In-Memory Key-Value Store Live Migration with NetMigrate

Zeying Zhu, Yibo Zhao, Alan Zaoxing Liu

UNIVERSITY OF MARYLAND

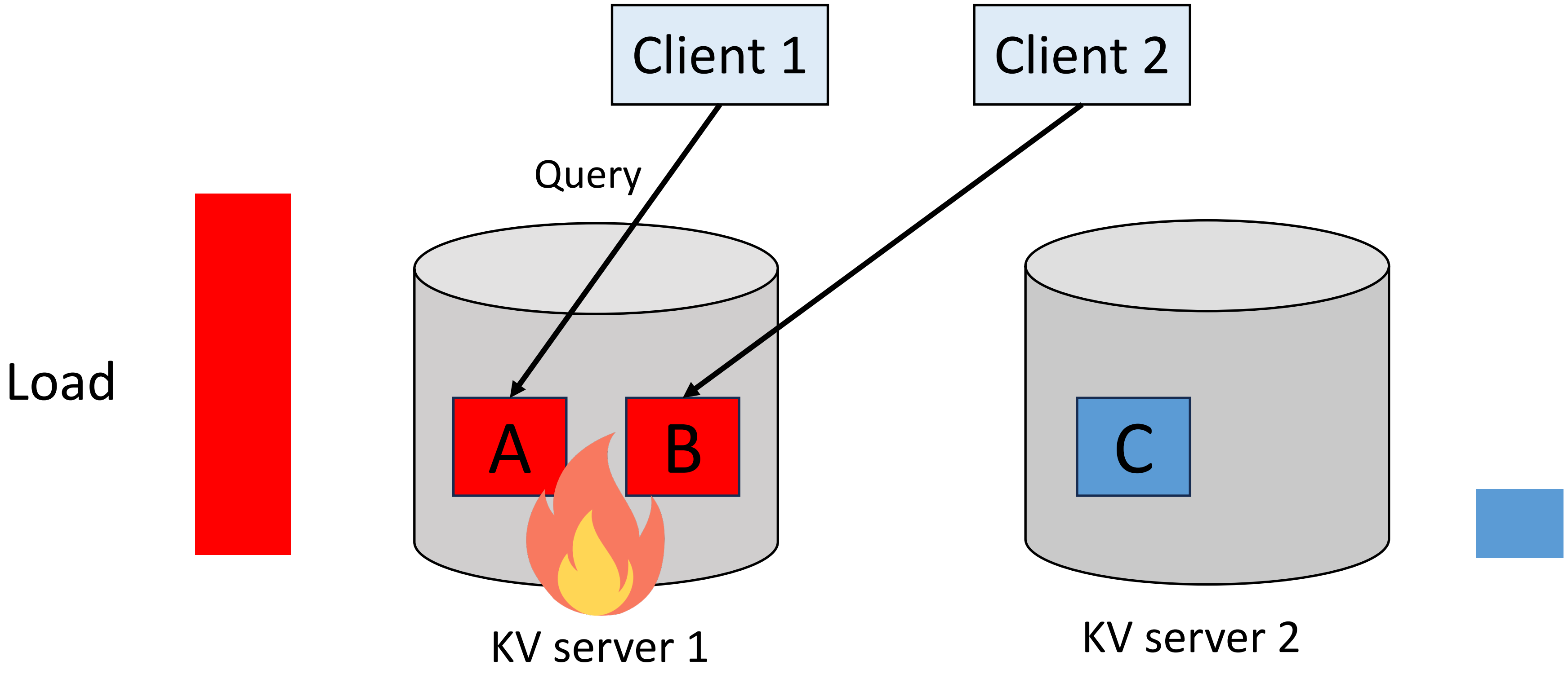
In-Memory Key-Value Stores

- Key-value stores are widely used
  - Feature store of machine learning inference
  - In-memory caching
  - Real-time analytics

- Data amount is large
  - Store billions of records
  - Retrieve millions of records under low latency constraints

