A Survey of State Management in Big Data Processing Systems

Quoc-Cuong To\textsuperscript{1} \quad Juan Soto\textsuperscript{1,2} \quad Volker Markl\textsuperscript{1,2}

\textsuperscript{1}DFKI
Alt-Moabit 91c
10559 Berlin, Germany

\texttt{quoc_cuong.to@dfki.de}

\textsuperscript{2}TU Berlin
Sekr. E-N 7, Einsteinufer 17
10587 Berlin, Germany

\texttt{juan.soto@tu-berlin.de}

\texttt{volker.markl@tu-berlin.de}

ABSTRACT
The management and usage of state are issues of paramount importance in big data processing systems (BDPS) today. They are increasingly gaining attention due to their utility in supporting complex operations in various applications. In this paper, we survey state management in BDPS and introduce a taxonomy to classify current research in this field by key characteristics. First, we present some of the most important applications of state. Then, we discuss the handling and implementation of state. Finally, we show some potential research directions in state management. This article aims to provide insight into the varying state management techniques in the literature, motivate others to pursue research in this field, and present some open questions.

CCS Concepts
- information systems\•database management system engines\•computing methodologies\•massively parallel and high-performance computer systems

Keywords
State management, big data processing, taxonomy, and survey

1. Introduction
Big data frameworks aim to process massive amounts of data efficiently with fast response time. These frameworks generally fall into two categories, namely, batch-oriented and stream-oriented processing. Batch-oriented processing operates on chunks of large data files, whereas stream-oriented processing handles continuously arriving data.

For batch-oriented data processing, one of the first proposals was MapReduce [Dean08], which became popularized via Hadoop, an open source batch processing system. Due to its features, such as flexibility, fault-tolerance, programming ease, and scalability, the MapReduce model attracted a large number of works. Today, it is widely regarded as the pioneer for large-scale batch-oriented data analysis. However, despite its merits, MapReduce has several drawbacks (e.g., redundant processing, lack of iteration), which severely affects performance. Consequently, its alternatives [Doulkeridis14] were proposed to overcome its limitations. For a comprehensive survey on batch-oriented data processing, interested readers are encouraged to examine [Doulkeridis14]. In contrast, with an ever-increasing number of real-time applications demanding both low latency and high throughput, the need for novel data stream processing solutions arose, bringing numerous frameworks (e.g., Flink [Alexandrov14], Spark [Zaharia12]) into existence.

The most important components in BDPS are operators that receive and transform input data directly. The fundamental processing logic of operators includes consuming and transforming input, and then producing output. Despite their diversity, operators generally come in two varieties, namely, stateless and stateful. While the former (e.g., select, project, filter) processes data on an element-by-element basis without maintaining any data structure, the latter (e.g., sort, join, aggregate) holds and updates an internal data structure while processing input data, so that this data structure can participate to the subsequent computations. This internal data structure, called state, is very important because its current value can directly affect the result of the processing logic.

However, some large-scale batch data processing frameworks, such as MapReduce, limit programmers on using state in data-parallel programs. Previous approaches (e.g., online MapReduce systems [Condie10], Twister [Ekanyane09]) that incorporate state “can result in custom, fragile code and disappointing performance” [Logothetis10]. Therefore, some stream processing frameworks, such as Flink [Alexandrov14], were proposed to smoothly incorporate state into programs. Since then, a large body of researchers has proposed new ways to represent, manage and use state. For example, the windowing concept is the predominant abstraction used to implement the internal state of operators in data stream processing [Matteis16]. Other data structures can also be used to represent the various state types [Fernandez13].

Due to its importance, state usage has increasingly received much attention. Systems researchers are ardously 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 a number of applications?” Indeed, state can be used in various scenarios. Besides stateful computation that requires both current state value and input data, keeping state in a reliable location can help to achieve fault tolerance when we need efficient recovery, in the event of failure. Furthermore, state can facilitate the iterative processing of most machine learning algorithms, where iteration is inevitable. State also aids in achieving elasticity and load balance. Furthermore, state can leverage integrative optimizations, including fission, migration, and scaling.
Managing state efficiently presents numerous challenges. For example, to save storage, state can be shared among different processes [Brito08]. To speed up performance, state can be maintained incrementally [Fegaras16]. In terms of storage, state can be stored remotely or locally, using in-memory techniques [Ren16, Nicolae13] or disk spilling methods [Liu06]. It can even be geographically maintained across distant locations [Ananthanarayanan13]. State can also be migrated among operators or nodes in a cluster [Ding16, Feng11]. State can be exposed to programmers for easier use [Fernandez13, Wu15]. However, managing state incurs significant overhead, including processing latency and recovery time. Therefore, efficient methods for handling state must be devised to cover all aspects of state management and exploitation. Unfortunately, most of the major processing frameworks do not satisfy all of these requirements.

Reducing the overhead of state management is another challenging problem. One approach employs setting the appropriate intervals among state checkpoints. This setting can have a significant influence on the execution time [Sayed14]. In fact, the problem of state checkpoint placement (i.e., where to place checkpoints to have the fastest running time) is NP-complete [Bouguerra13]. Therefore, we need a novel solution to reduce the complexity, achieve low processing latency, and enable fast recovery time. Unfortunately, today’s leading big data processing frameworks, such as Flink\(^1\) [Alexandrov14], Spark\(^2\) [Zaharia12], Storm\(^3\) [Toshniwal14], Trident\(^4\), and Samza\(^5\) have not addressed all of these issues. Therefore, we point out the promising research direction on adaptive checkpoints to overcome this shortcoming at the end of this paper.

Figure 1 depicts our classification of state management. Throughout this paper, we explain each branch of this classification in more details. Using the intuitive illustration similar to [Hirzel14], the simple figure at the beginning of each section visualizes the main idea of the methods mentioned in that section (from Sections 2 to 8).

The rest of this paper is organized as follows. In Sections 2 to 6, we present the applications of state, namely, stateful computation, fault tolerance, iterative computation, elasticity and load balancing, and integrative optimizations. In Sections 7 and 8, we show how to share and maintain state. In Section 9, we highlight the varying methods used to handle state, including storing, updating, migrating, purging, and exposing. In Section 10, we discuss the overhead and complexity of state management. In Section 11, we compare the current implementations of state in popular frameworks and their limitations. In Section 12, we point out interesting research directions and finally, in Section 13 we present our closing remarks.

2. State for Stateful Computation

Stream processing can be divided into two primary categories, namely, stateless and stateful computation. In the former case, stateless processing applications merely accept inputs and generate answers based only on the current data. As such, they do not need to save or accumulate anything for later use. While these are still helpful in practice (e.g., filtering, simple operations), more challenging streaming applications, such as aggregations over time windows, complex event processing, and transaction processing require interactions with input data received thus far, raising the need for stateful processing. In contrast to stateless computing, stateful computation requires us to save the state in persistent storage for subsequent use.

Most of the current popular frameworks (e.g., Flink, Spark, Storm, Trident) support stateful operators. All of them share the same characteristic in supporting stateful computing, where they store state somewhere (e.g., in the state backend) and then use it in combination with current input to produce results for complex operations. However, they differ in the way of state implementation. Earlier versions of Storm focused on stateless processing and required state management at the application level. Based on Storm, Trident provides an API for state management. Samza was designed to support large state management using a local database to enable persistence. Shifting from Spark’s batch-processing approach towards real-time processing, Spark Streaming can carry out state computation via DStream (i.e., a discretized stream). Finally, Flink considers state as a first class citizen to enable stateful applications. In Section 11, we present
the comparison of state implementations among these frameworks in more detail.

Adding state manually by programmers in bulk processing systems hinders the integration of state into their data-parallel programs. Reprocessing all the data is another option but it suffers from long processing time. In any cases, both ways still underestimate the role of state, limiting the opportunities to improve the performance. Based on this observation, Logothetis et al. [Logothetis10] take state as an explicit input into program and allows state to be easily stored and retrieved as new input arrives. The core component of this work is the stateful groupwise operator, called translate, that incorporates state into data-parallel processing. This incorporation provides several opportunities for minimizing data movement in the underlying system.

Developed from [Logothetis10] with the same scenario of stateful bulk computing, Logothetis et al. [Logothetis09] introduce a data indexing technique to enable stateful groupwise processing to access state randomly and thus avoid costly sequential scans. This indexing technique supports incremental stateful groupwise processing, allowing operations to incorporate data updates without recomputing from scratch.

To employ stateful data parallelism, Gedik et al. [Gedik14] focus on partitioned stateful operators (e.g., streaming aggregation, progressive sort, one-way join, and user-defined operators). Such operators keep state on a sub-stream basis, where sub-streams are defined by a partitioning key. In contrast to stateless operators in which tuples routing can be accomplished in a round-robin fashion, the partitioning must be performed by a hash function. This hash partitioning method always route tuples to a suitable parallel channel for partitioned stateful operators. Gedik et al. [Gedik14] develop an efficient partitioning function structure with good balance and cheap migration under a wide range of workload and application characteristics.

Recently, based on the approach of algorithmic skeleton, Matteis et al. [Matteis16] present four parallel patterns for window-based stateful operators. These patterns, known as window farming, key partitioning, pane farming, and window partitioning, can leverage the usage of state in stateful computation. The window farming pattern applies each computation (e.g., a function) to a window. The results of windows’ computation are independent from each other. The key partitioning pattern extends the window farming pattern with a constrained assignment policy. In this policy, the same worker processes the windows of the same substream sequentially, limiting the parallelism. The main idea of the pane farming pattern is to split each window into non-overlapping partitions, called panes. This fine-grained division increases throughput and decreases latency by sharing results of overlapping panes. Finally, the window partitioning pattern processes a window by multiple workers. Similar to pane farming, this pattern improves throughput and reduces latency. However, the latency reduction of this pattern depends on the number of workers while the pane farming pattern does not.

Besides stateful computation, state can also be used to achieve fault tolerance, thereby, allowing for quick recovery from varying types of failures. State is persisted in reliable storage and updated periodically. When a failure occurs, the system restores the state to another node, recovering the computation from the last checkpoint. Each state includes internal data, which is placed inside operators, and external data (i.e., via input and output queues). To checkpoint state, we therefore have to save all data to stable storage.

According to [Hwang05], there are three fault-tolerance mechanisms: passive standby, active standby, and upstream backup. In the passive standby, only the modified part of the state is copied to the backup periodically. In contrast to passive standby, active standby uses redundant execution, in which each backup server also receives and processes in parallel the same input data (from upstream servers) as its primary server. Finally, in upstream backup, each primary server keeps its output data while the backup remains inactive. If a primary server fails, the backup rebuilds the primary server’s state from scratch by processing tuples logged at upstream servers. Each method has its own advantages in terms of network utilization, recovery speed, recovery semantics, and the effect on system performance [Hwang05]. Most works choose to focus on passive standby (i.e., checkpointing) as the underlying fault-tolerant method mainly due to the observation that checkpointing can effectively address more workload and configurations than other options [Hwang07]. Because each unit can be restored in parallel, this method reduces the overall recovery overhead.

Orthogonal to the taxonomy of [Hwang05], this section classifies the fault-tolerant methods into three main categories. We base on how state was handled (i.e., independently, dependently, and incrementally) to classify these methods.

3.1 State for Independent Checkpoints

In the literature, there are two types of failures, namely, independent and correlated. The checkpoint techniques in this section assume that either failures are independent of one another or they occur at the same time.

