Making State Explicit for Imperative Big Data Processing

Raul Castro Fernandez*, Matteo Migliavacca†, Evangelia Kalyvianaki♯, Peter Pietzuch*
*Imperial College London, †University of Kent, ♯City University London

Abstract

Data scientists often implement machine learning algorithms in imperative languages such as Java, Matlab and R. Yet such implementations fail to achieve the performance and scalability of specialised data-parallel processing frameworks. Our goal is to execute imperative Java programs in a data-parallel fashion with high throughput and low latency. This raises two challenges: how to support the arbitrary mutable state of Java programs without compromising scalability, and how to recover that state after failure with low overhead.

Our idea is to infer the dataflow and the types of state accesses from a Java program and use this information to generate a *stateful dataflow graph (SDG)*. By explicitly separating *data* from mutable *state*, SDGs have specific features to enable this translation: to ensure scalability, distributed state can be *partitioned* across nodes if computation can occur entirely in parallel; if this is not possible, *partial* state gives nodes local instances for independent computation, which are reconciled according to application semantics. For fault tolerance, large in-memory state is checkpointed asynchronously without global coordination. We show that the performance of SDGs for several imperative online applications matches that of existing data-parallel processing frameworks.

1 Introduction

Data scientists want to use ever more sophisticated implementations of machine learning algorithms, such as collaborative filtering [32], k-means clustering and logistic regression [21], and execute them over large datasets while providing "fresh", low latency results. With the dominance of imperative programming, such algorithms are often implemented in languages such as Java, Matlab or R. Such implementations though make it challenging to achieve high performance.

On the other hand, data-parallel processing frameworks, such as MapReduce [8], Spark [38] and Naiad [26], can scale computation to a large number of nodes. Such frameworks, however, require developers to adopt particular functional [37], declarative [13] or dataflow [15] programming models. While early frameworks such as MapReduce [8] followed a restricted functional model, resulting in wide-spread adoption, recent more expressive frameworks such as Spark [38] and Naiad [26] require developers to learn more complex programming models, e.g. based on a richer set of higher-order functions.

Our goal is therefore to translate imperative Java implementations of machine learning algorithms to a representation that can be executed in a data-parallel fashion. The execution should scale to a large number of nodes, achieving high throughput and low processing latency. This is challenging because Java programs support arbitrary mutable state. For example, an implementation of collaborative filtering [32] uses a mutable matrix to represent a model that is refined iteratively: as new data arrives, the matrix is updated at a fine granularity and accessed to provide up-to-date predictions.

Having stateful computation raises two issues: first, the state may grow large, e.g. on the order of hundreds of GBs for a collaborative filtering model with tens of thousands of users. Therefore the state and its associated computation must be distributed across nodes; second, large state must be restored efficiently after node failure. The failure recovery mechanism should have a low impact on performance.

Current data-parallel frameworks do not handle large state effectively. In stateless frameworks [8, 37, 38], computation is defined through side-effect-free functional tasks. Any modification to state, such as updating a single element in a matrix, must be implemented as the creation of new immutable data, which is inefficient. While recent frameworks [26, 10] have recognised the need for per-task mutable state, they lack abstractions for distributed state and exhibit high overhead under fault-tolerant operation with large state (see §6.1).

Imperative programming model. We describe how, with the help of a few annotations by developers, Java programs can be executed automatically in a distributed data-parallel fashion. Our idea is to infer the dataflow and the types of state accesses from a Java program and use this information to translate the program to an executable distributed dataflow representation. Using program analysis, our approach extracts the processing tasks and state fields from the program and infers the variable-level dataflow.

Stateful dataflow graphs. This translation relies on the features of a new fault-tolerant data-parallel processing model called *stateful dataflow graphs (SDGs)*. An SDG explicitly distinguishes between *data* and *state*: it is a cyclic graph of pipelined data-parallel tasks, which execute on different nodes and access local in-memory state.

SDGs include abstractions for maintaining large state efficiently in a distributed fashion: if tasks can process state entirely in parallel, the state is *partitioned* across nodes; if this is not possible, tasks are given local instances of *partial* state for independent computation. Computation can include synchronisation points to access all partial state instances, and instances can be reconciled according to application semantics.

Data flows between tasks in an SDG, and cycles specify iterative computation. All tasks are pipelined—this leads to low latency, less intermediate data during failure recovery and simplified scheduling by not having to compute data dependencies. Tasks are replicated at runtime to overcome processing bottlenecks and stragglers.

Failure recovery. When recovering from failures, nodes must restore potentially gigabytes of in-memory state. We describe an asynchronous checkpointing mechanism with log-based recovery that uses data structures for dirty state to minimise the interruption to tasks while taking local checkpoints. Checkpoints are persisted to multiple disks in parallel, from which they can be restored to multiple nodes, thus reducing recovery time.

With a prototype system of SDGs, we execute Java implementations of collaborative filtering, logistic regression and a key/value store on a private cluster and Amazon EC2. We show that SDGs execute with high throughput (comparable to batch processing systems) and low latency (comparable to streaming systems). Even with large state, their failure recovery mechanism has a low performance impact, recovering in seconds.

The paper contributions and its structure are as follows: based on a sample Java program (§2.1) and the features of existing dataflow models (§2.2), we motivate the need for stateful dataflow graphs and describe their properties (§3); §4 explains the translation from Java to SDGs; §5 describes failure recovery; and §6 presents evaluation results, followed by related work (§7).

Algorithm 1: Online collaborative filtering

```
1  @Partitioned Matrix userItem = new Matrix();
2  @Partial Matrix coOcc = new Matrix();
3
4  void addRating(int user, int item, int rating) {
5    userItem.setElement(user, item, rating);
6    Vector userRow = userItem.getRow(user);
7    for (int i = 0; i < userRow.size(); i++)
8      if (userRow.get(i) > 0) {
9        int count = coOcc.getElement(item, i);
10       coOcc.setElement(item, i, count + 1);
11       coOcc.setElement(i, item, count + 1);
12     }
13 }
14 Vector getRec(int user) {
15   Vector userRow = userItem.getRow(user);
16   @Partial Vector userRec = @Global coOcc.multiply(
          userRow);
17   Vector rec = merge(@Global userRec);
18   return rec;
19 }
20 Vector merge(@Collection Vector[] allUserRec) {
21   Vector rec = new Vector(allUserRec[0].size());
22   for (Vector cur : allUserRec)
23     for (int i = 0; i < allUserRec[0].size(); i++)
24       rec.set(i, cur.get(i) + rec.get(i));
25   return rec;
26 }
```

2 State in Data-Parallel Processing

We describe an imperative implementation of a machine learning algorithm and investigate how it can execute in a data-parallel fashion on a set of nodes, paying attention to its use of mutable state (§2.1). Based on this analysis, we discuss the features of existing data-parallel processing models for supporting such an execution (§2.2).

2.1 Application example

Alg. 1 shows a Java implementation of an online machine learning algorithm, *collaborative filtering* (CF) [32].[1] It outputs up-to-date recommendations of items to users (function `getRec`) based on previous item ratings (function `addRating`).

The algorithm maintains state in two data structures: the matrix `userItem` stores the ratings of items made by users (line 1); the co-occurrence matrix `coOcc` records correlations between items that were rated together by multiple users (line 2).

For many users and items, `useritem` and `coOcc` become large and must be distributed: `userItem` can be *partitioned* across nodes based on the `user` identifier as an access key; since the access to `coOcc` is random, it cannot be partitioned but only *replicated* on multiple nodes in order to parallelise updates. In this case, results from a single instance of `coOcc` are *partial*, and must be merged with other partial results to obtain a complete result, as described below.

