DATA+AI SUMMIT 2022

Ensuring Correct Distributed Writes to Delta Lake in Rust with Formal Verification

QP Hou

ORGANIZED BY databricks

A bit about myself

https://about.houqp.me

- Currently core software lead at Neuralink
  - https://neuralink.com/careers/
- Formally tech-lead of Data and AI platform at Scribd
  - https://www.scribd.com/about/engineering
  - Delta-rs is a team effort between Christian (@xianwill), Mykhailo (@mosyp) and Tyler (@rtyler).
- Authors and contributors of many open-source projects:
  - Delta-rs
  - Apache Arrow
  - Apache Airflow
  - ROAPI
  - KOReader
  - ...

Delta-rs

- Native Delta lake implementation in pure Rust
  - With Python and Ruby bindings
- Previous Data+AI 2021 talk: https://databricks.com/session_na21/growing-the-delta-ecosystem-to-rust-and-python-with-delta-rs

Agenda

- Delta Lake 101
- Distributed concurrent table writes in delta-rs
- Formal verification overview
- Intro to stateright

Delta Lake 101

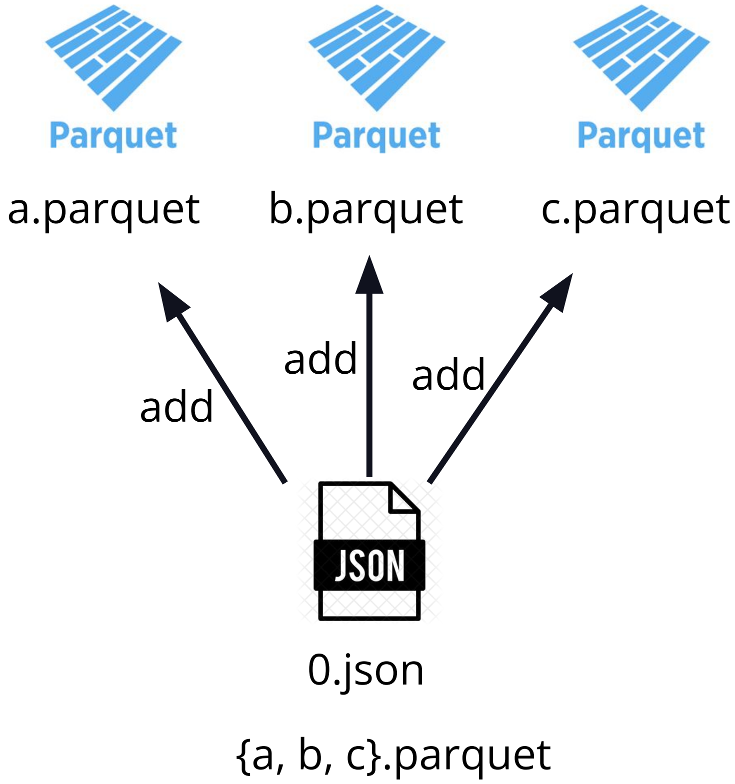
What is a Delta lake anyway?

- Brings ACID to traditional data warehouse solution
- Single data source for both batching and streaming workload

