A Survey of State Management in Big Data Processing Systems

Quoc-Cuong To1

Juan Soto^{1,2}

Volker Markl^{1,2}

¹DFKI Alt-Moabit 91c 10559 Berlin, Germany ²Technische Universität Berlin
 Sekr. EN 7 Raum 728, Einsteinufer 17
 10587 Berlin, Germany

quoc cuona.to@dfki.de

juan.soto@tu-berlin.de

volker.markl@tu-berlin.de

ABSTRACT

State management and its use in diverse applications varies widely across big data processing systems. This is evident in both the research literature and existing systems, such as Apache Flink, Apache Samza, Apache Spark, and Apache Storm. Given the pivotal role that state management plays in various use cases, in this survey, we present some of the most important uses of state as an enabler, discuss the alternative approaches used to handle and implement state, propose a taxonomy to capture the many facets of state management, and highlight new research directions. Our aim is to provide insight into disparate state management techniques, motivate others to pursue research in this area, and draw attention to some open problems.

CCS Concepts

computing methodologies; database management systems; information systems; massively parallel and high-performance computer systems

Keywords

state management, big data processing, taxonomy, and survey

1. Introduction

Big data systems process massive amounts of data efficiently often with fast response times and are generally classified by their data processing approach (i.e., batch-oriented vs. stream-oriented). In batch-oriented systems, processing occurs on chunks of large data files, whereas in stream-oriented systems, processing happens on continuously arriving data.

One of the first proposals for batch-oriented data processing (BDP) is MapReduce [24], which becomes popularized via Hadoop, an open source batch processing system, due to its features, including *flexibility*, *fault-tolerance*, *programming ease*, and *scalability*. Today, it is widely regarded as the pioneer for large-scale batch-oriented data analysis. However, despite its merits, MapReduce has several drawbacks, such as redundant processing and a lack of support for iterations, which severely affect performance. Consequently, alternatives are proposed to overcome these limitations. Among them are the BDP approaches surveyed by Doulkeridis et al. [28] and novel data stream

processing solutions, such as Apache Flink [19] (formerly, Stratosphere [6]) and Apache Spark [116], which arise to meet the needs of an ever-increasing number of real-time applications demanding both *low latency* and *high throughput*.

Big data processing systems encompass a wide range of concepts. *Operators*, for example, are one of the most important elements that directly affect data. There are *stateless* operators (e.g., select, project, filter) and *stateful* operators (e.g., sort, join, aggregate). Stateless operators work on current data and do not depend on or carry over intermediate information from previous processing steps. Stateful operators, on the other hand, maintain an internal data structure called *state¹*. This structure preserves the history of past operations and affects the processing logic in subsequent computations.

Some large-scale BDP frameworks, such as MapReduce, hinder programmers from fully using state in distributed programs. Earlier approaches, such as online MapReduce systems [23] and Twister [29], which incorporate state "can result in custom, fragile code and disappointing performance," as stated by Logothetis et al. [67]. Thus, some stream processing frameworks, such as Apache Flink, are proposed to smoothly incorporate state into programs. Since then, many researchers have proposed new ways to represent, manage, and use state. For example, windowing is the main abstraction used to implement the state of operators in data stream processing, as reflected by Matteis and Mencagli [72]. In contrast, Fernandez et al. [36] present alternative data structures that can be used to represent the various state types.

State usage has increasingly received much attention in recent years. Systems researchers are actively seeking to address key questions, such as "How to efficiently use state in a specific context?" and "How to fully exploit functionalities of state for many applications?" Indeed, state can be used in various scenarios, as portrayed in **Figure 1**, which depicts our taxonomy for state management. In total, there are ten distinct scenarios, falling into four primary categories, namely, *applications*, *handling*, *overhead complexity*, and *implementation*.

¹ Roy and Haridi [123] define **state** to be "a sequence of values in time that contain the intermediate results of a desired computation."

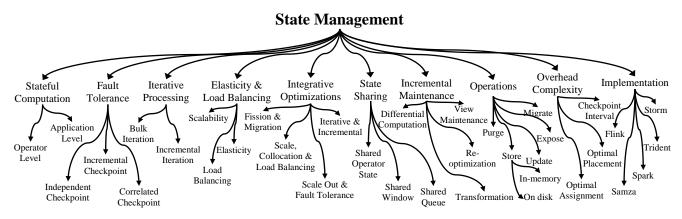


Figure 1. A Taxonomy for State Management.

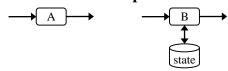
Among the applications of state there are seven scenarios. Stateful computation, for example, requires both the input data and the current state value. To achieve fault tolerance, state must be kept in a reliable location, to enable efficient recovery. State can facilitate the iterative processing of most machine learning algorithms, where iteration is inevitable. State can also aid in achieving elasticity and load balancing. Furthermore, state can also leverage integrative optimizations, including fission, migration, and scaling. The remaining two scenarios, state sharing and incremental maintenance are briefly discussed below.

The efficient handling of state presents numerous technical challenges, including the matters of *state sharing*, *incremental maintenance*, and *operations*. For example, state can be: (i) *shared among different processes* [15] to save storage, (ii) *maintained incrementally* [33] to speed-up performance, (iii) *stored remotely or locally*, using in-memory [83, 92] or disk spilling [63] techniques, (iv) *maintained geographically across distant locations* [8], (v) *migrated among operators or nodes in a cluster* [26, 35], and (vi) *exposed to programmers for easier use* [36, 111].

Moreover, managing state incurs significant *overhead complexity*, including processing latency and recovery time. Hence, varying approaches have been proposed to reduce the overhead. For example, setting the intervals among checkpoints appropriately² can reduce the execution time [94]. Lastly, the *implementation* of state can be examined in today's leading big data processing frameworks, such as Flink³, Spark⁴, Storm⁵ [102], Trident⁶, and Samza⁷. Unfortunately, none of these frameworks have addressed all the above-mentioned issues. Thus, a novel solution is sought to reduce the complexity, lower processing latency, and enable fast recovery. Today, researchers [94] are actively working on solving this problem and in this endeavor *adaptive checkpointing* holds promise.

The rest of this paper is structured as follows. An overview of the uses of state as an enabler in varying settings, i.e., *stateful computation*, *fault tolerance*, *iterative computation*, *elasticity* and *load balancing*, and *integrative optimizations* is presented in Sections 2-6, correspondingly. Approaches for *sharing* and

2. State for Stateful Computation



Naturally, *state* serves to enable stateful computation. Although streaming computation can either be *stateless*⁸ or *stateful*, the focus of this section is on the latter. Stateful operators (e.g., aggregations over time windows or some other stream discretization), by definition, must interact with earlier computations. Thus, *state* must be saved in persistent storage for subsequent use. This is evident in today's popular frameworks, such as Flink, Spark, Storm, and Trident, each of which supports stateful operators.

Despite commonalities among frameworks, there are contrasting views on how to best implement state. For example, early versions of **Storm** focus on stateless processing and require state management at the application level. **Trident**, an extension of **Storm**, enables state management via an API. **Samza** manages large states using a local database to enable persistence. **Spark Streaming** enables state computation via DStream (i.e., discretized streams). Finally, **Flink** treats state as a first-class citizen, which eases stateful application development. In Section 11, we compare state implementations among these five frameworks in more detail. Next, we discuss four representative papers on stateful computation.

maintaining state are discussed in Sections 7-8, respectively. To help the reader to develop their intuition, a graphic is incorporated (akin to Hirzel et al. [48]) at the beginning of Sections 2-5 and 7-8 to illuminate the corresponding state usage case. Strategies for storing, updating, migrating, purging, and exposing state are described in Section 9. The overhead and complexity of state management are considered in Section 10. The state of today's leading frameworks, including their limitations are examined in Section 11. Promising new research directions are underscored in Section 12. Finally, closing remarks are offered in Section 13.

² The *state checkpoint placement* problem has been shown to be NP-complete [13].

³ https://flink.apache.org/

⁴ http://spark.apache.org/

⁵ http://storm.apache.org/

⁶ http://storm.apache.org/releases/1.0.1/Trident-state.html

http://samza.apache.org/

⁸ In stateless operators (e.g., filtering), there is no record of previous computations. Instead, each computation is handled entirely based on the current input.

In the late 2000s, bulk data processing systems, like MapReduce, were growing in popularity. However, they were criticized for not offering data indexing, which could conceivably increase performance. These findings lead Logothetis et al. [66] to devise a data indexing scheme to support *stateful groupwise processing*. They observe that by offering access to persistent state, operations, such as reduce, could cope with data updates and circumvent the need to recompute from scratch. Additionally, that indexing can avoid expensive sequential scans and grant groupwise processing random access to state.

Logothetis et al. [67] discuss two (suboptimal) solutions for stateful bulk processing. One solution requires running the entire dataflow once again, whenever new data arrives. In contrast, the other solution requires programmers to employ data-parallel programs, to incorporate and use state. However, due to limitations in tools, such as MapReduce, this will be difficult. Instead, they propose an alternative approach by treating state as an explicit input that can *store* and *retrieve* as new data arrives. Moreover, by employing a stateful groupwise operator (i.e., *translate*), data movement is minimized and state is smoothly integrated into a data-parallel processing framework.

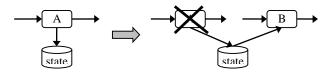
Gedik et al. [43] endeavour to exploit partitioning functions (PF) for stateful data parallelism in stream processing systems, to improve application throughput. They note that *partitioned stateful operators*⁹ are well-suited for data parallelism and demonstrate that these can hold state on partitioning-key defined sub-streams. Further, that hash functions must be employed when partitioning, to ensure that tuples with the same partitioning key value are routed to the same parallel channel, consistently. Consequently, that PF enable adequate *memory load balance*, *communication*, and *computation*, while concurrently maintaining the migration overhead low under a variety of workloads.

Matteis et al. [72] employ *algorithmic skeletons*¹⁰ to address parallelism challenges involving stateful operators arising in modern stream processing engines (e.g., Spark Streaming, Storm). They present four parallel patterns for window-based stateful operators on data streams: *window farming*, *key partitioning*, *pane farming*, and *window partitioning*.

The window farming pattern (WFP) applies each computation (e.g., a function) to a window and the corresponding results will be independent of one another. The key partitioning pattern extends the WFP by adding a constrained assignment policy. In this policy, the same worker processes windows originating from the common sub-stream sequentially, however, this limits the parallelism.

The pane farming pattern splits each window into non-overlapping partitions called *panes*. This fine-grained division increases throughput and decreases latency by sharing the results of overlapping panes. Finally, the window partitioning pattern requires multiple workers to process each individual window. Akin to pane farming, this pattern improves throughput and reduces latency. However, this latency reduction depends on the total number of workers, in contrast to the pane farming pattern, which does not.

3. State for Fault Tolerance



State can be used to enable fault tolerance and facilitate failure recovery. It is persisted in reliable storage and updated periodically. When failure occurs, big data processing systems restore the state to another node, thereby, recovering the computation from the last checkpoint. Each state includes *internal data*, which is placed inside operators, and *external data*, which is contained in input and output queues. Therefore, to checkpoint state, we must save all data to stable storage.

Fault tolerance, in general, requires redundancy, which can be achieved in several ways. One approach enables the redundant storage (or replication) of computations. A second approach enables the redundant storage of the computational logic, which involves a variant of state called *lineage* (e.g., prevalent in Spark). Alternatively, a third approach employs redundant computation [95], which exploits algorithmic properties and does not use state.

According to Hwang et al. [49], there are three fault-tolerance mechanisms: passive standby, active standby, and upstream backup. In the case of passive standby, only the modified part of the state is backed up periodically. In the case of active standby, redundant execution enables each backup server to receive and process the same input from upstream servers, in parallel, as its primary server. Lastly, in the case of upstream backup, each primary server retains its output, while the backup is still inactive. If a primary server fails, the backup restores the primary server's state by reprocessing tuples stored at upstream servers.

Each method has its own advantages, in terms of network usage, recovery latency, recovery semantics, and system performance [49]. Most researchers prefer passive standby (or checkpointing), to achieve fault-tolerance because it is effective in addressing more configuration and workload needs than the alternative approaches [50]. Additionally, this method reduces the overall recovery overhead, since each checkpoint can be restored in parallel.

Orthogonal to the taxonomy of Hwang et al. [49], we classify fault-tolerance methods into three key categories, i.e., independent, dependent, and incremental. These categories are determined via the state handling approach employed, as a classification criterion.

3.1 State for Independent Checkpointing

In the research literature, there are two types of failures, namely, *independent* failures and *correlated* failures. The assumption is that failures are either independent of one another or occur simultaneously, i.e., correlated.

Hwang et al. [50] introduce the concept of a *maximal* connected subgraph, which is regarded as an atomic (i.e., a high-availability or HA) unit for independent checkpointing. These units can be checkpointed onto independent servers at varying times, since they have no interdependencies and thus avoid inconsistent backup checkpoints. Consequently, spreading out independent

⁹ Examples of partitioned stateful operators include: streaming aggregation, one-way join, progressive sort, and those that are user-defined.

Algorithmic skeletons (a.k.a. parallelism patterns) are a high-level parallel programming model for parallel and distributed computing. They are useful in hiding the complexity parallel and distributed applications.

checkpoints to multiple servers can reduce the checkpointing overhead.

Comparably, Kwon et al. [57] split state into partitions and can independently checkpoint states, while ensuring consistency, in the event of failures. By splitting operator state into disparate parts, Sebepou et al. [96] produce independent partial checkpoints asynchronously. These independent checkpoints, in the form of control tuples are integrated with regular tuples in the operator's output queue.

3.2 State for Correlated Checkpointing

Correlated failure events involve the simultaneous failure of multiple nodes. They generally occur, whenever switches, routers, or electrical power fail. Indeed, whenever failures occur in bursts, varying coping strategies [20, 46, 58, 100, 105, 108] have been proposed.

Chen et al. [20] consider consistency an important issue and thus chose to checkpoint the entire system. They employ *scalable coding strategies* to simultaneously handle multiple node or link failures. Unlike traditional fault tolerance schemes¹¹, in this framework, applications are not aborted. Instead, they keep all of their surviving processes and adapt to the failures. Furthermore, they introduce several checkpoint encoding algorithms to improve scalability, such that "the overhead to survive k failures in p processes does not increase as the number of processes p increases."

To cope with concurrent failures, Hakkarinen et al. [46] propose an alternative approach, namely, an *N-level diskless checkpointing* method that minimizes the fault tolerance overhead. In comparison to a one-level scheme, layering diskless checkpointing can enable failure tolerances up to a maximum of *N* processes and considerably reduces the runtime. In addition, they develop and verify an analytical cost model for diskless checkpointing. Lastly, their checkpointing scheme can also calculate the optimal number of checkpoints and levels, to avoid an exhaustive search.

Koldehofe et al. [58] propose a novel method that can survive multiple simultaneous operator failures without using persistent checkpoints. They observe that "at certain points in time, the execution of an event-processing operator solely depends on a distinct selection of events from the incoming streams, which are reproducible by predecessor operators." This leads them to design a method that preserves the operator state in savepoints, instead of checkpoints. Consequently, the operator state solely requires the information necessary for the incoming streams and the relevant selection events. Their proposed savepoint recovery system can: (1) identify an empty operator state, (2) capture and replicate savepoints and ensure the reproducibility of corresponding events, and (3) tolerate multiple simultaneous operator failures.