The function `addRating` first adds a new rating to `userItem` (line 5). It then incrementally updates `coOcc` by increasing the co-occurrence counts for the newly-rated

[1]The annotations (starting with '@') will be explained in §4 and should be ignored for now.

Computational model	*Systems*	*Programming model*	*State handling*			*Dataflow*			*Failure recovery*
			Represen-tation	*Large state size*	*Fine-grained updates*	*Execution*	*Low latency*	*Iter-ation*	
Stateless dataflow	MapReduce [8]	map/reduce	as data	n/a	✗	scheduled	✗	✗	recompute
	DryadLINQ [37]	functional	as data	n/a	✗	scheduled	✗	✓	recompute
	Spark [38]	functional	as data	n/a	✗	hybrid	✗	✓	recompute
	CIEL [25]	imperative	as data	n/a	✗	scheduled	✗	✓	recompute
	HaLoop [5]	map/reduce	cache	✓	✗	scheduled	✗	✓	recompute
Incremental dataflow	Incoop [4]	map/reduce	cache	✓	✗	scheduled	✗	✗	recompute
	Nectar [11]	functional	cache	✓	✗	scheduled	✗	✗	recompute
	CBP [19]	dataflow	loopback	✓	✓	scheduled	✗	✗	recompute
Batched dataflow	Comet [12]	functional	as data	n/a	✗	scheduled	✓	✗	recompute
	D-Streams [39]	functional	as data	n/a	✗	hybrid	✓	✓	recompute
	Naiad [26]	dataflow	explicit	✗	✓	hybrid	✓	✓	sync. global checkpoints
Continuous dataflow	Storm, S4	dataflow	as data	n/a	✗	pipelined	✓	✗	recompute
	SEEP [10]	dataflow	explicit	✗	✓	pipelined	✓	✗	sync. local checkpoints
Parallel in-memory	Piccolo [30]	imperative	explicit	✓	✓	n/a	✓	✓	async. global checkpoints
Stateful dataflow	SDG	imperative	explicit	✓	✓	pipelined	✓	✓	async. local checkpoints

Table 1: Design space of data-parallel processing frameworks

`item` and existing items with non-zero ratings (line 7–12). This requires `userItem` and `coOcc` to be mutable, with efficient fine-grained access. Since `userItem` is partitioned based on the key `user`, and `coOcc` is replicated, `addRating` only accesses a single instance of each.

The function `getRec` takes the rating vector of a user, `userRow` (line 15), and multiplies it by the co-occurrence matrix to obtain a recommendation vector `userRec` (line 16). Since `coOcc` is replicated, this must be performed on *all* instances of `coOcc`, leading to multiple partial recommendation vectors. These partial vectors must be *merged* to obtain the final recommendation vector `rec` for the user (line 17). The function `merge` simply computes the sum of all partial recommendation vectors (lines 21–24).

Note that `addRating` and `getRec` have different performance goals when handling state: `addRating` must achieve *high throughput* when updating `coOcc` with new ratings; `getRec` must serve requests with *low latency*, e.g. when recommendations are included in dynamically generated web pages.

2.2 Design space

The above example highlights a number of required features of a dataflow model to enable the translation of imperative online machine learning algorithms to executable dataflows: (i) the model should support *large state sizes* (on the order of GBs), which should be represented explicitly and handled with acceptable performance; in particular, (ii) the state should permit efficient *fine-grained updates*. In addition, due to the need for up-to-date results, (iii) the model should process data with *low latency*, independently of the amount of input data; (iv) algorithms such as logistic regression and k-means clustering also require *iteration*; and (v) even with large state, the model should support fast *failure recovery*.

In Table 1, we classify existing data-parallel processing models according to the above features.

State handling. *Stateless dataflows*, first made popular by MapReduce [8], define a functional dataflow graph in which vertices are stateless data-parallel tasks. They do not distinguish between state and data: e.g. in a word-count job in MapReduce, the partial word counts, which are the state, are output by map tasks as part of the dataflow [8]. Dataflows in Spark, represented as RDDs, are immutable, which simplifies failure recovery but requires a new RDD for each state update [38]. This is inefficient for online algorithms such as CF in which only part of a matrix is updated each time.

Stateless models also cannot treat data differently from state. They cannot use custom index data structures for state access, or cache only state in memory: e.g. Shark [36] needs explicit hints which dataflows to cache.

Incremental dataflow avoids rerunning entire jobs after updates to the input data. Such models are fundamentally stateful because they maintain results from earlier computation. Incoop [4] and Nectar [11] treat state as a cache of past results. Since they cannot infer which data will be reused, they cache all. CBP transforms batch jobs automatically for incremental computation [19].

Our goals are complementary: SDGs do not infer incremental computation but support stateful computation efficiently, which can realise incremental algorithms.

Existing models that represent state explicitly, such as SEEP [10] and Naiad [26], permit tasks to have access to in-memory data structures but face challenges related to state sizes: they assume that state is small compared to the data. When large state requires distributed processing through partitioning or replication, they do not provide abstractions to support this.

In contrast, Piccolo [30] supports scalable distributed state with a key/value abstraction. However, it does not offer a dataflow model, which means that it cannot execute an inferred dataflow from a Java program but requires computation to be specified as multiple kernels.

Latency and iteration. Tasks in a dataflow graph can be *scheduled* for execution or materialised in a *pipeline*, each with different performance implications. Some frameworks follow a *hybrid* approach in which tasks on the same node are pipelined but not between nodes.

Since tasks in *stateless dataflows* are scheduled to process coarse-grained batches of data, such systems can exploit the full parallelism of a cluster but they cannot achieve low processing latency. For lower latency, *batched dataflows* divide data into small batches for processing and use efficient, yet complex, task schedulers to resolve data dependencies. They have a fundamental trade-off between the lower latency of smaller batches and the higher throughput of larger ones—typically they burden developers with making this trade-off [39].

Continuous dataflow adopts a streaming model with a pipeline of tasks. It does not materialise intermediate data between nodes and thus has lower latency without a scheduling overhead: as we show in §6, batched dataflows cannot achieve the same low latencies. Due to our focus on online processing with low latency, SDGs are fully pipelined (see §3.1).

To improve the performance of *iterative computation* in dataflows, early frameworks such as HaLoop [5] cache the results of one iteration as input to the next. Recent frameworks [15, 38, 25, 9] generalise this concept by permitting iteration over arbitrary parts of the dataflow graph, executing tasks repeatedly as part of loops. Similarly SDGs support iteration explicitly by permitting cycles in the dataflow graph.

Failure recovery. To recover from failure, frameworks either recompute state based on previous data or checkpoint state to restore it. For *recomputation*, Spark represents dataflows as RDDs [38], which can be recomputed deterministically based on their lineage. *Continuous dataflow* frameworks use techniques such as *upstream backup* [14] to reprocess buffered data after failure. Without checkpointing, recomputation can lead to long recovery times.

Checkpointing periodically saves state to disk or the memory of other nodes. With large state, this becomes resource-intensive. SEEP recovers state from memory, thus doubling the memory requirement of a cluster [10].

A challenge is how to take consistent checkpoints while processing data. *Synchronous global* checkpointing stops processing on all nodes to obtain consistent snapshots, thus reducing performance. For example, Naiad's "stop-the-world" approach exhibits low throughput with large state sizes [26]. *Asynchronous global* checkpointing, as used by Piccolo [30], permits nodes to take consistent checkpoints at different times.

