends in modern clusters:

Application variety Users' interest has expanded from batch analytics applications (e.g., MapReduce) to include streaming, iterative (e.g., machine learning), and interactive computations.

Large shared clusters Instead of using dedicated clusters for each application, diverse workloads are consolidated on clusters of thousands or even tens of thousands of machines. This consolidation avoids unnecessary data movement, allows for better resource utilization, and enables pipelines with different application classes.

High resource utilization Operating large clusters involves a significant cost of ownership. Hence, cluster operators rely on resource managers to achieve high cluster utilization and improve their return on investment.

Predictable execution Production jobs typically come with Service Level Objectives (SLOs), such as completion deadlines, which have to be met in order for the output of the jobs to be consumed by downstream services. Execution predictability is often more important than pure application performance when it comes to business-critical jobs.

This diverse set of requirements has introduced new challenges to the resource management layer. To address these new demands, YARN has evolved from a platform for batch analytics workloads to a production-ready, general-purpose resource manager that can support a wide range of applications and user requirements over large shared clusters. In the

remainder of this entry, we first give a brief overview of YARN's architecture and dedicate the rest of the paper to the new functionality that was added to YARN these last years.

YARN Architecture

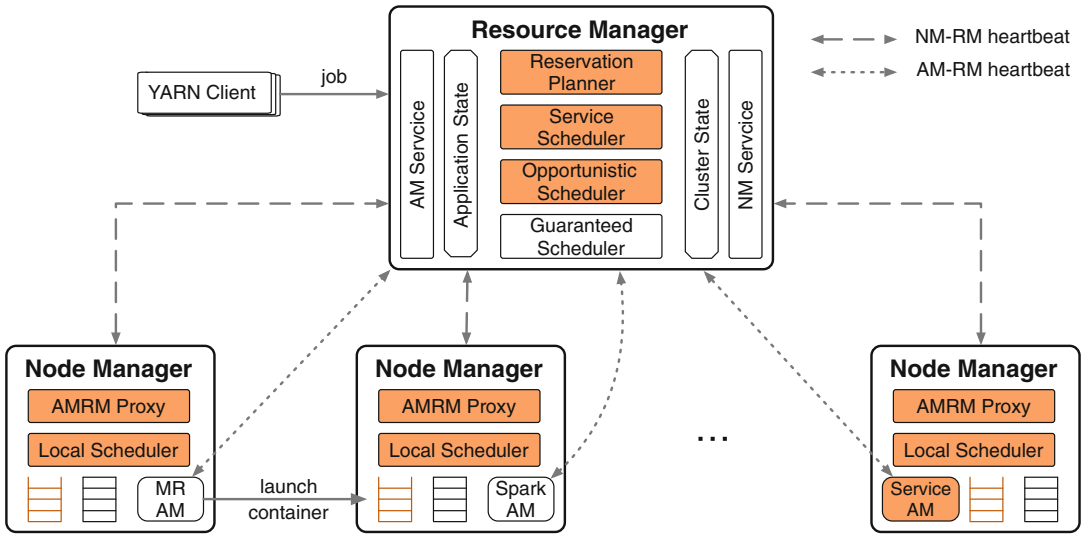
YARN follows a centralized architecture in which a single logical component, the resource manager (RM), allocates resources to jobs submitted to the cluster. The resource requests handled by the RM are intentionally generic, while specific scheduling logic required by each application is encapsulated in the application master (AM) that any framework can implement. This allows YARN to support a wide range of applications using the same RM component. YARN's architecture is depicted in Fig. 2. Below we describe its main components. The new features, which appear in orange, are discussed in the following sections.

Node Manager (NM) The NM is a daemon running at each of the cluster's worker nodes. NMs are responsible for monitoring resource availability at the host node, reporting faults, and managing containers' life cycle (e.g., start, monitor, pause, and kill containers).

Resource Manager (RM) The RM runs on a dedicated machine, arbitrating resources among various competing applications. Multiple RMs can be used for high availability, with one of them being the master. The NMs periodically inform the RM of their status, which is stored at the *cluster state*. The RM-NM communication is heartbeat-based for scalability. The RM also maintains the resource requests of all applications (*application state*). Given its global view of the cluster and based on application demand, resource availability, scheduling priorities, and sharing policies (e.g., fairness), the *scheduler* performs the matchmaking between application requests and machines and hands leases, called *containers*, to applications. A container is a logical resource bundle (e.g., 2 GB RAM, 1 CPU) bound to a specific node.

YARN includes two scheduler implementations, namely, the Fair and Capacity Schedulers. The former imposes fairness between applications, while the latter dedicates a share of the cluster resources to groups of users.

Jobs are submitted to the RM via the *YARN Client* protocol and go through an



Advancements in YARN Resource Manager, Fig. 2 YARN architecture and overview of new features (in orange)

admission control phase, during which security credentials are validated and various operational and administrative checks are performed.

Application Master (AM) The AM is the job orchestrator (one AM is instantiated per submitted job), managing all its life cycle aspects, including dynamically increasing and decreasing resource consumption, managing the execution flow (e.g., running reducers against the output of mappers), and handling faults. The AM can run arbitrary user code, written in any programming language. By delegating all these functions to AMs, YARN's architecture achieves significant scalability, programming model flexibility, and improved upgrading/testing.

An AM will typically need to harness resources from multiple nodes to complete a job. To obtain containers, the AM issues resource requests to the RM via heartbeats, using the *AM Service* interface. When the scheduler assigns a resource to the AM, the RM generates a lease for that resource. The AM is then notified and presents the container lease to the NM for launching the container at that node. The NM checks the authenticity of the lease and then initiates the container execution.

In the following sections, we present the main advancements made in YARN, in particular with respect to resource utilization, scalability, support for services, and execution predictability.

Resource Utilization

In the initial versions of YARN, the RM would assign containers to a node only if there were unallocated resources on that node. This **guaranteed** type of allocation ensures that once an AM dispatches a container to a node, there will be sufficient resources for its execution to start immediately.

Despite the predictable access to resources that this design offers, it has the following shortcomings that can lead to suboptimal resource utilization:

Feedback delays The heartbeat-based AM-RM and NM-RM communications can cause idle node resources from the moment a container finishes its execution on a node to the moment an AM gets notified through the RM to launch a new container on that node.

Underutilized resources The RM assigns containers based on the allocated resources at each node, which might be significantly higher than the actually utilized ones (e.g., a 4GB container using only 2 GB of its memory).

In a typical YARN cluster, NM-RM heartbeat intervals are set to 3 s, while AM-RM intervals vary but are typically up to a few seconds. Therefore, feedback delays are more pronounced for workloads with short tasks.

Below we describe the new mechanisms that were introduced in YARN to improve cluster resource utilization. These ideas first appeared in the Mercury and Yaq systems (Karanasos et al. 2015; Rasley et al. 2016) and are part of Apache Hadoop as of version 2.9 (Opportunistic scheduling 2017; Distributed scheduling 2017).

Opportunistic containers Unlike guaranteed containers, opportunistic ones are dispatched to an NM, even if there are no available resources on that node. In such a case, the opportunistic containers will be placed in a newly introduced **NM queue** (see Fig. 2). When resources become available, an opportunistic container will be picked from the queue, and its execution will start immediately, avoiding any feedback delays.

These containers run with lower priority in YARN and will be preempted in case of resource contention for guaranteed containers to start their execution. Hence, opportunistic containers improve cluster resource utilization without impacting the execution of guaranteed containers. Moreover, whereas the original NM passively executes conflict-free commands from the RM, a modern NM uses these two-level priorities as inputs to local scheduling decisions. For instance, low-priority jobs with non-strict execution guarantees or tasks off the critical path of a DAG, are good candidates for opportunistic containers.

The AMs currently determine the execution type for each container, but the system could use automated policies instead. The AM can also request promotion of opportunistic containers to guarantee to protect them from preemption.

Hybrid scheduling Opportunistic containers can be allocated centrally by the RM or in a distributed fashion through a local scheduler that runs at each NM and leases containers on other NMs without contacting the RM. Centralized allocation allows for higher-quality placement decisions and sharing policies. Distributed allocation offers lower allocation latencies, which can be beneficial for short-lived containers. To prevent conflicts, guaranteed containers are always assigned by the RM.

To determine the least-loaded nodes for placing opportunistic containers, the RM periodically gathers information about the running and queued containers at each node and propagates this information to the local schedulers too. To account for occasional load imbalance across nodes, YARN performs dynamic rebalancing of queued containers.

Resource overcommitment Currently, opportunistic containers can be employed to avoid feedback delays. Ongoing development also focuses on overcommitting resources using opportunistic containers (Utilization-based scheduling 2017). In this scenario, opportunistic containers facilitate reclaiming overcommitted resources on demand, without affecting the performance and predictability of jobs that opt out of overcommitted resources.

Cluster Scalability

A single YARN RM can manage a few thousands of nodes. However, production analytics clusters at big cloud companies are often comprised of tens of thousands of machines, crossing YARN's limits (Burd et al. 2017).

YARN's scalability is constrained by the resource manager, as load increases proportionally to the number of cluster nodes and the application demands (e.g., active containers, resource requests per second). Increasing the heartbeat intervals could improve scalability in terms of number of nodes, but would be detrimental to utilization (Vavilapalli et al. 2013) and would still pose problems as the number of applications increases.

Instead, as of Apache Hadoop 2.9 (YARN Federation 2017), a **federation-based** approach scales a single YARN cluster to tens of thousands of nodes. This approach divides the cluster into smaller units, called **subclusters**, each with its own YARN RM and NMs. The federation system negotiates with subcluster RMs to give applications the experience of a single large cluster, allowing applications to schedule their tasks to any node of the federated cluster.

The state of the federated cluster is coordinated through the **State Store**, a central component that holds information about (1) subcluster liveliness and resource availability via heartbeats sent by each subcluster RM, (2) the YARN subcluster at which each AM is being deployed, and (3) policies used to impose global cluster invariants and perform load rebalancing.

To allow jobs to seamlessly span subclusters, the federated cluster relies on the following components:

Router A federated YARN cluster is equipped with a set of routers, which hide the presence of multiple RMs from applications. Each application gets submitted to a router, which, based on a policy, determines the subcluster for the AM to be executed, gets the subcluster URL from the State Store, and redirects the application submission request to the appropriate subcluster RM.

AMRM Proxy This component runs as a service at each NM of the cluster and acts as a proxy for every AM-RM communication. Instead of directly contacting the RM, applications are forced by the system to access their local AMRM Proxy. By dynamically routing the AM-RM messages, the AMRM Proxy provides the applications with transparent access to multiple YARN RMs. Note that the AMRM Proxy is also used to implement the

local scheduler for opportunistic containers and could be used to protect the system against misbehaving AMs.

This federated design is scalable, as the number of nodes each RM is responsible for is bounded. Moreover, through appropriate policies, the majority of applications will be executed within a single subcluster; thus the number of applications that are present at each RM is also bounded. As the coordination between subclusters is minimal, the cluster's size can be scaled almost linearly by adding more subclusters. This architecture can provide tight enforcement of scheduling invariants within a subcluster, while continuous rebalancing across subclusters enforces invariants in the whole cluster.

A similar federated design has been followed to scale the underlying store (HDFS Federation 2017).

Long-Running Services

As already discussed, YARN's target applications were originally batch analytics jobs, such as MapReduce. However, a significant share of today's clusters is dedicated to workloads that include stream processing, iterative computations, data-intensive interactive jobs, and latency-sensitive online applications. Unlike batch jobs, these applications benefit from long-lived containers (from hours to months) to amortize container initialization costs, reduce scheduling load, or maintain state across computations. Here we use the term *services* for all such applications.

Given their long-running nature, these applications have additional demands, such as support for restart, in-place upgrade, monitoring, and discovery of their components. To avoid using YARN's low-level API for enabling such operations, users have so far resorted to AM libraries such as Slider (Apache Slider 2017). Unfortunately, these external libraries only partially solve the problem, e.g., due to lack of common standards for YARN to optimize resource demands across libraries or version incompatibilities between the libraries and YARN.

To this end, the upcoming Apache Hadoop 3.1 release adds first-class support for long-running services in YARN, allowing for both traditional process-based and Docker-based containers. This service framework allows users to deploy existing services on YARN, simply by providing a JSON file with their service specifications, without having to translate those requirements into low-level resource requests at runtime.

The main component of YARN's service framework is the **container orchestrator**, which facilitates service deployment. It is an AM that, based on the service specification, configures the required requests for the RM and launches the corresponding containers. It deals with various service operations, such as starting components given specified dependencies, monitoring their health and restarting failed ones, scaling up and down component resources, upgrading components, and aggregating logs.

A **RESTful API server** is developed to allow users to manage the life cycle of services on YARN via simple commands, using framework-independent APIs. Moreover, a **DNS server** enables service discovery via standard DNS lookups and greatly simplifies service failovers.

Scheduling services Apart from the aforementioned support for service deployment and management, service owners also demand precise control of container placement to optimize the performance and resilience of their applications. For instance, containers of services are often required to be collocated (affinity) to reduce network costs or separated (anti-affinity) to minimize resource interference and correlated failures. For optimal service performance, even more powerful constraints are useful, such as complex intra- and inter-application constraints that collocate services with one another or put limits in the number of specific containers per node or rack.

When placing containers of services, cluster operators have their own, potentially conflicting, global optimization objectives. Examples include minimizing the violation of placement constraints, the resource fragmentation, the load

imbalance, or the number of machines used. Due to their long lifetimes, services can tolerate longer scheduling latencies than batch jobs, but their placement should not impact the scheduling latencies of the latter.

To enable high-quality placement of services in YARN, Apache Hadoop 3.1 introduces support for rich placement constraints (Placement constraints 2017).

Jobs with SLOs

In production analytics clusters, the majority of cluster resources is usually consumed by **production jobs**. These jobs must meet strict Service Level Objectives (SLOs), such as completion deadlines, for their results to be consumed by downstream services. At the same time, a large number of smaller **best-effort jobs** are submitted to the same clusters in an ad hoc manner for exploratory purposes. These jobs lack SLOs, but they are sensitive to completion latencies.

Resource managers typically allocate resources to jobs based on *instantaneous* enforcement of job priorities and sharing invariants. Although simpler to implement and impose, this instantaneous resource provisioning makes it challenging to meet job SLOs without sacrificing low latency for best-effort jobs.

To ensure that important production jobs will have predictable access to resources, YARN was extended with the notion of *reservations*, which provide users with the ability to reserve resources over (and ahead of) time. The ideas around reservations first appeared in Rayon (Curino et al. 2014) and are part of YARN as of Apache Hadoop 2.6.

Reservations This is a construct that determines the resource needs and temporal requirements of a job and translates the job's completion deadline into an SLO over predictable resource allocations. This is done ahead of the job's execution, aimed at ensuring a predictable and timely execution. To this end, YARN introduced a reservation definition language (RDL) to express a rich class of time-aware resource requirements, including deadlines, malleable and gang parallelism, and inter-job dependencies.

Reservation planning and scheduling RDL provides a uniform and abstract representation of jobs' needs. Reservation requests are received ahead of a job's submission by the **reservation planner**, which performs online admission control. It accepts all jobs that can fit in the cluster agenda over time and rejects those that cannot be satisfied. Once a reservation is accepted by the planner, the scheduler is used to dynamically assign cluster resources to the corresponding job.

Periodic reservations Given that a high percentage of production jobs are recurring (e.g., hourly, daily, or monthly), YARN allows users to define periodic reservations, starting with Apache Hadoop 2.9. A key property of recurring reservations is that once a periodic job is admitted, each of its instantiations will have a predictable resource allocation. This isolates periodic production jobs from the noisiness of sharing.

Toward predictable execution The idea of recurring reservations was first exposed as part of the Morpheus system (Jyothi et al. 2016). Morpheus analyzes inter-job data dependencies and ingress/egress operations to automatically derive SLOs. It uses a resource estimator tool, which is also part of Apache Hadoop as of version 2.9, to estimate jobs' resource requirements based on historic runs. Based on the derived SLOs and resource demands, the system generates recurring reservations and submits them for planning. This guarantees that periodic production jobs will have guaranteed access to resources and thus predictable execution.

Further Improvements

In this section, we discuss some additional improvements made to YARN.

Generic resources As more heterogeneous applications with varying resource demands are deployed to YARN clusters, there is an increasing

need for finer control of resource types other than memory and CPU. Examples include disk bandwidth, network I/O, GPUs, and FPGAs.

Adding new resource types in YARN used to be cumbersome, as it required extensive code changes. The upcoming Apache Hadoop 3.1 release (Resource profiles 2017) follows a more flexible resource model, allowing users to add new resources with minimal effort. In fact, users can define their resources in a configuration file, eliminating the need for code changes or recompilation. The Dominant Resource Fairness (Ghodsi et al. 2011) scheduling algorithm at the RM has also been adapted to account for generic resource types, while *resource profiles* can be used for AMs to request containers specifying predefined resource sets. Ongoing work focuses on the isolation of resources such as disk, network, and GPUs.

Node labels Cluster operators can group nodes with similar characteristics, e.g., nodes with public IPs or nodes used for development or testing. Applications can then request containers on nodes with specific labels. This feature is supported by YARN's Capacity Scheduler from Apache Hadoop 2.6 on (Node labels 2017) and allows at most one label to be specified per node, thus creating nonoverlapping node partitions in the cluster. The cluster administrator can specify the portion of a partition that a queue of the scheduler can access, as well as the portion of a queue's capacity that is dedicated to a specific node partition. For instance, queue A might be restricted to access no more than 30% of the nodes with public IPs, and 40% of queue A has to be on dev machines.

Changing queue configuration Several companies use YARN's Capacity Scheduler to share clusters across non-coordinating user groups. A hierarchy of queues isolates jobs from each department of the organization. Due to changes in the resource demands of each department, the queue hierarchy or the cluster condition, operators modify the amount of resources assigned to each organization's queue and to the sub-queues used within that department. However, queue reconfiguration has two main drawbacks: (1) setting and changing configurations is a tedious process that can only be performed by modifying XML files; (2) queue owners cannot perform any modifications to their sub-queues; the cluster admin must do it on their behalf.

To address these shortcomings, Apache Hadoop 2.9 (OrgQueue 2017) allows configurations to be stored in an in-memory database instead of XML files. It adds a RESTful API to programmatically modify the queues. This has the additional benefit that queues can be dynamically reconfigured by automated services, based on the cluster conditions or on organization-specific criteria. Queue ACLs allow queue owners to perform modifications on their part of the queue structure.

Timeline server Information about current and previous jobs submitted in the cluster is key for debugging, capacity planning, and performance tuning. Most importantly, observing historic data enables us to better understand the cluster and jobs' behavior in aggregate to holistically improve the system's operation.

The first incarnation of this effort was the application history server (AHS), which supported only MapReduce jobs. The AHS was superseded by the timeline server (TS), which can deal with generic YARN applications. In its first version, the TS was limited to a single writer and reader that resided at the RM. Its applicability was therefore limited to small clusters.

Apache Hadoop 2.9 includes a major redesign of TS (YARN TS v2 2017), which separates the collection (writes) from the serving (reads) of data, and performs both operations in a distributed manner. This brings several scalability and flexibility improvements.

The new TS collects metrics at various granularities, ranging from *flows* (i.e., sets of YARN applications logically grouped together) to jobs, job attempts, and containers. It also collects cluster-wide data, such as user and queue information, as well as configuration data.

The data collection is performed by collectors that run as services at the RM and at every NM.

The AM of each job publishes data to the collector of the host NM. Similarly, each container pushes data to its local NM collector, while the RM publishes data to its dedicated collector. The readers are separate instances that are dedicated to serving queries via a REST API. By default Apache HBase (Apache HBase 2017) is used as the backing storage, which is known to scale to large amounts of data and read/write operations.

Conclusion

YARN was introduced in Apache Hadoop at the end of 2011 as an effort to break the strong ties between Hadoop and MapReduce and to allow generic applications to be deployed over a common resource management fabric. Since then, YARN has evolved to a fully fledged production-ready resource manager, which has been deployed on shared clusters comprising tens of thousands of machines. It handles applications ranging from batch analytics to streaming and machine learning workloads to low-latency services while achieving high resource utilization and supporting SLOs and predictability for production workloads. YARN enjoys a vital community with hundreds of monthly contributions.

Cross-References

▸ Hadoop

Acknowledgements The authors would like to thank Subru Krishnan and Carlo Curino for their feedback while preparing this entry. We would also like to thank the diverse community of developers, operators, and users that have contributed to Apache Hadoop YARN since its inception.

References

Apache Hadoop (2017) Apache Hadoop. http://hadoop.apache.org

Apache HBase (2017) Apache HBase. http://hbase.apache.org

Apache Slider (2017) Apache Slider (incubating). http://slider.incubator.apache.org

Burd R, Sharma H, Sakalanaga S (2017) Lessons learned from scaling YARN to 40 K machines in a multi-tenancy environment. In: DataWorks Summit, San Jose

Curino C, Difallah DE, Douglas C, Krishnan S, Ramakrishnan R, Rao S (2014) Reservation-based scheduling: if you're late don't blame us! In: ACM symposium on cloud computing (SoCC)

Dean J, Ghemawat S (2004) MapReduce: simplified data processing on large clusters. In: USENIX symposium on operating systems design and implementation (OSDI)

Distributed scheduling (2017) Extend YARN to support distributed scheduling. https://issues.apache.org/jira/browse/YARN-2877

Ghodsi A, Zaharia M, Hindman B, Konwinski A, Shenker S, Stoica I (2011) Dominant resource fairness: fair allocation of multiple resource types. In: USENIX symposium on networked systems design and implementation (NSDI)

HDFS Federation (2017) Router-based HDFS federation. https://issues.apache.org/jira/browse/HDFS-10467

Jyothi SA, Curino C, Menache I, Narayanamurthy SM, Tumanov A, Yaniv J, Mavlyutov R, Goiri I, Krishnan S, Kulkarni J, Rao S (2016) Morpheus: towards automated slos for enterprise clusters. In: USENIX symposium on operating systems design and implementation (OSDI)

Karanasos K, Rao S, Curino C, Douglas C, Chaliparambil K, Fumarola GM, Heddaya S, Ramakrishnan R, Sakalanaga S (2015) Mercury: hybrid centralized and distributed scheduling in large shared clusters. In: USENIX annual technical conference (USENIX ATC)

Node Labels (2017) Allow for (admin) labels on nodes and resource-requests. https://issues.apache.org/jira/browse/YARN-796

Opportunistic Scheduling (2017) Scheduling of opportunistic containers through YARN RM. https://issues.apache.org/jira/browse/YARN-5220

OrgQueue (2017) OrgQueue for easy capacityscheduler queue configuration management. https://issues.apache.org/jira/browse/YARN-5734

Placement Constraints (2017) Rich placement constraints in YARN. https://issues.apache.org/jira/browse/YARN-6592

Rasley J, Karanasos K, Kandula S, Fonseca R, Vojnovic M, Rao S (2016) Efficient queue management for cluster scheduling. In: European conference on computer systems (EuroSys)

Resource Profiles (2017) Extend the YARN resource model for easier resource-type management and profiles. https://issues.apache.org/jira/browse/YARN-3926

Utilization-Based Scheduling (2017) Schedule containers based on utilization of currently allocated containers. https://issues.apache.org/jira/browse/YARN-1011

Vavilapalli VK, Murthy AC, Douglas C, Agarwal S, Konar M, Evans R, Graves T, Lowe J, Shah H, Seth S, Saha B, Curino C, O'Malley O, Radia S, Reed B,

Baldeschwieler E (2013) Apache Hadoop YARN: yet another resource negotiator. In: ACM symposium on cloud computing (SoCC)
YARN Federation (2017) Enable YARN RM scale out via federation using multiple RMs. https://issues.apache.org/jira/browse/YARN-2915
YARN JIRA (2017) Apache JIRA issue tracker for YARN. https://issues.apache.org/jira/browse/YARN
YARN TS v2 (2017) YARN timeline service v.2. https://issues.apache.org/jira/browse/YARN-5355

ADWIN Algorithm

▸ Adaptive Windowing

Alignment Creation

▸ Business Process Model Matching

Analytics Benchmarks

Todor Ivanov and Roberto V. Zicari
Frankfurt Big Data Lab, Goethe University Frankfurt, Frankfurt, Germany

Synonyms

Big data benchmarks; Decision support benchmarks; Machine learning benchmarks; OLAP benchmarks; OLTP benchmarks; Real-time streaming benchmarks

Definitions

The meaning of the word benchmark is (Andersen and Pettersen 1995) *A predefined position, used as a reference point for taking measures against.* There is no clear formal definition of analytics benchmarks.

Jim Gray (1992) describes the benchmarking as follows: "This quantitative comparison starts with the definition of a benchmark or workload. The benchmark is run on several different systems, and the performance and price of each system is measured and recorded. Performance is typically a throughput metric (work/second) and price is typically a five-year cost-of-ownership metric. Together, they give a price/performance ratio." In short, we define that a software benchmark is a program used for comparison of software products/tools executing on a pre-configured hardware environment.

Analytics benchmarks are a type of domain-specific benchmark targeting analytics for databases, transaction processing, and big data systems. Originally, the TPC (Transaction Processing Performance Council) (TPC 2018) defined online transaction processing (OLTP) (TPC-A and TPC-B) and decision support (DS) benchmarks (TPC-D and TPC-H). The DS systems can be seen as some sort of special online analytical processing (OLAP) system with an example of the TPC-DS benchmark, which is a successor of TPC-H (Nambiar and Poess 2006) and specifies many OLAP and data mining queries, which are the predecessors of the current analytics benchmarks. However, due to the many new emerging data platforms like hybrid transaction/analytical processing (HTAP) (Kemper and Neumann 2011; Özcan et al. 2017), distributed parallel processing engines (Sakr et al. 2013; Hadoop 2018; Spark 2018; Flink 2018; Carbone et al. 2015, etc.), Big data management (AsterixDB 2018; Alsubaiee et al. 2014), SQL-on-Hadoop-alike (Abadi et al. 2015; Hive 2018; Thusoo et al. 2009; SparkSQL 2018; Armbrust et al. 2015; Impala 2018; Kornacker et al. 2015, etc.), and analytics systems (Hu et al. 2014) integrating machine learning (MLlib 2018; Meng et al. 2016; MADlib 2018; Hellerstein et al. 2012), Deep Learning (Tensorflow 2018) and more, the emerging benchmarks try to follow the trend to stress these new system features. This makes the currently standardized benchmarks (such as TPC-C, TPC-H, etc.) only partially relevant for the emerging big data management systems as they offer new features that require new analytics benchmarks.

Overview

This chapter reviews the evolution of the analytics benchmarks and their current state today (as of 2017). It starts overview of the most relevant benchmarking organizations their benchmark standards and outlines the latest benchmark development and initiatives targeting the emerging Big Data Analytics systems. Last but not least the typical benchmark components are described as well as the different goals that these benchmarks try to achieve.

Historical Background

OLTP and DSS/OLAP

In the end of the 1970s, many businesses started implementing transaction-based systems (Rockart et al. 1982), which later became known as the term online transaction processing (OLTP) systems and represent the instant interaction between the user and the data management system. This type of transaction processing systems became a key part of the companies' operational infrastructure and motivated TPC (TPC 2018) to target these systems in their first formal benchmark specification. At the same time, the decision support systems (DSS) evolved significantly and became a standard tool for the enterprises that assisted in the human decision-making (Shim et al. 2002).

In the early 1990s, a different type of system, called online analytical processing (OLAP) systems by Codd et al. (1993), was used by the enterprises to dynamically manipulate and synthesize historic information. The historic data was aggregated from the OLTP systems, and through the application of dynamic analysis, the users were able to gain important knowledge for the operational activities over longer periods of time.

Over the years, the DS systems were enhanced by the use of the OLAP systems (Shim et al. 2002). They became an essential decision-making tool for the enterprise management and a core element of the company infrastructure. With the wide adaption of multipurpose database systems to build both an OLTP and DS system, the need for standardized database benchmarks arose. This resulted in an intense competition between database vendors to dominate the market, which leads to the need of domain-specific benchmarks to stress the database software. The use of sample workloads together with a bunch of metrics was not enough to guarantee the product capabilities in a transparent way. Another arising issue was the use of benchmarks for benchmarketing. It happens when a company uses a particular benchmark to highlight the strengths of its product and hide its weaknesses and then promotes the benchmark as a "standard," often without disclosing the details of the benchmark (Gray 1992). All of these opened the gap for standardized benchmarks that are formally specified by recognized expert organizations. Therefore, a growing number of organizations are working on defining and standardizing of benchmarks. They operate as consortia of public and private organizations and define domain-specific benchmarks, price, and performance metrics, measuring and reporting rules as well as formal validation and auditing rules.

TPC

The TPC (Transaction Processing Performance Council) (TPC 2018) is a nonprofit corporation operating as an industry consortium of vendors that define transaction processing, database, and big data system benchmarks. TPC was formed on August 10, 1988, by eight companies convinced by Omri Serlin (TPC 2018). In November 1989 was published the first standard benchmark TPC-A with 42-page specification (Gray 1992). By late 1990, there were 35 member companies. As of 2017, TPC has 21 company members and 3 associate members. There are 6 obsolete benchmarks (TPC-A, TPC-App, TPC-B, TPC-D, TPC-R, and TPC-W), 14 active benchmarks (TPC-C (Raab 1993), TPC-E (Hogan 2009), TPC-H (Pöss and Floyd 2000), TPC-DS (Poess et al. 2017; Pöss et al. 2007; Nambiar and Poess 2006), TPC-DI (Poess et al. 2014), TPC-V (Sethuraman and Taheri 2010), TPCx-HS (Nambiar 2014),

Analytics Benchmarks, Table 1 Active TPC benchmarks (TPC 2018)

Benchmark domain	Specification name
Transaction processing (OLTP)	TPC-C, TPC-E
Decision support (OLAP)	TPC-H, TPC-DS, TPC-DI
Virtualization	TPC-VMS, TPCx-V, TPCx-HCI
Big data	TPCx-HS V1, TPCx-HS V2, TPCx-BB, TPC-DS V2
IoT	TPCx-IoT
Common specifications	TPC-pricing, TPC-energy

Analytics Benchmarks, Table 2 Active SPEC benchmarks (SPEC 2018)

Benchmark domain	Specification name
Cloud	SPEC cloud IaaS 2016
CPU	SPEC CPU2006, SPEC CPU2017
Graphics and workstation performance	SPECapc for solidWorks 2015, SPECapc for siemens NX 9.0 and 10.0, SPECapc for PTC Creo 3.0, SPECapc for 3ds Max 2015, SPECwpc V2.1, SPECviewperf 12.1
High-performance computing, OpenMP, MPI, OpenACC, OpenCL	SPEC OMP2012, SPEC MPI2007, SPEC ACCEL
Java client/server	SPECjvm2008, SPECjms2007, SPECjEnterprise2010, SPECjbb2015
Storage	SPEC SFS2014
Power	SPECpower ssj2008
Virtualization	SPEC VIRT SC 2013

TPCx-BB (Ghazal et al. 2013)), and 2 common specifications (pricing and energy) used across all benchmarks. Table 1 lists the active TPC benchmarks grouped by domain.

SPEC

The SPEC (Standard Performance Evaluation Corporation) (SPEC 2018) is a nonprofit corporation formed to establish, maintain, and endorse standardized benchmarks and tools to evaluate performance and energy efficiency for the newest generation of computing systems. It was founded in 1988 by a small number of workstation vendors. The SPEC organization is an umbrella organization that covers four groups (each with their own benchmark suites, rules, and dues structure): the Open Systems Group (OSG), the High-Performance Group (HPG), the Graphics and Workstation Performance Group (GWPG), and the SPEC Research Group (RG). As of 2017, there are around 19 active SPEC benchmarks listed in Table 2.

STAC

The STAC Benchmark Council (STAC 2018) consists of over 300 financial institutions and more than 50 vendor organizations whose purpose is to explore technical challenges and solutions in financial services and to develop technology benchmark standards that are useful to financial organizations. Since 2007, the council is working on benchmarks targeting fast data, big data, and big compute workloads in the finance industry. As of 2017, there are around 11 active benchmarks listed in Table 3.

Other historical benchmark organizations and consortia are The Perfect Club (Gray 1992; Hockney 1996) and the Parkbench Committee (Hockney 1996).

Big Data Technologies

In the recent years, many emerging data technologies have become popular, trying to solve the challenges posed by the new big data and Internet of things application scenarios. In a historical

Analytics Benchmarks, Table 3 Active STAC benchmarks (STAC 2018)

Benchmark domain	Specification name
Feed handlers	STAC-M1
Data distribution	STAC-M2
Tick analytics	STAC-M3
Event processing	STAC-A1
Risk computation	STAC-A2
Backtesting	STAC-A3
Trade execution	STAC-E
Tick-to-trade	STAC-T1
Time sync	STAC-TS
Big data	In-development
Network I/O	STAC-N1, STAC-T0

overview of the trends in data management technologies, Nambiar et al. (2012) highlight the role of big data technologies and how they are currently changing the industry. One such technology is the NoSQL storage engines (Cattell 2011) which relax the ACID (atomicity, consistency, isolation, durability) guarantees but offer faster data access via distributed and fault-tolerant architecture. There are different types of NoSQL engines (key value, column, graph, and documents stores) covering different data representations.

In the meantime, many new dig data technologies such as (1) Apache Hadoop (2018) with HDFS and MapReduce; (2) general parallel processing engines like Spark (2018) and Flink (2018); (3) SQL-on-Hadoop systems like Hive (2018) and Spark SQL (2018); (4) real-time stream processing engines like Storm (2018), Spark Streaming (2018) and Flink; and (5) graph engines on top of Hadoop like GraphX (2018) and Flink Gelly (2015) have emerged. All these tools enabled advanced analytical techniques from data science, machine learning, data mining, and deep learning to become common practices in many big data domains. Because of all these analytical techniques, which are currently integrated in many different ways in both traditional database and new big data management systems, it is hard to define the exact features that a successor of the DS/OLAP systems should have.

Big Data Analytics Benchmarks

Following the big data technology trends, many new benchmarks for big data analytics have emerged. Good examples for OLTP benchmarks targeting the NoSQL engines are the Yahoo! Cloud Serving Benchmark (short YCSB), developed by Yahoo, and LinkBench developed by Facebook, described in Table 4. However, most big data benchmarks stress the capabilities of Hadoop as the major big data platform, as listed in Table 5. Others like BigFUN (Pirzadeh et al. 2015) and BigBench (Ghazal et al. 2013) are technology-independent. For example, BigBench (standardized as TPCx-BB) addresses the Big Data 3V's characteristics and relies on workloads which can be implemented by different SQL-on-Hadoop systems and parallel processing engines supporting advanced analytics and machine learning libraries. Since there are no clear boundaries for the analytical capabilities of the new big data systems, it is also hard to formally specify what is an analytics benchmark.

A different type of benchmark, called benchmark suites, has become very popular. Their goal is to package a number of micro-benchmarks or representative domain workloads together and in this way enable the users to easily test the systems for the different functionalities. Some of these suites target one technology like SparkBench (Li et al. 2015; Agrawal et al. 2015), which stresses only Spark, while others like HiBench offer implementations for multiple processing engines. Table 6 lists some popular big data benchmarking suites.

A more detailed overview of the current big data benchmarks is provided in a SPEC Big Data Research Group survey by Ivanov et al. (2015) and a journal publication by Han et al. (2018).

Foundations

In the last 40 years, the OLTP and DS/OLAP systems have been the industry standard systems for data storage and management. Therefore, all popular TPC benchmarks were specified in these areas. The majority TPC benchmark specifications (Poess 2012) have the following main components:

Analytics Benchmarks, Table 4 OLTP benchmarks

Name	Benchmark description
YCSB (Cooper et al. 2010; Patil et al. 2011)	A benchmark designed to compare emerging cloud serving systems like Cassandra, HBase, MongoDB, Riak, and many more, which do not support ACID. It provides a core package of six predefined workloads A–F, which simulate a cloud OLTP application
LinkBench (Armstrong et al. 2013)	A benchmark, developed by Facebook, using synthetic social graph to emulate social graph workload on top of databases such as MySQL and MongoDB

Analytics Benchmarks, Table 5 DS/OLAP/Analytics benchmarks

Name	Benchmark description
MRBench (Kim et al. 2008)	Implementing the TPC-H benchmark queries directly in map and reduce operations
CALDA (Pavlo et al. 2009)	It consists of five tasks defined as SQL queries among which is the original MR Grep task, which is a representative for most real user MapReduce programs
AMP lab big data benchmark (AMPLab 2013)	A benchmark based on CALDA and HiBench, implemented on five SQL-on-Hadoop engines (RedShift, Hive, Stinger/Tez, Shark, and Impala)
BigBench (Ghazal et al. 2013)	An end-to-end big data benchmark that represents a data model simulating the volume, velocity, and variety characteristics of a big data system, together with a synthetic data generator for structured, semi-structured, and unstructured data, consisting of 30 queries
BigFrame (BigFrame 2013)	BigFrame is a benchmark generator offering a benchmarking-as-a-service solution for big data analytics
PRIMEBALL (Ferrarons et al. 2013)	A novel and unified benchmark specification for comparing the parallel processing frameworks in the context of big data applications hosted in the cloud. It is implementation- and technology-agnostic, using a fictional news hub called New Pork Times, based on a popular real-life news site
BigFUN (Pirzadeh et al. 2015)	It is based on a social network use case with synthetic semi- structured data in JSON format. The benchmark focuses exclusively on micro-operation level and consists of queries with various operations such as simple retrieves, range scans, aggregations, and joins, as well as inserts and updates
BigBench V2 (Ghazal et al. 2017)	BigBench V2 separates from TPC-DS with a simple data model, consisting only of six tables. The new data model still has the variety of structured, semi-structured, and unstructured data as the original BigBench data model. The semi-structured data (weblogs) are generated in JSON logs. New queries replace all the TPC-DS queries and preserve the initial number of 30 queries

- Preamble – Defines the benchmark domain and the high level requirements.
- Database Design – Defines the requirements and restrictions for implementing the database schema.
- Workload – Characterizes the simulated workload.
- ACID – Atomicity, consistency, isolation, and durability requirements.
- Workload scaling – Defines tools and methodology on how to scale the workloads.
- Metric/Execution rules – Defines how to execute the benchmark and how to calculate and derive the metrics.

Analytics Benchmarks, Table 6 Big data benchmark suites

Name	Benchmark description
MRBS (Sangroya et al. 2012)	A comprehensive benchmark suite for evaluating the performance of MapReduce systems in five areas: recommendations, BI (TPC-H), bioinformatics, text processing, and data mining
HiBench (Huang et al. 2010)	A comprehensive benchmark suite consisting of multiple workloads including both synthetic micro-benchmarks and real-world applications. It features several ready-to-use benchmarks from 4 categories: micro benchmarks, Web search, machine learning, and HDFS benchmarks
CloudSuite (Ferdman et al. 2012)	A benchmark suite consisting of both emerging scale-out workloads and traditional benchmarks. The goal of the benchmark suite is to analyze and identify key inefficiencies in the processors core micro-architecture and memory system organization when running todays cloud workloads
CloudRank-D (Luo et al. 2012)	A benchmark suite for evaluating the performance of cloud computing systems running big data applications. The suite consists of 13 representative data analysis tools, which are designed to address a diverse set of workload data and computation characteristics (i.e., data semantics, data models and data sizes, the ratio of the size of data input to that of data output)
BigDataBench (Wang et al. 2014)	An open-source big data benchmark suite consisting of 15 data sets (of different types) and more than 33 workloads. It is a large effort organized in China available with a toolkit that adopts different other benchmarks
SparkBench (Li et al. 2015; Agrawal et al. 2015)	SparkBench, developed by IBM, is a comprehensive Spark-specific benchmark suite that comprises of four main workload categories: machine learning, graph processing, streaming, and SQL queries.

- Benchmark driver – Defines the requirements for implementing the benchmark driver/program.
- Full disclosure report – Defines what needs to be reported and how to organize the disclosure report.
- Audit requirements – Defines the requirements for performing a successful auditing process.

The above structure was typical for the OLTP and DS/OLAP benchmarks defined by TPC, but due to the emerging hybrid OLTP/OLAP systems and big data technologies, these trends have changed (Bog 2013) adapting the new system features. For example, the database schema and ACID properties are not anymore a key requirement in the NoSQL and big data management systems and are replaced by more general one like system under test (SUT). For example, new categories in TPCx-BB are:

- System under test – Describes the system architecture with its hardware and software components and their configuration requirements.
- Pricing – Defines the pricing of the components in the system under test including the system maintenance.
- Energy – Defines the methodology, rules, and metrics to measure the energy consumption of the system under test in the TPC benchmarks.

The above components are part of the standard TPC benchmark specifications and are not representative for the entire analytics benchmarks

spectrum. Many of the newly defined big data benchmarks are open-source programs. However, the main characteristics of a good domain-specific benchmark are still the same. Jim Gray (1992) defined four important criteria that domain-specific benchmarks must meet:

- Relevant: It must measure the peak performance and price/performance of systems when performing typical operations within that problem domain.
- Portable: It should be easy to implement the benchmark on many different systems and architectures.
- Scalable: The benchmark should apply to small and large computer systems. It should be possible to scale the benchmark up to larger systems and to parallel computer systems as computer performance and architecture evolve.
- The benchmark must be understandable/interpretable; otherwise it will lack credibility.

Similarly, Karl Huppler (2009) outlines five key characteristics that all good benchmarks have:

- Relevant – A reader of the result believes the benchmark reflects something important.
- Repeatable – There is confidence that the benchmark can be run a second time with the same result.
- Fair – All systems and/or software being compared can participate equally.
- Verifiable – There is confidence that the documented result is real.
- Economical – The test sponsors can afford to run the benchmark.

In reality, many of the new benchmarks (in Tables 4, 5 and 6) do not have clear specifications and do not follow the practices defined by Gray (1992) and Huppler (2009) but just provide a workload implementation that can be used in many scenarios. This opens the challenge that the reported benchmark results are not really comparable and strictly depend on the environment in which they were obtained.

In terms of component specification, the situation looks similar. All TPC benchmarks use synthetic data generators, which allow for scalable and deterministic workload generation. However, many new benchmarks use open data sets or real workload traces like BigDataBench (Wang et al. 2014) or a mix between real data and synthetically generated data. This influences also the metrics reported by these benchmarks. They are often not clearly specified or very simplistic (like execution time) and cannot be used for an accurate comparison between different environments.

The ongoing evolution in the big data systems and the data science, machine learning, and deep learning tools and techniques will open many new challenges and questions in the design and specification of standardized analytics benchmarks. There is a growing need for new standardized big data analytics benchmarks and metrics.

Key Applications

The analytics benchmarks can be used for multiple purposes and in different environments. For example, vendors of database-related products can use them to test the features of their data products both in the process of development and after it is released, to position them in the market. The final benchmarking of a data product is usually done by an accredited organization. For example, TPC and SPEC have certified auditors that perform transparent auditing of the complete benchmarking process. The database and big data system administrators can regularly run benchmarks to ensure that the systems are properly configured and perform as expected. Similarly, system architects and application developers use benchmarks to test and compare the performance of different data storage technologies in the process of choosing the best tool for their requirements. Furthermore, benchmarks can be used for different comparisons as in the four categories defined by Jim Gray 1992:

- **To compare different software and hardware systems**: The goal is to use metric reported by the benchmark as a comparable unit for evaluating the performance of different data technologies on different hardware running the same application. This case represents classical competitive situation between hardware vendors.
- **To compare different software on one machine**: The goal is to use the benchmark to evaluate the performance of two different software products running on the same hardware environment.This case represents classical competitive situation between software vendors.
- **To compare different machines in a comparable family**: The objective is to compare similar hardware environments by running the same software product and application benchmark on each of them. This case represents a comparison of different generations of vendor hardware or for a case comparing of different hardware vendors.
- **To compare different releases of a product on one machine**: The objective is to compare different releases of a software product by running benchmark experiments on the same hardware. Ideally the new releases should perform faster (based on the benchmark metric) than its predecessors. This can be also seen as performance regression tests that can assure the new release support all previous system features.