To overcome weaknesses in fault tolerance methods (FTM), (e.g., active FTM, which require extra resources and passive FTM, whose recovery process is expensive), Su et al. [100] develop the passive and partially active (PPA) scheme. Their scheme employs passive checkpointing for all tasks and partially-active checkpointing for a selected number of tasks, since resources are limited. Consequently, their scheme provides very fast recovery for a selected number of tasks that use active fault tolerance and tentative output for those tasks that exploit passive fault tolerance.

Although the tentative output is less accurate than the exact output, its accuracy improves when more data are available. To generate the maximum quality of the tentative outputs, the PPA scheme employs a bottom-up dynamic programming algorithm to optimize the replication plan for correlated failures.

Upadhyaya et al. [105] propose using varying fault-tolerance techniques for distinct operators that correspond to a single query plan. Incidentally, such a strategy will require a cost-based optimization plan to achieve fault-tolerance. Thus, the authors introduce a fault-tolerance optimizer, called FTOpt, to automatically pair each operator with the most suitable technique in a query plan. FTOpt aims to reduce the execution time of the entire query despite failures. Their approach, like the PPA scheme, does not limit checkpointing to a single method. However, it is better than PPA, in terms of the quality of the result, since FTOpt produces exact output, as opposed to tentative output.

Wang et al. [108] propose the *Meteor Shower* stream processing system, which utilizes *tokens* when checkpointing. As a first step, source operators initiate the flow of tokens throughout a streaming graph. Then, when an operator obtains these tokens, the system checkpoints the operator state. *Meteor Shower* is comprised of three techniques: (1) source preservation, to avoid the cost of handling redundant tuples in previous check-pointing mechanisms [49, 50, 57], (2) parallel and asynchronous checkpointing, to enable operators to keep running during the checkpointing process, and (3) application-aware checkpointing that can both adapt to changes to an operator's state size and checkpoint whenever the state size attains a minimum value.

3.3 State for Incremental Checkpointing

The approaches discussed in subsections 3.1 and 3.2 depend on the periodic checkpointing of state (PCoS) for failure recovery. However, Carbone et al. [18] discuss two key drawbacks. First, the PCoS often interrupts the overall computation, which slows down the data flow processing speed. Second, they greedily persist all tuples jointly with the operation states, thereby, resulting in larger than expected state sizes. Thus, to overcome these drawbacks, researchers propose methods based on incremental checkpointing [50, 96, 111], which only checkpoint changes to the state (not the entire state). By capturing the *delta* of the state (i.e., the latest changes in content, since the last checkpoint), these methods considerably reduce the checkpoint overhead and yield smaller state sizes. Next, we highlight seven incremental checkpointing methods.

Hwang et al. [50] propose a fine-grained checkpointing method that employs a divide-and-conquer strategy. In their scheme, the entire graph is divided into several subgraphs, each of which is then allocated to a different backup server. By employing a so-called *delta-checkpointing* technique, each server checkpoints a small fragment of its query graph. To guarantee state consistency, changes to state are incrementally checkpointed to backup servers. When failure occurs, query fragments are collectively recovered in parallel, thereby, achieving fast failure recovery and experiencing a small run-time overhead.

Using a cost model, the incremental checkpointing (IC) method due to Naksinehaboon et al. [82] computes the *optimal number* of incremental checkpoints between two full checkpoints. Consequently, it reduces the checkpointing overhead, in comparison to full checkpoint (FC) models. Improving upon the IC

4

¹¹ That is, performing a restart from a checkpoint.

approach, Paun et al. [90] extend it to include the *Weibull failure distribution* case. Their experiments show that the overhead of the IC method is significantly smaller than that of the FC method.

The continuous eventual checkpointing (CEC) method due to Sebepou et al. [96] guarantees fault tolerance by employing incremental state checkpoints continually, while minimizing interruptions to operator processors. To achieve this, operator state is split into parts and independently checkpointed, as needed. These partial state checkpoints are expressed as control tuples that contain both window state and actual tuples. Unlike traditional schemes, in the CEC approach, checkpoints are updated incrementally and continuously. Consequently, the CEC method can efficiently handle continuous incremental state checkpoints and adjust checkpoint intervals to strike a balance between recovery time and running time.

To exploit the similarity of access patterns, among writes to memory in iterative applications, Nicolae et al. [83] propose the Adaptive Incremental Checkpointing (AI-Ckpt) approach for iterative computations under memory limitations. Under the assumption that "first-time writes to memory generate the same kind of interference as they did in past iterations," the AI-Ckpt method enables the prediction of future memory accesses for subsequent iterations. Consequently, this prediction leverages both current and historical access trends for flushing memory pages to stable storage in an optimal order. This asynchronous checkpointing approach is well suited for computing environments with limited memory resources. It can dynamically adapt to various applications, utilize access pattern history, and minimize the intervention of the checkpointing process running in the background.

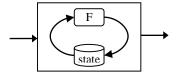
Since the I/O and network bandwidth to distant storage heavily influences the checkpointing execution time (for large-scale systems), Jangjaimon et al. [51] propose the *adaptive incremental checkpointing* (AIC) method. This approach *reduces the state size*, to use bandwidth more efficiently, *lowers the overhead*, and *improves performance*. It employs multiple cores to perform adaptive multi-level checkpointing with delta compression, which can significantly minimize the incremental checkpoint file size. The authors also introduce a new Markov model to predict the performance of a multi-level concurrent checkpointing scheme. In comparison to checkpointing schemes employing fixed checkpoint intervals, the AIC method substantially reduces the expected running time (e.g., by 47%), when evaluated against six SPEC benchmarks.

To minimize space requirements in dataflow execution engines, Carbone et al. [18] devise the *Asynchronous Barrier Snapshotting* (ABS) algorithm, which is suited for both acyclic and cyclic dataflows. On acyclic topologies, stage barriers, injected into data sources by a coordinator, can trigger the snapshot of current state. The algorithm solely materializes operator states in acyclic dataflows. On the other hand, on the cyclic execution graphs, ABS solely stores a minimal set of records on cyclic dataflows. Upon failure, the ABS algorithm reprocesses logged records to recover the system. Experiments show that ABS can achieve linear scalability and performs well with frequent state captures.

Also employing a divide-and-conquer strategy, Wu et al. [111] propose *ChronoStream*, which splits states into a collection of fine-grained *slice units*. Comparable to the recently mentioned subgraph to backup server assignment scheme [50], units can be selectively distributed and checkpointed into specific nodes. Upon failure, ChronoStream transparently rebuilds the distributed slice units, and

thus incurs small overhead. In comparison to other methods [50, 96], ChronoStream models application-level internal states differently.

4. State for Iterative Processing



State can be used to efficiently enable iterative processing in big data frameworks (BDF). This is of paramount importance for large-scale data analysis, since most machine learning and graph mining algorithms are iterative in nature. Yet, they are ill-suited for BDF, such as MapReduce [24], since they incur a large overhead. For example, at each iteration, data is needlessly reloaded and reprocessed, given that much of the data remains unchanged [124]. Additionally, each iteration is executed as a separate job [95], which is inefficient. These drawbacks lead to the development of iterative mechanisms and their integration into data-parallel processing systems [31, 32]. In his vision paper [70], Markl affirms that the native support of *state* in iterative data analysis programs is a key design for future platforms.

Iterative computations come in two varieties, namely, *bulk* and *incremental*. In bulk iterations, each step produces an entirely different intermediate computation in contrast to the (final) result. Examples of bulk iteration include machine learning algorithms, such as batch gradient descent [109] and distributed stochastic gradient descent [122]. In incremental iterations, the result of a current iteration (at time step *i*) slightly differs from the result of the previous iteration (at time step *i-1*). As discussed in [95], the elements of the intermediate computations exhibit "*sparse computational dependencies*." That is, changes in one element solely affect a few other elements. For example, in the connected components algorithm, an update to a single vertex impacts only its surrounding neighbors. Next, we discuss four approaches that use state for iterative processing.

The first approach, due to Ewen et al. [31], overcomes some performance issues in existing dataflow systems, which treat incremental iterations as bulk iterations. As a result, some iterative algorithms perform poorly. To resolve this, the authors devise a method that integrates incremental iterations into parallel dataflow systems, by exploiting *sparse computational dependencies* that are intrinsic in many iterative algorithms. Rather than creating a specialized system, their method facilitates expressing analytical pipelines in a unified manner and disregards the need for an orchestration framework. As a proof-of-concept, the authors [32] illustrate the implementation, compilation, optimization, and execution of iterative algorithms in *Stratosphere*.

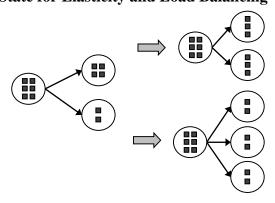
The second approach, due to Schelter et al. [95], utilizes state to address fault-recovery in the iterative processing of fixpoint algorithms, which are common in machine learning. In their paper, the authors introduce a mechanism based on the principle of algorithmic compensations to achieve optimistic recovery. Algorithmic compensations concern the exploitation of a fixpoint algorithm property, namely, the ability to converge to the solution from several intermediate consistent states. Optimistic recovery concerns resuming computation from the latest iteration, in contrast to rollback recovery, where computation starts from scratch. Using their novel ideas, the authors are able to rebuild state, using a user-

defined *compensation function*. Furthermore, their approach outperforms rollback recovery methods, since state checkpointing occurs in the background, independent of and not interfering with the processing of data. Additionally, they show how their method can be employed in three areas: *factorizing matrices*, *performing linking and centrality computations* in networks, and *identifying paths in graphs*. Lastly, Dudoladov et al. [27] demonstrate the efficiency of the optimistic recovery mechanism for both the Connected Components and PageRank algorithms in Apache Flink.

The third approach, due to Xu et al. [112], introduces the concepts of head and tail state checkpointing, to lower checkpointing costs and reduce failure recovery time. In their approach, they use an unblocking mechanism to write checkpoints, transparently in the background without requiring program to interrupt. Thereby, this manner avoids the overhead associated with delayed execution at checkpoint creation time. By injecting checkpoints directly into dataflows, this method takes advantage of both low-latency execution (by disregarding pipeline process interrupts), and the seamless integration into existing systems. Furthermore, the use of local log files on each node circumvents the need to recompute from scratch upon failure and yields a faster (or *confined*) recovery. Their experiments show that for iterative computations, head checkpointing and confined recovery outperformed blocking checkpointing and complete recovery, where the latest checkpoint state is loaded and fully recomputed.

The fourth approach, called MRQL Streaming [33], improves iterative processing performance over the two earlier approaches. It relies on two techniques, namely, lineage tracking [12] and homomorphisms, to reduce the state size. In the lineage tracking technique, attributes in join and group-by clauses are moved to query outputs, to establish connections between the input data and query results. In contrast, the homomorphism based technique combines the current state value with new input data to generate new output. To apply these two methods, MRQL Streaming automatically converts a SQL query to an incremental, distributed program that runs on a stream processing engine. Then, it derives incremental programs by storing a small state during the query evaluation process and using a novel incremental evaluation technique that merges the current state value and the latest data. This strategy can cope with many query types on data collections with nested structures, such as nested and iterative queries, groupby, aggregation, and equi-joins. More importantly, Fegaras [33] extends this method to raise the efficiency of iterative algorithms (e.g., k-means clustering, PageRank). A prototype implementation of this framework in Spark is used to validate the effectiveness of this method.

5. State for Elasticity and Load Balancing



State can be used to address elasticity and load balancing challenges. Elasticity characterizes a computing system's ability to provide additional computing resources in light of increasing workloads. Load balancing characterizes a computing system's ability to redistribute its workload across computing resources, particularly, when some nodes have heavier loads than others. Today's data-parallel computation frameworks achieve elasticity by maintaining and migrating state, while jobs are actively running. To migrate state, the number of parallel channels need to dynamically adapt (i.e., nodes are added or removed) at runtime to match the computing resource and workload availability, which may unexpectedly fluctuate. In particular, in the event of workload skew, the states of overwhelmed nodes are repartitioned and reallocated to underwhelmed nodes. Similarly, when resources are scarce, the states of the affected tasks (i.e., job partitions) need to be reallocated. To mitigate this, we need mechanisms that enable systems to scale and maintain workload balance. Next, we discuss three approaches that employ state to satisfy elasticity and load balancing needs.

Dataflow scalability in streaming systems is limited by stateful operators. In order for these operators to scale, they will need to be partitioned (e.g., across a shared-nothing platform). However, over time, this will lead to load unbalancing. To resolve this problem, Shah et al. [98] propose Flux, a dataflow operator that encapsulates adaptive state partitioning and dataflow routing. Placed between producer and consumer stages in pipelined dataflows, Flux repartitions stateful operators transparently, without interrupting the pipeline under execution. Flux provides two mechanisms to adapt to both short term and long term imbalances. In the shortterm case, Flux utilizes a buffer and a reordering mechanism to adjust local imbalances. In the long-term case, Flux detects imbalances across the entire cluster and allows state repartitioning in lookup-based operators to manage the problem. Furthermore, for other cases, the authors propose some policies to repartition state for continuous query operators.

ChronoStream [111] takes a different approach to address the elasticity and load balancing problem. By treating the internal state as a built-in component, ChronoStream achieves flexible scalability. That includes horizontal elasticity, where resources vary in all of the computing nodes and vertical elasticity, where resources vary at a single node. Consequently, this enables ChronoStream to efficiently manage both workload fluctuation and dynamic resource reclamation. For horizontal elasticity, transparent workload re-allocation is achieved using a lightweight transactional migration protocol based on the reconstruction of state at the stage-level. To support vertical elasticity, ChronoStream provides fine-grained runtime resource allocation that maps an OSlevel thread to many application-level computation slices. A thread-control table stored in the configuration state can be used to record this thread-to-slice mapping. To scale vertically, ChronoStream utilizes this table to reschedule the computation. At any time during the execution, the workload in each thread can be dynamically reorganized to rebalance the load.

Skewed workloads trigger varying imbalances, including memory usage, computation, and communication costs across parallel channels. To cope with this, Gedik et al. [43] devise new partitioning functions to distribute the load evenly among the computing nodes. In addition, they introduce several desirable properties that these functions must meet. These properties include: (1) balance properties (e.g., memory, communication and processing balance), (2) structural properties (e.g., fast lookup, compactness), and (3) adaptation properties (e.g., minimal

migration, fast computation). Experiments show that the proposed partitioning functions possess these desirable properties over a variety of workloads and thus provide better load balance than uniform and consistent hashing. These functions are especially effective for workloads with large key domains (i.e., the cardinality of the partitioning key). In this case, they can efficiently balance communication costs, computation costs, and memory load, yet still ensure low migration overhead despite workload skew.

6. State for Integrative Optimization

Thus far, *state* has been shown to be fruitful for many single-use cases (e.g., fault tolerance, elasticity, load balance). However, state can also be used to simultaneously address multiple use-cases (e.g., scalability and fault tolerance [36, 43, 80, 81, 111]). It is in this scenario that multi-objective (or integrative) optimization arises. Otherwise, optimizing independently (per each use-case) would yield a suboptimal solution. Next, we investigate six ways in which the application of state is treated as an integrative optimization problem.

In the first approach, McSherry et al. [73] propose a new computational model, called *differential dataflow*, which supports both incremental and iterative computation. Extended from batchoriented models (e.g., MapReduce, DryadLINQ), their model enables *arbitrarily nested* fixed-point iteration and simultaneously supports the efficient, incremental updates to inputs. Rather than using the entire temporal order, *changes to collections* are described in terms of the partial order. This allows the collections to evolve and eliminate the need to restart the computation to reflect changes. *Multi-dimensional lattice times*¹², which contain loops to fixed point iterations are used to combine the incremental and nested iterative computations.