Both techniques include all global state in a checkpoint and thus require all nodes to restore state after failure. Instead, SDGs use an asynchronous checkpointing mechanism with log-based recovery. As described in §5, it does not require global coordination between nodes during recovery, and it uses dirty state to minimise the disruption to processing during local checkpointing.

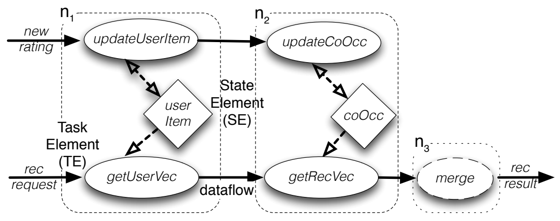


Figure 1: Stateful dataflow graph for CF algorithm

3 Stateful Dataflow Graphs

The goal of *stateful dataflow graphs* (SDGs) is to make it easy to translate imperative programs with mutable state to a dataflow representation that performs parallel, iterative computation with low latency. Next we describe their model (§3.1), how they support distributed state (§3.2) and how they are executed (§3.3).

3.1 Model

We explain the main features of SDGs using the CF algorithm from §2.1 as an example. As shown in Fig. 1, an SDG has two types of vertices: *task elements*, $t \in T$, transform input to output dataflows; and *state elements*, $s \in S$, represent the state in the SDG.

Access edges, $a = (t,s) \in A$, connect task elements to the state elements that they read or update. To facilitate the allocation of task and state elements to nodes, each task element can only access a single state element, i.e. A is a partial function: $(t_i, s_j) \in A, (t_i, s_k) \in A \Rightarrow s_j = s_k$. *Dataflows* are edges between task elements, $d = (t_i, t_j) \in D$, and contain data items.

Task elements (TEs) are not scheduled for execution but the entire SDG is materialised, i.e. each TE is assigned to one or more physical nodes. Since TEs are pipelined, it is unnecessary to generate the complete output dataflow of a TE before it is processed by the next TE. Data items are therefore processed with low latency, even across a sequence of TEs, without scheduling overhead, and fewer data items are handled during failure recovery (see §5).

The SDG in Fig. 1 has five TEs assigned to three nodes: the `updateUserItem`, `updateCoOcc` TEs realise the `addRating` function from Alg. 1; and the `getUserVec`, `getRecVec` and `merge` TEs implement the `getRec` function. We explain the translation process in §4.2.

State elements (SEs) encapsulate the state of the computation. They are implemented using efficient data structures, such as hash tables or indexed sparse matrices. In the next section, we describe the abstractions for distributed SEs, which span multiple nodes.

Fig. 1 shows the two SEs of the CF algorithm: the `userItem` and the `coOcc` matrices. The access edges specify that `userItem` is updated by the `updateUserItem` TE and read by the `getUserVec` TE; `coOcc` is updated by `updateCoOcc` and read by `getRecVec`.

Parallelism. For data-parallel processing, a TE t_i can be instantiated multiple times to handle parts of a dataflow, resulting in multiple TE instances, $\hat{t}_{i,j} : j \leq n_i$. As we explain in §3.3, the number of instances n_i for each TE is chosen at runtime and adjusted based on workload demands and the occurrence of stragglers.

An appropriate dispatching strategy sends items in dataflows to TE instances: items can be (i) partitioned using hash- or range-partitioning on a key; or (ii) dispatched to an *arbitrary* instance, e.g. in a round-robin fashion for load-balancing.

Iteration. In iterative algorithms, SEs are accessed multiple times by TEs. There are two cases to be distinguished: (i) if the repeated access is from a single TE, the iteration is entirely local and can be supported efficiently by a single node; and (ii) if the iteration involves multiple pipelined TEs, a *cycle* in the dataflow of the SDG can propagate updates between TEs.

With cycles in the dataflow, SDGs do not provide coordination during iteration by default. This is sufficient for many iterative machine learning and data mining algorithms because they can converge from different intermediate states [31], even without explicit coordination. A strong consistency model for SDGs could be realised with per-loop timestamps, as used by Naiad [26].

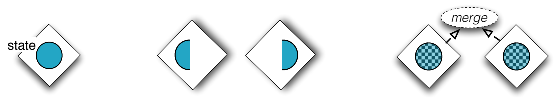
3.2 Distributed state

The SDG model provides abstractions for distributed state. An SE s_i may be distributed across nodes, leading to multiple SE instances $\hat{s}_{i,j}$, because (i) it is too large to fit into the memory of a single node; or (ii) it is accessed by a TE that has multiple instances to process the dataflow in parallel. This requires also multiple SE instances so that the TE instances access state locally.

Fig. 1 illustrates these two cases: (i) the `userItem` SE may grow larger than the main memory of a single node; and (ii) the data-parallel execution of the CPU-intensive `updateCoOcc` TE leads to multiple instances, each requiring local access to the `coOcc` SE.

An SE can be distributed in different ways, which are depicted in Fig. 2: a *partitioned* SE splits its internal data structure into disjoint partitions; if this is not possible, a *partial* SE duplicates its data structure, creating multiple copies that are updated independently. As we describe in §4, developers selected the required type of distributed state using source-level annotations according to the semantics of their algorithm.

Partitioned state. For algorithms for which state can be partitioned, SEs can be split and SE instances placed on separate nodes (see Fig. 2b). Access to the SE instances occurs in parallel.



(a) SE (b) Partitioned SE (c) Partial SE

Figure 2: Types of distributed state in SDGs

Developers can use predefined data structures for SEs (e.g. `Vector`, `HashMap`, `Matrix` and `DenseMatrix`) or define their own by implementing dynamic partitioning and dirty state support (see §5). Different data structures support different partitioning strategies: e.g. a map can be hash- or range-partitioned; a matrix can be partitioned by row or column. To obtain a unique partitioning, TEs cannot access partitioned SEs using conflicting strategies, such as accessing a matrix by row and by column.

In addition, the dataflow partitioning strategy must be compatible with the data access pattern by the TEs, as specified in the program (see §4.2). For example, multiple TE instances with an access edge to a partitioned SE must use the same partitioning key on the dataflow so that they access SE instances locally: in the CF algorithm, the `userItem` SE and the `new rating` and `rec request` dataflows must all be partitioned by row, i.e. the users for which ratings are maintained.

Partial state. In some cases, the data structure of an SE cannot be partitioned because the access pattern of TEs is arbitrary. For example, in the CF algorithm, the `coOcc` matrix has an access pattern, in which the `updateCoOcc` TE may update any row or column. In this case, an SE is distributed by creating multiple *partial* SE instances, each containing the whole data structure (see Fig. 2c). Partial SE instances can be updated independently by different TE instances.

When a TE accesses a partial SE, there are two possible types of accesses based on the semantics of the algorithm: a TE instance may access (i) the *local* SE instance on the same node; or (ii) the *global* state by accessing all of the partial SE instances, which introduces a synchronisation point. As we describe in §4.2, the type of access to partial SEs is determined by annotations.

When accessing all partial SE instances, it is possible to execute computation that *merges* their values, thus reconciling the differences between them. This is done by a `merge` TE that computes a single global value from partial SE instances. Merge computation is application-specific and must be defined by the developer. In the CF algorithm, the `merge` function takes all partial `userRec` vectors and computes a single recommendation vector.