Hwang et al. [Hwang07] introduce the concept of a maximal connected subgraph as an atomic unit (i.e., a high-availability or HA unit) for independent checkpointing. These units can be checkpointed onto different servers at different times because they have no interdependency with each other and thus avoid inconsistent backup checkpoints. Due to this independence, spreading out independent checkpoints to multiple servers can reduce the overhead of checkpointing. Comparedly, the work in [Kwon08] splits state into independent partitions. Therefore, they can checkpoint their states independently and still ensure consistency in case of failure. By splitting operator state into independent parts, the work in [Sebeou11] produces independent partial checkpoints of these parts. These checkpoints are performed asynchronously at different times. In this work, independent checkpoints are in the form of control tuples that are integrated with regular tuples at the operator’s output queue.

3.2 State for Correlated Checkpoints

In this subsection, we discuss correlated failure events. These generally occur whenever switches, routers, or electrical power fails and may involve a number of nodes failing simultaneously. Complementing the single node failure assumption of previous works, the works of [Chen09, Upadhyaya11, Wang12,
Koldehofe13, Hakkarinen13, and Su16 consider tolerating correlated failures, whenever failures occur in bursts.

Chen et al. [Chen09] consider consistency an important issue and thus choose to checkpoint the entire system. In their work, they use scalable coding strategies to survive simultaneous multiple node or link failures. Unlike the classical checkpoint/restart fault tolerance paradigm, the application in this framework is not aborted, keeps all of its surviving processes, and adapts itself to failures. They introduce several checkpoint encoding algorithms to improve the scalability in which the overhead to survive k failures in p processes does not increase as the number of processes p increases.

The innovation of Meteor Shower proposed in [Wang12] is the use of tokens for checkpointing. The source operators initiate the tokens that flow through the stream graph. When an operator receives these tokens, the system checkpoints the state of this operator. Meteor Shower comprises three new techniques: (1) source preservation to avoid the overhead of redundant tuple saving in prior schemes, (2) parallel, asynchronous checkpointing to enable operators to continue processing streams during checkpointing, and (3) application-aware checkpointing to learn the changing pattern of an operator’s state size and initiate checkpoints only when the state size is minimal.

Without using persistent checkpoints, Koldehofe et al. [Koldehofe13] propose a novel method to tolerate simultaneous operator failures. This work mainly depends on this observation: “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 observation allows the system to preserve the operator state in savepoints instead of checkpoints. Thus, the operator state only comprises the necessary information of the incoming streams and the relevant selection events. This proposed savepoint recovery system can: (1) identify an empty operator processing state, (2) capture and replicate savepoints and ensure the reproducibility of corresponding events, and (3) tolerate multiple simultaneous operator failures.

Similar to [Koldehofe13] on tolerating simultaneous failures, Hakkarinen et al. [Hakkarinen13] propose an N-level diskless checkpointing scheme to reduce the overhead of fault tolerance. Layering the diskless checkpointing can tolerate failures of maximum N processes and reduce the runtime significantly compared with a one-level scheme. This work also develops and verifies an analytical cost model for N-level diskless checkpointing. Finally, the empirical search calculates the optimal number of checkpoints and levels in case of expensive exhaustive search.

While the active fault tolerance approach requires extra resources, and the passive one suffers a long recovery latency, especially when a number of correlated nodes fail simultaneously, the work in [Su16] proposes a new fault tolerance framework, called Passive and Partially Active (PPA), to overcome the weakness of both approaches. In the PPA scheme, while they checkpoint the states of all tasks using the passive approach, they use the active approach for only a small selected set of tasks because there is limited resource to actively checkpoint all the tasks. Thus, this scheme provides very fast recovery for selected tasks that employ the active fault tolerance approach and tentative output for those tasks that exploit the passive one. 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.

Not restricted to a single fault-tolerance approach for all operators, Upadhyaya et al. [Upadhyaya11] apply different fault-tolerance techniques at different operators within a query plan. Employing such a variety of checkpointing techniques for a single query requires a cost-based optimization plan for fault-tolerance.

To implement this strategy, they develop FTOpt, a cost-based fault-tolerance optimizer that can automatically select the best strategy for each operator in a query plan. FTOpt aims at minimizing the expected running time with failures for the entire query. While this approach is similar to the PPA scheme where checkpointing is not restricted to a single method, it is better than PPA in terms of the result’s quality because FTOpt produces exact output rather than tentative one.

3.3 State for Incremental Checkpoints

Previous approaches for fault tolerance depend on periodic state checkpoints for failure recovery. These approaches suffer from two main drawbacks as discussed in [Carbone15]. First, they often interrupt the overall computation, slowing down the processing speed of data flows. Second, they greedily persist all tuples in transit along with the operation states, which results in larger state sizes than expected. In response to these drawbacks, instead of checkpointing the whole big state at once, some works [Wang07, Sebepou11, Wu15] consider incremental state checkpoint, where only the small changes of state are persisted, considerably reducing checkpointing overhead. In other words, these methods capture only the delta in the state (i.e., what has changed since the last checkpoint), resulting in smaller state sizes.

Rather than taking a checkpoint of the entire state, the work in [Sebepou11] splits the state into smaller independent parts and then produces partial checkpoints of these parts independently. They introduce the continuous eventual checkpointing (CEC) mechanism for fault-tolerance with minimal interruption of operator processing. In CEC, partial state checkpoints are taken when control tuples contain both window state and actual tuples. CEC continuously and incrementally updates these evolving states by adding partial checkpoints to them. In this way, CEC can take continuous incremental state checkpoints efficiently. Furthermore, this method can adjust the checkpoint interval to achieve a good balance between recovery time and running time.

Quite similar to the CEC’s idea on checkpointing intensity, Naksinehaboon et al. [Naksinehaboon08] propose a cost model of incremental checkpoint to find the optimal number of incremental checkpoints between two full checkpoints. With this optimal number, the incremental checkpoint method can achieve lower overhead than that of full checkpoint model. This idea is further developed for the case of the Weibull failure distribution in [Pau10]. Their experiments show that the waste time of the incremental checkpoint model is significantly smaller than that of the full checkpoint model.

Sharing the same idea of CEC in partitioning the state into smaller parts for partial checkpoints, Hwang et al. [Hwang07] propose the fine-grained checkpointing model that employs a similar divide-and-conquer strategy. Specifically, by dividing the query graph into several subgraphs, each subgraph can be assigned to a different backup server. In this delta-checkpointing technique, each server checkpoints only a small fragment of its query graph. Because this method distributes the backups to multiple servers, any change of state is incrementally
checkpointed to the backups to guarantee the consistency of state. In case of failure, the query fragments at multiple servers are collectively recovered in parallel, thereby, achieving fast failure recovery and suffering only a small run-time overhead.

Using the similar divide-and-conquer strategy of the fine-grained checkpointing model suggested in [Hwang07] and of CEC, Wu et al. [Wu15] propose ChronoStream, which splits states into a collection of fine-grained slice units. These units then can be selectively distributed and checkpointed into identified nodes, similar to the way of assigning subgraphs to backup servers in [Hwang07]. When a failure occurs, ChronoStream transparently rebuilds these distributed slice units with small overhead. ChronoStream differs from similar previous works [Hwang07, Sebepoul11] in the way they model the application-level internal states.

In order to exploit the similarity of access patterns among writes to memory in the iterative applications, Nicolae et al. [Nicolae13] propose the Adaptive Incremental Checkpointing (AIC) approach for iterative computations with scarce memory storage. Specifically, the rule “first-time writes to memory generate the same kind of interference as they did in past iterations” allows AIC to predict the future accesses to memory for the next iterations. This prediction leverages both current and past access pattern trends for flushing memory pages to stable storage in the optimal order. This asynchronous checkpointing approach is also perfectly suitable for computing environments with scarce memory resource. The reason is that we can dynamically adapt to the access pattern of the application and minimize the interference of the checkpointing process running in the background.

Because the I/O bandwidth to remote storage has a major influence on running time for checkpointing on large-scale systems, reducing the state size to use bandwidth more efficiently can improve the performance. Jangjaimon et al. [Jangjaimon13] propose adaptive incremental checkpointing (AIC) by employing multiple cores to perform multi-level checkpointing adaptively with delta compression. In this way, the incremental checkpoints can reduce the checkpointing file size considerably. This reduction in state size, in turn, helps to lower the overhead and to reduce the expected job turnaround time. They also introduce a new Markov model to predict the performance of multi-level concurrent checkpointing. When testing under six SPEC benchmarks, AIC shows a substantial reduction in the normalized expected turnaround time (by up to 47%) in comparison with checkpointing schemes that employ fixed checkpoint intervals.

To minimize the space requirements in dataflow execution engines, Carbone et al. [Carbone15] propose Asynchronous Barrier Snapshotting (ABS), a lightweight algorithm that can work on both acyclic and cyclic dataflows. On acyclic topologies, the stage barriers, which are injected into the sources by a coordinator, can trigger the snapshot of current state. So, ABS persists only operator states in acyclic dataflows. On the other hand, on the cyclic execution graphs, ABS stores only a minimal set of records on cyclic dataflows. In case of failure, ABS can reprocess these logged records to recover the system. Experiments show that ABS can achieve linear scalability and performs well with frequent state captures.

4. State for Iterative Processing

Graph mining and machine learning algorithms are iterative in nature (i.e., iterating some computations until reaching the termination condition). Unfortunately, they are ill-suited for paradigms, such as MapReduce [Dean08] since they incur a large overhead, given that every iteration is run as a separate job that rescans iteration-invariant data [Schelter13]. These shortcomings have led to the incorporation of iteration mechanisms into data-parallel processing systems [Ewen12, Ewen13]. Markl [Markl14] also emphasizes the importance of iterative computation on machine learning community in his vision and requires native support of state for iterative data analysis programs as one key design for future platforms. This section shows how state is used in such iterative processing.

There are two different kinds of iterations. The first one is bulk iterations in which each iterative step produces an entirely different partial solution from the result of previous steps. Representative examples in this category include machine learning algorithms such as Batch Gradient Descend [Weimer11] and Distributed Stochastic Gradient Descent [Zinkevich10]. The second one is incremental iterations in which each iteration’s result has only a small change from the previous iteration’s result. Among the elements of the partial solutions, they have an inter-dependent relationship in which changes on one element affect only a few other elements. For example, in the Connected Components algorithm, an update to a vertex directly impacts only its neighbors.

Since most existing dataflow systems treat incremental iterations as bulk counterparts, they are not suitable for many iterative algorithms due to poor performance. To overcome this, Ewen et al. [Ewen12] propose a method to integrate incremental iterations into parallel dataflow systems and their optimizers. This integration enables the utilization of the sparse computational dependencies intrinsic in many iterative algorithms. Rather than creating specialized systems, this integration method allows for expressing many analytical pipelines in an integrated manner, without deploying any orchestration framework. This method has been demonstrated in [Ewen13] to illustrate the process of implementing, compiling, optimizing, and executing iterative algorithms on Stratosphere [Alexandrov14].

In data mining and machine learning, many fixpoint iterative algorithms converge to the correct solution from a number of intermediate consistent states. Leveraging the robust and self-correcting capacity of these algorithms, Schelter et al. [Schelter13] propose an optimistic recovery mechanism using algorithmic compensations. In case of failures, while traditional fault tolerance techniques use the rollback mechanism to recover the checkpointed state, this method applies a user-defined compensate function to build such a consistent state. Without interrupting the data processing to checkpoint any state, this method provides optimal failure-free performance and outperforms the rollback recovery methods. This method is applicable to three important classes of problems (i.e., link analysis and centrality in networks, path problems in graphs, and matrix factorization) that require iterative data processing. This
recovery mechanism is demonstrated in [Dudoladov15] with Apache Flink to run various iterative graph algorithms (e.g., Connected Components, PageRank).