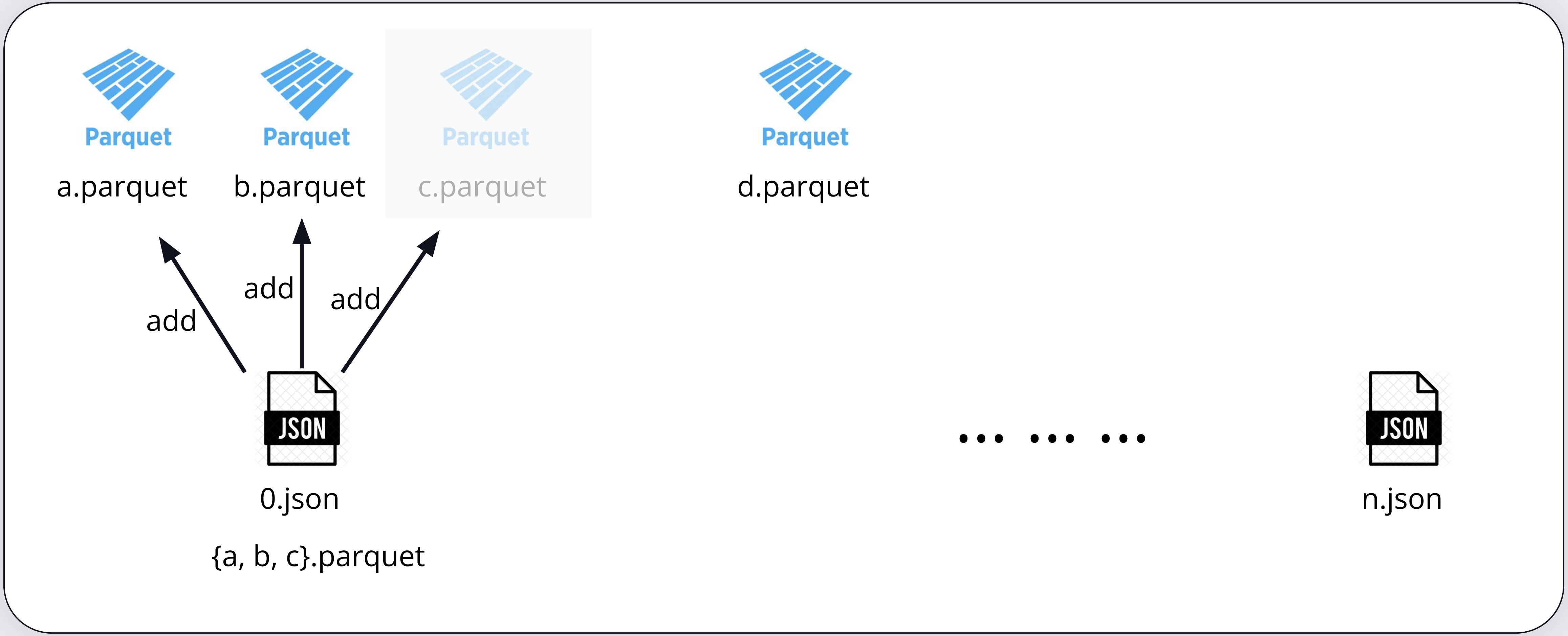
Traditional data warehouse

a.parquet b.parquet c.parquet

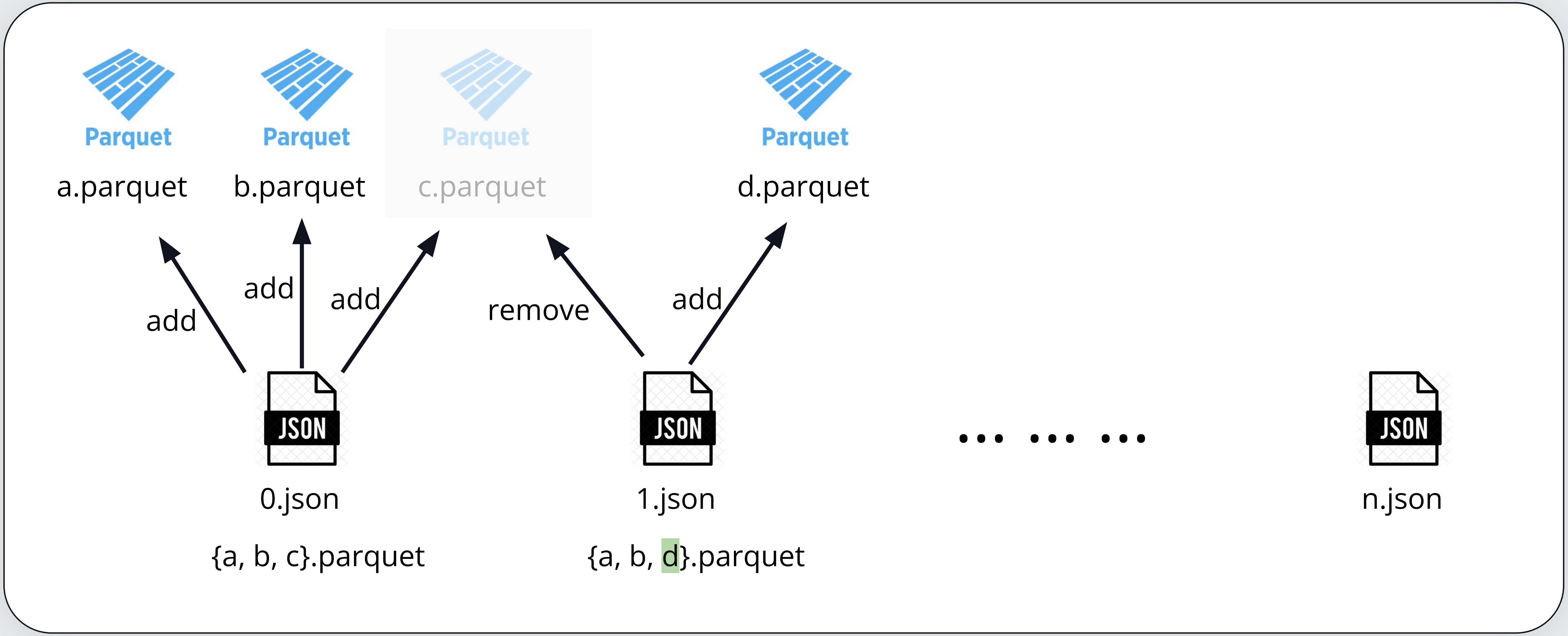
Delta Lake's transaction log



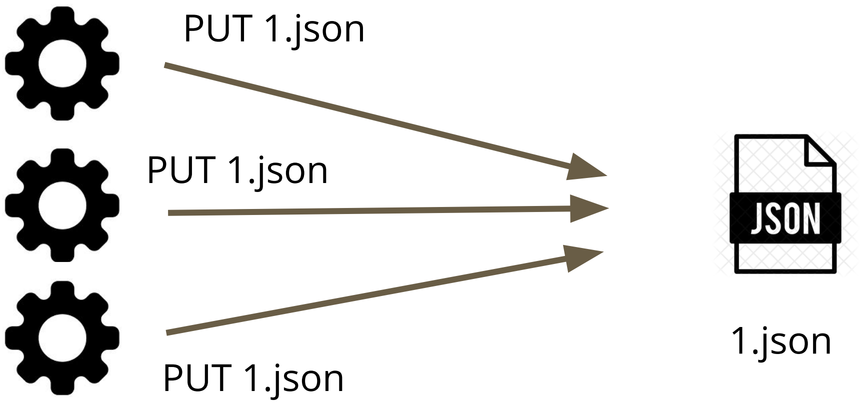
Delta Lake’s transaction log



Delta Lake's transaction log