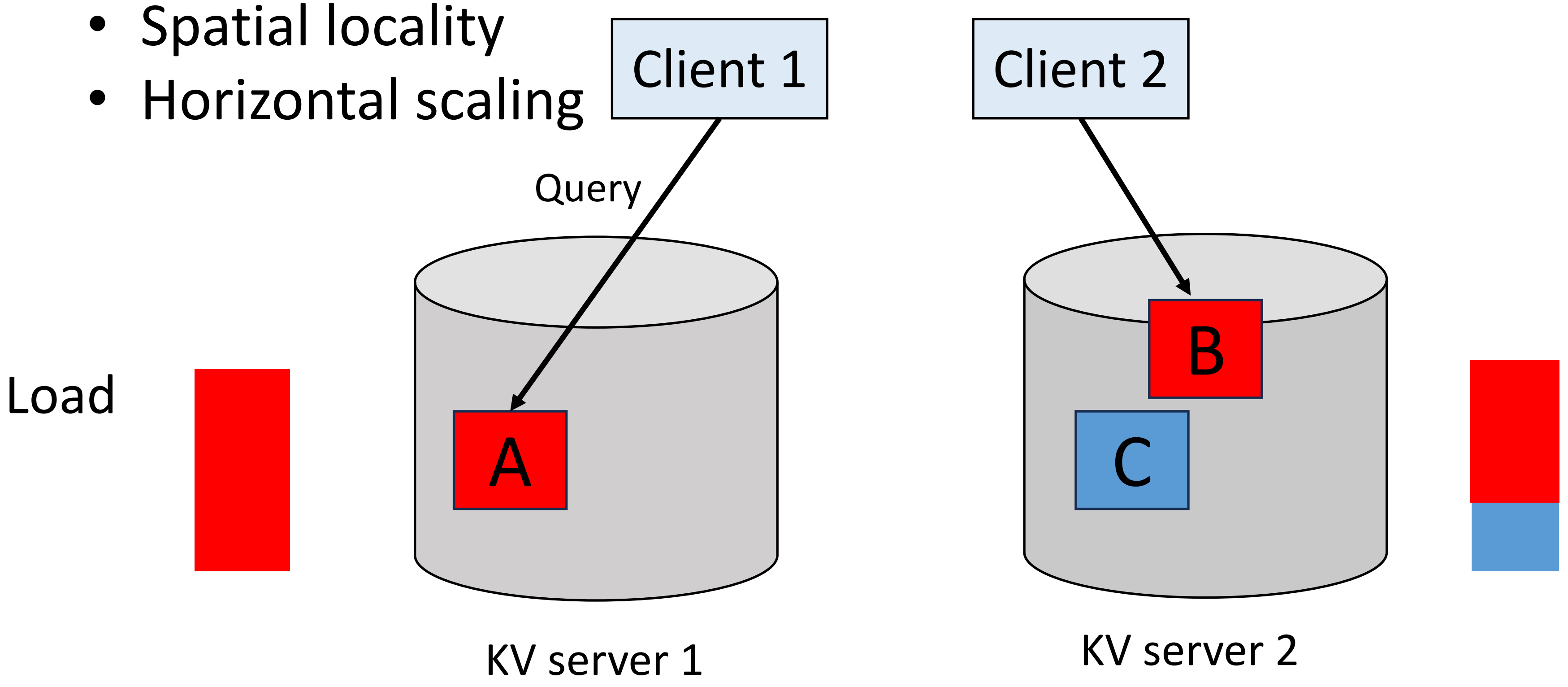
Live Migration is A Key Technique

- No service downtime during key-value shard migration between nodes.
- Why migrate data?
  - Load balancing