With the same objective of [Schelter13] in tolerating the faults in iterative graph processing on distributed dataflow systems, Xu et al. [Xu16] propose two new checkpointing concepts, namely head and tail checkpoints, to reduce the checkpointing cost and failure recovery time. The main idea of this work is to use the unblocking mechanism (i.e., transparently writing the checkpoints in the background without requiring program to pause) to avoid the overhead of blocking counterpart. By injecting the checkpoints directly into the dataflow, this method takes advantage of both low-latency execution without interrupting the pipelined processes and seamless integration to the existing systems. Furthermore, the local log files on each node can prevent the recomputation from scratch in case of failures, allowing for faster recovery (i.e., confined recovery). Their experiments show that head checkpointing and confined recovery outperform blocking checkpointing and complete recovery (i.e., reload the latest checkpointed state and apply a full recomputation) for iterative computations.

While two previous works [Schelter13, Xu16] speed up the iterative computations by avoiding the expensive rollback recovery, MRQL Streaming [Fegaras16a] improves the performance of iterative data processing by reducing the state size. This method applies two techniques (i.e., homomorphism and lineage tracking [Benjelloun96]) to make state smaller. Specifically, MRQL Streaming first converts the SQL-like query to an incremental distributed stream processing engine program automatically. It then derives incremental programs by holding a small state during the query evaluation process and by using a novel incremental evaluation technique that merges the current state and the latest batches of data. This method can handle many forms of queries on nested data collections, including iterative and nested queries, group-by with aggregation, and equi-joins. More importantly, Fegaras extends this method to efficiently support iterative algorithms, such as PageRank and k-means clustering. The prototype implementation of this framework running on top of Spark validates the effectiveness of this method.

5. State for Elasticity and Load Balancing

Streaming applications running continuously for a long time may encounter variations in the workload, or skew over time. These factors can overload certain computing nodes in the data center, causing severe reduction of performance. To stand with this, the system must provide mechanism to scale well and balance the load among computing nodes. Data-parallel computation frameworks achieve elasticity, a key property, by maintaining and migrating state while tasks are actively running. To migrate, the number of parallel channels is changed at runtime (e.g., nodes can be dynamically added or removed) to match the computing resource and workload availability which may fluctuate in an unpredictable manner. In particularly, in case of workload overloads, the states of overloaded nodes are repartitioned and moved to other nodes with lighter load. Similarly, when resources are scarce, the internal states of the involved tasks should be relocated respectively.

As the stateful operators have a major influence on the scalability of dataflows, scaling these operators requires partitioning them across shared-nothing platform. However, as the continuously evolving workload can cause load imbalance, Shah et al. [Shah03] propose Flux, a dataflow operator, to encapsulate adaptive state partitioning and dataflow routing. Being placed between producer consumer stages in the pipelined dataflows, Flux can transparently repartition stateful operators without interrupting the running pipeline. Flux provides two mechanisms to adapt to both short and long term imbalances. In the first case, Flux uses a buffer and a reordering mechanism to adjust the local imbalances. In the second case, Flux can detect imbalances across the whole cluster, and allow state repartitioning in lookup-based operators to deal with long-term imbalances. To handle different causes of imbalance, this work also proposes some policies to repartition state for continuous query operators.

By considering the internal state as a built-in component, ChronoStream [Wu15] supports flexible scalability in both vertical and horizontal elasticity (i.e., varying resources at a single node in a system, and computing nodes in a system, respectively). This allows ChronoStream to efficiently deal with workload fluctuation and dynamic resource reclamation. For horizontal elasticity, transparent workload re-allocation is achieved by 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 by mapping one OS-level 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 directly handles this table to reschedule the computation. At any time during the execution, the workload in each thread can be dynamically reorganized to balance the load at thread-level.

Skewed workloads can cause problems of imbalanced memory usage, computation and communication cost across parallel channels. To deal with such skews, Gedik et al. [Gedik14] develop partitioning functions to evenly distribute load among computing nodes. They point out a number of desirable properties that a partitioning function must meet. These important properties include: (1) balance properties (e.g., memory, processing, and communication balance), (2) structural properties (e.g., compactness and fast lookup), and (3) adaptation properties (e.g., fast computation and minimal migration). Experiments show that the proposed partitioning functions possess these desirable properties over a variety of workload and thus provide better load balance than that of uniform and consistent hashing. These functions are especially effective for workloads with large key domain. In this case, they provide good computation, communication, and memory load balance, while still ensuring the low overhead of migration in case of data skews.
6. State for Integrative Optimizations

Instead of serving only a single purpose as presented in previous sections, state can also be used for a combination of purposes. In previous works, the applications of state (e.g., fault tolerance, elasticity, load balance, etc.) are typically tackled separately and independently. In some senses, these problems are closely related to each other and thus optimizing them independently may result in suboptimal solutions. In response, some works exploit this benefit of using state for integrative optimizations. In this section, we investigate the ways in which state’s applications can be modeled as one integrated optimization problem.

To support both incrementalisation and iteration, McSherry et al. [McSherry12] propose a new computational model, called differential dataflow, which comprises both incremental and iterative computations. Extended from the batch-oriented models (e.g., MapReduce, DryadLINQ), this model efficiently supports both incremental updates to the inputs and arbitrarily nested fixed-point iteration. Rather than using the total order, changes to the collections are described in the partial order, allowing the collections to evolve without restarting to reflect the changes. Multi-dimensional lattice times are used to combine the incremental and nested iterative computations (a lattice time can be represented as a tuple of integers).

To satisfy the three requirements of high throughput of batch processing, low latency of stream processing, and the efficient iterative and incremental computations, existing applications have to count on multiple platforms. As a result, these multi-platforms bring inefficiency, complexity, and maintenance problem to users. Naiad [Murray13, McSherry13], a distributed system for dataflow programs, is developed to combine all these features into a single framework. Figure 2 shows the high-level concept of Naiad (the iterative processing is represented by the dashed rectangle). This figure shows that Naiad can support both iterative and interactive queries on data stream, generating fresh and consistent results that can be incrementally updated as new data arrive continuously. This work suggests a novel computational model, called timely dataflow, to boost the parallelism across various classes of algorithms. To describe the logical points in the computation, this timely dataflow improves the dataflow computation by introducing the concept of timestamps. This new concept is essential to support efficient and lightweight coordination mechanism with three features: (1) structured loops for feedback; (2) stateful dataflow vertices for records processing without global coordination; and (3) notifications for vertices when all records for a given round of input or loop iteration were received. The first two features support low-latency iterative and incremental computations, while the third one ensures the result’s consistency. Naiad’s low-level primitives can play the role of a basic platform for a large number of powerful high-level programming models, supporting a diversity of tasks (e.g., streaming data analysis, iterative machine learning, and interactive graph mining).

Figure 2. A Naiad application that supports real-time queries on continually updated data [McSherry13].

In addition to fault tolerance, ChronoStream proposed in [Wu15] can support scalability and elasticity as well. By separating application-level computation parallelism from OS-level execution concurrency, they design a low-latency stream processing system with transparent workload reconfiguration in a unified model. With this core idea of level detaching, ChronoStream can achieve transparent elasticity, fault tolerance, and high availability without sacrificing performance. The slice-reconstruction approach in ChronoStream is similar to the state-migration approach in SLEEP [Fernandez13]. Furthermore, ChronoStream and [Fernandez14] both support dynamic reconfiguration at runtime. However, the state repartitioning approach in SLEEP and [Fernandez14] can incur high cost in state migration.

As presented in Section 5, to address both problems of load balancing and operator migration, Gedik et al. [Gedik14] introduce the partitioning function to achieve good load balance (auto-fission) and small migration cost. This partitioning function structure is a hybrid between a consistent hash and an explicit mapping. As a result, this flexible compact hash function is able to evenly balance the workload with high skew. Furthermore, they develop the construction algorithms and two metrics to build and assess the partitioning functions to see whether these functions can achieve good balance and cheap migration. More precisely, they define the load imbalance as the ratio of the difference between the maximum and minimum loads to the maximum allowable load difference. Items in the partially constructed partitioning function have their migration costs normalized using the ideal migration cost. The utility function that combines these two metrics (i.e., the relative imbalance and the relative migration cost) is used to assign the elements to parallel channels.

Similar to [Fernandez13] in integrating scalability and fault tolerance in stream processing based on operator state management, the work in [Madsen15] utilizes the checkpoints that are already backed up for failure recovery to efficiently improve the dynamic load migrations. They formally define a checkpoint allocation problem with some constraints. Then they provide a practical algorithm so as to reuse the checkpoints for effective load migration at the maximum possible benefit. In particular, if the skewed workload at key groups A increases, the system has to...
transfer many checkpoints for key groups in A to the slightly loaded nodes in advance to quickly react to this fluctuation. To increase the probability of this availability, the checkpoints of the key groups in A must be allocated to the nodes having key groups that are “as negatively as possible correlated with the key groups of A”. Due to this relationship between fault tolerance and migration, checkpoint can be viewed as the proactive load-balancing.

To arrive at low-latency processing, optimized resource usage, and minimized communication cost, Madsen et al. [Madsen16] model three issues (i.e., load balancing, operator instance collocations and horizontal scaling) as a single integrated optimization problem. They address the integration problem of load balancing and horizontal scaling, giving a mixed-integer linear program (MILP) to arrive at a feasible solution. This model is suitable when the collocation of operator instances does not considerably affect the communication cost. By using a linear program solver, MILP balances the load better than existing heuristic approaches. Extending to the cases where the collocation of operator instances has a major influence on the system performance, they broaden MILP to another solution, called Autonomic Load Balancing with Integrated Collocation (ALBIC). This improved version can generate better operator instances’ collocation, load balance, and overhead than MILP. This is because ALBIC gradually improves the collocation at runtime while still satisfying the load balance constraint.

7. State Sharing

Hirzel et al. [Hirzel14] consider the idea of state sharing as one of the optimization techniques for stream processing. They examine a streaming application that continuously computes the statistics (e.g., the average prices of stocks) for different window times (e.g., both one hour and one day). However, if these operations differ only on the time granularity (e.g., hours versus days), then they can share the same aggregation window with each other. This sharing can allow for more efficient resource (e.g., memory) utilization among operations. Despite this benefit, sharing a state among processes can lead to some inherent problems, such as conflicting accesses, consistency or deadlocks. Therefore, Hirzel et al. [Hirzel14] point out the following three safety conditions in state sharing. The first one is visibility in which state must be visible and accessible to all operators. The second one is the prevention of race conditions in which state is immutable or synchronization among processes must be properly set. The last condition is the safe management of memory. Proper memory management for the shared state is necessary in order to prevent early reclamation (i.e., release) or uncontrollable expansion (i.e., leak) of memory.

In [Hirzel14], they divide the state sharing techniques into three variants, depending on the type of shared state. The most general form of state sharing in the literature is shared operator state [Brito08]. In this variant, the operators can have arbitrary complex state. The key challenges with this variant are the synchronization and memory management. Using shared memory and mutual-exclusion locks to deal with these challenges is the most popular method to solve the conflicts. However, when conflicts are rare, this method results in prohibitive restriction for concurrency. Therefore, another approach [Brito08] uses software transactional memory to manage shared data.

