Poster: Log Pruning in Distributed Event-sourced Systems

Benjamin Erb
Dominik Meißner
Ferdinand Ogger
Frank Kargl
[firstname].[lastname]@uni-ulm.de
Institute of Distributed Systems
Ulm University, Germany

ABSTRACT

Event sourcing is increasingly used and implemented in event-based systems for maintaining the evolution of application state. However, unbounded event logs are impracticable for many systems, as it is difficult to align scalability requirements and long-term runtime behavior with the corresponding storage requirements. To this end, we explore the design space of log pruning approaches suitable for event-sourced systems. Furthermore, we survey specific log pruning mechanisms for event-sourced logs. In a brief evaluation, we point out the trade-offs when applying pruning to event logs and highlight the applicability of log pruning to event-sourced systems.

CCS CONCEPTS

• **Applied computing → Event-driven architectures**; • **Computing methodologies →** *Distributed computing methodologies*; • **Computer systems organization →** *Distributed architectures*;

KEYWORDS

event sourcing, event processing, log pruning

ACM Reference Format:
Benjamin Erb, Dominik Meißner, Ferdinand Ogger, and Frank Kargl. 2018. Poster: Log Pruning in Distributed Event-sourced Systems. In *DEBS '18: The 12th ACM International Conference on Distributed and Event-based Systems, June 25–29, 2018, Hamilton, New Zealand.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3210284.3219767

1 INTRODUCTION

In modern software architectures, event-driven approaches experience appreciation for their inherent properties: asynchronous, message-based interaction of loosely coupled, modular components [9] — eventually enabling important features for distributed architectures such as scalability, elasticity, extensibility, and reliability. Recent trends in distributed, event-driven architectures include the reactive manifesto [3] and the emergence of microservice architectures [13].

A complementary property of event-driven systems is the handling of application state. Event sourcing is an architectural style that suggests an append-only log of state-modifying events for application state persistence [8]. Events are immutable and capture the changes introduced by the event instead of directly replacing the previous state. Application state or the state of individual application entities can be reconstructed by applying the sequence of all existing events. Notably, state reconstruction is not limited to the latest states, since any previous application state can be reconstructed (i.e., by applying events up to a certain position in the log). This feature is useful for a number of use cases, including the exploration of application history, as an audit trail, for debugging purposes, and for data lineage [12]. Popular programming solutions based on event sourcing include the Lagom microservices framework[1], the Eventuate platform[2], or the Axon framework[3].

As an application progresses, and thus the associated application state evolves and encounters updates, the log of an event-sourced application grows steadily. This is an inherent consequence of the write-once, append-only semantics of the log and the immutability of the events. However, unbounded event logs are disadvantageous for many applications due to storage requirements, hindered scalability and restricted maintainability.

With this in mind, we explore the applicability of pruning mechanisms to the logs of event-sourced systems. Based on a model for distributed event-sourced systems from section 2, we survey log pruning approaches in section 3. In section 4, we evaluate their implications before we summarize our findings in section 5.

2 SYSTEM MODEL

We consider an event sourcing model that takes into account command sourcing and event sourcing, complemented by the Command Query Responsibility Segregation (CQRS) principle [2, 14]: The system maintains application state by accepting application-level intents for state-changing operations (i.e., commands) and for decoupled state-reading operations (i.e., queries). A command is handled by a command processor, that yields a state modification in the form of an event. Both the command and the corresponding event are persisted as a tuple in the event log. The sequence of all events can be used to reconstruct states of the application or its entities while the sequence of commands allows for a re-execution.

For the concrete system model, we choose a distributed system in which multiple processes hold their own internal state and interact via asynchronous message passing with immutable content [1]. More specifically, we consider an event-sourced actor model [5],

DEBS '18, June 25–29, 2018, Hamilton, New Zealand
© 2018 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-5782-1/18/06.
https://doi.org/10.1145/3210284.3219767

[1]https://www.lagomframework.com/
[2]https://www.eventuate.io/
[3]http://www.axonframework.org/

in which each actor maintains an internal event log which defines its state. An incoming message is regarded as a command for the receiving actor. For message handling, an actor can update its internal state, effectively adding a new event to its event log and potentially sending new message (i.e., new commands) to adjacent actors. This purely functional model of asynchronously communicating, shared-nothing actors is complemented with a second type of actor for interactions with external systems. These I/O actors are not taken into account for event sourcing. However, they can exchange messages with event-sourced actors, thereby bridging external systems and the network of event-sourced actors [7]. A global application state in this model is represented by the set of individual states of its actors with causally consistent properties — i.e., a receiving actor must not be in a state where it has processed a message that is not yet dispatched by the sending actor at its corresponding state [6].