Live Migration is A Key Technique

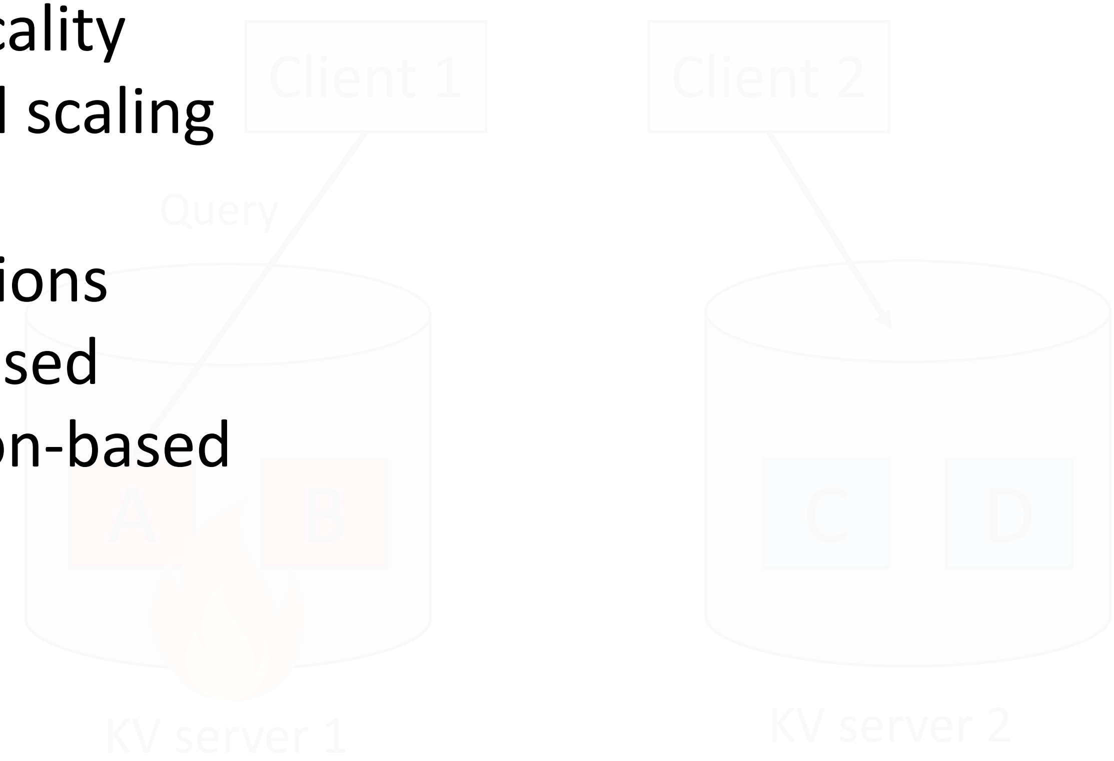
- No service downtime during key-value shard migration between nodes.
- Why migrate data?
  - Load balancing
  - Spatial locality
  - Horizontal scaling



Live Migration is A Key Technique

- No service downtime during key-value shard migration between nodes.
- Why migrate data?
  - Load balancing
  - Spatial locality
  - Horizontal scaling

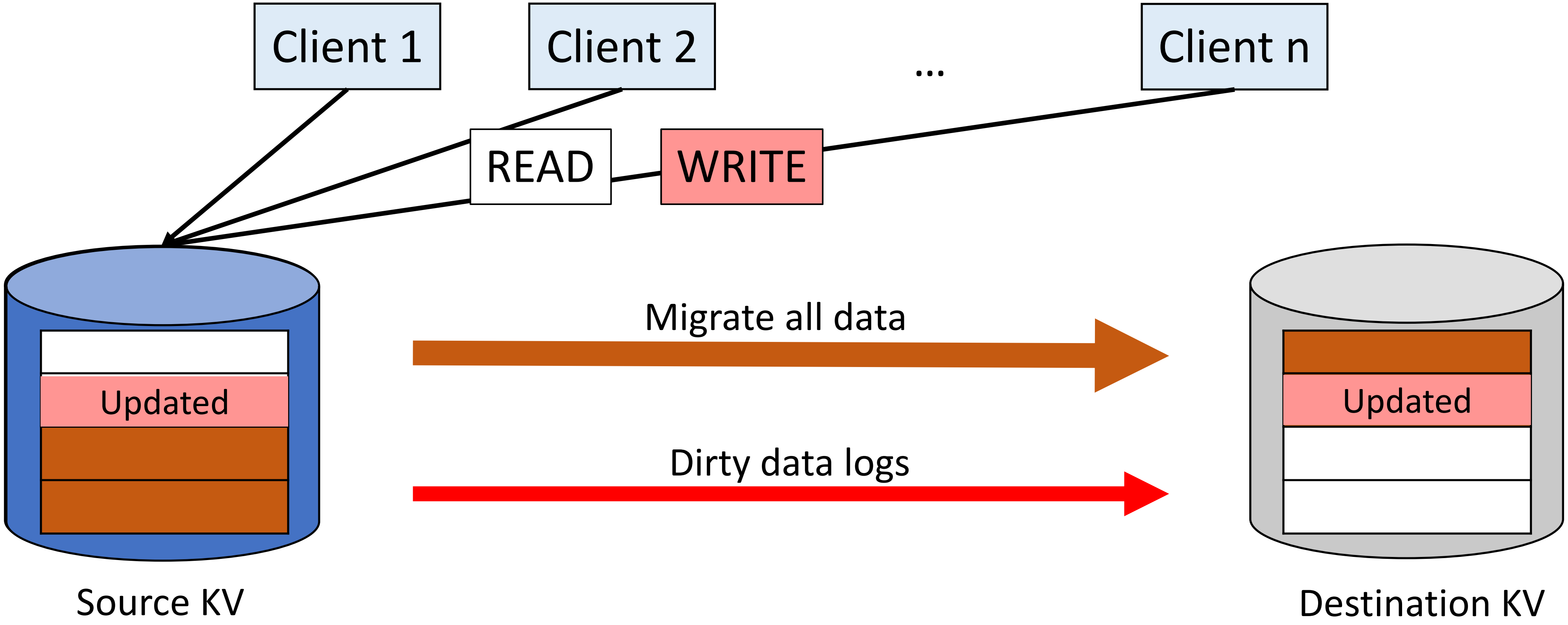
- Existing solutions
  - Source-based
  - Destination-based
  - Hybrid