Often, a single system alone cannot meet all processing requirements, such as high-throughput batch processing, lowlatency stream processing, and efficient iterative and incremental computations. Therefore, multiple systems must be employed to achieve coverage. However, the use of a federation of platforms brings numerous problems, including inefficiency, complexity, and maintenance challenges. Hence, a new system is developed. Due to McSherry et al., Naiad [74, 79] is a distributed system for dataflow programs that is developed to satisfy all three of earlier referenced requirements into a single framework. Figure 2 presents Naiad at a high-conceptual level. Naiad supports both iterative and interactive queries on data streams and generates up-to-date and consistent results that can be incrementally updated, as new data arrive continuously. Furthermore, in [74, 79], McSherry et al., present a novel computational model, called timely dataflow, to boost the parallelism prevalent across various classes of algorithms (e.g., iterative, graph-based, tree-based).

To describe the logical points during execution, Naiad employs *timestamps* to enhance dataflow computation. Timestamps are essential in supporting an efficient and lightweight coordination mechanism. This is due to three features, namely, structured loops for feedback, stateful dataflow vertices for records processing (without using global coordination), and notifying vertices when all tuples have been received by the system for a specific input or iteration round. While the first two features support low-latency iterative and incremental computation, the third feature ensures the result is consistent. Low-level primitives

12 Lattice times are times that are represented as tuples, which contain integer attributes. in Naiad enable it to serve as a base for many other high-level programming models, supporting a diversity of tasks, such as iterative machine learning, streaming data analysis, and interactive graph mining.

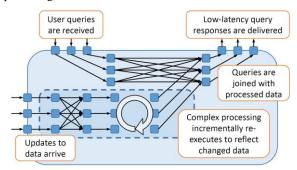


Figure 2. A *Naiad* application that answers real-time queries on continuously updated data [74]. Iterative, incremental processing is represented by the dashed rounded rectangle.

In the second approach, Fernandez et al. [36] develop a unified approach based on stateful dataflow graphs (SDG) for dynamic scalability and failure recovery, to parallelize stateful operators (when workloads fluctuate) and achieve fast recovery times (with low overhead). In their approach, they use the upstream backup to periodically checkpoint stateful operators. Their system detects bottlenecks in operators and enables them to scale by automatically allocating new machines. Consequently, the state must be repartitioned, accordingly. In the event of a failure, the checkpointed state will need to be rebuilt on a new machine and tuples will need to be reprocessed to recover the failed operators. To achieve these goals, the proposed system: (i) uses a well-defined interface to allow for the easy access to operator state, (ii) reflects information about the exact set of processed tuples by an operator in its state, and (iii) preserves operator semantics using a key attribute to partition the state.

In the third approach, Wu and Tan's *ChronoStream* [111] concurrently offers fault tolerance, scalability, and elasticity. Their low-latency stream processing system provides transparent workload reconfiguration in a unified model, by separating application-level parallel computation (i.e., computation states) from OS-level execution concurrency. As a result, ChronoStream achieves transparent elasticity, fault tolerance, and high availability without having to sacrifice performance. This is due to the reduction in the overhead triggered by state synchronization. The slice-reconstruction approach in ChronoStream is akin to the statemigration approach in SEEP [36]. Furthermore, both Wu and Tan's ChronoStream and SDG [37] support dynamic reconfiguration at runtime. However, state repartitioning incurs high state migration costs in both SEEP and SDG.

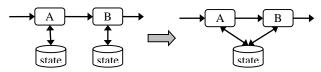
In the fourth approach, we revisit the method of Gedik et al. [43] to address both load balancing and operator migration. Recall that their solution employs a partitioning function, to achieve improved load balance (auto-fission) and low migration costs. The structure of the partitioning function is a hybrid involving an explicit map and a consistent hash. Consequently, this compact hash function can balance the workload uniformly and adapt accordingly, even under high skew. Furthermore, they construct algorithms and metrics to build and assess the partitioning

functions, to determine whether these can achieve good balance and efficient migration. More precisely, they define *load imbalance* to be the proportion of the difference between the maximum and minimum loads to the maximum permissible load difference. Data items in the partially constructed partitioning function have their migration costs normalized based on the ideal migration cost. The *utility function* combines the relative imbalance metric and the migration cost metric, to assign the items to parallel channels.

In the fifth approach, Madsen et al. [80] re-use the checkpoints meant for failure recovery, to efficiently improve the dynamic migration of the state, like Fernandez et al. [36]. As a first step, they formally define a checkpoint allocation problem with some constraints. Then they propose a practical (i.e., efficient) algorithm to reuse the checkpoints for effective load balancing. If the workload is increasingly skewed at key groups, then the system must transfer many checkpoints, for groups of keys in A to nodes with lighter loads in advance, to quickly react to fluctuations. To increase the chance of this availability, the checkpoints of the key groups in A must be allocated to the nodes with key groups that are "as negatively as possible correlated with the key groups of A." Due to the relationship between fault tolerance and migration, checkpointing can be viewed as proactive load-balancing, i.e., utilizing checkpoints for state migration to help balance the load.

Lastly, in the sixth approach, Madsen et al. [81] model load balancing, operator instance placement, and horizontal scaling, jointly, to enable low-latency processing, optimize resource usage, and minimize communication costs. They integrate horizontal scaling and load balancing using mixed-integer linear programs (MILP) to arrive at a feasible solution. This model is suitable when the placement of operator instances does not considerably affect communication costs. By using the MILP approach and linear program solvers, they improve the load balance over existing heuristic approaches. Yet, using the so-called Autonomic Load Balancing with Integrated Collocation (ALBIC) solution enables them to further achieve gains over the MILP based approach. Using ALBIC, they can: (i) generate an improved operator instance collocation, (ii) balance the load, and (iii) lower the overhead. This holds because ALBIC gradually improves the placement at runtime, while still satisfying load balance constraints.

7. Sharing State



There are instances when sharing state is desirable. Sharing state, for example, can optimize streaming processing systems, and reduce data transmission over networks. Next, we present four cases, where state is shared.

Sharing state facilitates optimizing stream processing systems. For example, Hirzel et al. [48] examine a streaming application that continuously calculates statistics (e.g., average stock price) for different time windows (e.g., hours, days). Since these operations differ only on the *time granularity* (e.g., hours vs. days), then it is natural to share the aggregation window. By doing so, this will increase resource utilization (e.g., memory) efficiency among operations. However, sharing state can lead to some inherent problems, such as access conflicts, consistency issues, or deadlocks. Therefore, Hirzel et al. point out three safety conditions requirements. First, *ensuring visibility* can make state visible and

accessible to all operators. Second, *prevention of race conditions* can assure state is immutable and/or that synchronization among processes is properly set. Lastly, *safe management of memory* can prevent the early release of memory or uncontrollable expansion, which could lead to memory leaks.

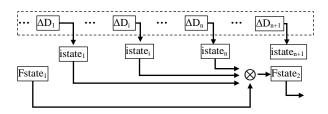
In their paper, Hirzel et al. discuss three forms of state sharing. The first form involves shared operator state [15], where state can be arbitrarily complex. In this form, synchronization and memory management present key challenges. Indeed, sharing memory may introduce conflicts, which are often resolved using mutualexclusion locks. However, when conflicts are rare, this approach is cost prohibitive (e.g., when performing concurrency handling). Therefore, an alternative approach [15] uses software transactional memory to manage data sharing. The second form entails shared windows [9, 44] that enable multiple consumers to utilize the same window. Window sharing is indeed one of the simplest cases of state sharing [44]. For example, the continuous query language implements windows by using non-shared arrays of pointers to reference shared data. This model of many-to-one pointer reference can allow many windows and event queues [9] to access a single data item. Lastly, the third form encompasses shared queue [97]. Here, the simultaneous access of both producer and consumer to a single element (i.e., the producer writes a new item and the consumer concurrently reads an old item) can lead to conflicts. To guarantee synchronization and preserve concurrency, queues must be able to buffer two data items at a minimum. Turning shared queues into a local one and statically computing queue-size at compile time (instead of runtime) can improve the performance of a shared queue.

In their paper [59], Kuntschke et al. recognize instances of computational inefficiencies in large-scale data processing that can be eliminated by sharing state. Examples of these include the unnecessary execution of operators and data transfers, among other redundancies. By sharing data streams, we avoid redundant transmissions and save network bandwidth. Another benefit of discarding unnecessary computation is reducing the execution time, by sharing previously computed results and early filtering and aggregation (e.g., the combine function in MapReduce). They propose two optimization techniques. *In-network query processing*, which distributes and performs (newly registered) continuous queries and *multi-subscription optimization*, which enables the reuse and sharing of generated data streams.

In their paper [68], Losa et al. propose *CAPSULE*, a language and system that supports *state sharing across operators*, using a less structured method than point-to-point dataflows. It shares variables (a.k.a. states) using a data structure at the language level. Besides supporting the efficient sharing of state in distributed stream processing systems, CAPSULE provides three features. That is, (i) *custom code generation*, to produce shared variable servers that fit a given scenario based on runtime information and configuration parameters, (ii) *composability*, to achieve suitable levels of scalability, fault-tolerance, and efficiency using shared variable servers, and (iii) *extensibility*, to support, for example, additional protocols, transport mechanisms, and caching methods, using simple interfaces.

In their paper [75], Meehan et al. introduce S-Store, a system designed to maintain correctness and ACID13 guarantees that are essential to handle shared mutable state. By employing shared state, the system achieves high throughput and consistency for both transaction processing and stream processing applications. In this context, the proper coordination and sharing among successive executions of a window state differ from other sorts of state (e.g., where state is privately shared with other transactions). In this way, S-Store achieves low latency with correctness in stream processing and high performance with ACID guarantees in transaction processing. Tatbul et al. [101] further explore correctness criteria, including ACID guarantees, ordered execution guarantees, and exactly-once processing guarantees. To support these threecomplementary correctness guarantees, S-Store provides efficient scheduling and recovery mechanisms. Although Naiad, SEEP, and Samza all view state as mutable, they do not inherently support transactional access to shared state. Thus, Meehan et al. show that S-Store's consistency guarantees are better than those of these

8. Incremental Maintenance of State



Updates to state are more cost-effective when performed incrementally. As previously discussed in Section 3.3, researchers [82, 96] focus on reducing the overhead associated with performing incremental checkpointing. Similarly, many other researchers [33, 54, 55, 56, 64, 74, 84, 85] have sought to maintain state incrementally, to cope with frequent data updates and avoid expensive full state updates. By generating delta values, they all can update persisted state more efficiently, whenever inputs vary marginally, instead of recomputing from scratch. Next, we elaborate on eight different approaches that enable the incremental maintenance of state.

The first approach due to McSherry et al. [74] presents differential computation, which generalizes existing methods for incremental computation with continuously changing input data. Their method differs from traditional incremental computations by supporting arbitrarily nested iterative computations. Akin to the Naiad system, the key innovations come from two factors. First, changes in state adhere to a partially ordered sequence, instead of a totally ordered one, which conforms to incremental computation. Second, an indexed data-structure maintains a set of updates that is essential to rebuild the state. This second feature is different from the other incremental systems, in that updates are usually discarded after being merged with the current state snapshot.

The second approach due to Koch [54], employs monoid algebra to address the incremental view maintenance (IVM) problem and extends an algebraic structure of a ring of databases to form a powerful aggregate query calculus. This calculus inherits the key properties of rings, such as distributivity and the existence of an additive inverse. Thereby, this makes calculus closed under a universal difference operator that expresses the k^{th} delta queries of

the IVM. These key properties provide the basis for delta processing and incremental evaluation. The multi-layered IVM scheme can maintain a view (using a hierarchy of auxiliary materialized views) and refresh it, whenever there are updates. Furthermore, their findings lay a foundation for subsequent research [5, 55, 84, 85] in incremental state maintenance.

The third approach due to Ahmad et al. [5], introduces a recursive, finite differencing technique, called viewlet transforms, that unifies historical and current data. Their technique materializes a query and its corresponding views, that support the mutual incremental maintenance, thereby, reducing the overall view maintenance cost. Similarly, Koch et al. [55] fully describe and experimentally evaluate the performance of the DBToaster system, using the ring theory. DBToaster can continuously update materialized views, despite frequent data changes, using an aggressive compilation technique or a recursive finite differencing technique.

The fourth approach due to Nikolic et al. [84], introduces the LINVIEW framework and the concept of deltas, which captures changes to linear algebra programs (LAP) and highlights the use of IVM in LAP involving iterations in machine learning. Linear algebra operations can trigger a ripple effect (e.g., small input changes can propagate and affect intermediate results and the final view). This can negatively affect the performance of IVM upon reevaluation. To mitigate this problem, LINVIEW employs matrix factorization methods to enable IVM to be suitable and less expensive than recomputing from scratch.

The fifth approach due to Nikolic et al. [85], generalizes the results of Koch et al. and presents recursive and incremental techniques to handle queries containing nested aggregates. They compare the performance between tuple and batch incremental updates to identify scenarios when batch processing can substantially improve the efficiency of IVM. Their experimental findings show that single-tuple execution outperforms generic batch processing in many situations, thus contradicting the belief that batch processing outperforms single-tuple processing [87]. Finally, they describe a novel approach to compile IVM code into optimal programs that can run efficiently in distributed environments.

The sixth approach due to Koch et al. [56] provides an efficient solution to incrementally compute the positive nested relational calculus (NRC+) on bags. They develop a cost model for NRC+ operators that enables them to calculate the cost of delta computations. A query can be considered efficiently incrementalizable if the cost of its delta is strictly lower than that of recomputation from scratch. A large part of NRC+, called IncNRC+, which satisfies the efficient incrementalization condition is translated from NRC+ without losing its semantics. Further, they show how to apply a recursive IVM method to IncNRC+ and demonstrate how the method can perform better than conventional IVM methods (for flat queries).

The seventh approach due to Fegaras [33], introduces a prototype, called MRQL Streaming [33] that returns (at each time interval) continuous answers, by merging the last materialized state and the delta result of the most recent data batches. The novelty of this approach comes from algebraic transformation rules that convert queries to homomorphisms. MRQL Streaming decomposes a non-homomorphic streaming query q(S) into two functions, a and

¹³ That is, atomicity, consistency, isolation, and durability.

h, such that q(S) = a(h(S)), where h is a homomorphism (i.e., $h(S + \Delta S) = h(S) \otimes h(\Delta S)$) and a is a non-homomorphic component of the query that forms the answer function. Accordingly, state stores the result of the incremental calculation h, using the current state value to compute the next h value (i.e., $state = state \otimes h(\Delta S)$). Initially, state is either empty or set to h(S), if there are initial streams. Then, at every interval, Δt , the answer to the query is computed from the state that is equal to $h(S + \Delta S)$.

Lastly, the eighth approach due to Liu et al. [64], introduces a re-optimization technique that proposes a cost-based optimizer to re-plan the optimal execution in the event of unexpected performance changes. Given continuously changing inputs, a streaming system repeatedly updates outputs by recalculating the optimal plan incrementally. Their incremental re-optimizer uses optimizer memoization table to maintain state throughout runs when re-optimizing. Furthermore, determining which plans to prune from this state is important to re-plan execution. Liu et al. define a semantic for tracking and re-computing state, using a declarative specification, to enable re-pruning. Their declarative method enables them to identify state-pruning strategies that are "agnostic to the order of control and data flow during plan enumeration." Lastly, they propose three new state-pruning strategies, i.e., aggregate selection with tuple source suppression, reference counting, and recursive bounding. Their experiments show that each state-pruning technique offers an alternative and meaningful way to make incremental re-optimization more efficient.