This system model aligns well with microservices architectures, which also rely on lightweight, isolated services with a shared-nothing design, message-based loose coupling, and a fine-grained modularity of dedicated process or service instances. By examining a single actor instance and treating others like external entities, the model additionally allows to analyze event-sourced systems with a single log and without distribution and concurrency aspects.

Note that we only consider application-level mechanisms for log pruning in our model, namely the event log. We do not take into account complementary optimizations provided on other levels such as modifications of the runtime platform or storage engine.

3 APPROACHES

Due to the fact that a current application state is entirely derived from the sequence of past events, log entries cannot be simply deleted [11]. Furthermore, any deliberate reduction of the log inevitably reduces the granularity of reproducible past states. In fact, log pruning is always a lossy process. We now consider primitives for reducing and pruning log segments and the design space for log pruning approaches.

3.1 Event Log Primitives

An individual actor a at the local, logical clock value t is defined by the sequence of past commands c and events e:

$$\text{Log}_a^t = (\{c_a^{t_0}, e_a^{t_0}\}, \{c_a^{t_1}, e_a^{t_1}\}, \ldots, \{c_a^t, e_a^t\})$$

The state s_a^t is defined by *left-folding* all previous events of a:

$$s_a^t = \int_{i=0}^{t} e_a^{t_i} = \text{fold}_l \left(\{\}, \left[e_a^{t_0}, e_a^{t_1}, \ldots, e_a^t\right]\right)$$

An incoming command c_a^{t+1} yields a new event e_a^{t+1} after is has been processed by the command handler f_a^t:

$$e_a^{t+1} = f_a^t(c_a^{t+1}, s_a^t) = \Delta\left(s_a^t, s_a^{t+1}\right)$$

In order to prevent repetitive recomputations of past states, an event log tuple can be augmented by a *checkpoint*, i.e., a persisted full state representation $s_a^{t'}$ after the application of $e_a^{t'}$.

$$\text{Log}_a^t = (\{c_a^{t_0}, e_a^{t_0}\}, \ldots, \{c_a^{t'}, e_a^{t'}, s_a^{t'}\}, \ldots, \{c_a^t, e_a^t\})$$

This speeds up reconstructions at the expense of increased storage:

$$s_a^t = \text{fold}_l \left(s_a^{t'}, \left[e_a^{t'+1}, \ldots, e_a^t\right]\right)$$

Left-folding can also be used to merge multiple log entries, but decreases the number of reconstructable intermediate states:

$$\Delta\left(s_a^t, s_a^{t+i}\right) = \text{fold}_l \left(e_a^t, e_a^{t+1}, \ldots, e_a^{t+i}\right)$$

For a distributed consideration, we first require a set of clock values in such a way that they satisfy the aforementioned causality requirements of their logical clocks. *Consistent cuts* issue such global properties and can be computed for distributed event logs of individual actors [6]:

$$\text{Cut}_a^t = \{b^{t'} \in \text{Actors} \mid c_a^t \not\rightarrow c_b^{t'}\}$$

In turn, a *snapshot* stands for the reconstruction of the states of all actors from a given consistent cut at corresponding clock values. The snapshot provides a globally consistent state defined by the set of the local states. In distributed event-sourced systems, snapshots can be calculated retrospectively, if the actor state also contains causality information [6].

Checkpointing and snapshotting are fundamental primitives for log pruning. They ensure the reconstructability of later states while they simultaneously eliminate dependencies from any previous log entries. Left-folding of event sequences provides a complementary primitive to reduce a sequence of events to a single entry.

3.2 Design Space

In a distributed event-sourced system, there are multiple dimensions to consider when applying log pruning. Log pruning mechanisms can be conducted *proactively* (e.g., continuous operation) or *reactively* (e.g., once a certain condition occurs). The *initiation of log pruning* can be caused within the application (e.g., a domain-specific trigger), by the underlying runtime platform (e.g., a log size exceeds a threshold), or by a user action.

Regarding the *scope of pruning*, mechanisms can either target the log of a single actor, subsets of the actor network, or the logs from all actors in the network.