Source-based Migration

READ: served by source

WRITE: served by source



RAMCloud [TOCS ‘15], Remus [SIGMOD’22]

Source-based Migration

READ: served by source

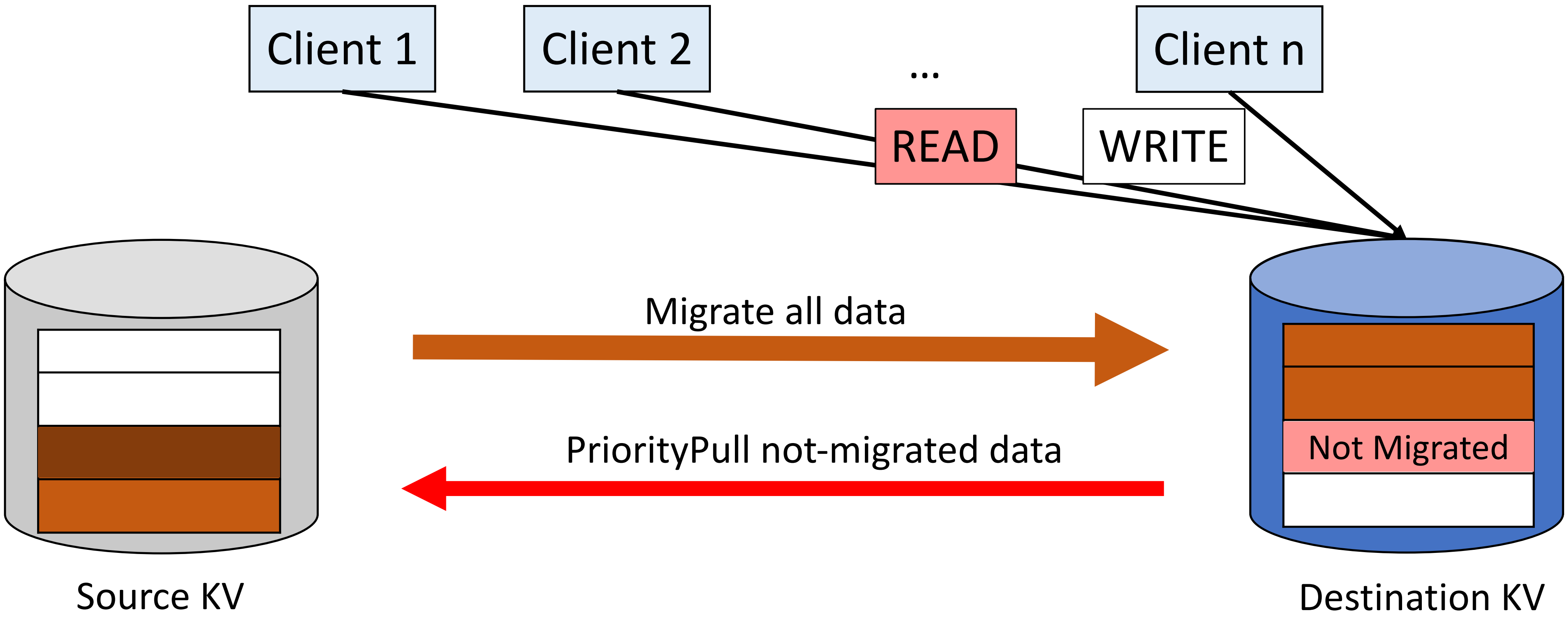
WRITE: served by source

- Low query latency during migration because source node already has the queried data
- Extra dirty data transfer from source to destination
- Downtime when terminating migration

RAMCloud [TOCS ‘15], Remus [SIGMOD’22]