Cross-References

- ▸ Auditing
- ▸ Benchmark Harness
- ▸ CRUD Benchmarks
- ▸ Component Benchmark
- ▸ End-to-End Benchmark
- ▸ Energy Benchmarking
- ▸ Graph Benchmarking
- ▸ Metrics for Big Data Benchmarks
- ▸ Microbenchmark
- ▸ SparkBench
- ▸ Stream Benchmarks
- ▸ System Under Test
- ▸ TPC
- ▸ TPC-DS
- ▸ TPC-H
- ▸ TPCx-HS
- ▸ Virtualized Big Data Benchmarks
- ▸ YCSB

References

Abadi D, Babu S, Ozcan F, Pandis I (2015) Tutorial: SQL-on-Hadoop systems. PVLDB 8(12):2050–2051

Agrawal D, Butt AR, Doshi K, Larriba-Pey J, Li M, Reiss FR, Raab F, Schiefer B, Suzumura T, Xia Y (2015) SparkBench – a spark performance testing suite. In: TPCTC, pp 26–44

Alsubaiee S, Altowim Y, Altwaijry H, Behm A, Borkar VR, Bu Y, Carey MJ, Cetindil I, Cheelangi M, Faraaz K, Gabrielova E, Grover R, Heilbron Z, Kim Y, Li C, Li G, Ok JM, Onose N, Pirzadeh P, Tsotras VJ, Vernica R, Wen J, Westmann T (2014) Asterixdb: a scalable, open source BDMS. PVLDB 7(14):1905–1916

AMPLab (2013) https://amplab.cs.berkeley.edu/benchmark/

Andersen B, Pettersen PG (1995) Benchmarking handbook. Champman & Hall, London

Armbrust M, Xin RS, Lian C, Huai Y, Liu D, Bradley JK, Meng X, Kaftan T, Franklin MJ, Ghodsi A, Zaharia M (2015) Spark SQL: relational data processing in spark. In: Proceedings of the 2015 ACM SIGMOD international conference on management of data, Melbourne, 31 May–4 June 2015, pp 1383–1394

Armstrong TG, Ponnekanti V, Borthakur D, Callaghan M (2013) Linkbench: a database benchmark based on the Facebook social graph. In: Proceedings of the ACM SIGMOD international conference on management of data, SIGMOD 2013, New York, 22–27 June 2013, pp 1185–1196

AsterixDB (2018) https://asterixdb.apache.org

BigFrame (2013) https://github.com/bigframeteam/BigFrame/wiki

Bog A (2013) Benchmarking transaction and analytical processing systems: the creation of a mixed workload benchmark and its application. PhD thesis. http://d-nb.info/1033231886

Carbone P, Katsifodimos A, Ewen S, Markl V, Haridi S, Tzoumas K (2015) Apache Flink™: stream and batch processing in a single engine. IEEE Data Eng Bull 38(4):28–38. http://sites.computer.org/debull/A15dec/p28.pdf

Cattell R (2011) Scalable SQL and NoSQL data stores. SIGMOD Rec 39(4):12–27

Codd EF, Codd SB, Salley CT (1993) Providing OLAP (On-line analytical processing) to user-analysis: an IT mandate. White paper

Cooper BF, Silberstein A, Tam E, Ramakrishnan R, Sears R (2010) Benchmarking cloud serving systems with YCSB. In: Proceedings of the 1st ACM symposium on cloud computing, SoCC 2010, Indianapolis, 10–11 June 2010, pp 143–154

Ferdman M, Adileh A, Koçberber YO, Volos S, Alisafaee M, Jevdjic D, Kaynak C, Popescu AD, Ailamaki A, Falsafi B (2012) Clearing the clouds: a study of emerging scale-out workloads on modern hardware. In: Proceedings of the 17th international conference on architectural support for programming languages and operating systems, ASPLOS, pp 37–48

Ferrarons J, Adhana M, Colmenares C, Pietrowska S, Bentayeb F, Darmont J (2013) PRIMEBALL: a parallel processing framework benchmark for big data applications in the cloud. In: TPCTC, pp 109–124

Flink (2018) https://flink.apache.org/

Gelly (2015) https://flink.apache.org/news/2015/08/24/introducing-flink-gelly.html

Ghazal A, Rabl T, Hu M, Raab F, Poess M, Crolotte A, Jacobsen H (2013) Bigbench: towards an industry standard benchmark for big data analytics. In: Proceedings of the ACM SIGMOD international conference on management of data, SIGMOD 2013, New York, 22–27 June 2013, pp 1197–1208

Ghazal A, Ivanov T, Kostamaa P, Crolotte A, Voong R, Al-Kateb M, Ghazal W, Zicari RV (2017) Bigbench V2: the new and improved bigbench. In: 33rd IEEE international conference on data engineering, ICDE 2017, San Diego, 19–22 Apr 2017, pp 1225–1236

GraphX (2018) https://spark.apache.org/graphx/

Gray J (1992) Benchmark handbook: for database and transaction processing systems. Morgan Kaufmann Publishers Inc., San Francisco

Hadoop (2018) https://hadoop.apache.org/

Han R, John LK, Zhan J (2018) Benchmarking big data systems: a review. IEEE Trans Serv Comput 11(3):580–597

Hellerstein JM, Ré C, Schoppmann F, Wang DZ, Fratkin E, Gorajek A, Ng KS, Welton C, Feng X, Li K, Kumar A (2012) The MADlib analytics library or MAD skills, the SQL. PVLDB 5(12):1700–1711

Hive (2018) https://hive.apache.org/

Hockney RW (1996) The science of computer benchmarking. SIAM, Philadelphia

Hogan T (2009) Overview of TPC benchmark E: the next generation of OLTP benchmarks. In: Performance evaluation and benchmarking, first TPC technology conference, TPCTC 2009, Lyon, 24–28 Aug 2009, Revised Selected Papers, pp 84–98

Hu H, Wen Y, Chua T, Li X (2014) Toward scalable systems for big data analytics: a technology tutorial. IEEE Access 2:652–687

Huang S, Huang J, Dai J, Xie T, Huang B (2010) The HiBench benchmark suite: characterization of the MapReduce-based data analysis. In: Workshops proceedings of the 26th IEEE ICDE international conference on data engineering, pp 41–51

Huppler K (2009) The art of building a good benchmark. In: Nambiar RO, Poess M (eds) Performance evaluation and benchmarking. Springer, Berlin/Heidelberg, pp 18–30

Impala (2018) https://impala.apache.org/

Ivanov T, Rabl T, Poess M, Queralt A, Poelman J, Poggi N, Buell J (2015) Big data benchmark compendium. In: TPCTC, pp 135–155

Kemper A, Neumann T (2011) Hyper: a hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In: Proceedings of the 27th international conference on data engineering, ICDE 2011, Hannover, 11–16 Apr 2011, pp 195–206

Kim K, Jeon K, Han H, Kim SG, Jung H, Yeom HY (2008) Mrbench: a benchmark for mapreduce framework. In: 14th international conference on parallel and distributed systems, ICPADS 2008, Melbourne, 8–10 Dec 2008, pp 11–18

Kornacker M, Behm A, Bittorf V, Bobrovytsky T, Ching C, Choi A, Erickson J, Grund M, Hecht D, Jacobs M, Joshi I, Kuff L, Kumar D, Leblang A, Li N, Pandis I, Robinson H, Rorke D, Rus S, Russell J, Tsirogiannis D, Wanderman-Milne S, Yoder M (2015) Impala: a modern, open-source SQL engine for Hadoop. In: CIDR 2015, seventh biennial conference on innovative data systems research, Asilomar, 4–7 Jan 2015, Online proceedings

Li M, Tan J, Wang Y, Zhang L, Salapura V (2015) SparkBench: a comprehensive benchmarking suite for in memory data analytic platform spark. In: Proceedings of the 12th ACM international conference on computing frontiers, pp 53:1–53:8

Luo C, Zhan J, Jia Z, Wang L, Lu G, Zhang L, Xu C, Sun N (2012) CloudRank-D: benchmarking and ranking cloud computing systems for data processing applications. Front Comp Sci 6(4):347–362

MADlib (2018) https://madlib.apache.org/

Meng X, Bradley JK, Yavuz B, Sparks ER, Venkataraman S, Liu D, Freeman J, Tsai DB, Amde M, Owen S, Xin D, Xin R, Franklin MJ, Zadeh R, Zaharia M, Talwalkar A (2016) Mllib: machine learning in Apache spark. J Mach Learn Res 17:34:1–34:7

MLlib (2018) https://spark.apache.org/mllib/

Nambiar R (2014) Benchmarking big data systems: introducing TPC express benchmark HS. In: Big data benchmarking – 5th international workshop, WBDB 2014, Potsdam, 5–6 Aug 2014, Revised Selected Papers, pp 24–28

Nambiar RO, Poess M (2006) The making of TPC-DS. In: Proceedings of the 32nd international conference on very large data bases, Seoul, 12–15 Sept 2006, pp 1049–1058

Nambiar R, Chitor R, Joshi A (2012) Data management – a look back and a look ahead. In: Specifying big data benchmarks – first workshop, WBDB 2012, San Jose, 8–9 May 2012, and second workshop, WBDB 2012, Pune, 17–18 Dec 2012, Revised Selected Papers, pp 11–19

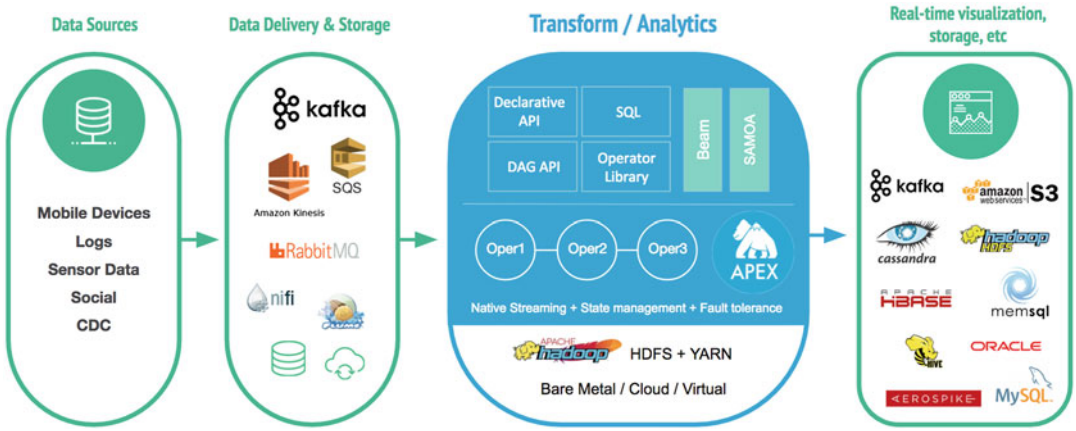
Özcan F, Tian Y, Tözün P (2017) Hybrid transactional/-analytical processing: a survey. In: Proceedings of the 2017 ACM international conference on management of data, SIGMOD conference 2017, Chicago, 14–19 May 2017, pp 1771–1775
Patil S, Polte M, Ren K, Tantisiriroj W, Xiao L, López J, Gibson G, Fuchs A, Rinaldi B (2011) YCSB++: benchmarking and performance debugging advanced features in scalable table stores. In: ACM symposium on cloud computing in conjunction with SOSP 2011, SOCC'11, Cascais, 26–28 Oct 2011, p 9
Pavlo A, Paulson E, Rasin A, Abadi DJ, DeWitt DJ, Madden S, Stonebraker M (2009) A comparison of approaches to large-scale data analysis. In: Proceedings of the ACM SIGMOD international conference on management of data, SIGMOD 2009, Providence, 29 June–2 July 2009, pp 165–178
Pirzadeh P, Carey MJ, Westmann T (2015) BigFUN: a performance study of big data management system functionality. In: 2015 IEEE international conference on big data, pp 507–514
Poess M (2012) Tpc's benchmark development model: making the first industry standard benchmark on big data a success. In: Specifying big data benchmarks – first workshop, WBDB 2012, San Jose, 8–9 May 2012, and second workshop, WBDB 2012, Pune, 17–18 Dec 2012, Revised Selected Papers, pp 1–10
Poess M, Rabl T, Jacobsen H, Caufield B (2014) TPC-DI: the first industry benchmark for data integration. PVLDB 7(13):1367–1378
Poess M, Rabl T, Jacobsen H (2017) Analysis of TPC-DS: the first standard benchmark for SQL-based big data systems. In: Proceedings of the 2017 symposium on cloud computing, SoCC 2017, Santa Clara, 24–27 Sept 2017, pp 573–585
Pöss M, Floyd C (2000) New TPC benchmarks for decision support and web commerce. SIGMOD Rec 29(4):64–71
Pöss M, Nambiar RO, Walrath D (2007) Why you should run TPC-DS: a workload analysis. In: Proceedings of the 33rd international conference on very large data bases, University of Vienna, 23–27 Sept 2007, pp 1138–1149
Raab F (1993) TPC-C – the standard benchmark for online transaction processing (OLTP). In: Gray J (ed) The benchmark handbook for database and transaction systems, 2nd edn. Morgan Kaufmann, San Mateo
Rockart JF, Ball L, Bullen CV (1982) Future role of the information systems executive. MIS Q 6(4): 1–14
Sakr S, Liu A, Fayoumi AG (2013) The family of MapReduce and large-scale data processing systems. ACM Comput Surv 46(1):11:1–11:44
Sangroya A, Serrano D, Bouchenak S (2012) MRBS: towards dependability benchmarking for Hadoop MapReduce. In: Euro-Par: parallel processing workshops, pp 3–12
Sethuraman P, Taheri HR (2010) TPC-V: a benchmark for evaluating the performance of database applications in virtual environments. In: Performance evaluation, measurement and characterization of complex systems – second TPC technology conference, TPCTC 2010, Singapore, 13–17 Sept 2010. Revised Selected Papers, pp 121–135
Shim JP, Warkentin M, Courtney JF, Power DJ, Sharda R, Carlsson C (2002) Past, present, and future of decision support technology. Decis Support Syst 33(2):111–126
Spark (2018) https://spark.apache.org
SparkSQL (2018) https://spark.apache.org/sql/
SparkStreaming (2018) https://spark.apache.org/streaming/
SPEC (2018) www.spec.org/
STAC (2018) www.stacresearch.com/
Storm (2018) https://storm.apache.org/
Tensorflow (2018) https://tensorflow.org
Thusoo A, Sarma JS, Jain N, Shao Z, Chakka P, Anthony S, Liu H, Wyckoff P, Murthy R (2009) Hive – a warehousing solution over a map-reduce framework. PVLDB 2(2):1626–1629
TPC (2018) www.tpc.org/
Wang L, Zhan J, Luo C, Zhu Y, Yang Q, He Y, Gao W, Jia Z, Shi Y, Zhang S, Zheng C, Lu G, Zhan K, Li X, Qiu B (2014) BigDataBench: a big data benchmark suite from internet services. In: 20th IEEE international symposium on high performance computer architecture, HPCA 2014, pp 488–499

Apache Apex

Ananth Gundabattula[1] and Thomas Weise[2]
[1]Commonwealth Bank of Australia, Sydney, NSW, Australia
[2]Atrato Inc., San Francisco, CA, USA

Introduction

Apache Apex (2018; Weise et al. 2017) is a large-scale stream-first big data processing framework that can be used for low-latency, high-throughput, and fault-tolerant processing of unbounded (or bounded) datasets on clusters. Apex development started in 2012, and it became a project at the Apache Software Foundation in 2015. Apex can be used for real-time and batch processing, based



Apache Apex, Fig. 1 Apex as distributed stream processor

on a unified stateful streaming architecture, with support for event-time windowing and exactly-once processing semantics (Fig. 1).

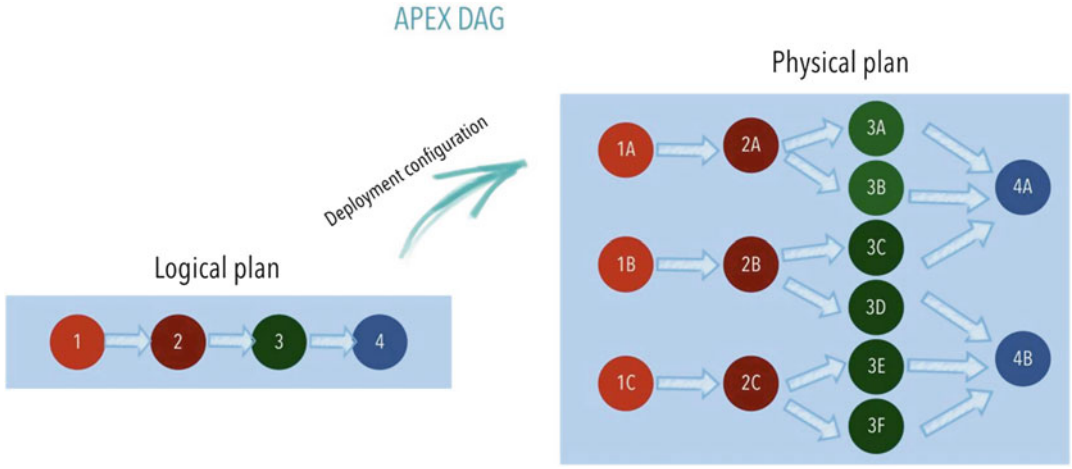
Application Model and APIs

DAG: The processing logic of an Apex application is represented by a directed acyclic graph (DAG) of operators and streams. Streams are unbounded sequences of events (or tuples), and operators are the atomic functional building blocks for sources, sinks, and transformations. With the DAG, arbitrary complex processing logic can be arranged in sequence or in parallel. With an acyclic graph, the output of an operator can only be transmitted to downstream operators. For pipelines that require a loop (or iteration), a special delay operator is supported that allows the output of an operator to be passed back as input to upstream operators (often required in machine learning). The engine also defines a module interface, which can be used to define composite operators that represent a reusable DAG fragment (Fig. 2).

The user-defined DAG is referred to as the logical plan. The Apex engine will expand it into the physical plan, where operators are partitioned (for parallelism) and grouped into containers for deployment. Attributes in the logical plan can influence these translations, such as the affinity to control which operators should be deployed into the same thread, process, or host.

Low-level, compositional API: The Apex engine defines the low-level API, which can be used to assemble a pipeline by composing operators and streams into a DAG. The API offers a high degree of flexibility and control: Individual operators are directly specified by the developer, and attributes can be used to control details such as resource constraints, affinity, stream encoding, and more. On the other hand, the API tends to be more verbose for those use cases that don't require such flexibility. That's why Apex, as part of the library, offers higher-level constructs, which are based on the DAG API.

High-level, declarative API: The Apex library provides the high-level stream API, which allows the application developer to specify the application through a declarative, fluent style API (similar to API found in Apache Spark and Apache Flink). Instead of identifying individual operators, the developer specifies sources, transformations, and sinks by chaining method calls on the stream interface. The API internally keeps track of operator(s) and streams for eventual expansion into the lower-level DAG. The stream API does not require knowledge of individual operator classes and more concise for use cases that don't require advanced constructs.



Apache Apex, Fig. 2 Apex converts a logical plan into a physical execution model by means of configuration

It is still possible to add customizations as the needs of the application evolve.

SQL: SQL is important for analytics and widely used in the big data ecosystem, with BI tools that can connect to engines such as Apache Hive, Apache Impala, Apache Drill, and others, besides the traditional database systems. SQL isn't limited to querying data; it can also be used to specify transformations. Recent efforts to bring SQL to stream processing like Apache Calcite (2018), Flink (Carbone et al. 2015a), Beam (Akidau et al. 2015), and KSQL (Confluent blog 2018) promise to target a wider audience without lower-level programming expertise, for use cases including ad hoc data exploration, ETL, and more. Apex provides a SQL API that is implemented based on Calcite. It currently covers select, insert, where clause, inner join, and scalar functions. Supported endpoints (read and write) can be file, Kafka, or any other stream defined with the DAG API.

Operators

Operators are Java classes that implement the *Operator* interface and other optional interfaces that are defined in the Apex API and recognized by the engine.

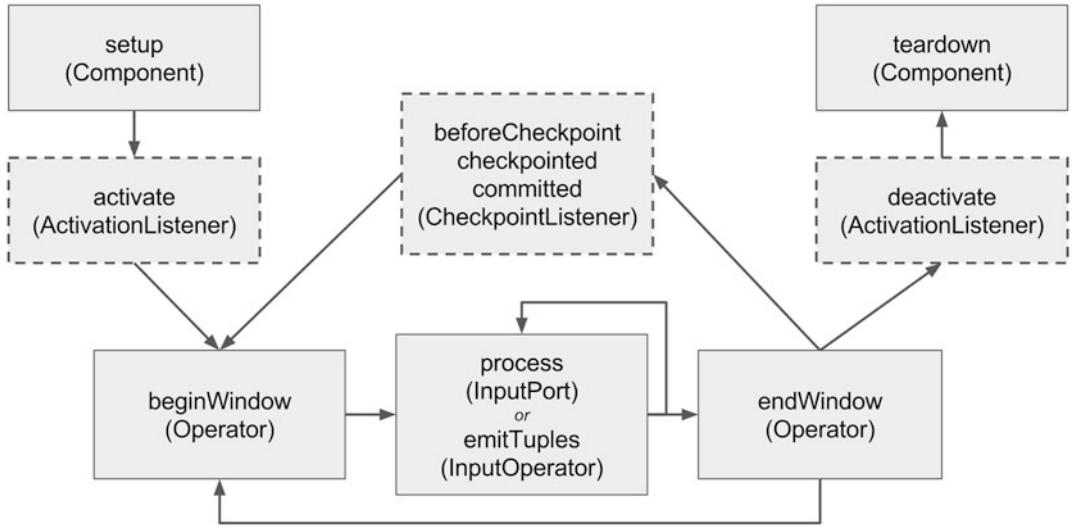
Operators have *ports* (type-safe connection points for the streams) to receive input and emit output. Each operator can have multiple ports. Operators that don't have input ports are sources, and operators that don't emit output are sinks. These operators interface with external systems. Operators that have both types of ports typically transform data.

The operator interfaces inform how the engine will interact with the operator at execution time, once it is deployed into a container (Fig. 3).

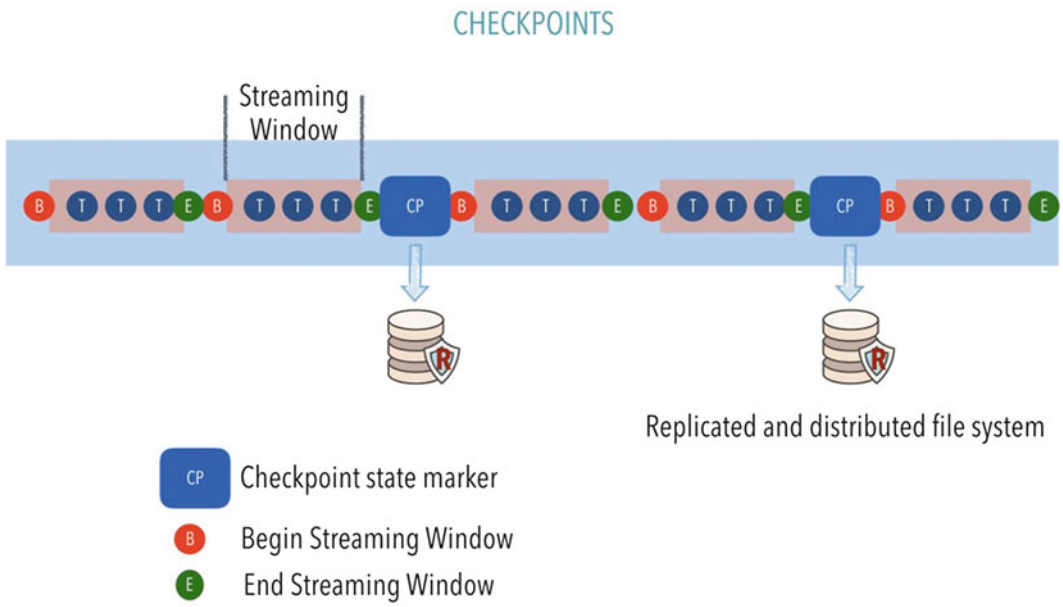
Apex has a continuous operator model; the operator, once deployed, will process data until the pipeline is terminated. The *setup* and *activate* calls can be used for initialization. From then on, data (individual events) will be processed. Periodically the engine will call begin/endWindow to demarcate streaming intervals (processing time). This presents an opportunity to perform work that is less suitable for every event. The operator can also implement the optional *CheckpointListener* interface to perform work at checkpoint boundaries.

Checkpointing

Checkpointing in Apex is the foundation for failure handling as well as on-demand parallelism a.k.a. dynamic partitioning. Checkpointing enables application recovery in case of failure by storing the state of the DAG at configurable units



Apache Apex, Fig. 3 Operator lifecycle and interfaces



Apache Apex, Fig. 4 Checkpoint markers injected into the tuple stream at streaming window boundaries

of processing time windows. In case of partitioning (described in the next section), checkpointing allows for resharding the operator states for existing as well as newly spawned operators for elastic scaling (Fig. 4).

Asynchronous and decentralized: Checkpointing is enabled by the framework injecting checkpoint markers into the stream at configured intervals of ingress/arrival time. An operator instance on the arrival of this marker triggers the checkpointing mechanism. Saving of the state itself is executed asynchronously, without blocking the operator. Completed checkpoints are reported to the application master, which has a view of the entire DAG and can mark a checkpoint as "committed" when it was reported by all operators.

Checkpointing in Apex is thus an asynchronous, lightweight, and decentralized approach.

Idempotency for resumption from checkpoint: Introduction of checkpoint markers into the stream in Apex is a slight variation from Carbone et al. (2015b). The checkpoint markers are introduced at the input operator (source) and traverse the entire DAG. Upon resumption from failure, the stream needs to be replayed in the exact same sequence to enable the entire application as a reproducible state machine. Apex provides the necessary building blocks to achieve idempotency. Individual operators in the DAG ensure idempotency is maintained in their processing logic. For each stream that crosses a container/process boundary, Apex provides a buffer server like (Lin et al. 2016) that can be used by a downstream operator to start consumption from a specific position. This enables fine-grained recovery and dynamic partitioning without a full DAG reset. In case of an input operator, idempotency of replay is either backed by capabilities of the source (in cases like Kafka and files by offset tracking and sequential scan) or the input operator can record the source data for replay by using a write-ahead log (WAL) implementation that is provided by the Apex library.

Opt-in independence: Operators in Apex can choose to opt out of checkpointing cycles to avoid serialization overhead. As an example, an operator that is representing a matrix transpose operation can choose to not be part of the checkpointing process as it does not need a state to be accumulated across tuples or across streaming windows. On the other hand, an operator which is computing a cumulative sum or part of sort, join logic in SQL DAG needs the state to be handed from one streaming window to another and hence needs to be stateful. Stateless operators can be enabled either by an annotation marker on the operator implementation or declaring it in the configuration provided as input to the application launch. For stateful operators, Apex ensures that the state is serialized at checkpoint intervals. The default serialization mechanism allows the operator implementation to decide (and optimize) which fields are serialized (or skipped by marking them transient) and also how fields are serialized through optional annotations.

Exactly-once state and exactly-once processing: Exactly-once processing is the holy grail of distributed stream processing engines. In reality though, reprocessing cannot be avoided in a distributed system, and therefore exactly-once refers to the effect on the state, which is also why it is alternatively referred to as "effectively once." While some engines define exactly-once semantics purely from internal state persistence point of view, others aim to provide constructs for exactly-once processing in addition to state with varying degrees of implementation complexity as given in Kulkarni et al. (2015), Noghabi et al. (2017), and Jacques-Silva et al. (2016). Exactly-once state management in Apex is a distributed process by the virtue of each operator triggering its own state passivation when the checkpoint tuple is processed. While some streaming systems only focus on exactly-once semantics for internal state, the Apex library provides support for end-to-end exactly-once for many of its connectors, thereby also covering the effect of processing on the state in respective external systems. This is possible through interface contracts in the low-level API and idempotency and by utilizing capabilities that are specific to the integrated system, such as transactions, atomic renames, etc.

At-most-once and at-least-once processing semantics: It is not atypical for at-least-once processing semantics for use cases like finding a max value in a given stream, while at-most-once processing is required for use cases where recency of data processing state matters the most. At-least-once semantics is achieved by the upstream operator replaying the tuples from the checkpoint and the downstream operator not implementing any lightweight checkpoint state-based checks. For at-most-once semantics, the upstream operator needs to just stream tuples to downstream without starting from a checkpoint.

Low-level API for processing semantics: As described in the Operator section, the operator is given a chance to perform business logic-specific process at streaming window boundaries as well as checkpoint boundaries. Several operators in Apex implement incremental lightweight state

saving at window boundaries and a complete operator state at checkpointing boundaries. Utilities exist in the framework to get/set data structures that represent state using a window ID marker as input thus allowing for a lightweight state passivation. Since a checkpoint resumption can result in reprocessing multiple windows, this incremental state can come in handy to completely skip windows that are processed from a certain checkpoint. Let us consider how this approach provides for exactly-once processing semantics in systems that do not provide two-phase commit transaction support. Until the last streaming window before a crash (referred to as orphaned window hereafter), state can be passivated incrementally at streaming window boundaries. For the orphaned window, for which neither a lightweight state has been persisted nor a full checkpoint cycle is complete, custom business logic can be invoked using a "check/read and then process" pattern. This "check and process along with a business helper function" is only executed for the orphaned window reprocessing. Processing resumes a normal pattern beyond this orphaned window processing. Thus a combination of lightweight checkpointing at window boundaries, full checkpointing at checkpoint boundaries, and the check and process pattern can achieve exactly-once processing semantics in Apex for this class of systems.

Managed state and spillable data structures: State saving at checkpointing boundary may be sub-optimal when the state is really large (Del Monte 2017). Incremental state persistence at smaller intervals and/or spillable state can be effective patterns to mitigate this (To et al. 2017; Fernandez et al. 2013; Akidau et al. 2013; Sebepou and Magoutis 2011). Apex supports both. To handle very large state, the Apex library provides for spillable data structures similar to Carbone et al. (2017). Instead of a full copy local store like RocksDB, spillable data structures in Apex are block structured and by default backed by the distributed file system (DFS), allowing for delta writes and on-demand load (from DFS) when operators are restored.

Flexibility, ease of use, and efficiency have been core design principles for Apex, and checkpointing is no exception. When desired, the user has complete control to define the data structures that represent the operator state. For recovery, Apex will only reset to checkpoint the part of the DAG that is actually affected by a failure. These are examples of features that set Apex apart from other streaming frameworks like Zaharia et al. (2012) and Carbone et al. (2015b). With flexibility at the lower level, higher-level abstractions are provided as part of the library that optimize for specific use cases.

High Availability

High availability in Apex is achieved by extending two primary constructs. Leverage YARN as cluster manager to handle process and node failure scenarios and use the distributed file system to recover the state. An Apex application is associated with an application ID and is either given a new ID or resume from a previous application identity as identified by an ID given as a startup parameter. Apex being a YARN native application utilizes the YARN Resource manager to allocate the application master (AM) at the time of launch. AM either instantiates a new physical deployment plan using the logical plan or reuses an existing physical plan if an existing application ID is passed at startup. As the DAG is physicalized and starts executing, each of the JVM operators sends heartbeats to the AM using a separate thread. Subsequent to this, there are many scenarios that can result in a fault scenario and recovery options possible as given in Nasir (2016).

Worker container failure: In case of a container JVM crashing or stalling for a long time due to a GC pause, AM detects the absence of the heartbeat and initiates a kill (if applicable) and redeploys sequence. AM would negotiate with YARN for the replacement container and request to kill the old container (if it was stalled). The new instance would then resume from a checkpointed state. It may be noted that the entire sub-DAG downstream to the failed operator will be redeployed as well to ensure the semantics of processing are upheld especially for cases when the

downstream operators are implementing exactly-once semantics.

AM failure: During an AM crash, the individual worker containers continue processing the tuples albeit without any process taking care of the recovery and monitoring. In the meantime, YARN would detect that AM container has gone down and redeploy a new instance of it and pass the previous application ID as part of the restart process. This new instance would then resume from its checkpointed state. The checkpointed state of AM is more related to the execution state of the DAG. This new AM container would update its metadata in the distributed file system representing the application ID which in turn would be picked up by the worker containers to reestablish their heartbeat cycles.

Machine failures: In the event of a physical node failure, YARN resource manager (RM) would be notified of the node manager (NM) death. This would result in RM deploying all containers currently running on that NM. It is possible that there are multiple containers of the application on the failed host, and all of these would be migrated to new host(s) using the process described before.

Resource Management

Apex is a YARN native application that follows the YARN principles of resource request and grant model. The application master upon launch generates a physical execution plan from the DAG logical representation. The application master itself is a YARN container. The application master then spawns the operator instances by negotiating operator containers from the YARN resource manager, taking into account the memory and compute resource settings in the DAG (Fig. 5).

Deployment models: The Apex operator deployment model allows for four different patterns:

1. Thread – Operator logic is invoked by a single thread in the same JVM.
2. Container – Operators coexist in the same JVM.
3. Host – Operators are available on the same node.
4. Default – Highest cost in terms of serialization and IPC.

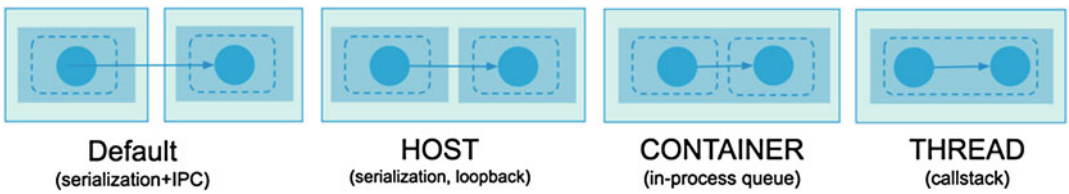
Affinity or anti-affinity patterns further allow for customized location preferences while deploying the operators by the application master. Affinity allows for multiple operators to be located on the same host or share the same container or thread.

Partitioning/Scaling

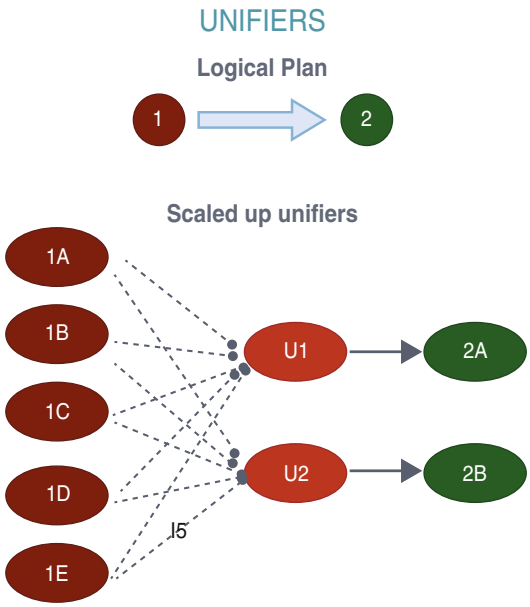
Operator parallelism a.k.a partitioning helps in dealing with latency and throughput trade-off aspects of a streaming application design. Apex enables developers to concentrate on the business logic and enable a configuration-based approach to scale a logical implementation to a scalable physical deployment model. This approach is referred to as partitioning and is enabled by configuring a partitioner for an operator.

Partitioners

A partitioner implementation can be specified how to partition the operator instances. While use case like specifying a fixed number of partitions



Apache Apex, Fig. 5 Operator deployment models



Apache Apex, Fig. 6 Unifiers allow for varying parallelism between operators. U1 and U2 unifiers are automatically injected into the DAG at deployment time

at startup can be met by using one of the partitioners available as part of the framework, Apex allows for custom implementation as well.

Such custom implementations can cater to many use cases like deciding partitions on the input source system characteristics. As an example, the Kafka partitioners as part of the Apex library have the capability to scale to as many Kafka partitions that each of the configured Kafka topics have or alternatively support mapping multiple Kafka partitions to one Apex operator partition. A partitioner can be stateful; it can re-shard the state of the old partitions (Fig. 6).

Unifiers

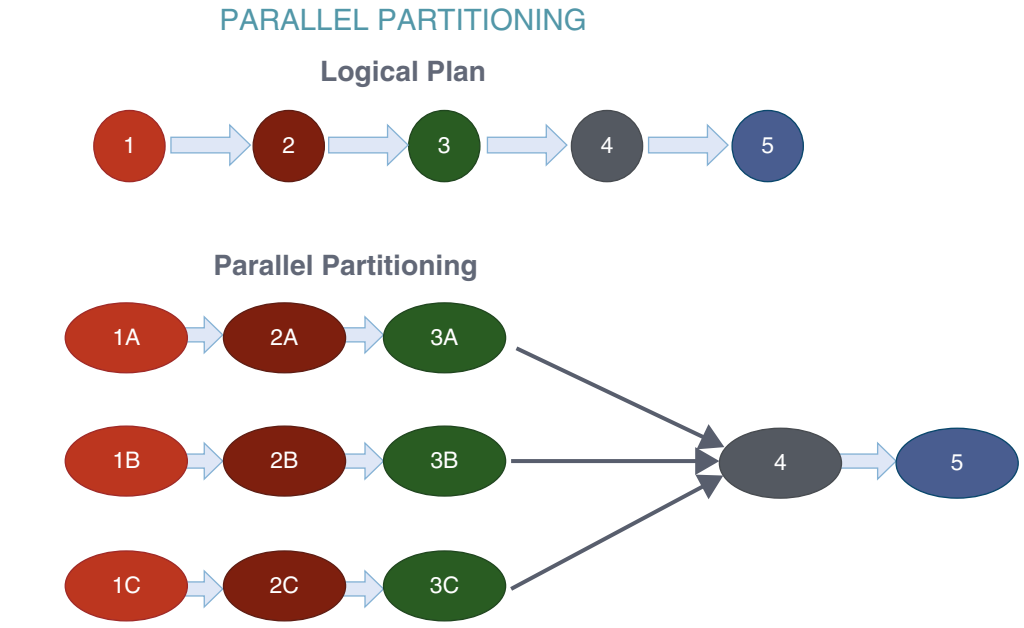
While partitioning allows for scalability of the operator business logic, it invariably results in an impedance mismatch if the downstream operator is not at the right scaling factor. Apex allows parallelism to vary between operators. In such cases, unifiers are automatically injected into the physical plan and define how the upstream results are shuffled to the new parallelism level. The user can customize the unifier implementation, which typically depends on the operator logic. A unifier is an operator without input port that is set on the output port of the partitioned operator. Cascading unifier patterns can also be implemented wherein the partitioned operator output can itself be shuffled in stages before streaming the merged result to the downstream operator. For a reduce operation, this can be used to overcome resource limits (network, CPU, etc.) for a latency trade-off (Fig. 7).

Parallel Partitioning

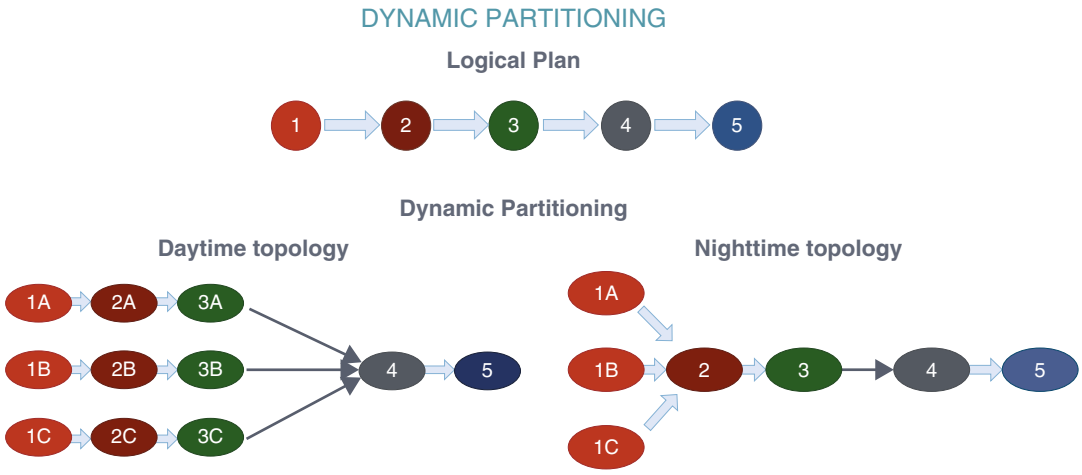
While partitioners and unifiers allow for independent scaling of each operator of an application, there will be use cases where downstream operators can align with the upstream operators' level of parallelism and avoid a unifier, thus further decreasing the overall latencies due to decreased network hops (in some systems, this also referred to as chaining). This is enabled via a configuration parameter set on the downstream operator, and this parallel partitioning can be configured for as many downstream operators as the application design mandates it to be (Fig. 8).

Dynamic Partitioning

Load variation is a common pattern across many streaming systems, and some of these engines provide for dynamic scaling as given in Floratou et al. (2017) and Bertolucci et al. (2015). Also the advent of cloud computing where cost is based on resource consumption models as given in Hummer et al. (2013) and Sattler and Beier (2013) makes compelling case for the need to dynamically adjust resources based on context. Static partitioning may not be sufficient to achieve latency SLAs or optimize resource consumption. Apex allows partitions to be scaled or contracted dynamically at checkpointing boundaries. A partitioner implementation can use the operational metrics of the physical operator instances to decide on the optimal scaling configuration. Unlike other streaming engines like Spark which need stand-alone shuffle service, Apex dynamic partitioning can be aligned with the application patterns.



Apache Apex, Fig. 7 Parallel partitioning can avoid shuffling overheads when it is not necessary



Apache Apex, Fig. 8 Dynamic partitioning allows for scale-up and scale-down strategies at runtime. Example above has different topologies at daytime and nighttime allowing for efficient use of hardware

Integration Using Apex Library

Apex enables a faster time to market model by shipping many connectors to external systems that can act as a source or a sink or both. The Apex library is also referred to as Malhar. Some of the common integrations include JDBC driver-enabled databases like Oracle, MySQL, and Postgres; replayable sources like Kafka and files; NOSQL databases like Cassandra and HBase; and JMS-enabled systems like MQ besides many others. The library implementations provide value add by not only implementing read/write patterns to these systems but also aiming to provide exactly-once processing.

File systems: Apex operators bring in lot more to the maturity of the implementation by enabling enterprise patterns like polling directories for new

file arrivals, offset tracking backed sequential scanning approach, resumption from checkpoints in case of failures, handling of varied formats of the file contents, and compression formats while writing contents and even splitting very large files into blocks while reading to enable efficient partitioning strategies.

Kafka: Kafka integration is one of the battle-tested integrations of Apex. Kafka operators allow for flexible mapping of operators wherein the mapping configuration can map "m" Kafka partitions across "n" Apex operators. It may be noted that m can represent partitions that may span across multiple Kafka clusters and multiple Kafka topics spread across these Kafka topics. Apex Kafka operators also support exactly-once semantics for Kafka versions prior to 0.11 version thus taking the burden of a transaction to Apex processing layer as opposed to relying on Kafka. The integration in Apex thus can be considered far richer as compared to the other systems.

Non-replayable sources like JMS: Exactly-once semantics at input sources is a bit more involved in case of systems like JMS wherein the source does not have a contract to replay a data point once a handover handshake is complete. Apex provides for a WAL implementation which is used by the JMS operator to replay a series of previously acknowledged messages when restarting from a checkpoint.

JDBC-enabled databases: JDBC-enabled databases like Oracle, Postgres, and MySQL allow for an exactly-once write pattern through transactions. The Apex connector can use this to commit transactions at streaming windows or checkpoint boundaries. The JDBC source supports idempotent read by tracking the offset (for a query with order by clause).

Conclusion

Apex is an enterprise-grade distributed streaming engine that comes with many foundational constructs as well as connectors that help in faster time to market as compared to some similar systems. Dockerized containers and support for next-generation container orchestration are some of the features that would benefit Apex in the near future.

References

Akidau T et al (2013) MillWheel: fault-tolerant stream processing at internet scale. PVLDB 6:1033–1044

Akidau T et al (2015) The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. PVLDB 8:1792–1803

Apache Apex (2018) https://apex.apache.org/

Apache Calcite (2018) https://calcite.apache.org/

Bertolucci M et al (2015) Static and dynamic big data partitioning on Apache Spark. PARCO

Carbone P et al (2015a) Apache Flink™: stream and batch processing in a single engine. IEEE Data Eng Bull 38:28–38

Carbone P et al (2015b) Lightweight asynchronous snapshots for distributed dataflows. CoRR abs/1506.08603: n. pag

Carbone P et al (2017) State management in Apache Flink®: consistent stateful distributed stream processing. PVLDB 10:1718–1729

Confluent blog (2018) https://www.confluent.io/blog/ksql-open-source-streaming-sql-for-apache-kafka/

Del Monte B (2017) Efficient migration of very large distributed state for scalable stream processing. PhD@VLDB

Fernandez RC et al (2013) Integrating scale out and fault tolerance in stream processing using operator state management. SIGMOD conference

Floratou A et al (2017) Dhalion: self-regulating stream processing in Heron. PVLDB 10:1825–1836

Hummer W et al (2013) Elastic stream processing in the cloud. Wiley Interdisc Rew: Data Min Knowl Discov 3:333–345

Jacques-Silva G et al (2016) Consistent regions: guaranteed tuple processing in IBM streams. PVLDB 9: 1341–1352

Kulkarni S et al (2015) Twitter Heron: stream processing at scale. SIGMOD conference

Lin W et al (2016) StreamScope: continuous reliable distributed processing of dig data streams. NSDI

Nasir MAU (2016) Fault tolerance for stream processing engines. CoRR abs/1605.00928: n. pag

Noghabi SA et al (2017) Stateful scalable stream processing at LinkedIn. PVLDB 10:1634–1645

Sattler K-U, Beier F (2013) Towards elastic stream processing: patterns and infrastructure. BD3@VLDB

Sebepou Z, Magoutis K (2011) CEC: continuous eventual checkpointing for data stream processing operators. In: 2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN). pp 145–156

To Q-C et al (2017) A survey of state management in big data processing systems. CoRR abs/1702.01596: n. pag

Weise T et al (2017) Learning Apache Apex. Packt Publishing

Zaharia M et al (2012) Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. NSDI
Zaharia M et al (2013) Discretized streams: fault-tolerant streaming computation at scale. SOSP

Apache Flink

Fabian Hueske and Timo Walther
Data Artisans GmbH, Berlin, Germany

Synonyms

Stratosphere platform

Definitions

Apache Flink is a system for distributed batch and stream processing. It addresses many challenges related to the processing of bounded and unbounded data. Among other things, Flink provides flexible windowing support, exactly-once state consistency, event-time semantics, and stateful stream processing. It offers abstractions for complex event processing and continuous queries.

Overview

Today, virtually all data is continuously generated as streams of events. This includes business transactions, interactions with web or mobile application, sensor or device logs, and database modifications. There are two ways to process continuously produced data, namely batch and stream processing. For stream processing, the data is immediately ingested and processed by a continuously running application as it arrives. For batch processing, the data is first recorded and persisted in a storage system, such as a file system or database system, before it is (periodically) processed by an application that processes a bounded data set. While stream processing typically achieves lower latencies to produce results, it induces operational challenges because streaming applications which run 24×7 make high demands on failure recovery and consistency guarantees.

The most fundamental difference between batch and stream processing applications is that stream processing applications process continuously arriving data. The two core building blocks when processing streaming data are state and time. Applications require state for every non-trivial computation that involves more than a single event. For example, state is required to collect multiple events before performing a computation or to hold the result of partial computations. Time on the other hand is important to determine when all relevant data was received and a computation can be performed. Having good control over time enables applications to treat result completeness for latency. Time is not relevant in batch processing because all input data is known and present when the processing starts.

Stateful stream processing is a very versatile architectural pattern that can be applied to a wide spectrum of data-related use cases. Besides providing better latency than batch processing solutions, stateful stream processing applications can also address use cases that are not suitable for batch processing approaches at all. Event-driven applications are stateful stream processing applications that ingest continuous streams of events, apply business logic to them, and emit new events or trigger external actions. These applications share more characteristics with transactional workloads than analytical applications. Another common use case for stateful stream processing is complex event processing (CEP), which is applied to evaluate patterns over event streams.

Historical Background

Apache Flink originates from an academic research project. In 2010, the Stratosphere project was started by five research groups from Technische Universität Berlin, Humboldt

Universität zu Berlin, and Hasso Plattner Institute Potsdam (http://gepris.dfg.de/gepris/projekt/132320961?language=en. Visited on 22 Dec 2017). The goal of the project was to develop novel approaches for large-scale distributed data processing. In the course of the project, the researchers developed the prototype of a data processing system to evaluate the new approaches and released the software as open source under the Apache software license (Alexandrov et al. 2014).

