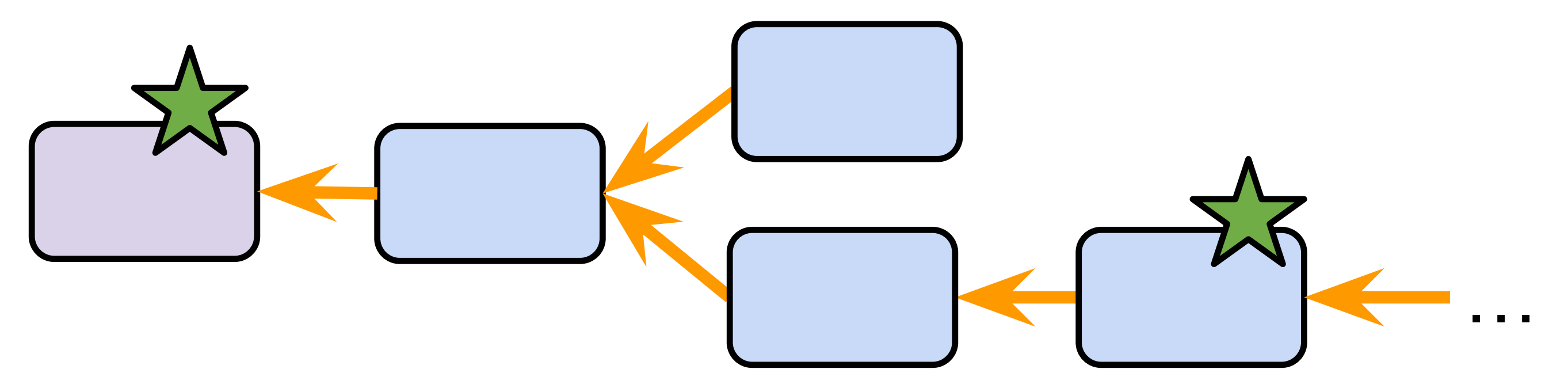
Secure Distributed Storage Plane with DataCapsules

Samuel Berkun, Ted Lin, Saketh Malyala

Introduction

- ***DataCapsules*** are data containers with **built-in integrity, provenance, and eventual consistency**.
- Designed to be a **standardized abstraction** for **distributed storage over untrusted infrastructure**.

Basic Model



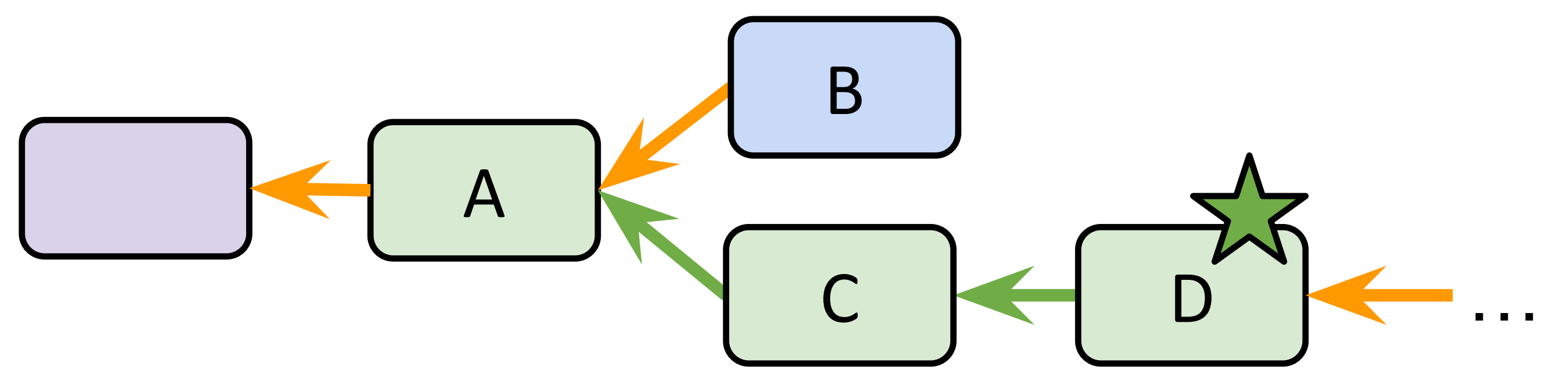
- A DataCapsule fundamentally consists of a DAG of **immutable records** linked by **secure hashes**.
- The root of the DAG is **metadata** for the DataCapsule as a whole, representing global identification, provenance, and access control.
- Designated writers **digitally sign** records such that readers can obtain proofs of integrity and provenance.