The second variant of state sharing is shared window [Gordon06, Arasu06]. In this variant, multiple consumers can use the same window. Window sharing is one of the simplest cases of state sharing [Gordon06]. Continuous query language implements windows by using non-shared arrays of pointers to reference to the shared data items. This model of many-to-one pointer reference can allow multiple windows and event queues [Arasu06] to access a single data item.

The final, also the least general, form of state sharing is shared queue [Sermulins05]. In this variant, the simultaneous access of both producer and consumer to a single element (i.e., the producer writes a new item into a queue while the consumer reads an old item at the same time) can lead to conflicts. To guarantee synchronization and avoid sacrificing concurrency, the queue must be able to buffer at least two data items. Turning the shared queue into local one and statically computing all offsets at compile time (instead of run time) can improve the performance of a shared queue [Sermulins05].

One way to efficiently handle a large amount of data in big data processing is reusing the computational results as much as possible. Another way is to reduce the data generated (e.g., by scientific experiments) as early as possible. To achieve this, Kuntschke et al. [Kuntschke05] try to avoid the redundant execution of operators, the transmission of unnecessary data, and the redundant transmission of data to the maximum extent. Sharing data streams during processing can therefore prevent the redundant transmission, saving the network bandwidth. Furthermore, sharing previously computed results and early filtering and aggregation (e.g., the combine function in MapReduce) can facilitate the removal of unnecessary parts of data. Based on this idea, Kuntschke et al. [Kuntschke05] propose a data stream sharing approach to address these issues. This approach includes two main optimization techniques: (1) in-network query processing to distribute and execute newly registered continuous queries and (2) multi-subscription optimization to enable the reuse and sharing of generated data streams.

To share data across operators using a less structured mechanism than point-to-point dataflows, Losa et al. [Losai2] propose CAPSULE (termed for shared state). CAPSULE allows for the realization of shared variables by using a data structure at the language level. In addition to supporting efficient state sharing in distributed stream processing systems, CAPSULE provides the following features: (1) Custom Code Generation to generate shared variable servers that fit to the specific scenario based on runtime information and configuration parameters, (2) Composability to achieve suitable levels of scalability, performance and fault-tolerance by using shared variable servers, and (3) Extensibility to support more protocols, transports, caching mechanisms, etc. with simple interfaces.

Meehan et al. [Meehan15] design S-Store to maintain correctness and ACID (Atomicity, Consistency, Isolation, and Durability) guarantees which are essential to handle the shared mutable state. The employment of this shared state can achieve high throughput and consistency in both streaming and transaction processing. In this hybrid (i.e., streaming and transaction) processing, the way to share the state of a window differs from that of other stored state (i.e., privately shared with other transaction executions). More precisely, the proper coordination...
and sharing among consecutive executions of a window’s state is support each other’s incremental maintenance, reducing overall low latency with correctness in stream processing and very high view maintenance cost. Similarly, Koch et al. [Koch14] provide throughput with ACID guarantees in transaction processing. a complete description, and a thorough experimental performance Tatbul et al. [Tatbul15] further explore correctness criteria, evaluation of the DBToaster system by using the theory of rings. including ACID guarantees, ordered execution guarantees, and DBToaster can retain materialized views continuously fresh as exactly-once processing guarantees. To support these three data changes frequently. This is achieved by two techniques (i.e., complementary correctness guarantees, S-Store provides efficient aggressive compilation and original recursive finite differencing). scheduling and recovery mechanisms. Although Naiad, SEEP and Samza all view state as mutable, they do not inherently support transactional access to shared state. Therefore, S-Store’s consistency guarantees are better than those of these systems.

8. Incremental State Maintenance

As pointed out earlier, previous works (e.g., [Naksinehhaboon08], [Sebeou11]) propose incremental checkpoints to reduce the overhead of full checkpoints. Shari ng this idea and considering state as view, some works [Koch10,14,16, Nikolic14,16, Liu16, Naiad, and Fegaras16] try to maintain the state incrementally to deal with frequent data updates and to avoid the expensive full updates on state. The main idea is to generate the delta values that can update the persisted state more efficiently than recomputing from scratch when there is only a small change in the input.

McSherry et al. [McSherry13] present differential computation to generalize existing techniques for incremental computation with continuously changing input data. This method differs from traditional incremental computations in supporting arbitrarily nested iterative computations. Similar to Naiad system, the key innovations of differential computation come from two factors: (1) first, the changes of state follow a partially ordered set of versions rather than a totally ordered sequence, conforming to the incremental computation; and (2) second, an indexed datastructure maintains a set of updates required to rebuild the state. This second feature is different from the other incremental systems in which updates are usually discarded after being merged to the current version of the state.

Using algebra to explore the incremental view maintenance (IVM) problem, Koch [Koch10] extends the ring of databases structure to form a powerful aggregate query calculus. Inherited the key properties of rings, such as distributivity and the existence of an additive inverse, this calculus is 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 query evaluation. The multi-layered IVM scheme can maintain a view using a hierarchy of auxiliary materialized views. This hierarchy can simply refresh all views when there is an update. This theory lays a foundation of incremental state maintenance for subsequent research [Ahmad12, Koch14, Nikolic14,16].

Based on the theoretical foundation of IVM in [Koch10], Ahmad et al. [Ahmad12] introduce viewlet transforms, a recursive finite differencing technique to combine current and historical data. The viewlet transform materializes a query and its views that
re-plan the optimal execution in case of unexpected performance changes. Given a continuously changing input, the streaming system repeatedly updates its outputs by recalculating the optimal plan incrementally. This incremental re-optimizer requires the maintenance of state (i.e., the optimizer memoization table) throughout the runs to re-optimize. Furthermore, determining which plans to prune from this state is important to re-plan the execution. To enable such re-pruning ability, they define a semantic for tracking and re-computing state, using a declarative specification. This declarative method allows them to identify state-pruning strategies that are “agnostic to the order of control and data flow during plan enumeration”. This work proposes three new state-pruning strategies, namely aggregate selection with tuple source suppression, reference counting, and recursive bounding. Experiments show that each state-pruning technique has a different and meaningful way to contribute to the overall performance of incremental re-optimization.

9. Operations on State

After surveying the applications of state, we now consider the handling (i.e., storing, updating, purging, migrating, and exposing) of state in this section.

9.1 Store

State size is the key factor to decide where to store state. Some works [Zhang15, Ren16] store state in memory due to the small state size. However, for big state size, other works [Liu06, Kwon08, Nicolae13] persist state on disk, incurring more overhead.

With small state size, storing state in memory can accelerate processing [Zhang15]. However, continuous queries running in a very long time, in particularly complex queries with huge operator states such as multi-joins, can be extremely memory intensive during runtime. When system resources are scarce and therefore cannot meet the demand of processing the huge workload at runtime, techniques such as load shedding [Tu06] are useful to remove some workloads from the system, sacrificing the accuracy for performance. The system can get rid of these moved workloads permanently or reprocess them later when the computing resources are sufficient [Liu06]. However, many cases require having accurate results for long-running queries even though the system may not have enough resources to keep up with the query workload during runtime. Therefore, techniques such as load shedding [Tu06] are not applicable in these cases anymore. To address such problem of scarce memory resource and still guarantee the correctness of results, some works (e.g., XJoin [Urhan00], Hash-Merge Join [Mokbel04] and MJoin [Viglas03]) flush the states stored in memory into disks temporarily when memory is full. By delaying the processing of states stored on disks (aka. state cleanup) until more resources become available, these solutions achieve both low-latency processing and completeness of results. However, these strategies of adaptive state spilling (i.e., pushing and cleaning processes) support only one single stateful operator. Consequently, these works fail to support memory intensive queries with multiple stateful operators which are common in data integration or data warehouse.

To overcome previous limitation of supporting multiple stateful operators, Liu et al. [Liu06] target multiple state intensive operators with interactions among operators’ data spilling in the same pipeline. They explore state spilling strategies to deal with complex queries by selectively pushing operator states into disks. To avoid memory overflow and increase query throughput, the strategy to select appropriate parts of the operator states to spill to disks at runtime is essential. Exploiting the interdependency among multiple operators when spilling states, this method performs more effectively than the strategies that do not take this interdependency into account. This work introduces some data spill strategies, ranging from operator level to partition level, to maximize query throughput in the condition of scarce memory resource. With equal handling of all data in one operator state, they propose the bottom-up state spilling method that belongs to the operator-level strategy. Employing input data with different characteristics is applicable in the more complex partition-level data spilling strategies. These strategies include the local output, the global output and the global output with penalty strategies. The implementations of these strategies in the D-CAPE system allow for the performance comparison among them. Experiments show that the global output strategy and the global output with penalty strategy perform better than the localized strategies. To ensure the completeness, they also propose some effective clean-up 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 completion of run-time execution phase. For queries having window constraints, interleaving the in-memory execution and the disk cleanup process is necessary at runtime. In this scenario, this work, however, still does not resolve some challenging issues, such as timing of spill, timing of clean-up, and selection of data to cleanup, opening new interesting research problem in this direction.

To save more memory resources for normal stream processing, SGuard [Kwon08] stores states into a distributed and replicated file system (DFS) such as Google File System (GFS) or Hadoop Distributed File System (HDFS). These file systems optimize reading and writing of large data volumes. Due to simultaneous checkpoints of multiple nodes, resolving resource conflict is a critical requirement and therefore SGuard adds a scheduler to the DFS to meet this requirement. By coordinating large batches of write requests, this scheduler not only reduces individual checkpoint times but also provides good resource utilization in general. Similar to previous methods [Hwang05] on rollback recovery, SGuard periodically checkpoints the state and recovers failed nodes from their last checkpoints. Unlike previous approaches, however, SGuard performs checkpoints asynchronously: SGuard uses a new memory management middleware to checkpoint the state of an operator while the system keeps executing. Consequently, this asynchronous mechanism can prevent the potential interruption and overhead caused by the checkpointing process.

Sharing the same idea of [Liu06] in efficiently flushing the state from memory to disk, Nicolae et al. [Nicolae13] propose AI-Ckpt, an asynchronous checkpointing runtime, for adaptive incremental checkpointing. AI-Ckpt exploits the trends of both current and past access patterns to generate the optimal order of flushing memory pages to stable storage. In iterative applications, they observe that there is a similarity of the writing patterns to memory among iterations. Based on this observation, AI-Ckpt proposes an optimization that leverages these access patterns to flush modified pages with minimum overhead. They first present five design principles: (1) managing protected memory areas properly; (2) tracking modified pages to capture the access patterns and checkpoint increments asynchronously, (3) using bounded copy-on-write to avoid unnecessary waiting; (4) conforming page flushing to access patterns; and (5) optimizing this flushing process by the access pattern history. Then, they materialize these design principles through a number of
algorithmic descriptions. Their experiments show that optimal flushing can considerably improve performance, especially for applications that naturally exhibit iteration (e.g., machine learning or graph algorithms) and repetitive access patterns. However, this method uses only the access order to flush the pages, omitting the temporal aspect. Thus, a promising research direction is to integrate timestamp into this order to further optimize the flushing process.