When the project started in 2010, the Apache Hadoop project (https://hadoop.apache.org. Visited on 22 Dec 2017), an open-source implementation of Google's MapReduce (Dean and Ghemawat 2008) and GFS (Ghemawat et al. 2003) publications, had gained a lot of interest in research and industry. MapReduce's strong points were its ability to scale data processing tasks to a large number of commodity machines and its excellent tolerance for hardware and software failures. However, the database research community had also realized that MapReduce was neither the most user-friendly nor most efficient approach to define complex data analysis applications. Therefore, the Stratosphere project aimed to build a system that combined the advantages of MapReduce and relational database systems. The first result was a system consisting of the PACT programming model, the distributed dataflow processing engine Nephele, and an optimizer that translated PACT programs into Nephele dataflows (Battré et al. 2010). The PACT programming model generalized the MapReduce programming model by providing more parallelizable operator primitives and defining programs as directed acyclic dataflows. Hence, specifying complex analytical applications became much easier (Alexandrov et al. 2011). The runtime operators to execute PACT programs were implemented based on well-known algorithms from database system literature, such as external merge-sort, block-nested loop join, hybrid-hash join, and sort-merge join. Having a choice in runtime operators and data distribution strategies due to the flexibility of the distributed dataflow execution engine Nephele resulted in different alternatives of how a PACT program could be executed. Similar to physical optimization in relational database systems, a cost-based optimizer enumerated execution plans under consideration of interesting and existing physical properties (such as partitioning, sorting, and grouping) and chose the least expensive plan for execution. Compared to the MapReduce programming and execution model, the Stratosphere research prototype provided a programming API that was as versatile as MapReduce but eased the definition of advanced data analysis applications, an efficient runtime based on concepts and algorithms of relational database systems, and a database-style optimizer to automatically choose efficient execution plans. At the same time, the prototype had similar hardware requirements and offered similar scalability as MapReduce. Based on the initial Stratosphere prototype, further research was conducted on leveraging static code analysis of user-defined function for logical program optimization (Hueske et al. 2012) and the definition and execution of iterative programs to support machine learning and graph analysis applications (Ewen et al. 2012).

In May 2014, the developers of the Stratosphere prototype decided to donate the source code of the prototype to the Apache Software Foundation (ASF) (https://wiki.apache.org/incubator/StratosphereProposal. Visited on 22 Dec 2017). Due to trademark issues, the project was renamed to Apache Flink. Flink is the German word for "nimble" or "swift." The initial group of committers consisted of the eight Stratosphere contributors and six members of the Apache Incubator to teach the Flink community the Apache Way. In August 2014, version 0.6 of Apache Flink was released as the first release under the new name Apache Flink (http://flink.apache.org/news/2014/08/26/release-0.6.html. Visited on 22 Dec 2017). Three months later in November 2014, version 0.7 was released. This release included a first version of Flink's DataStream API. With this addition, Flink was able to address stream as well as batch processing use cases with a single processing engine (Carbone et al. 2015a). Because the distributed dataflow processor (formerly known as Nephele) had always supported pipelined data transfers, the new stream processing capabilities did not require major changes

to the engine. In January 2015, 9 months after the start of the incubation, Flink left the incubator and became a top-level project of the Apache Software Foundation (https://blogs.apache.org/foundation/entry/the_apache_software_foundation_announces69. Visited 22 Dec 2017).

While until Flink's graduation the community was mostly working on batch processing features, stream processing slowly became the new focus of the community. Over the next couple of releases, features such as windowing support, exactly-once state consistency, event-time semantics, stateful stream processing, and high availability for worker and master processes were added. Moreover, the DataStream API was declared stable and support for persisting application state in savepoints, and restarting applications from savepoints as well as maintaining very large operator state in RocksDB were implemented. When version 1.0.0 was released in March 2016 (https://flink.apache.org/news/2016/03/08/release-1.0.0.html. Visited on 22 Dec 2017), Flink had become a fully fletched stream processor with a feature set that was unique among other open-source stream processors.

Since Flink's 1.0.0 release until today (December 2017, version 1.4.0), many more new significant features were added, such as SQL support for unified batch and streaming queries (https://flink.apache.org/news/2017/04/04/dynamic-tables.html. Visited on 22 Dec 2017), a complex event processing (CEP) library to identify and react on patterns in event streams (https://data-artisans.com/blog/complex-event-processing-flink-cep-update. Visited 22 Dec 2017), support for scaling applications in and out (Carbone et al. 2017), and support for querying operator state from external applications. By now, the Flink community has grown to 34 committers, and more than 350 individuals have contributed to Flink. Apache Flink is used in production at very large scale by enterprises around the world and across various industries, such as Alibaba, DellEMC, ING, King, Netflix, and Uber. Some of these users have built services for internal users or even expose these services to paying customers.

Foundations

Apache Flink is a system for batch and stream processing use cases (Carbone et al. 2015b). The main APIs, namely, the *DataSet API* for batch and *DataStream API* for streaming programs, allow to fluently specify a data processing plan by using first-order and second-order functions known from functional programming. *Second-order functions* give certain guarantees about the distributed execution. *First-order functions* implement custom business logic within the provided guarantees. For example, the map operator guarantees to apply a first-order function to every record, and keyBy partitions and distributes the stream by using a key specified in the first-order function. In the following, the discussion focuses on the DataStream API and Flink's stream processing capabilities since this reflects the current evolution of the system and the majority of its production use cases.

Figure 1 shows an example of a Flink DataStream API program that reads from a publish-subscribe messaging system. The program transforms the input by applying a user-defined map function to every record. Afterward, it partitions the records by a key and aggregates multiple records over a window of 5 seconds that discretizes the stream. The result is written into a continuous single log file. Flink automatically gathers information about the return type of each operator and generates appropriate record serializers for the network shipping. The API calls only construct a logical representation and are translated into a directed acyclic job graph that is submitted to the cluster for execution.

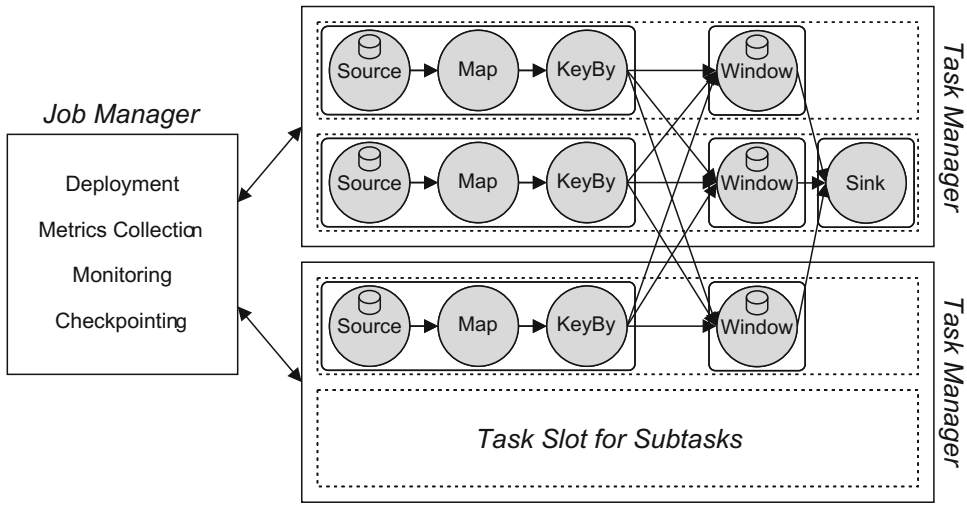
Figure 2 illustrates the distributed execution of the example program. A Flink setup consists of two types of processes: the *JobManager* and the *TaskManagers*. For high availability, there might be multiple standby JobManagers. The JobManager is the master that is responsible for cluster coordination, metrics collection, and monitoring. It receives a job graph and constructs an execution graph with knowledge of the available TaskManagers and their allocation. The execution graph is deployed on the TaskManagers. The TaskManagers are the workers that are

```
// setup the environment
StreamExecutionEnvironment env =
  StreamExecutionEnvironment.getExecutionEnvironment();

// define a streaming pipeline
env.addSource(new FlinkKafkaConsumer011<>(...))
  .map(new UserDefinedMapFunction())
  .keyBy("userId")
  .window(TumblingProcessingTimeWindows.of(Time.seconds(5)))
  .apply(new UserDefinedWindowFunction())
  .writeAsText("/file.log").setParallelism(1);

// execute the constructed plan
env.execute();
```

Apache Flink, Fig. 1 Apache Flink DataStream API program



Apache Flink, Fig. 2 Flink distributed execution model

responsible for executing operator pipelines and exchanging data streams.

The example from Fig. 2 is executed with a degree of parallelism equal to 3 and thus occupies three task slots. *Task slots* split the resources of a TaskManager into a finer granularity and define how many slices of a program can be executed concurrently by a TaskManager; the dotted lines in Fig. 2 highlight the task slots. Each pipeline is independently evaluated. Once an operator's result is ready, it will be forwarded immediately to the next operator without additional synchronization steps. A pipeline consists of one or more subtasks. Depending on the operations, a *subtask* executes only one operator, such as the window operator, or multiple operators that are

concatenated by so-called operator chaining. In the example, the reading of a record from the source, its transformation, and the extraction of a key are performed in a chain without having to serialize intermediate data. Both the keyBy and the sink operation break the chain, because keyBy requires a shuffling step and the sink has a lower parallelism than its predecessor.

Operators in Flink DataStream programs can be stateful. In our example, the sources need to store the current read offset in the log of the publish-subscribe system to remember which records have been consumed so far. Window operators need to buffer records until the final aggregation at the end of a window can be triggered. The JobManager is responsible for coordinating the creation of consistent checkpoints of a program's state, i.e., the state of all operators, and restarting the program in case of failures by loading the latest complete checkpoint and replaying parts of a stream. Flink's checkpoint and recovery mechanism will be covered later in more detail.

Time Handling

The handling of *time* receives special attention in Flink's DataStream API. Most streaming applications need a notion of time to define operations on conceptually never-ending inputs. The previous example included a *tumbling time window* to discretize the stream into segments containing the records received within an interval of 5 seconds. Similarly, *sliding windows* allow for similar fixed-length windows but with a possible overlap, e.g., a window of 1 minute length that is evaluated every 10 seconds. *Session windows* have a variable length and are evaluated depending on a gap of inactivity. These window types as well as several other operations on data streams depend on a notion of time to specify an interval of the stream that they interact with. Time-based operators need to be able to look up the "current time" while processing a record of a stream. Flink supports two time modes.

Processing time is defined by the local clock of the machine that processes an operator. Performing operations based on the wall-clock time is the easiest notion of time with little overhead. However, it assumes a perfect environment where all records arrive in a strict order, can be processed on time without any side effects, and do not need to be reprocessed. In the real world, events might arrive in the wrong order, or large amounts of events might arrive at the same time and, thus, logically belong to the same window but do not reach the window operator punctually. Moreover, the amount of hardware resource available for processing can affect which records are grouped together. Hence, processing time is not applicable if quality-of-service specifications require that results, which are computed from a log of events, and must be reproducible.

Event time addresses the issues above by relying on a timestamp that is associated with every record and by defining a strategy to making progress that guarantees consistent results. Flink adopts the *dataflow model* (Akidau et al. 2015). In addition to timestamped records, data streams need to be enriched with *watermarks* in order to leverage event time. Watermarks are metadata records with a timestamp that indicate that no record with a timestamp lower than the watermark's timestamp will be received from a connection in the future. Every operator in Flink contains logic to compute a new watermark from the watermarks it receives via its incoming connections. The watermark of an operator acts as its internal clock and triggers computations, such as the computation of a window when the watermark passes its end time. An operator tracks for each incoming connection the highest observed watermark and computes its own watermark as the smallest watermark across the current watermarks of its incoming connections. Whenever an operator advances its own watermark, it performs all computations that are triggered by the new watermark, emits the resulting records, and subsequently broadcasts its new watermark across all its outgoing connections. This mechanism ensures that watermarks are strictly increasing and that emitted records remain aligned with the emitted watermarks.

State and Fault Tolerance

The ability to memorize information that was received or computed for future computations is another crucial feature in streaming applications. *State* can be used to buffer records until a certain event occurs, to maintain (partial) aggregates based on incoming events, or to train and store machine learning models. From a user's point of view, state can be considered as a set of member variables in each operator. A member variable is empty at the beginning and can be accessed and modified depending on its data type. Since the values in state variables affect the computed result, state must be resilient to failures, recoverable, and flexible enough for rescaling.

Flink takes care of snapshotting the content of state variables by performing so-called Checkpoints in regular intervals (e.g., every minute). Flink employs *lightweight asynchronous snapshots* (Carbone et al. 2015c) based on the *Chandy-Lamport algorithm* (Mani Chandy and Lamport 1985) for distributed snapshots. When a snapshot is triggered, *checkpoint barriers* with a specific checkpoint ID are inserted at each source task and broadcasted across all outgoing connections. When an operator received a checkpoint barrier with the same checkpoint ID from all incoming connections, it draws a snapshot of its current state, starts to write it out asynchronously, and broadcasts the checkpoint barrier to all outgoing connections. A checkpoint is completed once all operators finished persisting their state. An important property of checkpoints is that they are consistent, i.e., the snapshotted state of all operators depends on the same set of input records, i.e., all records that were ingested before the checkpoint barriers were injected by the source tasks. In case of a failure, Flink restores the state of all operators from the most recent completed checkpoint. Many source operators persist the read positions (or offsets) on their input streams as regular state, such that read positions and operator states are consistently restored in case of a failure which results in exactly-once state consistency.

Usually, checkpoints are continuously replaced by newer ones. Flink allows for creating special persistent checkpoints, called *savepoints*. Savepoints can be created at any time and hold the complete state of an application. An application can be restarted from a savepoint, which means its internal state, usually including read positions on input streams, is completely restored. Savepoints make it possible to save multiple versions of a streaming application, recover state, or perform A/B testing with modified business logic but same state in a test environment.

State management also affects whether and how the parallelism of stateful operators can be changed, a feature that is important to support scale out of streaming applications during peak times or increasing loads and scale in afterward to save resources and money. Adjusting the parallelism of a stateful operator requires to redistribute its state to fewer or more parallel tasks. Flink provides mechanisms for scaling operators and applications by taking a savepoint from a running job and restarting it with a different parallelism from the savepoint.

Flink distinguishes between two types of state, *operator state* and *keyed state*. Operator state maintains state that is scoped to the parallel task of an operator. Operator state needs to be splittable and mergeable to support operator rescaling. Keyed state is scoped to a key of the data and requires a keyed stream, i.e., a stream partitioned on a key attribute. When processing a record, an operator with keyed state can only access the state that corresponds to the key of the current record. Operators with keyed state are rescaled by reassigning key ranges and redistributing the corresponding state to fewer or more operator tasks.

Flink maintains the state of operators in a so-called state backend. A state backend implements the internal data structures to hold the data and logic to checkpoint the state. Heap-based state backends use regular Java data structures to store the state. They are limited by the size of the JVM heap of the TaskManagers. Heap-based state backends can checkpoint to the JVM heap of the JobManager or to a remotely accessible file system, such as HDFS, NFS, or S3. The RocksDB-based state backend writes state to disk and therefore allows for state that exceeds the size of available memory. Flink exploits

many RocksDB features for checkpointing to a remote file system, and maintaining state that becomes very large (on the order of multiple terabytes). When using the RocksDB state backend, Flink can not only create checkpoints asynchronously but also incrementally which speeds up the checkpointing operation.

Key Applications

In the past, analytical data was commonly dumped into database systems or distributed file systems and, if at all, processed in nightly or monthly batch jobs. Nowadays, in a globalized, faster-moving world, reacting to events quickly is crucial for economic success. For Flink, processing of unbounded data means handling events when they occur in a non-approximated but accurate fashion. This requires support for event-time processing, exactly-once state consistency, and fault tolerance capabilities. Applications must be scalable to handle both current and increasing data volumes in the future, leading to a parallelized and distributed execution. Moreover, Flink applications can consume data from a variety of sources such as publish-subscribe message queues, file systems, databases, and sockets.

However, the use cases for Flink are not limited to faster data analytics. Its flexibility to model arbitrary dataflows and its precise control over state and time allow a variety of new applications. So-called event-driven applications are becoming more and more popular as part of an overall *event-driven architecture*. Such applications trigger computations based on incoming events and might store an accumulated history of events to relate it to newly arrived events. Applications can output their result to a database or key-value store, trigger an immediate reaction via an RPC call, or emit a new event which might trigger subsequent actions. In contrast to traditional two-tier architectures where application and database system separate business logic and data from each other, event-driven applications own computation and state and keep them close together. The benefits of this approach are that (1) state access is always local instead of accessing a remote database, (2) state and computation are scaled together, and (3) applications define the schema of their own state, similar to microservices.

Entire social networks have been built with Flink (Koliopoulos 2017) following the event-driven approach. Incoming user requests are written to a distributed log that acts as the single source of truth. The log is consumed by multiple distributed stateful applications which build up their state independently from each other for aggregated post views, like counts, and statistics. Finally, the updated results are stored in a materialized view that web servers can return to the user.

In addition, Flink offers a domain-specific API for complex event processing to detect and react on user-defined patterns in event streams. CEP is useful for fraud or intrusion detection scenarios or to monitor and validate business processes. Furthermore, Flink provides relational APIs to unify queries on bounded and unbounded data streams. With its Table and SQL APIs, Flink can continuously update a result table which is defined by query on one or more input streams, similar to a materialized view as known from relational database systems.

Cross-References

- Apache Spark
- Apache Apex
- Apache Samza
- Continuous Queries
- Definition of Data Streams
- Introduction to Stream Processing Algorithms
- Stream Window Aggregation Semantics and Optimization
- Streaming Microservices

References

Akidau T et al (2015) The dataflow model: a practical approach to balancing correctness, latency, and cost in

massive-scale, unbounded, out-of-order data processing. Proc VLDB Endowment 8(12):1792–1803
Alexandrov A, Ewen S, Heimel M, Hueske F, Kao O, Markl V, ..., Warneke D (2011) MapReduce and PACT-comparing data parallel programming models. In BTW, pp 25–44
Alexandrov A, Bergmann R, Ewen S, Freytag JC, Hueske F, Heise A, ..., Naumann F (2014) The stratosphere platform for big data analytics. VLDB J 23(6):939–964
Battré D, Ewen S, Hueske F, Kao O, Markl V, Warneke D (2010) Nephele/PACTs: a programming model and execution framework for web-scale analytical processing. In: Proceedings of the 1st ACM symposium on cloud computing. ACM, pp 119–130
Carbone P, Katsifodimos A, Ewen S, Markl V, Haridi S, Tzoumas K (2015a) Apache Flink: stream and batch processing in a single engine. In: Bulletin of the IEEE Computer Society Technical Committee on Data Engineering, vol 36, no. 4
Carbone P et al (2015b) Apache Flink: stream and batch processing in a single engine. In: Bulletin of the IEEE Computer Society Technical Committee on Data Engineering, vol 36, no. 4
Carbone P et al (2015c) Lightweight asynchronous snapshots for distributed dataflows. In CoRR abs/1506.08603. http://arxiv.org/abs/1506.08603
Carbone P, Ewen S, Fóra G, Haridi S, Richter S, Tzoumas K (2017) State management in apache flink®: consistent stateful distributed stream processing. Proc VLDB Endowment 10(12):1718–1729
Dean J, Ghemawat S (2008) MapReduce: simplified data processing on large clusters. Commun ACM 51(1):107–113
Ewen S, Tzoumas K, Kaufmann M, Markl V (2012) Spinning fast iterative data flows. Proc VLDB Endowment 5(11):1268–1279
Ghemawat S, Gobioff H, Leung ST (2003) The google file system. ACM SIGOPS Oper Syst Rev 37(5):29–43. ACM
Hueske F, Peters M, Sax MJ, Rheinländer A, Bergmann R, Krettek A, Tzoumas K (2012) Opening the black boxes in data flow optimization. Proc VLDB Endowment 5(11):1256–1267
Koliopoulos A (2017) Drivetribe's modern take on CQRS with Apache Flink. Drivetribe. https://data-artisans.com/blog/drivetribe-cqrs-apache-flink. Visited on 7 Sept 2017
Mani Chandy K, Lamport L (1985) Distributed snapshots: determining global states of distributed systems. ACM Trans Comp Syst (TOCS) 3(1):63–75
The Apache Software Foundation. RocksDB|A persistent key-value store|RocksDB. http://rocksdb.org/. Visited on 30 Sept 2017

Recommended Reading

Friedman E, Tzoumas K (2016) Introduction to Apache Flink: stream processing for real time and beyond. O'Reilly Media, Sebastopol. ISBN 1491976586
Hueske F, Kalavri V (2018) Stream processing with Apache Flink: fundamentals, implementation, and operation of streaming applications. O'Reilly Media, Sebastopol. ISBN 149197429X

Apache Hadoop

▶ Architectures

Apache Kafka

Matthias J. Sax
Confluent Inc., Palo Alto, CA, USA

Definitions

Apache Kafka (Apache Software Foundation 2017b; Kreps et al. 2011; Goodhope et al. 2012; Wang et al. 2015; Kleppmann and Kreps 2015) is a scalable, fault-tolerant, and highly available distributed streaming platform that can be used to store and process data streams.

Kafka consists of three main components:

- the Kafka cluster,
- the Connect framework (Connect API),
- and the Streams programming library (Streams API).

The Kafka cluster stores data streams, which are sequences of messages/events continuously produced by applications and sequentially and incrementally consumed by other applications. The Connect API is used to ingest data into Kafka and export data streams to external systems like distributed file systems, databases, and others. For data stream processing, the Streams API allows developers to specify sophisticated stream processing pipelines that read input streams from the Kafka cluster and write results back to Kafka.

Kafka supports many different use cases categories such as traditional publish-subscribe

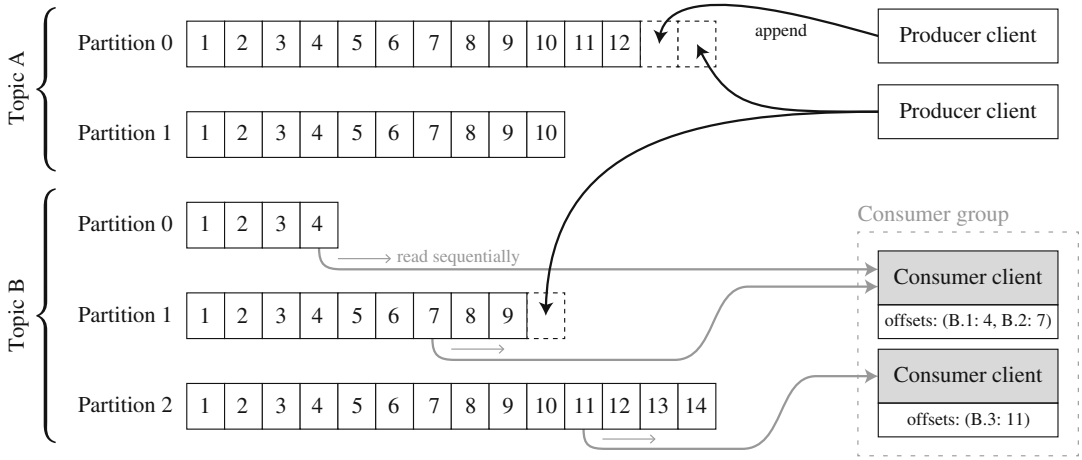
messaging, streaming ETL, and data stream processing.

Overview

A Kafka cluster provides a publish-subscribe messaging service (Fig. 1). Producer clients (publishers) write messages into Kafka, and consumer clients (subscribers) read those messages from Kafka. *Messages* are stored in Kafka servers called *brokers* and organized in named *topics*. A topic is an append-only sequence of messages, also called a log. Thus, each time a message is written to a topic, it is appended to the end of the log. A Kafka message is a key-value pair where key and value are variable-length byte arrays. Additionally, each message has a time stamp that is stored as 64-bit integer.

Topics are divided into *partitions*. When a message is written to a topic, the producer must specify the partition for the message. Producers can use any partitioning strategy for the messages they write. By default, messages are hash-partitioned by key, and thus all messages with the same key are written to the same partition. Each message has an associated *offset* that is the message's position within the partition, i.e., a monotonically increasing sequence number. The message's offset is implicitly determined by the order in which messages are appended to a partition. Hence, each message within a topic is uniquely identified by its partition and offset. Kafka guarantees strict message ordering within a single partition, i.e., it guarantees that all consumers reading a partition receive all messages in the exact same order as they were appended to the partition. There is no ordering guarantee between messages in different partitions or different topics.

Topics can be written by multiple producers at the same time. If multiple producers write to the same partition, their messages are interleaved. If consumers read from the same topic, they can form a so-called consumer group. Within a consumer group, each individual consumer reads data from a subset of partitions. Kafka ensures that each partition is assigned to exactly one consumer within a group. Different consumer groups or consumers that don't belong to any consumer group are independent from each other. Thus, if two consumer groups read the same topic, all messages are delivered to both groups. Because consumers are independent from each other, each consumer can read messages at its own pace. This results in decoupling—a desirable property for a distributed system—and makes the system robust against stragglers. In summary, Kafka supports



Apache Kafka, Fig. 1 Kafka topics are divided into partitions that are ordered sequences of messages. Multiple producers can write simultaneously into the same topic. Consumers track their read progress and can form a consumer group to share the read workload over all consumers within the group (Source: Kleppmann and Kreps 2015)

multiple producers and can deliver the same data to multiple consumers.

Kafka Brokers

Kafka brokers store messages reliably on disk. In contrast to traditional messaging/publish-subscribe systems, Kafka can be used for long-term storage of messages, because Kafka does not delete messages after delivery. Topics are configured with a so-called *retention time* that specifies how long a message should be stored. Topic retention can also be specified in bytes instead of time, to apply an upper bound on disk space. If the retention boundaries are reached, Kafka truncates partitions at the end of the log.

From a semantic point of view, messages are immutable facts, and thus it is not reasonable to support deleting individual messages from a topic. Users can only apply a "time-to-live" via topic retention to truncate old data.

Log Compaction

Kafka also supports so-called *compacted* topics. If a topic is configured for log compaction, users apply different *semantics* to the stored messages. While regular topics store *immutable facts*, a compacted topic can be used to store *updates*. Note that a compacted topic is still an append-only sequence of messages, and there are no in-place updates. Appending an update message to a compacted topic implies that a newer messages "replaces" older messages with the same key. The difference of compacted topics to topics with log retention is that Kafka guarantees that the latest update of a key is never deleted, while older updates can be garbage collected.

Log compaction is applied on a per-partition basis; thus updates for the same key should be written to the same partition. For performance reasons, brokers don't delete older messages immediately, but compaction is triggered as background process in regular intervals. In contrast to "regular" topics, compacted topics also support delete semantics for individual record via so-called *tombstone* messages. A tombstone is a message with a `null` value, and it indicates that all previous updates for the corresponding key can be deleted (including the tombstone itself).

Scaling and Load Balancing

As described in section "Overview," messages are stored in topics, and topics are divided into partitions. Partitions allow brokers to scale out horizontally and to balance load within the cluster, because partitions are independent units within a topic. Even if partitions of the same topic are stored at different brokers, there is no need for broker synchronization, and thus a Kafka cluster scales linearly with the number of brokers. A single broker only limits the capacity of a single partition, but because topics can be created with an arbitrary number of partitions, this is not a limitation in practice. Overall the read/write throughput and storage requirements of topics are not limited by the size of a single server, and the cluster capacity can be increased by adding new brokers to the system. Last but not least, partitions can be reassigned from one broker to another to balance load within a cluster.

There is no master node in a Kafka cluster: all brokers are able to perform all services provided by the cluster. This design supports linear scale-out, as a master node could become a bottleneck. For broker coordination, Kafka uses Apache ZooKeeper (Apache Software Foundation 2017d; Hunt et al. 2010), a scalable, fault-tolerant, and highly available distributed coordination service. Kafka uses ZooKeeper to store all cluster metadata about topics, partitions, partition-to-broker mapping, etc., in a reliable and highly available manner.

Fault Tolerance and High Availability

To ensure fault tolerance and high availability, partitions can be replicated to multiple brokers. Each topic can be configured with an individual replication factor that indicates how many copies of a partition should be maintained. To ensure strict message ordering guarantees per partitions, replication uses a *leader-follower* pattern. Each partition has a single leader and a configurable number of followers. All read and write requests are handled by the leader, while the followers

replicate all writes to the leader in the background. If the broker hosting the leader fails, Kafka initiates a leader election via ZooKeeper, and one of the followers becomes the new leader. All clients will be updated with the new leader information and send all their read/write request to the new leader. If the failed broker recovers, it will rejoin the cluster, and all hosted partitions become followers.

Message Delivery

Reading data from Kafka works somewhat differently compared to traditional messaging/publish-subscribe systems. As mentioned in section "Kafka Brokers," brokers do not delete messages after delivery. This design decision has multiple advantages:

- Brokers do not need to track the reading progress of consumers. This allows for an increased read throughput, as there is no progress tracking overhead for the brokers.
- It allows for in-order message delivery. If brokers track read progress, consumers need to acknowledge which messages they have processed successfully. This is usually done on a per-message basis. Thus, if an earlier message is not processed successfully, but a later message is processed successfully, redelivery of the first message happens out of order.
- Because brokers don't track progress and don't delete data after delivery, consumers can go back in time and reprocess old data again. Also, newly created consumers can retrieve older data.

The disadvantage of this approach is that consumer clients need to track their progress themselves. This happens by storing the offset of the next message a consumer wants to read. This offset is included in the read request to the broker, and the broker will deliver consecutive messages starting at the requested offset to the consumer. After processing all received messages, the consumer updates its offset accordingly and sends the next read request to the cluster.

If a consumer is stopped and restarted later on, it usually should continue reading where it left off. To this end, the consumer is responsible for storing its offset reliably so it can retrieve it on restart. In Kafka, consumers can commit their offsets to the brokers. On offset commit brokers store consumer offsets reliably in a special topic called *offset topic*. As offset topic is also partitioned and replicated and is thus scalable, fault-tolerant, and highly available. This allows Kafka to manage a large number of consumers at the same time. The offset topic is configured with log compaction enabled (cf. section "Log Compaction") to guarantee that offsets are never lost.

Delivery Semantics

Kafka supports multiple delivery semantics, namely, *at-most-once*, *at-least-once*, and *exactly once*. What semantics a user gets depends on multiple factors like cluster/topic configuration as well as client configuration and user code.

It is important to distinguish between the write and read path when discussing delivery semantics. Furthermore, there is the concept of end-to-end processing semantics that applies to Kafka's Streams API. In this section, we will only cover the read and write path and refer to section "Kafka Streams" for end-to-end processing semantics.

Writing to Kafka

When a producer writes data into a topic, brokers acknowledge a successful write to the producer. If a producer doesn't receive an acknowledgement, it can ignore this and follow at-most-once semantics, as the message might not have been written to the topic and thus could be lost. Alternatively, a producer can retry the write resulting in at-least-once semantics. The first write could have been successful, but the acknowledgement might be lost. Kafka also supports exactly once writes by exploiting idempotence. For this, each message is internally assigned a unique identifier. The producer attaches this identifier to each message it writes, and the broker stores the identifier as metadata of each message in the topic. In case of a producer write retry, the broker can detect the

duplicate write by comparing the message identifiers. This deduplication mechanism is internal to producer and broker and does not guard against application-level duplicates.

Atomic Multi-partition Writes Kafka also support atomic multi-partition writes that are called *transactions*. A Kafka transaction is different to a database transaction, and there is no notion of ACID guarantees. A transaction in Kafka is similar to an atomic write of multiple messages that can span different topics and/or partitions. Due to space limitations, we cannot cover the details of transactional writes and can only give a brief overview. A transactional producer is first initialized for transactions by performing a corresponding API call. The broker is now ready to accept transactional writes for this producer. All messages sent by the producer belong to the current transaction and won't be delivered to any consumer as long as the transaction is not completed. Messages within a transaction can be sent to any topic/partition within the cluster. When all messages of a transaction have been sent, the producer commits the transaction. The broker implements a two-phase commit protocol to commit a transaction, and it either successfully "writes" all or none of the messages belonging to a transaction. A producer can also abort a transaction; in this case, none of the messages will be deleted from the log, but all messages will be marked as aborted.

Reading from Kafka

As discussed in section "Message Delivery," consumers need to track their read progress themselves. For fault-tolerance reasons, consumers commit their offsets regularly to Kafka. Consumers can apply two strategies for this: after receiving a message, they can either first commit the offset and process the message afterwards, or they do it in reverse order and first process the message and commit the offset at the end. The commit-first strategy provides at-most-once semantics. If a message is received and the offset is committed before the message is processed, this message would not be redelivered in case of failure: after a failure, the consumer would recover its offsets as the last committed offset and thus would resume reading after the failed message.

In contrast, the process-first strategy provides at-least-once semantics. Because the consumer doesn't update its offsets after processing the receive message successfully, it would always fall back to the old offset after recovering from an error. Thus, it would reread the processed message, resulting in potential duplicates in the output.

Transactional Consumers Consumers can be configured to read all (including aborted messages) or only committed messages (cf. paragraph *Atomic multi-partition writes* in section "Writing to Kafka"). This corresponds to a read uncommitted and read committed mode similar to other transactional systems. Note that aborted messages are not deleted from the topics and will be delivered to all consumers. Thus, consumers in read committed mode will filter/drop aborted messages and not deliver them to the application.

Kafka Connect Framework

Kafka Connect—or the Connect API—is a framework for integrating Kafka with external systems like distributed file systems, databases, key-value stores, and others. Internally, Kafka Connect uses producer/consumer clients as described in section "Kafka Brokers," but the framework implements much of the functionality and best practices that would otherwise have to be implemented for each system. It also allows running in a fully managed, fault-tolerant, and highly available manner.

The Connect API uses a *connector* to communicate with each type of external system. A *source connector* continuously reads data from an external source system and writes the records into Kafka, while a *sink connector* continuously consumes data from Kafka and sends the records to the external system. Many connectors are available for a wide variety of systems, including HDFS, S3, relational databases, document database systems, other messaging systems, file

systems, metric systems, analytic systems, and so on. Developers can create connectors for other systems.

Kafka Connect can be either used as standalone or deployed to a cluster of machines. To connect with an external system, a user creates a configuration for a connector that defines the specifics of the external system and the desired behavior, and the user uploads this configuration to one of the Connect workers. The worker, which is running in a JVM, deploys the connector and distributes the connector's *tasks* across the cluster. Source connector tasks load data from the external system and generate records, which Connect then writes to the corresponding Kafka topics. For sink connector tasks, Connect consumes the specified topics and passes these records to the task, which then is responsible for sending the records to the external system.

Single-Message Transforms

The Connect API allows to specify simple transformation—so-called single-message transforms, SMT—for individual messages that are imported/exported into/from Kafka. Those transformations are independent of the connector and allow for stateless operations. Standard transformation functions are already provided by Kafka, but it is also possible to implement custom transformations to perform initial data cleaning. If SMTs are not sufficient because a more complex transformation is required, the Streams API (described in the next section) can be used instead.

Kafka Streams

Kafka Streams—or the Streams API—is the stream processing library of Apache Kafka. It provides a high-level DSL that supports stateless as well as stateful stream processing operators like joins, aggregations, and windowing. The Streams API also supports exactly once processing guarantees and event-time semantics and handles out-of-order and late-arriving data. Additionally, the Streams API introduces *tables* as a first-class abstraction next to data *streams*. It shares a few ideas with Apache Samza (cf. article on "▸ Apache Samza") (Apache Software Foundation 2017c; Noghabi et al. 2017; Kleppmann and Kreps 2015) such as building on top of Kafka's primitives for fault tolerance and scaling. However, there are many notable differences to Samza: for example, Kafka Streams is implemented as a library and can run in any environment, unlike Samza, which is coupled to Apache Hadoop's resource manager YARN (Apache Software Foundation 2017a; Vavilapalli et al. 2013); it also provides stronger processing guarantees and supports both streams and tables as core data abstractions.

Streams and Tables

Most stream processing frameworks provide the abstraction of a *record stream* that is an append-only sequence of immutable facts. Kafka Streams also introduces the notion of a *changelog* stream, a mutable collection of data items. A changelog stream can also be described as a continuously updating table. The analogy between a changelog and a table is called the *stream-table duality* (Kleppmann 2016, 2017). Supporting tables as first-class citizens allow to enrich data streams via stream-table joints or populate tables as self-updating caches for an application.

The concept of changelogs aligns with the concept of compacted topics (cf. section "Log Compaction"). A changelog can be stored in a compacted topic, and the corresponding table can be recreated by reading the compacted topic without data loss as guaranteed by the compaction contract.

Kafka Streams also uses the idea of a changelog stream for (windowed) aggregations. An aggregation of a record stream yields both a table and a changelog stream as a result—not a record stream. Thus, the table always contains the current aggregation result that is updated for each incoming record. Furthermore, each update to the result table is propagated downstream as a changelog record that is appended to the changelog stream.

State Management

Operator state is a first-class citizen in Kafka's Streams API similar to Samza and uses the aforementioned table abstraction. For high performance, state is kept local to the stream processing operators using a RocksDB (Facebook Inc. 2017) store. Local state is not fault-tolerant, and thus state is additionally backed by a topic in the Kafka cluster. Those topics have log compaction (cf. section "Log Compaction") enabled and are called *changelog topics*. Using log compaction ensures that the size of the changelog topic is linear in the size of the state. Each update to the store is written to the changelog topic; thus, the persistent changelog topic is the source of truth, while the local RocksDB store is an ephemeral materialized view of the state.

If an application instance fails, another instance can recreate the state by reading the changelog topic. For fast fail-over, Kafka Streams also support *standby replicas*, the hold hot standbys of state store. Standby replicas can be maintained by continuously reading all changes to the primary store from the underlying changelog topic.

Fault Tolerance and Scaling

Kafka Streams uses the same scaling/parallelism abstraction as Samza, namely, partitions. This is a natural choice as Kafka Streams reads input data from partitioned topics. Each input topic partition is mapped to a *task* that processed the records of this partition. Tasks are independent units of parallelism and thus can be executed by different threads that might run on different machines. This allows to scale out a Kafka Streams application by starting multiple instances on different machines. All application instances form a consumer group, and thus Kafka assigns topic partitions in a load balanced manner to all application instances (cf. section "Kafka Brokers").

Because Kafka Streams is a library, it cannot rely on automatic application restarts of failed instances. Hence, it relies on the Kafka cluster to detect failures. As mentioned above, a Streams application forms a consumer group, and the Kafka cluster monitors the liveness of all members of the group. In case of a failure, the cluster detects a dead group member and reassigns the corresponding input topic partitions of the failed application instance to the remaining instances. This process is called an consumer group *rebalance*. During a rebalance, Kafka Streams also ensures that operator state is migrated from the failing instance (cf. section "State Management"). Kafka's consumer group management mechanism also allows for fully elastic deployments. Application instances can be added or removed during runtime without any downtime: the cluster detects joining/leaving instances and rebalances the consumer group automatically. If new instances are joining, partitions are *revoked* from existing members of the groups and assigned to the new members to achieve load balancing. If members are leaving a consumer group, this is just a special case of fail-over, and the partitions of the leaving instance are assigned to the remaining ones. Note that in contrast to a fail-over rebalance, a scaling rebalance guarantees a clean hand over of partitions, and thus each record is processed exactly once.

Time Semantics

Time is a core concept in stream processing, and the operators in Kafka's stream processing DSL are based on time: for example, windowed aggregations require record time stamps to assign records to the correct time windows. Handling time also requires to handle late-arriving data, as record might be written to a topic out of order (note: Kafka guarantees offset based in-order delivery; there is no time stamp-based delivery guarantee).

Stream processing with Kafka supports three different time semantics: event time, ingestion time, and processing time. From a Streams API point of view, there are only two different semantics though: *event time* and *processing time*.

Processing time semantics are provided when there is no record time stamp, and thus Streams needs to use wall clock time when processing data for any time-based operation like windowing. Processing time has the disadvantage that there is no relationship between the time when data was created and when data gets processed.

Furthermore, processing time semantics is inherently non-deterministic.

Event time semantics are provided when the record contains a time stamp in its payload or metadata. This time-stamp is assigned when a record is created and thus allows for deterministic data reprocessing. Applications may include a time stamp in the payload of the message, but the processing of such time stamps is then application-dependent. For this reason, Kafka supports record metadata time stamps that are stored in topics and automatically set by the producer when a new record is created.

Additionally, Kafka topics can be configured to support *ingestion time* for time-based operations: ingestion time is the time when data is appended to the topic, i.e., the current broker wall clock time on write. Ingestion time is an approximation of event time, assuming that data is written to a topic shortly after it was created. It can be used if producer applications don't provide a record metadata time stamp. As ingestion time is an approximation of event time, the Streams API is agnostic to ingestion time (it is treated as a special case of event time).

Exactly Once Processing Semantics

Processing an input record can be divided into three parts: first, the actual processing including any state updates; second, writing the result records to the output topics; third, recording the progress by committing the consumer offset. To provide exactly once processing semantics, all three parts must be performed "all or nothing."

As described in section "State Management," updating operator state is actually a write to a changelog topic that is the source of truth (the local state is only a materialized view). Furthermore, committing input offsets is a write to the special *offset topic* as discussed in section "Message Delivery." Hence, all three parts use the same underlying operation: writing to a topic.

This allows the Streams API to leverage Kafka's transactions (cf. section "Writing to Kafka") to provide end-to-end exactly once stream processing semantics. All writes that happen during the processing of a record are part of the same transaction. As a transaction is an atomic multi-partition write, it ensures that either all writes are committed or all changes are aborted together.

Summary

Apache Kafka is a scalable, fault-tolerant, and highly available distributed streaming platform. It allows fact and changelog streams to be stored and processed, and it exploits the stream-table duality in stream processing. Kafka's transactions allow for exactly once stream processing semantics and simplify exactly once end-to-end data pipelines. Furthermore, Kafka can be connected to other systems via its Connect API and can thus be used as the central data hub in an organization.

Cross-References

- Apache Flink
- Apache Samza
- Continuous Queries

References

Apache Software Foundation (2017a) Apache Hadoop project web page. https://hadoop.apache.org/

Apache Software Foundation (2017b) Apache Kafka project web page. https://kafka.apache.org/

Apache Software Foundation (2017c) Apache Samza project web page. https://samza.apache.org/

Apache Software Foundation (2017d) Apache ZooKeeper project web page. https://zookeeper.apache.org/

Facebook Inc (2017) RocksDB project web page. http://rocksdb.org/

Goodhope K, Koshy J, Kreps J, Narkhede N, Park R, Rao J, Ye VY (2012) Building Linkedin's real-time activity data pipeline. IEEE Data Eng Bull 35(2):33–45. http://sites.computer.org/debull/A12june/pipeline.pdf

Hunt P, Konar M, Junqueira FP, Reed B (2010) ZooKeeper: wait-free coordination for internet-scale systems. In: Proceedings of the 2010 USENIX conference on USENIX annual technical conference, USENIX ATC'10. USENIX Association, Berkeley, p 11. http://dl.acm.org/citation.cfm?id=1855840.1855851

Kleppmann M (2016) Making sense of stream processing, 1st edn. O'Reilly Media Inc., 183 pages

Kleppmann M (2017) Designing data-intensive applications. O'Reilly Media Inc., Sebastopol
Kleppmann M, Kreps J (2015) Kafka, Samza and the Unix philosophy of distributed data. IEEE Data Eng Bull 38(4):4–14. http://sites.computer.org/debull/A15dec/p4.pdf
Kreps J, Narkhede N, Rao J (2011) Kafka: a distributed messaging system for log processing. In: Proceedings of the NetDB, pp 1–7
Noghabi SA, Paramasivam K, Pan Y, Ramesh N, Bringhurst J, Gupta I, Campbell RH (2017) Samza: stateful scalable stream processing at LinkedIn. Proc VLDB Endow 10(12):1634–1645. https://doi.org/10.14778/3137765.3137770
Vavilapalli VK, Murthy AC, Douglas C, Agarwal S, Konar M, Evans R, Graves T, Lowe J, Shah H, Seth S, Saha B, Curino C, O'Malley O, Radia S, Reed B, Baldeschwieler E (2013) Apache Hadoop YARN: yet another resource negotiator. In: 4th ACM symposium on cloud computing (SoCC). https://doi.org/10.1145/2523616.2523633
Wang G, Koshy J, Subramanian S, Paramasivam K, Zadeh M, Narkhede N, Rao J, Kreps J, Stein J (2015) Building a replicated logging system with Apache Kafka. PVLDB 8(12):1654–1655. http://www.vldb.org/pvldb/vol8/p1654-wang.pdf

Apache Mahout

Andrew Musselman
Apache Software Foundation, Seattle, WA, USA

Definitions

Apache Mahout (http://mahout.apache.org) is a distributed linear algebra framework that includes a mathematically expressive domain-specific language (DSL). It is designed to aid mathematicians, statisticians, and data scientists to quickly implement numerical algorithms while focusing on the mathematical concepts in their work, rather than on code syntax. Mahout uses an extensible plug-in interface to systems such as Apache Spark and Apache Flink.

Historical Background

Mahout was founded as a sub-project of Apache Lucene in late 2007 and was promoted to a top-level Apache Software Foundation (ASF) (ASF 2017) project in 2010 (Khudairi 2010). The goal of the project from the outset has been to provide a machine learning framework that was both accessible to practitioners and able to perform sophisticated numerical computation on large data sets.

Mahout has undergone two major stages of architecture design. The first versions relied on the Apache Hadoop MapReduce framework, a popular tool for orchestrating large-scale data flow and computation. Hadoop, while flexible enough to handle many typical workflows, has severe limitations when applied to highly iterative processes, like those used in most machine learning methods.

The Hadoop implementation of MapReduce does not allow for caching of intermediate results across steps of a long computation. This means that algorithms that iteratively use the same data many times are at a severe disadvantage – the framework must read data from disk every iteration rather than hold it in memory. This type of algorithm is common in data science and machine learning: examples include *k*-means clustering (which has to compute distances for all points in each iteration) and stochastic gradient descent (which has to compute gradients for the same points over many iterations).

Since enabling iterative work on large data sets is a core requirement of a machine learning library geared toward big data, Mahout moved away from Hadoop in its second design phase. Starting with release 0.10.0 (PMC 2015), Mahout switched its computation engine to Apache Spark, a framework designed specifically to facilitate distributed in-memory computation. At the same time, Mahout jobs built using Hadoop MapReduce were deprecated to discourage users from relying on them long-term, and a new interface on the front end was released, named "Samsara," after the Hindu and Buddhist concept of rebirth. This new front end has a simple syntax for matrix math modeled after other systems such as MATLAB and R, to allow maximal readability and quick prototyping. Samsara can be run in an interactive shell where results are displayed inline, enabling a live back-and-forth with code and

results, much like an interactive SQL or other interpreted environment familiar to many users.

As an example of the Samsara DSL syntax for calculating, for instance, the transpose of a matrix A multiplied with A itself, a common operation in machine learning jobs, can be written in Scala code as

```
val C = A.t %*% A.
```

Note the declaration of the resulting matrix `C`, the transpose operator `t`, and the matrix multiplication operator `%*%`. Behind the scenes, the right-hand side of the statement is parsed and the back end takes an optimal approach when physically performing the multiplication.