Our Goals

- Build a **practical Rust-based implementation** of DataCapsule servers.
- Explore how to provide **efficient integrity proofs**, including taking advantage of secure hash chaining and client-side caching. Explore the associated tradeoffs between read and write performance.
- Develop a DataCapsule-specific **anti-entropy** strategy to patch unintended branches and holes in the DAG that appear due to **faults under replication**.
- Experiment with **variations of the DataCapsule model/API** that may inherently perform better.
- Evaluate the **overhead of guaranteeing the secure properties of DataCapsules** compared to existing storage solutions without them.

Efficient Integrity Proofs

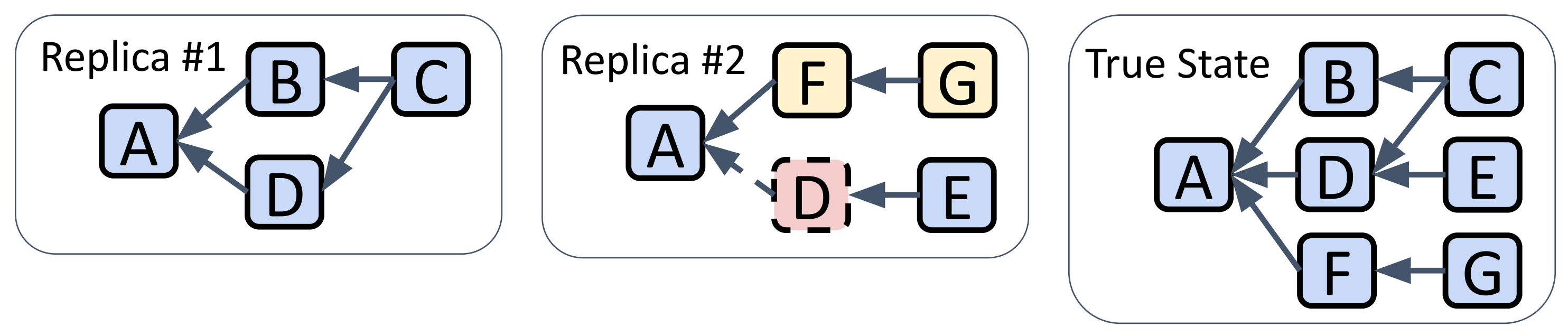
- Problem: **Durably signing every single record can be inefficient/impractical for writer clients** like sensors or other low-power devices.
- Observation 1: The **secure hash chain** from (unsigned) record-to-prove "A" **to a proven descendant record** "D" is a **sufficient proof** for "A".



- Observation 2: It doesn't matter *how* we got the proof for "D". D itself may not even be signed!
    - **Caching any previously-proven record (not just direct signatures) can speed up subsequent proofs.**
- Observation 3: There may be multiple hash chains from the record-to-prove to proven descendants - how does a server choose one to return?
    - Note it may be impractical/unscalable for server to track what individual readers have cached.
    - **Simple heuristic: closest signed descendant.** Can be **eagerly updated by each server** on every signature received. Also **globally monotonic**, which lets replicas safely gossip updates.