Destination-based Migration

READ: served by destination
WRITE: served by destination



Rocksteady [SOSP'17]

Destination-based Migration

READ: served by destination
WRITE: served by destination

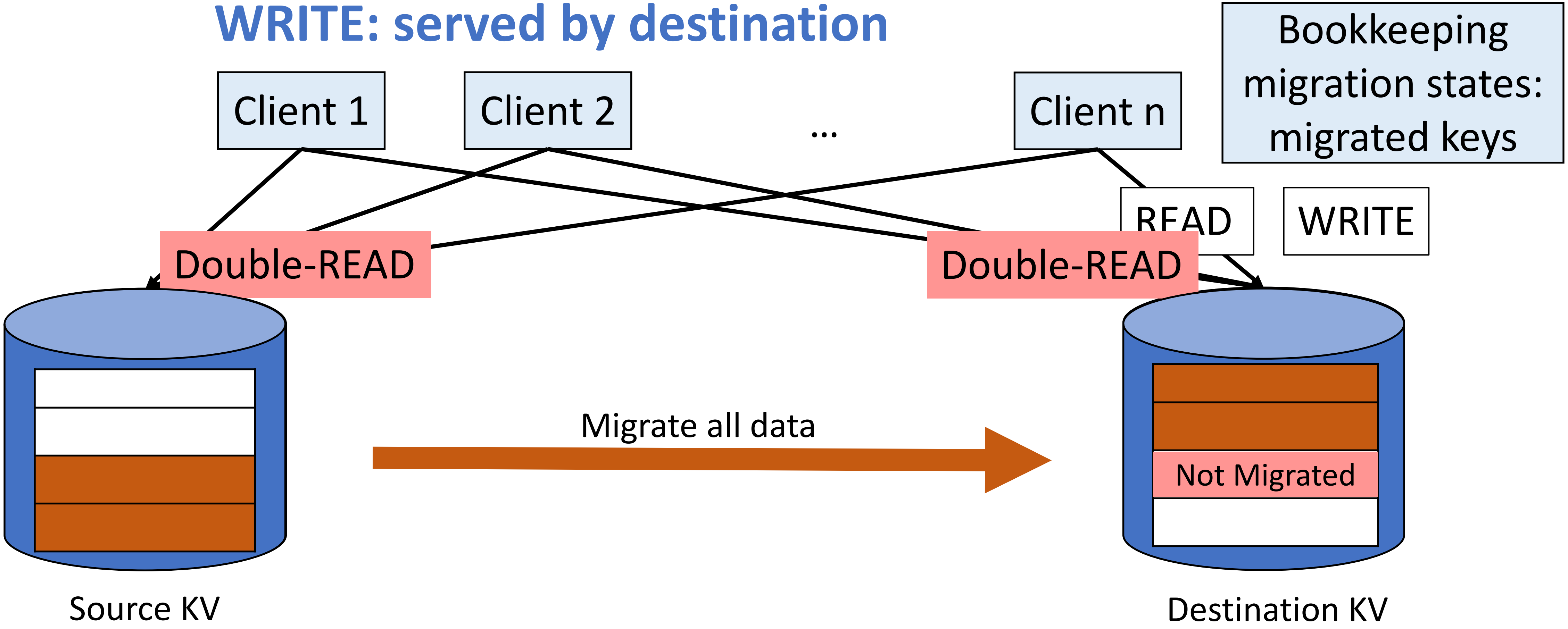
- Quickly shift source node’s pressure, short migration time
- High query latency due to missed data access in the destination (increase 100%~400%)
- Low throughput (drop 66%)