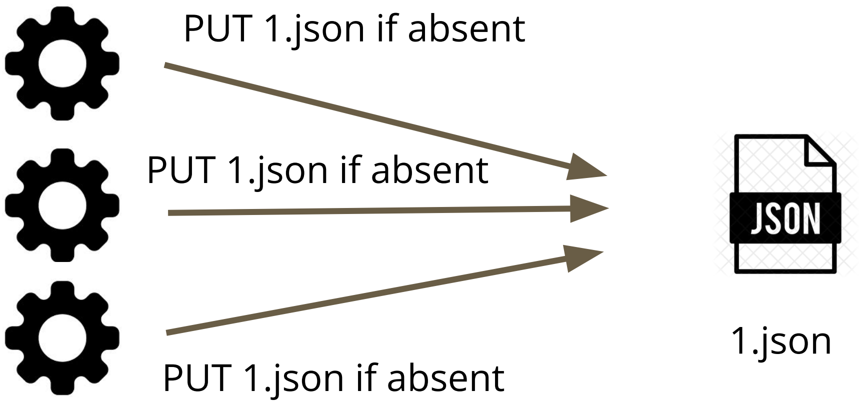


Concurrent table writes



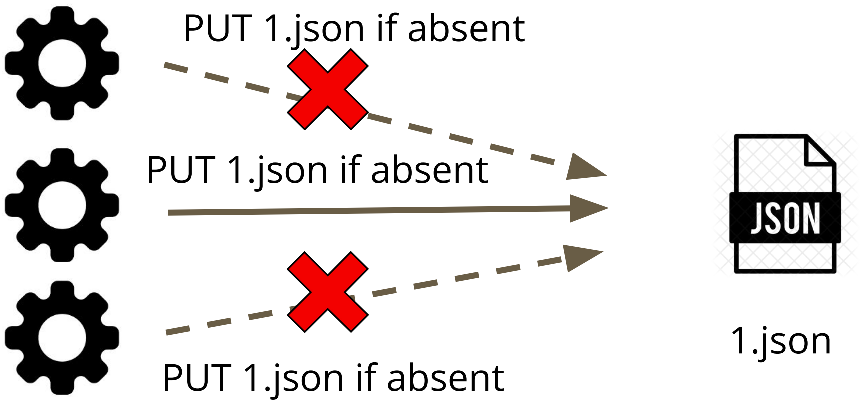
Concurrent table writes

Optimistic concurrency control



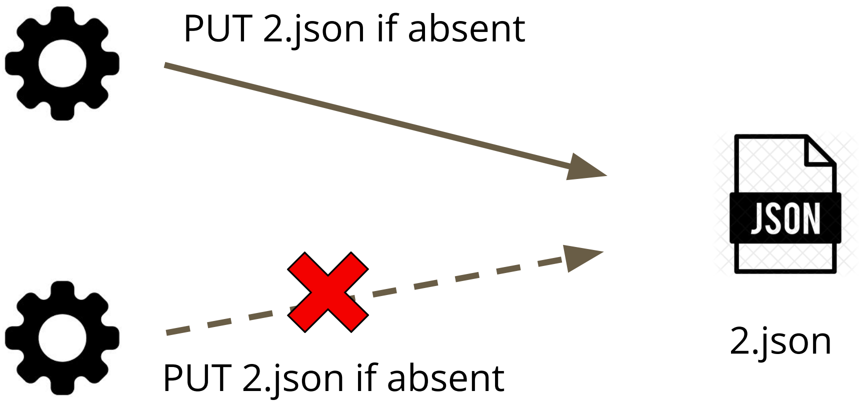
Concurrent table writes

Optimistic concurrency control



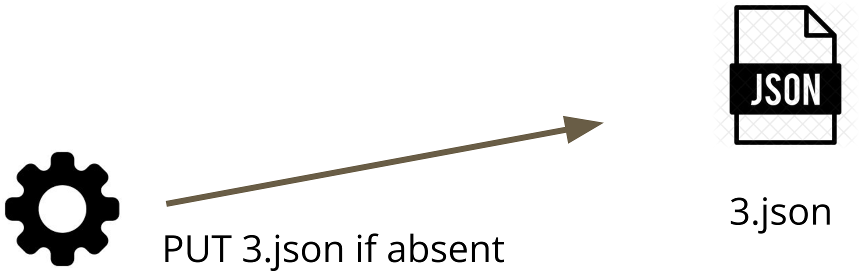