Anti-Entropy / Convergence of Replicas

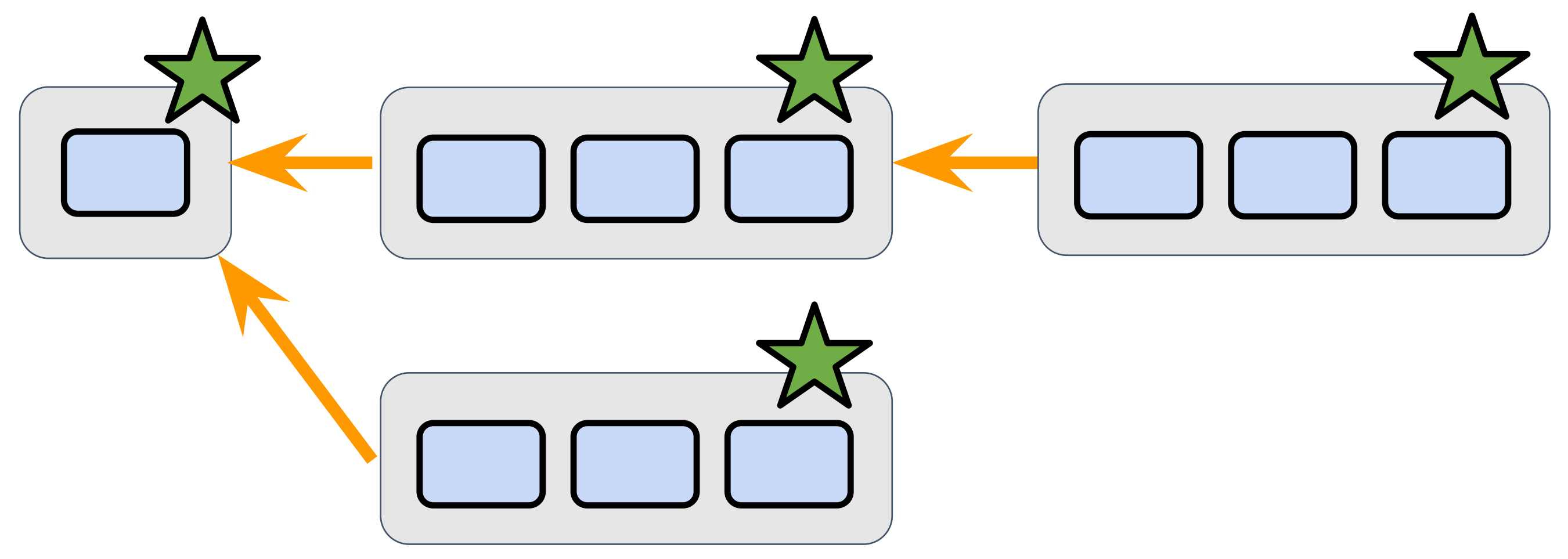
- DataCapsules are meant to be **replicated and distributed**, which introduces classic **consistency** problems.
- Unintended branches and holes in a DataCapsule's DAG can appear when a subset of the network / servers fail.



- DataCapsules have CRDT properties, so **convergence of replicas can be safely achieved by gossiping**, e.g. as a background task.
- In existing work on anti-entropy for DataCapsules, pairs of replicas share their DAG sources+sinks to deterministically reach convergence. This can be **expensive under frequent failures**.
- Extension: Use **Bloom filters** (improving network bandwidth) to detect which records the other replica definitely doesn't have. To account for false positives, fall back on deterministic algorithm.

Experimental Variation of Model/API

- Many applications create groups of commits with a Merkle tree structure (i.e. KV store, datacapsule-based file system).
- Performance gains can be achieved by optimizing the system around Merkle trees.



- Under the experimental model, each record consists of one or more sub-records, which are automatically linked together by a Merkle tree. The outer records still form a DAG. Each sub-record can be read and proved individually, thus allowing them to fill the role of the "normal" model's records.
- Extreme example: in an immutable KV store, the entire KV store may be one record, with individual key-value pairs being sub-records.
- Equivalent to base design if every record has exactly one sub-record.