More recently, beginning in early 2017 with version 0.13.0 (PMC 2017), Mahout added the capability to perform computation directly on hardware, outside the Java Virtual Machine (JVM), using operation solvers written specifically to optimize performance according to native processor and memory architecture. This allows for speed gains from using all cores of all CPUs on the host and the option to leverage matrix-math-specific instruction sets built into graphics processing units (GPUs).

Foundations

The motivation for the Mahout project is to bring large-scale machine learning to practitioners in real-world environments. This boils down to three concepts: first, providing a convenient syntax for writing programs and doing experimental and exploratory work; second, handling all the nitty-gritty details of distributed matrix arithmetic without requiring the user to understand the low-level implementation; and last, making deployment to production simple and straightforward.

Many machine learning libraries and tools either provide an array of algorithms, but do not operate on very large data sets, or else manage computation at scale but require significant programming experience and skill. Bridging those gaps, Mahout works at scale, is flexible with regard to hardware and compute engines, and allows data science and machine learning practitioners to focus on the math in their work symbolically.

For example, in the distributed stochastic principal component analysis (dSPCA) job in Mahout, there is a computation that is expressed symbolically as

$$G = BB^T - C - C^T + \xi^T \xi s_q{}^T s_q .$$

This equation can be expressed in code in Mahout as

```
val G = B %*% B.t - C - C.t
  + (xi dot xi) * (s_q cross s_q),
```

which is readable and compact compared to many alternative representations.

The main reason many machine learning operations need to be distributed in real-world environments is because one of the key strengths of machine learning methods is their ability to build predictions from very large data sets. Data sets at this scale include, for example, HTTP access logs for high-traffic websites, large text corpora consisting of millions of documents, and streaming, high-volume data from sensors in industrial settings. With smaller data sets, the input is often held in memory, and all computation is performed on one host computer. However, as soon as a data set scales beyond the memory on a single host, inefficiency caused by spilling data to storage and re-reading it slows down operations and makes them too costly to be viable.

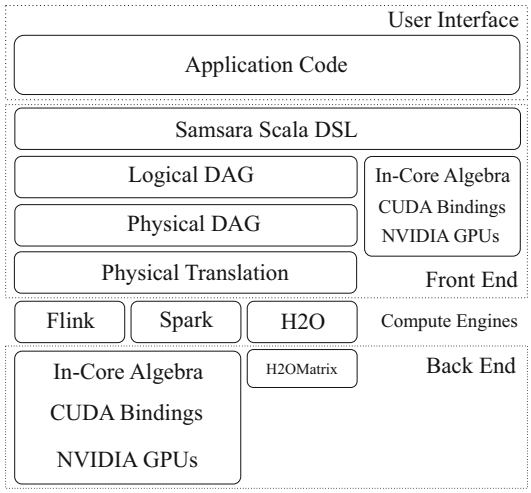
Distributed computing frameworks that allow computation on large data sets are continuously evolving based on shifting market requirements and ongoing research and development. Mahout takes advantage of many of the abstractions provided by these frameworks and then builds further performance improvements and refinements. Many refinements come from exploiting well-studied properties of matrices and matrix arithmetic and using "tricks" and shortcuts to avoid doing more operations than required,

and others come from solvers purpose-built for specific types of hardware and environments.

Overview of Architecture

The most recent design of Mahout allows for iterative rapid prototyping in the Samsara DSL which can be used directly in production with minimal changes. Computation on the back end is adaptable to varied configurations of compute engine and hardware. For example, an algorithm can be handed off to a Spark or Flink engine for computation, and any required matrix arithmetic can be computed in the JVM, on the CPU (using all available cores), or on compatible GPUs, depending on the shape and characteristics of the vectors and matrices being operated on. All these combinations will work with the same front-end code with minimal changes.

All the moving parts involved are seen in Fig. 1, moving from the top layer where application code is written to the front end which hands off computation to the appropriate engine or to native solvers (which perform the computations), the pluggable compute engine layer, and the back end which handles computation according to instructions from compute engines. Noteworthy for the Mahout user is the Scala DSL layer which allows for interactive, iterative development which can be captured directly into Scala packages and run as application code in production environments with minimal changes.



Apache Mahout, Fig. 1 Current architecture of Mahout, showing four main components: the user interface where application code is written, the front end which handles which subsystem will perform computation (which could be native and in-core), the pluggable compute engine layer, and the back end which takes instructions from a compute engine and performs mathematical operations

Also notable is the pluggable compute engine layer, allowing flexibility in deployment as well as future-proofing for when new compute engines become viable and beneficial to users' needs. In fact, code written for the Spark engine, for example, can be directly re-used on Flink with minor changes to import and initialization statements in the code.

Another recent introduction to Mahout is a template that simplifies algorithm contributions from the project's maintainers as well as from its users. The interface is modeled after the machine learning training and evaluation patterns found in many R packages and the Python-based scikit-learn package.

As seen in Fig. 2, a new algorithm need only define a `fit` method in a `Fitter` class, which populates a `Model` class, which contains parameter estimates, statistics about the fit, and an overall summary of performance, and which finally uses its predict method to make predictions on new data. The bulk of the work any new algorithm performs is written inside the `fit` method.

Key Applications

Practical applications of machine learning usually fall into one of three categories, the so-called Three Cs: collaborative filtering (known commonly as "recommender systems," or "recommenders"), classification, and clustering.

Recommender systems typically come into play in situations where the goal is to present new items to users which they are likely to need or want. These recommendations are based on historical user behavior across all users and across all items they interact with. Examples

Apache Mahout, Fig. 2 Code listing of a skeleton for a new algorithm, showing both the Fitter and Model classes, including their fit and predict methods

```
class Foo[K] extends RegressorFitter[K] {

  def fit(drmX: DrmLike[K],
          drmTarget: DrmLike[K],
          hyperparameters: (Symbol, Any)*): FooModel[K] = {
    /**
      * Normally this section would have more code
      *
      */
    var model = new FooModel[K]
    model.summary = "This model has been fit, etc."
    model
  }
}

class FooModel[K] extends RegressorModel[K] {

  def predict(drmPredictors: DrmLike[K]): DrmLike[K] = {
    drmPredictors.mapBlock(1) {
      case (keys, block: Matrix) => {
        var outputBlock = new DenseMatrix(block.nrow, 1)
        keys -> (outputBlock += 1.0)
      }
    }
  }
}
```

include online retail, where increasing customer spending and site interactions is a key business goal. A common use of a recommender is to present examples of products that other customers have bought, calculated by the similarity between other customers' purchases or between products themselves.

Classifiers cover a broad range of methods which can be summarized as predicting either a categorical value for a user or an item, such as "likely or unlikely to default on a bank loan," or a numerical value such as age or salary. These methods are used across most scientific fields and industrial sectors, for anything from screening for illnesses in medicine, to determining risk in the financial sector, to predictive plant maintenance in manufacturing operations.

Clustering methods are generally used to make sense of groups of users, documents, or other items. Presented as a whole, a data set can be daunting or even impossible to understand as a pile of records full of numbers and values, and it is often desirable to segment the whole set into smaller pieces, each of which collects most-similar items together for more close analysis and inspection. A common use for clustering is with a text corpus that contains documents on a variety of subjects. In order to categorize all the documents into one or more relevant topics which could be used for quicker filing and retrieval, the entire corpus can be analyzed by a clustering method which separates documents into groups, members of each being relevant to similar topics or concepts.

In real-world applications, the distinction between these methods can be less clear-cut. Often more than one type of machine learning method will be used in combination to achieve further nuance and improve relevance and effectiveness of predictions. For example, after a corpus of documents is organized into topics, if a new document is added to the corpus, it can effectively be "classified" based on which cluster it is closest to. Similarly, the input to a recommender system in production can be a customer's recent browsing history on the product website. The customer's history can be fed into a classifier to determine which cohort of customers they belong to. This information can then be the input to a recommender built specifically per cohort, so that any products recommended are more relevant to that customer.

Cross-References

- Apache Flink
- Apache Hadoop
- Apache Spark
- Apache SystemML
- Big Data and Recommendation
- Columnar Storage Formats
- GPU-Based Hardware Platforms
- Python
- Scala
- Scalable Architectures for Big Data Analysis

References

ASF (2017) Welcome to the apache software foundation! https://www.apache.org

Khudairi S (2010) The apache software foundation blog. https://blogs.apache.org/foundation/entry/the_apache_software_foundation_announces4

PMC AM (2015) Apache mahout 0.10.0 release notes. http://mahout.apache.org/release-notes/Apache-Mahout-0.10.0-Release-Notes.pdf

PMC AM (2017) Apache mahout 0.13.0 release notes. https://mail-archives.apache.org/mod_mbox/www-announce/201704.mbox/%3CCANg8BGBe+WwdZC6z6BAm3hqTOMjA2ma76y0dig0Jf5LHtg F56g@mail.gmail.com%3E

Apache Samza

Martin Kleppmann
University of Cambridge, Cambridge, UK

Definitions

Apache Samza is an open source framework for distributed processing of high-volume event streams. Its primary design goal is to support high throughput for a wide range of processing patterns, while providing operational robustness at the massive scale required by Internet companies. Samza achieves this goal through a small number of carefully designed abstractions: partitioned logs for messaging, fault-tolerant local state, and cluster-based task scheduling.

Overview

Stream processing is playing an increasingly important part of the data management needs of many organizations. Event streams can represent many kinds of data, for example, the activity of users on a website, the movement of goods or vehicles, or the writes of records to a database.

Stream processing jobs are long-running processes that continuously consume one or more event streams, invoking some application logic on every event, producing derived output streams, and potentially writing output to databases for subsequent querying. While a batch process or a database query typically reads the state of a dataset at one point in time, and then finishes, a stream processor is never finished: it continually awaits the arrival of new events, and it only shuts down when terminated by an administrator.

Many tasks can be naturally expressed as stream processing jobs, for example:

- aggregating occurrences of events, e.g., counting how many times a particular item has been viewed;
- computing the rate of certain events, e.g., for system diagnostics, reporting, and abuse prevention;
- enriching events with information from a database, e.g., extending user click events with information about the user who performed the action;
- joining related events, e.g., joining an event describing an email that was sent with any events describing the user clicking links in that email;
- updating caches, materialized views, and search indexes, e.g., maintaining an external full-text search index over text in a database;
- using machine learning systems to classify events, e.g., for spam filtering.

Apache Samza, an open source stream processing framework, can be used for any of the above applications (Kleppmann and Kreps 2015; Noghabi et al. 2017). It was originally developed at LinkedIn, then donated to the Apache Software Foundation in 2013, and became a top-level Apache project in 2015. Samza is now used in production at many Internet companies, including LinkedIn (Paramasivam 2016), Netflix (Netflix Technology Blog 2016), Uber (Chen 2016; Hermann and Del Balso 2017), and TripAdvisor (Calisi 2016).

Samza is designed for usage scenarios that require very high throughput: in some production settings, it processes millions of messages per second or trillions of events per day (Feng 2015; Paramasivam 2016; Noghabi et al. 2017). Consequently, the design of Samza prioritizes scalability and operational robustness above most other concerns.

The core of Samza consists of several fairly low-level abstractions, on top of which high-level operators have been built (Pathirage et al. 2016). However, the core abstractions have been carefully designed for operational robustness, and the scalability of Samza is directly attributable to the choice of these foundational abstractions. The remainder of this article provides further detail on those design decisions and their practical consequences.

Partitioned Log Processing

A Samza job consists of a set of Java Virtual Machine (JVM) instances, called *tasks*, that each processes a subset of the input data. The code running in each JVM comprises the Samza framework and user code that implements the required application-specific functionality. The primary API for user code is the Java interface `StreamTask`, which defines a method `process()`. Figure 1 shows two examples of user classes implementing the `StreamTask` interface.

Once a Samza job is deployed and initialized, the framework calls the `process()` method once for every message in any of the input streams. The execution of this method may have various effects, including querying or updating local state and sending messages to output streams. This model of computation is closely analogous to a map task in the well-known MapReduce programming model (Dean and Ghemawat 2004), with the difference that

```
class SplitWords implements StreamTask {

  static final SystemStream WORD_STREAM =
    new SystemStream("kafka", "words");

  public void process(
        IncomingMessageEnvelope in,
        MessageCollector out,
        TaskCoordinator _) {

    String str = (String) in.getMessage();

    for (String word : str.split(" ")) {
      out.send(
        new OutgoingMessageEnvelope(
          WORD_STREAM, word, 1));
    }
  }
}
```

```
class CountWords implements StreamTask,
                            InitableTask {

  private KeyValueStore<String, Integer> store;

  public void init(Config config,
                   TaskContext context) {
    store = (KeyValueStore<String, Integer>)
      context.getStore("word-counts");
  }

  public void process(
        IncomingMessageEnvelope in,
        MessageCollector out,
        TaskCoordinator _) {

    String word = (String) in.getKey();
    Integer inc = (Integer) in.getMessage();

    Integer count = store.get(word);
    if (count == null) count = 0;
    store.put(word, count + inc);
  }
}
```

Apache Samza, Fig. 1 The two operators of a streaming word-frequency counter using Samza's *StreamTask* API (Image source: Kleppmann and Kreps 2015, © 2015 IEEE, reused with permission)

a Samza job's input is typically never-ending (*unbounded*).

Similarly to MapReduce, each Samza task is a single-threaded process that iterates over a sequence of input records. The inputs to a Samza job are partitioned into disjoint subsets, and each input partition is assigned to exactly one processing task. More than one partition may be assigned to the same processing task, in which case the processing of those partitions is interleaved on the task thread. However, the number of partitions in the input determines the job's maximum degree of parallelism.

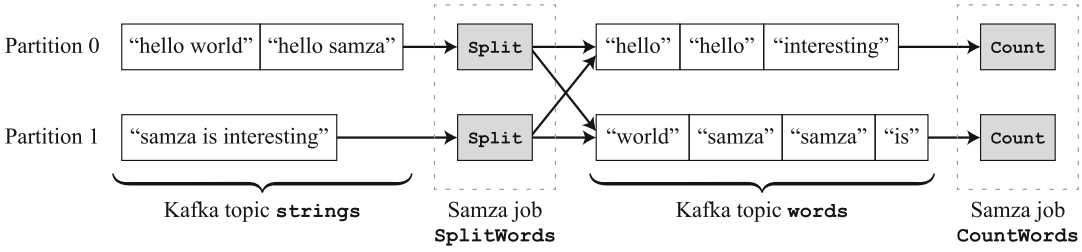
The log interface assumes that each partition of the input is a totally ordered sequence of records and that each record is associated with a monotonically increasing sequence number or identifier (known as *offset*). Since the records in each partition are read sequentially, a job can track its progress by periodically writing the offset of the last read record to durable storage. If a stream processing task is restarted, it resumes consuming the input from the last recorded offset.

Most commonly, Samza is used in conjunction with Apache Kafka (see separate article on Kafka). Kafka provides a partitioned, fault-tolerant log that allows publishers to append messages to a log partition and consumers (subscribers) to sequentially read the messages in a log partition (Wang et al. 2015; Kreps et al. 2011; Goodhope et al. 2012). Kafka also allows stream processing jobs to reprocess previously seen records by resetting the consumer offset to an earlier position, a fact that is useful during recovery from failures.

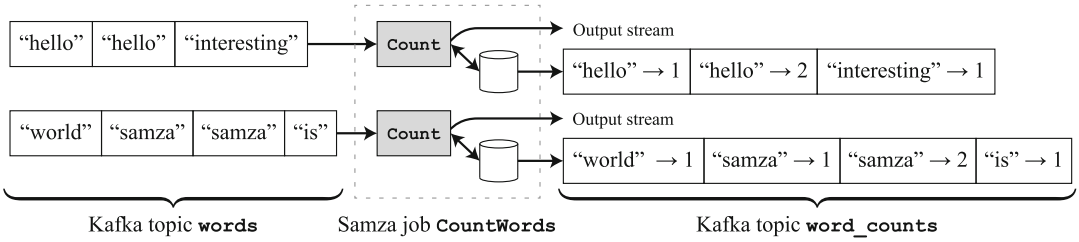
However, Samza's stream interface is pluggable: besides Kafka, it can use any storage or messaging system as input, provided that the system can adhere to the partitioned log interface. By default, Samza can also read files from the Hadoop Distributed Filesystem (HDFS) as input, in a way that parallels MapReduce jobs, at competitive performance (Noghabi et al. 2017). At LinkedIn, Samza is commonly deployed with *Databus* inputs: Databus is a change data capture technology that records the log of writes to a database and makes this log available for applications to consume (Das et al. 2012; Qiao et al. 2013). Processing the stream of writes to a database enables jobs to maintain external indexes or materialized views onto data in a database and is especially relevant in conjunction with Samza's support for local state (see section "Fault-Tolerant Local State") (Fig. 3).

While every partition of an input stream is assigned to one particular task of a Samza job, the output partitions are not bound to tasks. That is, when a `StreamTask` emits output messages, it can assign them to any partition of the output stream. This fact can be used to group related data items into the same partition: for example, in the word-counting application illustrated in Fig. 2, the `SplitWords` task chooses the output partition for each word based on a hash of the word. This ensures that when different tasks encounter occurrences of the same word, they are all written to the same output partition, from where a downstream job can read and aggregate the occurrences.

When stream tasks are composed into multi-stage processing pipelines, the output of one task



Apache Samza, Fig. 2 A Samza task consumes input from one partition, but can send output to any partition (Image source: Kleppmann and Kreps 2015, © 2015 IEEE, reused with permission)



Apache Samza, Fig. 3 A task's local state is made durable by emitting a changelog of key-value pairs to Kafka (Image source: Kleppmann and Kreps 2015, © 2015 IEEE, reused with permission)

becomes the input to another task. Unlike many other stream processing frameworks, Samza does not implement its own message transport layer to deliver messages between stream operators. Instead, Kafka is used for this purpose; since Kafka writes all messages to disk, it provides a large buffer between stages of the processing pipeline, limited only by the available disk space on the Kafka brokers.

Typically, Kafka is configured to retain several days or weeks worth of messages in each topic. Thus, if one stage of a processing pipeline fails or begins to run slow, Kafka can simply buffer the input to that stage while leaving ample time for the problem to be resolved. Unlike system designs based on backpressure, which require a producer to slow down if the consumer cannot keep up, the failure of one Samza job does not affect any upstream jobs that produce its inputs. This fact is crucial for the robust operation of large-scale systems, since it provides fault containment: as far as possible, a fault in one part of the system does not negatively impact other parts of the system.

Messages are dropped only if the failed or slow processing stage is not repaired within the retention period of the Kafka topic. In this case, dropping messages is desirable because it isolates the fault: the alternative – retaining messages indefinitely until the job is repaired – would lead to resource exhaustion (running out of memory or disk space), which would cause a cascading failure affecting unrelated parts of the system.

Thus, Samza's design of using Kafka's on-disk logs for message transport is a crucial factor in its scalability: in a large organization, it is often the case that an event stream produced by one team's job is consumed by one or more jobs that are administered by other teams. The jobs may be operating at different levels of maturity: for example, a stream produced by an important production job may be consumed by several unreliable experimental jobs. Using Kafka as a buffer between jobs ensures that adding an unreliable consumer does not negatively impact the more important jobs in the system.

Finally, an additional benefit of using Kafka for message transport is that every message stream in the system is accessible for debugging and monitoring: at any point, an additional consumer can be attached to inspect the message flow.

Fault-Tolerant Local State

Stateless stream processing, in which any message can be processed independently from any other message, is easy to implement and scale. However, many important applications require that stream processing tasks maintain state. For example:

- when performing a join between two streams, a task must maintain an index of messages seen on each input within some time window, in order to find messages matching the join condition when they arrive;
- when computing a rate (number of events per time interval) or aggregation (e.g., sum of a particular field), a task must maintain the current aggregate value and update it based on incoming events;

- when processing an event requires a database query to look up some related data (e.g., looking up a user record for the user who performed the action in the event), the database can also be regarded as stream processor state.

Many stream processing frameworks use transient state that is kept in memory in the processing task, for example, in a hash table. However, such state is lost when a task crashes or when a processing job is restarted (e.g., to deploy a new version). To make the state fault-tolerant, some frameworks such as Apache Flink periodically write checkpoints of the in-memory state to durable storage (Carbone et al. 2015); this approach is reasonable when the state is small, but it becomes expensive as the state grows (Noghabi et al. 2017).

Another approach, used, for example, by Apache Storm, is to use an external database or key-value store for any processor state that needs to be fault-tolerant. This approach carries a severe performance penalty: due to network latency, accessing a database on another node is orders of magnitude slower than accessing local in-process state (Noghabi et al. 2017). Moreover, a high-throughput stream processor can easily overwhelm the external database with queries; if the database is shared with other applications, such overload risks harming the performance of other applications to the point that they become unavailable (Kreps 2014).

In response to these problems, Samza pioneered an approach to managing state in a stream task that avoids the problems of both checkpointing and remote databases. Samza's approach to providing fault-tolerant local state has subsequently been adopted in the Kafka Streams framework (see article on Apache Kafka).

Samza allows each task to maintain state on the local disk of the processing node, with an in-memory cache for frequently accessed items. By default, Samza uses RocksDB, an embedded key-value store that is loaded into the JVM process of the stream task, but other storage engines can also be plugged in its place. In Fig. 1, the `CountWords` task accesses this managed state through the `KeyValueStore` interface. For workloads with good locality, Samza's RocksDB with cache provides performance close to in-memory stores; for random-access workloads on large state, it remains significantly faster than accessing a remote database (Noghabi et al. 2017).

If a job is cleanly shut down and restarted, for example, to deploy a new version, Samza's host affinity feature tries to launch each StreamTask instance on the machine that has the appropriate RocksDB store on its local disk (subject to available resources). Thus, in most cases the state survives task restart without any further action. However, in some cases – for example, if a processing node suffers a full system failure – the state on the local disk may be lost or rendered inaccessible.

In order to survive the loss of local disk storage, Samza again relies on Kafka. For each store containing state of a stream task, Samza creates a Kafka topic called a *changelog* that serves as a replication log for the store. Every write to the local RocksDB store is also encoded as a message and published to this topic, as illustrated in Fig. 3. These writes can be performed asynchronously in batches, enabling much greater throughput than synchronous random-access requests to a remote data store. The write queue needs to only be flushed when the offsets of input streams are written to durable storage, as described in the last section.

When a Samza task needs to recover its state after the loss of local storage, it reads all messages in the appropriate partition of the changelog topic and applies them to a new RocksDB store. When this process completes, the result is a new copy of the store that contains the same data as the store that was lost. Since Kafka replicates all data across multiple nodes, it is suitable for fault-tolerant durable storage of this changelog.

If a stream task repeatedly writes new values for the same key in its local storage, the changelog contains many redundant messages, since only the most recent value for a given key is required in order to restore local storage. To remove this redundancy, Samza uses a Kafka feature called *log compaction*

on the changelog topic. With log compaction enabled, Kafka runs a background process that searches for messages with the same key and discards all but the most recent of those messages. Thus, whenever a key in the store is overwritten with a new value, the old value is eventually removed from the changelog. However, any key that is not overwritten is retained indefinitely by Kafka. This compaction process, which is very similar to internal processes in log-structured storage engines, ensures that the storage cost and recovery time from a changelog corresponds to the size of the state, independently of the total number of messages ever sent to the changelog (Kleppmann 2017).

Cluster-Based Task Scheduling

When a new stream processing job is started, it must be allocated computing resources: CPU cores, RAM, disk space, and network bandwidth. Those resources may need to be adjusted from time to time as load varies and reclaimed when a job is shut down.

At large organizations, hundreds or thousands of jobs need to run concurrently. At such scale, it is not practical to manually assign resources: task scheduling and resource allocation must be automated. To maximize hardware utilization, many jobs and applications are deployed to a shared pool of machines, with each multi-core machine typically running a mixture of tasks from several different jobs.

This architecture requires infrastructure for managing resources and for deploying the code of processing jobs to the machines on which it is to be run. Some frameworks, such as Storm and Flink, have built-in mechanisms for resource management and deployment. However, frameworks that perform their own task scheduling and cluster management generally require a static assignment of computing resources – potentially even dedicated machines – before any jobs can be deployed to the cluster. This static resource allocation leads to inefficiencies in machine utilization and limits the ability to scale on demand (Kulkarni et al. 2015).

By contrast, Samza relies on existing cluster management software, which allows Samza jobs to share a pool of machines with non-Samza applications. Samza supports two modes of distributed operation:

- A job can be deployed to a cluster managed by Apache Hadoop YARN (Vavilapalli et al. 2013). YARN is a general-purpose resource scheduler and cluster manager that can run stream processors, MapReduce batch jobs, data analytics engines, and various other applications on a shared cluster. Samza jobs can be deployed to existing YARN clusters without requiring any special cluster-level configuration or resource allocation.
- Samza also supports a *stand-alone* mode in which a job's JVM instances are deployed and executed through some external process that is not under Samza's control. In this case, the instances use Apache ZooKeeper (Junqueira et al. 2011) to coordinate their work, such as assigning partitions of the input streams.

The stand-alone mode allows Samza to be integrated with an organization's existing deployment and cluster management tools or with cloud computing platforms: for example, Netflix runs Samza jobs directly as EC2 instances on Amazon Web Services (AWS), relying on the existing cloud facilities for resource allocation (Paramasivam 2016). Moreover, Samza's cluster management interface is pluggable, enabling further integrations with other technologies such as Mesos (Hindman et al. 2011).

With large deployments, an important concern is resource isolation, that is, ensuring that each process receives the resources it requested and that a misbehaving process cannot starve colocated processes of resources. When running in YARN, Samza supports the Linux cgroups feature to enforce limits on the CPU and memory use of stream processing tasks. In virtual machine environments such as EC2, resource isolation is enforced by the hypervisor.

Summary

Apache Samza is a stream processing framework that is designed to provide high throughput and operational robustness at very large scale. Efficient resource utilization requires a mixture of different jobs to share a multi-tenant computing infrastructure. In such an environment, the primary challenge in providing robust operation is fault isolation, that is, ensuring that a faulty process cannot disrupt correctly running processes and that a resource-intensive process cannot starve others.

Samza isolates stream processing jobs from each other in several different ways. By using Kafka's on-disk logs as a large buffer between producers and consumers of a stream, instead of backpressure, Samza ensures that a slow or failed consumer does not affect upstream jobs. By providing fault-tolerant local state as a common abstraction, Samza improves performance and avoids reliance on external databases that might be overloaded by high query volume. Finally, by integrating with YARN and other cluster managers, Samza builds upon existing resource scheduling and isolation technology that allows a cluster to be shared between many different applications without risking resource starvation.

Cross-References

- Apache Flink
- Apache Kafka
- Continuous Queries

References

Calisi L (2016) How to convert legacy Hadoop Map/Reduce ETL systems to Samza streaming. https://www.youtube.com/watch?v=KQ5OnL2hMBY

Carbone P, Katsifodimos A, Ewen S, Markl V, Haridi S, Tzoumas K (2015) Apache flink: stream and batch processing in a single engine. IEEE Data Eng Bull 38(4):28–38. http://sites.computer.org/debull/A15dec/p28.pdf

Chen S (2016) Scalable complex event processing on Samza @Uber. https://www.slideshare.net/ShuyiChen2/scalable-complex-event-processing-on-samza-uber

Das S, Botev C, Surlaker K, Ghosh B, Varadarajan B, Nagaraj S, Zhang D, Gao L, Westerman J, Ganti P, Shkolnik B, Topiwala S, Pachev A, Somasundaram N, Subramaniam S (2012) All aboard the Databus! LinkedIn's scalable consistent change data capture platform. In: 3rd ACM symposium on cloud computing (SoCC). https://doi.org/10.1145/2391229.2391247

Dean J, Ghemawat S (2004) MapReduce: simplified data processing on large clusters. In: 6th USENIX symposium on operating system design and implementation (OSDI)

Feng T (2015) Benchmarking apache Samza: 1.2 million messages per second on a single node. http://engineering.linkedin.com/performance/benchmarking-apache-samza-12-million-messages-second-single-node

Goodhope K, Koshy J, Kreps J, Narkhede N, Park R, Rao J, Ye VY (2012) Building LinkedIn's real-time activity data pipeline. IEEE Data Eng Bull 35(2):33–45. http://sites.computer.org/debull/A12june/A12JUN-CD.pdf

Hermann J, Balso MD (2017) Meet michelangelo: uber's machine learning platform. https://eng.uber.com/michelangelo/

Hindman B, Konwinski A, Zaharia M, Ghodsi A, Joseph AD, Katz R, Shenker S, Stoica I (2011) Mesos: a platform for fine-grained resource sharing in the data center. In: 8th USENIX symposium on networked systems design and implementation (NSDI)

Junqueira FP, Reed BC, Serafini M (2011) Zab: high-performance broadcast for primary-backup systems. In: 41st IEEE/IFIP international conference on dependable systems and networks (DSN), pp 245–256. https://doi.org/10.1109/DSN.2011.5958223

Kleppmann M (2017) Designing data-intensive applications. O'Reilly Media. ISBN:978-1-4493-7332-0

Kleppmann M, Kreps J (2015) Kafka, Samza and the Unix philosophy of distributed data. IEEE Data Eng Bull 38(4):4–14. http://sites.computer.org/debull/A15dec/p4.pdf

Kreps J (2014) Why local state is a fundamental primitive in stream processing. https://www.oreilly.com/ideas/why-local-state-is-a-fundamental-primitive-in-stream-processing

Kreps J, Narkhede N, Rao J (2011) Kafka: a distributed messaging system for log processing. In: 6th international workshop on networking meets databases (NetDB)

Kulkarni S, Bhagat N, Fu M, Kedigehalli V, Kellogg C, Mittal S, Patel JM, Ramasamy K, Taneja S (2015) Twitter heron: stream processing at scale. In: ACM international conference on management of data (SIGMOD), pp 239–250. https://doi.org/10.1145/2723372.2723374

Netflix Technology Blog (2016) Kafka inside Keystone pipeline. http://techblog.netflix.com/2016/04/kafka-inside-keystone-pipeline.html

Noghabi SA, Paramasivam K, Pan Y, Ramesh N, Bringhurst J, Gupta I, Campbell RH (2017) Samza:

stateful scalable stream processing at LinkedIn. Proc VLDB Endow 10(12):1634–1645. https://doi.org/10.14778/3137765.3137770

Paramasivam K (2016) Stream processing with Apache Samza – current and future. https://engineering.linkedin.com/blog/2016/01/whats-new-samza

Pathirage M, Hyde J, Pan Y, Plale B (2016) SamzaSQL: scalable fast data management with streaming SQL. In: IEEE international workshop on high-performance big data computing (HPBDC), pp 1627–1636. https://doi.org/10.1109/IPDPSW.2016.141

Qiao L, Auradar A, Beaver C, Brandt G, Gandhi M, Gopalakrishna K, Ip W, Jgadish S, Lu S, Pachev A, Ramesh A, Surlaker K, Sebastian A, Shanbhag R, Subramaniam S, Sun Y, Topiwala S, Tran C, Westerman J, Zhang D, Das S, Quiggle T, Schulman B, Ghosh B, Curtis A, Seeliger O, Zhang Z (2013) On brewing fresh Espresso: LinkedIn's distributed data serving platform. In: ACM international conference on management of data (SIGMOD), pp 1135–1146. https://doi.org/10.1145/2463676.2465298

Vavilapalli VK, Murthy AC, Douglas C, Agarwal S, Konar M, Evans R, Graves T, Lowe J, Shah H, Seth S, Saha B, Curino C, O'Malley O, Radia S, Reed B, Baldeschwieler E (2013) Apache Hadoop YARN: yet another resource negotiator. In: 4th ACM symposium on cloud computing (SoCC). https://doi.org/10.1145/2523616.2523633

Wang G, Koshy J, Subramanian S, Paramasivam K, Zadeh M, Narkhede N, Rao J, Kreps J, Stein J (2015) Building a replicated logging system with Apache Kafka. Proc VLDB Endow 8(12):1654–1655. https://doi.org/10.14778/2824032.2824063

Apache Spark

Alexandre da Silva Veith and Marcos Dias de Assuncao
Inria Avalon, LIP Laboratory, ENS Lyon, University of Lyon, Lyon, France

Definitions

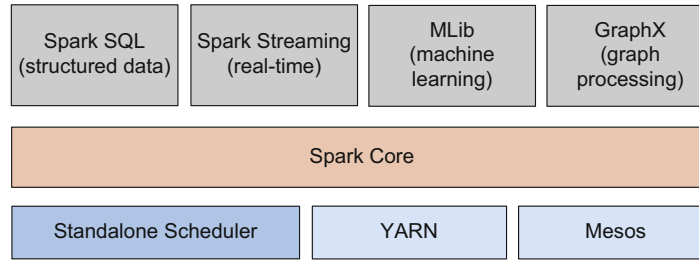
Apache Spark is a cluster computing solution and in-memory processing framework that extends the MapReduce model to support other types of computations such as interactive queries and stream processing (Zaharia et al. 2012). Designed to cover a variety of workloads, Spark introduces an abstraction called Resilient Distributed Datasets (RDDs) that enables running computations in memory in a fault-tolerant manner. RDDs, which are immutable and partitioned collections of records, provide a programming interface for performing operations, such as map, filter, and join, over multiple data items. For fault-tolerance purposes, Spark records all transformations carried out to build a dataset, thus forming a *lineage graph*.

Overview

Spark (Zaharia et al. 2016) is an open-source big data framework originally developed at the University of California at Berkeley and later adopted by the Apache Foundation, which has maintained it ever since. Spark was designed to address some of the limitations of the MapReduce model, especially the need for speed processing of large datasets. By using RDDs, purposely designed to store restricted amounts of data in memory, Spark enables performing computations more efficiently than MapReduce, which runs computations on the disk.

Although the project contains multiple components, at its core (Fig. 1) Spark is a computing engine that schedules, distributes, and monitors applications comprising multiple tasks across nodes of a computing cluster (Karau et al. 2015). For cluster management, Spark supports its native Spark cluster (standalone), Apache YARN (Vavilapalli et al. 2013), or Apache Mesos (Hindman et al. 2011). At the core also lies the RDD abstraction. RDDs are sets of data items distributed across the cluster nodes and that can be manipulated in parallel. At a higher level, it provides support for multiple tightly integrated components for handling various types of workloads such as SQL, streaming, and machine learning.

Figure 2 depicts an example of a word count application using Spark's Scala API for manipulating datasets. Spark computes RDDs in a lazy fashion, the first time they are used. Hence, the code in the example is evaluated when the counts are saved to disk, in which moment the results of the computation are required. Spark can also read data from



Apache Spark, Fig. 1 The Apache Spark stack (Karau et al. 2015)

```
// Create a Scala Spark configuration context
val config = new SparkConf().settAppName("WordCount")
val sc = new SparkContext(config)

// Load the input data
val input = sc.textFile(theInputFile)

// Split it into words
val words = input.flatMap(line => line.split(" "))

// Transform into pairs and count
val counts = words.map(word =>
      (word, 1)).reduceByKey{case (x, y) => x + y}

// Save the word count to a text file
counts.saveAsTextFile(theOutputFile)
```

Apache Spark, Fig. 2 Word count example using Spark's Scala API (Karau et al. 2015)

various sources, such as Hadoop Distributed File System (HDFS), Cassandra, OpenStack Swift (https://wiki.openstack.org/wiki/Swift), and Amazon Simple Storage Service (S3) (https://aws.amazon.com/s3/).

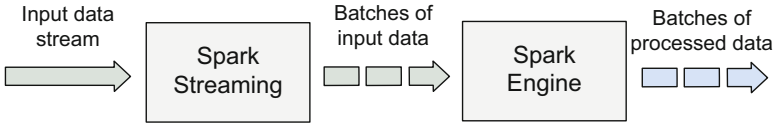
Spark SQL

Spark SQL (Armbrust et al. 2015) is a module for processing structured data (https://spark.apache.org/docs/latest/sql-programming-guide.html). It builds on the RDD abstraction by providing Spark core engine with more information about the structure of the data and the computation being performed. In addition to enabling users to perform SQL queries, Spark SQL provides the Dataset API, which offers datasets and DataFrames. A dataset can be built using JVM objects that can then be manipulated using functional transformations. A DataFrame can be built from a large number of sources and is analogous to a table in a relational database; it is a dataset organized into named columns.

Spark Streaming

Spark Streaming provides a micro-batch-based framework for processing data streams. Data can be ingested from systems such as Apache Kafka (https://kafka.apache.org/), Flume, or Amazon Kinesis (https://aws.amazon.com/kinesis/). Under the traditional stream processing approach based on a graph of continuous operators that process tuples as they arrive (i.e., the dataflow model), it is arguably difficult to achieve fault tolerance and handle stragglers. As application state is often kept by multiple operators, fault tolerance is achieved either by replicating sections of the processing graph or via upstream backup. The former demands synchronization of operators via a protocol such as Flux (Shah et al. 2003) or other transactional protocols (Wu et al. 2015), whereas the latter, when a node fails, requires parents to replay previously sent messages to rebuild the state.

Spark Streaming uses a high-level abstraction called *discretised stream* or *DStream* (Zaharia et al. 2013). As depicted in Fig. 3, DStreams



Apache Spark, Fig. 3 High-level view of discretized streams

follow a micro-batch approach that organizes stream processing as batch computations carried out periodically over small time windows. During a short time interval, DStreams store the received data, which the cluster resources then use as input dataset for performing parallel computations once the interval elapses. These computations produce new datasets that represent an intermediate state or computation outputs. The intermediate state consists of RDDs that DStreams process along with the datasets stored during the next interval. In addition to providing a strong unification with batch processing, this model stores the state in memory as RDDs that DStreams can deterministically recompute. This micro-batch approach, however, sacrifices response time as the delay for processing events is dependent on the length of the micro-batches.

MLlib

Spark contains a library with common machine learning (ML) functionality such as learning algorithms for classification, regression, clustering, and collaborative filtering and featurization including feature extraction, transformation, dimensionality reduction, and selection (Meng et al. 2016). MLlib also enables the creation of ML pipelines and persistence of algorithms, models, and pipelines. These features are designed to scale out across large computing clusters using Spark's core engine.

GraphX

GraphX (Gonzalez et al. 2014) extends the RDD API by enabling the creation of a multigraph (i.e., the property graph) with arbitrary properties attached to vertices and edges. The library is designed for manipulating graphs, exposing a set of operators (e.g., subgraph, joinVertices), and for carrying out parallel computations. It contains a library of common graph algorithms, such as PageRank and triangle counting.

Examples of Applications

Spark has been used for several data processing and data science tasks, but the range of applications that it enables is endless. Freeman et al. (2014), for instance, designed a library called Thunder on top of Spark for large-scale analysis of neural data. Many machine learning and statistical algorithms have been implemented for MLlib, which simplifies the construction of machine learning pipelines. The source code of Spark has also grown substantially since it became an Apache project. Numerous third-party libraries and packages have been included for performing tasks in certain domains or for simplifying the use of existing APIs.

Spark provides the functionalities that data scientists need to perform data transformation, processing, and analysis. Data scientists often need to perform ad hoc exploration during which they have to test new algorithms or verify the results in the least amount of time. Spark provides APIs, libraries, and shells that allow scientists to perform such tasks while enabling them to test their algorithms on large problem sizes. Once the exploration phase is performed, the solution is productized by engineers who integrate the data analysis tasks into an often more complex business application.

Examples of applications built using Apache Spark include analysis of data from mobile devices (Alsheikh et al. 2016) and Internet of Things (IoT), web-scale graph analytics, anomaly detection of user behavior and network traffic for information security (Ryza et al. 2017), real-time machine learning, data stream processing pipelines, engineering workloads, and geospatial data processing, to cite just a few. New application scenarios are presented each year during Spark's Summit (https://spark-summit.org/), an event that has become a showcase of next-generation big data applications.

Future Directions of Research

Spark APIs shine both during exploratory work and when engineering a solution deployed in production. Over the years, much effort has been paid toward making APIs easier to use and to optimize, for instance, the introduction of the Dataset API to avoid certain performance degradations that could occur if a user did not properly design the chain of operations executed by the Spark engine. As mentioned earlier, considerable work has also focused on creating new libraries and packages, including for processing live streams. The discretized model employed by Spark's stream processing API, however, introduces some delays depending on the time length of micro-batches. More recent and emerging application scenarios, such as analysis of vehicular traffic and networks, monitoring of operational infrastructure, wearable assistance (Ha et al. 2014), and 5G services, require data processing and service response under very short delays. Spark can be used as part of a larger service workflow, but alone it does not provide means to address some of the challenging scenarios that require very short response times. This requires the use of other frameworks such as Apache Storm.

To reduce the latency of applications delivered to users, many service components are also increasingly being deployed at the edges of the Internet (Hu et al. 2015) under a model commonly called edge computing. Some frameworks are available for processing streams of data using resources at the edge (e.g., Apache Edgent (https://edgent.apache.org/)), whereas others are emerging. There are also frameworks that aim to provide high-level programming abstractions for creating dataflows (e.g., Apache Beam (https://beam.apache.org/)) with underlying execution engines for several processing solutions. There is, however, a lack of unifying solutions on programming models and abstractions for exploring resources from both cloud and edge computing deployments, as well as scheduling and resource management tools for deciding what processing tasks need to be offloaded to the edge.

Cross-References

- Cloud Computing for Big Data Analysis
- Definition of Data Streams
- Graph Processing Frameworks
- Hadoop
- Large-Scale Graph Processing System
- Programmable Memory/Processing in Memory (PIM)
- Streaming Microservices

References

Alsheikh MA, Niyato D, Lin S, Tan H-P, Han Z (2016) Mobile big data analytics using deep learning and Apache Spark. IEEE Netw 30(3):22–29

Armbrust M, Xin RS, Lian C, Huai Y, Liu D, Bradley JK, Meng X, Kaftan T, Franklin MJ, Ghodsi A, Zaharia M (2015) Spark SQL: relational data processing in Spark. In: Proceedings of the 2015 ACM SIGMOD international conference on management of data (SIGMOD'15). ACM, New York, pp 1383–1394

Freeman J, Vladimirov N, Kawashima T, Mu Y, Sofroniew NJ, Bennett DV, Rosen J, Yang C-T, Looger LL, Ahrens MB (2014) Mapping brain activity at scale with cluster computing. Nat Methods 11(9):941–950

Gonzalez JE, Xin RS, Dave A, Crankshaw D, Franklin MJ, Stoica I (2014) Graphx: graph processing in a distributed dataflow framework. In: OSDI, vol 14, pp 599–613

Ha K, Chen Z, Hu W, Richter W, Pillai P, Satyanarayanan M (2014) Towards wearable cognitive assistance. In: 12th annual international conference on mobile systems, applications, and services, MobiSys'14. ACM, New York, pp 68–81. https://doi.org/10.1145/2594368.2594383

Hindman B, Konwinski A, Zaharia M, Ghodsi A, Joseph AD, Katz R, Shenker S, Stoica I (2011) Mesos: a platform for fine-grained resource sharing in the data center. In: NSDI, vol 11, pp 22–22

Hu YC, Patel M, Sabella D, Sprecher N, Young V (2015) Mobile edge computing – a key technology towards 5G. ETSI White Paper 11(11):1–16

Karau H, Konwinski A, Wendell P, Zaharia M (2015) Learning Spark: lightning-fast big data analysis. O'Reilly Media, Inc., Beijing

Meng X, Bradley J, Yavuz B, Sparks E, Venkataraman S, Liu D, Freeman J, Tsai D, Amde M, Owen S et al (2016) Mllib: machine learning in Apache Spark. J Mach Learn Res 17(1):1235–1241

Ryza S, Laserson U, Owen S, Wills J (2017) Advanced analytics with Spark: patterns for learning from data at scale. O'Reilly Media, Inc., Sebastopol

Shah MA, Hellerstein JM, Chandrasekaran S, Franklin MJ (2003) Flux: an adaptive partitioning operator

for continuous query systems. In: 19th international conference on data engineering (ICDE 2003). IEEE Computer Society, pp 25–36
Vavilapalli VK, Murthy AC, Douglas C, Agarwal S, Konar M, Evans R, Graves T, Lowe J, Shah H, Seth S, Saha B, Curino C, O'Malley O, Radia S, Reed B, Baldeschwieler E (2013) Apache hadoop YARN: yet another resource negotiator. In: 4th annual symposium on cloud computing (SOCC'13). ACM, New York, pp 5:1–5:16. https://doi.org/10.1145/2523616.2523633
Wu Y, Tan KL (2015) ChronoStream: elastic stateful stream computation in the cloud. In: 2015 IEEE 31st international conference on data engineering, pp 723–734
Zaharia M, Chowdhury M, Das T, Dave A, Ma J, McCauley M, Franklin MJ, Shenker S, Stoica I (2012) Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In: 9th USENIX conference on networked systems design and implementation (NSDI'12). USENIX Association, Berkeley, pp 2–2
Zaharia M, Das T, Li H, Hunter T, Shenker S, Stoica I (2013) Discretized streams: fault-tolerant streaming computation at scale. In: 24th ACM symposium on operating systems principles (SOSP'13). ACM, New York, pp 423–438
Zaharia M, Xin RS, Wendell P, Das T, Armbrust M, Dave A, Meng X, Rosen J, Venkataraman S, Franklin MJ, Ghodsi A, Gonzalez J, Shenker S, Stoica I (2016) Apache Spark: a unified engine for big data processing. Commun ACM 59(11):56–65

Apache Spark Benchmarking

▸ SparkBench

Apache SystemML

Declarative Large-Scale Machine Learning

Matthias Boehm
IBM Research – Almaden, San Jose, CA, USA

Definitions

Apache SystemML (Ghoting et al. 2011; Boehm et al. 2016) is a system for declarative, large-scale machine learning (ML) that aims to increase the productivity of data scientists. ML algorithms are expressed in a high-level language with R- or Python-like syntax, and the system automatically generates efficient, hybrid execution plans of single-node CPU or GPU operations, as well as distributed operations using data-parallel frameworks such as MapReduce (Dean and Ghemawat 2004) or Spark (Zaharia et al. 2012). SystemML's high-level abstraction provides the necessary flexibility to specify custom ML algorithms while ensuring physical data independence, independence of the underlying runtime operations and technology stack, and scalability for large data. Separating the concerns of algorithm semantics and execution plan generation is essential for the automatic optimization of execution plans regarding different data and cluster characteristics, without the need for algorithm modifications in different deployments.

Overview

In SystemML (Ghoting et al. 2011; Boehm et al. 2016), data scientists specify their ML algorithms using a language with R- or Python-like syntax. This language supports abstract data types for scalars, matrices and frames, and operations such as linear algebra, element-wise operations, aggregations, indexing, and statistical operations but also control structures such as loops, branches, and functions. These scripts are parsed into a hierarchy of statement blocks and statements, where control flow delineates the individual blocks. For each block of statements, the system then compiles DAGs (directed acyclic graphs) of high-level operators (HOPs), which is the core internal representation of SystemML's compiler. Size information such as matrix dimensions and sparsity are propagated via intra- and inter-procedural analysis from the inputs through the entire program. This size information is then used to compute memory estimates per operator and accordingly select physical operators resulting in a DAG of low-level operators (LOPs). These LOP DAGs are finally compiled into executable instructions.