3.3 Execution

To execute an SDG, the runtime system allocates TE and SE instances to nodes, creating instances on-demand.

Allocation to nodes. Since we want to avoid remote state access, the general strategy is to colocate TEs and

SEs that are connected by access edges on the same node. The runtime system uses four steps for mapping TEs and SEs to nodes: if there is a cycle in the SDG, all SEs accessed in the cycle are colocated if possible to reduce communication in iterative algorithms (step 1); the remaining SEs are allocated on separate nodes to increase available memory (step 2); TEs are colocated with the SEs that they access (step 3); and finally, any unallocated TEs are assigned to separate nodes (step 4).

Fig. 1 illustrates the above steps for allocating the SDG to nodes n_1 to n_3: since there are no cycles (step 1), the `userItem` SE is assigned to node n_1, and the `coOcc` SE is assigned to n_2 (step 2); the `updateUserItem` and `getUserVec` TEs are assigned to n_1, and the `updateCoOcc` and `getRecVec` TEs are assigned to n_2 (step 3); finally, the `merge` TE is allocated to a new node n_3 (step 4).

Runtime parallelism and stragglers. Processing bottlenecks in the deployed SDG, e.g. caused by the computational cost of TEs, cannot be predicted statically, and TEs instances may become stragglers [40]. Previous work [26] tries to reduce stragglers *proactively* for low latency processing, which is hard due to the many non-deterministic causes of stragglers.

Instead, similar to speculative execution in MapReduce [40], SDGs adopt a *reactive* approach. Using a dynamic dataflow graph approach [10], the runtime system changes the number of TE instances in response to stragglers. Each TE is monitored to determine if it constitutes a processing bottleneck that limits throughput. If so, a new TE instance is created, which may result in new partitioned or partial SE instances.

3.4 Discussion

With an explicit representation of state, a single SDG can express multiple workflows over that state. In the case of the CF algorithm from Alg. 1, the SDG processes new ratings by updating the SEs for the user/item and co-occurrence matrices, and also serves recommendation requests using the same SEs with low latency.

Without SDGs, these two workflows would require separate offline and online systems [23, 32]: a batch processing framework would incorporate new ratings periodically, and online recommendation requests would be served by a dedicated system from memory. Since it is inefficient to rerun the batch job after each new rating, the recommendations would be computed on stale data.

A drawback of the materialised representation of SDGs is the start-up cost. For short jobs, the deployment cost may dominate the running time. Our prototype implementation deploys an SDG with 50 TE and SE instances on 50 nodes within 7 s, and we assume that jobs are sufficiently long-running to amortise this delay.

4 Programming SDGs

We describe how to translate stateful Java programs statically to SDGs for parallel execution. We do not attempt to be completely transparent for developers or to address the general problem of automatic code parallelisation. Instead, we exploit data and pipeline parallelism by relying on source code annotations. We require developers to provide a single Java class with annotations that indicate how state is distributed and accessed.

4.1 Annotations

When defining a field in a Java class, a developer can indicate if its content can be *partitioned* or is *partial* by annotating the field declaration with `@Partitioned` or `@Partial`, respectively.

`@Partitioned`. This annotation specifies that a field can be split into disjoint partitions (see §3.2). A reference to a `@Partitioned` field always refers to a single partition. This requires that access to the field uses an access key to infer the partition. In the CF algorithm in Alg. 1, rows of the `userItem` matrix are updated with information about a single user only, and thus `userItem` can be declared as a partitioned field.

`@Partial`. Fields are annotated with `@Partial` if distributed instances of the field should be accessed independently (see §3.2). Partial fields enable developers to define distributed state when it cannot be partitioned. In CF, matrix `coOcc` is annotated with `@Partial`, which means that multiple instances of the matrix may be created, and each of them is updated independently for users in a partition (lines 10–11).

`@Global`. By default, a reference to a `@Partial` field refers to only one of its instances. While most of the time, computation should apply to one instance to make independent progress, it may also be necessary to support operations on *all* instances. A field reference annotated with `@Global` forces a Java expression to apply to all instances, denoting "global" access to a partial field, which introduces a synchronisation barrier in the SDG (see §4.2).

Java expressions deriving from `@Global` access become logically *multi-valued* because they include results from all instances of a partial field. As a result, any local variable that is assigned the result of a global field access becomes partial and must be annotated as such.

In CF, the access to the `coOcc` field carries the `@Global` annotation to compute *all* partial recommendations: each instance of `coOcc` is multiplied with the user rating vector `userRow`, and the results are stored in the partial local variable `userRec` (line 16).

`@Collection`. Global access to a partial field applies to all instances, but it hides the individual instances from the developer. At some point in the program, however, it may be necessary to reconcile all instances. The

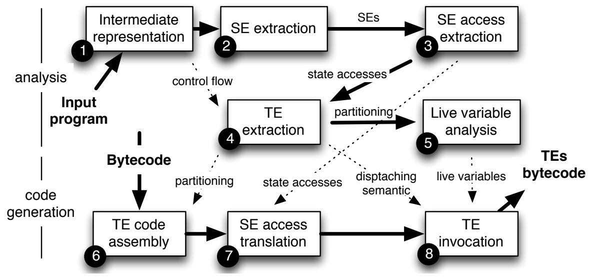


Figure 3: Translation of an annotated Java program to an SDG

`@Collection` annotation therefore exposes all instances of a partial field or variable as a Java array after `@Global` access. This enables the program to iterate over all values and, for example, merge them into a single value.

In CF, the partial recommendations are combined by accessing them using the `@Global` annotation and then invoking the `merge` method (line 17). The parameter of `merge` is annotated with `@Collection`, which specifies that the method can access all instances of the partial `userRec` variable to compute the final recommendation result.

Limitations. Java programs need to obey certain restrictions to be translated to SDGs due to their dataflow nature and fault tolerance properties:

Explicit state classes. All state in the program must be implemented using the set of SE classes (see §3.2). This gives the runtime system the ability to partition objects of these classes into multiple instances (for partitioned state) or distribute them (for partial state), and recover them after failure (see §5).

Location independence. Each object accessed in the program must support transparent serialisation/deserialisation: as SDGs are distributed, objects are propagated between nodes. The program also cannot make assumptions about its execution environment, e.g. by relying on local network sockets or files.

Side-effect-free parallelism. To support the parallel evaluation of multi-valued expressions under `@Global` state access, such expressions must not affect single-valued expressions. For example, the statement, `@Global coOcc.multiply(userRow)`, in line 16 in Alg. 1 cannot update `userRow`, which is single-valued.

Deterministic execution. The program must be deterministic, i.e. it should not depend on system time or random input. This enables the runtime system to re-execute computation when recovering after failure (see §5).

4.2 Translating programs to SDGs

Annotated Java programs are translated to SDGs by the `java2sdg` tool. Fig. 3 shows the steps performed by `java2sdg`: it first statically analyses the Java class to identify SEs, TEs and their access edges (steps 1–5); it then transforms the Java bytecode of the class to generate TE code, ready for deployment (steps 6–8).

1. SE generation. The class is compiled to *Jimple* code, a typed intermediate representation for static analysis used by the *Soot* framework [33] (step 1). The Jimple code is analysed to identify SEs with partitioned or partial fields and partial local variables (step 2). Based on the annotations in the code, access to SEs is classified as local, partitioned or global (step 3).