Concurrent table writes

Optimistic concurrency control



Concurrent table writes

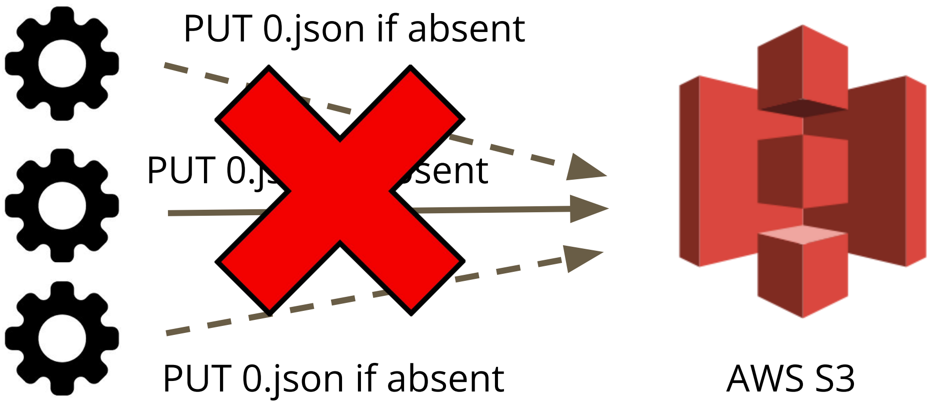
Optimistic concurrency control



Safe concurrent write in S3

S3 limitation

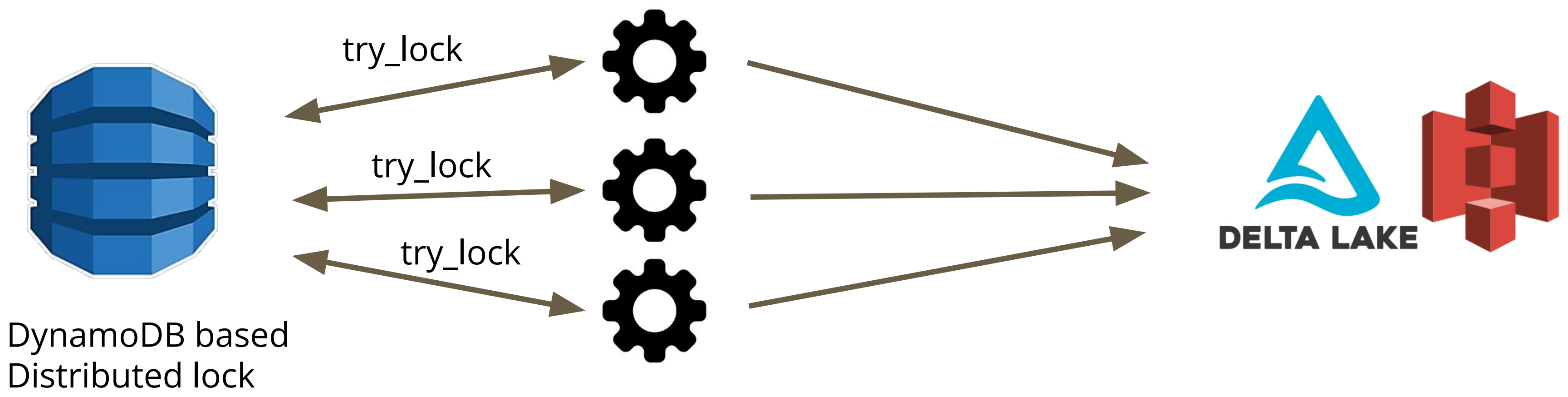
S3 doesn't support PUT if absent operation