Ren et al. [Ren16] present an asynchronous technique to capture database snapshots without using any physical consistent checkpoint. This technique, known as Checkpointing Asynchronously using Logical Consistency (CALC), avoids the obvious overhead incurred by other snapshotting schemes. The key idea of this technique is to create a virtual point of consistency in the sequence of transaction commits. A checkpoint includes only the state changes caused by transactions that commit before this virtual point. In other words, this checkpoint does not register the following transactions’ modifications. Moreover, they suggest pCALC, considered as another partial version of CALC, to further improve the checkpointing performance. To improve performance, pCALC takes only partial state checkpoints that have changed since the last checkpoint. Then merging these partial checkpoints in background can reduce the recovery cost in case of failures. Testing across a wide range of transactional workloads, their experiments prove that CALC and pCALC achieve smaller cost and lower memory usage than other checkpointing systems.

Not restricted to local memories or disks, Ananthanarayanan et al. [Ananthanarayanan13] store large states even in across geographically distant locations. This work presents Photon, a geographically distributed system for joining multiple unordered data streams to ensure high scalability, low latency, and exactly-once semantics. Without human involvement, Photon can automatically solve the problem of infrastructure degradation and datacenter-level outages. The critical state, stored in the IdRegistry and shared between the workers, comprises a set of event identifiers that have already been joined in the last N days. Balancing the costs of storage and dropping events determines the value of N. The synchronous duplication of IdRegistry to multiple datacenters in different geographical regions ensures that its service is always accessible even when one or more datacenters face problems.

9.2 Update

For continuous bulk processing (CBP), Logothetis et al. [Logothetis10] update only a fragment of the state to optimize system performance. Similar to [Logothetis10], Fegaras [Fegaras16a] updates the state incrementally. By using homomorphism, every time the system produces a small delta result based on the subset of data (∆S), merging the previous value of state and the current delta result can incrementally produce the new value of state (i.e., state ← state ⊕ h(∆S)). Figure 3 illustrates this update process with two streaming sources. Fegaras implements a new physical operator, called Incr, which is a stateful operator to efficiently update this state.

Fernandez et al. [Fernandez14] consider fine-grained update to examine how update can affect the throughput and latency. They compare this update granularity among several systems to check which one is capable of supporting the fine-grained update. To do this, they vary the window’s size because the granularity of updates to the state depends on this size (i.e., the smaller window size leads to the less batching, and thus the finer granularity).

Their experiments show that Naiad [Murray13] can achieve low latency when using small batch size (1000 messages) and high throughput for large batch size (20000 messages). This result is due to the Naiad’s capability to configure the batch size independently of the window size. The stateful dataflow graph (SDG) [Fernandez14] can handle all window sizes and achieve higher throughput than Naiad. The overhead of micro-batching is substantial in other deployments: Streaming Spark’s throughput is equivalent to that of SDG, but its smallest sustainable window size is 250 ms. Passing this limit, its throughput collapses.

Low et al. [Low12] introduce the GraphLab framework for graph-parallel computation to ensure data consistency when updating the program state. GraphLab represents the modifiable program state as a directed graph, called data graph. This state includes mutable user-defined data and sparse computational dependencies. To update this state, the update function transforms data on the data graph into small overlapping contexts (called scopes). To preserve data consistency, GraphLab presents three consistency models (i.e., full, edge and vertex consistency model) for update functions. 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 update functions that perform simultaneously do not overlap. However, this consistency model limits the 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 vertex and adjacent edges 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.

Last but not least, several big data processing frameworks [Alexandrov14, Zaharia12, Toshniwal14] explore and compare the semantic update of state. Basically, there are three kinds of semantic guarantees (i.e., at-least-once, at-most-once, and exactly-once) to assess the correctness of state. Systems with at-least-once semantic fully process every tuple but they cannot guarantee duplications. In the at-most-once semantic, systems either do not process tuple at all or execute exactly once. Unlike the at-least-once semantic, this semantic does not require the detection of duplicate tuples. Finally, systems with exactly-once semantic process tuple once and only once, providing the strongest guarantee. Section 11 compares these semantic guarantees among popular big data frameworks.

9.3 Purge

When systems no longer need a specific piece of data for subsequent operations, state can purge that data (e.g., a buffer state removes expired tuples). This section presents efficient ways to purge state.
Ding et al. [Ding04] propose several join algorithms to efficiently purge state by leveraging punctuation on data attributes. More precisely, they introduce a stream join operator, called $P\text{Join}$, to delete data which is no longer useful from the state using punctuations. This removal frees the memory for other operations and accelerates the probing process in join operations. They then equip $P\text{Join}$ with two strategies (i.e., eager and lazy) to purge states. *Eager purge* immediately purges states whenever obtaining a punctuation in order to minimize the memory overhead and efficiently probe the state of join operation. Eager purge is not applicable in cases of frequent arrival of punctuations because the cost of probing is less than that of scanning the join state. They therefore propose *lazy (batch) purge* that starts purging only when the number of new punctuations since the last purge reaches 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 equals 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.

Continuous joining of multiple streams requires the join operator to maintain states that can grow infinitely and eventually exceed the memory capacity. Tucker et al. [Tucker03] propose punctuation semantics to address this problem. More precisely, systems inject punctuations to explicitly mark the end of a data’s subset. This way can safely enable purging of the stored data that will not affect any new results. This work considers a continuous join query (CJQ) as unsafe (and thus prevented from running) if it requires an infinite storage. Li et al. [Li06] introduce the punctuation graph structure to analyze the query’s safety (i.e., checking whether a CJQ satisfies the safety condition under a given set of punctuation schemes). This punctuation graph can verify the safety of a query in polynomial time. To do so, they first formally define the condition for the purgeability of a join operator. Then, they classify the safety checking of a CJQ into two categories: data purgeability and punctuation purgeability. This work considers a punctuation as a special tuple and therefore makes punctuation purging possible as well. Finally, this work also proposes the chained purge strategy to generalize the binary join case to the multi-way join case.

Stream processing systems often impose an order on data streams during execution to purge state from stateful operators. This processing requires the preservation of order and thus suffers from significant overhead. Li et al. [Li08] design a new architecture for out-of-order processing (OOP) to avoid order preservation. OOP uses explicit *stream progress indicators*, such as punctuation or heartbeat. Previous works [Tucker03, Ding04, Li06] use punctuation as a general mechanism to purge state from stateful operators. This work presents a new type of punctuation called joint punctuation, to reduce the delay in join operators.

### 9.4 Migrate

Dynamic state migration is a crucial operation in stream processing systems, involving the efficient transition from one place to another without losing the semantics. For stateless operators, existing migration approaches usually implement the *pause-drain-resume* strategy that stops accepting new data, and removes old data. This strategy may create a deadlock during the migration process: “the migration is waiting 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” [Zhu04]. Besides the deadlock, this strategy does not address the problem of migration for stateful operators. Stateful operator’s state migration is important for operations such as joins, aggregates, or the addition/removal of nodes. As a result, this can deal with fluctuations in workloads, data characteristics, and resource availabilities. State migration involves two main problems [Ding16]: (1) how to migrate (i.e., the mechanism to reduce the overhead caused by synchronization and result delay during migration); and (2) what to migrate (i.e., the decision to choose the optimal task assignment that minimizes migration costs).

Concerning these problems, Zhu et al. [Zhu04] introduce a dynamic plan migration for continuous query plans that contain stateful operators. More precisely, they propose two strategies (i.e., *moving state* and *parallel track*) to exploit reusability and parallelism. This exploitation is useful for seamlessly migrating among continuous join query plans while still ensuring the correctness of the query’s results. The moving state strategy comprises three important steps: state matching, state moving, and state recomputing. The moving state strategy first stops the query plan execution and purges tuples inside intermediate queues. Then mapping and moving all tuples in the states of the old query plan to their new corresponding location in the new plan is necessary to resume the execution of the query plan. Rather than immediately moving tuples to the new query plan and purging the old one, the parallel track strategy gradually migrates state by plugging in the new query plan and executing both query plans at the same time. So this strategy keeps producing output tuples even during the migration process. In case of sufficient computing resources, the moving state strategy usually finishes the migration process faster and performs better than the parallel track strategy. In contrast, when resource is scarce, the parallel track strategy has less intermediate results and better output rate during the migration process. So, the choice of migration strategy depends on the system resource’s availability.

While the work in [Zhu04] proposes dynamic plan migration to relocate states among operators, Ding et al. [Ding16] migrate states among nodes within an operator. Also, even though SEEP [Fernandez13] and StreamCloud [Gulisano12] propose the idea of operator states migration before [Ding16], they provide few details on the implementation of these systems. Therefore, Ding et al. [Ding16] overcome this shortcoming by proposing algorithms that support live and progressive migrations. Consequently, the result delay caused by migration process is negligible and controllable. Furthermore, they propose the task assignment algorithm to compute the optimal assignment, addressing two optimizations: (1) minimizing the migration costs, and (2) balancing the workloads. Moreover, to predict the cost of future migrations, they suggest another algorithm based on statistics of the past workloads. This work criticizes ChronoStream [Wu15], which “claims to have achieved migration with zero service disruption”, by pointing out that the synchronization issues can affect the correctness of the result in their migration implementation. To overcome this, the proposed mechanism does not migrate and execute tasks at the same time. It also ensures to send all misrouted tuples to their correct destinations. To control the result delay caused by migration, they perform migrations progressively. Particularly, they perform multiple mini-migrations, in which the number of migrated tasks must be smaller than a given threshold.

For migration, determining the placement locations (i.e., selection of the physical node to manage the operator [Pietzuch06]) is another challenging problem due to the variations of network and node conditions over time and the interactions
among streams. This challenge opens interesting research venues for migrations of operators’ state. In this issue, the local placement optimizer examines the current placement of local operators and launches the migrations of operators when the saving in network usage is higher than a pre-defined value. This minimum migration threshold depends on the cost of operator migrations and keeps an operator remain at its current location if the minimum migration threshold cannot be met.

To efficiently determine the placement location and to reduce the network utilization, Pietzuch et al. [Pietzuch06] introduce stream-based overlay network (SBON). 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 the existing placements and trigger the operator migrations to new hosts if necessary. SBON uses the network usage metric for operator placement to balance the application delay and the consumed network bandwidth. SBON runs two components: (1) the data stream processing system (DSPS) is responsible for operator instantiation, migration, and state, and also data transport and (2) the SBON layer monitors local performance, manages the cost space, and triggers migrations. They evaluate SBON with a deployment on PlanetLab and show that their placement optimization technique increases network utilization, incurs low latency, and supports dynamic optimization efficiently (e.g., reduced network usage by 17% and reduced re-use by 21%).

Similar to [Pietzuch06] in network-aware optimization, Ottenwalder et al. [Ottenwalder13] propose MigCEP by planning the migration in advance to minimize the network usage. Specifically, they introduce an algorithm to generate the Migration Plan, a probabilistic data structure for describing future targets and times for the migration. They propose another migration algorithm to minimize both network usage and latency. Enabling the migration coordination of multiple operators that may require the same mutable state can further improve the network utilization.

With high computation load, stateful stream processing systems that monitor, migrate, and replicate a large number of states to backups for reliability impose expensive overhead. Feng et al. [Feng11] present two novel methods (i.e., randomized replication representation and overloaded replication scheme) to address this issue of expensive overhead. In the first method, a hashing structure, called Multilevel Counting Bloom Filter (MLCBF), can enable the replication with low resource consumption to improve the performance of state transfers for replicated operators. In addition, they use an adaptive scheme, called dynamic lazy insertion, to reduce the influence of replication on overflowing system and increase the cluster’s throughput. Experiments show that MLCBF reduces network and memory usages of replication by more than 90% for URL categorization. Moreover, MLCBF is quite simple and pragmatic in terms of implementation and maintenance.