2. TE and dataflow generation. Next TEs are created so that each TE only accesses a single SE, i.e. a new TE is created from a block of code when access to a different SE or a different instance of the current SE is detected (step 4). The dispatching semantics of the dataflows between created TEs (i.e. partitioned, all-to-one, one-to-all or one-to-any) is chosen based on the type of state access. More specifically, a new TE is created:

1. for each entry point of the class;
2. when a TE uses *partitioned access* to a new SE (or to a previously-accessed SE with a new access key). The access key is extracted using reaching expression analysis, and the dataflow edge between the two TEs is annotated with the access key;
3. when a TE uses *global access* to a new partial SE. In this case, the dataflow edge between the two TEs is annotated with *one-to-all* dispatching semantics;
4. when a TE uses *local access* to a new partial SE, the dataflow edge is annotated with *one-to-any* dispatching semantics. In case of local (or partitioned) access after global access, all TE instances must be synchronised using a distributed barrier before control is transferred to the new TE, and the dataflow edge has *all-to-one* dispatching semantics; and
5. for `@Collection` expressions. A synchronisation barrier collects values from multiple TEs instances, and its dataflow edge has *all-to-one* semantics.

After generating the TEs, `java2sdg` identifies the variables that must propagate across TEs boundaries (step 5). For each dataflow, live variable analysis identifies the set of variables that are associated with that dataflow edge.

3. Bytecode generation. Next `java2sdg` synthesises the bytecode for each TE that will be executed by the runtime system. It compiles the code assigned with each TE in step 4 to bytecode and injects it into a TE template (step 6) using *Javassist*. State accesses to fields and partial variables are translated to invocations of the runtime system, which manages the SE instances (step 7).

Finally data dispatching across TEs is added (step 8): `java2sdg` injects code, (i) at the exit point of TEs, to serialise live variables and send them to the correct successor TE instance; and, (ii) at the entry point of a TE, to add barriers for all-to-one dispatching and to gather partial results for merge TEs.

5 Failure Recovery

To recover from failures, it is necessary to replace failed nodes and re-instantiate their TEs and SEs. TEs are stateless and thus are restored trivially, but the state of SEs must be recovered. We face the challenge of designing a recovery mechanism that: (i) can scale to save and recover the state of a large number of nodes with low overhead, even with frequent failures; (ii) has low impact on the processing latency; and (iii) achieves fast recovery time when recovering large SEs.

We achieve these goals with a mechanism that (a) combines *local checkpoints* with *message replay*, thus avoiding both global checkpoint coordination and global rollbacks; (b) divides state of SEs into *consistent state*, which is checkpointed, and *dirty state*, which permits continued processing while checkpointing; and (c) partitions checkpoints and saves them to multiple nodes, which enables *parallel recovery*.

Approach. Our failure recovery mechanism combines local checkpointing and message logging and is inspired by failure recovery in distributed stream processing systems [14]. Nodes periodically take checkpoints of their local SEs and output communication buffers. Dataflows include increasing TE-generated scalar timestamps, and a vector timestamp of the last data item from each input dataflow that modified the SEs is included in the checkpoint. Once the checkpoint is saved to stable storage, upstream nodes can trim their output buffers of data items that are older than all downstream checkpoints.

After failure, a node recovers its SEs from the last checkpoint, replays its output buffers and reprocesses data items received from the upstream output buffers. Downstream nodes detect duplicate data items based on the timestamps and discard them. This approach allows nodes to recover SEs locally beyond the last checkpoint, without requiring nodes to coordinate global rollback, and it avoids the output commit problem.

State checkpointing. We use an asynchronous parallel checkpointing mechanism that minimises the processing interruption when checkpointing large SEs with GBs of memory. The idea is to record updates in a separate data structure, while taking a checkpoint. For each type of data structure held by an SE, there must be an implementation that supports the separation of dirty state and its subsequent consolidation.

Checkpointing of a node works as follows: (1) to initiate a checkpoint, each SE is flagged as *dirty* and the output buffers are added to the checkpoint; (2) updates from TEs to an SE are now handled using a *dirty state* data structure: e.g. updates to keys in a dictionary are written to the dirty state, and reads are first served by the dirty state and, only on a miss, by the dictionary; (3) asynchronously to the processing, the now consistent state is added to the checkpoint; (4) the checkpoint is backed up to multiple nodes (see below); and (5) the SE is locked and its state is consolidated with the dirty state.

State backup and restore. To be memory-efficient, checkpoints must be stored on disk. We overcome the problem of low I/O performance by splitting checkpoints across *m* nodes. To reduce recovery time, a failed SE instance can be restored to *n* new partitioned SE instances in parallel. This *m*-to-*n* pattern prevents a single node from becoming a disk, network or processing bottleneck.

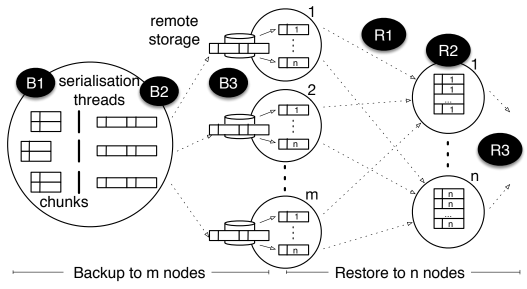


Figure 4: Parallel, *m*-to-*n* state backup and restore

Fig. 4 shows the distributed protocol for backing up checkpoints. In step B1, checkpoint chunks, e.g. obtained by hash-partitioning checkpoint data, are created, and a thread pool serialises them in parallel (step B2). Checkpoint chunks are streamed to *m* nodes, selected in a round-robin fashion (step B3). Nodes write received checkpoint chunks directly to disk.

After failure, *n* new nodes are instantiated with the lost TEs and SEs. Each node with a checkpoint chunk splits it into *n* partitions, each of which is streamed to one of the recovering instances (step R1). The new SE instances reconcile the chunks, reverting the partitioning (step R2). Finally, data items from output buffers are reprocessed to bring the recovered SE state up-to-date (step R3).

6 Evaluation

The goal of our experimental evaluation is to explore if SDGs can (i) execute stateful online processing applications with low latency and high throughput while supporting large state sizes with fine-grained updates (§6.1); (ii) scale in terms of nodes comparable to stateless batch processing frameworks (§6.2); handle stragglers at runtime with low impact on throughput (§6.3); and (iii) recover from failures with low overhead (§6.4).

We extend the SEEP streaming platform to implement SDGs and deploy our prototype on Amazon EC2 and a private cluster with 7 quad-core 3.4 GHz Intel Xeon servers with 8 GB of RAM. To support fast recovery, the checkpointing frequency for all experiments is 10 s unless stated otherwise. Candlesticks in plots show the 5th, 25th, 50th, 75th and 95th percentiles, respectively.

6.1 Stateful online processing

Throughput and latency. First we investigate the performance of SDGs using the *online collaborative filtering* (CF) application (see §2.1). We deploy it on 36 EC2 VM instances ("c1.xlarge"; 8 vCPUs with 7 GB) using the Netflix dataset, which contains 100 million movie

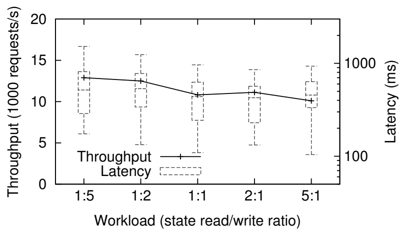


Figure 5: Throughput and latency with different read/write ratios (online collaborative filtering)

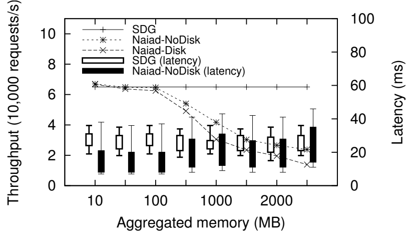


Figure 6: Throughput and latency with increasing state size on single node (key/value store)

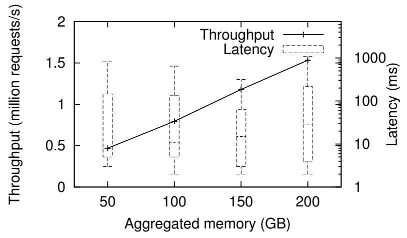


Figure 7: Throughput and latency with increasing state size on multiple nodes (key/value store)

ratings for evaluating recommender systems. We add new ratings continuously (addRating), while requesting fresh recommendations (getRec). The state size maintained by the system grows to 12 GB.