Example: As an example, consider the following script for linear regression via a closed-form method that computes and solves the normal equations:

```
 1:  X = read($X);
 2:  y = read($Y);
 3:  lambda = 0.001;
 4:  if( $icpt == 1 )
 5:     X = cbind(X, matrix(1, nrow(X), 1));
 6:  I = matrix(1, ncol(X), 1);
 7:  A = t(X) %*% X + diag(I) * lambda;
 8:  b = t(X) %*% y;
 9:  beta = solve(A, b);
10:  write(beta, $B);
```

This script reads the feature matrix $\mathbf{X}$ and labels $\mathbf{y}$, optionally appends a column of 1s to $\mathbf{X}$ for computing the intercept, computes the normal equations, and finally solves the resulting linear system of equations. SystemML then compiles, for example, for lines 6–10, a HOP DAG that contains logical operators such as matrix multiplications for $\mathbf{X}^\top\mathbf{X}$ and $\mathbf{X}^\top\mathbf{y}$. Given input metadata (e.g., let $\mathbf{X}$ be a dense $10^7 \times 10^3$ matrix), the compiler also computes memory estimates for each operation (e.g., 80.08 GB for $\mathbf{X}^\top\mathbf{y}$). If the memory estimate of an operation exceeds the driver memory budget, this operation is scheduled for distributed execution, and appropriate physical operators are selected (e.g., `mapmm` as a broadcast-based operator for $\mathbf{X}^\top\mathbf{y}$).

Static and Dynamic Rewrites: SystemML's optimizer applies a broad range of optimizations throughout its compilation chain. An important class of optimizations with high-performance impact are rewrites. SystemML applies static and dynamic rewrites (Boehm et al. 2014a). Static rewrites are size-independent and include traditional programming language techniques – such as common subexpression elimination, constant folding, and branch removal – algebraic simplifications for linear algebra, as well as backend-specific transformations such as caching and partitioning directives for Spark. For instance, in the above example, after constant propagation, constant folding, and branch removal, the block of lines 4–5 is removed if `$icpt==0`, which further allows unconditional size propagation and the merge of the entire program into a single HOP DAG. Furthermore, the expression 'diag(matrix(1, ncol($\mathbf{X}$), 1)) $\odot$ lambda' is simplified to 'diag(matrix(lambda, ncol($\mathbf{X}$), 1))', which avoids unnecessary operations and intermediates. SystemML's rewrite system contains hundreds of such rewrites, some of which even change the asymptotic behavior (e.g., trace($\mathbf{XY}$) $\rightarrow$ sum($\mathbf{X} \odot \mathbf{Y}^\top$)). Additional dynamic rewrites are size-dependent because they require sizes for cost estimation or validity constraints. Examples are matrix multiplication chain optimization as well as dynamic simplification rewrites such as sum($\mathbf{X}^2$) $\rightarrow$ $\mathbf{X}^\top\mathbf{X}$ | ncol($\mathbf{X}$) $= 1$. The former exploits the associativity of matrix multiplications and aims to find an optimal parenthesization for which SystemML applies a textbook dynamic programming algorithm (Boehm et al. 2014a).

Operator Selection and Fused Operators: Another important class of optimizations is the selection of execution types and physical operators (Boehm et al. 2016). SystemML's optimizer analyzes the memory budgets of the driver and executors and selects – based on worst-case memory estimates (Boehm et al. 2014b) – local or distributed execution types. Besides data and cluster characteristics (e.g., data size/shape, memory,

and parallelism), the compiler also considers matrix and operation properties (e.g., diagonal/symmetric matrices, sparse-safe operations) as well as data flow properties (e.g., co-partitioning and data locality). Depending on the chosen execution types, different physical operators are considered with a selection preference from local or special-purpose to shuffle-based operators. For example, the multiplication $\mathbf{X}^\top\mathbf{X}$ from line 7 allows for a special transpose-self operator (`tsmm`), which is an easily parallelizable unary operator that exploits the output symmetry for less computation. For distributed operations, this operator has a block size constraint because it requires access to entire rows. If special-purpose or broadcast-based operators do not apply, the compiler falls back to the shuffle-based `cpmm` and `rmm` operators (Ghoting et al. 2011). Additionally, SystemML replaces special patterns with hand-coded fused operators to avoid unnecessary intermediates (Huang et al. 2015; Elgamal et al. 2017) and unnecessary scans (Boehm et al. 2014a; Ashari et al. 2015; Elgamal et al. 2017), as well as to exploit sparsity across chains of operations (Boehm et al. 2016; Elgamal et al. 2017). For example, computing the weighted squared loss via $\text{sum}(\mathbf{W} \odot (\mathbf{X} - \mathbf{U}\mathbf{V}^\top)^2)$ would create huge dense intermediates. In contrast, sparsity-exploiting operators leverage the sparse driver (i.e., the sparse matrix $\mathbf{W}$ and the sparse-safe multiply $\odot$) for selective computation that only considers nonzeros in $\mathbf{W}$.

Dynamic Recompilation: Dynamic rewrites and operator selection rely on size information for memory estimates, cost estimation, and validity constraints. Hence, unknown dimensions or sparsity lead to conservative fallback plans. Example scenarios are complex conditional control flow or function call graphs, user-defined functions, data-dependent operations, computed size expressions, and changing sizes or sparsity as shown in the following example:

```
1: while( continue ) {
2:    parfor( i in 1:n ) {
3:       if( fixed[1,i] == 0 ) {
4:          X = cbind(Xg, Xorig[,i]);
5:          AIC[1,i] = linregDS(X,y); }}
6:    #select & append best to Xg
7: }
```

This example originates from a stepwise linear regression algorithm for feature selection that iteratively selects additional features and calls the previously introduced regression algorithm. SystemML addresses this challenge of unknown sizes via dynamic recompilation (Boehm et al. 2014a) that recompiles subplans – at the granularity of HOP DAGs – with exact size information of intermediates during runtime. During initial compilation, operations and DAGs with unknowns are marked for dynamic recompilation, which also includes splitting DAGs after data-dependent operators. During runtime, the recompiler then deep copies the HOP DAG, updates sizes, applies dynamic rewrites, recomputes memory estimates, generates new runtime instructions, and resumes execution with the intermediate results computed so far. This approach yields good plans and performance even in the presence of initial unknowns.

Runtime Integration: At runtime level, SystemML interprets the generated instructions. Single-node and Spark operations directly map to instructions, whereas for the MapReduce backend, instructions are packed into a minimal number of MR jobs. For distributed operations, matrices (and similarly frames) are stored in a blocked representation of pairs of block indexes and blocks with fixed block size, where individual blocks can be dense, sparse, or ultra-sparse. In contrast, for single-node operations, the entire matrix is represented as a single block, which allows reusing the block runtime across backends. Data transfers between the local and distributed backends and driver memory management are handled by a multilevel buffer pool (Boehm et al. 2016) that controls local evictions, parallelizes and collects RDDs, creates broadcasts, and reads/writes data from/to the distributed file system. For example, a single-node instruction first pins its inputs into memory – which triggers reads from HDFS or RDDs if necessary – performs the block operation, registers the output in the buffer pool, and finally unpins its inputs. Many block operations and the I/O system for different formats are multi-threaded to exploit parallelism in scale-up environments. For compute-intensive deep

learning workloads, SystemML further calls native CPU and GPU libraries for BLAS and DNN operations. In contrast to other deep learning frameworks, SystemML also supports sparse neural network operations. Memory management for the GPU device is integrated with the buffer pool allowing for lazy data transfer on demand. Finally, SystemML uses numerically stable operations based on Kahan addition for descriptive statistics and certain aggregation functions (Tian et al. 2012).

Key Research Findings

In additional to the compiler and runtime techniques described so far, there are several advanced techniques with high-performance impact.

Task-Parallel Parfor Loops: SystemML's primary focus is data parallelism. However, there are many use cases such as ensemble learning, cross validation, hyper-parameter tuning, and complex models with disjoint or overlapping data that are naturally expressed in a task-parallel manner. These scenarios are addressed by SystemML's `parfor` construct for parallel for loops (Boehm et al. 2014b). In contrast to similar constructs in high-performance computing (HPC), `parfor` only asserts the independence of iterations, and a dedicated `parfor` optimizer reasons about hybrid parallelization strategies that combine data and task parallelism. Reconsider the stepwise linear regression example. Alternative plan choices include (1) a local, i.e., multi-threaded, `parfor` with local operations, (2) a remote `parfor` that runs the entire loop as a distributed Spark job, or (3) a local `parfor` with concurrent data-parallel Spark operations if the data does not fit into the driver.

Resource Optimization: The selection of execution types and operators is strongly influenced by memory budgets of the driver and executor processes. Finding a good static cluster configuration that works well for a broad range of ML algorithms and data sizes is a hard problem. SystemML addresses this challenge with a dedicated resource optimizer for automatic resource provisioning (Huang et al. 2015) on resource negotiation frameworks such as YARN or Mesos. The key idea is to optimize resource configurations via an online what-if analysis with regard to the given ML program as well as data and cluster characteristics. This framework optimizes performance without unnecessary over-provisioning, which can increase throughout in shared on-premise clusters and save money in cloud environments.

Compressed Linear Algebra: Furthermore, there is a broad class of iterative ML algorithms that use repeated read-only data access and I/O-bound matrix-vector multiplications to converge to an optimal model. For these algorithms, it is crucial for performance to fit the data into available single-node or distributed memory. However, general-purpose, lightweight, and heavyweight compression techniques struggle to achieve both good compression ratios and fast decompression to enable block-wise uncompressed operations. Compressed linear algebra (CLA) (Elgohary et al. 2016) tackles this challenge by applying lightweight database compression techniques – for column-wise compression with heterogeneous encoding formats and co-coding – to matrices and executing linear algebra operations such as matrix-vector multiplications directly on the compressed representation. CLA yields compression ratios similar to heavyweight compression and thus allows fitting large datasets into memory while achieving operation performance close to the uncompressed case.

Automatic Operator Fusion: Similar to query compilation and loop fusion in databases and HPC, the opportunities for fused operators – in terms of fused chains of operations – are ubiquitous. Example benefits are a reduced number of intermediates, reduced number of scans, and sparsity exploitation across operations. Despite their high-performance impact, hand-coded fused operators are usually limited to few operators and incur a large development effort. Automatic operator fusion via code generation (Elgamal et al. 2017) overcomes this challenge

by automatically determining valid fusion plans and generating access-pattern-aware operators in the form of hand-coded skeletons with custom body code. In contrast to existing work on operator fusion, SystemML introduced a cost-based optimizer framework to find optimal fusion plans in DAGs of linear algebra programs for dense, sparse, and compressed data as well as local and distributed operations.

Examples of Application

SystemML has been applied in a variety of ML applications including statistics, classification, regression, clustering, matrix factorization, survival analysis, and deep learning. In contrast to specialized systems for graphs like GraphLab (Low et al. 2012) or deep learning like TensorFlow (Abadi et al. 2016), SystemML provides a unified system for small- to large-scale problems with support for dense, sparse, and ultra-sparse data, as well as local and distributed operations. Accordingly, SystemML's primary application area is an environment with diverse algorithms, varying data characteristics, or different deployments.

Example deployments include large-scale computation on top of MapReduce or Spark and programmatic APIs for notebook environments or embedded scoring. Thanks to deployment-specific compilation, ML algorithms can be reused without script changes. This flexibility enabled the integration of SystemML into systems with different architectures. For example, SystemML has been shipped as part of the open-source project R4ML and the IBM products BigInsights, Data Science Experience (DSX), and multiple Watson services.

Future Directions for Research

Given the goal of a unified system for ML applications and recent algorithm and hardware trends, there are many directions for future research throughout the stack of SystemML and similar systems (Kumar et al. 2017):

Specification Languages: SystemML focuses on optimizing fixed algorithm specifications. However, end-to-end applications would further benefit from even higher levels of abstractions for feature engineering, model selection, and life cycle management in general. A promising direction is a stack of declarative languages that allows for reuse and cross-level optimization.

Optimization Techniques: Regarding the automatic optimization of ML programs, further work is required regarding size and sparsity estimates, adaptive query processing and storage (as an extension of dynamic recompilation), and principled approaches to automatic rewrites and automatic operator fusion.

Runtime Techniques: A better support for deep learning and scientific applications requires the extension from matrices to dense/sparse tensors of different data types and their operations. Additionally, further research is required regarding the automatic exploitation of accelerators and heterogenous hardware.

Benchmarks: Finally, SystemML – but also the community at large – would benefit from dedicated benchmarks for the different classes of ML workloads and different levels of specification languages.

Cross-References

- Apache Mahout
- Cloud Computing for Big Data Analysis
- Hadoop
- Python
- Scalable Architectures for Big Data Analysis
- Tools and Libraries for Big Data Analysis

References

Abadi M et al (2016) TensorFlow: a system for large-scale machine learning. In: OSDI

Ashari A, Tatikonda S, Boehm M, Reinwald B, Campbell K, Keenleyside J, Sadayappan P (2015) On optimizing machine learning workloads via Kernel fusion. In: PPoPP

Boehm M, Burdick DR, Evfimievski AV, Reinwald B, Reiss FR, Sen P, Tatikonda S, Tian Y (2014a) SystemML's optimizer: plan generation for large-scale machine learning programs. IEEE Data Eng Bull 37(3):52–62

Boehm M, Tatikonda S, Reinwald B, Sen P, Tian Y, Burdick D, Vaithyanathan S (2014b) Hybrid parallelization strategies for large-scale machine learning in SystemML. PVLDB 7(7):553–564

Boehm M, Dusenberry M, Eriksson D, Evfimievski AV, Manshadi FM, Pansare N, Reinwald B, Reiss F, Sen P, Surve A, Tatikonda S (2016) SystemML: declarative machine learning on spark. PVLDB 9(13): 1425–1436

Dean J, Ghemawat S (2004) MapReduce: simplified data processing on large clusters. In: OSDI

Elgamal T, Luo S, Boehm M, Evfimievski AV, Tatikonda S, Reinwald B, Sen P (2017) SPOOF: sum-product optimization and operator fusion for large-scale machine learning. In CIDR

Elgohary A, Boehm M, Haas PJ, Reiss FR, Reinwald B (2016) Compressed linear algebra for large-scale machine learning. PVLDB 9(12): 960–971

Ghoting A, Krishnamurthy R, Pednault EPD, Reinwald B, Sindhwani V, Tatikonda S, Tian Y, Vaithyanathan S (2011) SystemML: declarative machine learning on MapReduce. In: ICDE

Huang B, Boehm M, Tian Y, Reinwald B, Tatikonda S, Reiss FR (2015) Resource elasticity for large-scale machine learning. In: SIGMOD

Kumar A, Boehm M, Yang J (2017) Data management in machine learning: challenges, techniques, and systems. In: SIGMOD

Low Y, Gonzalez J, Kyrola A, Bickson D, Guestrin C, Hellerstein JM (2012) Distributed GraphLab: a framework for machine learning in the cloud. PVLDB 5(8)

Tian Y, Tatikonda S, Reinwald B (2012) Scalable and numerically stable descriptive statistics in SystemML. In: ICDE

Zaharia M, Chowdhury M, Das T, Dave A, Ma J, McCauly M, Franklin MJ, Shenker S, Stoica I (2012) Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In: NSDI

Applications

▸ Big Data in Computer Network Monitoring

Applications of Big Spatial Data: Health

Lance A. Waller
Department of Biostatistics and Bioinformatics, Rollins School of Public Health, Emory University, Atlanta, GA, USA

Definitions

The term "big spatial data" encompasses all types of big data with the addition of geographic reference information, typically a location associated with a point in space (e.g., latitude, longitude, and altitude coordinates), an area (e.g., a country, a district, or a census enumeration zone), a line or curve (e.g., a river or a road), or a pixel (e.g., high-resolution satellite images or a biomedical imaging scan). When applied to questions of health, big spatial data can aid in attempts to understand geographic variations in the risks and rates of disease (e.g., is risk here greater than risk there?), to identify local factors driving geographic variations in risks and rates (e.g., does local nutritional status impact local childhood mortality?), and to evaluate the impact of local health policies (e.g., district-specific adjustments to insurance reimbursements).

In addition to defining big spatial data, it is also important to define what is meant by "health." The World Health Organization defines *health* as "a state of complete physical, mental and social well-being and not merely the absence of disease or infirmity" (Preamble to the WHO Constitution, http://www.who.int/suggestions/faq/en/). This can be interpreted at the individual level and at the population level. At the population level, the US Centers for Disease Control and Prevention define *public health* as "the science of protecting and improving the health of people and their communities. This work is achieved by promoting healthy lifestyles, researching disease and injury prevention, and detecting, preventing and responding to

infectious diseases" (https://www.cdcfoundation.org/what-public-health). More recently, there has been increasing interest in the concept of *population health*, i.e., "the health outcomes of a group of individuals, including the distribution of such outcomes within the group" (Kindig and Stoddard 2003). Both public and population health focus on measures of health in an overall community with public health typically focusing on health promotion and disease prevention and population health as defining current and target distributions of health outcomes across a population, typically, to evaluate the impact of policy (e.g., government funding or insurance coverage).

Overview

The rapid expansion of measurement technology, data storage, informatics, and analytics feeds rapid growth in applications of big data within the fields of individual, population, and public health. The sequencing of the human genome in 2001 was rapidly followed by the ability to measure multiple aspects of the human metabolome, transcriptome, microbiome, and overall exposure to the environment (the exposome) (Miller and Jones 2014; Wild 2005). These detailed views of the complex world of each individual's genetic, environmental, and social makeup provide quantitative insight into a more nuanced view of "health" than simply the absence of disease; rather, the measures begin to provide a view into *how* health works at individual, local, societal, and global levels. The measures move from tracking individual biomarkers to panels of biomarkers and toward integration of heterogeneous streaming of information from wearable sensors, local exposure monitors, and georeferenced tracking devices. These heterogeneous data sources enable increasingly detailed exploration of associations between an individual's location and their phenotype, genotype, and multiple influencing factors relating to diet, activity level, environment, and prevention activities, some of which vary locally themselves. More specifically, the multiple measures provide insight into aspects of and interactions between the elements of individual, public, and population health, defined above.

Geography provides a key context for all three types of health; specifically, the locations where one lives and works contribute greatly to one's individual exposome and to the range of environmental and social determinants of disease (and health) associated with the health of the local population. As a result, big spatial data play a key (and expanding) role in the definition, description, and understanding of health.

Key Research Findings

The rapid increase of health-related data and the continued growth of geographic information systems and science (Goodchild 1992) impact both health care and health research. In addition, the ecosocial theory of social determinants of health (Krieger 2001) motivated many health leaders (including the Director of the Robert Wood Johnson Foundation and the Director of the US National Institutes of Health) to comment that an individual's ZIP code (US postal code) is a more powerful predictor of health than that individual's genetic code.

Spatial analysis of health traditionally includes geographically referenced, local information on health outcomes, demographics, economics, policies, and potential exposures. Analyses often focus on predicting exposures from a fixed set of observations, assessing of whether point locations of disease occur in geographically concentrated "clusters" or quantifying associations between health outcomes and local measures of potential risk factors (Waller and Gotway 2004). A variety of biostatistical and epidemiologic methods have been extended to allow for potential spatial and spatiotemporal correlations.

The incorporation of big data into a spatial setting provides intriguing opportunities to refine both concepts of health and concepts of location as illustrated below.

Examples of Application

Beginning at the individual patient level, electronic health records (EHRs) contain a potential trove of medical information regarding individual's interaction with the healthcare system, consolidating information invaluable for health care (Murdoch and Detsky 2013). This information typically is stored in multiple, heterogeneous formats designed to allow access by healthcare professionals for determining patient care. Within a given health system, these records are often consolidated into secure data warehouses to allow system-wide assessment of quality of care, monitoring of costs, and reimbursement by health insurance systems. Moving from health care of an individual to research often requires assessing information from multiple systems in different formats. Recent years have seen an increase in explorations of the research potential of EHRs, including (but not limited to) applications of natural language processing of text fields filled by healthcare professionals, development of searchable database of biomedical images, and system-wide assessments of healthcare performance.

In addition to the development of informatics and analytics for EHRs, there is also a rapid increase in applications of informatics and distributed computing services to manage, merge, search, and summarize streaming data from multiple devices within a healthcare setting such as an intensive care unit, emergency department, or surgery facility. Some measures are monitored almost continuously (e.g., blood pressure, blood oxygen levels), while others occur on less frequent timescales (e.g., baseline levels, treatment protocols, caregivers on call). Similar issues arise outside of the healthcare setting in the increasing popularity of wearable monitoring technology providing feedback on activity levels, heart rate, and, in some cases, measures of one's ambient environment. In many cases, such streaming technologies are linked to mobile devices that are also recording location information leading to the development of georeferenced mobile health (mHealth) platforms and analytics (Estrin and Sim 2010; Nilsen et al. 2012).

Recent years have also seen a rise in the amount of, connectivity between, and use of heterogeneous data from disparate sources to address health questions. Examples include (but again are not limited to) the use of remote sensing data to assess exposure potential for airborne pollutants on local (Liu et al. 2005) and global scales (Shaddick et al. 2017), the use of Internet information in public health surveillance (Brownstein et al. 2009), and the expanded use of administrative data across multiple countries coupled with reported epidemiologic associations to provide assessments of the global burden of disease (Murray and Lopez 1997). The potential is remarkable, but desire for accurate health information can also lead to the temptation to ignore past epidemiologic findings in light of the promise of predictive analytics, a temptation that can lead to gross inaccuracies and potential false alarms (Lazar et al. 2014).

In addition to expanded possibilities of referencing individual-level attributions, volunteered geographic information via location-based apps and related services provide near real-time access to individual-level movement, providing novel opportunities for assessing exposures (Sui et al. 2013). Such information on location and movement provide additional context for moving from maps of local risk (Kitron 1998) to maps of an individual's cumulative experienced risk in moving through such an exposure space. Vazquez-Prokopec et al. (2009) provide a creative example exploring space-time exposures of taxi drivers to mosquitos carrying dengue virus in Iquitos, Peru.

The expanding availabilities of individual-level health and location data raise ongoing discussions of data privacy (determined by an individual's choices of which of their individual data to reveal), data confidentiality (the ability of the data holders to maintain each individual's privacy), and data access (determining which agencies, healthcare systems, physicians can access an individual's personal information). Concepts and comfort levels relating to data privacy vary culturally and between countries leading to differences in regulatory processes to evaluate whether data holders maintain confidentiality. For example, in the United States, the Health Insurance Portability and Accountability Act (HIPAA) of 1996 defines data protection standards for "protected health

information" (PHI) and its use with the healthcare system. HIPAA lists individual location data at a finer resolution than broadscale (3-digit) US ZIP codes among its list of PHI data elements, requiring care and documentation in the collection, storage, visualization, and reporting of geographic information at this and finer resolutions.

The competitive environment of US healthcare systems and the proprietary environment of providers of electronic health record (EHR) systems result in considerable informatics challenges when individual patients interact with multiple providers across different units. Some interesting progress has been made toward informatics solutions (e.g., the Fast Healthcare Interoperability Resources (FHIR)) to address these administrative barriers to data sharing and allow data sharing through app-based platforms (Mandel et al. 2016), but this remains an active area of research in the fields of medical informatics, particularly for FHIR extensions involving geolocation data extensions.

Future Directions for Research

The examples above illustrate that health applications of big spatial data technologies draw from traditional spatial analyses within individual, population, and public health but also illustrate that, to date, such work typically occurs in application-specific settings. The development of cohesive concepts and associated accurate and reliable toolboxes for spatial analytics and their health applications remains an area of active research. These developments include many challenging issues relating to accurate healthcare monitoring, assessment, and provision for each individual patient. Health applications extend well beyond that of individual health, and provide detailed insight into population and public health as well. There is great need for creativity in informatics to allow connection between, linkage across, and searches of elements of spatially referenced health and health-related measurements. As noted above, "health-related" can range from within-individual measures of immune response to satellite-based assessments of local environmental conditions or global models of climate impact on pollution levels or food sources. Robust and reproducible spatial health analysis requires coordination of expert knowledge across multiple disciplines in order to fully reach its vast potential.

Cross-References

- ▸ Big Data for Health
- ▸ Privacy-Preserving Record Linkage
- ▸ Spatio-Social Data
- ▸ Spatiotemporal Data: Trajectories
- ▸ Using Big Spatial Data for Planning User Mobility

References

Brownstein JS, Freifeld CC, Madoff LC (2009) Digital disease detection: harnessing the web for public health surveillance. N Engl J Med 360:2153–2157

Estrin D, Sim I (2010) Open mHealth architecture: an engine for health care innovation. Science 330: 759–760

Goodchild MF (1992) Geographic information science. Int J Geogr Inf Syst 6:31–45

Kindig D, Stoddart G (2003) What is population health? Am J Public Health 93:380–383

Kitron U (1998) Landscape ecology and epidemiology of vector-borne diseases: tools for spatial analysis. J Med Entomol 35:435–445

Krieger N (2001) Theories for social epidemiology in the 21st century: an ecosocial perspective. Int J Epidemiol 30:668–677

Lazar D, Kennedy R, King G, Vespignani A (2014) The parable of Google Flu: traps in big data analysis. Science 343:1203–1205

Liu Y, Sarnat JA, Kilaru V, Jacob DJ, Koutrakis P (2005) Estimating ground-level PM2.5 in the Eastern United States using satellite remote sensing. Environ Sci Technol 39:3269–3278

Mandel JC, Kreda DA, Mandl KD, Kohane IS, Romoni RB (2016) SMART on FHIR: a standards-based, interoperable apps platform for electronic health records. J Am Med Inform Assoc 23:899–908

Miller GM, Jones DP (2014) The nature of nurture: refining the definition of the exposome. Toxicol Sci 137:1–2

Murdoch TB, Detsky AS (2013) The inevitable application of big data to heath care. J Am Med Assoc 309:1351–1352

Murray CJL, Lopez AD (1997) Alternative projections of mortality and disability by cause 1990–2020: Global Burden of Disease Study. Lancet 349:1498–1504

Nilsen W, Kumar S, Shar A, Varoquiers C, Wiley T, Riley WT, Pavel M, Atienza AA (2012) Advancing the science of mHealth. J Health Commun 17(supplement 1):5–10

Shaddick G, Thomas ML, Green A, Brauer M, van Donkelaar A, Burnett R, Chang HH, Cohen A, van Dingenen R, Dora C, Gumy S, Liu Y, Martin R, Waller LA, West J, Zidek JV, Pruss-Ustun A (2017) Data integration model for air quality: a hierarchical approach to the global estimation of exposures to air pollution. J R Stat Soc Ser C 67:231–253

Sui D, Elwood S, Goodchild M (eds) (2013) Crowdsourcing geographic knowledge: volunteered geographic information in theory and practice. Springer, Dondrecht

Vazquez-Prokopec GM, Stoddard ST, Paz-Soldan V, Morrison AC, Elder JP, Kochel TJ, Scott TW, Kitron U (2009) Usefulness of commercially available GPS data-loggers for tracking human movement and exposure to dengue virus. Int J Health Geogr 8:68. https://doi.org/10.1186/1476-072X-8-68

Waller LA, Gotway CA (2004) Applied spatial statistics for public health data. Wiley, Hoboken

Wild CP (2005) Complementing the genome with the "exposome": the outstanding challenge of environmental exposure measurement in molecular epidemiology. Cancer Epidemiol Biomark Prev 14:1847–1850

Approximate Computation

▸ Tabular Computation

Approximate Computing for Stream Analytics

Do Le Quoc[1], Ruichuan Chen[2,4], Pramod Bhatotia[3], Christof Fetzer[1], Volker Hilt[2], and Thorsten Strufe[1]
[1]TU Dresden, Dresden, Germany
[2]Nokia Bell Labs, Stuttgart, Germany
[3]The University of Edinburgh and Alan Turing Institute, Edinburgh, UK
[4]Nokia Bell Labs, NJ, USA

Introduction

Stream analytics systems are extensively used in the context of modern online services to transform continuously arriving raw data streams into useful insights (Foundation 2017a; Murray et al. 2013; Zaharia et al. 2013). These systems target low-latency execution environments with strict service-level agreements (SLAs) for processing the input data streams.

In the current deployments, the low-latency requirement is usually achieved by employing more computing resources. Since most stream processing systems adopt a data-parallel programming model (Dean and Ghemawat 2004), almost linear scalability can be achieved with increased computing resources (Quoc et al. 2013, 2014, 2015a,b). However, this scalability comes at the cost of ineffective utilization of computing resources and reduced throughput of the system. Moreover, in some cases, processing the entire input data stream would require more than the available computing resources to meet the desired latency/throughput guarantees.

To strike a balance between the two desirable, but contradictory design requirements – low latency and efficient utilization of computing resources – there is a surge of *approximate computing* paradigm that explores a novel design point to resolve this tension. In particular, approximate computing is based on the observation that many data analytics jobs are amenable to an approximate rather than the exact output (Doucet et al. 2000; Natarajan 1995). For such workflows, it is possible to trade the output accuracy by computing over a subset instead of the entire data stream. Since computing over a subset of input requires less time and computing resources, approximate computing can achieve desirable latency and computing resource utilization.

Unfortunately, the advancements in approximate computing are primarily geared towards batch analytics (Agarwal et al. 2013; Srikanth et al. 2016), where the input data remains unchanged during the course of computation. In particular, these systems rely on pre-computing a set of samples on the static database, and take an appropriate sample for the query execution based on the user's requirements (i.e., query execution budget). Therefore, the state-of-the-art systems cannot be deployed in the context of stream processing, where the new data continuously arrives as an unbounded stream.

As an alternative, one could in principle *repurpose* the available sampling mechanisms in well-known big data processing frameworks such as Apache Spark to build an approximate computing system for stream analytics. In fact, as a starting point for this work, based on the available sampling mechanisms, an approximate computing system is designed and implemented for stream processing in Apache Spark. Unfortunately, Spark's stratified sampling algorithm suffers from three key limitations for approximate computing. First, Spark's stratified sampling algorithm operates in a "batch" fashion, i.e., all data items are first collected in a batch as Resilient Distributed Datasets (RDDs) (Zaharia et al. 2012), and thereafter, the actual sampling is carried out on the RDDs. Second, it does not handle the case where the arrival rate of sub-streams changes over time because it requires a pre-defined sampling fraction for each stratum. Lastly, the stratified sampling algorithm implemented in Spark requires synchronization among workers for the expensive join operation, which imposes a significant latency overhead.

To address these limitations, this work designed an *online stratified reservoir sampling algorithm* for stream analytics. Unlike existing Spark-based systems, the algorithm performs the sampling process "on-the-fly" to reduce the latency as well as the overheads associated in the process of forming RDDs. Importantly, the algorithm *generalizes* to two prominent types of stream processing models: (1) batched stream processing employed by Apache Spark Streaming (Foundation 2017b), and (2) pipelined stream processing employed by Apache Flink (Foundation 2017a).

More specifically, the proposed sampling algorithm makes use of two techniques: reservoir sampling and stratified sampling. It performs reservoir sampling for each sub-stream by creating a fixed-size reservoir per stratum. Thereafter, it assigns weights to all strata respecting their arrival rates to preserve the statistical quality of the original data stream. The proposed sampling algorithm naturally adapts to varying arrival rates of sub-streams, and requires no synchronization among workers (see section "Design"). Based on the proposed sampling algorithm, StreamApprox– an approximate computing system for stream analytics – is designed.

Overview and Background

This section gives an overview of StreamApprox (section "System Overview"), its computational model (section "Computational Model"), and its design assumptions (section "Design Assumptions").

System Overview

StreamApprox is designed for real-time stream analytics. In this system, the input data stream usually consists of data items arriving from diverse sources. The data items from each source form a *sub-stream*. The system makes use of a stream aggregator (e.g., Apache Kafka Foundation 2017c) to combine the incoming data items from disjoint sub-streams. StreamApprox then takes this combined stream as the input for data analytics.

StreamApprox facilitate data analytics on the input stream by providing an interface for users to specify the streaming query and its corresponding query budget. The query budget can be in the form of expected latency/throughput guarantees, available computing resources, or the accuracy level of query results.

StreamApprox ensures that the input stream is processed within the specified query budget. To achieve this goal, the system makes use of approximate computing by processing only a subset of data items from the input stream, and produce an approximate output with rigorous error bounds. In particular, StreamApprox uses a parallelizable online sampling technique to select and process a subset of data items, where the sample size can be determined based on the query budget.

Computational Model

The state-of-the-art distributed stream processing systems can be classified in two prominent categories: (i) batched stream processing model,

and (ii) pipelined stream processing model. These systems offer three main advantages: (a) efficient fault tolerance, (b) "exactly-once" semantics, and (c) unified programming model for both batch and stream analytics. *The proposed algorithm for approximate computing is generalizable to both stream processing models, and preserves their advantages.*

Batched stream processing model. In this computational model, an input data stream is divided into small batches using a pre-defined batch interval, and each such batch is processed via a distributed data-parallel job. Apache Spark Streaming (Foundation 2017b) adopted this model to process input data streams.

Pipelined stream processing model. In contrast to the batched stream processing model, the pipelined model streams each data item to the next operator as soon as the item is ready to be processed without forming the whole batch. Thus, this model achieves low latency. Apache Flink (Foundation 2017a) implements this model to provide a truly native stream processing engine.

Note that both stream processing models support the time-based sliding window computation (Bhatotia et al. 2014). The processing window slides over the input stream, whereby the newly incoming data items are added to the window and the old data items are removed from the window. The number of data items within a sliding window may vary in accordance to the arrival rate of data items.

Design Assumptions

StreamApprox is based on the following assumptions. The possible means to address these assumptions are discussed in section "Discussion".

1. There exists a virtual cost function which translates a given query budget (such as the expected latency guarantees, or the required accuracy level of query results) into the appropriate sample size.
2. The input stream is stratified based on the source of data items, i.e., the data items from each sub-stream follow the same distribution and are mutually independent. Here, a *stratum* refers to one sub-stream. If multiple sub-streams have the same distribution, they are combined to form a stratum.

Design

In this section, first the StreamApprox's workflow (section "System Workflow") is presented. Then, its sampling mechanism (section "Online Adaptive Stratified Reservoir Sampling") and its error estimation mechanism (section "Error Estimation") are described (see details in Quoc et al. 2017d,c).

System Workflow

This section shows the workflow of StreamApprox. The system takes the user-specified streaming *query* and the query *budget* as the input. Then it executes the query on the input data stream as a sliding window computation (see section "Computational Model").

For each time interval, StreamApprox first derives the sample size (*sampleSize*) using a cost function based on the given query budget. Next, the system performs a proposed sampling algorithm (detailed in section "Online Adaptive Stratified Reservoir Sampling") to select the appropriate $sample$ in an online fashion. This sampling algorithm further ensures that data items from all sub-streams are fairly selected for the sample, and no single sub-stream is overlooked.

Thereafter, the system executes a data-parallel job to process the user-defined *query* on the selected sample. As the last step, the system performs an error estimation mechanism (as described in section "Error Estimation") to compute the error bounds for the approximate query result in the form of $output \pm error\ bound$. The whole process repeats for each time interval as the computation window slides (Bhatotia et al. 2012a).

Online Adaptive Stratified Reservoir Sampling

To realize the real-time stream analytics, a novel sampling technique called Online Adaptive Stratified Reservoir Sampling (OASRS) is proposed. It achieves both stratified and reservoir samplings without their drawbacks. Specifically, OASRS does not overlook any sub-streams regardless of their popularity, does not need to know the statistics of sub-streams before the sampling process, and runs efficiently in real time in a distributed manner.

The high-level idea of OASRS is simple. The algorithm first stratifies the input stream into sub-streams according to their sources. The data items from each sub-stream are assumed to follow the same distribution and are mutually independent. (Here, a *stratum* refers to one sub-stream. If multiple sub-streams have the same distribution, they can be combined to form a stratum.) The algorithm then samples each sub-stream independently, and perform the reservoir sampling for each sub-stream individually. To do so, every time a new sub-stream S_i is encountered, its sample size N_i is determined according to an adaptive cost function considering the specified query budget. For each sub-stream S_i, the algorithm performs the traditional reservoir sampling to select items at random from this sub-stream, and ensures that the total number of selected items from S_i does not exceed its sample size N_i. In addition, the algorithm maintains a counter C_i to measure the number of items received from S_i within the concerned time interval.

Applying reservoir sampling to each sub-stream S_i ensures that algorithm can randomly select at most N_i items from each sub-stream. The selected items from different sub-streams, however, should *not* be treated equally. In particular, for a sub-stream S_i, if $C_i > N_i$ (i.e., the sub-stream S_i has more than N_i items in total during the concerned time interval), the algorithm randomly selects N_i items from this sub-stream and each selected item represents C_i/N_i original items on average; otherwise, if $C_i \leq N_i$, the algorithm selects all the received C_i items so that each selected item only represents itself. As a result, in order to statistically recreate the original items from the selected items, the algorithm assigns a specific weight W_i to the items selected from each sub-stream S_i:

$$W_i = \begin{cases} C_i/N_i & \text{if } C_i > N_i \\ 1 & \text{if } C_i \leq N_i \end{cases} \quad (1)$$

StreamApprox supports *approximate linear queries* which return an approximate weighted sum of all items received from all sub-streams. Though linear queries are simple, they can be extended to support a large range of statistical learning algorithms (Blum et al. 2005, 2008). It is also worth mentioning that, OASRS not only works for a concerned time interval (e.g., a sliding time window), but also works with unbounded data streams.

Distributed execution. OASRS can run in a distributed fashion naturally as it does not require synchronization. One straightforward approach is to make each sub-stream S_i be handled by a set of w worker nodes. Each worker node samples an equal portion of items from this sub-stream and generates a local reservoir of size no larger than N_i/w. In addition, each worker node maintains a local counter to measure the number of its received items within a concerned time interval for weight calculation. The rest of the design remains the same.

Error Estimation

This section describes how to apply OASRS to randomly sample the input data stream to generate the approximate results for linear queries. Next, a method to estimate the accuracy of approximate results via rigorous error bounds is presented.

Similar to section "Online Adaptive Stratified Reservoir Sampling", suppose the input data stream contains X sub-streams $\{S_i\}_{i=1}^{X}$. StreamApprox computes the approximate sum of all items received from all sub-streams by randomly sampling only Y_i items from each sub-stream S_i. As each sub-stream is sampled independently, the variance of the approximate sum is: $\text{Var(SUM)} = \sum_{i=1}^{X} \text{Var(SUM}_i)$.

Further, as items are randomly selected for a sample within each sub-stream, according to the random sampling theory (Thompson 2012), the variance of the approximate sum can be estimated as:

$$\widehat{\text{Var}}(\text{SUM}) = \sum_{i=1}^{X} \left(C_i \times (C_i - Y_i) \times \frac{s_i^2}{Y_i} \right) \quad (2)$$

Here, C_i denotes the total number of items from the sub-stream S_i, and s_i denotes the standard deviation of the sub-stream S_i's sampled items:

$$s_i^2 = \frac{1}{Y_i - 1} \times \sum_{j=1}^{Y_i} (I_{i,j} - \bar{I}_i)^2, \text{where}$$
$$\bar{I}_i = \frac{1}{Y_i} \times \sum_{j=1}^{Y_i} I_{i,j} \quad (3)$$

Next, the estimation of the variance of the approximate mean value of all items received from all the X sub-streams is described. This approximate mean value can be computed as:

$$\text{MEAN} = \frac{\text{SUM}}{\sum_{i=1}^{X} C_i} = \frac{\sum_{i=1}^{X} (C_i \times \text{MEAN}_i)}{\sum_{i=1}^{X} C_i} = \sum_{i=1}^{X} (\omega_i \times \text{MEAN}_i) \quad (4)$$

Here, $\omega_i = \frac{C_i}{\sum_{i=1}^{X} C_i}$. Then, as each sub-stream is sampled independently, according to the random sampling theory (Thompson 2012), the variance of the approximate mean value can be estimated as:

$$\widehat{\text{Var}}(\text{MEAN}) = \sum_{i=1}^{X} \text{Var}(\omega_i \times \text{MEAN}_i) = \sum_{i=1}^{X} \left(\omega_i^2 \times \text{Var}(\text{MEAN}_i) \right) = \sum_{i=1}^{X} \left(\omega_i^2 \times \frac{s_i^2}{Y_i} \times \frac{C_i - Y_i}{C_i} \right) \quad (5)$$

Above, the estimation of the variances of the approximate sum and the approximate mean of the input data stream has been shown. Similarly, the variance of the approximate results of any linear queries also can be estimated by applying the random sampling theory.

Error bound. According to the "68-95-99.7" rule (Wikipedia 2017), approximate result falls within one, two, and three standard deviations away from the true result with probabilities of 68%, 95%, and 99.7%, respectively, where the standard deviation is the square root of the variance as computed above. This error estimation is critical because it gives a quantitative understanding of the accuracy of the proposed sampling technique.

Discussion

The design of StreamApprox is based on the assumptions mentioned in section "Design Assumptions". This section discusses some approaches that could be used to meet the assumptions.

I: Virtual cost function. This work currently assumes that there exists a virtual cost function to translate a user-specified query budget into the sample size. The query budget could be specified as either available computing resources, desired accuracy, or latency.

For instance, with an accuracy budget, the sample size for each sub-stream can be determined based on a desired width of the confidence interval using Eq. (5) and the "68-95-99.7" rule. With a desired latency budget, users can specify it by defining the window time interval or the slide interval for the computations over the input data stream. It becomes a bit more challenging to specify a budget for resource utilization. Nevertheless, there are two existing techniques that could be used to implement such a cost function to achieve the desired resource target: (a) virtual data center (Angel et al. 2014), and (b) resource prediction model (Wieder et al. 2012) for latency requirements.

Pulsar (Angel et al. 2014) proposes an abstraction of a virtual data center (VDC) to provide performance guarantees to tenants in the cloud. In particular, Pulsar makes use of a virtual cost function to translate the cost of a request processing into the required computational resources using a multi-resource token algorithm. The cost function could be adapted for StreamApprox as follows: a data item in the input stream is considered as a request and the "amount of resources" required to process it as the cost in tokens. Also, the given resource budget is converted in the form of tokens, using the pre-advertised cost model per resource. This allows computing the sample size that can be processed within the given resource budget.

For any given latency requirement, resource prediction model (Wieder et al. 2010a,b, 2012) could be employed. In particular, the prediction model could be built by analyzing the diurnal patterns in resource usage (Charles et al. 2012) to predict the future resource requirement for the given latency budget. This resource requirement can then be mapped to the desired sample size based on the same approach as described above.

II: Stratified sampling. This work currently assume that the input stream is already stratified based on the source of data items, i.e., the data items within each stratum follow the same distribution – it does not have to be a normal distribution. This assumption ensures that the error estimation mechanism still holds correct since StreamApprox applies the Central Limit Theorem. For example, consider an IoT use-case which analyzes data streams from sensors to measure the temperature of a city. The data stream from each individual sensor follows the same distribution since it measures the temperature at the same location in the city. Therefore, a straightforward way to stratify the input data streams is to consider each sensor's data stream as a stratum (sub-stream). In more complex cases where StreamApprox cannot classify strata based on the sources, the system needs a pre-processing step to stratify the input data stream. This stratification problem is orthogonal to this work, nevertheless for completeness, two proposals for the stratification of evolving streams, bootstrap (Dziuda 2010) and semi-supervised learning (Masud et al. 2012), are discussed in this section.

Bootstrap (Dziuda 2010) is a well-studied non-parametric sampling technique in statistics for the estimation of distribution for a given population. In particular, the bootstrap technique randomly selects "bootstrap samples" with replacement to estimate the unknown parameters of a population, for instance, by averaging the bootstrap samples. A bootstrap-based estimator can be employed for the stratification of incoming sub-streams. Alternatively, a semi-supervised algorithm (Masud et al. 2012) could be used to stratify a data stream. The advantage of this algorithm is that it can work with both labeled and unlabeled streams to train a classification model.

Related Work

Over the last two decades, the databases community has proposed various approximation techniques based on sampling (Al-Kateb and Lee 2010; Garofalakis and Gibbon 2001), online aggregation (Hellerstein et al. 1997), and sketches (Cormode et al. 2012). These techniques make different trade-offs w.r.t. the output quality, supported queries, and workload. However, the early work in approximate computing was mainly geared towards the centralized database architecture.

Recently, sampling-based approaches have been successfully adopted for distributed data analytics (Agarwal et al. 2013; Srikanth et al. 2016; Krishnan et al. 2016; Quoc et al. 2017b,a). In particular, BlinkDB (Agarwal et al. 2013) proposes an approximate distributed query processing engine that uses stratified sampling (Al-Kateb and Lee 2010) to support ad-hoc queries with error and response time constraints. Like BlinkDB, Quickr (Srikanth et al. 2016) also supports complex ad-hoc queries in big-data clusters. Quickr deploys distributed sampling operators to reduce execution costs of

parallelized queries. In particular, Quickr first injects sampling operators into the query plan; thereafter, it searches for an optimal query plan among sampled query plans to execute input queries. However, these "big data" systems target batch processing and cannot provide required low-latency guarantees for stream analytics.

IncApprox (Krishnan et al. 2016) is a data analytics system that combines two computing paradigms together, namely, approximate and incremental computations (Bhatotia et al. 2011a,b, 2012b) for stream analytics. The system is based on an online "biased sampling" algorithm that uses self-adjusting computation (Bhatotia 2015; Bhatotia et al. 2015) to produce incrementally updated approximate output. Lastly, PrivApprox (Quoc et al. 2017a,b) supports privacy-preserving data analytics using a combination of randomized response and approximate computation. By contrast, StreamApprox supports low-latency in stream processing by employing the proposed "online" sampling algorithm solely for approximate computing, while avoiding the limitations of existing sampling algorithms.

Conclusion

This paper presents StreamApprox, a stream analytics system for approximate computing. StreamApprox allows users to make a systematic trade-off between the output accuracy and the computation efficiency. To achieve this goal, StreamApprox employs an online stratified reservoir sampling algorithm which ensures the statistical quality of the sample selected from the input data stream. The proposed sampling algorithm is generalizable to two prominent types of stream processing models: batched and pipelined stream processing models.