Pruning strategies define which parts of the log are retained and which parts are considered for reduction. Window-based approaches define a period of time (logical event time or wall-clock time) in which any intermediate state of an actor can be reconstructed. After this window, state reconstruction granularity is limited. Similarly, a bounded ring buffer limits the number of latest log entries to store before switching to a coarser granularity for older entries. Once an entry leaves the window or ring buffer, it is moved to the lossy tail of the log and will eventually become subject of a reduction step. Both the window-based and buffer-based approach can also be extended to represent a hierarchical strategy with multiple levels, each further decreasing the reconstruction granularity. A probabilistic strategy randomly selects entries for pruning, based on a certain distribution (e.g., skewed towards older entries).

The *pruning step* defines the actual pruning operation — either merging an event with its predecessor or successor. or ensuring that the ramifications of the event are included in a checkpoint.

In a combined command-sourced/event-sourced log, the duality of commands and events allows for another pruning mechanism provided command logic is deterministic. Given state s_a^{t-1}, the state s_a^t can either be reconstructed by applying the event e_a^t, or by re-executing the command c_a^t. Hence, storing a tuple of both log items is redundant. Commands and events differ in their characteristics, as commands are usually more terse, but require an actual re-execution and cannot be merged easily.

3.3 Log Pruning Approaches

Based on these primitives, we now outline a number of pruning approaches for the individual logs in event-sourced systems.

3.3.1 Baseline Approaches. As baseline, we use a regular command sourcing/event sourcing approach (CE) without any pruning. Furthermore, we consider variants in which either commands (CS) or events (ES) are used. In addition, we use a baseline approach that only maintains a latest checkpoint and directly applies new events to that checkpoint (CP). This approach does not store commands/events and equals predominant state handling approaches.

3.3.2 Bounded Approaches. The basic bounded buffer approach (BB) limits each event log to the latest k entries and pulls behind a single checkpoint for earlier states. The tumbling window approach (TW) maintains a window of k up to $k + w$ latest entries. Furthermore, every w-th entry, the past log beyond the window is compacted to a full checkpoint.

3.3.3 Probabilistic Approach. The probabilistic approach (PB) selects random entries and merges the event with the previous event, while discarding the command. The choice of the random function allows a bias towards older entries (e.g., power law-based).

3.3.4 Hierarchical Approach. The hierarchical approach (HR) provides a number of consecutive windows, each with decreased reconstruction granularity by increasingly merging together entries. In an exponential configuration, the latest k entries remain in the newest sliding window. Thereafter, in the subsequent windows, each of size k, two elements from the previous window are merged into a single entry.

4 EVALUATION

For the evaluation, we selected a simulation-based approach that reproduced a distributed, event-sourced microservice architecture. We examined the actual storage space requirements and the extent of state reconstructability for the event logs when applying the different pruning mechanisms.

The simulated service handles external requests by orchestrating a number of internal service functions (i.e., stateful actors) in order to provide responses. The resulting topology and interaction patterns resemble a microservice architecture that follows the scatter-gather pattern. Each incoming external request yields a number of interactions, which in turn result in commands and corresponding events in the logs of the actors involved.

Our evaluation artifacts follow the Popper convention [10] and are available online[4]. This repository also contains more detailed information on the simulated microservices, workload patterns, and implementations of the pruning approaches that had to be left out for brevity.

4.1 Method

Simulation workloads were defined by the sequence of incoming requests and characteristics of the participating microservice instances. For the backend microservices, we differentiated request/response-based services for application-level data (e.g., caching, session handling, storage) with large numbers of updates and compute services with more complex command handling logic and only minor state changes.

The simulation was implemented in JavaScript (i.e., ECMAScript 2017) using JSON for all states, and JSON patches [4] for capturing state changes (i.e., events). All measurements were conducted on a machine equipped with an Intel Core i7 7700 CPU (quad-core with SMT; 3.60 GHz) and 32 GB of memory. The evaluation consisted of three steps: (i) creation of event logs for all actors according to the workloads, (ii) separate execution of all pruning mechanisms, and (iii) computation of their respective measurement metrics.

4.2 Results

Figure 1 highlights the evaluation results of 35 randomized workload streams, based on the simulated microservice architecture and each with $N = 100,000$ log entries. Log sizes represent the sums of all individual log files of all actors. Reconstructability denotes the percentage of earlier states that the system is able to reconstruct based on available log entries.