9.5 Expose

Exposing state to processing systems reveals several advantages: (1) it allows systems to quickly recover checkpointed states in cases of failures; (2) it enables systems to efficiently reallocate stateful operators across a set of new partitioned operators to support scale out [Fernandez13]; and (3) it permits some integrative optimizations as pointed out in section 6. Due to these advantages of exposing state, some works [Logothetis10, Fernandez13,14, 16, Wu15] target on externalizing state.

Considering state as the first-class citizen, Logothetis et al. [Logothetis10] propose a flexible, groupwise processing operator that takes state as an obvious input together with the main inputs. To handle state explicitly, they develop a set of flexible dataflow primitives to perform large-scale data analytics and graph mining. The translate operator can access state directly through a familiar and powerful groupwise processing abstraction, allowing users to easily store and access state during execution. This general abstraction allows for the operations of inserts, updates, and removals on state. This work plans to develop a compiler to translate an upper-layer language into processing dataflows, supporting easier use of state.

To expose state, the main idea in [Fernandez13] is to externalize internal operator state so that stream processing systems can perform explicit operator state management. This work classifies 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 under the forms of efficient data structures. Systems can translate processing state externally to key/value pairs when necessary. They model buffer state as the output buffer of operator to store 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 correct 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 the keys to a partitioned downstream operator. To manipulate these three states, they define a set of primitives for state management to allow systems to checkpoint, backup, restore, and partition operator state. These primitives are only the minimum set required for scaling out and fault tolerance. It is possible to build more state primitives to cover a wider range of previously mentioned functionality. For example, abundant resources allow for merging the states of two operators [Gulisano12] for scaling in. To deal with large state sizes, spilling state [Li06] to disk frees memory for useful computations. Persisting part of the operator state into external storage enables the combination of data-at-rest and data-in-motion [Ahmad12].

Developed from [Fernandez13], Fernandez et al. [Fernandez14] make state explicit for imperative big data processing by using stateful dataflow graph (SDG). Requiring fine-grained access to large state, the imperative machine learning algorithms pose challenges to big data frameworks. SDGs address these challenges by efficiently translating imperative programs with large distributed state into a dataflow representation, enabling low-latency iterative computation. By explicitly differentiating data from state, SDGs use state elements (SEs), encapsulating the computation’s state to enable this translation. They use efficient data structures, such as hash tables or indexed sparse matrices, to implement SEs. Figure 4 illustrates two

![Figure 4. Types of distributed state in SDGs [Fernandez14].](image-url)
distributed ways to represent a SE. In the first way, a partitioned SE splits 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 update. Partitioning state across nodes can support scalability if it is possible to fully deploy computation in parallel. On the contrary, if it is not the case, partial SE deploys independent computations. Application semantics can settle these computations. The important point of SDGs is that their tasks can directly access the distributed mutable state, allowing SDGs 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 type of access to partial SEs according to the semantics of machine learning algorithms. Fernandez et al. [Fernandez16] demonstrate this work by developing the JAVA2SDG compiler to translate the annotated Java programs to SDGs. As an example, they demonstrate the translation of machine learning algorithms in Java, including collaborative filtering and logistic regression, into SDGs and how to run on a cluster of machines.

To easily manipulate state, ChronoStream [Wu15] distinguishes states in each operator in two forms: computation state and configuration state. Computation state is a set of application-level data structures that systems can directly access and handle, conforming to the user-defined processing logic. Without loss of generality, ChronoStream represents each user-defined data structure as a key-value store. The get, set, or delete operations can correspondingly reflect any change to this data structure. Systems hash-partition the computation state, maintained in an operator, in an aggressive manner into an array of fine-grained computation slices. Equally distributing these slices to multiple resource containers enables load balancing. Every subset of input data corresponds to each independent slice that generates corresponding output streams. Configuration state is a collection of states at container level used to maintain the runtime parameters. The configuration state associates with each container and its contents differ among containers. The configuration state associated with each resource container comprises three components: (1) an input routing table to deliver the input events to corresponding slices; (2) an output routing table to direct output events to the associated resource container in the downstream operator; and (3) a thread-control table to keep the information about the schedule of the OS-level threads to support the computation of the upper-layer slices. Generally, configuration state plays a role as the intermediate connection between application-level parallelism and local OS-level multithreads.

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

10. State Management Overhead and Complexity
State management approaches incur overhead, coming from setting the checkpointing interval, finding optimal assignment, and checkpoint placement. We therefore investigate these factors in this section.

10.1 Impact of State Checkpoint Interval
Checkpointing regularly can help to resume the computation quickly when failures occur. Systems, however, waste a large amount of time and resource for checkpointing instead of performing useful computation. In contrast, fewer checkpoints can lead to longer recovery time in case of failures. A large body of works [Naksinehaboon08, Fernandez13, Sayed14] focuses on finding the optimal checkpoint interval (i.e. the distance between two consecutive checkpoints), trading off between waste time for checkpointing and useful time for computations.

Naksinehaboon et al. [Naksinehaboon08] study the optimal placement of checkpoints to minimize the total waste time of rollback recovery and checkpoint overhead. They propose a checkpoint frequency function that takes the failure probability distribution as an important parameter. The optimal checkpoint interval can then be derived from this function.

In [Fernandez13], the proposed state management approach using periodic checkpointing imposes an overhead. Then, by measuring the processing latency, they show that using different checkpointing intervals can generate different latencies. This method reveals a trade-off in which larger intervals lead to the lower influence on data processing, but incur longer recovery time in the event of failures. They suggest setting the checkpointing interval based on the estimated failure rate and the performance requirements of a query. This conclusion complies with the work of [Naksinehaboon08].

Similar to this conclusion, Sayed et al. [Sayed14] evaluate multiple methods to see the impact of checkpointing interval to the wasted work. They first criticize the ad-hoc rules of periodic checkpoints (e.g., checkpoint once every half an hour). Then, they find that the model of Young [Young74] achieves near optimal performance, and is applicable in practice. They further investigate more advanced methods that can dynamically change the checkpoint interval. They show that these methods significantly improve the Young’s model for only a small subset of systems.

10.2 Complexity of Optimal State Placement
How to effectively place the checkpoints is another complicated problem. Indeed, some works [Bouguerra13, Robert12] formally prove that this problem is NP-complete. They therefore propose some algorithms to solve this problem in polynomial time.

Robert et al. [Robert12] focus on the complexity of scheduling computational workflows with failures following the exponential distribution. In case of such failures, they use the rollback and recovery method to resume the computation since the
last checkpoint. With the objective of optimizing the expected processing time, deciding in which order to execute several independent tasks, and whether to checkpoint or not when each task finishes are difficult combinatorial problems. They prove that this scheduling problem is strongly NP-complete. They propose a dynamic programming algorithm that can run in polynomial time for this problem.

Not restricted to exponentially distributed failures as the previous work of [Robert12], Bouguerra et al. [Bouguerra13] study the computational complexity of the checkpoints scheduling problem with failures following general distributions. Taking an arbitrary failures distribution function into account, the costs among checkpoints differ and the parallel computation of data blocks has different durations. Particularly, they propose a new complexity analysis to deeply exploit the relations among the probabilistic failure model, the checkpoint cost, and the computational model. Furthermore, they introduce a new mathematical formulation to optimize the checkpoint scheduling in parallel applications. The objective of this formulation is to minimize the waste time when the checkpoint costs and recovery time differ. In particular, they prove that the checkpoint scheduling problem is NP-complete even in the simple case of a uniform failure distribution. To solve this problem, they propose a dynamic programming algorithm to determine the optimal checkpointing times in all variants of the problem.

### 10.3 Complexity of Optimal State Assignment

Ding et al. [Ding16] target the problem of finding the optimal task assignment that minimizes the costs for state migration and satisfies load balancing condition. The migration cost is the total size of operator states transferred among nodes. They assume that the output of partitioning function \( f \) to input record \( r \) is an integer \( f(r) \) with \( 1 \leq f(r) \leq m \). Each node \( N_i \) \((1\leq i \leq n)\) is assigned a continuous interval \( I_i=[lb_i,ub_i],1 \leq lb_i \leq ub_i \leq n\), called the task interval of \( N_i \). Given a threshold \(\tau\), a task assignment is considered to be load balancing if (and only if) the workload \( W_i \) for every node \(N_i\) satisfies this condition \(W_i \leq (1+\tau)W/n\). In other words, this condition means that the workload of each node is not too high 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 \( n \) task intervals, and then assigning these intervals to \( n \) nodes. How to partition the task efficiently is a challenging problem because it has an exponential search space.

By dividing this problem into sub-problems and by enumerating and solving each possible sub-problems, they propose a basic solution, called Simple_SSM, which has \(O(m^n.n^n)\) possible sub-problems. This solution incurs a space complexity of \(O(m^n.n^n)\) and time complexity of \(O(m^n.n^n)\). To reduce this complexity, they propose an advanced solution by using a series of optimizations. Taking advantage of these optimizations, the proposed solutions gradually improve the space and time complexity over time. Finally, the best solution uses only \(O(m.n)\) space and \(O(m^n.n^n)\) time, a significant reduction in compared with the basic solution.

### 11. State Implementations and Limitations

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

Storm targets only stateless processing and thus implements state management at the application level to support fault-tolerance and scalability in stateful applications. In other words, Storm does not natively support state management. To overcome this limitation, Trident, a high-level abstraction layer for Storm, is proposed to handle state. Built on top of Storm, Trident is a micro-batching system, adding state management to Storm. Trident with its own API for fault tolerance provides exactly-once semantics guarantee. More precisely, inheriting the acknowledgement mechanism of Storm, Trident is capable of preventing data loss and ensuring that every tuple is processed only once. Currently, there are two alternatives of state management supported in Storm. The first one only stores the sequence number of the last batch and the current state. This method may introduce the blocking problem. The second one overcomes this shortcoming but incurs more overhead by also keeping the last-known state. To ensure correct semantics, it is vital to strictly maintain the order of state updates. This leads to the severe delay with big states and therefore Trident is suitable to handle only small states.

In contrast to Storm and Trident, Samza aims at managing the large state (i.e., GBs in each partition) by keeping state in local storage and using Kafka to duplicate the state’s changes. Kafka stores the log of state’s updates and therefore the Kafka’s topic can easily restore state. Samza uses a key-value store by default to support stateful operators, but other storage systems are also applicable if they offer richer querying capabilities.

### Table 1. State implementation comparison among frameworks

| State Management | Storm | Storm Trident | Spark Streaming | Samza | Flink |
|------------------|-------|---------------|-----------------|-------|-------|
| Fault Tolerance  | Tuples acknowledge | Tuples acknowledge | RDD lineage | Log of updates | State checkpoint |
| Guarantees       | At least once | Exactly once | Exactly once | At least once | Exactly once |

Spark implements state management by the concept of state DStream (discretised stream) that uses DStream transformation for update operations. The key concept of Spark is the distributed immutable collections, called RDD (resilient distributed dataset). Spark can tolerate faults by using lineage [Zaharia12] to avoid the overhead of checkpointing. State in Spark Streaming plays the role of another micro-batched stream. For that reason, during micro-batch processing, Spark takes a current state to generate another micro-batch result and a new state.