9. Operations on State

There are many operations on state, including *store*, *update*, *purge*, *migrate*, and *expose*. In the following sub-sections, we discuss each of these operations and their impact on *state* in greater detail.

9.1 Storing State

Storage solutions for *state* vary widely and generally *state size* determines where state will be stored. For small sizes, researchers [92, 118] propose storing state *in-memory*, which can accelerate processing [118]. In contrast, for large sizes, researchers [57, 63, 83] recommend state be kept in *persistent storage*. However, this incurs greater overhead. Nonetheless, deciding where to optimally store state is not always trivial. Next, we discuss three approaches to handle large state sizes, i.e., *load shedding*, *state spilling*, and *state cleanup delay*.

Processing long-running queries (LRO) over data streams (i.e., complex queries with huge operator states, such as multijoins) can be memory-intensive. When system resources are scarce and processing demands cannot be met (e.g., due to huge workloads at runtime), varying handling methods can be employed. One example is load shedding [103], which can reduce workloads and increase performance, but at the expense of lowering accuracy. Workloads can be shed permanently or alternatively processed later, when computing resources are again available [63]. For those cases where accuracy is paramount, load shedding is not a viable solution. Thus, an alternative approach, called state spilling, proposes using join-variants (e.g., XJoin [106], Hash-Merge Join [78], MJoin [107]) to temporarily flush states stored in-memory to disks when memory is at capacity. Yet another option is delaying state cleanup (i.e., processing states stored on disks) until resources are readily available. These three state handling solutions achieve

¹⁴ Multiple state operators arise in data integration and data warehouse scenarios, where memory intensive queries abound. both low-latency processing and the completeness of results. Next, we present five approaches for storing state.

The first approach due to Liu et al. [63], addresses the LRQ problem. Unlike existing solutions, which can only handle a single state-intensive operator, their proposed strategies can handle multiple state operators 14. Their state spilling strategies selectively flush operator states to disks, to cope with complex queries. By appropriately spilling parts of operator states to disk at runtime, they avoid memory overflows and increase query throughput. In addition. they observe that by exploiting operator interdependencies they can achieve higher performance, in contrast to existing strategies, which do not. Further, they highlight two classes of data spilling strategies, namely, operator-level and partition-level. The operator-level strategy employs a bottom-up approach and regards all data in an operator state to be similarly important. In contrast, each of the partition-level data spill strategies (i.e., local output, global output, and global output with penalty) takes input data characteristics into account. In all of these strategies, when memory is scant, the appropriate partition to be spilled will need to be selected, to maximize query throughput.

Their *D-CAPE* system implements all the proposed strategies and enables researchers to compare their performance. Experiments show that the global output strategy (with or without penalty) outperforms the localized strategies. To ensure completeness, Liu et al. propose effective cleanup algorithms to produce the correct results from the data stored on disks. For continuous queries that process data streams with high input rates, performing the state cleanup process is only possible after the runtime execution phase is complete. For queries with window constraints, interleaving the in-memory computation and the disk cleanup process is necessary at runtime. However, some challenging issues remain unresolved, such as the *spilling time*, *cleanup time*, and the *selection of data to be cleaned-up*. Thus, these open problems have yet to be solved.

The second approach due to Kwon et al., called SGuard [57], stores state in a distributed and replicated file system (DFS), like GFS¹⁵ or HDFS¹⁶, to save memory for *normal* stream processing. One of the benefits of these file systems is that they are optimized for reading and writing large data volumes. Since multiple nodes may be checkpointed simultaneously, resolving resource conflicts is a critical requirement, which is met in SGuard by incorporating a scheduler into the DFS. The coordination of many write requests. enables the scheduler to reduce both individual checkpoint times and generally provides good resource utilization. Akin to rollback recovery methods [49], SGuard periodically checkpoints state and recovers failed nodes from their last checkpoints. Unlike previous approaches, however, SGuard takes checkpoints asynchronously: While the system is under execution, SGuard uses a new Memory Management Middleware to checkpoint the operator state. Consequently, this asynchronous mechanism can prevent potential interrupts and reduce the overhead incurred by the checkpointing process.

The third approach due to Nicolae et al. [83], proposes an asynchronous checkpointing runtime approach, called *AI-Ckpt*, designed for adaptive incremental checkpointing. AI-Ckpt exploits trends in *current* and *past* access patterns and generates an optimal ordering scheme to flush memory pages to stable storage. In their research paper, the authors observe that there are memory writing

¹⁵ Google File System

¹⁶ Hadoop Distributed File System

patterns in iterative applications. Consequently, AI-Ckpt leverages these patterns and optimizes the system to flush modified pages with minimum overhead. They describe five design principles: (1) managing protected memory areas properly, (2) tracking modified pages to capture access patterns and checkpoint increments asynchronously, (3) using bounded copy-on-write to prevent unnecessary waiting, (4) conforming page flushing to access patterns, and (5) optimizing the page flushing process based on the access pattern history. In addition, they implement several algorithms to illustrate these design principles. Their experiments show that flushing optimally can considerably improve performance, especially for iterative applications (e.g., machine learning, graph algorithms) that exhibit repetitive access patterns. However, this method only uses the access order to flush pages, and omits the temporal aspect. Thus, a promising research direction would integrate timestamps into the access order, to further improve the flushing process.

The fourth approach due to Ren et al. [92], presents CALC (Checkpointing Asynchronously using Logical Consistency), an asynchronous technique that captures database snapshots without physical consistent checkpoints and overheads common in other snapshotting schemes. The key idea is to create a virtual point of consistency among transaction commits. Checkpoints solely include state changes caused by transactions that happen prior to the creation of virtual points. That is, the checkpoints do not register subsequent transaction modifications. Moreover, they recommend pCALC (a partial version of CALC) be used to further improve checkpointing performance. In effect, pCALC takes partial state checkpoints that have changed from the most recent checkpoint, and merges these (in the background) to reduce failure recovery costs. Testing across a wide range of transactional workloads, their experiments prove that both CALC and pCALC reduce recovery costs and lower memory usage over competing checkpointing systems.

Lastly, the fifth approach due to Ananthanarayanan et al. [8], called *Photon*, is a distributed system that can store large states across geographically distant locations. It can join multiple unordered data streams to ensure high scalability, low latency, and exactly-once semantics. Without human involvement, Photon can automatically solve infrastructure breakdowns and server outages. The critical state stored in the *IdRegistry* and shared between workers consists of a set of event identifiers, joined over the last *N* days, where *N* is chosen such that it balances storage costs and drop events. To ensure services are always available, the IdRegistry is duplicated synchronously across multiple datacenters, which may be in different geographical regions.

9.2 Updating State

In this subsection, we turn our attention to four approaches to update state. Those are *incremental state update*, *fine-grained update*, *consistent update*, and *semantic update*.

In the first approach, Logothetis et al. [67] handle continuous bulk processing (CBP), by strictly updating a fragment of the *state* to optimize system performance. Similarly, Fegaras [33] updates *state* incrementally, via a new stateful operator, called *Incr*. Every time the MRQL Streaming system produces a small delta result, based on a data subset (ΔS_i) and involving a homomorphism, it merges the previous state value and the current delta result, then the system can incrementally produce a new state value, i.e., $state \leftarrow state \otimes h(\Delta S_i)$. Figure 3 below illustrates this update with two streaming sources.

In the second approach, Fernandez et al. [37] consider finegrained updates to examine how updates can affect throughput and latency. They compare the update granularity among several systems to determine which one can support fine-grained updates. To do this, they vary the window size, since it depends on the granularity of updates to the state. That is, the smaller window size leads to less batching and thus finer granularity. Their experiments show that Naiad [79] can achieve low latency when using small batch sizes (e.g., 1000 messages) and high throughput for large batch sizes (e.g., 20000 messages). This result is due to Naiad's capability to configure the batch size, which is independent of the window size. The stateful dataflow graph (SDG) [37] handles all window sizes and achieves higher throughput than Naiad. The overhead of micro-batching is substantial in other deployments: Spark Streaming throughput is equivalent to that of SDG, but its smallest window size is 250 ms. If this limit is surpassed, its throughput will collapse.

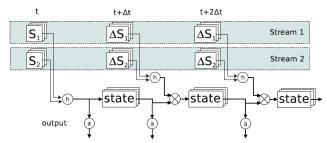


Figure 3. Incremental updates to state [33].

In the third approach, Low et al. [69] introduce the GraphLab framework for graph-parallel computation, to ensure data consistency when updating program state. GraphLab represents modifiable program state as a directed graph, called data graph. This state includes user-defined mutable data and sparse computational dependencies. To alter the state, an update function transforms the graph into scopes, which are small overlapping contexts. To preserve data consistency, GraphLab presents three consistency models: full, edge, and vertex for update functions (UF). These models enable the optimization of parallel execution and select the consistency level needed for correctness. The full consistency model achieves serializability by ensuring that the scopes of UF do not overlap and that the UF are executed concurrently. However, this consistency model limits potential parallelism and thus they propose two other consistency models to overcome this shortcoming. In the edge consistency model, each update function can read or write to its adjacent edges and vertex, but can only read adjacent vertices. Finally, all update functions can run in parallel in the vertex consistency model. As a result, these two models improve parallelism.

Lastly, in the fourth approach, several big data processing frameworks [6, 102, 116] both explore and compare *semantic updates* to *state*. Basically, there are three kinds of semantic guarantees, namely, *at-least-once*, *at-most-once*, and exactly-once, to assess the correctness of state. Systems with at-least-once semantics fully process every tuple, but they cannot guarantee duplications. In at-most-once semantics, systems either do not process tuples at all or execute exactly once. Unlike at-least-once semantics, at-most-once semantics do not require the detection of duplicate tuples. Finally, systems with exactly-once semantics process tuples once and only once, thereby providing the strongest guarantee. Section 11 compares these semantic guarantees among popular big data frameworks.

9.3 Purging State

When systems no longer need a specific piece of data for subsequent operations, state can *purge* that data (e.g., a buffer state removing expired tuples). This subsection presents three efficient ways to purge state.

In the first approach, Ding et al. [25] propose several join algorithms that effectively purge state using punctuation on data attributes. They introduce a stream join operator, called *PJoin*, that deletes data, which is no longer useful. The use of punctuations marks the end of transmission values, thereby allowing stateful operators to remove state during runtime. Consequently, this frees memory for other operations and accelerates the probing process in join operations. Then, they equip PJoin with two strategies, i.e., eager and lazy to purge states. Eager purge immediately purges states whenever punctuations are observed, to minimize memory overhead and efficiently probe the state of the join operation. If punctuations arrive too frequently, then eager purge is inapplicable, since the probing cost is less than the cost of scanning the join state. Therefore, they propose lazy (batch) purge, which can only initiate purging when the number of newly generated punctuations from the last purge approaches a given threshold. The number of punctuations between two state purges determines this threshold value. Eager purge is the special case of lazy purge when the threshold is set to one. Experiments confirm that the eager purge strategy is suitable to minimize the join state, whereas the lazy purge strategy is applicable for systems with abundant memory resource.

In the second approach, Tucker et al. [104] propose punctuation semantics as a solution to the following problem: a join operator will need to maintain states that can grow infinitely and eventually exceed memory capacity, when continually joining multiple streams. By injecting punctuations, systems can explicitly indicate the end of a data subset, thereby enabling the safe purging of log data that will not affect future results. In this paper, the authors consider a continuous join query (CJQ) to be unsafe (and thus not permitted to run), if it requires an infinite storage. Li et al. [60] introduce the punctuation graph structure to analyze query safety. That is, checking whether a CJQ satisfies safety conditions under a given number of punctuation schemes, in polynomial time. To do so, they must first formally define the purgeability condition of a join operator. Then, they classify the safety verification of a CJO into two categories: data and punctuation purgeability. The authors consider punctuation to be a special tuple that enables punctuation purging. Finally, they also propose a chained purge method to generalize a binary join to the n-way joins.

In the third approach, Li et al. [61] design a new architecture for out-of-order processing (OOP) that avoids order preservation. This is important since stream processing systems often impose an ordering of items on data streams during execution, which incurs a significant overhead when purging operator state. OOP uses punctuation or heartbeats to explicitly denote stream progress for purging operators. In addition, they introduce *joint punctuation*, a new punctuation used to reduce delay in join operators. In the past, researchers [25, 60, 104] use punctuation as a general mechanism to purge state from stateful operators.

9.4 Migrating State

Dynamic *state* migration is a crucial operation in stream processing systems that involves the efficient transition of state from one place to another, while preserving the semantics. For stateless operators, existing migration approaches usually implement the *pause-drainresume* strategy that ceases to accept new data and removes old

data. This strategy may create a deadlock during the migration process [121]: "[..., which waits] for all old tuples in operator states to be purged from the old plan, while the old tuples in those states are waiting for new tuples to be processed in order to be purged." Beyond the deadlock, this strategy does not address the problem of migration for stateful operators. Stateful operator state migration is important for operations, such as joins, aggregations, or the addition and removal of nodes. As a result, workloads, data characteristics, and resource availabilities may fluctuate. Ding et al. [26] note that state migration involves two main problems: (1) How to migrate? That is, selecting a mechanism that reduces the overhead triggered by synchronization and delaying the production of results during migration, and (2) What to migrate? That is, determining the optimal task assignment that minimizes migration costs. Next, we present five approaches for migrating state.

In the first approach, Zhu et al. [121] introduce dynamic migration for continuous query plans that contain stateful operators. They propose two strategies, i.e., moving state and parallel track that exploit reusability and parallelism. Exploiting these is useful for seamlessly migrating continuous join query plans, while ensuring the correctness of query results. In the moving state strategy there are three key steps: (i) state moving, (ii) state matching, and (iii) state recomputing. Initially, the moving state step terminates the current query plan execution and purges records from intermediate queues. Then, the next step is matching and moving all records belonging to the states of the current query plan to the new query plan. This is necessary to resume the processing of the new query plan. In the parallel track strategy, state migrates gradually, by plugging in the new query plan and executing both query plans at the same time. Thereby, this strategy continues to produce output records throughout the migration process. When there are enough computing resources, the *moving state* strategy usually completes the migration process sooner and performs better than the parallel track strategy. In contrast, when resources are scarce, the parallel track strategy has fewer intermediate results and a higher output rate during the migration process. Hence, the choice of migration strategy depends on the availability of system resources.

In the second approach, Ding et al. [26] migrate states among nodes within a single operator. Although both SEEP [36] and StreamCloud [45] propose the idea of operator state migration, prior to Ding et al., however, they provide few details. In contrast, Ding et al. develop algorithms that perform both live and progressive state migration. Consequently, the result delay prevalent in the migration process is negligible. Furthermore, they propose a (migration) task assignment algorithm that computes an optimal assignment, minimizes migration costs, and balances workloads. Moreover, they propose a new algorithm that draws on statistics from past workloads to predict future migration costs. Ding et al. criticize ChronoStream [111], which "claims to have achieved migration with zero service disruption," by pointing out that synchronization issues can affect the correctness of the result. To overcome this, the proposed mechanism does not migrate and execute tasks concurrently. It also ensures that all misrouted tuples are sent to their correct destinations. To control result delays originating from migrations, they perform migration progressively. Particularly, they perform multiple mini-migrations, where the number of migrated tasks must be below a given threshold.