Cross-References

▸ Incremental Approximate Computing

References

Agarwal S, Mozafari B, Panda A, Milner H, Madden S, Stoica I (2013) BlinkDB: queries with bounded errors and bounded response times on very large data. In: Proceedings of the ACM European conference on computer systems (EuroSys)

Al-Kateb M, Lee BS (2010) Stratified reservoir sampling over heterogeneous data streams. In: Proceedings of the 22nd international conference on scientific and statistical database management (SSDBM)

Angel S, Ballani H, Karagiannis T, O'Shea G, Thereska E (2014) End-to-end performance isolation through virtual datacenters. In: Proceedings of the USENIX conference on operating systems design and implementation (OSDI)

Bhatotia P (2015) Incremental parallel and distributed systems. PhD thesis, Max Planck Institute for Software Systems (MPI-SWS)

Bhatotia P, Wieder A, Akkus IE, Rodrigues R, Acar UA (2011a) Large-scale incremental data processing with change propagation. In: Proceedings of the conference on hot topics in cloud computing (HotCloud)

Bhatotia P, Wieder A, Rodrigues R, Acar UA, Pasquini R (2011b) Incoop: MapReduce for incremental computations. In: Proceedings of the ACM symposium on cloud computing (SoCC)

Bhatotia P, Dischinger M, Rodrigues R, Acar UA (2012a) Slider: incremental sliding-window computations for large-scale data analysis. Technical report MPI-SWS-2012-004, MPI-SWS. http://www.mpi-sws.org/tr/2012-004.pdf

Bhatotia P, Rodrigues R, Verma A (2012b) Shredder: GPU-accelerated incremental storage and computation. In: Proceedings of USENIX conference on file and storage technologies (FAST)

Bhatotia P, Acar UA, Junqueira FP, Rodrigues R (2014) Slider: incremental sliding window analytics. In: Proceedings of the 15th international middleware conference (Middleware)

Bhatotia P, Fonseca P, Acar UA, Brandenburg B, Rodrigues R (2015) iThreads: a threading library for parallel incremental computation. In: Proceedings of the 20th international conference on architectural support for programming languages and operating systems (ASPLOS)

Blum A, Dwork C, McSherry F, Nissim K (2005) Practical privacy: the sulq framework. In: Proceedings of the twenty-fourth ACM SIGMOD-SIGACT-SIGART symposium on principles of database systems (PODS)

Blum A, Ligett K, Roth A (2008) A learning theory approach to non-interactive database privacy. In: Proceedings of the fortieth annual ACM symposium on theory of computing (STOC)

Charles R, Alexey T, Gregory G, Randy HK, Michael K (2012) Towards understanding heterogeneous clouds at scale: Google trace analysis. Techical report

Cormode G, Garofalakis M, Haas PJ, Jermaine C (2012) Synopses for massive data: samples, histograms, wavelets, sketches. Foundations and Trends in Databases. Now, Boston

Dean J, Ghemawat S (2004) MapReduce: simplified data processing on large clusters. In: Proceedings of the USENIX conference on operating systems design and implementation (OSDI)

Doucet A, Godsill S, Andrieu C (2000) On sequential monte carlo sampling methods for bayesian filtering. Stat Comput 10:197–208

Dziuda DM (2010) Data mining for genomics and proteomics: analysis of gene and protein expression data. Wiley, Hoboken

Foundation AS (2017a) Apache flink. https://flink.apache.org

Foundation AS (2017b) Apache spark streaming. https://spark.apache.org/streaming

Foundation AS (2017c) Kafka – a high-throughput distributed messaging system. https://kafka.apache.org

Garofalakis MN, Gibbon PB (2001) Approximate query processing: taming the terabytes. In: Proceedings of the international conference on very large data bases (VLDB)

Hellerstein JM, Haas PJ, Wang HJ (1997) Online aggregation. In: Proceedings of the ACM SIGMOD international conference on management of data (SIGMOD)

Krishnan DR, Quoc DL, Bhatotia P, Fetzer C, Rodrigues R (2016) IncApprox: a data analytics system for incremental approximate computing. In: Proceedings of the 25th international conference on World Wide Web (WWW)

Masud MM, Woolam C, Gao J, Khan L, Han J, Hamlen KW, Oza NC (2012) Facing the reality of data stream classification: coping with scarcity of labeled data. Knowl Inf Syst 33:213–244

Murray DG, McSherry F, Isaacs R, Isard M, Barham P, Abadi M (2013) Naiad: a timely dataflow system. In: Proceedings of the twenty-fourth ACM symposium on operating systems principles (SOSP)

Natarajan S (1995) Imprecise and approximate computation. Kluwer Academic Publishers, Boston

Quoc DL, Martin A, Fetzer C (2013) Scalable and real-time deep packet inspection. In: Proceedings of the 2013 IEEE/ACM 6th international conference on utility and cloud computing (UCC)

Quoc DL, Yazdanov L, Fetzer C (2014) Dolen: user-side multi-cloud application monitoring. In: International conference on future internet of things and cloud (FICLOUD)

Quoc DL, D'Alessandro V, Park B, Romano L, Fetzer C (2015a) Scalable network traffic classification using distributed support vector machines. In: Proceedings of the 2015 IEEE 8th international conference on cloud computing (CLOUD)

Quoc DL, Fetzer C, Felber P, Étienne Rivière, Schiavoni V, Sutra P (2015b) Unicrawl: a practical geographically distributed web crawler. In: Proceedings of the 2015 IEEE 8th international conference on cloud computing (CLOUD)

Quoc DL, Beck M, Bhatotia P, Chen R, Fetzer C, Strufe T (2017a) Privacy preserving stream analytics: the marriage of randomized response and approximate computing. https://arxiv.org/abs/1701.05403

Quoc DL, Beck M, Bhatotia P, Chen R, Fetzer C, Strufe T (2017b) PrivApprox: privacy-preserving stream analytics. In: Proceedings of the 2017 USENIX conference on USENIX annual technical conference (USENIX ATC)

Quoc DL, Chen R, Bhatotia P, Fetzer C, Hilt V, Strufe T (2017c) Approximate stream analytics in apache flink and apache spark streaming. CoRR, abs/1709.02946

Quoc DL, Chen R, Bhatotia P, Fetzer C, Hilt V, Strufe T (2017d) StreamApprox: approximate computing for stream analytics. In: Proceedings of the international middleware conference (middleware)

Srikanth K, Anil S, Aleksandar V, Matthaios O, Robert G, Surajit C, Ding B (2016) Quickr: lazily approximating complex ad-hoc queries in big data clusters. In: Proceedings of the ACM SIGMOD international conference on management of data (SIGMOD)

Thompson SK (2012) Sampling. Wiley series in probability and statistics. The Australasian Institute of Mining and Metallurgy, Carlton

Wieder A, Bhatotia P, Post A, Rodrigues R (2010a) Brief announcement: modelling mapreduce for optimal execution in the cloud. In: Proceedings of the 29th ACM SIGACT-SIGOPS symposium on principles of distributed computing (PODC)

Wieder A, Bhatotia P, Post A, Rodrigues R (2010b) Conductor: orchestrating the clouds. In: Proceedings of the 4th international workshop on large scale distributed systems and middleware (LADIS)

Wieder A, Bhatotia P, Post A, Rodrigues R (2012) Orchestrating the deployment of computations in the cloud with conductor. In: Proceedings of the 9th USENIX symposium on networked systems design and implementation (NSDI)

Wikipedia (2017) 68-95-99.7 Rule

Zaharia M, Chowdhury M, Das T, Dave A, Ma J, McCauley M, Franklin MJ, Shenker S, Stoica I (2012) Resilient distributed datasets: a fault tolerant abstraction for in-memory cluster computing. In: Proceedings of the 9th USENIX conference on networked systems design and implementation (NSDI)

Zaharia M, Das T, Li H, Hunter T, Shenker S, Stoica I (2013) Discretized streams: fault-tolerant streaming computation at scale. In: Proceedings of the twenty-fourth ACM symposium on operating systems principles (SOSP)

Approximate Reasoning

▸ Automated Reasoning

Approximated Neighborhood Function

▸ Degrees of Separation and Diameter in Large Graphs

Architectures

Erik G. Hoel
Environmental Systems Research Institute, Redlands, CA, USA

Synonyms

Apache Hadoop; Lambda architecture; Location analytic architecture

Definitions

Spatial big data is a spatio-temporal data that is too large or requires data-intensive computation that is too demanding for traditional computing architectures. *Stream processing* in this context is the processing of spatio-temporal data in motion. The data is observational; it is produced by sensors – moving or otherwise. Computations on the data are made as the data is produced or received. A *distributed processing cluster* is a networked collection of computers that communicate and process data in a coordinated manner. Computers in the cluster are coordinated to solve a common problem. A *lambda architecture* is a scalable, fault-tolerant data-processing architecture that is designed to handle large quantities of data by exploiting both stream and batch processing methods. *Data partitioning* involves physically dividing a dataset into separate data stores on a distributed processing cluster. This is done to achieve improved scalability, performance, availability, and fault-tolerance. *Distributed file systems*, in the context of big data architectures, are similar to traditional distributed file systems but are intended to persist large datasets on commodity hardware in a fault-tolerant manner with simple coherency models. The *MapReduce programming model* is intended for large-scale distributed data processing and is based upon simple concepts involving iterating over data, performing a computation on key/value pairs, grouping the intermediary values by key, iterating over the resulting groups, and reducing each group to provide a result. A *GPU-accelerated distributed processing framework* is an extension to a traditional distributed processing framework that supports offloading tasks to GPUs for further acceleration.

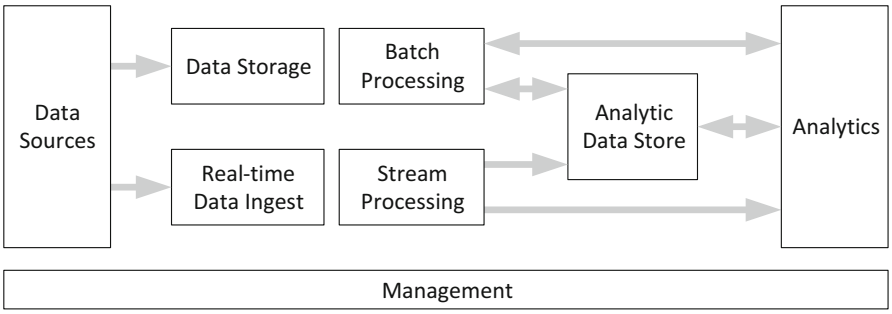
Overview

Spatial big data architectures are intended to address requirements for spatio-temporal data that is too large or computationally demanding for traditional computing architectures (Shekhar et al. 2012). This includes operations such as real-time data ingest, stream processing, batch processing, storage, and spatio-temporal analytical processing.

Spatial big data offers additional challenges beyond what is commonly faced in the big data space. This includes advanced indexing, querying, analytical processing, visualization, and machine learning. Examples of spatial big data include moving vehicles (peer-to-peer ridesharing, delivery vehicles, ships, airplanes, etc.), stationary sensor data (e.g., SCADA, AMI), cell phones (call detail records), IoT devices, as well as spatially enabled content from social media (Fig. 1).

Spatial big data architectures are used in a variety of workflows:

- Real-time processing of observational data (data in motion). This commonly involves monitoring and tracking dynamic assets in real-time; this can include vehicles, aircraft, and vessels, as well as stationary assets such as weather and environmental monitoring sensors.
- Batch processing of persisted spatio-temporal data (data at rest). Workflows with data at rest incorporate tabular and spatial processing



Architectures, Fig. 1 Generic big data architecture

(e.g., summarizing data, analyzing patterns, and proximity analysis), geoenrichment and geoenablement (adding spatial capabilities to non-spatial data, adding information from contextual spatial data collections), and machine learning and predictive analytics (clustering, classification, and prediction).

Architectural Requirements

When designing and developing scalable software systems that can address the high-level workflows encountered in spatial big data systems, a number of basic requirements must be identified. It is important to note that most of these also generally apply to traditional non-spatial big data systems (Mysore et al. 2013; Klein et al. 2016; Sena et al. 2017):

Scalable
Spatial big data systems must be scalable. Scalability encompasses being able to increase and support different amounts of data, processing them, and allocating computational resources without impacting costs or efficiency. To meet this requirement, it is required to distribute data sets and their processing across multiple computing and storage nodes.

High Performant
Spatial big data systems must be able to process large streams of data in a short period of time, thus returning the results to users as efficiently as possible. In addition, the system should support computation intensive spatio-temporal analytics.

Real-time
Big data systems must be able to manage the continuous flow of data and its processing in real time, facilitating decision-making.

Consistent
Big data systems must support data consistency, heterogeneity, and exploitation. Different data formats must be also managed to represent useful information for the system.

Secure
Big data systems must ensure security in the data and its manipulation in the architecture, supporting integrity of information, exchanging data, multilevel policy-driven, access control, and prevent unauthorized access.

Available
Big data systems must ensure high data availability, through data replication horizontal scaling (i.e., distribute a data set over clusters of computers and storage nodes). The system must allow replication and handle hardware failures.

Interoperable
Big data systems must be transparently intercommunicated to allow exchanging information between machines and processes, interfaces, and people.

Key Research Findings

Spatial big data architectures have existed since the early 2000s with some of the original implementations at Google and Microsoft. Some of the key research issues related to spatial big

data architectures include data storage (repositories, distributes storage, NoSQL databases), distributed spatio-temporal analytic processing, stream processing, scalable cloud computing, and GPU-enabled distributed processing frameworks.

Data Storage

Parallel Database Systems

Big data repositories have existed in many forms, frequently built by corporations and governmental agencies with special requirements. Beginning in the 1990s, commercial vendors began offering parallel database management systems to address the needs of big data (DeWitt and Gray 1992). Parallel database systems are classified as being shared memory (processing elements sharing memory), shared disk (processing elements do not share memory, but do share disk storage), or shared nothing (neither memory nor disk storage is shared between processing elements). Significant efforts included those by Teradata, IBM, Digital, Microsoft, Oracle, Google, and Amazon. Additionally, beginning in the 1980s, there were numerous research systems that contributed to these efforts (e.g., GAMMA and Bubba; DeWitt et al. 1986, Alexander and Copeland 1988).

Distributed File Stores

Hadoop Distributed File System (HDFS) is a distributed file-system that stores data on commodity machines, providing very high aggregate bandwidth across the cluster (Shvachko et al. 2010). It was inspired by the Google File System (GFS; Ghemawat et al. 2003). HDFS stores large files (typically in the range of gigabytes to terabytes) across multiple machines. It achieves reliability by replicating the data across multiple hosts, and hence theoretically does not require redundant array of independent disks (RAID) storage on hosts (but to increase input-output (I/O) performance, RAID configurations may be employed). With the default replication value of three, data is stored on three nodes: two on the same rack and one on a different rack. Data nodes can talk to each other to rebalance data, to move copies around, and to keep the replication of data high.

NoSQL Databases

A NoSQL (originally referencing "non-SQL" or "non-relational") database is a mechanism for the storage and retrieval of data that is modeled differently from standard relational databases (NoSQL 2009; Pavlo and Aslett 2016). NoSQL databases are often considered next generation databases; they are intended to address weaknesses of traditional relational databases such as being readily distributable, simpler in design, open-source, and horizontally scalable (often problematic for relational databases). Many databases supporting these characteristics originated in the late 1960s; the "NoSQL" description was employed beginning in the late 1990s with the requirements imposed by companies such as Facebook, Google, and Amazon. NoSQL databases are commonly used with big data applications. NoSQL systems are also sometimes called "Not only SQL" to emphasize that they may support SQL-like query languages.

In order to achieve increased performance and scalability, NoSQL databases commonly used data structures (e.g., key-value, columnar, document, or graph) that are different from those used in relational databases. NoSQL databases vary in terms of applicability to particular problem domains. NoSQL databases are often classified by their primal data structures; examples include:

- Key-value: Apache Ignite, Couchbase, Dynamo, Oracle NoSQL Database, Redis, Riak
- Columnar: Accumulo, Cassandra, Druid, HBase, Vertica
- Document: Apache CouchDB, Cosmos DB, IBM Domino, MarkLogic, MongoDB
- Graph: AllegroGraph, Apache Giraph, MarkLogic, Neo4J
- Multi-model: Apache Ignite, Couchbase, MarkLogic

Spatial Batch Processing

Spatio-temporal analysis in a batch context involves a very wide scope of functionality. In academia, much of the research has focused on the spatial join (or spatio-temporal join) func-

tion. In commercial systems, spatial analysis also includes summarizing data, incident and similar location detection, proximity analysis, pattern analysis, and data management (import, export, cleansing, etc.).

Spatial Joins

Spatial joins have been widely studied in both the standard sequential environment (Jacox and Samet 2007), as well as in the parallel (Brinkhoff et al. 1996) and distributed environments (Abel et al. 1995). For over 20 years, algorithms have been developed to take advantage of parallel and distributed processing architectures and software frameworks. The recent resurgence in interest in spatial join processing is the results of new-found interest in distributed, fault-tolerant, computing frameworks such as Apache Hadoop, as well as the explosion in observational and IoT data.

With distributed processing architectures, there are two principal approaches that are employed when performing spatial joins. The first, termed a broadcast (or mapside) spatial join, is designed for joining a large dataset with another small dataset (e.g., political boundaries). The large dataset is partitioned across the processing nodes and the complete small dataset is broadcast to each of the nodes. This allows significant optimization opportunities. The second approach, termed a partitioned (or reduce side) spatial join, is a more general technique that is used when joining two large datasets. Partitioned joins use a divide-and-conquer approach (Aji et al. 2013). The two large datasets are divided into small pieces via a spatial decomposition, and each small piece is processed independently.

SJMR (Spatial Join with MapReduce) introduced the first distributed spatial join on Hadoop using the MapReduce programming model (Dean and Ghemawat 2008; Zhang et al. 2009). SpatialHadoop (Eldawy and Mokbel 2015) optimized SJMR with a persistent spatial index (it supports grid files, R − trees, and R + trees) that is precomputed. Hadoop-GIS (Aji et al. 2013), which is utilized in medical pathology imaging, features both 2D and 3D spatial join. GIS Tools for Hadoop (Whitman et al. 2014) is an open source library that implements range and distance queries and k-NN. It also supports a distributed PMR quadtree-based spatial index. GeoSpark (Yu et al. 2015) is a framework for performing spatial joins, range, and k-NN queries. The framework supports quadtree and R-tree indexing of the source data. Magellan (Sriharsha 2015) is an open source library for geospatial analytics that uses Spark (Zaharia et al. 2010). It supports a broadcast join and a reduce-side optimized join and is integrated with Spark SQL for a traditional, SQL user experience. SpatialSpark (You et al. 2015) supports both a broadcast spatial join and a partitioned spatial join on Spark. The partitioning is supported using either fixed-grid, binary space partition or a sort-tile approach. STARK (Hagedorn et al. 2017) is a Spark-based framework that supports spatial joins, k-NN, and range queries on both spatial and spatio-temporal data. STARK supports three temporal operators: contains, containedBy, and intersects) and also supports the DBSCAN density-based spatial clusterer (Ester et al. 1996). MSJS (multi-way spatial join algorithm with Spark (Du et al. 2017)) addresses the problem of performing multi-way spatial joins using the common technique of cascading sequences of pairwise spatial joins. Simba (Xie et al. 2016) offers range, distance (circle range), and k-NN queries as well as distance and k-NN joins. Two-level indexing, global and local, is employed, similar to the various indexing work on Hadoop MapReduce. Location-Spark (Tang et al. 2016) supports range query, k-NN, spatial join, and k-NN join. Location-Spark uses global and local indices – Grid, R-tree, Quadtree, and IR-tree. GeoMesa is an open-source, distributed, spatio-temporal index built on top of Bigtable-style databases (Chang et al. 2008) using an implementation of the Geohash algorithm implemented in Scala (Hughes et al. 2015). The Esri GeoAnalytics Server (Whitman et al. 2017) supports many types of spatial analysis is a distributed environment (leveraging the Spark framework). It provides functionality for summarizing data (e.g., aggregation, spatio-temporal join, polygon overlay), incident and similar location detection, proximity

analysis, and pattern analysis (hot spot analysis, NetCDF generation).

MapReduce Programming Model

MapReduce is a programming model and an associated implementation for processing big data sets with a parallel, distributed algorithm (Sakr et al. 2013). A MapReduce program is composed of a map procedure (or method), which performs filtering and sorting (such as sorting students by first name into queues, one queue for each name), and a reduce method, which performs a summary operation (such as counting the number of students in each queue, yielding name frequencies). A MapReduce framework manages the processing by marshalling the distributed cluster nodes, running the various tasks and algorithms in parallel, managing communications and data transfers between cluster nodes, while supporting fault tolerance and redundancy.

The MapReduce model is inspired by the map and reduces functions commonly used in functional programming (note that their purpose in the MapReduce framework is not the same as in their original forms). The key contributions of the MapReduce model are not the actual map and reduce functions that resemble the Message Passing Interface (MPI) standard's reduce and scatter operations. The major contributions of the MapReduce model are the scalability and fault-tolerance that is supported through optimization of the execution engine. A single-threaded implementation of MapReduce is commonly slower than a traditional (non-MapReduce) implementation; gains are typically realized with multi-node or multi-threaded implementations.

MapReduce libraries have been written in many programming languages, with different levels of optimization. The most popular open-source implementation is found in Apache Hadoop.

Stream Processing

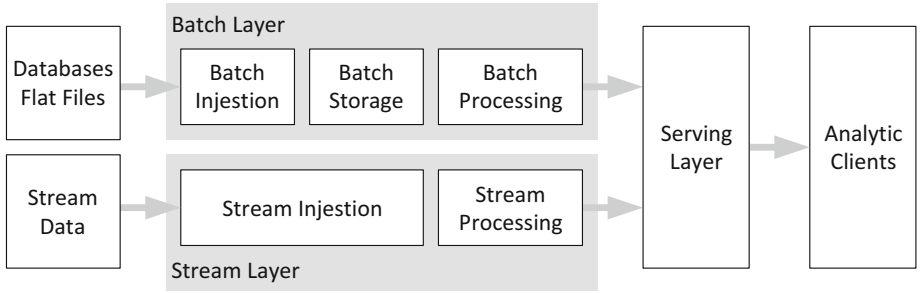
Stream processing is a computer programming paradigm, equivalent to dataflow programming, event stream processing, and reactive programming that allows some applications to more easily exploit a limited form of parallel processing (Gedik et al. 2008). Such applications can use multiple computational units, such as the floating point unit on a graphics processing unit or field-programmable gate arrays (FPGAs), without explicitly managing allocation, synchronization, or communication among those units.

The stream processing paradigm simplifies parallel software and hardware by restricting the parallel computation that can be performed. Given a sequence of data (a stream), a series of operations (kernel functions) is applied to each element in the stream. Kernel functions are usually pipelined, and optimal local on-chip memory reuse is attempted, in order to minimize the loss in bandwidth, accredited to external memory interaction. Uniform streaming, where one kernel function is applied to all elements in the stream, is typical. Since the kernel and stream abstractions expose data dependencies, compiler tools can fully automate and optimize on-chip management tasks.

Lambda Architecture

A Lambda Architecture is an architecture that is intended to process large volumes of data by incorporating both batch and real-time processing techniques (Marz and Warren 2015). This approach to architecture attempts to balance latency, throughput, and fault-tolerance by using batch processing to provide comprehensive and accurate views of batch data, while simultaneously using real-time stream processing to provide views of online data. The two view outputs may be joined before presentation. In addition, historic analysis is used to tune the real-time analytical processing as well as building models for prediction (machine learning). The rise of lambda architecture is correlated with the growth of big data, real-time analytics, and the drive to mitigate the latencies of map-reduce (Fig. 2).

The Lambda architecture depends on a data model with an append-only, immutable data source that serves as a system of record. It is intended for ingesting and processing timestamped events that are appended to existing events rather than overwriting them. State is determined from the natural time-based ordering of the data.



Architectures, Fig. 2 Generic lambda architecture

GPU-Accelerated Distributed Frameworks

Distributed processing frameworks such as Spark (which support in-memory processing) have been extended and enhanced with the incorporation of GPUs for key computationally intensive operations (e.g., machine learning and graph theoretic algorithms, Prasad et al. 2015). Researchers have observed that a few Spark nodes with GPUs can outperform a much larger cluster of non-GPU nodes (Grossman and Sarkar 2016; Hassaan and Elghandour 2016; Yuan et al. 2016; Hong et al. 2017). The main bottlenecks when incorporating GPUs in hybrid architectures often involve data communication, memory and resource management, and differences in programming models. Different approaches to solving these problems have employed GPU-wrapper APIs (e.g., PyCUDA), hybrid RDDs (resilient distributed datasets) where the RDD is stored in the CPU, generating native GPU code from high-level source code written for the distributed framework (e.g., Scala, Java, or Python code with Spark) or native GPU RDDs where data is processed and stored in the GPU device memory (Fig. 3).

Examples of Application

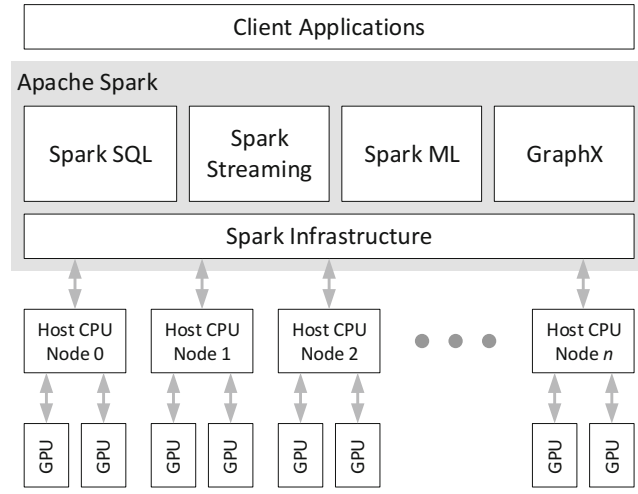
The application of technologies related to spatial big data architectures is broad given the rapidly growing interest in spatial data that has emerged during the twenty-first century. Notable among this family of technologies in terms of significance and application include distributed processing frameworks, geo-spatial stream processing, and the numerous implementations of platform as a service (PaaS).

Apache Hadoop

Apache Hadoop (Apache 2006) is an open-source software framework and associated utilities that facilitate using a network of commodity computers to solve problems involving large amounts of data and computation. Inspired by the seminal work at Google on MapReduce and the Google File System (GFS), Hadoop provides a software framework for both the distributed storage and processing of big data using the MapReduce programming model.

Similar to the efforts at Google, Hadoop was designed for computer clusters built from commodity hardware (still the common usage pattern). Hadoop has also been employed on large clusters of higher-end hardware. All the modules in Hadoop are designed with a fundamental assumption that hardware failures are common occurrences and should be automatically handled by the framework.

The core of Apache Hadoop consists of a storage part, known as Hadoop Distributed File System (HDFS), a resource manager, a collection of utilities, and a processing framework that is an implementation of the MapReduce programming model that runs against large clusters of machines. HDFS splits very large files (including those of size gigabytes and larger) into blocks that are distributed across multiple nodes in a cluster. Reliability is achieved by replicating the blocks across multiple nodes (with a default replication factor of 3). Hadoop distributes packaged code into nodes to process the data in parallel.



Architectures, Fig. 3 GPU-accelerated Spark framework

Architectures, Fig. 4 Spark streaming micro-batch architecture

This approach takes advantage of data locality, where nodes manipulate the data they have access to. This allows the dataset to be processed faster and more efficiently than it would be in a more conventional supercomputer architecture that relies on a parallel file system where computation and data are distributed via high-speed networking.

Hadoop has been deployed in traditional datacenters as well as in the cloud. The cloud allows organizations to deploy Hadoop without the need to acquire hardware or specific setup expertise. Vendors who currently have an offering for the cloud that incorporate Hadoop include Microsoft, Amazon, IBM, Google, and Oracle. Most of the Fortune 50 companies currently deploy Hadoop clusters.

Spark Streaming

Spark Streaming is an extension to Spark API that supports scalable, high-throughput, fault-tolerant, stream processing of real-time data streams (Garillot and Maas 2018). Data can be ingested from many sources (e.g., Kafka, Flume, or TCP sockets) and can be processed using temporally aware algorithms expressed with high-level functions like map, reduce, join, and window. Finally, processed data can be pushed out to filesystems, databases, and live dashboards. Spark's machine learning (Spark ML) and graph processing (GraphX) algorithms can be applied to these data streams.

Internally, Streaming receives live input data streams and divides the data into batches, which are then processed by the Spark engine to generate the final stream of results in batches (Fig. 4).

Spark Streaming provides a high-level abstraction called *discretized stream* or *DStream*, which represents a continuous stream of data. DStreams can be created either from input data streams from sources such as Kafka, Flume, and Kinesis, or by applying high-level operations on other DStreams. Internally, a DStream is represented as a sequence of RDDs.

Big Data as a Service

Platform as a Service (PaaS) is a category of cloud computing services that provides a platform allowing customers to run and

manage applications without the complexity of building and maintaining the infrastructure usually associated with developing and launching an application (Chang et al. 2010). PaaS is commonly delivered in one of three ways:

- As a public cloud service from a provider
- As a private service (software or appliance) inside the firewall
- As software deployed on a public infrastructure as a service

Big Data as a Service (BDaaS) is a new concept that combines Software as a Service (SaaS), Platform as a Service (PaaS), and Data as a Service (DaaS) in order to address the requirements of working with massively large data sets. BDaas offerings commonly incorporate the Hadoop stack (e.g., HDFS, Hive, MapReduce, Pig, Storm, and Spark), NoSQL data stores, and stream processing capabilities.

Microsoft Azure is a cloud computing service utilizing Microsoft-managed data centers that supports both software as a service (SaaS) and platform as a service (PaaS). It provides data storage capabilities including Cosmos DB (a NoSQL database), the Azure Data Lake, and SQL Server-based databases. Azure supports a scalable event processing engine and a machine learning service that supports predictive analytics and data science applications.

The Google Cloud is a Paas offering that supports big data with data warehousing, batch and stream processing, data exploration, and support for the Hadoop/Spark framework. Key components include BigQuery, a managed data warehouse supporting analytics at scale, Cloud Dataflow, which supports both stream and batch processing, and Cloud Dataproc, a framework for running Apache MapReduce and Spark processes.

Amazon AWS, though commonly considered an Infrastructure as a Service (IaaS) where the user is responsible for configuration, AWS also provides PaaS functionality. Amazon supports Elastic MapReduce (EMR) that works in conjunction with EC2 (Elastic Compute Cloud) and S3 (Simple Storage Service). Data storage is provided through DynamoDB (NoSQL), Redshift (columnar), and RDS (relational data store). Machine learning and real-time data processing infrastructures are also supported.

Other significant examples of BDaaS providers include the IBM Cloud and the Oracle Data Cloud. Big data Infrastructure as a Service (IaaS) offerings (that work with other clouds such as AWS, Azure, and Oracle) are available from Hortonworks, Cloudera, Esri, and Databricks.

Future Directions for Research

Despite the significant advancements that have been made over the past decade on key topics relates to spatial big data architectures, much further research is necessary in order to further democratize the capabilities and application to broader problem domains. Some of the more significant areas needing attention include:

- Spatio-temporally enabling distributed and NoSQL databases such as Accumulo, Cassandra, HBase, Dynamo, and Elasticsearch. This involves not only supporting spatial types but also incorporating rich collections of topological, spatial, and temporal operators.
- Spatio-temporal analytics is another area requiring attention. Much research to date has focused on supporting spatial (or spatio-temporal) joins on distributed frameworks such as MapReduce or Spark. While beneficial, spatio-temporal analytics is a far richer domain that also includes geostatistics (e.g., kriging), spatial statistics, proximity analysis, and pattern analysis.
- Spatially enabling machine learning algorithms that run in a distributed cluster (e.g., extending Spark ML or Scikit-learn (Pedregosa et al. 2011)) is another significant research area given the growing interest and importance of machine learning, predictive analytics, and deep learning. To date, research

has primarily focused on density-based clustering algorithms such as DBSCAN, HDBSCAN (McInnes and Healy 2017), and OPTICS.

- Recently, much attention has been paid to incorporating GPU processing capabilities into distributed processing frameworks such as Spark. While some basic spatial capabilities can currently be supported (e.g., aggregation and visualization of point data), much work needs to be done to further streamline and optimize the integration of GPU processors and extend the native spatio-temporal capabilities.

Cross-References

- ▸ Big Data Architectures
- ▸ Real-Time Big Spatial Data Processing
- ▸ Streaming Big Spatial Data

References

Abel DJ, Ooi BC, Tan K-L, Power R, Yu JX (1995) Spatial join strategies in distributed spatial DBMS. In: Advances in spatial databases – 4th international symposium, SSD'95. Lecture notes in computer science, vol 1619. Springer, Portland, pp 348–367

Aji A, Wang F, Vo H, Lee R, Liu Q, Zhang X, Saltz J (2013) Hadoop-GIS: a high performance spatial data warehousing system over mapreduce. Proc VLDB Endow 6(11):1009–1020

Alexander W, Copeland G (1988) Process and dataflow control in distributed data-intensive systems. In: Proceedings of the 1988 ACM SIGMOD international conference on management of data (SIGMOD '88), pp 90–98. https://doi.org/10.1145/50202.50212

Apache (2006) Welcome to Apache Hadoop!. http://hadoop.apache.org. Accessed 26 Mar 2018

Brinkhoff T, Kriegel HP, Seeger B (1996) Parallel processing of spatial joins using r-trees. In: Proceedings of the 12th international conference on data engineering, New Orleans, Louisiana, pp 258–265

Chang F, Dean J, Ghemawat S, Hsieh W, Wallach DA, Burrows M, Chandra T, Fikes A, Gruber RE (2008) Bigtable: a distributed storage system for structured data. ACM Trans Comput Syst 26(2). https://doi.org/10.1145/1365815.1365816

Chang WY, Abu-Amara H, Sanford JF (2010) Transforming Enterprise Cloud Services. Springer, London, pp 55–56

Dean J, Ghemawat S (2008) MapReduce: simplified data processing on large clusters. Commun ACM 51(1):107–113. https://doi.org/10.1145/1327452.1327492

DeWitt D, Gray J (1992) Parallel database systems: the future of high performance database systems. Commun ACM 35(6). https://doi.org/10.1145/129888.129894

DeWitt DJ, Gerber RH, Graefe G, Heytens ML, Kumar KB, Muralikrishna M (1986) GAMMA – a high performance dataflow database machine. In: Proceedings of the 12th international conference on very large data bases (VLDB '86), Kyoto, Japan, pp 228–237

Du Z, Zhao X, Ye X, Zhou J, Zhang F, Liu R (2017) An effective high-performance multiway spatial join algorithm with spark. ISPRS Int J Geo-Information 6(4):96

Eldawy A, Mokbel MF (2015) SpatialHadoop: a mapreduce framework for spatial data. In: IEEE 31st international conference on data engineering (ICDE), Seoul, South Korea, pp 1352–1363

Ester M, Kriegel HP, Sander J, Xu X (1996) A density-based algorithm for discovering clusters in large spatial databases with noise. In: Proceedings of the 2nd international conference on knowledge discovery and data mining (KDD-96), Portland, Oregon, pp 226–231

Garillot F, Maas G (2018) Stream processing with apache spark: best practices for scaling and optimizing Apache spark. O'Reilly Media, Sebastopol. http://shop.oreilly.com/product/0636920047568.do

Gedik B, Andrade H, Wu K-L, Yu PS, Doo M (2008) SPADE: the system s declarative stream processing engine. In: Proceedings of the 2008 ACM SIGMOD international conference on management of data (SIGMOD '08), pp 1123–1134. https://doi.org/10.1145/1376616.1376729

Ghemawat S, Gobioff H, Leung S (2003) The Google file system. In: Proceedings of the 19th ACM symposium on operating systems principles, Oct 2003, pp 29–43. https://doi.org/10.1145/945445.945450

Grossman M, Sarkar, V (2016) SWAT: a programmable, in-memory, distributed, high-performance computing platform. In: Proceedings of the 25th ACM international symposium on high-performance parallel and distributed computing (HPDC '16). ACM, New York, pp 81–92. https://doi.org/10.1145/2907294.2907307

Hagedorn S, Götze P, Sattler KU (2017) The STARK framework for spatio-temporal data analytics on spark. In: Proceedings of the 17th conference on database systems for business, technology, and the web (BTW 2017), Stuttgart

Hassaan M, Elghandour I (2016) A real-time big data analysis framework on a CPU/GPU heterogeneous cluster: a meteorological application case study. In: Proceedings of the 3rd IEEE/ACM international conference on big data computing, applications and tech-

nologies (BDCAT '16). ACM, New York, pp 168–177. https://doi.org/10.1145/3006299.3006304
Hong S, Choi W, Jeong W-K (2017) GPU in-memory processing using spark for iterative computation. In: Proceedings of the 17th IEEE/ACM international symposium on cluster, cloud and grid computing (CCGrid '17), pp 31–41. https://doi.org/10.1109/CCGRID.2017.41
Hughes JN, Annex A, Eichelberger CN, Fox A, Hulbert A, Ronquest M (2015) Geomesa: a distributed architecture for spatio-temporal fusion. In: Proceedings of SPIE defense and security. https://doi.org/10.1117/12.2177233
Jacox EH, Samet H (2007) Spatial join techniques. ACM Trans Database Syst 32(1):7
Klein J, Buglak R, Blockow D, Wuttke T, Cooper B (2016) A reference architecture for big data systems in the national security domain. In: Proceedings of the 2nd international workshop on BIG data software engineering (BIGDSE '16). https://doi.org/10.1145/2896825.2896834
Marz N, Warren J (2015) Big data: principles and best practices of scalable realtime data systems, 1st edn. Manning Publications, Greenwich
McInnes L, Healy J (2017) Accelerated hierarchical density based clustering. In: IEEE international conference on data mining workshops (ICDMW), New Orleans, Louisiana, pp 33–42
Mysore D, Khupat S, Jain S (2013) Big data architecture and patterns. IBM, White Paper, 2013. http://www.ibm.com/developerworks/library/bdarchpatterns1. Accessed 26 Mar 2018
NoSQL (2009) NoSQL definition. http://nosql-database.org. Accessed 26 Mar 2018
Pavlo A, Aslett M (2016) What's really new with NewSQL? SIGMOD Rec 45(2):45–55. https://doi.org/10.1145/3003665.3003674
Pedregosa F, Varoquaux G, Gramfort A, Michel V, Thirion B, Grisel O, Blondel M, Prettenhofer P, Weiss R, Dubourg V, Vanderplas J, Passos A, Cournapeau D, Perrot M, Duchesnay É (2011) Scikit-learn: machine learning in Python. J Mach Learn Res 12:2825–2830
Prasad S, McDermott M, Puri S, Shah D, Aghajarian D, Shekhar S, Zhou X (2015) A vision for GPU-accelerated parallel computation on geo-spatial datasets. SIGSPATIAL Spec 6(3):19–26. https://doi.org/10.1145/2766196.2766200
Sakr S, Liu A, Fayoumi AG (2013) The family of mapreduce and large-scale data processing systems. ACM Comput Surv 46(1):1. https://doi.org/10.1145/2522968.2522979
Sena B, Allian AP, Nakagawa EY (2017) Characterizing big data software architectures: a systematic mapping study. In: Proceeding of the 11th Brazilian symposium on software components, architectures, and reuse (SBCARS '17). https://doi.org/10.1145/3132498.3132510
Shekhar S, Gunturi V, Evans MR, Yang KS. 2012. Spatial big-data challenges intersecting mobility and cloud computing. In: Proceedings of the eleventh ACM international workshop on data engineering for wireless and mobile access (MobiDE '12), pp 1–6. https://doi.org/10.1145/2258056.2258058
Shvachko K, Kuang H, Radia S, Chansler R (2010) The Hadoop distributed file system. In: Proceedings of the 2010 IEEE 26th symposium on mass storage systems and technologies (MSST). https://doi.org/10.1109/MSST.2010.5496972
Sriharsha R (2015) Magellan: geospatial analytics on spark. https://hortonworks.com/blog/magellan-geospatial-analytics-in-spark/. Accessed June 2017
Tang M, Yu Y, Malluhi QM, Ouzzani M, Aref WG (2016) LocationSpark: a distributed in-memory data management system for big spatial data. Proc VLDB Endow 9(13):1565–1568. https://doi.org/10.14778/3007263.3007310
Whitman RT, Park MB, Ambrose SM, Hoel EG (2014) Spatial indexing and analytics on Hadoop. In: Proceedings of the 22nd ACM SIGSPATIAL international conference on advances in geographic information systems (SIGSPATIAL '14), pp 73–82. https://doi.org/10.1145/2666310.2666387
Whitman RT, Park MB, Marsh BG, Hoel EG (2017) Spatio-temporal join on Apache spark. In: Hoel E, Newsam S, Ravada S, Tamassia R, Trajcevski G (eds) Proceedings of the 25th ACM SIGSPATIAL international conference on advances in geographic information systems (SIGSPATIAL'17). https://doi.org/10.1145/3139958.3139963
Xie D, Li F, Yao B, Li G, Zhou L, Guo M (2016) Simba: efficient in-memory spatial analytics. In: Proceedings of the 2016 international conference on management of data (SIGMOD '16), pp 1071–1085. https://doi.org/10.1145/2882903.2915237
You S, Zhang J, Gruenwald L (2015) Large-scale spatial join query processing in Cloud. In: 2015 31st IEEE international conference on data engineering workshops, Seoul, 13–17 April 2015, pp 34–41
Yu J, Wu J, Sarwat M (2015) Geospark: a cluster computing framework for processing large-scale spatial data. In: Proceedings of the 23rd SIGSPATIAL international conference on advances in geographic information systems, Seattle, WA
Yuan Y, Salmi MF, Huai Y, Wang K, Lee R, Zhang X (2016) Spark-GPU: an accelerated in-memory data processing engine on clusters. In: Proceedings of the 2016 IEEE international conference on big data (Big Data 2016), Washington, DC, pp 273–283
Zaharia M, Chowdhury M, Franklin MJ, Shenker S, Stoica I (2010) Spark: cluster computing with working sets. In: Proceedings of the 2nd USENIX conference on hot topics in cloud computing (HotCloud'10), Boston, MA
Zhang S, Han J, Liu Z, Wang K, Xu Z (2009) SJMR: parallelizing spatial join with mapreduce on clusters. In: IEEE international conference on Cluster computing (CLUSTER'09), New Orleans, Louisiana, pp 1–8

Artifact-Centric Process Mining

Dirk Fahland
Eindhoven University of Technology,
Eindhoven, The Netherlands

Synonyms

Multi-instance process mining; Object-centric process mining

Definitions

Artifact-centric process mining is an extension of classical process mining (van der Aalst 2016) that allows to analyze event data with more than one case identifier in its entirety. It allows to analyze the dynamic behavior of (business) processes that create, read, update, and delete multiple data objects that are related to each other in relationships with one-to-one, one-to-many, and many-to-many cardinalities. Such event data is typically stored in relational databases of, for example, Enterprise Resource Planning (ERP) systems (Lu et al. 2015). Artifact-centric process mining comprises artifact-centric process discovery, conformance checking, and enhancement. The outcomes of artifact-centric process mining can be used for documenting the actual data flow in an organization and for analyzing deviations in the data flow for performance and conformance analysis.

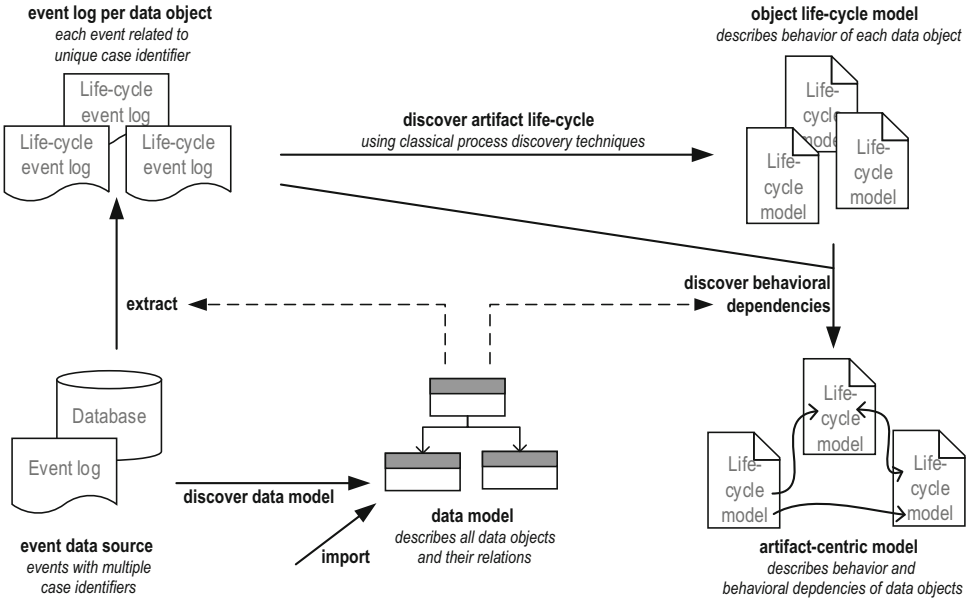
The input to *artifact-centric process discovery* is either an event log where events carry information about the data objects and their changes, or a relational database also containing records about data creation, change, and deletion events. The output of artifact-centric process discovery is a data model of the objects (each defining its own case identifier) and relations between objects and an artifact-centric process model, as illustrated in Fig. 1. An artifact-centric process model describes the dynamics of each data object on its own in an *object life-cycle model*, and the *behavioral dependencies between the different data objects*. To this end, artifact-centric process discovery integrates the control-flow analysis of event data of classical process mining with an analysis of the data structures and data records related to the events.

During artifact-centric process discovery, each event is associated with one data object in the data source. From the behavioral relations between all events associated with one data object, a *life-cycle model* of the data object is learned using automated process discovery techniques. Each life-cycle model describes the possible changes to the object and their ordering as they have been observed in reality. From behavioral relationships between events in different related data objects, information about *behavioral dependencies between changes in different data objects* is discovered preserving the one-to-one, one-to-many, and many-to-many cardinalities.

Several modeling languages have been proposed to describe a complete artifact-centric model of all object life cycles and their behavioral interdependencies. Existing behavioral modeling languages can be extended to express interdependencies of one-to-many and many-to-many cardinalities including Petri nets (van der Aalst et al. 2001) and UML (Estañol et al. 2012). Specifically designed languages including the Guard-Stage-Milestone (GSM) model (Hull et al. 2011) or data-centric dynamic systems (Hariri et al. 2013) employ both data and behavioral constructs as primary modeling concepts. The Case Management Model and Notation (CMMN) standard v1.1 incorporates several modeling concepts of GSM (OMG 2016).

Artifact-centric conformance checking compares event data to an artifact-centric model with the aim to identify where recorded events *deviate* from the behavior described in the artifact-centric model. Deviations may exist between observed and specified data models, between observed events and the life-cycle model of an artifact, and between observed events and the interactions of two or more artifacts.

Artifact-centric model enhancement uses event data to enrich an artifact-centric model, for example, with information about the frequency of



Artifact-Centric Process Mining, Fig. 1 Overview on artifact-centric process discovery

paths through a life-cycle model or interactions, or to identify infrequent behavior as outliers.