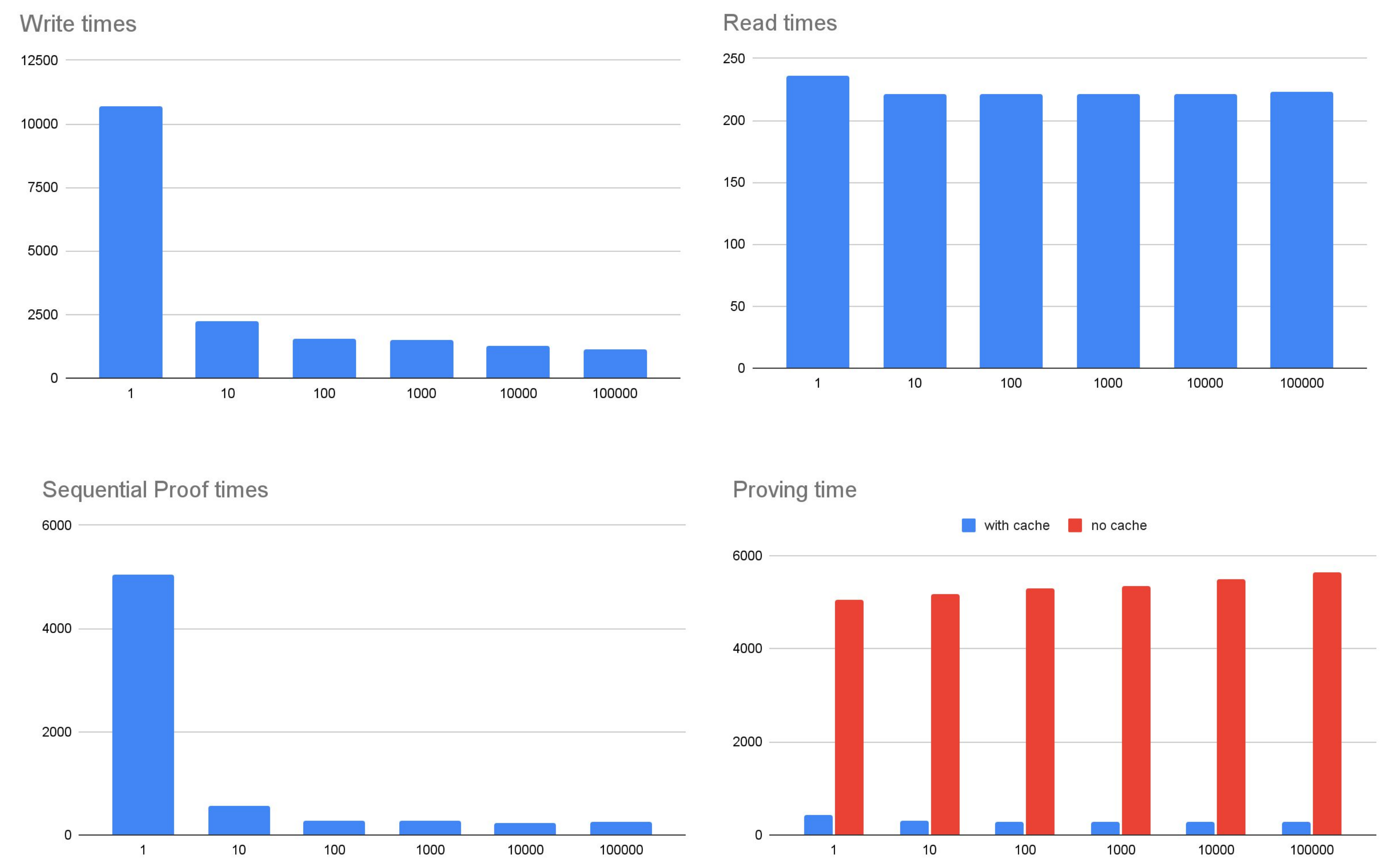


Fig 4: Benchmarks. Each bar is the time it took to do 10k operations, in milliseconds. The x-axis is how many sub-records there are per record.

- For workloads with many sub-records per record, up to 9.5x faster writes and 19x faster integrity proofs than base design.
- Caching for integrity proofs (see Efficient Integrity Proofs section) works particularly well under this model; allows reverse sequential proving times to be up to 20x faster than they would be without a cache.