In the third approach, Pietzuch et al. [91] propose a solution for migration that determines the placement locations, i.e., the selection of a physical node to manage an operator. This is indeed challenging due to variations in network and node conditions over time and the interactions among streams. In their approach, an optimizer examines the current placement of local operators and launches the migration of operators, when the savings in network usage exceeds a predefined value. This *minimum migration threshold* (MMT) depends on the cost of operator migrations and maintains an operator at its current location, if the MMT is not exceeded.

In addition, they introduce SBON (a stream-based overlay network) that efficiently determines the placement location and reduces network utilization. SBON is a layer between a streaming system and a physical network to manage operator placement and migration. The varying conditions cause SBON to re-evaluate existing placements and trigger operator migrations in new hosts, if necessary. It uses a network usage metric for operator placement, to balance both application delay and consumed network bandwidth. SBON has two main components: (1) the data stream processing system, which is responsible for operations related to operator state (e.g., instantiation, migration) and data transfer, and (2) the SBON layer, which records local performance, handles the cost space, and triggers migrations. Using PlanetLab, the authors evaluate SBON and show that their placement optimization technique increases network utilization, incurs low latency, and supports dynamic optimization efficiently (e.g., reducing network usage by 17% and re-use by 21%).

In the fourth approach, Ottenwalder et al. [86] propose *MigCEP*, which plans migration in advance, to minimize network usage. They introduce an algorithm that generates a *Migration Plan*, i.e., a probabilistic data structure that describes future targets and times for migration. In addition, they propose another migration algorithm that minimizes both network usage and latency. It enables multiple operators to coordinate their migration (e.g., for those that may require the same mutable state) and this can further improve network utilization.

Lastly, in the fifth approach, Feng et al. [35] present two novel methods, randomized replication representation and an overloaded replication scheme to address high computational workloads (e.g., due to monitoring, migrating, replicating, and backing up states) in stateful stream processing systems. In the first method, a hashing structure, called MLCBF (Multilevel Counting Bloom Filter), replicates operators using minimal resources, to increase the performance of state migration. In addition, they use dynamic lazy insertion, an adaptive scheme to reduce the influence of replication, prevent the system from being overloaded, and increase cluster throughput. Experiments show that MLCBF minimizes network and memory usage by over 90% for URL categorization. Moreover, MLCBF is quite simple and pragmatic in terms of implementation and maintenance.

9.5 Exposing State

Exposing *state* in processing systems offers several advantages by enabling: (1) systems to quickly recover from failures via checkpoints, (2) systems to efficiently reallocate stateful operators across several newly partitioned operators to provide scale out [36], and (3) some integrative optimizations as discussed in Section 6. Consequently, researchers [36, 37, 38, 67, 111] have also opted to externalize state. Next, we discuss four approaches to expose state.

In the first approach, Logothetis et al. [67] propose a groupwise processing operator that considers *state* to be an input parameter. To handle state explicitly, they develop a set of flexible *primitives* for dataflow to perform large-scale data analysis and graph mining. For example, the translate operator can access state directly via a powerful *groupwise processing abstraction*, which

permits users to store and access state during execution. In addition, this general abstraction supports other operations, such as *insertions*, *updates*, and *removals* of state. Lastly, the authors plan to develop a compiler that translates an upper-layer language into processing dataflows, to facilitate *state* access.

In the second approach, Fernandez et al. [36] seek to externalize internal operator state, so that stream processing systems can explicitly perform operator state management. The authors classify state into three types, namely, processing state, buffer state, and routing state. Processing state is responsible for maintaining an internal summary of the history of input tuples. The internal representation of processing state is contained in efficient data structures. Systems can translate processing state externally to key/value pairs when necessary. They model buffer state as an output buffer of an operator that stores a limited number of past output tuples. Operators have output buffers between them to store unprocessed tuples. Upstream operators must cache these tuples, so that downstream operators can reprocess them after failure. With this caching mechanism, buffer state can absorb short-term variations of input rates and network bandwidth. Directing tuples in the output buffer to the exact partitioned downstream operator is necessary after dynamic scaling out. To do so, they use routing state to route a tuple to the suitable partitioned operator by mapping keys to a partitioned downstream operator. To manipulate these three types of states, they define a set of operators for state management that enables systems to checkpoint, backup, partition, and restore operator state. These primitives are the minimum set required for scale out and fault tolerance. It is possible to build more state primitives to augment the functionality. For example, the availability of abundant resources enables operator states to merge [45] for scale-in. To deal with large state sizes, spilling state [63] to disk can free memory for useful computations. Persisting parts of an operator state into external storage enables the combination of data-at-rest and data-in-motion [5].



Figure 4. Distributed state types in stateful dataflow graphs [37].

In the third approach, Fernandez et al. [37] make state explicit for imperative big data processing via the use of SDG (stateful dataflow graphs). Consequently, this presents a problem for big data frameworks with imperative machine learning algorithms, given that fine-grained access to large state is required. SDG address these challenges by efficiently translating imperative programs with large distributed state into a dataflow representation, thereby enabling low-latency iterative computation. By explicitly differentiating data from state, SDG use state elements, to encapsulate computation state and enable translation. They accomplish this by using several efficient data structures (e.g., indexed sparse matrices, hash tables) to implement SEs. Figure 4 illustrates two distributed ways to represent an SE. In the first way, a partitioned SE divides its data structure into disjoint parts. In the second way, a partial SE replicates its internal data structure into multiple versions to allow for independent updates. Partitioning state across nodes can support scalability, if it is possible to fully deploy the computation in parallel. On the contrary, if it is not the case, a partial SE deploys independent computations. Application semantics can then interpret these computations. The important point concerning SDG is that their tasks can directly access the

distributed mutable state, allowing SDG to comprehend the semantics of stateful programs. At the source code level, they use *annotations* to indicate how systems distribute and access state. Annotations determine the access type to partial SE, according to the semantics of machine learning algorithms. Fernandez et al. [38] demonstrate this by developing the *JAVA2SDG* compiler to translate annotated Java programs to SDG. For example, they demonstrate the translation of machine learning algorithms (e.g., collaborative filtering, logistic regression) implemented in Java into SDG and how to run them on a cluster of machines.

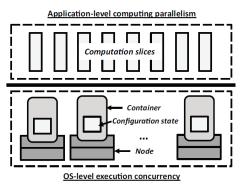


Figure 5. An internal state management abstraction design [111].

In the fourth approach, ChronoStream [111] views operator state from two perspectives, i.e., computation state or configuration state. Computation state is a set of data structures (at an application level) that systems can directly access, and conform to the userdefined processing logic. Without loss of generality, ChronoStream represents each user-defined data structure in the form of a key/value store. Some operations, such as get, set, or delete can correspondingly reflect any change to this data structure. Systems hash-partition the computation state, which is kept in an operator, aggressively into an array of fine-grained computation slices. To enable load balancing, slices are distributed equally among resource containers. Every subset of input data corresponds to an independent slice that generates a corresponding output stream. Configuration state is a collection of states (at the container level), which is used to maintain runtime parameters. This state is associated with each resource container and its contents differ among containers. The configuration state associated with each container comprises three components: (1) an input routing table, to deliver input events to corresponding slices, (2) an *output routing* table, to direct output events to a resource container associated with a downstream operator, and (3) a thread-control table, to preserve the thread schedule (at the operating system level) and compute the upper-layer slices. Generally, configuration state plays a role as the intermediate connection between parallelism at the application level and local multithreads at the operating system level.

Figure 5 illustrates the relation and order among computation states, configuration states, resource containers, and the computing nodes in an operator. Using the concept of slices, ChronoStream supports horizontal and vertical elasticity by scaling the underlying computing nodes logically and managing the configuration states associated with these nodes rather than handling the computation states at the application level.

10. State Management Overhead & Complexity

The overhead and complexity associated with state management approaches vary widely. Next, we discuss the *impact of frequent*

checkpointing on performance, the complexity of optimal state placement, and the complexity of optimal state assignment.

10.1 Impact of Frequent Checkpointing

In practice, heuristics are often used to decide when to checkpoint (e.g., periodic or aperiodic checkpointing). Periodic checkpointing enables systems to quickly recover from failure. However, systems will expend resources and time that could be better used elsewhere. In contrast, aperiodic checkpointing leads to longer failure recovery times. Thus, in recent years, systems researchers [36, 82, 94] have focused on determining an optimal checkpointing frequency.

Naksinehaboon et al. [82] investigate the optimal placement of checkpoints to minimize the total overhead, i.e., both the *rollback recovery* and *checkpointing* overhead. By employing a checkpointing frequency function, they can derive an optimal checkpointing interval based on a user-provided failure probability distribution.

Fernandez et al. [36] measure processing latency and demonstrate that aperiodic checkpointing would generate varying latencies. Their method reveals that wider intervals have less impact on data processing, but lengthen the failure recovery time. Instead, they propose setting the checkpointing interval, according to the estimated failure frequency and the query performance requirements.

Sayed et al. [94] evaluate the impact of checkpointing intervals across methods. They critique ad-hoc periodic checkpointing rules, such as *checkpointing every 30 minutes*. They observe that the model due to Young [114] achieves near optimal performance and is applicable in practice. They further investigate more advanced methods that dynamically change the checkpointing interval. Their findings show that these methods significantly improve over Young's model for only a small subset of systems.

10.2 Complexity of Optimal State Placement

Determining when to effectively place checkpoints is yet another challenging problem. Researchers [13, 93] formally prove that this problem is NP-complete and propose approximation algorithms to solve this problem in polynomial time.

Robert et al. [93] focus on the *complexity of computational* workflow scheduling with failures that follow an exponential distribution. They aim to optimize the expected processing time, processing schedule of independent tasks, and checkpointing time, which are combinatorial problems. They prove that this optimization problem is *strongly NP-complete* and propose a dynamic programming algorithm that runs in polynomial time.

Bouguerra et al. [13] examine the *computational complexity* of checkpoint scheduling with failures that follow arbitrary probability distributions. They note that both costs among checkpoints as well as the processing time for data blocks vary. Therefore, they develop a new complexity analysis to exploit relationships among failure probabilities, checkpoint overhead, and a computational model. Additionally, they introduce a new mathematical formulation to optimize checkpoint scheduling in parallel applications. They prove that checkpoint scheduling is NP-complete and propose a dynamic programming algorithm to determine the optimal times for checkpointing.

10.3 Complexity of Optimal State Assignment

Determining an effective strategy to partition tasks efficiently is a challenging problem, given an exponential search space. Ding et al. [26] calculate the optimal task assignment to minimize state migration costs (i.e., the total storage size of all the operator states transferred among nodes) and meet load balancing conditions. Let the output of partitioning function f to input record r be an integer f(r) with $1 \le f(r) \le m$. Each node N_i ($1 \le i \le n$) is assigned an interval $I = [I_i.lb, I_i.ub)$, $1 \le lb_i \le ub_i \le m$, called the *task interval* of N_i . Given a threshold τ , a task assignment is considered to be load balancing if and only if the workload W_i for each node N_i satisfies this condition $W_i \le (1+\tau)W/n$. In other words, this condition means that each node does not have too high workload when comparing to the average value of the perfect case where every node shares exactly the same amount of work W/n. The optimal task assignment includes two consecutive steps: dividing all tasks into n' separate task intervals, and then allocating these task intervals to n' different nodes.

To address the task partitioning problem, the researchers split it into numerous sub-problems, then solve each sub-problem using $Simple_SSM$, a proposed basic solution with $O(m^2n^2n'^2)$ possible sub-problems. $Simple_SSM$ incurs a space complexity of $O(m^2n^2n'^2)$ and time complexity of $O(m^3n^3n'^2)$. To improve upon this, they propose another solution that exploits optimizations and gradually improves the space and time complexity over time. The best solution uses only O(mn') space and $O(m^2n')$ time, which is a significant reduction over the basic solution.

11. State Implementations and Limitations

In this section, we survey the implementations of state in five popular open-source big data processing frameworks, i.e., *Storm*, *Storm Trident*, *Samza*, *Spark*, and *Flink*.

Storm solely supports stateless processing and implements state management at the application level, to support fault-tolerance and scalability in stateful applications. It is not equipped with any native mechanism to manage state. To overcome this limitation, an abstraction layer called *Trident* that extends Storm is proposed. It is a micro-batch system that adds state management and guarantees exactly-once semantics using its own API designed for fault tolerance. It not only inherits Storm's acknowledgement mechanism, it can prevent data loss and ensure that each tuple is processed only once. Currently, there are two state management alternatives supported in Storm. The first alternative keeps information about the order of the most recent batch and the current state, however, it may block execution. The second alternative overcomes the previously stated shortcoming, however, it incurs more overhead, by also maintaining the last state. To ensure *correct* semantics, it is vital to maintain the order of state updates. Trident is not well-suited for big states, for otherwise, it would incur severe delays. It is better suited for small states.

Samza can manage large states (e.g., GBs in each partition) by preserving state in local storage and using Kafka to duplicate state changes. Kafka stores the log of state updates and can easily restore state. By default, Samza uses a key-value store to support stateful operators. However, alternative storage systems are also available, if richer querying capabilities are required.

Spark implements state management using the concept of a DStream (i.e., a discretized stream), which updates operations via transformations. Distributed immutable collections or RDDs (resilient distributed datasets) are key concepts of Spark. Fault tolerance in Spark is achieved using lineage [116], to avoid checkpointing overhead. *State* in Spark streaming plays the role of another micro-batching stream. For this reason, during micro-batch processing, Spark uses an old *state* to generate another micro-batch result and a new *state*.

| Systems | State Management | Fault Tolerance | Guarantees | | |
|-----------|---------------------|--------------------|------------|--|--|
| Storm | not | tuples | at least | | |
| | native | acknowledge | once | | |
| Storm | specific | tuples | exactly | | |
| Trident | operators | acknowledge | once | | |
| Spark | state | RDD | exactly | | |
| Streaming | DStream | lineage | once | | |
| Samza | stateful | log of | at least | | |
| | operators | updates | once | | |
| Flink | stateful | state | exactly | | |
| | operators | checkpoint | once | | |

Table 1. State implementations across five systems.

Flink employs a single-pass algorithm that superimposes global snapshotting to normal execution [18], to support exactlyonce semantics. This approach is akin to the Chandy-Lamport algorithm, which uses markers. However, unlike the Chandy-Lamport algorithm, which assumes a strongly connected distributed system, this Flink-specific algorithm also applies to weakly connected execution graphs. Two kinds of state exist in Flink, i.e., the local state of an operator instance and the state of the whole partition. Flink programs define state in three ways: (i) using time-based, count-based, or generalized-custom windows based on transformations, (ii) registering any object type based on the checkpoint interface, and (iii) partitioning the cluster by a key based on Flink's key/value state interface. To checkpoint state, Flink offers a wide range of configurable state backends, with various levels of complexity and persistence. Currently, Flink keeps state in memory (i.e., holds state internally as objects on the Java heap), backs up state in a file system (e.g., HDFS), or persists state in RocksDB. The Flink community plans to offer additional state backend solutions (e.g., use Flink-managed memory that can spill to disk). Flink also introduces the concept of queryable state [125], which enables real-time queries to directly access event-time windows, thereby avoiding the overhead associated with writing to key/value stores. Consequently, with these enhancements to state. Flink can also support many other operations, such as software patches, testing, and system upgrades. Furthermore, researchers [125] show the complete design of state in Flink from the conceptual view to the physical level.