Two design directions

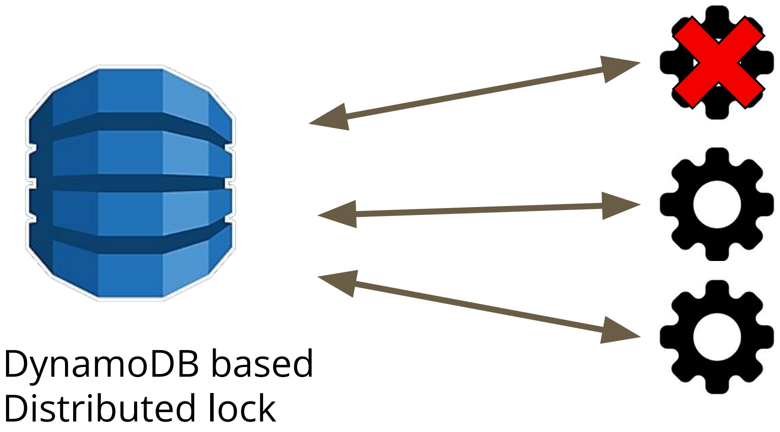
- Use DynamoDB as the log store
  - Less complexity, a lot easier to implement
- Use DynamoDB to implement put if absent for S3
  - Fully transparent and compatible with other Delta Lake readers

Let’s use a distributed lock?