Fig. 5 shows the throughput of getRec and addRating requests and the latencies of getRec requests when the ratio between the two is varied. The achieved throughput is sufficient to serve 10,000–14,000 requests/s, with the 95th percentile of responses being at most 1.5 s stale. As the workload ratio includes more state reads (getRec), the throughput decreases slightly due to the cost of the synchronisation barrier that aggregates the partial state in the SDG. The result shows that SDGs can combine the functionality of a batch and an online processing system, while serving fresh results with low latency and high throughput over large mutable state.

State size. Next we evaluate the performance of SDGs as the state size increases. As a synthetic benchmark, we implement a distributed partitioned *key/value store* (KV) using SDGs because it exemplifies an algorithm with pure mutable state. We compare to an equivalent implementation in Naiad (version 0.2) with global checkpointing, which is the only fault-tolerance mechanism available in the open-source version. We deploy it in one VM ("m1.xlarge") and measure the performance of serving update requests for keys.

Fig. 6 shows that, for a small state size of 100 MB, both SDGs and Naiad exhibit similar throughput of 65,000 requests/s with low latency. As the state size increases to 2.5 GB, the SDG throughput is largely unaffected but Naiad's throughput decreases due to the overhead of its disk-based checkpoints (Naiad-Disk). Even with checkpoints stored on a RAM disk (Naiad-NoDisk), its throughput with 2.5 GB of state is 63% lower than that of SDGs. Similarly, the 95th percentile latency in Naiad increases when it stops processing during checkpointing—SDGs do not suffer from this problem.

To investigate how SDGs can support large distributed state across multiple nodes, we scale the KV store by increasing the number of VMs from 10 to 40, keeping the number of dictionary keys per node constant at 5 GB.

Fig. 7 shows the throughput and the latency for read requests with a given total state size. The aggregate throughput scales near linearly from 470,000 requests/s for 50 GB to 1.5 million requests/s for 200 GB. The median latency increases from 8–29 ms, while the 95th percentile latency varies between 800 ms and 1000 ms.

This result demonstrates that SDGs can support stateful applications with large state sizes without compromising throughput or processing latency, while executing in a fault-tolerant fashion.

Update granularity. We show the performance of SDGs when performing frequent, fine-grained updates to state. For this, we deploy a streaming *wordcount* (WC) application on 4 nodes in our private cluster. WC reports the word frequencies over a wall clock time window while processing the Wikipedia dataset. We compare to WC implementations in Streaming Spark [39] and Naiad.

We vary the size of the window, which controls the granularity at which input data updates the state: the smaller the window size, the less batching can be done when updating the state. Since Naiad permits the configuration of the batch size independently of the window size, we use a small batch size (1000 messages) for low-latency (Naiad-LowLatency) and a large one (20,000 messages) for high-throughput processing (Naiad-HighThroughput).

Fig. 8 shows that only SDG and Naiad-LowLatency can sustain processing for all window sizes, but SDG has a higher throughput due to Naiad's scheduling overhead. The other deployments suffer from the overhead of micro-batching: Streaming Spark has a throughput similar to SDG, but its smallest sustainable window size is 250 ms, after which its throughput collapses; Naiad-HighThroughput achieves the highest throughput of all, but it also cannot support windows smaller than 100 ms. This shows that SDGs can perform fine-grained state updates without trading off throughput for latency.

6.2 Scalability

We explore if SDGs can scale to higher throughput with more nodes in a batch processing scenario. We deploy an implementation of *logistic regression* (LR) [21] on EC2 ("m1.xlarge"; 4 vCPUs with 15 GB). We compare to LR from Spark [38], which is designed for iterative processing, using the 100 GB dataset provided in its release.

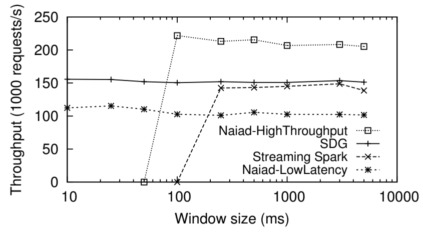


Figure 8: Latency with different window sizes (streaming wordcount)

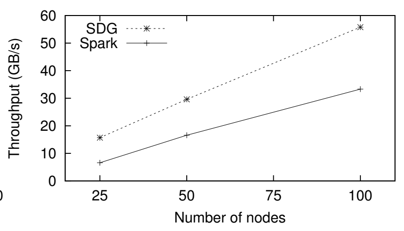


Figure 9: Scalability in terms of throughput (batch logistic regression)

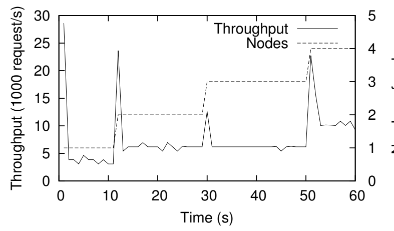


Figure 10: Runtime parallelism for handling stragglers (collaborative filtering)

Fig. 9 shows the throughput of our SDG implementation and Spark for 25–100 nodes. Both systems exhibit linear scalability. The throughput of SDGs is higher than Spark, which is likely due to the pipelining in SDGs, which avoids the re-instantiation of tasks after each iteration. With higher throughput, iterations are shorter, which leads to a faster convergence time. We conclude that the management of partial state in the LR application does not limit scalability compared to existing stateless dataflow systems.

6.3 Stragglers

We explore how SDGs handle straggling nodes by creating new TE and SE instances at runtime (see §3.3). For this, we deploy the CF application on our cluster and include a less powerful machine (2.4 GHz with 4 GB).

Fig. 10 shows how the throughput and the number of nodes changes over time as bottleneck TEs are identified by the system. At the start, a single instance of the `getRecVec` TE is deployed. It is identified as a bottleneck, and a second instance is added at $t = 10$ s, which also causes a new instance of the partial state in the `coOcc` matrix to be created. This increases the throughput from 3600–6200 requests/s. The throughput spikes occur when the input queues of new TE instances fill up.

Since the new node is allocated on the less powerful machine, it becomes a straggler, limiting overall throughput. At $t = 30$ s, adding a new TE instance without relieving the straggler does not increase the throughput. At $t = 50$ s, the straggling node is detected by the system, and a new instance is created to share its work. This increases the throughput from 6200–11,000 requests/s.

This shows how straggling nodes are mitigated by allocating new TE instances on-demand, distributing new partial or partitioned SE instances as required. In more extreme cases, a straggling node could even be removed and the job resumed from a checkpoint with new nodes.