Without pruning (i.e., CE, CS, ES), all states can be reconstructed. In the CP approach, by contrast, only the latest states of all actors can be reconstructed and reconstructability approaches nearly 0%. Thus, the log size of the CP approach can also be used as a reference for the actual state size of the application over time.

For the actual pruning approaches (BB, TW, PB, HR), the development of reconstructability and log size values allows an estimation of their utility. The bounding effects of BB became apparent in the slowly growing logs. However, the bounding factor k heavily affects the reconstructability for older entries. The TW approach uses periodic checkpoints to ensure the existence of a few past states at the expense of checkpoint storage space. The event-merging mechanism of the HR approach mitigates the bounding effects to some extent without excessive storage requirements. Outcomes of the PB approach heavily depend on the choice of configuration parameters. Results for the evaluated configuration — 10% chance of merging a random event log entry, Zipfian-based selection of the item to merge (bias towards older entries from the event log) — remain inconclusive for actual use.

When choosing between command sourcing, event sourcing, or a combination of both, the choice depends on the expected size of commands and corresponding events (in our workload, events were slightly larger). A common drawback of command sourcing is the necessity of recomputations, which would render this approach computationally expensive in most cases. For the fixed pruning approaches (BB, TW), the choice of the complementing checkpointing strategy (e.g., single, periodic) determines log growth significantly, but also affects state reconstructability.

[4]https://github.com/vs-uulm/debs2018-log-pruning-evaluation

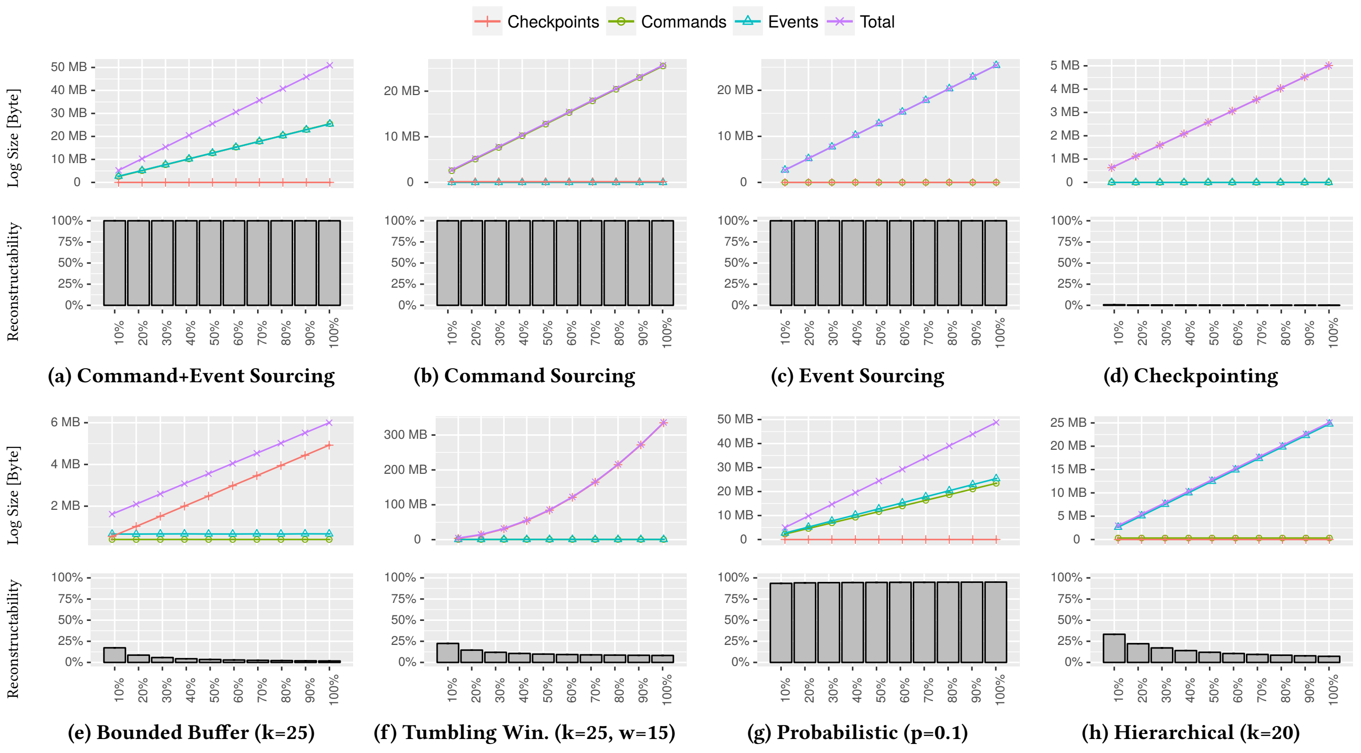

Figure 1: Overview of the log pruning approaches using a synthetic microservice workload. Each upper plot shows the mean combined log size and each lower plot provides the mean percentages of reconstructable states. Measurements have been taken progressively, starting at the point in time at which 10% of the workload had been processed up to its completion (100%).