To support exactly-once semantics, Flink bases on the Chandy-Lamport algorithm for distributed snapshots. Specifically, watermark units periodically inserted into the data stream can activate the receiving operator to checkpoint its state. There exist two kinds of states in Flink: local state of an operator instance, and state of the whole partitions. Flink programs can define state in various ways: (1) using time-based windows, count-based windows, and generalized custom windows based on window transformations, (2) registering any type of object based on the Checkpointed interface, and (3) partitioning the cluster by a key based on the 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 internally as objects on the Java heap), backs up state in a file system (e.g., HDFS), or persists
state on RocksDB database. Flink’s community tries to offer users with more choices of state backends (e.g., uses Flink-managed memory that can spill to disk). To alleviate the delay of loading state during recovery, Flink replicates state to K TaskManagers. This replication allows for failures of up to (N-1) TaskManagers without loading state. Flink also introduces the concept of queryable state. This queryable state allows real time queries to directly access to event time windows, avoiding the overhead of writing to key/value stores.

Although these implementations of state differ in the ways to represent and store state, the common limitation in these frameworks is the lack of adaptive checkpoints. Currently, these frameworks support only periodic checkpoints for fixed interval (e.g., checkpoint every hour). Some works (e.g., [Sayed14]) have proved that aperiodic checkpoints (i.e., the checkpoint interval is not fixed in advance and can vary during the runtime) may generate better performance than periodic checkpoints. Therefore, one appealing research direction is to extend these frameworks to support adaptive checkpoints (i.e., determining the most beneficial moments to take checkpoints adaptively rather than taking checkpoints periodically). We can calculate these beneficial moments based on the costs of checkpoints (and recovery) at the time checkpoints happen. These costs, in turn, depend on the probability that failures occur. Consequently, this cost-based adaptive checkpoint model must integrate the anticipation of failure probability as an important parameter. Another direction is to devise an efficient representation of state (e.g., approximate state, compressed state, or incrementally updatable state) by using a novel data structure to allow iterative algorithms, such as machine learning algorithms, to run more efficiently. Section 12 discusses this idea in more details.

12. Open Discussions

We close this survey by discussing some motivating research directions related to state management. These directions include: (1) ways to integrate state management into big data frameworks, (2) ways to develop state management for iterative algorithms, (3) ways to allow state to support hybrid systems, and (4) evaluation metrics for state management.

12.1 How to Integrate State Management into Big Data Frameworks

Current big data frameworks can further extend and integrate existing techniques for state management at many levels, ranging from low level (e.g., operator primitives, calculus algebra) to higher level (e.g., language level or platform level). At low level, primitive libraries, they are defined and written in specific languages to achieve high performance. As the underlying mechanism to propose the algebra for distributed computing. This algebra, which comprises as the formal basis of Apache MRQL for optimizing the incremental state computation. We can extend this incremental change of state at the algebra level to support more functions of state than previous works mentioned in this survey.

At high level, some works [Silva09, Alexandrov15] support declarative languages for big data processing. Silva et al. [Silva09] propose a language to allow users to easily define and parameterize checkpoint policies. This framework uses language annotations to apply fault tolerance policies for streaming applications. Based on the fact that developer understands his application semantics and failure patterns, using language-level annotations is believed to be a natural approach to describe such policies. This approach ties language primitives with code generation to provide a flexible way for state checkpointing in streaming systems. More precisely, users can selectively checkpoint their applications. Beside fault tolerance, we can extend this work by using annotations for other state management methods as well. For example, two interesting ideas on extending annotations are to: (1) efficiently specify which parts of the application should be actively fault-tolerant, and which ones should employ passive methods, similar to [Su16]; and (2) determine which operators to publicly expose to users, and which ones remain privately accessible among internal operators only, resembling the concept of encapsulation in object-oriented programming. In other directions, Alexandrov et al. [Alexandrov15] propose a language, called Emma, which deeply embeds the APIs in a host language (e.g., Scala) for complex data analysis. This approach is able to (1) allow for declarative specification of dataflows and (2) use the intermediate representation to transparently deploy parallel computations. We can extend this language to integrate the state management at language level. This language can deeply embed the state management methods mentioned in this survey in a host language, enabling the declarative state management.

At high level platform, Rheem [Agarwal16a,b] introduces a three-layer data processing (i.e., platform, core and application layers) and a storage abstraction to support both platform independence and interoperability across multiple platforms. They envision the data processing abstraction fully based on platform independence and interoperability among platforms, applications can be independent from the data processing platforms. Rheem divides a complex analytic task 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. Based on this concept of platform independence, we can extend Rheem to build a state management system that allows easy deployments on various platforms, achieving platform independence, interoperability among platforms, and performance improvement.

As seen from a large number of visions in this subsection, some works have already started the declarative high-level supports for big data analysis. How we can combine the strength of these systems to take advantage of their benefits to support state management is a challenging and interesting research problem.

12.2 How to Develop State Management to Support Iterative Algorithms

Many machine learning algorithms, such as PageRank, k-means, and its variations, require iterative steps to converge to the final solution. Due to big state size, some iterative algorithms use approximate state with smaller size, or approximate algorithms with less iterative steps to boost performance. Usually, these
approximate algorithms have to sacrifice accuracy to exchange for performance. However, some works still ensure correctness while achieve good performance. This section lists these works and points out research directions on how to extend these works. This extension allows us to represent state in approximate forms in order to optimize these approximate (or exact) iterative algorithms for better performance.

12.2.1 Approximate State
Since it is difficult to achieve low latency and high throughput in big data processing with the ever-increasing data volume, generating approximate solution by approximating state is a promising research direction. We list several ways to approximate state in this subsection.

The preliminary solution is computing quantile as the summary of state in [Lin04] for state approximation. In this work, they study the problem of continuously maintaining quantile summary of the most recently observed N elements over a stream in order to answer quantile queries with a guaranteed precision of εN (i.e., ε-approximate quantile summary). They also develop an algorithm that maintains the quantile summary for the most recent N data elements in such a way that can obtain quantile estimates for the n most recent elements for any n ≤ N.

Other works [Garofalakis02, Johnson05] implement state by compact data structures [Aggarwal07] used to compute aggregate statistics of the processed tuples. In the past, data stream processing systems have used various synopsis structures, including sampling, wavelets, sketches, and histograms. The simplest method for state approximation is state sampling [Johnson05] that estimates underlying data with provable error guarantees. Sampling methods include random and concise sampling. In the wavelet technique, the basic idea is to decompose the data characteristics into a set of wavelet and basic functions, which is useful for hierarchical data decomposition and summarization. Sketch-based methods originate their motivation from wavelet techniques. In fact, sketch-based methods play the role as a randomized version of wavelet techniques. Histogram-based methods divide data along any attribute into a set of ranges, and maintain the count for each bucket. If data is vertically (or horizontally) divided, we have equi-width (or equi-depth) histograms, respectively. Some recent techniques [Garofalakis02] explore the design of histograms for dynamic uses. Finally, another compact representation of data streams is micro-cluster based summarization [Aggarwal03]. This micro-cluster summarization is suitable for multi-dimensional data, and effectively adapts to continuously changing data streams.

Jangjaimon et al. [Jangjaimon13] propose a multi-level checkpointing mechanism with delta compression at any given time adaptively. While incremental checkpointing reflects only modified and new pages, employing delta compression (aka. differencing compression) between consecutive checkpoints can further reduce the checkpoint size. This compression saves only the difference (i.e., delta) between each dirty page (i.e., target data) and its corresponding old version (i.e., source data) stored in the previous checkpoint. This work dynamically selects a desirable point of time to generate the smallest state size after delta compression. It is different from other adaptive checkpointing mechanisms, in which they dynamically skip certain fixed checkpoints.

12.2.2 Approximate Algorithms
Since iterative graph processing poses extreme overhead, especially with large graphs, equipping this graph processing with efficient approximate approaches is necessary to boost performance. In the literature, designing machine learning algorithms in approximate ways can accelerate the computation. In this subsection, we list some of the most popular approaches ([Broder06], [Parreira06], [Yossef08], [Zhu13,15], [Liu15], [Mitliagkas15]) that approximate two well-known algorithms (i.e., PageRank, and k-means).

To approximate PageRank algorithms, Broder et al. [Broder06] perform calculations on a compact representation of graph, termed graph aggregation, which allows for aggregating many pages onto a single node. By partitioning the graph into classes of quasi-equivalent vertices, this representation requires less memory than uncompressed ones. Parreira et al. [Parreira06,08] propose the JXP algorithm in the context of peer-to-peer network to dynamically and collaboratively compute the PageRank score of websites. The main idea of JXP is to perform only local computations to benefit low storage costs, and coordinate the interactions among peers for approximating global scores. Sharing this idea, Yossef et al. [Yossef08] use only local information to approximate the PageRank scores of a specific node. They point out that graphs without following two features can efficiently use this local approximation method: (1) abundance of high in-degree nodes, and (2) slow convergence of the PageRank random walk. Bahmani et al. [Bahmani10] employ Monte Carlo methods to incrementally compute PageRank and top-k personalized PageRanks. Zhu et al. [Zhu13,15] propose an approximate algorithm, called FastPPV, to incrementally compute Personalized PageRank Vector. They apply the scheduled approximation to gradually and incrementally refine the estimated values and measure the accuracy of approximation. Liu et al. [Liu15] use an adaptive sampling method to sample the transition matrix many times. This method adaptively and iteratively adjusts the sample rate at runtime. This adaptive sample rate aims at balancing the accuracy and efficiency for PageRank approximation. Mitliagkas et al. [Mitliagkas15] propose FrogWild for partial synchronizations of mirror vertices to optimize the network usage incurred by PageRank algorithms. This method can decrease the cost in each iteration, and thus significantly improve the performance of PageRank algorithms.

In addition to PageRank, several works [Kanungo02, Chitta11, Bahmani12, Zeng12, Cohen15] propose approximate algorithms to accelerate the k-means clustering computation. Kanungo et al. [Kanungo02] propose an approximation algorithm for k-means clustering using single-swap and multiple-swap heuristics based on swapping centers. In [Chitta11], the randomized approximation approach, called approximate kernel k-means, aims to reduce both computational complexity and memory requirements for kernel k-means. The core idea is to approximate the cluster centers of the randomly chosen subset of data points by using vectors in the subspace spanned by this subset. Therefore, it involves only a small part of the full kernel matrix, accelerating the performance of kernel k-means. Zeng et al. [Zeng12] approximate k-means algorithm by using random spatial partition trees to pre-organize groups, leading to the reduced computational overhead in the assignment step. Bahmani et al. [Bahmani12] introduce an initialization algorithm, termed scalable k-means, to generate 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 final solution. Finally, by using sketching techniques to approximate a data matrix to a smaller one, Cohen et al. [Cohen15] can solve the general class of constrained k-rank
approximation problems (including k-means) to within \((1+\varepsilon)\) error.

As we can see from this survey, there are many approximate machine learning algorithms that try to reduce the memory consumption in each iterative step. How to approximate state using techniques in subsection 12.2.1 and integrate into these approximate algorithms to further accelerate computation is a challenging problem.