6.4 Failure recovery

We evaluate the performance and overhead of our failure recovery mechanism for SDGs. We (i) explore the recovery time under different recovery strategies; (ii) assess the advantages of our asynchronous checkpointing mechanism; and (iii) investigate the overhead with different checkpointing frequencies and state sizes. We deploy the KV store on one node of our cluster, together with spare nodes to store backups and replace failed nodes.

Recovery time. We fail the node under different recovery strategies: an *m*-to-*n* recovery strategy uses *m* backup nodes to restore to *n* recovered nodes (see §5). For each, we measure the time to restore the lost SE, re-process unprocessed data and resume processing.

Fig. 11 shows the recovery times for different SE sizes under different strategies: (i) the simplest strategy, 1-to-1, has the longest recovery time, especially with large state sizes, because the state is restored from a single node; (ii) the 2-to-1 strategy streams checkpoint chunks from two nodes in parallel, which improves disk I/O throughput but also increases the load on the recovering node when it reconstitutes the state; (iii) in the 1-to-2 strategy, checkpoint chunks are streamed to two recovering nodes, thus halving the load of state reconstruction; and (iv) the 2-to-2 strategy recovers fastest because it combines the above two strategies—it parallelises both the disk reads and the state reconstruction.

As the state becomes large, state reconstruction dominates over disk I/O overhead: with 4 GB, streaming from two disks does not improve recovery time. Adopting a strategy that recovers a failed node with multiple nodes, however, has significant benefit, compared to cases with smaller state sizes.

Synchronous vs. asynchronous checkpointing. We investigate the benefit of our asynchronous checkpointing mechanism in comparison with synchronous checkpointing that stops processing, as used by Naiad [26] and SEEP [10].

Fig. 12 compares the throughput and 99th percentile latency with increasing state sizes. As the checkpoint size grows from 1–4 GB, the average throughput under synchronous checkpointing reduces by 33%, and the latency increases from 2–8 s because the system stops processing while checkpointing. With asynchronous checkpointing, there is only a small (~5%) impact on throughput. Latency is an order of magnitude lower and only moderately affected (from 200–500 ms). This result shows that a synchronous checkpointing approach cannot achieve low-latency processing with large state sizes.

Overhead of asynchronous checkpointing. Next we evaluate the overhead of our checkpointing mechanism

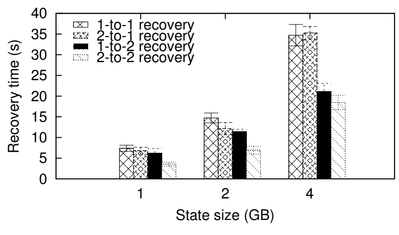


Figure 11: Recovery times with different *m*-to-*n* recovery strategies

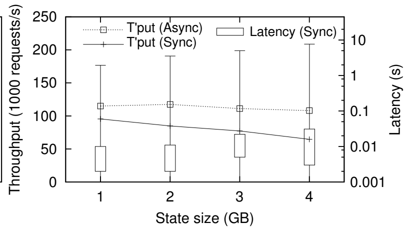


Figure 12: Comparison of sync. and async. checkpointing

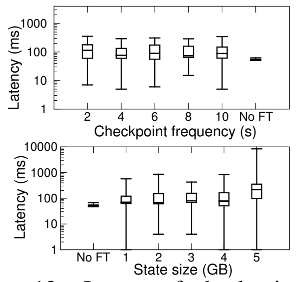


Figure 13: Impact of checkpoint frequency and size on latency

as a function of checkpointing frequency and state size.

Fig. 13 (top) shows the processing latency when varying the checkpointing frequency. The rightmost data point (No FT) represents the case where the checkpointing mechanism is disabled. The bottom figure reports the impact of the size of the checkpoint on latency.

Checkpointing has a limited impact on latency: without fault tolerance, the 95th percentile latency is 68 ms, and it increases to 500 ms when checkpointing 1 GB every 10 s. This is due to the overhead of merging dirty state and saving checkpoints to disk. Increasing the checkpointing frequency or size gradually also increases latency: the 95th percentile latency with 4 GB is 850 ms, while checkpointing 2 GB every 4 s results in 1 s.

Beyond that, the checkpointing overhead starts to impact higher percentiles more significantly. Checkpointing frequency and size behave almost proportionally: as the state size increases, the frequency can be reduced to maintain a low processing latency.

Overall this experiment demonstrates the strength of our checkpointing mechanism, which only locks state while merging dirty state. The locking overhead thus reduces proportionally to the state update rate.

7 Related Work

Programming model. Data-parallel frameworks typically support a functional/declarative model: MapReduce [8] only has two higher-order functions; more recent frameworks [15, 38, 13] permit user-defined functional operators; and Naiad [26] supports different functional and declarative programming models on top of its timely dataflow model. CBP [19], Storm and SEEP [10] expose a low-level dataflow programming model: algorithms are defined as a dataflow pipeline, which is harder to program and debug. While functional and dataflow models ease distribution and fault tolerance, SDGs target an imperative programming model, which remains widely used by data scientists [17].

Efforts exist to bring imperative programming to data-parallel processing. CIEL [25] uses imperative constructs such as task spawning and futures, but this exposes the low-level execution of the dynamic dataflow graph to developers. Piccolo [30] and Oolong [24] offer imperative compute kernels with distributed state, which requires algorithms to be structured accordingly.

In contrast, SDGs simplify the translation of imperative programs to dataflows using basic program analysis techniques, which infer state accesses and the dataflow. By separating different types of state access, it becomes possible to choose automatically an effective implementation for distributed state.

GraphLab [20] and Pregel [22] are frameworks for graph computations based on a shared-memory abstraction. They expose a vertex-centric programming model whereas SDGs target generic stateful computation.

Program parallelisation. Matlab has language constructs for parallel processing of large datasets on clusters. However, it only supports the parallelisation of sequential blocks or iterations and not of general dataflows.

Declarative models such as Pig [28], DyradLINQ [37], SCOPE [6] and Stratosphere [9] are naturally amenable to automatic parallelisation—functions are stateless, which allows data-parallel versions to execute on multiple nodes. Instead, we focus on an imperative model.

Other approaches offer new programming abstractions for parallel computation over distributed state. FlumeJava [7] provides distributed immutable collections. While immutability simplifies parallel execution, it limits the expression of imperative algorithms. In Piccolo [30], global mutable state is accessed remotely by parallel distributed functions. In contrast, tasks in SDGs only access local state with low latency, and state is always colocated with computation. Presto [35] has distributed partitioned arrays for the R language. Partitions can be collected but not updated by multiple tasks, whereas SDGs permit arbitrary dataflows.

Extracting parallel dataflows from imperative programs is a hard problem [16]. We follow an approach similar to that of Beck et al. [3], in which a dataflow graph is generated compositionally from the execution graph. While early work focused on hardware-based dataflow models [27], more recent efforts target thread-based execution [18]. Our problem is simpler because we do not extract task parallelism but only focus on data and pipeline parallelism in relation to distributed state access.

Similar to pragma-based techniques [34], we use annotations to transform access to distributed state into access to local instances. Blazes [2] uses annotations to generate automatically coordination code for distributed programs. Our goal is different: SDGs execute imperative code in a distributed fashion, and coordination is determined by the extracted dataflow.

Failure recovery. In-memory systems are prone to failures [1], and fast recovery is important for low-latency and high-throughput processing. With large state sizes, checkpoints cannot be stored in memory, but storing them on disk can increase recovery time. RAMCloud [29] replicates data across cluster memory and eventually backs it up to persistent storage. Similar to our approach, data is recovered from multiple disks in parallel. However, rather than replicating each write request, we checkpoint large state atomically, while permitting new requests to operate on dirty state.