DynamoDB as a distributed lock

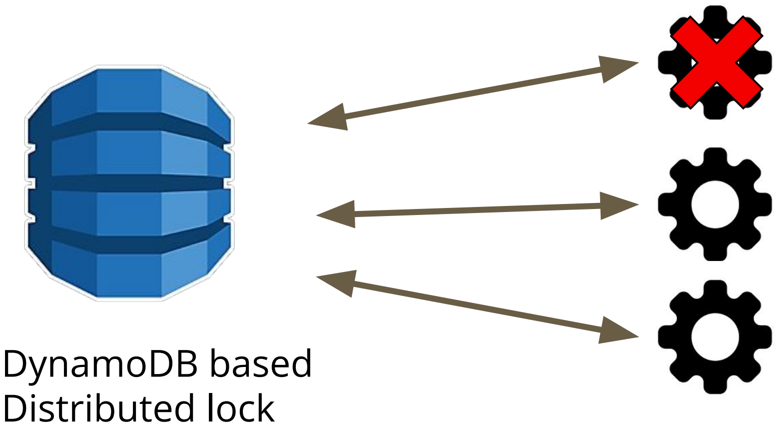
How hard could it possibly be?



What if one of the writers **crashed** while holding the lock?

Distributed lock with an expiration

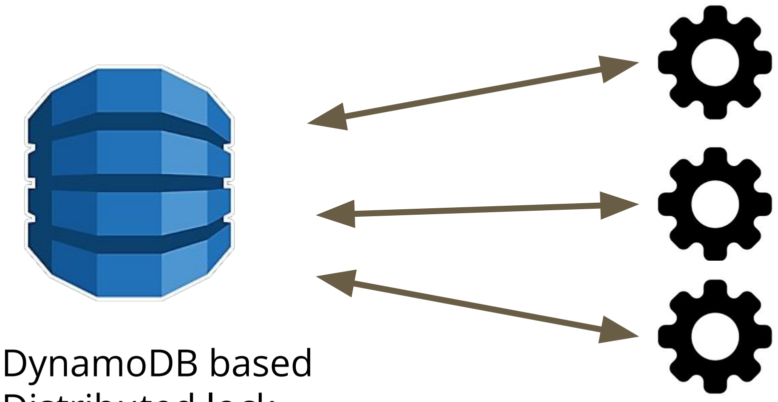
How hard could it possibly be?



If one writer crashed while holding the lock,
it will **eventually expire so another writer can acquire it**.

Distributed lock with an expiration

How hard could it possibly be?

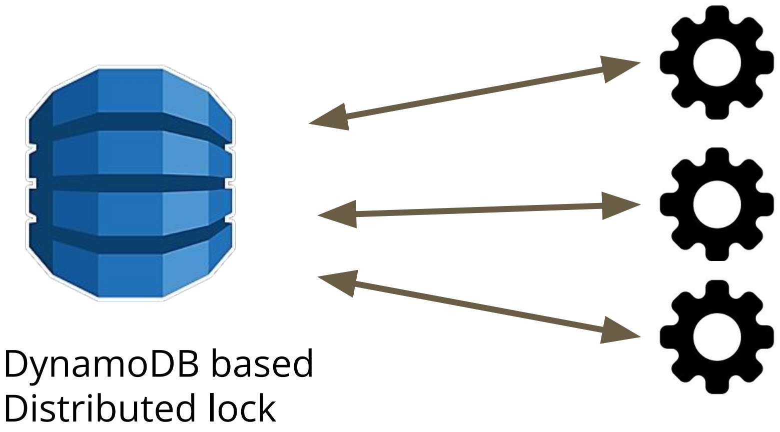


DynamoDB based
Distributed lock

Expirable distributed lock is a scam!