Although the implementations of state differ in these five frameworks, in terms of their representation and storage solutions, they all lack support for adaptive checkpointing. Currently, these five frameworks solely support periodic checkpointing (e.g., hourly checkpointing). Some researchers [94] prove that aperiodic checkpointing can improve performance over periodic checkpointing. Thus, one appealing research direction is to extend these frameworks to support adaptive checkpointing (i.e., determine when to optimally checkpoint adaptively as opposed to checkpointing periodically). We can calculate these optimal moments using the checkpointing (and recovery) costs at the time checkpoints happen. These costs, in turn, depend on the probability that failures occur. Consequently, this cost-based adaptive checkpointing model must integrate the anticipation of failure probability as an important parameter. Additionally, devising an efficient representation of state (e.g., approximate, compressed, incrementally-updateable) that enables iterative algorithms to run more efficiently is yet another opportunity for further research.

12. Open Discussions

We conclude this survey by motivating new research directions in state management. This includes novel approaches to: (i) integrate state management into big data frameworks, (2) enable state management for iterative algorithms, (3) use state to support hybrid systems, and (4) evaluate state management methods.

12.1 Integrating State Management into Big Data Frameworks

Current big data frameworks can be further extended to incorporate existing techniques for state management at varying abstraction levels, ranging from *low-level* (e.g., operator primitives, calculus algebra) to *high-level* (e.g., language level or platform level). Next, we discuss each level in greater detail.

At the *lowest* level, primitive operators can be further extended, beyond what was discussed in Section 9.5. By incorporating leading state management solutions into the current frameworks, managing state will be far easier to do and lead to greater efficiencies.

At the *calculus* level, researchers [17, 34, 47] focus on incremental state computation using algebra. Cai et al. [17] introduce a new mathematical theory (i.e., the theory of *changes* and *derivatives*) for incremental computation. Hammer et al. [47] use *first-class names* as the essential linguistic characteristic for efficient incremental computation. Fegaras [34] uses *monoid homomorphisms* as the underlying mechanism to propose an algebra for distributed computing. This algebra, which comprises an essential set of operations, serves as the formal foundation for Apache MRQL, to optimize incremental state computations. Consequently, at the algebraic level, we can extend the incremental change of state to support additional functions, beyond those discussed in this survey.

At the *high-language* level, some researchers [7, 99] devise novel declarative languages for big data processing. Silva et al. [99] propose a language to allow users to easily define and parameterize checkpointing policies. In this framework, *language annotations* are used to apply fault tolerance policies in streaming applications. Since developers understand their application semantics and failure patterns, the use of language-level annotations is believed to be a natural path to describe such policies. Further, this approach combines language primitives with code generation to facilitate checkpointing, per user specification. Beyond fault tolerance, language annotation extensions (LAE) can specify parts of an application that should be actively vs. passively (e.g., PPA scheme [100]) fault-tolerant. Additionally, LAE may be used to declare which operators should be made public (e.g., for users) vs. private (e.g., for internal operator use only).

Furthermore, Alexandrov et al. [7] propose the *Emma* language, which *deeply embeds* APIs into a host language (e.g., Scala) for complex data analysis. This approach enables the declarative specification of dataflows and the use of intermediate representations to transparently deploy parallel computations. Emma can be further extended to integrate state management methods at the language level, thereby enabling *declarative state management*.

At the *high platform* level, *Rheem* [3, 4] introduce multi-layer (i.e., *platform*, *core*, and *application*) data processing and a storage abstraction to support both *platform independence* and *interoperability* across platforms. They envision that a data processing abstraction based on user-defined functions can achieve

two purposes. First, users can solely focus on the logic of their data analytic tasks. Second, applications can be independent from data processing platforms. Rheem decomposes a complex analytic into smaller subtasks to leverage the availability of different processing platforms. This division allows a single task to run over multiple platforms to boost performance. Moreover, we can further extend Rheem to build a state management system that eases deployment on various platforms, achieves independence and interoperability among platforms, and improves performance.

In this subsection, many perspectives are presented. Some researchers have already begun to incorporate high-level support for declarative big data analysis. However, determining how to combine the strengths of each of these individual systems, in order to support state management remains a challenging research problem.

12.2 Enabling State Management for Iterative Algorithms

Many machine learning algorithms, such as PageRank, k-means, and its variants [110], require iterative steps to converge to the final solution. Due to big state sizes, some iterative algorithms use approximate state with small sizes or approximate algorithms with fewer iterative steps to boost performance. Usually, these approximate algorithms sacrifice accuracy for performance. However, some researchers develop solutions that ensure both correctness and performance. Next, we discuss these approaches and highlight areas in need of further investigation. That include mechanisms to represent state in an approximate form, approaches for optimizing approximate algorithms, and the development of exact iterative algorithms. Ultimately, these solutions focus on increasing performance.

12.2.1 Approximate State

Achieving both low-latency and high-throughput in big data processing is challenging with ever-increasing data volumes. Thus, there is a need for further research in this area, in particular, devising *approximate state* based solutions. Next, we discuss some of these.

One solution computes *quantiles* [62] for state approximation. By continuously computing the quantiles of the most recent N items in a stream, quantile queries can be answered with εN accuracy guarantees, i.e., an ε -approximate quantile summary. Additionally, researchers develop an algorithm that maintains the quantile summary, such that they can obtain quantile estimates of the n most recently observed items with $n \le N$.

Several researchers [42, 52] implement state using compact data structures [2], to compute aggregate statistics for tuples. In the past, data stream processing systems use various synopsis structures, including sampling, wavelets, sketches, and histograms. The simplest method for state approximation is state sampling (e.g., random sampling, concise sampling) [52], which estimates the underlying data with provable error guarantees. Wavelet techniques decompose data characteristics into a set of wavelet and basic functions, which are useful for hierarchical data decomposition and summarization. Sketch-based methods are a randomized version of wavelet techniques. Histogram-based methods divide data along an attribute into a set of ranges, and maintain a frequency count for each bucket. If data is vertically or horizontally divided, we have equi-width or equi-depth histograms, respectively. Recent techniques [42] explore the use of histograms for dynamic applications. Finally, micro-cluster

summarization [1], which is effective at adapting to continuously changing data streams, is proposed for multi-dimensional data.

Lastly, Jangjaimon et al. [51] propose an adaptive multi-level checkpointing mechanism that employs *delta compression* at any given time. Incremental checkpointing solely addresses recently updated and newly incoming data, whereas the use of delta compression (a.k.a. *differencing compression*) between consecutive checkpoints can further reduce checkpoint size. Delta compression stores the difference between each dirty memory page (i.e., target data) and its prior version (i.e., source data) contained in the previous checkpoint. Jangjaimon et al. dynamically select a suitable point of time to generate the smallest state size after delta compression. It differs from other adaptive checkpointing mechanisms by dynamically skipping certain fixed checkpoints.

12.2.2 Approximate Algorithms

Iterative graph processing has a large overhead, thus there is a need to develop efficient approximate algorithms to boost performance. In the research literature, the design, implementation, and usage of approximate algorithms for machine learning can accelerate computation. In this subsection, we list some of the most popular approaches [16, 65, 77, 88, 113, 119, 120] that approximate two well-known algorithms (i.e., PageRank, and k-means).

To approximate PageRank algorithms, Broder et al. [16] perform calculations on a compact representation of a graph (termed graph aggregation) and aggregate many pages onto a single node. By partitioning the graph into sub-graphs of quasiequivalent vertices, this representation requires less memory than uncompressed ones. Parreira et al. [88, 89] propose the JXP algorithm in the context of peer-to-peer network to dynamically and collaboratively compute the PageRank for websites. The main idea behind JXP is to perform local computations, so as to exploit low storage costs and coordinate interactions among peers for approximating global scores. Similarly, Yossef et al. [113] rely on local information to approximate the PageRank of a specific node. They point out that graphs lacking an abundance of high in-degree nodes and that exhibit fast convergence in PageRank random walks can efficiently use this local approximation method. Bahmani et al. [10] employ Monte Carlo methods to incrementally compute PageRank and top-k personalized PageRank.

Zhu et al. [119, 120] propose an approximate algorithm, called FastPPV, to incrementally compute a personalized PageRank vector. They apply the so-called scheduled approximation to gradually and incrementally refine estimates and measure the approximation accuracy. Liu et al. [65] use an adaptive method to sample transition matrices many times. This method adaptively and iteratively adjusts the sample rate at runtime, with an aim to balance the accuracy and performance for PageRank approximation. Mitliagkas et al. [77] propose FrogWild for partial synchronization of mirror vertices to optimize the network usage, decrease the cost in each iteration, and significantly improve the performance in PageRank algorithms.

Several researchers [11, 21, 22, 53, 117] propose approximate algorithms to accelerate *k-means* clustering. Kanungo et al. [53] develop an approximate algorithm for k-means clustering that swaps centers via single-swap and multiple-swap heuristics. Chitta et al. [21] devise a randomized approximation approach, called *approximate kernel k-means*, to reduce computational complexity and memory requirements for kernel *k-means*. The core idea is to approximate the cluster centers of a randomly chosen set of data points, say *S*, using vectors drawn from the subspace covered by *S*. Thus, it involves only a small part of the full kernel matrix, thereby

accelerating the performance of kernel k-means. Zeng et al. [117] approximate the k-means algorithm by using random spatial partitioning trees to pre-organize groups and reduce the computational overhead in the assignment step. Bahmani et al. [11] introduce an initialization algorithm that generates a near-optimal initial set of centers. It greatly reduces the number of necessary steps to generate a proper initialization, which is essential for obtaining a good solution. Finally, Cohen et al. [22] solve the general class of constrained k-rank approximation problems (including k-means) to within $(1+\varepsilon)$ error, using sketching techniques to approximate a data matrix.

Clearly, there are many approximate algorithms for machine learning that seek to reduce memory use in each iterative step. Nonetheless, determining how to approximate state using the proposed techniques in subsection 12.2.1 and integrate these into approximate algorithms to further accelerate computation remains a challenging problem.

12.2.3 Exact Algorithms

Researchers [39, 40, 41, 115] propose exact algorithms that ensure low latency. K-dash [39] finds k nodes with the closest proximities for a selected node. It is based on two principle ideas: use sparse matrices to efficiently compute the proximity of a given node and save time by computing only the necessary proximity. K-dash not only runs significantly faster than current approximate methods, but it also ensures exactness. Similarly, Fujiwara et al. [40] efficiently compute top-k nodes using a pruning technique. The key idea is to estimate the upper and lower bounds of relevance and use these to compute the top-k nodes exactly by eliminating unnecessary computations. This bound-based pruning technique efficiently generates correct results. Extending this idea, Fujiwara et al. [41] propose F-Rank to find top-k PageRank nodes efficiently, by iteratively estimating the lower and upper bounds of PageRank scores. F-Rank then constructs subgraphs in every iteration by removing unnecessary nodes and edges to get top-k nodes. Yu et al. [115] propose an algorithm, called *Inc-SR*, to incrementally compute SimRank (i.e., structural similarity between nodes based on hyperlinks) on link-evolving graphs with correctness guarantees. Inc-SR improves the incremental computation of SimRank for every link update and prunes unnecessary similarity recomputations, thereby reducing the computation time.

Seamlessly and efficiently incorporating approximate state representations (mentioned in subsection 12.2.1) into these exact algorithms is another challenging problem. Once this has been achieved, we can then compare the approximate and exact algorithms, in terms of precision and performance to determine how state approximation can help boost performance.

12.3 Using State Management for Hybrid Systems

While batch data provides comprehensive and historical views of data, real-time streaming data provides fresh and up-to-date information. Some researchers [14, 75, 76] propose hybrid systems to process these two types of data on a single platform. These hybrid systems handle both historical information and the most recent data.

The lambda architecture [71] tries to process both batch and streaming data by providing a software stack including: (1) a batch layer (e.g., implemented in Hadoop) to process batch data, (2) a speed layer (e.g., implemented in Storm) to process streaming data, and (3) a serving layer to index batch views and enable them to be queried in low-latency. This mixture of multiple systems is hard to

configure, manage, and maintain due to their diversity and heterogeneity. Moreover, data analysis tasks generally involve multiple systems, thereby limiting optimization opportunities. Thus, we cannot process data as efficiently as a single unified system.

To partially overcome this weakness in the lambda architecture, researchers [14, 75, 76] propose hybrid systems that integrate multiple data types (e.g., real-time with batch or streaming with OLAP). Boykin et al. [14] propose Summingbird to combine online and batch MapReduce computations into a single framework. To fuse the stream and transaction processing in a single system, Meehan et al. [75] build S-Store, initially beginning with a completely transactional OLTP database system and then integrating additional streaming functionality. This allows S-Store to simultaneously and seamlessly support OLTP and streaming applications. Meehan et al. [76] design BigDAWG to tightly integrate stream and batch processing to enable seamless and high performance querying capability over both new and historical data. Elmore et al. [30] demonstrate the effectiveness of BigDAWG through practical applications.

Systems, such as S-Store and Summingbird do not directly focus on combining batch and streaming data in a single system. Consequently, future research can encapsulate the entire functionality in a lambda architecture into a single system to take advantages of both batch and streaming worlds. Then devising novel state checkpointing methods is an essential requirement for stateful hybrid applications. Moreover, proposing new ways to manage state in incremental computations for both batch and streaming data in a single framework is a fascinating research problem. Batch and streaming data have specific characteristics. Thus, additional research will need to be conducted to develop novel methods for efficient state management that meets both batch and streaming data requirements.

12.4 Evaluating State Management Methods

Besides the research problems mentioned in subsections 12.1, 12.2 and 12.3, identifying evaluation methods for the proposed solutions is highly important. Which standards or criteria must we use to evaluate these state management solutions? As a starting point, we propose the following five metrics, listed below.

- Efficiency: State management methods must have a lower overhead over existing approaches.
- Ease of use/management: APIs that use and access state
 must be simple and easy to use. They cover most application
 scenarios and provide richer functions and encapsulations.
 This will help to reduce the human latency cost in deploying
 and using big data frameworks in the future.
- Functionality: State can efficiently support iterative algorithms in many different domains, such as machine learning, data mining, and artificial intelligence. In this case, it must support multiple consistency guarantees and allow users to choose which consistency level to use.
- Seamless integration: New methods should easily integrate
 into existing, ongoing, and future frameworks for big data
 processing. The integration must be effective (i.e., without
 spending too much effort to modify the existing underlying
 platforms).
- Benchmarks and metrics for state management evaluation: One appealing research direction is to introduce

benchmarks and devise metrics for state management solutions.

13. Conclusion

In this paper, we survey the *state management* research for big data processing applications across several dimensions. First, we discuss how state enables *stateful computation*, to support complex operations that combine state and input. Second, we illustrate how state is used to enable *fault tolerance* and facilitate failure recovery. Third, we showcase how state is used to accelerate *iterative computation* via the reuse of state values. Fourth, we demonstrate how state ensures *elasticity and load balance* in systems, by flexibly splitting workloads among computing nodes. Fifth, we present how state can be utilized to enable *multi-optimizations* in a single system (e.g., incremental and iterative computation, fault tolerance and elasticity). Sixth, we highlight how state can be *shared* among operations, to more efficiently utilize resources and reuse computational results.