Streaming Spark [39] and Spark [38] use RDDs for recovery. After a failure, RDDs are recomputed in parallel on multiple nodes. Such a recovery mechanism is effective if recomputation is inexpensive—for state that depends on the entire history of the data, it would be prohibitive. In contrast, the parallel recovery in SDGs retrieves partitioned checkpoints from multiple nodes, and only reprocesses data from output buffers to bring restored SE instances up-to-date.

8 Conclusions

Data-parallel processing frameworks must offer a familiar programming model with good performance. Supporting imperative online machine learning algorithms poses challenges to frameworks due to their use of large distributed state with fine-grained access.

We describe stateful dataflow graphs (SDGs), a data-parallel model that is designed to offer a dataflow abstraction over large mutable state. With the help of annotations, imperative algorithms can be translated to SDGs, which manage partitioned or partial distributed state. As we demonstrated in our evaluation, SDGs can support diverse stateful applications, thus generalising a number of existing data-parallel computation models.

Acknowledgements. This work was supported by a PhD CASE Award funded by EPSRC/BAE Systems (Nigel Wheadon). We thank our PC contact, Jinyang Li, and the ATC reviewers for their feedback and guidance.

References

[1] AKIDAU, T., BALIKOV, A., ET AL. MillWheel: Fault-Tolerant Stream Processing at Internet Scale. In *VLDB* (2013).

[2] ALVARO, P., CONWAY, N., ET AL. Blazes: Coordination Analysis for Distributed Programs. In *ICDE* (2014).

[3] BECK, M., AND PINGALI, K. From Control Flow to Dataflow. In *ICPP* (1990).

[4] BHATOTIA, P., WIEDER, A., ET AL. Incoop: MapReduce for Incremental Computations. In *SOCC* (2011).

[5] BU, Y., HOWE, B., ET AL. HaLoop: Efficient Iterative Data Processing on Large Clusters. In *VLDB* (2010).

[6] CHAIKEN, R., JENKINS, B., ET AL. SCOPE: Easy and Efficient Parallel Processing of Massive Data Sets. In *VLDB* (2008).

[7] CHAMBERS, C., RANIWALA, A., ET AL. FlumeJava: Easy, Efficient Data-Parallel Pipelines. In *PLDI* (2010).

[8] DEAN, J., AND GHEMAWAT, S. MapReduce: Simplified Data Processing on Large Clusters. In *CACM* (2008).

[9] EWEN, S., TZOUMAS, K., ET AL. Spinning Fast Iterative Data Flows. In *VLDB* (2012).

[10] FERNANDEZ, R. C., MIGLIAVACCA, M., ET AL. Integrating Scale Out and Fault Tolerance in Stream Processing using Operator State Management. In *SIGMOD* (2013).

[11] GUNDA, P. K., RAVINDRANATH, L., ET AL. Nectar: Automatic Management of Data and Comp. in Datacenters. In *OSDI* (2010).

[12] HE, B., YANG, M., ET AL. Comet: Batched Stream Processing for Data Intensive Distributed Computing. In *SOCC* (2010).

[13] HUESKE, F., PETERS, M., ET AL. Opening the Black Boxes in Data Flow Optimization. In *VLDB* (2012).

[14] HWANG, J.-H., BALAZINSKA, M., ET AL. High-Availability Algorithms for Distributed Stream Processing. In *ICDE* (2005).

[15] ISARD, M., BUDIU, M., ET AL. Dryad: Dist. Data-Parallel Programs from Sequential Building Blocks. In *EuroSys* (2007).

[16] JOHNSTON, W. M., HANNA, J., ET AL. Advances in Dataflow Programming Languages. In *CSUR* (2004).

[17] KDNUGGETS ANNUAL SOFTWARE POLL. RapidMiner and R vie for the First Place. `http://goo.gl/OLikb`, 2013.

[18] LI, F., POP, A., ET AL. Automatic Extraction of Coarse-Grained Data-Flow Threads from Imperative Programs. In *Micro* (2012).

[19] LOGOTHETHIS, D., OLSON, C., ET AL. Stateful Bulk Processing for Incremental Analytics. In *SOCC* (2010).

[20] LOW, Y., BICKSON, D., ET AL. Dist. GraphLab: A Framework for ML and Data Mining in the Cloud. In *VLDB* (2012).

[21] MA, J., SAUL, L. K., ET AL. Identifying Suspicious URLs: an Application of Large-Scale Online Learning. In *ICML* (2009).

[22] MALEWICZ, G., AUSTERN, M. H., ET AL. Pregel: A System for Large-scale Graph Processing. In *SIGMOD* (2010).

[23] MISHNE, G., DALTON, J., ET AL. Fast Data in the Era of Big Data: Twitter's Real-Time Related Query Suggestion Architecture. In *SIGMOD* (2013).

[24] MITCHELL, C., POWER, R., ET AL. Oolong: Asynchronous Distributed Applications Made Easy. In *APSYS* (2012).

[25] MURRAY, D., SCHWARZKOPF, M., ET AL. CIEL: A Universal Exec. Engine for Distributed Data-Flow Comp. In *NSDI* (2011).

[26] MURRAY, D. G., MCSHERRY, F., ET AL. Naiad: A Timely Dataflow System. In *SOSP* (2013).

[27] NIKHIL, R. S., ET AL. Executing a Program on the MIT Tagged-Token Dataflow Architecture. In *TC* (1990).

[28] OLSTON, C., REED, B., ET AL. Pig Latin: A Not-So-Foreign Language for Data Processing. In *SIGMOD* (2008).

[29] ONGARO, D., RUMBLE, S. M., ET AL. Fast Crash Recovery in RAMcloud. In *SOSP* (2011).

[30] POWER, R., AND LI, J. Piccolo: Building Fast, Distributed Programs with Partitioned Tables. In *OSDI* (2010).

[31] SCHELTER, S., EWEN, S., ET AL. All Roads Lead to Rome: Optimistic Recovery for Distributed Iterative Data Processing. In *CIKM* (2013).

[32] SUMBALY, R., KREPS, J., ET AL. The Big Data Ecosystem at LinkedIn. In *SIGMOD* (2013).

[33] VALLÉE-RAI, R., HENDREN, L., ET AL. Soot: A Java Optimization Framework. In *CASCON* (1999).

[34] VANDIERENDONCK, H., RUL, S., ET AL. The Paralax Infrastructure: Automatic Parallelization with a Helping Hand. In *PACT* (2010).

[35] VENKATARAMAN, S., BODZSAR, E., ET AL. Presto: Dist. ML and Graph Processing with Sparse Matrices. In *EuroSys* (2013).

[36] XIN, R. S., ROSEN, J., ET AL. Shark: SQL and Rich Analytics at Scale. In *SIGMOD* (2013).

[37] YU, Y., ISARD, M., ET AL. DryadLINQ: a System for General-Purpose Distributed Data-Parallel Computing using a High-Level Language. In *OSDI* (2008).

[38] ZAHARIA, M., CHOWDHURY, M., ET AL. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *NSDI* (2012).

[39] ZAHARIA, M., DAS, T., ET AL. Discretized Streams: Fault-tolerant Streaming Computation at Scale. In *SOSP* (2013).

[40] ZAHARIA, M., KONWINSKI, A., ET AL. Improving MapReduce Performance in Heterogeneous Environments. In *OSDI* (2008).