Distributed lock with an expiration

How hard could it possibly be?



If one writer **paused** while holding the lock,
it will **eventually expire so another writer will acquire it**.

Distributed lock with an expiration

How hard could it possibly be?

Time	Writer A	Writer B
T0	Acquired lock	
T1	**Paused**	
T2	**Lock expired**	
T3		Acquired expired lock
T4	Resumed	
T5	I have the lock!	I have the lock!

If one writer **paused** while holding the lock,
it will **eventually expire so another writer will acquire it**.

PUT if absent using repairs

- Write commit to a temp S3 location
  - Copy from s3://table/{uuid}.json to s3://table/1.json is idempotent
- Atomically acquire the lock with recorded S3 copy operation
- Expired lock needs to be repaired by another writer

PUT if absent using repairs

Time	Writer A	Writer B
T0	Acquired lock with “**copy uuid_a.json to 1.json**”	
T1	Paused	
T2	Lock expired	
T3		Acquired expired lock with “**copy uuid_b.json to 1.json**”
T4		Repair lock by executing “**copy uuid_a.json to 1.json**”
T5	Resumed	
T6	**copy uuid_a.json to 1.json**	
T7	Release the lock	
T8		Release the lock
T9		Retry acquiring the lock with “**copy uuid_b.json to 2.json**”

PUT if absent using repairs

- Full design discussion available at https://github.com/delta-io/delta-rs/discussions/89
- Dynamodb based lock implemented from scratch in Rust: https://crates.io/crates/dynamodb_lock
  - Shout out to my ex-colleague Mykhailo Osypov (@mosyp)

PUT if absent using repairs

What if we have more than two writers?

Time	Writer A	Writer B	Writer C	Writer D	Writer …
T0	Acquired lock				
T1	Paused				
T2	Lock expired				
T3		Acquired expired lock			
T4		Repair lock by executing			
T5	Resumed				
T6	Do work				
T7	Release the lock				
T8		Release the lock			

Enter formal verification

Informal verifications are everywhere

- Static type checks
  - Assert desired properties by enforcing type constraints
- Unit tests
  - Assert desired properties by matching inputs with expected outputs

Formal verification approaches

- Model checking
  - Exhaustive state exploration
- Deductive reasoning
  - Ensure conformance of system specifications through mathematical proofs

Model checker state exploration example

Possible state 1

Time	Writer A	Writer B
T0	Acquired lock	
T1	Paused	
T2	**Lock expired**	
T3		**Tried to acquire lock**
T4	…	…

Writer A:
- Paused
- Held an expired lock

Writer B:
- Acquired an expired lock

Model checker state exploration example

Possible state 2

Time	Writer A	Writer B
T0	Acquired lock	
T1	Paused	
T2		**Tried to acquire lock**
T3	**Locked expired**	
T4	…	…

Writer A:

- Paused
- Held an expired lock

Writer B:

- Failed to acquire lock

Model checker industrial use

- TLA+
  - AWS S3
  - Azure cosmos DB
  - Intel multi-processor cache-coherence protocols
  - See https://lamport.azurewebsites.net/tla/industrial-use.html

Shortcomings of TLA+

- Steep learning curve
- Rough tooling
- Separation of proof and implementation

Intro to stateright

Stateright

- Model checker as a Rust library
  - Comparison with TLA+: https://www.stateright.rs/comparison-with-tlaplus.html
- Homepage: https://www.stateright.rs/

Stateright

Specify systems as state machines

```
pub trait Model: Sized {
    type State;
    type Action;
    fn init_states(&self) -> Vec<Self::State>ⓘ;
    fn actions(&self, state: &Self::State, actions: &mut Vec<Self::Action>);
    fn next_state(
        &self,
        last_state: &Self::State,
        action: Self::Action
    ) -> Option<Self::State>;
    fn properties(&self) -> Vec<Property<Self>>ⓘ{ ... }
}
```