Further, we discuss varying approaches to *incrementally maintain* state, to prevent full state maintenance, which is expensive. In addition, we present a variety of methods that can be used to handle state, including storing, updating, purging, migrating, and exposing. Moreover, we consider state management overhead and complexity related issues. Furthermore, we compare state implementation among five popular frameworks and pointed out their limitations. Consequently, this enables us to propose new research directions to overcome these limitations (e.g., adaptive checkpointing). Finally, in Appendix A, we list state management research contributions by dimension. We hope this survey will pave the way for subsequent state management research (e.g., state integration and approximation, state usage in hybrid systems, evaluation metrics) for big data processing systems.

Acknowledgments

This work was funded by the H2020 STREAMLINE project under grant agreement No 688191 and by the German Federal Ministry for Education and Research (BMBF) funded Berlin Big Data Center (BBDC), under funding mark 01IS14013A.

References

- [1] C. Aggarwal, J. Han, J. Wang, P. Yu. A Framework for Clustering Evolving Data Streams. In *International Conference on Very Large Data Bases (VLDB)*, pages 81-92, 2003.
- [2] C. Aggarwal, P. Yu. A survey of synopsis construction in data streams. In Data Streams, Advances in Database Systems, vol. 31. Springer, New York, 2007.
- [3] D. Agrawal, S. Crawla, A. Elmagarmind, Z. Saoudi, M. Ouzzani, P. Papati, J.-A. Quiane-Ruiz, N. Tang, and M. J. Zaki. Road to Freedom in Big Data Analytics. In *International Conference on Extending Database Technology (EDBT)*, pages 479-484, 2016.
- [4] D. Agrawal et al. Rheem: Enabling Multi-Platform Task Execution. In ACM International Conference on Management of Data (SIGMOD), pages 2069-2072, 2016.
- [5] Y. Ahmad, O. Kennedy, C. Koch, M. Nikolic. DBToaster: Higher-order Delta Processing for Dynamic, Frequently Fresh Views. In *Proceedings of the VLDB Endowment (PVLDB)*, 5(10):968-979, 2012.
- [6] A. Alexandrov et al. The Stratosphere platform for big data analytics. VLDB Journal, 23(6):939-964, 2014.
- [7] A. Alexandrov et al. Implicit Parallelism through Deep Language Embedding. In ACM International Conference on Management of Data (SIGMOD), pages 47-61, 2015.
- [8] R. Ananthanarayanan, et al. Photon: fault-tolerant and scalable joining of continuous data streams. In ACM International Conference on Management of Data (SIGMOD), pages 577-588, 2013.
- [9] A. Arasu, S. Babu, and J. Widom. The CQL Continuous Query Language: Semantic Foundations and Query Execution. VLDB Journal, 15(2):121–142, 2006.
- [10] B. Bahmani, A. Chowdhury, and A. Goel. Fast incremental and personalized PageRank. In *Proceedings of the VLDB Endowment (PVLDB)*, 4(3):173-184, 2010.
- [11] B. Bahmani, B. Moseley, A. Vattani, R. Kumar, and S. Vassilvitskii. Scalable k-means++. In *Proceedings of the VLDB Endowment (PVLDB)*, 5(7):622-633, 2012.
- [12] O. Benjelloun, A. D. Sarma, A. Halevy, and J. Widom. ULDBs: Databases with Uncertainty and Lineage. In International Conference on Very Large Data Bases (VLDB), pages 953–964, 2006.
- [13] M. S. Bouguerra, D. Trystram, F. Wagner. Complexity Analysis of Checkpoint Scheduling with Variable Costs. *IEEE Transactions on Computers*, 62(6):1269-1275, 2013.
- [14] O. Boykin, S. Ritchie, I. O'Connell, J. Lin. Summingbird: A Framework for Integrating Batch and Online MapReduce Computations. In *Proceedings of the VLDB Endowment* (PVLDB), 7(13):1441-1451, 2014.
- [15] A. Brito, C. Fetzer, H. Sturzrehm, and P. Felber. Speculative Out-of-Order Event Processing with Software Transaction Memory. In *Distributed Event-Based Systems (DEBS)*, pages 265-275, 2008.
- [16] A. Z. Broder, R. Lempel, F. Maghoul, and J. Pedersen. Efficient PageRank approximation via graph aggregation. *Information Retrieval*, 9(2):123-138, 2006.

- [17] Y. Cai, P. G. Giarrusso, T. Rendel, and K. Ostermann. A theory of changes for higher-order languages: incrementalizing λ-calculi by static differentiation. In *Programming Language Design and Implementation (PLDI)*, pages 145-155, 2014.
- [18] P. Carbone, G. Fóra, S. Ewen, S. Haridi, K. Tzoumas. Lightweight Asynchronous Snapshots for Distributed Dataflows. *The Computing Research Repository (CoRR)*, abs/1506.08603, 2015.
- [19] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, K. Tzoumas. Apache FlinkTM: Stream and Batch Processing in a Single Engine. *IEEE Data Engineering Bulletin*, 38(4):28-38, 2015.
- [20] Z. Chen, J. Dongarra. Highly Scalable Self-Healing Algorithms for High Performance Scientific Computing. *IEEE Transactions on Computers*, 58(11):1512-1524, 2009.
- [21] R. Chitta, R. Jin, T. C. Havens, and A. K. Jain. Approximate kernel k-means: solution to large scale kernel clustering. In Knowledge discovery and data mining (KDD), pages 895-903, 2011.
- [22] M. B. Cohen, S. Elder, C. Musco, and M. Persu. Dimensionality Reduction for k-means Clustering and Low Rank Approximation. In Symposium on Theory of Computing (STOC), pages 163-172, 2015.
- [23] T. Condie, N. Conway, P. Alvaro, and J. M. Hellerstein. MapReduce online. In NSDI, 2010.
- [24] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Communication of ACM*, 51(1):107–113, 2008.
- [25] L. Ding, N. Mehta, E. A. Rundensteiner, G. T. Heineman. Joining Punctuated Streams. In *International Conference on Extending Database Technology (EDBT)*, pages 587-604, 2004
- [26] J. Ding, T. Z. J. Fu, R. T. B. Ma, M. Winslett, Y. Yang, Z. Zhang, H. Chao. Efficient Operator State Migration for Cloud-Based Data Stream Management Systems. *The Computing Research Repository (CoRR)*, abs/1501.03619, 2016.
- [27] S. Dudoladov, C. Xu, S. Schelter, A. Katsifodimos, S. Ewen, K. Tzoumas, V. Markl. Optimistic Recovery for Iterative Dataflows in Action. In ACM International Conference on Management of Data (SIGMOD), pages 1439-1443, 2015.
- [28] C. Doulkeridis and K. Nørvåg. A Survey of Large-Scale Analytical Query Processing in MapReduce. VLDB Journal, 23(3):355-380, 2014.
- [29] J. Ekanayake and G. Fox. High performance parallel computing with clouds and cloud technologies. In International Conference on Cloud Computing (CloudComp), 2009.
- [30] A. Elmore et al. A Demonstration of the BigDAWG Polystore System. In *Proceedings of the VLDB Endowment (PVLDB)*, 8(12), 2015.
- [31] S. Ewen, K. Tzoumas, M. Kaufmann, and V. Markl. Spinning fast iterative data flows. In *Proceedings of the VLDB Endowment (PVLDB)*, 5(11):1268–1279, 2012.
- [32] S. Ewen, S. Schelter, K. Tzoumas, D. Warneke, V. Markl. Iterative parallel data processing with stratosphere: an inside look. In ACM International Conference on Management of Data (SIGMOD), pages 1053-1056, 2013.

- [33] L. Fegaras. Incremental Query Processing on Big Data Streams. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 2016.
- [34] L. Fegaras. An Algebra for Distributed Big Data Analytics. Technical Report, 2016.
- [35] Y.-H. Feng, et al. Efficient and Adaptive Stateful Replication for Stream Processing Engines in High-Availability Cluster. *IEEE Transactions on Parallel and Distributed Systems* (TPDS), 22(11):1788-1796, 2011.
- [36] R. C. Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch. Integrating Scale Out and Fault Tolerance in Stream Processing using Operator State Management. In ACM International Conference on Management of Data (SIGMOD), 2013.
- [37] R. C. Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch. Making State Explicit for Imperative Big Data Processing. In USENIX ATC, 2014.
- [38] R. C. Fernandez, P. Garefalakis, P. Pietzuch. Java2SDG: Stateful Big Data Processing for the Masses. In *International Conference on Data Engineering (ICDE)*, pages 1390-1393, 2016.
- [39] Y. Fujiwara, M. Nakatsuji, M. Onizuka, and M. Kitsuregawa. Fast and exact top-k search for random walk with restart. In *Proceedings of the VLDB Endowment (PVLDB)*, 5(5):442-453, 2012.
- [40] Y. Fujiwara, M. Nakatsuji, T. Yamamuro, H. Shiokawa, and M. Onizuka. Efficient personalized pagerank with accuracy assurance. In *Knowledge discovery and data mining (KDD)*, pages 15-23, 2012.
- [41] Y. Fujiwara, M. Nakatsuji, H. Shiokawa, T. Mishima, and M. Onizuka. Fast and exact top-k algorithm for pagerank. In *Conference on Artificial Intelligence (AAAI)*, pages 1106-1112, 2013.
- [42] M. Garofalakis, J. Gehrke, R. Rastogi. Querying and mining data streams: you only get one look (a tutorial). In ACM International Conference on Management of Data (SIGMOD), 2002.
- [43] B. Gedik. Partitioning functions for stateful data parallelism in stream processing. *VLDB Journal*, 23(4):517-539, 2014.
- [44] M. I. Gordon, W. Thies, and S. Amarasinghe. Exploiting Coarse-Grained Task, Data, and Pipeline Parallelism in Stream Programs. In Architectural Support for Programming Languages and Operating Systems (ASPLOS), pages 151-162, 2006
- [45] V. Gulisano, R. J.-Peris, M. P.-Martínez, C. Soriente, P. Valduriez. StreamCloud: An Elastic and Scalable Data Stream System. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 23(12):2351-2365, 2012.
- [46] D. Hakkarinen and Z. Chen. Multilevel Diskless Checkpointing. *IEEE Transactions on Computers*, 62(4):772-783, 2013.
- [47] M. A. Hammer, J. Dunfield, K. Headley, N. Labich, J. S. Foster, M. Hicks, and D. V. Horn. 2015. Incremental computation with names. SIGPLAN, 50(10):748-766, 2015.
- [48] M. Hirzel, R. Soulé, S. Schneider, B. Gedik, and R. Grimm. A catalog of stream processing optimizations. ACM Computing Surveys (CSUR), 46(4), 2014.

- [49] J. H. Hwang, M. Balazinska, A. Rasin, U. Cetintemel, M. Stonebraker, S. Zdonik. High-availability algorithms for distributed stream processing. In *International Conference on Data Engineering (ICDE)*, pages 779–790, 2005.
- [50] J. H. Hwang, Y. Xing, U. Cetintemel, S. Zdonik. A Cooperative, Self-Configuring High-Availability Solution for Stream Processing. In *International Conference on Data Engineering (ICDE)*, 2007.
- [51] I. Jangjaimon and N.-F. Tzeng. Adaptive incremental checkpointing via delta compression for networked multicore systems. In *International Parallel & Distributed Processing Symposium (IPDPS)*, pages 7-18, 2013.
- [52] T. Johnson, S. Muthukrishnan, and I. Rozenbaum. Sampling algorithms in a stream operator. In ACM International Conference on Management of Data (SIGMOD), pages 1-12, 2005
- [53] T. Kanungo, D. M. Mount, N. S. Netanyahu, C. D. Piatko, R. Silverman, and A. Y. Wu. A local search approximation algorithm for k-means clustering. In *symposium on Computational geometry (SCG)*, pages 10-18, 2002.
- [54] C. Koch. Incremental query evaluation in a ring of databases. In Symposium on Principles of Database Systems (PODS), pages 87-98, 2010.
- [55] C. Koch, Y. Ahmad, O. Kennedy, M. Nikolic, A. Nötzli, D. Lupei, A. Shaikhha. DBToaster: higher-order delta processing for dynamic, frequently fresh views. *VLDB Journal*, 23(2):253-278, 2014.
- [56] C. Koch, D. Lupei, V. Tannen. Incremental View Maintenance for Collection Programming. In Symposium on Principles of Database Systems (PODS), pages 75-90, 2016.
- [57] Y. Kwon, M. Balazinska, and A. Greenberg. Fault-tolerant stream processing using a distributed, replicated file system. In *Proceedings of the VLDB Endowment (PVLDB)*, 1(1):574-585, 2008.
- [58] B. Koldehofe, R. Mayer, U. Ramachandran, K. Rothermel, M. Völz. Rollback-recovery without checkpoints in distributed event processing systems. In *Distributed Event-Based Systems* (DEBS), pages 27-38. 2013.
- [59] R. Kuntschke, B. Stegmaier, A. Kemper. Data Stream Sharing. Technical Report, TU Munich, 2005.
- [60] H. G. Li, S. Chen, J. Tatemura, D. Agrawal, K. S. Candan, W. P. Hsiung. Safety Guarantee of Continuous Join Queries over Punctuated Data Streams. In *International Conference on Very Large Data Bases (VLDB)*, pages 19-30, 2006.
- [61] J. Li, K. Tufte, V. Shkapenyuk, V. Papadimos, T. Johnson, and D. Maier. Out-of-order processing: a new architecture for high-performance stream systems. In *Proceedings of the* VLDB Endowment (PVLDB), 1(1):274-288, 2008.
- [62] X. Lin, H. Lu, J. Xu, J.X. Yu. Continuously maintaining quantile summaries of the most recent N elements over a data stream. In *International Conference on Data Engineering* (ICDE), pages 362-373, 2004.
- [63] B. Liu, Y. Zhu, E. A. Rundensteiner. Run-Time Operator State Spilling for Memory Intensive Long-Running Queries. In ACM International Conference on Management of Data (SIGMOD), pages 347-358, 2006.