Overview

Historically, artifact-centric process mining addressed the unsolved problem of process mining on event data with multiple case identifiers and one-to-many and many-to-many relationships by adopting the concept of a (business) *artifact* as an alternative approach to describing business processes.

Event Data with Multiple Case Identifiers

Processes in organizations are typically supported by information systems to structure the information handled in these processes in well-defined data objects which are often stored in relational databases. During process execution, various data objects are created, read, updated, and deleted, and various data objects are related to each other in one-to-one, one-to-many, and many-to-many relations. Figure 2 illustrates in a simplified form the data structures typically found in ERP systems. *Sales*, *Delivery*, and *Billing* documents are recorded in tables; relation *F1* links *Sales* to *Delivery* documents in a one-to-many relation: *S1* relates to *D1* and *D2*; correspondingly *F2* links *Billing* to *Delivery* documents in a one-to-many relation. Events on process steps and data access are recorded in time stamp attributes such as *Date created* or in a separate *Document Changes* table linked to all other tables.

Convergence and Divergence

Process mining requires to associate events to a *case identifier* in order to analyze behavioral relations between events in the same case (van der Aalst 2016). The data in Fig. 2 provides *three* case identifiers: *SD id*, *DD id*, and *BD id*. Classical process mining forces to associate all events to a single case identifier. However, this is equivalent to flattening and de-normalizing the relational structure along its one-to-many relationships.

For example, associating all *Create* events of Fig. 2 to *SD id* flattens the tables into the *event log* of Fig. 3(left) having two cases for *S1* and *S2*. Due to flattening, event *Create* of *B2* has been duplicated as it was extracted once for *S1* and once for *S2*, also called *divergence*. Further,

Sales Documents

SD id	Date created	Value	Last Change
S1	16-5-2020	100	10-6-2020
S2	17-5-2020	200	5-6-2020

Billing Documents

BD id	Date created	Clearing Date
B1	20-5-2020	31-5-2020
B2	24-5-2020	5-6-2020

F1 F2

Delivery Documents

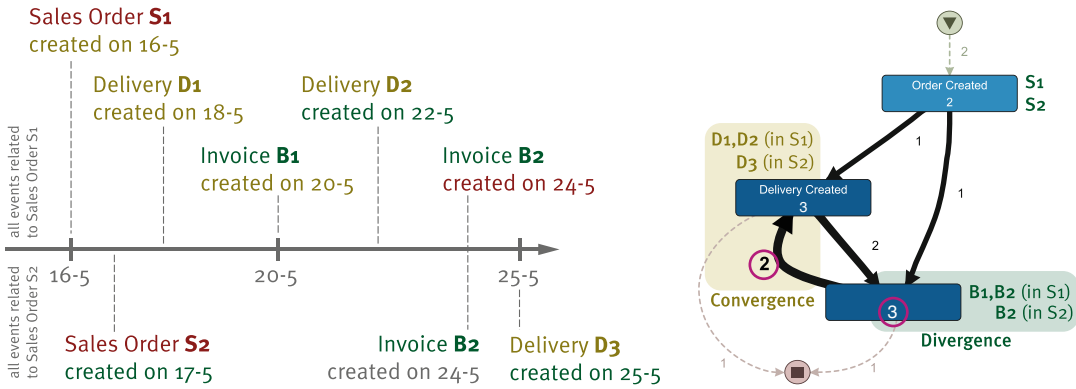
DD id	Date created	Reference SD id	Reference BD id	Picking Date
D1	18-5-2020	S1	B1	31-5-2020
D2	22-5-2020	S1	B2	5-6-2020
D3	25-5-2020	S2	B2	5-6-2020

F3

Document Changes

Change id	Date	Ref. id	Table	Change type	Old Value	New Value
1	17-5-2020	S1	SD	Price updated	100	80
2	19-5-2020	S1	SD	Delivery block released	X	-
3	19-5-2020	S1	SD	Billing block released	X	-
4	10-6-2020	B1	BD	Invoice date updated	20-6-2020	21-6-2020

Artifact-Centric Process Mining, Fig. 2 Event data stored in a relational database



Artifact-Centric Process Mining, Fig. 3 An event log (left) serializing the "create" events of the database of Fig. 2 based on the case identifier "SD id." The resulting directly-follows relation (right) suffers convergence and divergence

Create for *B1* is followed by *Create* for *D2* although *B1* and *D2* are unrelated in Fig. 2, also called *convergence* (Lu et al. 2015). The behavioral relations which underly automated process discovery become erroneous through convergence and divergence. For example, the directly-follows relation of the log (Fig. 3 right) states erroneously that three *Invoice* documents have been created – whereas the original data source contains only two – and that in two cases *Invoice* creation was followed by *Delivery* creation (between related data objects), whereas in the original data source this only happened once for *B2* and *D3*. Convergence and divergence may cause up to 50% of erroneous behavioral relations (Lu et al. 2015) in event logs. Convergence and divergence can be avoided partially by scoping extraction of event data into event logs with a single case identifier in a manual process (Jans 2017).

Artifact-Centric Process Models

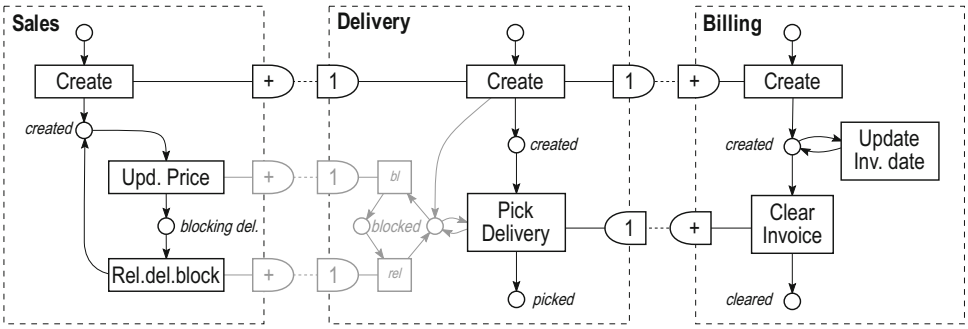
Artifact-centric process mining adopts modeling concept of a (business) artifact to analyze event data with multiple case identifiers in their entirety (Lu et al. 2015; Nooijen et al. 2012; van Eck et al. 2017). The notion of a (business) *artifact*

was proposed by Nigam and Caswell (2003) as an alternative approach to describing business processes. This approach assumes that any process materializes itself in the (data) objects that are involved in the process, for instance, sales documents and delivery documents; these objects have properties such as the values of the fields of a paper form, the processing state of an order, or the location of a package. Typically, a *data model* describes the (1) classes of objects that are relevant in the process, (2) the relevant properties of these objects in terms of class attributes, and (3) the relations between the classes. A process execution instantiates new objects and changes their properties according to the process logic. Thereby, the relations between classes describe how many objects of one class are related to how many objects of another class.

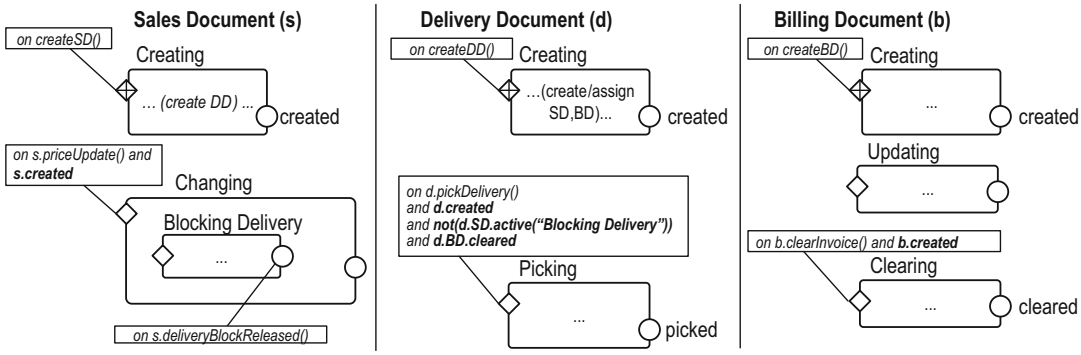
An *artifact-centric process model* enriches the classes of the data model themselves with process logic restricting how objects may evolve during execution. More precisely, one *artifact* (1) encapsulates several classes of the data model (e.g., *Sales Documents* and *Sales Document Lines*), (2) provides *actions* that can update the classes attributes and move the artifact to a particular state, and (3) defines a *life cycle*. The artifact life cycle describes when an instance of the artifact (i.e., a concrete object) is created, in which state of the instance which actions may occur to advance the instance to another state (e.g., from *created* to *cleared*), and which goal state the instance has to reach to complete a case. A complete artifact-centric process model provides a life-cycle model for each artifact in the process and describes which *behavioral dependencies* exist between actions and states of different artifacts (e.g., *pick delivery* may occur for a *Delivery* object only if all its *Billing* objects are in state *cleared*). Where business process models created in languages such as BPMN, EPC, or Petri nets describe a process in terms of activities and their ordering in a single case, an artifact-centric model describes process behavior in terms of creation and evolution of instances of multiple related data objects. In an artifact-centric process model, the unit of modularization is the *artifact*, consisting of data and behavior, whereas in an activity-centric process modeling notation, the unit of modularization is the *activity*, which can be an elementary task or a sub-process. A separate entry in this encyclopedia discusses the problem of automated discovery of activity-centric process models with sub-processes.

Figure 4 shows an artifact-centric process model in the notation of *Proclets* (van der Aalst et al. 2001) for the database of Fig. 2. The life cycle of each document (*Sales*, *Delivery*, *Billing*) is described as a Petri net. Behavioral dependencies between actions in different objects are described through *interface ports* and *asynchronous channels* that also express cardinalities in the interaction. For example, the port annotation "+" specifies that *Clear Invoice* in *Billing* enables *Pick Delivery* in *multiple related Delivery* objects. Port annotation "1" specifies that *Pick Delivery* can occur after *Clear Invoice* occurred in the *one Billing* object related to the *Delivery*.

The gray part of Fig. 4 shows a more involved behavioral dependency. Whenever *Update Price*



Artifact-Centric Process Mining, Fig. 4 Example of an artifact-centric process model in the proclet notation



Artifact-Centric Process Mining, Fig. 5 Example of an artifact-centric process model in the Guard-Stage-Milestone notation

occurs in a *Sales* document, all related *Delivery* documents get *blocked*. Only after *Release delivery block* occurred in *Sales*, the *Delivery* document may be updated again, and *Pick Delivery* may occur. For the sake of simplicity, the model does not show further behavioral dependencies such as "*Update price* also blocks related *Billing* documents."

Figure 5 shows the same model in the *Guard-Stage-Milestone* notation (Hull et al. 2011) (omitting some details). Each round rectangle denotes a *stage* that can be entered when its *guard* condition (diamond) holds and is left when its *milestone* condition (circle) holds. The guard and milestone conditions specify declarative constraints over data attributes, stages, and milestones of *all* artifacts in the model. For example, *Picking* can start when the *pickDelivery* event is triggered in the process, the delivery document has reached its *created* milestone, the billing document related to the delivery document has reached its *cleared* milestone (*d.BD.cleared*), and the stage *Blocking Delivery* is not active in the related sales document.

Artifact-Centric Process Mining

The behavior recorded in the database of Fig. 2 does not *conform* to the models in Figs. 4 and 5:

1. *Structural conformance* states how well the data model describes the data records observed in reality. The proclet model of Fig. 4 structurally conforms to the data in Fig. 2 regarding objects and relations but not regarding actions: the recorded event data shows two additional event types for the life cycle of the *Sales* document – *Release billing block* and *Last Change*.
2. *Life-cycle conformance* states how well the life-cycle model of each artifact describes the order of events observed in reality. This corresponds to conformance in classical process mining. For example, in Fig. 2, *Update invoice date* occurs in *Billing* after *Clear Invoice* which does not conform to the life-cycle model in Fig. 4.
3. *Interaction conformance* states how well the entire artifact centric model describes the behavioral dependencies between artifacts. In Fig. 2, instance *D3* of *Delivery* is *created* after its related instance *B2* of *Billing*. This does not conform to the channels and ports specified in Fig. 4.

The objective of artifact-centric process mining is to relate recorded behavior to modeled behavior, through (1) *discovering* an artifact-centric process model that conforms to the recorded behavior, (2) *checking* how well recorded behavior and an artifact-centric model *conform* to each other and detecting *deviations*, and (3) extending a given artifact-centric model with further information based on recorded event data.

Artifact-centric process discovery is a technique to automatically or semiautomatically learn artifact-centric process models from event data.

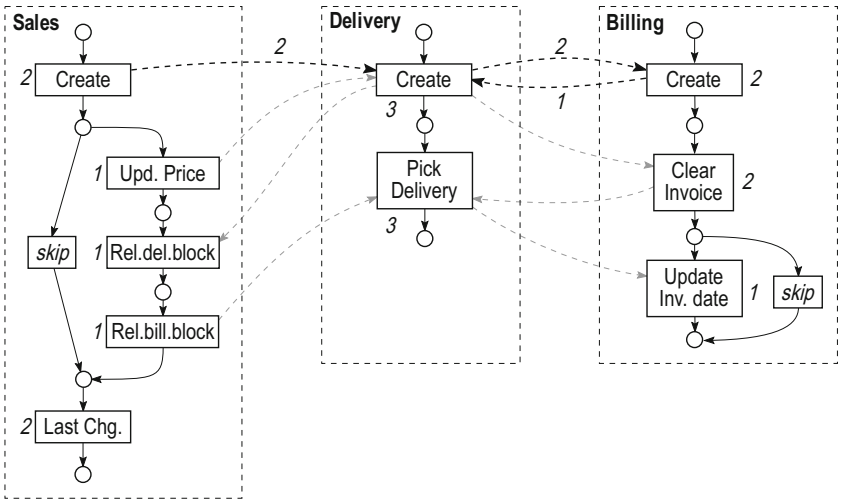
The problem is typically solved by a decomposition into the following four steps:

1. Discovering the data model of entities or tables, their attributes, and relations from the data records in the source data. This step corresponds to *data schema recovery*. It can be omitted if the data schema is available and correct; however, in practice foreign key relations may not be documented at the data level and need to be discovered.
2. Discovering artifact types and relations from the data model and the event data. This step corresponds to transforming the data schema discovered in step 1 into a domain model often involving undoing horizontal and vertical (anti-) partitioning in the technical data schema and grouping entities into domain-level data objects. User input or detailed information about the domain model are usually required.
3. Discovering artifact life-cycle models for each artifact type discovered in step 2. This step corresponds to automated process discovery for event data with a single case identifier and can be done fully automatically up to parameters of the discovery algorithm.
4. Discovering behavioral dependencies between the artifact life cycles discovered in step 3 based on the relations between artifact types discovered in step 2. This step is specific to artifact-centric process mining; several alternative, automated techniques have been proposed. User input may be required to select domain-relevant behavioral dependencies among the discovered ones.

In case the original data source is a relational database, steps 3 and 4 require to *automatically extract event logs* from the data source for discovering life-cycle models and behavioral dependencies. As in classical process discovery, it depends on the use case to which degree the discovered data model, life-cycle model, and behavioral dependencies shall conform to the original data.

Artifact-centric conformance checking and *Artifact-centric model enhancement* follow the same problem decomposition into data schema, artifact types, life cycles, and interactions as artifact-centric discovery. Depending on which models are available, the techniques may also be combined by first discovering data schema and artifact types, then extracting event logs, and then checking life-cycle and behavioral conformance for an existing model or enhancing an existing artifact model with performance information.

Figure 6 shows a possible result of artifact-centric process discovery on the event data in



Artifact-Centric Process Mining, Fig. 6 Possible result of artifact-centric process discovery from the event data in Fig. 2

Fig. 2 using the technique of Lu et al. (2015) where the model has been enhanced with information about the *frequencies* of occurrences of events and behavioral dependencies.

Key Research Findings

Artifact-type discovery. Nooijen et al. (2012) provide a technique for automatically discovering artifact types from a relational database, leveraging schema summarization techniques to cluster tables into artifact types based on information entropy in a table and the strength of foreign key relations. The semiautomatic approach of Lu et al. (2015) can then be used to refine artifact types and undo horizontal and vertical (anti-) partitioning and to discover relations between artifacts. Popova et al. (2015) show how to discover artifact types from a rich event stream by grouping events based on common identifiers into entities and then deriving structural relations between them.

Event log extraction. In addition to discovering artifact types, Nooijen et al. (2012) also automatically create a mapping from the relational database to the artifact-type specification. The technique of Verbeek et al. (2010) can use this mapping to generate queries for event log extraction for life-cycle discovery automatically. Jans (2017) provides guidelines for extracting specific event logs from databases through user-defined queries. The event log may also be extracted from database redo logs using the technique of de Murillas et al. (2015) and from databases through a meta-model-based approach as proposed by de Murillas et al. (2016).

Life-cycle discovery. Given the event log of an artifact, artifact life-cycle discovery is a classical automated process discovery problem for which various process discovery algorithms are available, most returning models based on or similar to Petri nets. Weerdt et al. (2012) compared various discovery algorithms using real-life event logs. Lu et al. (2015) advocates the use of the Heuristics Miner of Weijters and Ribeiro (2011) and vanden Broucke and Weerdt (2017) with the aim of visual analytics. Popova et al. (2015) advocate to discover models with precise semantics and free of behavioral anomalies that (largely) fit the event log (Leemans et al. 2013; Buijs et al. 2012) allowing for translating the result to the Guard-Stage-Milestone notation.

Behavioral dependencies. Lu et al. (2015) discover behavioral dependencies between two artifacts by extracting an *interaction event log* that combines the events of any two related artifact instances into one trace. Applying process discovery on this interaction event log then allows to extract "flow edges" between activities of the different artifacts, also across one-to-many relations, leading to a model as shown in Fig. 6. This approach has been validated to return only those dependencies actually recorded in the event data but suffers when interactions can occur in many different variants, leading to many different "flow edges."

van Eck et al. (2017) generalize the interaction event log further and create an integrated event log of all artifact types to be considered (two or more) where for each combination of related artifact instances, all events are merged into a single trace. From this log, a composite state machine model is discovered which describes the synchronization of all artifact types. By projecting the composite state machine onto the steps of each artifact type, the life-cycle model for each artifact is obtained, and the interaction between multiple artifacts can be explored interactively in a graphical user interface through their relation in the composite state machine. This approach assumes one-to-one relations between artifacts.

Popova and Dumas (2013) discover behavioral dependencies in the form of data conditions over data attributes and states of other artifacts, similar to the notation in Fig. 5 but is limited to one-to-one relations between artifacts.

Conformance checking. Artifact life-cycle conformance can be checked through extracting artifact life-cycle event logs and then applying classical conformance checking techniques (Fahland

et al. 2011a). The technique in Fahland et al. (2011b) checks interaction conformance in an artifact life-cycle model if detailed information about which artifact instances interact is recorded in the event data.

Models for artifacts. Artifact-centric process mining techniques originated and are to a large extent determined by the modeling concepts available to describe process behavior and data flow with multiple case identifiers. Several proposals have been made in this area. The Proclet notation (van der Aalst et al. 2001) extended Petri nets with ports that specify one-to-many and many-to-many cardinality constraints on messages exchanged over channels in an asynchronous fashion. Fahland et al. (2011c) discuss a normal form for proclet-based models akin to the second normal form in relational schemas. The Guard-Stage-Milestone (GSM) notation (Hull et al. 2011) allows to specify artifacts and interactions using event-condition-actions rules over the data models of the different artifacts. Several modeling concepts of GSM were adopted by the CMMN 1.1 standard of OMG (2016). Hariri et al. (2013) propose data-centric dynamic systems (DCDS) to specify artifact-centric behavior in terms of updates of database records using logical constraints. Existing industrial standards can also be extended to describe artifacts as shown by Lohmann and Nyolt (2011) for BPMN and by Estañol et al. (2012) for UML. Freedom of behavioral anomalies can be verified for UML-based models (Calvanese et al. 2014) and for DCDS (Montali and Calvanese 2016). Meyer and Weske (2013) show how to translate between artifact-centric and activity-centric process models, and Lohmann (2011) shows how to derive an activity-centric process model describing the interactions between different artifacts based on behavioral constraints.

Examples of Application

Artifact-centric process mining is designed for analyzing event data where events can be related to more than one case identifier or object and where more than one case identifier has to be considered in the analysis.

The primary use case is in analyzing processes in information systems storing multiple, related data objects, such as Enterprise Resource Planning (ERP) systems. These systems store documents about business transactions that are related to each other in one-to-many and many-to-many relations. Lu et al. (2015) correctly distinguish normal and outlier flows between 18 different business objects over 2 months of data of the Order-to-Cash process in an SAP ERP system using artifact-centric process mining. The same technique was also used for identifying outlier behavior in processes of a project management system together with end users. van Eck et al. (2017) analyzed the personal loan and overdraft process of a Dutch financial institution. Artifact-centric process mining has also been applied successfully on software project management systems such as Jira and customer relationship management systems such as Salesforce (Calvo 2017).

Artifact-centric process mining can also be applied on event data outside information systems. One general application area is analyzing the behavior of physical objects as sensed by multiple related sensors. For instance, van Eck et al. (2016) analyzed the usage of physical objects equipped with multiple sensors. Another general application area is analyzing the behavior of software components from software execution event logs. For instance, Liu et al. (2016) follow the artifact-centric paradigm to structure events of software execution logs into different components and discover behavioral models for each software component individually.

Future Directions for Research

At the current stage, artifact-centric process mining is still under development allowing for several directions for future research.

Automatically discovering artifact types from data sources is currently limited to summarizing the structures in the available data. Mapping these

structures to domain concepts still requires user input. Also the automated extraction of event logs from the data source relies on the mapping from the data source to the artifact-type definition. How to aid the user in discovering and mapping the data to domain-relevant structures and reducing the time and effort to extract event logs, possibly through the use of ontologies, is an open problem. Also little research has been done for improving the queries generated for automated event log extraction to handle large amount of event data.

For discovering behavioral dependencies between artifacts, only few and limited techniques are available. The flow-based discovery of Lu et al. (2015) that can handle one-to-many relations is limited to interactions between two artifacts and suffers in the presence of many different behavioral variants of the artifacts or the interactions. The alternative approaches (Popova and Dumas 2013; van Eck et al. 2017) are currently limited to one-to-one relations between artifacts. Solving the discovery of behavioral dependencies between artifacts thereby faces two fundamental challenges:

1. Although many different modeling languages and concepts for describing artifact-centric processes have been proposed, the proposed concepts do not adequately capture these complex dynamics in an easy-to-understand form (Reijers et al. 2015). Further research is needed to identify appropriate modeling concepts for artifact interactions.
2. Systems with multiple case identifiers are in their nature complex systems, where complex behaviors and multiple variants in the different artifacts multiply when considering artifact interactions. Further research is needed on how to handle this complexity, for example, through generating specific, interactive views as proposed by van Eck et al. (2017).

Although several, comprehensive conformance criteria in artifact-centric process mining have been identified, only behavioral conformance of artifact life cycles can currently be measured. Further research for measuring structural conformance and interaction conformance is required, not only for detecting deviations but also to objectively evaluate the quality of artifact-centric process discovery algorithms.

Cross-References

- ▸ Automated Process Discovery
- ▸ Business Process Analytics
- ▸ Conformance Checking
- ▸ Hierarchical Process Discovery
- ▸ Multidimensional Process Analytics
- ▸ Schema Mapping

References

Buijs JCAM, van Dongen BF, van der Aalst WMP (2012) A genetic algorithm for discovering process trees. In: IEEE congress on evolutionary computation

Calvanese D, Montali M, Estañol M, Teniente E (2014) Verifiable UML artifact-centric business process models. In: CIKM

Calvo HAS (2017) Artifact-centric log extraction for cloud systems. Master's thesis, Eindhoven University of Technology

de Murillas EGL, van der Aalst WMP, Reijers HA (2015) Process mining on databases: unearthing historical data from redo logs. In: BPM

de Murillas EGL, Reijers HA, van der Aalst WMP (2016) Connecting databases with process mining: a meta model and toolset. In: BMMDS/EMMSAD

Estañol M, Queralt A, Sancho MR, Teniente E (2012) Artifact-centric business process models in UML. In: Business process management workshops

Fahland D, de Leoni M, van Dongen BF, van der Aalst WMP (2011a) Behavioral conformance of artifact-centric process models. In: BIS

Fahland D, de Leoni M, van Dongen BF, van der Aalst WMP (2011b) Conformance checking of interacting processes with overlapping instances. In: BPM

Fahland D, de Leoni M, van Dongen BF, van der Aalst WMP (2011c) Many-to-many: some observations on interactions in artifact choreographies. In: ZEUS

Hariri BB, Calvanese D, Giacomo GD, Deutsch A, Montali M (2013) Verification of relational data-centric dynamic systems with external services. In: PODS

Hull R, Damaggio E, Masellis RD, Fournier F, Gupta M, Heath FT, Hobson S, Linehan MH, Maradugu S, Nigam A, Sukaviriya N, Vaculín R (2011) Business artifacts with guard-stage-milestone lifecycles: managing artifact interactions with conditions and events. In: DEBS

Jans M (2017) From relational database to event log: decisions with quality impact. In: First international workshop on quality data for process analytics
Leemans SJJ, Fahland D, van der Aalst WMP (2013) Discovering block-structured process models from event logs – a constructive approach. In: Petri nets
Liu CS, van Dongen BF, Assy N, van der Aalst WMP (2016) Component behavior discovery from software execution data. In: 2016 IEEE symposium series on computational intelligence (SSCI), pp 1–8
Lohmann N (2011) Compliance by design for artifact-centric business processes. Inf Syst 38(4): 606–618
Lohmann N, Nyolt M (2011) Artifact-centric modeling using BPMN. In: ICSOC workshops
Lu X, Nagelkerke M, van de Wiel D, Fahland D (2015) Discovering interacting artifacts from ERP systems. IEEE Trans Serv Comput 8:861–873
Meyer A, Weske M (2013) Activity-centric and artifact-centric process model roundtrip. In: Business process management workshops
Montali M, Calvanese D (2016) Soundness of data-aware, case-centric processes. Int J Softw Tools Technol Trans 18:535–558
Nigam A, Caswell NS (2003) Business artifacts: an approach to operational specification. IBM Syst J 42: 428–445
Nooijen EHJ, van Dongen BF, Fahland D (2012) Automatic discovery of data-centric and artifact-centric processes. In: Business process management workshops. Lecture notes in business information processing, vol 132. Springer, pp 316–327. https://doi.org/10.1007/978-3-642-36285-9_36
OMG (2016) Case management model and notation, version 1.1. http://www.omg.org/spec/CMMN/1.1
Popova V, Dumas M (2013) Discovering unbounded synchronization conditions in artifact-centric process models. In: Business process management workshops
Popova V, Fahland D, Dumas M (2015) Artifact lifecycle discovery. Int J Coop Inf Syst 24:44
Reijers HA, Vanderfeesten ITP, Plomp MGA, Gorp PV, Fahland D, van der Crommert WLM, Garcia HDD (2015) Evaluating data-centric process approaches: does the human factor factor in? Softw Syst Model 16:649–662
vanden Broucke SKLM, Weerdt JD (2017) Fodina: a robust and flexible heuristic process discovery technique. Decis Support Syst 100:109–118
van der Aalst WMP (2016) Process mining – data science in action, 2nd edn. Springer. https://doi.org/10.1007/978-3-662-49851-4
van der Aalst WMP, Barthelmess P, Ellis CA, Wainer J (2001) Proclets: a framework for lightweight interacting workflow processes. Int J Coop Inf Syst 10: 443–481
van Eck ML, Sidorova N, van der Aalst WMP (2016) Composite state machine miner: discovering and exploring multi-perspective processes. In: BPM
van Eck ML, Sidorova N, van der Aalst WMP (2017) Guided interaction exploration in artifact-centric process models. In: 2017 IEEE 19th conference on business informatics (CBI), vol 1, pp 109–118
Verbeek HMW, Buijs JCAM, van Dongen BF, van der Aalst WMP (2010) Xes, xesame, and prom 6. In: CAiSE forum
Weerdt JD, Backer MD, Vanthienen J, Baesens B (2012) A multi-dimensional quality assessment of state-of-the-art process discovery algorithms using real-life event logs. Inf Syst 37(7):654–676. https://doi.org/10.1016/j.is.2012.02.004
Weijters AJMM, Ribeiro JTS (2011) Flexible heuristics miner (FHM). In: 2011 IEEE symposium on computational intelligence and data mining (CIDM), pp 310–317

Assessment

▸ Auditing

Attestation

▸ Auditing

Attribute-Based Access Control (ABAC)

▸ Security and Privacy in Big Data Environment

Auditing

Francois Raab
InfoSizing, Manitou Springs, CO, USA

Synonyms

Assessment; Attestation; Certification; Review; Validation

Definitions

An examination of the implementation, execution, and results of a test or benchmark, generally

performed by an independent third-party, and resulting in a report of findings

Historical Background

The use of a third-party audit to attest to the veracity of a claim has historically been commonplace in the financial sector. The goal of such audit activities is to bolster the credibility of an organization's claims regarding its financial standing. Similar auditing activities are found in other fields where there is value in validating the level at which a set of requirements have been followed.

As formal definitions of computer systems performance benchmarks started to emerge, so did the call for independent certification of published results. The Transaction Processing Performance Council (TPC – www.tpc.org) was the first industry standard benchmark consortium to formalize the requirement for independent auditing of benchmark results.

Foundations

The purpose of auditing in the context of a performance test or benchmark is to validate that the test results were produced in compliance with the set of requirements defining the test. These requirements are used to define multiple aspects of the test, including what is being tested, how it is being tested, how the test results are measured, and how accurately are they documented. Auditing a test or benchmark result consists in validating some or all of these requirements.

Benchmark requirements can be viewed as belonging to one of the following categories: implementation rules, execution rules, results collection, pricing rules, and documentation. The motivation behind auditing a set of requirements, and the process involved in such validation, is largely based on which of these categories the requirements belong to.

Auditing the Implementation

Some benchmarks are provided in the form of a complete software kit that can simply be installed and executed to measure the underlying system. Other benchmarks are provided in the form of a set of functional requirements to be implemented using any fitting technology. In this later case, the level at which the implementation meets the stated requirements can directly affect the results of the test. Consider a benchmark requiring the execution of two tasks acting on the same data set. An implementation that correctly implements the tasks but fails to have them act on the same data set would avoid potential data access conflicts.

The process of auditing a benchmark implementation includes all aspects of that implementation. This may include reviewing custom code, examining data generation tools and their output, executing functional testing to verify the proper behavior of required features, and validating the integration of the various benchmark components.

Auditing the Execution

Most benchmarks involve the execution of multiple steps, for a prescribed duration and in some specified sequence. The level at which these execution rules are followed can greatly impact the outcome of the test. Consider a benchmark requiring that two tasks be executed concurrently for a specified duration. An execution that runs the tasks for the specified duration but schedules them in a serial manner would avoid potential contention for system resources.

The process of auditing a benchmark execution involves verifying that controls are in place to drive the execution based on the stated rules and that sufficient traces of the execution steps are captured. This may be accomplished by reviewing execution scripts, witnessing the execution in real time, and examining logs that were produced during the execution.

Auditing the Results

The end goal of implementing and executing a benchmark is to produce a result for use in performance engineering, marketing, or other venue. While a benchmark may be implemented correctly and executed according to all stated rules, it may produce a flawed outcome if the results

are not collected properly. Consider a benchmark that involves a workload that gradually ramps up until the system under test reaches saturation, with the goal of measuring the system's behavior at that saturation point. A test that measures the system's behavior too early during ramp-up would avoid the potential disruptions caused by saturation.

The process of auditing the collection of results during a benchmark execution involves verifying that all necessary conditions are met for the measurement to be taken. In cases where metrics are computed from combining multiple measurements, it also involves verifying that all the components of the metric are properly sourced and carry sufficient precision.

Auditing the Pricing

When benchmarks are executed for the purpose of competitive analysis, the cost of the tested configuration may also be part of the benchmark's metrics. Including a cost component in the metrics provides a measure of value, in addition to the measure of performance. This value metric may help differentiate between systems under test with comparable performance metrics. But the price of a SUT can be greatly influenced by the rules regulating the pricing methodology. For instance, the price of a leasing contract may not be comparable to that of an outright purchase, discount levels may vary widely across markets, and maintenance costs vary based on service level agreements.

The process of auditing the pricing of a SUT involved in a benchmark execution consists in verifying that all of the pricing rules defined by the benchmark have been adhered to. It may also involve a verification of the stated prices by attempting to obtain an independent pricing quotation from the vendors or distributors of the components in the SUT.

Auditing the Documentation

Benchmark results are best understood within their proper context. The purpose of documenting a benchmark result is to provide sufficient context to allow a full understanding of the significance of the result. Without proper context, a benchmark result can be misunderstood and lead to incorrect conclusions. Consider a benchmark with a body of existing results that have been produced on single-socket systems. A new, record-breaking result may be received more favorably if the documentation omits to disclose that it was produced on a dual-socket system, leaving the reader to assume that a single socket was used.

The process of auditing the documentation of a benchmark result consists in verifying that all relevant information has been provided with accuracy and candor. A criterion to determine if the documentation of a test result is sufficient is when the information provided allows another party to independently reproduce the result.

Key Applications

First Formal Use

According to Serlin (1993), one of the first formal audit of a benchmark result took place "in March, 1987, when Tandem hired Codd & Date Consulting to certify the 208 tps result obtained under a version of DebitCredit." Tom Sawyer, the auditor on this tandem test, later collaborated with Ormi Serlin to author a proposal for a formal performance test framework that became the impetus for the creation of the TPC.

Around the time of the release of its third standard benchmark specification, TPC Benchmark™ C (Raab 1993), in August 1992, the TPC added to its policies (TPC. TPC Policies) the call for a mandatory audit of all its benchmark results. This addition to its policies also established a process by which the TPC certifies individuals who are qualified to conduct such audits.

Auditing Adoption

The practice of independently auditing benchmark results has been adopted by multiple organizations involved in performance testing. A formal benchmark result auditing requirement was adopted by the Storage Performance Council (SPC – www.storageperformance.org), shortly after its inception in 1998. It is a common practice for system vendors to

publish nonstandard test results in support of performance claims or to document a proof of concept. A number of these publications include an independent certification to bolster the credibility of the results. The practice of independently auditing test results has also been adopted by private and governmental organizations conducting comparative performance testing as part of their technology selection process.

Internal Peer Review

Some standard benchmark organizations, such as the Standard Performance Evaluation Corporation (SPEC – www.spec.org), have opted for a peer review process to validate the benchmark results they publish. This review is conducted internally by members of the organization. The TPC also gives test sponsors the option of a peer review process for results against its benchmarks in a category called "express." In both of these cases of internal peer review, test results from one vendor are reviewed by competing vendors. The strength of this review process is in the assumption that the reviewers are intimately familiar with the testing requirements and the tested technology by having conducted similar tests themselves. The downside is in the potential for a loss of independence in the validation process.

Independent Third-Party Auditor

To maintain its integrity, auditing is best when conducted by an independent third party that has no conflict of interest with the object of the audit. These conflicts can take many forms in the context of a benchmark audit. Following are a few examples of conflicts of interest that may compromise the independence of validation process: having a vested interest in how the benchmark result may affect the image of a tested product; having a financial stake attached to the outcome of the test; or being directly involved in conducting the test or improving the test results.

Cross-References

- ▸ Big Data Benchmark
- ▸ Big Data Architectures
- ▸ Benchmark Harness
- ▸ Cloud Big Data Benchmarks
- ▸ Component Benchmark
- ▸ End-to-End Benchmark
- ▸ Energy Benchmarking
- ▸ Metrics for Big Data Benchmarks
- ▸ Microbenchmark
- ▸ TPC

References

Raab F (1993) Overview of the TPC benchmark C: a complex OLTP benchmark. In: The benchmark handbook for database and transaction processing systems, 2nd edn. Morgan Kaufmann Publishers, San Mateo, pp 131–144

Serlin O (1993) The history of DebitCredit and the TPC. In: The benchmark handbook for database and transaction processing systems, 2nd edn. Morgan Kaufmann Publishers, San Mateo, pp 21–40

TPC. TPC policies. http://www.tpc.org/tpc_documents_current_versions/current_specifications.asp

Authenticating Anonymized Data with Domain Signatures

- ▸ Privacy-Aware Identity Management

Authentication, Authorization, and Accounting (AAA)

- ▸ Security and Privacy in Big Data Environment

Automated Creation of Infographics

- ▸ Visualizing Semantic Data

Automated Discovery of Hierarchical Process Models

▸ Hierarchical Process Discovery

Automated Process Discovery

Sander J. J. Leemans
Queensland University of Technology, Brisbane, Australia

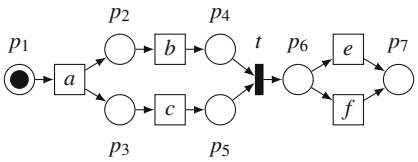
Definitions

An *event log* contains a historical record of the steps taken in a business process. An event log consists of *traces*, one for each case, customer, order, etc. in the process. A trace contains *events*, which represent the steps (*activities*) that were taken for a particular case, customer, order, etc.

An example of an event log derived from an insurance claim handling process is [⟨receive claim, check difficulty, decide claim, notify customer⟩^{10}, ⟨receive claim, check difficulty, check fraud, decide claim, notify customer⟩^{5}]. This event log consists of 15 traces, corresponding to 15 claims made in the process. In 10 of these traces, the claim was received, its difficulty assessed, the claim was decided and the customer was notified.

A *process model* describes the behaviour that can happen in a process. Typically, it is represented as a Petri net (Reisig 1992) or a BPMN model (OMG 2011).

A Petri net consists of places, which denote the states the system can be in, and transitions, which denote the state changes of the system. For instance, Fig. 1 shows an example of a Petri net. This net starts with a *token* in place p_1. *Firing* transition a removes the token from p_1 and puts tokens in p_2 and p_3. This denotes the execution of the activity a in the process. Then, transitions b and c can fire independently, each consuming the token of p_2 or p_3 and producing a token in p_4 or p_5. Next, the silent transition t fires and puts a token in p_6. As t is a silent transition, no corresponding activity is executed in the process. Finally, either e or f can be fired, putting a token in p_7 and ending the process.



Automated Process Discovery, Fig. 1 Example of a Petri net

A *workflow net* is a Petri net with an initial place (without incoming arcs), a final place (without outgoing arcs) and every place and transition lying on a path between these places. The behaviour of a workflow net is clear: a token is put in the initial place, and every sequence of transitions firings that leads to a token in the final place and nowhere else, is a trace of the behaviour of the net.

A workflow net is *sound* if the net is free of deadlocks, unexecutable transitions and other anomalies (van der Aalst 2016). A workflow net is *relaxed sound* if there is a sequence of transition firings that lead to a token in the final place and nowhere else.

Overview

Automated process discovery aims to extract information from recorded historical information about business processes by means of automatic methods. In this chapter, we discuss challenges and algorithms for process discovery.

Automated Process Discovery

Organisations nowadays store considerable amounts of data: in many business processes such as for booking a flight, lodging an insurance claim or hiring a new employee, every step is supported and recorded by an information system. From these information systems, event logs can be extracted, which contain the steps that were taken for a particular customer, booking,

claim, etc. Process mining aims to derive information and insights from these event logs.

Many process mining techniques depend on the availability of a process model. Process models can be elicited by hand, however this can be a tedious and error-prone task. Instead, if event logs are available, these can be used to discover a process model automatically.

In this chapter, the research field of algorithms that automatically discover process models from event logs is described. First, quality criteria for models are discussed, and how algorithms might have to tradeoff between them. Second, process discovery algorithms are discussed briefly.

Quality Criteria & Tradeoffs

The quality of a discovered model can be assessed using several concepts: whether it possesses clear semantics, whether it is simple, how well it represents the event log and how well it represents the process.

Semantics & Soundness

As a first quality criterion, the behaviour described by the model should be clear. That is, it should be clear which traces the model can produce. If the returned model is a Petri net or a BPMN model, this model should be free of deadlocks, unexecutable transitions and other anomalies (it should be *sound* (van der Aalst 2016)). While unsound nets can be useful for manual analysis, they should be used with care in automated analyses as, for instance, conformance checking techniques might give unreliable answers or simply not work on unsound nets. At a bare minimum, conformance checking techniques such as alignments (Adriansyah 2014) require relaxed sound models.

Simplicity

Second, given two models, all other things equal, the simplest model is usually the best of the two (a principle known as Occam's razor). That is, a model should be as understandable as possible, for instance sma.

Log Quality

Third, one can consider the quality of a discovered model with respect to the event log from which it was discovered, to assess whether the model represents the available information correctly. Typically, besides simplicity, three quality dimensions are considered: fitness, precision and generalisation. Fitness expresses the part of the event log that is captured in the behaviour of the process model. Precision expresses the part of the behaviour of the model that is seen in the event log. Generalisation expresses what part of future behaviour will be likely present in the model.

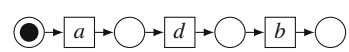
To illustrate these quality measures, consider the following event log L:

$$[\langle a,d,b\rangle^{50}, \quad \langle a,b,c,d,b\rangle^{20},$$
$$\langle a,b,d,c,b,c,b\rangle^{2}, \quad \langle a,b,c,d,b,c,b\rangle^{2},$$
$$\langle a,b,c,b,d,c,b\rangle, \quad \langle a,b,c,b,c,b,d\rangle,$$
$$\langle a,b,c,b,d,c,b,c,b\rangle, \quad \langle a,b,c\rangle]$$

Figure 2 contains a possible process model for L, which supports only a single trace. This model has a poor fitness, as many traces of L are not part of its behaviour. However, it has a high precision, as the single trace it represents was seen in L. Compared to the event log, this model is not very informative.

An extreme model is shown in Fig. 3. This model is a so-called *flower model*, as it allows for all behaviour consisting of a, b, c and d, giving it a low fitness and high precision. Even though this model is simple and certainly generalises, it is completely useless as it does not provide any information besides the presence of a-d in the process.

On the other end of the spectrum is the *trace model*, shown in Fig. 4. This model simply

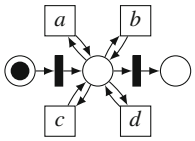


Automated Process Discovery, Fig. 2 A process model with low fitness, high precision, low generalisation and high simplicity w.r.t. L

lists all traces of L, thereby achieving perfect fitness and precision. However, this model does not generalise the behaviour in the event log, that is, it only shows the traces that were seen in L, and does not provide any extra information.

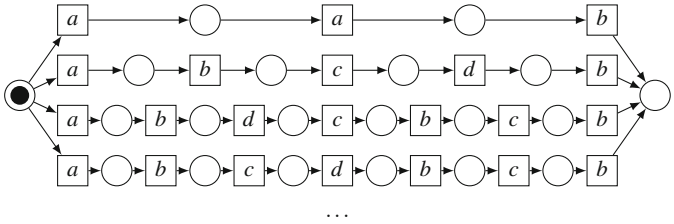
As a final model, we consider the model shown in Fig. 5. This model has a high fitness, precision, generalisation and simplicity. However, the model still does not score *perfect* as the last trace of L, $\langle a, b, c \rangle$, is not captured by this model, which lowers fitness a bit. Furthermore, precision is not perfect as the trace $\langle a, b, c, b, c, d, b \rangle$ is possible in the model but did not appear in L, which lowers precision.

The models shown for L illustrate that process discovery algorithms might have to tradeoff and strike a balance between quality criteria. For some event logs, a model scoring high on all log-quality measures and simplicity might not exist (Buijs et al. 2012b). The necessary balance might depend on the use case at hand. For instance, manual analysis where the "main flow" of a process is sought might require the omittance of the last trace of L from the model, yielding a simple and precise model. However, for auditing purposes, one might opt for a perfectly fitting model by including this last trace of L in the behaviour of the model.



Automated Process Discovery, Fig. 3 A process model ("flower model") with high fitness, low precision, high generalisation and high simplicity w.r.t. L

Automated Process Discovery, Fig. 4 A process model ("trace model") with high fitness, high precision, low generalisation and low simplicity w.r.t. L



Process Quality

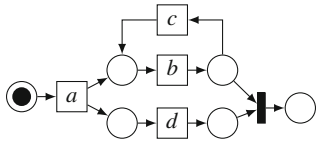
A downside of measuring the quality of a model with respect to the event log is that an event log contains only *examples* of behaviour of an (unknown) business process rather than the full behaviour, and that the log might contain traces that do not correspond to the business process (*noisy* traces). Therefore, one can also consider how it compares to the process from which the event log was recorded. In the ideal case, the behaviour of the process is rediscovered by a discovery algorithm. That is, the behaviour (language) of the model is the same as the behaviour of the process.

As the business process is assumed to be unknown, whether an algorithm can find a model that is behaviourally equivalent to the process (*rediscoverability*) is a formal property of the algorithm. Without rediscoverability, an algorithm is *unable* to find a model equivalent to the process, which makes the algorithm rather unsuitable to study this process.

Rediscoverability is typically proven using assumptions on the process and the event log, for instance that it is representible as a model in the formalism of the algorithm (Petri nets, BPMN), and for instance that the event log contains enough information and does not contain too much noise, as well as assumptions on the process.

Process Discovery Algorithms

In this section, a selection of process discovery algorithms is discussed. For each algorithm, the algorithmic idea is described briefly, as well as some general advantages and disadvantages, and where it can be downloaded.



Automated Process Discovery, Fig. 5 A process model with high fitness, high precision, high generalisation and high simplicity w.r.t. L

For benchmarks and a more exhaustive overview, please refer to Augusto et al. (2017b) (algorithms after 2012) and Weerdt et al. (2012) (algorithms before 2012). Not all algorithms can be benchmarked reliably; the selection here contains all benchmarked algorithms of Augusto et al. (2017b).

Several of these algorithms are available in the ProM framework (van Dongen et al. 2005), which is available for download from http://www.promtools.org, or in the Apromore suite (Rosa et al. 2011), which can be accessed via http://apromore.org.

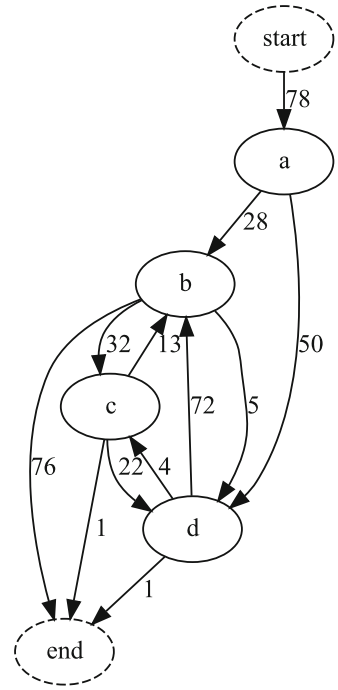
The algorithms are discussed in three stages: firstly, algorithms that do not support concurrency, secondly algorithms that guarantee soundness and thirdly the remaining algorithms.