Stateright

Specify systems as state machines

- See delta-rs's full stateright proof at https://github.com/delta-io/delta-rs/blob/main/proofs/src/main.rs (about 500 lines of Rust code)

Stateright

High level state machine based actor interface

```
pub trait Actor: Sized {
    type Msg: Clone + Debug + Eq + Hash;
    type State: Clone + Debug + PartialEq + Hash;
    fn on_start(&self, id: Id, o: &mut Out<Self>) -> Self::State;

    fn on_msg(
        &self,
        id: Id,
        state: &mut Cow<'_, Self::State>,
        src: Id,
        msg: Self::Msg,
        o: &mut Out<Self>
    ) { ... }
    fn on_timeout(
        &self,
        id: Id,
        state: &mut Cow<'_, Self::State>,
        o: &mut Out<Self>
    ) { ... }
}
```

Is formal verification worth it?

Yes!

- The exercise of formalization forces you to really think through the design step by step
    - We discovered a couple of design bugs during this process
- Automated model checking caught one correctness bug that all of us missed: https://github.com/delta-io/delta-rs/pull/540#issue-1097376239

Stateright Explorer

API Docs Book Crate Source

Status

- **Model:** AtomicRenameSys
- **States:** 287,395
- **Unique States:** 116,479
- **Progress:** Done

Properties

- ⚠️ **Counterexample found:** `Always no overwrite`
- ✅ **Safety holds:** `Always no unexpected rename`
- ✅ **Safety holds:** `Always not retry on successful rename`
- ✅ **Liveness holds:** `Eventually all writer clean shutdown`
- ✅ **Liveness holds:** `Eventually all source objects are purged`
- ⚠️ **Counterexample found:** `Eventually all renames are performed`
- ✅ **Example found:** `Sometimes lock contention`

Path of Actions

1. `Pre-init`
2. `Init 0`
3. `TryAcquireLock(0)`
4. `NewVersionObjectCheckExists(0)`
5. `TryAcquireLock(1)`
6. `TryAcquireLock(1)`
7. `TryAcquireLock(1)`
8. `TryAcquireLock(1)`
9. `RepairObjectCheckExists(1)`
10. `RepairObjectCopy(1)`
11. `UpdateLockData(1)`
12. `NewVersionObjectCheckExists(1)`
13. `NewVersionObjectCopy(0)`
14. `NewVersionObjectCopy(1)`

Next Action Choices

- `OldVersionObjectDelete(0)`
- `OldVersionObjectDelete(1)`
- `TryAcquireLock(2)`

Current State

☐ Complete State? ☐ Compact?

```
AtomicRenameState {
    writer_ctx: [
        WriterContext {
            state: NewVersionObjectCopied,
            lock_data: LockData {
                dst: "0",
                src: "writer_0",
            },
            acquired_expired_lock: false,
            released_expired_lock: false,
            rename_err: None,
            target_version: 0,
        },
        WriterContext {
            state: NewVersionObjectCopied,
            lock_data: LockData {
                dst: "0",
                src: "writer_1",
            },
            acquired_expired_lock: false,
            released_expired_lock: false,
            rename_err: None,
            target_version: 0,
        },
        WriterContext {
            state: Init,
            lock_data: LockData {
                dst: "",
                src: "",
            },
            acquired_expired_lock: false,
            released_expired_lock: false,
            rename_err: None,
            target_version: 0,
        },
    ],
    lock: Some(
        GlobalLock {
            data: LockData {
                dst: "0"
```

Building efficient, safe and correct systems

Combining Rust and Formal verification

- Rust compiler guarantees absence of unsafe memory access bugs
  - No more segfaults and race conditions
- Formal verification removes logical bugs*
- Model checking as a library (stateright) to keep proofs and implementations in sync

DATA+AI SUMMIT 2022

Thank you

QP Hou

https://about.houqp.me/