5 CONCLUSION

Event-sourcing enables the reconstruction of arbitrary past application states for event-based applications. However, an entirely unbounded log size can conflict with other application requirements. We provided a first exploration of log pruning approaches for event-sourced systems and evaluated their effects as the main contributions of this work. This included an overview of the design space of pruning approaches and an assessment of the impact of log pruning mechanisms on state reconstructability.

Unbounded command and event sourcing approaches enable a full reconstruction of previous states, but come with high costs in terms of storage. In practice, pure command sourcing is often not feasible due to complex re-computations. Bounded approaches restrict the log sizes, but enable reconstructions primarily for more recent states. Additional periodic checkpoints can help to maintain some older states as reference. The probabilistic approach provides a configurable bias to determine what to prune. Eventually, the choice of a log pruning approach has to reflect the application-level requirements — which entity states should be kept, how far the application needs to go back in time, and in which resolution.

Application-based and user-defined pruning mechanisms as well as a detailed analysis of pruning parameters constitute future work.

REFERENCES

[1] Gul Agha. 1986. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, MA, USA.

[2] Dominic Betts, Julian Dominguez, Grigori Melnik, Fernando Simonazzi, and Mani Subramanian. 2013. *Exploring CQRS and Event Sourcing: A journey into high scalability, availability, and maintainability with Windows Azure.* Microsoft.

[3] Jonas Bonér, Dave Farley, Roland Kuhn, and Martin Thompson. 2014. The Reactive Manifesto. http://www.reactivemanifesto.org/. (Sept. 2014). Accessed: 2018-02-18.

[4] Paul C. Bryan and Mark Nottingham. 2015. JavaScript Object Notation (JSON) Patch. RFC 6902. (14 Oct. 2015).

[5] Benjamin Erb, Gerhard Habiger, and Franz J Hauck. 2016. On the potential of event sourcing for retroactive actor-based programming. In *First Workshop on Programming Models and Languages for Distributed Computing.* ACM.

[6] Benjamin Erb, Dominik Meißner, Gerhard Habiger, Jakob Pietron, and Frank Kargl. 2017. Consistent Retrospective Snapshots in Distributed Event-sourced Systems. In *Conference on Networked Systems.* Göttingen, Germany.

[7] Benjamin Erb, Dominik Meißner, Jakob Pietron, and Frank Kargl. 2017. Chronograph: A Distributed Processing Platform for Online and Batch Computations on Event-sourced Graphs. In *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems.* ACM, New York, NY, USA, 78–87.

[8] Martin Fowler. 2005. Event Sourcing. http://martinfowler.com/eaaDev/EventSourcing.html. (Dec. 2005). Accessed: 2018-04-23.

[9] Gregor Hohpe. 2006. *Programming Without a Call Stack - Event-driven Architectures.* Technical Report. eaipatterns.com.

[10] Ivo Jimenez, Michael Sevilla, Noah Watkins, Carlos Maltzahn, Jay Lofstead, Kathryn Mohror, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2017. The popper convention: Making reproducible systems evaluation practical. In *Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2017 IEEE International.* IEEE, 1561–1570.

[11] Martin Kleppmann. 2017. *Designing Data-Intensive Applications.* O'Reilly.

[12] Luc Moreau, Paul Groth, Simon Miles, Javier Vazquez-Salceda, John Ibbotson, Sheng Jiang, Steve Munroe, Omer Rana, Andreas Schreiber, Victor Tan, and Laszlo Varga. 2008. The Provenance of Electronic Data. *Commun. ACM* 51, 4 (April 2008), 52–58.

[13] J. Thönes. 2015. Microservices. *IEEE Software* 32, 1 (Jan 2015), 116–116.

[14] Greg Young. 2010. CQRS Documents. https://cqrs.files.wordpress.com/2010/11/cqrs_documents.pdf. (Nov. 2010). Accessed: 2018-04-28.