Directly Follows-Based Techniques

As a first set, techniques based on the directly follows relations are discussed. The section starts with an explanation of directly follows graphs, after which some tools that use this concept are listed and the limitations of such techniques are discussed.

In a directly follows graph, the nodes represent the activities of the event log, and the edges represent that an activity is directly followed by another activity in the event log. Numbers on the edges indicate how often this happened. Additionally, a start and an end node denote the events with which traces in the event log start or end. For instance, Fig. 6 shows the directly follows graph for our event log L.

For more complicated processes, a directly follows graph might get uncomprehensibly complicated. Therefore, discovery techniques typically filter the directly follows graph, for instance by removing little-occurring edges. Commercial techniques that filter and show directly follows graphs include Fluxicon Disco (Fluxicon 2017), Celonis Process Mining (Celonis 2017) and ProcessGold Enterprise Platform (ProcessGold 2017). Another strategy to reduce complexity, applied by the Fuzzy Miner (Günther and van der Aalst 2007), is to cluster similar activities into groups, thereby providing capabilities to zoom in on details of the process (by clustering less), or to abstract to the main flow of the process by clustering more.



Automated Process Discovery, Fig. 6 Directly follows graph of event log L

While these graphs are intuitive, it can be challenging to distinguish repetitive and concurrent behaviour, as both manifest as edges forth- and back between activities. For instance, in Fig. 6, it seems that b, c and d can be executed repeatedly, while in the log L this never happened for d. In contrast, it *also* seems that b, c and d are concurrent, while in L, b and c are always executed repeatedly. Due to this, directly follows graphs tend to have a low precision and high generalisation: in our example, almost any sequence of b, c and d is included.

Nevertheless, directly follows-based techniques are often used to get a first idea of the process behind an event log.

Soundness-Guaranteeing Algorithms

Soundness is a prerequisite for further automated or machine-assisted analysis of business process models. In this section, soundness or relaxed soundness guaranteeing algorithms are discussed.

Evolutionary Tree Miner

To address the issue of soundness, the Evolutionary Tree Miner (ETM) (Buijs et al. 2012a) discovers process trees. A process tree is an abstract hierarchical view of a workflow net and is inherently sound.

ETM first constructs an initial population of models; randomly or from other sources. Second, some models are selected based on fitness, precision, generalisation and simplicity with respect to the event log. Third, the selected models are smart-randomly mutated. This process of selection and mutation is repeated until a satisfactory model is found, or until time runs out.

ETM is flexible as both the selection and the stopping criteria can be adjusted to the use case at hand; one can prioritise (combinations of) quality criteria. However, due to the repeated evaluation of models, on large event logs of complex processes, stopping criteria might force a user to make the decision between speed and quality.

Inductive Miner Family

The Inductive Miner (IM) family of process discovery algorithms, like the Evolutionary Tree Miner, discovers process trees to guarantee that all models that are discovered are sound. The IM algorithms apply a recursive strategy: first, the "most important" behaviour of the event log is identified (such as sequence, exclusive choice, concurrency, loop, etc.). Second, the event log is split in several parts, and these steps are repeated until a base case is encountered (such as a log consisting of a single activity). If no "most important" behaviour can be identified, then the algorithms try to continue the recursion by generalising the behaviour in the log, in the worst case ultimately ending in a flower model.

Besides a basic IM (Leemans et al. 2013a), algorithms exist that focus on filtering noise (Leemans et al. 2013b), handling incomplete behaviour (when the event log misses crucial information of the process) (Leemans et al. 2014a), handling lifecycle information of events (if the log contains information of e.g. when activities started and ended) (Leemans et al. 2015), discovering challenging constructs such as inclusive choice and silent steps (Leemans 2017), and handling very large logs and complex processes (Leemans et al. 2016), all available in the ProM framework.

Several IM-algorithms guarantee to return a model that perfectly fits the event log, and all algorithms are capable of rediscovering the process, assuming that the process can be described as a process tree (with some other restrictions, such as no duplicated activities) and assuming that the event log contains "enough" information. However, due to the focus on fitness, precision tends to be lower on event logs of highly unstructured processes.

All Inductive Miner algorithms are available as plug-ins of the ProM framework, and some as plug-ins of the Apromore framework. Furthermore, the plug-in Inductive visual Miner (Leemans et al. 2014b) provides an interactive way to apply these algorithms and perform conformance checking.

An algorithm that uses a similar recursive strategy, but lets constructs compete with one another is the Constructs Competition Miner (Redlich et al. 2014), however its implementation has not been published.

Structured Miner

The Structured Miner (STM) (Augusto et al. 2016) applies a different strategy to obtain highly block-structured models and to tradeoff the log quality criteria. Instead of discovering block-structured models directly, SM first discovers BPMN models and, second, structures these models. The models can be obtained from any other discovery technique, for instance Heuristics Miner or Fodina, as these models need not be

sound. These models are translated to BPMN, after which they are made block-structured by shifting BPMN-gateways in or out, thereby duplicating activities.

STM benefits from the flexibility of the used other discovery technique to strike a flexible balance between log-quality criteria and can guarantee to return sound models. However, this guarantee comes at the price of equivalence (the model is changed, not just restructured), simplicity (activities are duplicated) and speed (the restructuring is $O(n^n)$).

STM is available as both a ProM and an Apromore plugin.

(Hybrid) Integer Linear Programming Miner

The Integer Linear Programming Miner (ILP) van der Werf et al. (2009) constructs a Petri net, starting with all activities as transitions and no places, such that every activity can be arbitrarily executed. Second, it adds places using an optimisation technique: a place is only added if it does not remove any trace of the event log from the behaviour of the model. Under this condition, the behaviour is restricted as much as possible.

ILP focusses on fitness and precision: it guarantees to return a model that fits the event log, and the most precise model within its representational bias (Petri nets, no duplicated activities). However, the ILP miner does not guarantee soundness, does not handle noise and tends to return complex models (Leemans 2017).

The first of these two have been addressed in the HybridILPMiner (van Zelst et al. 2017), which performs internal noise filtering. Furthermore, it adjusts the optimisation step to guarantee that the final marking is always reachable, and, in some cases, returns workflow nets, thereby achieving *relaxed soundness*.

Declarative Techniques

Petri nets and BPMN models express what *can* happen when executing the model. In contrast, declarative models, such as Declare models, express what *cannot* happen when executing the model, thereby providing greater flexibility in modelling. Declare miners such as Maggi et al. (2011), Di Ciccio et al. (2016), and Ferilli et al. (2016) discover the constraints of which Declare models consist using several acceptance criteria, in order to be able to balance precision and fitness. However, using such models in practice tends to be challenging (Augusto et al. 2017b).

Other Algorithms

Unsound models are unsuitable for futher automated processing, however might be useful for manual analysis. In the remainder of this section, several algorithms are discussed that do not guarantee soundness.

α-Algorithms

The first process discovery algorithm described was the α-algorithm (van der Aalst et al. 2004). The α algorithm considers the directly follows graph and identifies three types of relations between sets of activities from the graph: sequence, concurrency and mutual exclusivity. From these relations, a Petri net is constructed by searching for certain maximal patterns.

The α algorithm is provably (Badouel 2012) able to rediscover some processes, assuming that the log contains enough information and with restrictions on the process. In later versions, several restrictions have been addressed, such as: (a) no short loops (activities can follow one another directly; addressed in α^+ (de Medeiros et al. 2004)), (b) no long-distance dependencies (choices later in the process depend on choices made earlier; addressed in Wen et al. (2006)), (c) no non-free-choice constructs (transitions that share input places have the same input places; addressed in α^{++} (Wen et al. 2007a)), and (d) no silent transitions (addressed in $\alpha^{\#}$ (Wen et al. 2007b, 2010) and in $\alpha^{\$}$ (Guo et al. 2015)). Furthermore, a variant has been proposed, called the Tsinghua-α (Wen et al. 2009), that deals with non-atomic event logs. That is, event logs in which executions of activities take time.

However, these algorithms guarantee neither soundness nor perfect fitness nor perfect precision, the algorithms cannot handle noise and cannot handle incompleteness. Furthermore, the α algorithms might be less fast on complex event logs, as typically they are exponential. Therefore,

the α-algorithms are not very suitable to be applied to real-life logs.

Little Thumb (Weijters and van der Aalst 2003) extends the α algorithms with noise-handling capabilities: instead of considering binary activity relations, these relations are derived probabilistically and then filtered according to a user-set threshold.

Causal-Net Miners

The Flexible Heuristics Miner (FHM) (Weijters and Ribeiro 2011) uses the probabilistic activity relations of Little Thumb and focuses on soundness. To solve the issue of soundness, FHM returns causal nets, a model formalism in which it is defined that non-sound parts of the model are not part of the behaviour of the net.

The Fodina algorithm (vanden Broucke and Weerdt 2017) extends FHM with long-distance dependency support and, in some cases, duplicate activities. The Proximity miner (Yahya et al. 2016) extends FHM by incorporating domain knowledge. For more algorithms using causal nets, please refer to Weerdt et al. (2012) and Augusto et al. (2017b).

Even though causal nets are sound by definition, they place the burden of soundness checking on the interpreter/user of the net, and this still does not guarantee, for instance, that every activity in the model can be executed. Therefore, translating a causal net to a Petri net or BPMN model for further processing does not guarantee soundness of the translated model.

FHM, Fodina (http://www.processmining.be/fodina) and Proximity Miner (https://sourceforge.net/projects/proxi-miner/) are all available as ProM plug-ins and/or Apromore plug-ins.

Split Miner

To strike a different balance in log-quality criteria compared to IM that favours fitness, while improving in speed over ETM, Split Miner (SPM) (Augusto et al. 2017a) preprocesses the directly follows graph before constructing a BPMN model. In the preprocessing of directly follows graphs, first, loops and concurrency are identified and filtered out. Second, the graph is filtered in an optimisation step: each node must be on a path from start to end, the total number of edges is minimised, while the sum of edge frequencies is maximised. Then, splits and joins (BPMN gateways) are inserted to construct a BPMN model.

SPM aims to improve over the precision of IM and the speed of ETM for real-life event logs. The balance between precision and fitness can be adjusted in the directly follows-optimisation step, which allows users to adjust the amount of noise filtering. However, the returned models are not guaranteed to be sound (proper completion is not guaranteed), and several OR-joins might be inserted, which increases complexity.

SPM is available as a plug-in of Apromore and as a stand-alone tool via https://doi.org/10.6084/m9.figshare.5379190.v1.

Conclusion

Many process mining techniques require a process model as a prerequisite. From an event log, process discovery algorithms aim to discover a process model, this model preferably having clear semantics, being sound, striking a user-adjustable balance between fitness, precision, generalisation and simplicity, and having confidence that the model represents the business process from which the event log was recorded. Three types of process discovery algorithms were discussed: directly follows-based techniques, soundness-guaranteeing algorithms and other algorithms, all targetting a subset of these quality criteria.

In explorative process mining projects, choosing a discovery algorithm and its parameters is a matter of repeatedly trying *soundness-guaranteeing* algorithms, evaluating their results using conformance checking and adjusting algorithm, parameters and event log as new questions pop up (van Eck et al. 2015).

Cross-References

▸ Conformance Checking
▸ Decision Discovery in Business Processes

- Event Log Cleaning for Business Process Analytics
- Process Model Repair

References

Adriansyah A (2014) Aligning observed and modeled behavior. PhD thesis, Eindhoven University of Technology

Augusto A, Conforti R, Dumas M, Rosa ML, Bruno G (2016) Automated discovery of structured process models: discover structured vs. discover and structure. In: Comyn-Wattiau I, Tanaka K, Song I, Yamamoto S, Saeki M (eds) Conceptual modeling – Proceedings of the 35th international conference, ER, Gifu, 14–17 Nov 2016. Lecture notes in computer science, vol 9974, pp 313–329. http://doi.org/10.1007/978-3-319-46397-1_25

Augusto A, Conforti R, Dumas M, Rosa ML (2017a) Split miner: discovering accurate and simple business process models from event logs. In: IEEE international conference on data mining, New Orleans. https://eprints.qut.edu.au/110153/

Augusto A, Conforti R, Dumas M, Rosa ML, Maggi FM, Marrella A, Mecella M, Soo A (2017b) Automated discovery of process models from event logs: review and benchmark. CoRR abs/1705.02288, http://arxiv.org/abs/1705.02288

Badouel E (2012) On the α-reconstructibility of workflow nets. In: Haddad S, Pomello L (eds) Application and theory of Petri Nets – Proceedings of the 33rd international conference, PETRI NETS, Hamburg, 25–29 June 2012. Lecture notes in computer science, vol 7347. Springer, pp 128–147. http://doi.org/10.1007/978-3-642-31131-4_8

Buijs JCAM, van Dongen BF, van der Aalst WMP (2012a) A genetic algorithm for discovering process trees. In: Proceedings of the IEEE congress on evolutionary computation, CEC, Brisbane, 10–15 June 2012. IEEE, pp 1–8. http://doi.org/10.1109/CEC.2012.6256458

Buijs JCAM, van Dongen BF, van der Aalst WMP (2012b) On the role of fitness, precision, generalization and simplicity in process discovery. In: Meersman R, Panetto H, Dillon TS, Rinderle-Ma S, Dadam P, Zhou X, Pearson S, Ferscha A, Bergamaschi S, Cruz IF (eds) On the move to meaningful internet systems: OTM 2012, Proceedings of the confederated international conferences: CoopIS, DOA-SVI, and ODBASE, Rome, part I, 10–14 Sept 2012. Lecture notes in computer science, vol 7565. Springer, pp 305–322. http://doi.org/10.1007/978-3-642-33606-5_19

Celonis (2017) Process mining. https://www.celonis.com/. [Online; Accessed 11 Nov 2017]

DBL (2011) Proceedings of the IEEE symposium on computational intelligence and data mining, CIDM 2011, part of the IEEE symposium series on computational intelligence, 11–15 Apr 2011. IEEE, Paris. http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=5937059

de Medeiros AKA, van Dongen BF, van der Aalst WMP, Weijters AJMM (2004) Process mining for ubiquitous mobile systems: an overview and a concrete algorithm. In: Baresi L, Dustdar S, Gall HC, Matera M (eds) Ubiquitous mobile information and collaboration systems, second CAiSE workshop, UMICS, Riga, 7–8 June 2004, Revised selected papers. Lecture notes in computer science, vol 3272. Springer, pp 151–165. http://doi.org/10.1007/978-3-540-30188-2_12

Di Ciccio C, Maggi FM, Mendling J (2016) Efficient discovery of target-branched declare constraints. Inf Syst 56:258–283. http://doi.org/10.1016/j.is.2015.06.009

Ferilli S, Esposito F, Redavid D, Angelastro S (2016) Predicting process behavior in woman. In: Adorni G, Cagnoni S, Gori M, Maratea M (eds) AI*IA 2016: advances in artificial intelligence – Proceedings of the XVth international conference of the Italian association for artificial intelligence, Genova, 29 Nov–1 Dec 2016. Lecture notes in computer science, vol 10037. Springer, pp 308–320. http://doi.org/10.1007/978-3-319-49130-1_23

Fluxicon (2017) Disco. http://fluxicon.com, [Online; Accessed 11 Nov 2017]

Günther C, van der Aalst W (2007) Fuzzy mining–adaptive process simplification based on multi-perspective metrics. Business process management. Springer, Berlin/Heidelberg, pp 328–343

Guo Q, Wen L, Wang J, Yan Z, Yu PS (2015) Mining invisible tasks in non-free-choice constructs. In: Motahari-Nezhad HR, Recker J, Weidlich M (eds) Business process management – Proceedings of the 13th international conference, BPM, Innsbruck, 31 Aug–3 Sept 2015. Lecture notes in computer science, vol 9253. Springer, pp 109–125. http://doi.org/10.1007/978-3-319-23063-4_7

Leemans S (2017) Robust process mining with guarantees. PhD thesis, Technische Universiteit Eindhoven

Leemans SJJ, Fahland D, van der Aalst WMP (2013a) Discovering block-structured process models from event logs – a constructive approach. In: Colom JM, Desel J (eds) Application and theory of Petri Nets and concurrency – Proceedings of the 34th international conference, PETRI NETS, Milan, 24–28 June 2013. Lecture notes in computer science, vol 7927. Springer, pp 311–329. http://doi.org/10.1007/978-3-642-38697-8_17

Leemans SJJ, Fahland D, van der Aalst WMP (2013b) Discovering block-structured process models from event logs containing infrequent behaviour. In: Lohmann N, Song M, Wohed P (eds) Business process management workshops – BPM 2013 international workshops, Beijing, 26 Aug 2013, Revised papers. Lecture notes in business information processing, vol 171. Springer, pp 66–78. http://doi.org/l10.1007/978-3-319-06257-0_6

Leemans SJJ, Fahland D, van der Aalst WMP (2014a) Discovering block-structured process models from incomplete event logs. In: Ciardo G, Kindler E (eds)

Application and theory of Petri Nets and concurrency – Proceedings of the 35th international conference, PETRI NETS, Tunis, 23–27 June 2014. Lecture notes in computer science, vol 8489. Springer, pp 91–110. http://doi.org10.1007/978-3-319-07734-5_6

Leemans SJJ, Fahland D, van der Aalst WMP (2014b) Process and deviation exploration with inductive visual miner. In: Limonad L, Weber B (eds) Proceedings of the BPM demo sessions 2014, co-located with the 12th international conference on business process management(BPM), Eindhoven, 10 Sept 2014, CEUR-WS.org, CEUR workshop proceedings, vol 1295, p 46. http://ceur-ws.org/Vol-1295/paper19.pdf

Leemans SJJ, Fahland D, van der Aalst WMP (2015) Using life cycle information in process discovery. In: Reichert M, Reijers HA (eds) Business process management workshops – BPM, 13th international workshops, Innsbruck, 31 Aug–3 Sept 2015, Revised papers. Lecture notes in business information processing, vol 256. Springer, pp 204–217. http://doi.org/10.1007/978-3-319-42887-1_17

Leemans SJJ, Fahland D, van der Aalst WMP (2016) Scalable process discovery and conformance checking. Softw Syst Model Special issue:1–33. http://doi.org/10.1007/s10270-016-0545-x

Maggi FM, Mooij AJ, van der Aalst WMP (2011) User-guided discovery of declarative process models. In: DBL (2011), pp 192–199. http://doi.org/10.1109/CIDM.2011.5949297

OMG (2011) Business process model and notation (BPMN) version 2.0. Technical report, Object management group (OMG)

ProcessGold (2017) Enterprise platform. http://processgold.com/en/, [Online; Accessed 11 Nov 2017]

Redlich D, Molka T, Gilani W, Blair GS, Rashid A (2014) Constructs competition miner: process control-flow discovery of bp-domain constructs. In: Sadiq SW, Soffer P, Völzer H (eds) Business process management – Proceedings of the 12th international conference, BPM, Haifa, 7–11 Sept 2014. Lecture notes in computer science, vol 8659. Springer, pp 134–150. http://doi.org/10.1007/978-3-319-10172-9_9

Reisig W (1992) A primer in Petri net design. Springer compass international. Springer, Berlin/New York

Rosa ML, Reijers HA, van der Aalst WMP, Dijkman RM, Mendling J, Dumas M, García-Bañuelos L (2011) APROMORE: an advanced process model repository. Expert Syst Appl 38(6):7029–7040. http://doi.org/10.1016/j.eswa.2010.12.012

vanden Broucke SKLM, Weerdt JD (2017) Fodina: a robust and flexible heuristic process discovery technique. Decis Support Syst 100:109–118. http://doi.org/10.1016/j.dss.2017.04.005

van der Aalst WMP (2016) Process mining – data science in action, 2nd edn. Springer http://www.springer.com/gp/book/9783662498507

van der Aalst W, Weijters A, Maruster L (2004) Workflow mining: discovering process models from event logs. IEEE Trans Knowl Data Eng 16(9):1128–1142

van der Werf JMEM, van Dongen BF, Hurkens CAJ, Serebrenik A (2009) Process discovery using integer linear programming. Fundam Inform 94(3–4):387–412. http://doi.org/10.3233/FI-2009-136

van Dongen BF, de Medeiros AKA, Verbeek HMW, Weijters AJMM, van der Aalst WMP (2005) The prom framework: a new era in process mining tool support. In: Ciardo G, Darondeau P (eds) Applications and theory of Petri Nets 2005, Proceedings of the 26th international conference, ICATPN, Miami, 20–25 June 2005. Lecture notes in computer science, vol 3536. Springer, pp 444–454. http://doi.org/10.1007/11494744_25

van Eck ML, Lu X, Leemans SJJ, van der Aalst WMP (2015) PMˆ2 : a process mining project methodology. In: Zdravkovic J, Kirikova M, Johannesson P (eds) Advanced information systems engineering – Proceedings of the 27th international conference, CAiSE, Stockholm, 8–12 June 2015. Lecture notes in computer science, vol 9097. Springer, pp 297–313. http://doi.org/10.1007/978-3-319-19069-3_19

van Zelst SJ, van Dongen BF, van der Aalst WMP, Verbeek HMW (2017) Discovering relaxed sound workflow nets using integer linear programming. CoRR abs/1703.06733. http://arxiv.org/abs/1703.06733

Weerdt JD, Backer MD, Vanthienen J, Baesens B (2012) A multi-dimensional quality assessment of state-of-the-art process discovery algorithms using real-life event logs. Inf Syst 37(7):654–676. http://doi.org/10.1016/j.is.2012.02.004

Weijters AJMM, Ribeiro JTS (2011) Flexible heuristics miner (FHM). In: DBL (2011), pp 310–317. http://doi.org/10.1109/CIDM.2011.5949453

Weijters AJMM, van der Aalst WMP (2003) Rediscovering workflow models from event-based data using little thumb. Integr Comput Aided Eng 10(2):151–162. http://content.iospress.com/articles/integrated-computer-aided-engineering/ica00143

Wen L, Wang J, Sun J (2006) Detecting implicit dependencies between tasks from event logs. In: Zhou X, Li J, Shen HT, Kitsuregawa M, Zhang Y (eds) Frontiers of WWW research and development – APWeb 2006, Proceedings of the 8th Asia-Pacific web conference, Harbin, 16–18 Jan 2006. Lecture notes in computer science, vol 3841. Springer, pp 591–603. http://doi.org/10.1007/11610113_52

Wen L, van der Aalst WMP, Wang J, Sun J (2007a) Mining process models with non-free-choice constructs. Data Min Knowl Discov 15(2):145–180. http://doi.org/10.1007/s10618-007-0065-y

Wen L, Wang J, Sun J (2007b) Mining invisible tasks from event logs. In: Dong G, Lin X, Wang W, Yang Y, Yu JX (eds) Advances in data and web management, Joint 9th Asia-Pacific web conference, APWeb 2007, and Proceedings of the 8th international conference, on web-age information management, WAIM, Huang Shan, 16–18 June 2007. Lecture notes in computer science, vol 4505. Springer, pp 358–365. http://doi.org/10.1007/978-3-540-72524-4_38

Wen L, Wang J, van der Aalst WMP, Huang B, Sun J (2009) A novel approach for process mining based on

event types. J Intell Inf Syst 32(2):163–190. http://doi.org/10.1007/s10844-007-0052-1
Wen L, Wang J, van der Aalst WMP, Huang B, Sun J (2010) Mining process models with prime invisible tasks. Data Knowl Eng 69(10):999–1021. http://doi.org/10.1016/j.datak.2010.06.001
Yahya BN, Song M, Bae H, Sul S, Wu J (2016) Domain-driven actionable process model discovery. Comput Ind Eng 99:382–400. http://doi.org/10.1016/j.cie.2016.05.010

Automated Reasoning

Jeff Z. Pan[1] and Jianfeng Du[2]
[1]University of Aberdeen, Scotland, UK
[2]Guangdong University of Foreign Studies, Guangdong, China

Synonyms

Approximate reasoning; Inference; Logical reasoning; Semantic computing

Definitions

Reasoning is the process of deriving conclusions in a logical way. Automatic reasoning is concerned with the construction of computing systems that automate this process over some knowledge bases.

Automated Reasoning is often considered as a subfield of artificial intelligence. It is also studied in the fields of theoretical computer science and even philosophy.

Overview

The development of formal logic (Frege 1884) played a big role in the field of automated reasoning, which itself led to the development of artificial intelligence.

Historically, automated reasoning is largely related to theorem proving, general problem solvers, and expert systems (cf. the section of "A Bit of History"). In the context of big data processing, automated reasoning is more relevant to modern knowledge representation languages, such as the W3C standard Web Ontology Language (OWL) (https://www.w3.org/TR/owl2-overview/), in which a knowledge base consists of a schema component (TBox) and a data component (ABox).

From the application perspective, perhaps the most well-known modern knowledge representation mechanism is Knowledge Graph (Pan et al. 2016b, 2017). In 2012, Google popularized the term "Knowledge Graph" by using it for improving its search engine. Knowledge Graphs are then adopted by most leading search engines (such as Bing and Baidu) and many leading IT companies (such as IBM and Facebook). The basic idea of Knowledge Graph is based on the knowledge representation formalism called semantic networks. There is a modern W3C standard for semantic networks called RDF (Resource Description Framework, https://www.w3.org/TR/rdf11-concepts/). Thus RDF/OWL graphs can be seen as exchangeable knowledge graphs, in the big data era.

While this entry will be mainly about automated reasoning techniques in the big data era, their classifications, key contributions, typical systems, as well as their applications, it starts with a brief introduction of the history.

A Bit of History

Many consider the Cornell Summer Meeting of 1957, which brought together many logicians and computer scientists, as the origin of automated reasoning.

The first automated reasoning systems were theorem provers, systems that represent axioms and statements in first-order logic and then use rules of logic, such as modus ponens, to infer new statements. The first system of this kind is the implementation of Presburger's decision procedure (which proved that the sum of two even numbers is even) by Davis (1957).

Another early type of automated reasoning system were general problem solvers, which at-

tempt to provide a generic planning engine that could represent and solve structured problems, by decomposing problems into smaller more manageable subproblems, solving each subproblem and assembling the partial answers into one final answer. The first system of this kind is Logic Theorist from Newell et al. (1957).

The first practical applications of automated reasoning were expert systems, which focused on much more well-defined domains than general problem solving, such as medical diagnosis or analyzing faults in an aircraft, and on more limited implementations of first-order logic, such as modus ponens implemented via IF-THEN rules. One of the forerunners of these systems is MYCIN by Shortliffe (1974).

Since 1980s, there have been prosperous studies of practical subsets of first-order logics as ontology languages, such as description logics (Baader et al. 2003) and answer set programming (Lifschitz 2002), as well as the standardization of ontology language OWL (version 1 in 2004 and version 2 in 2009). The wide adoption of ontology and Knowledge Graph (Pan et al. 2016b, 2017), including by Google and many other leading IT companies, confirms the status of ontology language in big data era.

In the rest of the entry, we will focus on automated reasoning with Ontology languages.

Classification

There can be different ways of classifying research problems related to automated ontology reasoning.

From the *purpose* point of view, automatic ontology reasoning can be classified into (1) deductive ontology reasoning (Levesque and Brachman 1987), which draws conclusions from given premises; (2) abductive ontology reasoning (Colucci et al. 2003), which finds explanations for observations that are not consequences of given premises; as well as (3) inductive ontology reasoning (Lisi and Malerba 2003), which concludes that all instances of a class have a certain property if some instances of the class have the property.

From the *direction* point of view, automatic ontology reasoning can be classified into (1) forward reasoning Baader et al. (2005), in which the inference starts with the premises, moves forward, and ends with the conclusions; (2) backward reasoning (Grosof et al. 2003), in which the inference starts with the conclusions, moves backward, and ends with the premises; as well as (3) bi-directional reasoning (MacGregor 1991) in which the inference starts with both the premises and the conclusions and moves forward and backward simultaneously or interactively, until the intermediate conclusions obtained by forward steps include all intermediate premises required by backward steps.

From the *monotonicity* point of view, automatic ontology reasoning can be classified into (1) monotonic ontology reasoning in which no existing conclusions will be dropped when new premises are added, as well as (2) nonmonotonic ontology reasoning (Quantz and Suska 1994) in which some existing conclusions can be dropped when new premises are added.

From the *scalability* point of view, automatic ontology reasoning can be classified into (1) parallel ontology reasoning (Bergmann and Quantz 1995), in which reasoning algorithms can exploit multiple computation cores in a computation nodes, and (2) distributed ontology reasoning (Borgida and Serafini 2003) and (Serafini 2005), in which reasoning algorithms can exploit a cluster of computation nodes. Scalable ontology reasoning is also often related to strategies of modularization (Suntisrivaraporn et al. 2008) and approximation (Pan and Thomas 2007).

From the *mobility* point of view, automated ontology reasoning can be classified into (1) reasoning with temporal ontologies (Artale and Franconi 1994), in which the target ontologies contain temporal constructors for class and property descriptions, and (2) stream ontology reasoning (Stuckenschmidt et al. 2010; Ren and Pan 2011), which, given some continuous updates of the ontology, requires updating reasoning results without naively recomputing all results.

From the *certainty* point of view, automatic reasoning can be classified into (1) ontology reasoning with certainty in which both premises and

conclusions are certain and either true or false, as well as (2) uncertainty ontology reasoning (Koller et al. 1997) in which either premises or conclusions are uncertain and often have truth values between $0/-1$ and 1. There are different kinds of uncertainties within ontologies, such as probabilistic ontologies (Koller et al. 1997), fuzzy ontologies (Straccia 2001), and possibilistic ontologies (Qi et al. 2011).

Key Contributions

The highlight on contributions of automated ontology reasoning is the standardization of the Web Ontology Language (OWL).

The first version of OWL (or OWL 1) was standardized in 2004. It is based on the $\mathcal{SHOIQ}$ DL (Horrocks and Sattler 2005). However, there are some limitations of OWL 1:

1. The datatype support is limited (Pan and Horrocks 2006);
2. The only sub-language, OWL-Lite, of OWL 1 is not tractable;
3. The semantics of OWL 1 and RDF are not fully compatible (Pan and Horrocks 2003).

The second version of OWL (or OWL 2) was standardized in 2009. It is based on the $\mathcal{SROIQ}$ DL (Horrocks et al. 2006). On the one hand, OWL 2 has more expressive power, such as the stronger support of datatypes (Pan and Horrocks 2006; Motik and Horrocks 2008) and rules (Krötzsch et al. 2008). On the other hand, OWL 2 has three tractable sub-languages, including OWL 2 EL (Baader et al. 2005), OWL 2 QL (Calvanese et al. 2007), and OWL 2 RL (Grosof et al. 2003).

This two-layer architecture of OWL 2 allows approximating OWL 2 ontologies to those in its tractable sub-languages, such as approximations toward OWL 2 QL (Pan and Thomas 2007), toward OWL 2 EL (Ren et al. 2010), and toward OWL 2 RL (Zhou et al. 2013), so as to exploit efficient and scalable reasoners of the sub-languages. The motivation is based on the fact that real-world knowledge and data are hardly perfect or completely digitalized.

Typical Reasoning Systems

Below are descriptions of some well-known OWL reasoners (in alphabetical order).

CEL

CEL (Baader et al. 2006) is a LISP-based reasoner for $\mathcal{EL}+$ (Baader et al. 2008), which covers the core part of OWL 2 EL. CEL is the first reasoner for the description logic $\mathcal{EL}+$, supporting as its main reasoning task the computation of the subsumption hierarchy induced by $\mathcal{EL}+$ ontologies.

ELK

ELK (Kazakov et al. 2012) is an OWL 2 EL reasoner. At its core, ELK uses a highly optimized parallel algorithm (Kazakov et al. 2011). It supports stream reasoning in OWL 2 EL (Kazakov and Klinov 2013).

FaCT

FaCT Horrocks (1998) is a reasoner for the description logic $\mathcal{SHIF}$ (OWL-Lite). It is the first modern reasoner that demonstrates the feasibility of using optimized algorithms for subsumption checking in realistic applications.

FaCT++

FaCT++ (Tsarkov and Horrocks 2006) is a reasoner for (partially) OWL 2. It is the new generation of the well-known FaCT reasoner which is implemented using C++, with a different internal architecture and some new optimizations.

HermiT

HermiT (Glimm et al. 2014) is a reasoner for OWL 2. It is the first publicly available OWL 2 reasoner based on a hypertableau calculus (Motik et al. 2009), with a highly optimized algorithm for ontology classification (Glimm et al. 2010). HermiT can handle DL-Safe rules (Motik et al. 2005) on top of OWL 2.

Konclude

Konclude (Steigmiller et al. 2014b) is a reasoner for OWL 2. It supports almost all datatypes in OWL 2. Konclude implements a highly optimized version of tableau calculus enhanced with tableau saturation (Steigmiller and Glimm 2015). It supports parallel reasoning and nominal schemas (Steigmiller et al. 2014a) and DL-safe rules.

Mastro

Mastro (Calvanese et al. 2011) is an ontology-based data access (OBDA) management system for OWL 2 QL. It allows data to be managed by external relational data management or data federation systems. It uses the Presto algorithm Rosati and Almatelli (2010) for query rewriting.

Ontop

Ontop (Calvanese et al. 2016) is an ontology-based data access (OBDA) management system for RDF and OWL 2 QL, as well as SWRL with limited forms of recursions. It also supports efficient SPARQL-to-SQL mappings via R2RML (Rodriguez-Muro and Rezk 2015). Ontop has some optimizations on query rewriting based on database dependencies (Rodriguez-Muro et al. 2013).

Pellet

Pellet (Sirin et al. 2007) is a reasoner for OWL 2. It also has dedicated support for OWL 2 EL. It incorporates optimizations for nominals, conjunctive query answering, and incremental reasoning.

Racer

Racer (Haarslev and Möller 2001) is a reasoner for OWL 1. It has a highly optimized version of tableau calculus for the description logic $\mathcal{SHIQ}(D)$ (Horrocks and Patel-Schneider 2003).

RDFox

RDFox (Motik et al. 2014) is a highly scalable in-memory RDF triple store that supports shared memory parallel datalog (Ceri et al. 1989) reasoning. It supports stream reasoning (Motik et al. 2015b) and has optimizations for owl:sameAs (Motik et al. 2015a).

TrOWL

TrOWL (Thomas et al. 2010) is a highly optimized approximate reasoner (Pan et al. 2016a) for OWL 2. TrOWL outperforms some sound and complete reasoners in the time-constrained ORE (Ontology Reasoner Evaluation) competitions designed for sound and complete ontology reasoners. TrOWL has stream reasoning capabilities for both OWL 2 and OWL 2 EL (Ren and Pan 2011; Ren et al. 2016). It supports local closed world reasoning in NBox or closed predicates (Lutz et al. 2013).

Applications

Automated ontology reasoning has been widely used in web applications, such as for content management (BBC), travel planning and booking (Skyscanner), and web search (Google, Bing, Baidu).

It is also being applied in a growing number of vertical domains. One typical example is life science. For instance, OBO Foundry includes more than 100 biological and biomedical ontologies. The SNOMED CT (Clinical Terminology) ontology is widely used in healthcare systems of over 15 countries, including the USA, the UK, Australia, Canada, Denmark, and Spain. It is also used by major US providers, such as Kaiser Permanente. Other vertical domains include, but not limited to, agriculture, astronomy, oceanography, defense, education, energy management, geography, and geoscience.

While ontologies are widely used as structured vocabularies, providing integrated and user-centric view of heterogeneous data sources in the big data era, benefits of using automated ontology reasoning include:

1. Reasoning support is critical for development and maintenance of ontologies, in particular on derivation of taxonomy from class definitions and descriptions.

2. Easy location of relevant terms within large structured vocabulary;
3. Query answers enhanced by exploiting schema and class hierarchy.

An example in the big data context is the use of ontology and automated ontology reasoning for data access in Statoil, where about 900 geologists and geophysicists use data from previous operations in nearby locations to develop stratigraphic models of unexplored areas, involving diverse schemata and TBs of relational data spread over 1000s of tables and multiple databases. Data analysis is the most important factor for drilling success. 30–70% of these geologists and geophysicists' time is spent on data gathering. The use of ontologies and automated ontology reasoning enables better use of experts' time, reducing turnaround for new queries significantly.

Outlook

Despite the current success of automated ontology reasoning, there are still some pressing challenges in the big data era, such as the following:

1. Declarative data analytics (Kaminski et al. 2017) based on automated ontology reasoning;
2. Effective approaches of producing high-quality (Ren et al. 2014; Konev et al. 2014) ontologies and Knowledge Graphs;
3. Integration of automated ontology reasoning with data mining (Lecue and Pan. 2015) and machine learning (Chen et al. 2017) approaches.

References

Artale A, Franconi E (1994) A computational account for a description logic of time and action. In: KR1994, pp 3–14

Baader F, Calvanese D, McGuinness DL, Nardi D, Patel-Schneider PF (eds) (2003) The description logic handbook: theory, implementation, and applications. Cambridge University Press, New York

Baader F, Brandt S, Lutz C (2005) Pushing the EL envelope. In: IJCAI2015

Baader F, Lutz C, Suntisrivaraporn B (2006) CEL—a polynomial-time reasoner for life science ontologies. In: IJCAR'06, pp 287–291

Baader F, Brandt S, Lutz C (2008) Pushing the EL envelope further. In: OWLED2008

Bergmann F, Quantz J (1995) Parallelizing description logics. In: KI1995, pp 137–148

Borgida A, Serafini L (2003) Distributed description logics. J Data Semant 1:153–184

Calvanese D, Giacomo GD, Lembo D, Lenzerini M, Rosati R (2007) Tractable reasoning and efficient query answering in description logics: the dl-lite family. J Autom Reason 39:385–429

Calvanese D, De Giacomo G, Lembo D, Lenzerini M, Poggi A, Rodriguez-Muro M, Rosati R, Ruzzi M, Savo DF (2011) The MASTRO system for ontology-based data access. J Web Semant 2:43–53

Calvanese D, Cogrel B, Komla-Ebri S, Kontchakov R, Lanti D, Rezk M, Rodriguez-Muro M, Xiao G (2016) Ontop: answering SPARQL queries over relational databases. Semant Web J 8:471-487

Ceri S, Gottlob G, Tancar L (1989) What you always wanted to know about Datalog (and never dared to ask). IEEE Trans Knowl Data Eng 1:146–166

Chen J, Lecue F, Pan JZ, Chen H (2017) Learning from ontology streams with semantic concept drift. In: IJCAI-2017, pp 957–963

Colucci S, Noia TD, Sciascio ED, Donini F (2003) Concept abduction and contraction in description logics. In: DL2003

Davis M (1957) A computer program for Presburgers' algorithm. In: Summaries of talks presented at the summer institute for symbolic logic, Cornell University, pp 215–233

Frege G (1884) Die Grundlagen der Arithmetik. Breslau, Wilhelm Kobner

Glimm B, Horrocks I, Motik B, Stoilos G (2010) Optimising ontology classification. In: ISWC2010, pp 225–240

Glimm B, Horrocks I, Motik B, Stoilos G, Wang Z (2014) HermiT: an OWL 2 reasoner. J Autom Reason 53: 245–269

Grosof BN, Horrocks I, Volz R, Decker S (2003) Description logic programs: combining logic programs with description logic. In: WWW2003, pp 48–57

Haarslev V, Möller R (2001) RACER system description. In: IJCAR2001, pp 701–705

Horrocks I (1998) Using an expressive description logic: FaCT or fiction? In: KR1998

Horrocks I, Patel-Schneider P (2003) From SHIQ and RDF to OWL: the making of a web ontology language. J Web Semant 1:7–26

Horrocks I, Sattler U (2005) A tableaux decision procedure for shoiq. In: IJCAI2005, pp 448–453

Horrocks I, Kutz O, Sattler U (2006) The even more irresistible sroiq. In: KR2006, pp 57–67

Kaminski M, Grau BC, Kostylev EV, Motik B, Horrocks I (2017) Foundations of declarative data analysis using limit datalog programs. In: Proceedings of the twenty-

sixth international joint conference on artificial intelligence, IJCAI-17, pp 1123–1130. https://doi.org/10.24963/ijcai.2017/156
Kazakov Y, Klinov P (2013) Incremental reasoning in OWL EL without bookkeeping. In: ISWC2013, pp 232–247
Kazakov Y, Krötzsch M, Simancik F (2011) Concurrent classification of EL ontologies. In: ISWC2011, pp 305–320
Kazakov Y, Krötzsch M, Simancik F (2012) ELK reasoner: architecture and evaluation. In: ORE2012
Koller D, Levy A, Pfeffer A (1997) P-CLASSIC: a tractable probablistic description logic. In: AAAI1997
Konev B, Lutz C, Ozaki A, Wolter F (2014) Exact learning of lightweight description logic ontologies. In: KR2014, pp 298–307
Krötzsch M, Rudolph S, Hitzler P (2008) Description logic rules. In: ECAI2008, pp 80–84
Serafini L, Tamilin A (2005) Drago: distributed reasoning architecture for the semantic web. In: ESWC2005, pp 361–376
Lecue F, Pan JZ (2015) Consistent knowledge discovery from evolving ontologies. In: AAAI-15
Levesque HJ, Brachman RJ (1987) Expressiveness and tractability in knowledge representation and reasoning. Comput Intell 3:78–93
Lifschitz V (2002) Answer set programming and plan generation. Artif Intell 138:39–54
Lisi FA, Malerba D (2003) Ideal refinement of descriptions in AL-Log. In: ICILP2003
Lutz C, Seylan I, Wolter F (2013) Ontology-based data access with closed predicates is inherently intractable (Sometimes). In: IJCAI2013, pp 1024–1030
MacGregor RM (1991) Inside the LOOM description classifier. ACM SIGART Bull – special issue on implemented knowledge representation and reasoning systems 2:88–92
Motik B, Horrocks I (2008) Owl datatypes: design and implementation. In: ISWC2008, pp 307–322
Motik B, Sattler U, Studer R (2005) Query answering for OWL-DL with rules. J Web Semant 3:41–60
Motik B, Shearer R, Horrocks I (2009) Hypertableau reasoning for description logics. J Artif Intell Res 36: 165–228
Motik B, Nenov Y, Piro R, Horrocks I, Olteanu D (2014) Parallel materialisation of datalog programs in centralised, main-memory RDF systems. In: AAAI2014
Motik B, Nenov Y, Piro R, Horrocks I (2015a) Handling of owl: sameAs via rewriting. In: AAAI2015
Motik B, Nenov Y, Robert Piro IH (2015b) Incremental update of datalog materialisation: the backward/forward algorithm. In: AAAI2015
Newell A, Shaw C, Simon H (1957) Empirical explorations of the logic theory machine. In: Proceedings of the 1957 western joint computer conference
Pan JZ, Horrocks I (2003) RDFS(FA) and RDF MT: two semantics for RDFS. In: Fensel D, Sycara K, Mylopoulos J (eds) ISWC2003
Pan JZ, Horrocks I (2006) OWL-Eu: adding customised datatypes into OWL. J Web Semant 4: 29–39
Pan JZ, Thomas E (2007) Approximating OWL-DL ontologies. In: The proceedings of the 22nd national conference on artificial intelligence (AAAI-07), pp 1434–1439
Pan JZ, Ren Y, Zhao Y (2016a) Tractable approximate deduction for OWL. Artif Intell 235:95–155
Pan JZ, Vetere G, Gomez-Perez J, Wu H (2016b) Exploiting linked data and knowledge graphs for large organisations. Springer, Switzerland
Pan JZ, Calvanese D, Eiter T, Horrocks I, Kifer M, Lin F, Zhao Y (2017) Reasoning web: logical foundation of knowledge graph construction and querying answering. Springer, Cham
Qi G, Ji Q, Pan JZ, Du J (2011) Extending description logics with uncertainty reasoning in possibilistic logic. Int J Intell Syst 26:353–381
Quantz JJ, Suska S (1994) Weighted defaults in description logics: formal properties and proof theory. In: Annual conference on artificial intelligence, pp 178–189
Ren Y, Pan JZ (2011) Optimising ontology stream reasoning with truth maintenance system. In: CIKM 2011
Ren Y, Pan JZ, Zhao Y (2010) Soundness preserving approximation for TBox reasoning. In: AAAI2010
Ren Y, Parvizi A, Mellish C, Pan JZ, van Deemter K, Stevens R (2014) Towards competency question-driven ontology authoring. In: Proceedings of the 11th conference on extended semantic web conference (ESWC 2014)
Ren Y, Pan JZ, Guclu I, Kollingbaum M (2016) A combined approach to incremental reasoning for EL ontologies. In: RR2016
Rodriguez-Muro M, Rezk M (2015) Efficient SPARQL-to-SQL with R2RML mappings. J Web Semant 33:141–169
Rodriguez-Muro M, Kontchakov R, Zakharyaschev M (2013) Query rewriting and optimisation with database dependencies in ontop. In: DL2013
Rosati R, Almatelli A (2010) Improving query answering over DL-Lite ontologies. In: KR2010, pp 290–300
Shortliffe EH (1974) MYCIN: a rule-based computer program from advising physicians regarding antimicrobial therapy selection. PhD thesis, Stanford University
Sirin E, Parsia B, Grau B, Kalyanpur A, Katz Y (2007) Pellet: a practical owl-dl reasoner. J Web Semant 5:51–53
Steigmiller A, Glimm B (2015) Pay-As-You-Go description logic reasoning by coupling tableau and saturation procedure. J Artif Intell Res 54:535–592
Steigmiller A, Glimm B, Liebig T (2014a) Reasoning with nominal schemas through absorption. J Autom Reason 53:351–405
Steigmiller A, Liebig T, Glimm B (2014b) Konclude: system description. J Web Semant 27:78–85
Straccia U (2001) Reasoning within fuzzy description logics. J Artif Intell Res 14:147–176

Stuckenschmidt H, Ceri S, Valle ED, van Harmelen F (2010) Towards expressive stream reasoning. In: Semantic challenges in sensor networks, no. 10042 in Dagstuhl seminar proceedings

Suntisrivaraporn B, Qi G, Ji Q, Haase P (2008) A modularization-based approach to finding all justifications for OWL DL entailments. In: ASWC2008, pp 1–15

Thomas E, Pan JZ, Ren Y (2010) TrOWL: tractable OWL 2 reasoning infrastructure. In: ESWC2010

Tsarkov D, Horrocks I (2006) FaCT++ description logic reasoner: system description. In: IJCAR2006, pp 292–297

Zhou Y, Grau BC, Horrocks I, Wu Z, Banerjee J (2013) Making the most of your triple store: query answering in OWL 2 using an RL reasoner. In: WWW2013, pp 1569–1580

Availability

▸ Security and Privacy in Big Data Environment