- [64] M. Liu, Z. G. Ives, and B. T. Loo. Enabling Incremental Query Re-Optimization. In ACM International Conference on Management of Data (SIGMOD), pages 1705-1720, 2016.
- [65] W. Liu, G. Li, and J. Cheng. Fast PageRank approximation by adaptive sampling. *Knowledge of Information System*, 42(1):127-146, 2015.
- [66] D. Logothetis, K. Yocum. Data Indexing for Stateful, Largescale Data Processing. In NETDB, 2009.
- [67] D. Logothetis, C. Olston, B. Reed, K.C. Webb, K. Yocum. Stateful Bulk Processing for Incremental Analytics. In ACM Symposium on Cloud Computing (SoCC), pages 51-62, 2010.
- [68] G. Losa, V. Kumar, H. Andrade, B. Gedik, M. Hirzel, R. Soulé, and K.-L. Wu. CAPSULE: language and system support for efficient state sharing in distributed stream processing systems. In *Distributed Event-Based Systems* (DEBS), pages 268-277, 2012.
- [69] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, J. Hellerstein. Distributed GraphLab: A Framework for Machine Learning in the Cloud. In *Proceedings of the VLDB Endowment (PVLDB)*, 5(8):716-727, 2012.
- [70] V. Markl: Breaking the Chains: On Declarative Data Analysis and Data Independence in the Big Data Era. In *Proceedings of the VLDB Endowment (PVLDB)*, 7(13):1730-1733, 2014.
- [71] N. Marz and J. Warren. Big Data: Principles and best practices of scalable realtime data systems. ISBN 9781617290343, 328 pages, 2015.
- [72] T. D. Matteis, G. Mencagli. Parallel Patterns for Window-Based Stateful Operators on Data Streams: An Algorithmic Skeleton Approach. *Journal of Parallel Programming*, pages 1-20, 2016.
- [73] F. McSherry, R. Isaacs, M. Isard, D. G. Murray. Composable Incremental and Iterative Data-Parallel Computation with Naiad. Technical Report number MSR-TR-2012-105. Microsoft Research Silicon Valley, 2012.
- [74] F. McSherry, D. G. Murray, R. Isaacs, M. Isard. Differential Dataflow. In CIDR, 2013.
- [75] J. Meehan et al. S-Store: streaming meets transaction processing. In *Proceedings of the VLDB Endowment* (PVLDB), 8(13):2134-2145, 2015.
- [76] J. Meehan, S. Zdonik, S. Tian, Y. Tian, N. Tatbul, A. Dziedzic, A. Elmore. Integrating Real-Time and Batch Processing in a Polystore. In *High-Performance Extreme Computing Conference (HPEC)*, 2016.
- [77] I. Mitliagkas, M. Borokhovich, A. G. Dimakis, and C. Caramanis. FrogWild!: fast PageRank approximations on graph engines. In *Proceedings of the VLDB Endowment* (PVLDB), 8(8):874-885, 2015.
- [78] M. Mokbel, M. Lu, and W. Aref. Hash-merge join: A non-blocking join algorithm for producing fast and early join results. In *International Conference on Data Engineering (ICDE)*, pages 251-262, 2004.
- [79] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: a timely dataflow system. In ACM Symposium on Operating Systems Principles (SOSP), pages 439-455, 2013.
- [80] K. G. S. Madsen, Y. Zhou. Dynamic Resource Management in a Massively Parallel Stream Processing Engine. In ACM

- International on Conference on Information and Knowledge Management (CIKM), pages 13-22, 2015.
- [81] K. G. S. Madsen, Y. Zhou, J. Cao. Integrative Dynamic Reconfiguration in a Parallel Stream Processing Engine. *The Computing Research Repository (CoRR)*, abs/1602.03770, 2016.
- [82] N. Naksinehaboon, Y. Liu, C. B. Leangsuksun, R. Nassar, M. Paun, S. L. Scott. Reliability-Aware Approach: An Incremental Checkpoint/Restart Model in HPC Environments. In CCGRID, pages 783-788, 2008.
- [83] B. Nicolae and F. Cappello. AI-Ckpt: leveraging memory access patterns for adaptive asynchronous incremental checkpointing. In *High-performance parallel and distributed* computing (HPDC), pages 155-166, 2013.
- [84] M. Nikolic, M. Elseidy, C. Koch. LINVIEW: incremental view maintenance for complex analytical queries. In ACM International Conference on Management of Data (SIGMOD), pages 253-264, 2014.
- [85] M. Nikolic, M. Dashti, C. Koch. How to Win a Hot Dog Eating Contest: Distributed Incremental View Maintenance with Batch Updates. In ACM International Conference on Management of Data (SIGMOD), pages 511-526, 2016.
- [86] B. Ottenwalder, B. Koldehofe, K. Rothermel, and U. Ramachandran. MigCEP: operator migration for mobility driven distributed complex event processing. In *DEBS*, pages 183-194, 2013.
- [87] S. Padmanabhan, T. Malkemus, A. Jhingran, and R. Agarwal. Block Oriented Processing of Relational Database Operations in Modern Computer Architectures. In *International Conference on Data Engineering (ICDE)*, pages 567–574, 2001.
- [88] J. X. Parreira, D. Donato, S. Michel, and G. Weikum. Efficient and decentralized PageRank approximation in a peer-to-peer web search network. In *Proceedings of the VLDB Endowment* (PVLDB), 415-426, 2006.
- [89] J. X. Parreira, Carlos Castillo, Debora Donato, Sebastian Michel, Gerhard Weikum. The JXP Method for Robust PageRank Approximation in a Peer-to-Peer Web Search Network. The VLDB Journal, 17(2):291-313, 2008.
- [90] M. Paun, N. Naksinehaboon, R. Nassar, C. Leangsuksun, S. L. Scott, N. Taerat. Incremental Checkpoint Schemes for Weibull Failure Distribution. *Journal on Foundation of Computer Science*, 21(3):329-344, 2010.
- [91] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, M. Seltzer. Network-aware operator placement for stream-processing systems. In *International Conference on Data Engineering (ICDE)*, 2006.
- [92] K. Ren, T. Diamond, D. J. Abadi, and A. Thomson. Low-Overhead Asynchronous Checkpointing in Main-Memory Database Systems. In ACM International Conference on Management of Data (SIGMOD), pages 1539-1551, 2016.
- [93] Y. Robert, F. Vivien, D. Zaidouni. On the complexity of scheduling checkpoints for computational workflows. In Dependable Systems & Networks (DSN), pages 1-6, 2012.
- [94] N. E. Sayed and B. Schroeder. Checkpoint/Restart in Practice: When Simple is Better. In *IEEE International Conference on Cluster Computing (CLUSTER)*, pages 84-92, 2014.

- [95] S. Schelter, S. Ewen, K. Tzoumas, V. Markl. "All Roads Lead to Rome:" Optimistic Recovery for Distributed Iterative Data Processing. In ACM International on Conference on Information and Knowledge Management (CIKM), pages 1919-1928, 2013.
- [96] Z. Sebepou, and K. Magoutis. CEC: Continuous Eventual Checkpointing for data stream processing operators. In International Conference on Dependable Systems & Networks (DSN), pages 145-156, 2011.
- [97] J. Sermulins, W. Thies, R. Rabbah, and S. Amarasinghe. Cache Aware Optimization of Stream Programs. In Languages, Compiler, and Tool Support for Embedded Systems (LCTES), pages 115-126, 2005.
- [98] M. A. Shah, J. M. Hellerstein, S. Chandrasekaran and M. J. Franklin. Flux: An Adaptive Partitioning Operator for Continuous Query Systems. In *International Conference on Data Engineering (ICDE)*, 2003.
- [99] G. J. Silva, B. Gedik, H. Andrade, K.-L. Wu. Language Level Checkpointing Support for Stream Processing Applications. In *Dependable Systems & Networks (DSN)*, 2009.
- [100] L. Su and Y. Zhou. Tolerating correlated failures in Massively Parallel Stream Processing Engines. In International Conference on Data Engineering (ICDE), pages 517-528, 2016.
- [101] N. Tatbul, S. Zdonik, J. Meehan, C. Aslantas, M. Stonebraker, K. Tufte, C. Giossi, H. Quach. Handling Shared, Mutable State in Stream Processing with Correctness Guarantees. *IEEE Data Engineering Bulletin*, 38(4):94-104, 2015
- [102] A. Toshniwal et al. Storm@twitter. In ACM International Conference on Management of Data (SIGMOD), pages 147-156, 2014.
- [103] Y. -C. Tu, S. Liu, S. Prabhakar, and B. Yao. Load shedding in stream databases: a control-based approach. In *International Conference on Very Large Data Bases (VLDB)*, pages 787-798, 2006.
- [104] P. A. Tucker, D. Maier, T. Sheard, and L. Fegaras. Exploiting punctuation semantics in continuous data streams. *IEEE Transactions on Knowledge and Data Engineering* (TKDE), 15(3):555–568, 2003.
- [105] P. Upadhyaya et al. A Latency and Fault-Tolerance Optimizer for Online Parallel Query Plans. In *ACM* International Conference on Management of Data (SIGMOD), pages 241-252, 2011.
- [106] T. Urhan and M. J. Franklin. Xjoin: A reactively-scheduled pipelined join operator. *IEEE Data Engineering Bulletin*, 23(2):27–33, 2000.
- [107] S. Viglas, J. F. Naughton, and J. Burger. Maximizing the output rate of multi-way join queries over streaming information sources. In *International Conference on Very Large Data Bases (VLDB)*, pages 285–296, 2003.
- [108] H. Wang, L.-S. Peh, E. Koukoumidis, S. Tao, M. C. Chan. Meteor Shower: A Reliable Stream Processing System for Commodity Data Centers. In *IEEE IPDPS*, pages 1180-1191, 2012.
- [109] M. Weimer, T. Condie, and R. Ramakrishnan. Machine learning in ScalOps, a higher order cloud computing language. In NIPS BigLearn, Vol. 9, pages 389-396, 2011.

- [110] X. Wu, et al. Top 10 algorithms in data mining. Knowledge Information System, 14(1):1-37, 2007.
- [111] Y. Wu, K. Tan. ChronoStream: elastic stateful stream computation in the cloud. In *International Conference on Data Engineering (ICDE)*, pages 723-734, 2015.
- [112] C. Xu, M. Holzemer, M. Kaul, V. Markl. Efficient fault-tolerance for iterative graph processing on distributed dataflow systems. In *International Conference on Data Engineering (ICDE)*, pages 613-624, 2016.
- [113] Z. B. Yossef and L. Mashiach. Local approximation of PageRank and reverse PageRank. In *Research and development in information retrieval (SIGIR)*, pages 865-866, 2008.
- [114] J. W. Young. A first order approximation to the optimum checkpoint interval. *Communication of ACM*, 17(9), 1974.
- [115] W. Yu, X. Lin, W. Zhang. Fast incremental SimRank on link-evolving graphs. In *International Conference on Data Engineering (ICDE)*, pages 304-315, 2014.
- [116] M. Zaharia et al. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In NSDI, 2012.
- [117] G. Zeng. Fast approximate k-means via cluster closures. In Computer Vision and Pattern Recognition (CVPR), pages 3037-3044, 2012.
- [118] H. Zhang, G. Chen, B. C. Ooi, K. L. Tan, M. Zhang. Inmemory big data management and processing: A survey. *IEEE Transactions on Knowledge and Data Engineering* (TKDE), 27 (7):1920-1948, 2015.
- [119] F. Zhu, Y. Fang, K. C.-C. Chang, and J. Ying. Incremental and accuracy-aware personalized pagerank through scheduled approximation. In *Proceedings of the* VLDB Endowment (PVLDB), 6(6):481-492, 2013.
- [120] F. Zhu, Y. Fang, K. C.-C. Chang, and J. Ying. Scheduled approximation for Personalized PageRank with Utility-based Hub Selection. VLDB Journal, 24(5):655-679, 2015.
- [121] Y. Zhu, E. Rundensteiner, G. T. Heineman. Dynamic Plan Migration for Continuous Queries over Data Streams. In ACM International Conference on Management of Data (SIGMOD), 2004.
- [122] M. Zinkevich, M. Weimer, A. J. Smola, and L. Li. Parallelized stochastic gradient descent. In *Neural Information Processing Systems (NIPS)*, pages 2595–2603, 2010.
- [123] P. Van Roy, S. Haridi. Concepts, Techniques, and Models of Computer Programming. MIT Press, Cambridge, 2004.
- [124] S. Sakr, A. Liu, A. Fayoumi. The Family of MapReduce and Large Scale Data Processing Systems. Journal of ACM Computing Surveys (ACM CSUR), 46(1), 2013.
- [125] P. Carbone, S. Ewen, G. Fóra, S. Haridi, S. Richter, K. Tzoumas. State Management in Apache Flink®: Consistent Stateful Distributed Stream Processing. PVLDB 10(12): 1718-1729, 2017.

Appendix A. Classification of State Management Across Approaches by Functionality

| Approach/Paper/ System | Stateful Computation | Fault Tolerance | Iterative Processing | Elasticity & Load Balance | Integrative Optimization | State Sharing | Incremental Maintenance | Operations | Overhead Complexity |
|---|-------------------------|--------------------|-------------------------|---------------------------------|-----------------------------|------------------|----------------------------|------------|------------------------|
| Flink | 1 | 1 | 1 | 1 | | | | | |
| Spark | 1 | 1 | | | | | | | |
| Storm | | 1 | | | | | | | |
| Samza | 1 | 1 | | | | | | | |
| Logothetis et al. [67] | 1 | | | | | | | 1 | |
| Logothetis et al. [66] | 1 | | | | | | | | |
| Gedik et al. [43] | 1 | | | 1 | 1 | | | | |
| Matteis et al. [72] | 1 | | | | | | | | |
| CEC [96] | | 1 | | | | | | | |
| Hwang et al. [50] | | 1 | | | | | | | |
| scalable coding strategies [20] | | 1 | | | | | | | |
| Meteor Shower [108] | | 1 | | | | | | | |
| Koldehofe et al. [58], Hakkarinen et al. [46] | | 1 | | | | | | | |
| PPA [100] | | 1 | | | | | | | |
| FTOpt [105] | | 1 | | | | | | | |
| AI-Ckpt [83] | | 1 | | | | | | 1 | |
| Naksinehaboon et al. [82] | | 1 | | | | | | | / |
| Paun et al. [90] | | 1 | | | | | | | |
| AIC [51] | | 1 | | | | | | | |
| ABS [18] | | 1 | | | | | | | |
| Ewen et al. [31, 32] | | | 1 | | | | | | |
| Schelter et al. [95], Dudoladov et al. [27] | | 1 | 1 | | | | | | |
| head & tail checkpoint [112] | | 1 | 1 | | | | | | |
| MRQL Streaming [33] | | | 1 | | | | / | / | |
| Flux [98] | | | | 1 | | | | | |
| ChronoStream [111] | | 1 | | 1 | 1 | | | 1 | |
| differential dataflow [73] | | | | | 1 | | | | |
| Naiad [74] | | | | | 1 | | | | |
| Fernandez et al. [36] | | 1 | | | 1 | | | 1 | 1 |
| Madsen et al. [80, 81] | | | | | 1 | | | | |
| Brito et al. [15] | | | | | | 1 | | | |
| Gordon et al. [44], Arasu et al. [9] | | | | | | / | | | |

| Sermulins et al. [97] | | | 1 | | | |
|--|---|--|---|---|---|---|
| Kuntschke et al. [59] | | | 1 | | | |
| CAPSULE [68] | | | 1 | | | |
| S-Store [75], [101] | | | 1 | | | |
| ring of databases [54] | | | | 1 | | |
| viewlet transforms [5] | | | | 1 | | |
| DBToaster [55] | | | | 1 | | |
| LINVIEW [84] | | | | 1 | | |
| Nikolic et al. [85], Koch et al. [56] | | | | 1 | | |
| Liu et al. [64] | | | | 1 | | |
| Zhang et al. [118] | | | | | 1 | |
| Liu et al. [63] | | | | | 1 | |
| SGuard [57] | 1 | | | | 1 | |
| CALC [92] | | | | | 1 | |
| Photon [8] | | | | | 1 | |
| SDG [37] | | | | | 1 | |
| GraphLab [69] | | | | | 1 | |
| PJoin [25] | | | | | 1 | |
| Punctuation semantics [104] | | | | | 1 | |
| Li et al. [61], Zhu et al. [121] | | | | | 1 | |
| Ding et al. [26] | | | | | 1 | 1 |
| StreamCloud [45] | | | | | 1 | |
| SBON [91] | | | | | 1 | |
| MigCEP [86] | | | | | 1 | |
| MLCBF [35] | | | | | 1 | |
| Sayed et al. [94] | | | | | | 1 |
| Robert et al. [93] | | | | | | 1 |
| Bouguerra et al. [13] | | | | | | 1 |