12.2.3 Exact Algorithms

Beside approximate algorithms, some works [Fujiwara12a, Fujiwara13, Yu14] propose exact algorithms while still ensure low latency. K-dash [Fujiwara12a] aims at finding \(k\) nodes with highest proximities for a selected node. K-dash comprises two main ideas: (1) use sparse matrices to efficiently compute the proximity of a given node, and (2) save time by computing only the necessary proximity. K-dash not only runs significantly faster than existing approximate methods, but also ensures exactness. Similarly, Fujiwara et al. [Fujiwara12b] efficiently compute top-\(k\) nodes by pruning technique. The key idea is to estimate the upper and lower bounds of relevance and use these bounds to compute top-\(k\) nodes exactly by pruning unnecessary relevance computations. This bound-based pruning technique can efficiently generate correct results. Extending this idea, Fujiwara et al. [Fujiwara13] propose F-Rank to find top-\(k\) PageRank nodes efficiently by iteratively estimating lower and upper bounds of PageRank scores. F-Rank then constructs subgraphs in each iteration by pruning unnecessary nodes and edges to get top-\(k\) nodes. Yu et al. [Yu14] 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 guarantee. This algorithm has two important features: (1) improving the incremental computation of SimRank for every link update; (2) pruning unnecessary similarity recomputations for link updates, reducing the computation time of SimRank.

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

12.3 State Management for Hybrid Systems

While batching data provides comprehensive and historical views of data, real-time streaming data provides fresh and up-to-date information. In order to benefit the advantage of both kinds of data, some works [Boykin14, Meehan15,16] propose hybrid systems to process these two kinds of data at the same time. These hybrid systems get not only the overview of historical information but also the update on the most recent data.

The lambda architecture [Marz05] tries to process batching and streaming data at the same time by providing a software stack including several systems: (1) a batch layer (usually implemented in Hadoop) to process batching data, (2) a speed layer (implemented in Storm) to process streaming data, and (3) a serving layer to index the batch views (to allow 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 have to span across multiple systems, limiting its opportunity for optimizations. Therefore we cannot process data as efficiently as a single unified system can.

To partly overcome this weakness of lambda architecture, some works [Boykin14, Meehan15,16] propose hybrid systems to integrate multiple data types processing (e.g., real-time with batch or streaming with OLAP). Boykin et al. [Boykin14] propose Summingbird to integrate online and batch MapReduce computations into a single framework. To fuse the stream and transaction processing in a single system, Meehan et al. [Meehan15] build S-Store by starting with a fully transactional OLTP main-memory database system and then integrating additional streaming functionality. In this way, S-Store can simultaneously and seamlessly accommodate OLTP and streaming applications. Meehan et al. [Meehan16] design BigDAWG to tightly integrate real-time and batch processing, enabling seamless and high performance querying capability over both fresh and historical data. Elmore et al. [Elmore15] demonstrate the effectiveness of BigDAWG through practical applications.

The previous systems (e.g., S-Store, and Summingbird) do not focus directly on combining batching and streaming data in a single system. Consequently, future works can encapsulate the entire functionality of the lambda architecture in a single system to take advantages of both worlds, batch and stream. 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 batching and streaming data in a single framework is a fascinating research problem. Since batching and streaming data have specific characteristics, how to develop methods for efficient state management to satisfy both characteristics of these kinds of data is a non-trivial problem.

12.4 State Management Evaluation Metrics

Together with the research problems mentioned in subsections 12.1, 12.2 and 12.3, the next question is how to evaluate the proposed solutions. What standards or criteria we must base on to evaluate these solutions of state management? We propose to use the following metrics to serve this purpose.

- **Efficiency:** state management methods must have low overhead in comparison to existing approaches.
- **Ease of Use/Management:** APIs for using and accessing state must be simple and easy to use. They can cover most application scenarios and provide richer functions and encapsulations. This can help to reduce the cost of human latency in deploying and using big data frameworks in future.
- **Functionality:** state can efficiently support iterative algorithms in many different domains such as machine learning, data mining, artificial intelligence, etc. In this case, it must support multiple consistency guarantees and allow users to choose which consistency level to use.
- **Seamless Integration:** new methods can easily integrate into existing, under-development, and future frameworks for big data processing. This integration must be effective (i.e., without spending too much effort to modify the existing underlying platforms).
- **Benchmark for state management evaluation:** one appealing research direction is to introduce benchmarks for state management solutions. Then, the metrics for benchmarking evaluation must be devised.
How the proposed solutions for state management satisfy these metrics introduces a large number of research problems in this area.

13. Conclusion
In this paper, we survey the state management in its applications for big data processing, namely stateful computation, fault tolerance, iterative processing, elasticity & load balance, integrative optimizations, and state sharing. Then we show how to maintain, and operate on it. Finally, we discuss its complexity and implementations. Together with the high-level classification in Figure 1, the table in the appendix section lists all the systems mentioned in this paper as well as their corresponding contributions to the big picture of state management. Finally, the open discussion at the end of this survey can help researchers to have a more general understanding of state management and to encourage more research ideas in this field. We hope that this survey will pave the way for subsequent research on state management for big data processing.

Acknowledgments
This work was partially funded by the STREAMLINE project under grant number xxx.

References
[1] D. Aggarwal, S. Crawla, A. Elmagarmid, Z. Saoudi, M. Ouzzani, P. Papati, J.-A. Quian’e-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.
[2] D. Agrawal et al. Rheem: Enabling Multi-Platform Task Execution. In ACM International Conference on Management of Data (SIGMOD), pages 2069-2072, 2016.
[3] 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.
[4] 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.
[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. Ananthanarayan, 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. Sharma, A. Haley, 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. Sturzheim, 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. Föra, S. Ewen, S. Haridi, K. Tzoumas. Lightweight Asynchronous Snapshots for Distributed Dataflows. The Computing Research Repository (CoRR), abs/1506.08603, 2015.
[19] Z. Chen, J. Dongarra. Highly Scalable Self-Healing Algorithms for High Performance Scientific Computing. IEEE Transactions on Computers, 58(11):1512-1524, 2009.
[20] 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.
[21] M. B. Cohen, S. Elder, C. Musco, 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.
[22] T. Condie, N. Conway, P. Alvaro, and J. M. Hellerstein. MapReduce online. In NSDI, 2010.
[23] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. Communication of ACM, 51(1):107–113, 2008.
[24] 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.
[25] J. Ding, T. Z. 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.
[26] S. Dudoladov, C. Xu, S. Schelter, A. Katsifodimos, S. Ewen, K. Tzoumas, V. Markl. Optimistic Recovery for Iterative
Appendix. A Classification of State Management Approaches By Functionality

| Approach/Paper | Stateful Computation | Fault Tolerance | Iterative Processing | Elasticity & Load Balance | Integrative Optimization | State Sharing | Incremental Maintenance | Operations | Overhead Complexity |
|----------------|----------------------|-----------------|----------------------|---------------------------|--------------------------|---------------|-------------------------|------------|---------------------|
| Flink          | ✓                    | ✓               | ✓                    | ✓                         |                          |               |                         |            |                     |
| Spark          | ✓                    | ✓               |                       |                           |                          |               |                         |            |                     |
| Storm          | ✓                    |                 |                       |                           |                          |               |                         |            |                     |
| Sansa          | ✓                    | ✓               |                       |                           |                          |               |                         |            |                     |
| [Logothetis10] | ✓                    |                 |                       |                           |                          |               |                         |            |                     |
| [Logothetis09] | ✓                    |                 |                       |                           |                          |               |                         |            |                     |
| [Gedik14]      | ✓                    | ✓               | ✓                    |                           |                          |               |                         |            | ✓                  |
| [Matten16]     | ✓                    |                 |                       |                           |                          |               |                         |            | ✓                  |
| CEC (Sebepou11)|                     |                 |                       |                           |                          |               |                         |            | ✓                  |
| [Hwang07], [Kwon08] |                 |                 |                       |                           |                          |               |                         |            | ✓                  |
| scalable coding strategies (Chen09) |               |                 |                       |                           |                          |               |                         |            | ✓                  |
| MetroShower (Wang12) |                  |                 |                       |                           |                          |               |                         |            | ✓                  |
| [Koldehofe13], [Hakkarinen13] |               |                 |                       |                           |                          |               |                         |            | ✓                  |
| PPA (Sal16)    | ✓                    |                 |                       |                           |                          |               |                         |            | ✓                  |
| FTSr (Upadhyaya11) |                 |                 |                       |                           |                          |               |                         |            | ✓                  |
| AF-Colt [Nicolae13] |                 |                 |                       |                           |                          |               |                         |            | ✓                  |
| [Naksinehaboon08] | ✓                  |                 |                       |                           |                          |               |                         |            | ✓                  |
| [Pram10]       | ✓                    |                 |                       |                           |                          |               |                         |            | ✓                  |
| ADC (Jangjaimon13) |                 |                 |                       |                           |                          |               |                         |            | ✓                  |
| ABS (Carbone15) | ✓                    |                 |                       |                           |                          |               |                         |            | ✓                  |
| [Ewen12,13]    |                     |                 |                       |                           |                          |               |                         |            |                     |
| [Scheufler13]  | ✓                    |                 |                       |                           |                          |               |                         |            |                     |
| [Dudoladov15]  | ✓                    |                 |                       |                           |                          |               |                         |            |                     |
| head & tail checkpoint [Xu16] |               |                 |                       |                           |                          |               |                         |            | ✓                  |
| MRQL Streaming [Fegaras16a] |               |                 |                       |                           |                          |               |                         |            | ✓                  |
| Flux (Shah05)  | ✓                    |                 |                       |                           |                          |               |                         |            | ✓                  |
| ChronoStream [Wu15] |                 |                 |                       |                           |                          |               |                         |            | ✓                  |
| differential dataflow [McSherry12] |               |                 |                       |                           |                          |               |                         |            | ✓                  |
| Naiad [McSherry13] |                 |                 |                       |                           |                          |               |                         |            | ✓                  |
| [Fernandez13]  | ✓                    |                 |                       |                           |                          |               |                         |            | ✓                  |
| [Madani15, 16] | ✓                    |                 |                       |                           |                          |               |                         |            | ✓                  |
| [Brito08]      |                     |                 |                       |                           |                          |               |                         |            | ✓                  |
| [Gordon06, Arasu06] |                 |                 |                       |                           |                          |               |                         |            | ✓                  |
| [Semmler05]    | ✓                    |                 |                       |                           |                          |               |                         |            | ✓                  |
| [Kuntschke05]  | ✓                    |                 |                       |                           |                          |               |                         |            | ✓                  |
| CAPSULE [Lina12] |                 |                 |                       |                           |                          |               |                         |            | ✓                  |
| S-Store [Meehan15] [Tabu15] | ✓ |
|---|---|
| ring of databases [Koch10] | ✓ |
| viewlet transforms [Ahmad12] | ✓ |
| DiffToaster [Koch14] | ✓ |
| LDVIEW [Nikolic14] | ✓ |
| [Nikolic16], [Koch16] | ✓ |
| [Liu16] | ✓ |
| [Zhang15] | ✓ |
| [Liu06] | ✓ |
| SGuard [Kwon08] | ✓ |
| CALC [Ren16] | ✓ |
| Photon [Ananthanarayanan13] | ✓ |
| SDG [Fernandez14] | ✓ |
| GraphLab [Low12] | ✓ |
| Plan [Ding04] | ✓ |
| Punctuation semantics [Tucker03] | ✓ |
| [Li08], [Zhu04] | ✓ |
| [Ding16] | ✓ |
| StreamCloud [Gulisano12] | ✓ |
| SIRON [Pietzuch06] | ✓ |
| MegCEP [Ostenwalder13] | ✓ |
| MLCBF [Feng11] | ✓ |
| [Sayed14] | ✓ |
| [Roberti12] | ✓ |
| [Bouguerra13] | ✓ |