Rocksteady [SOSP’17]

Hybrid Migration

READ: served by both source and destination

WRITE: served by destination



Fulva [SRDS '19]

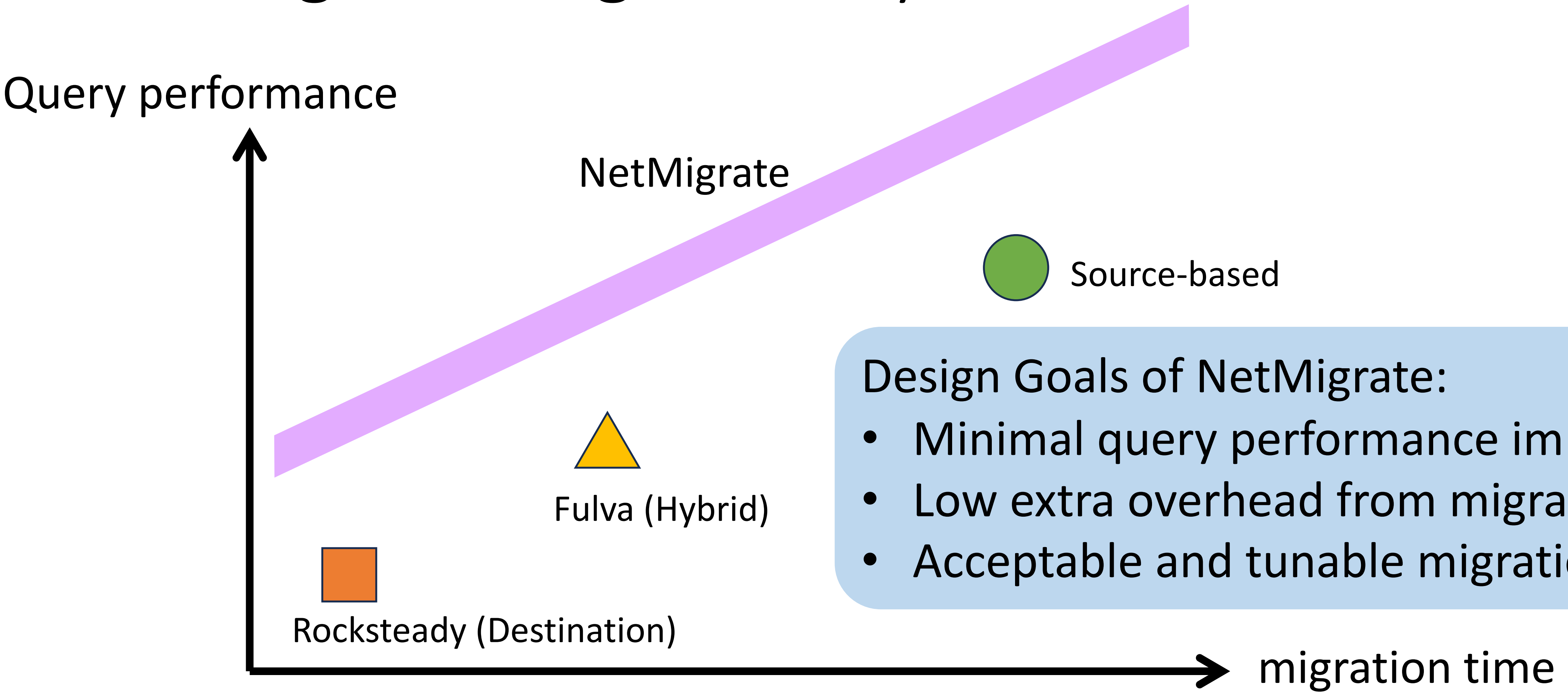
Hybrid Migration

READ: served by both source and destination
WRITE: served by destination

- Leverage both so performance is better when most of data is in the source.
- Double-read incurs large bandwidth overhead between clients and servers (~50%) because of no fine-grained state tracking.

Fulva [SRDS ‘19]

Existing Live Migration Systems



Design Goals of NetMigrate:

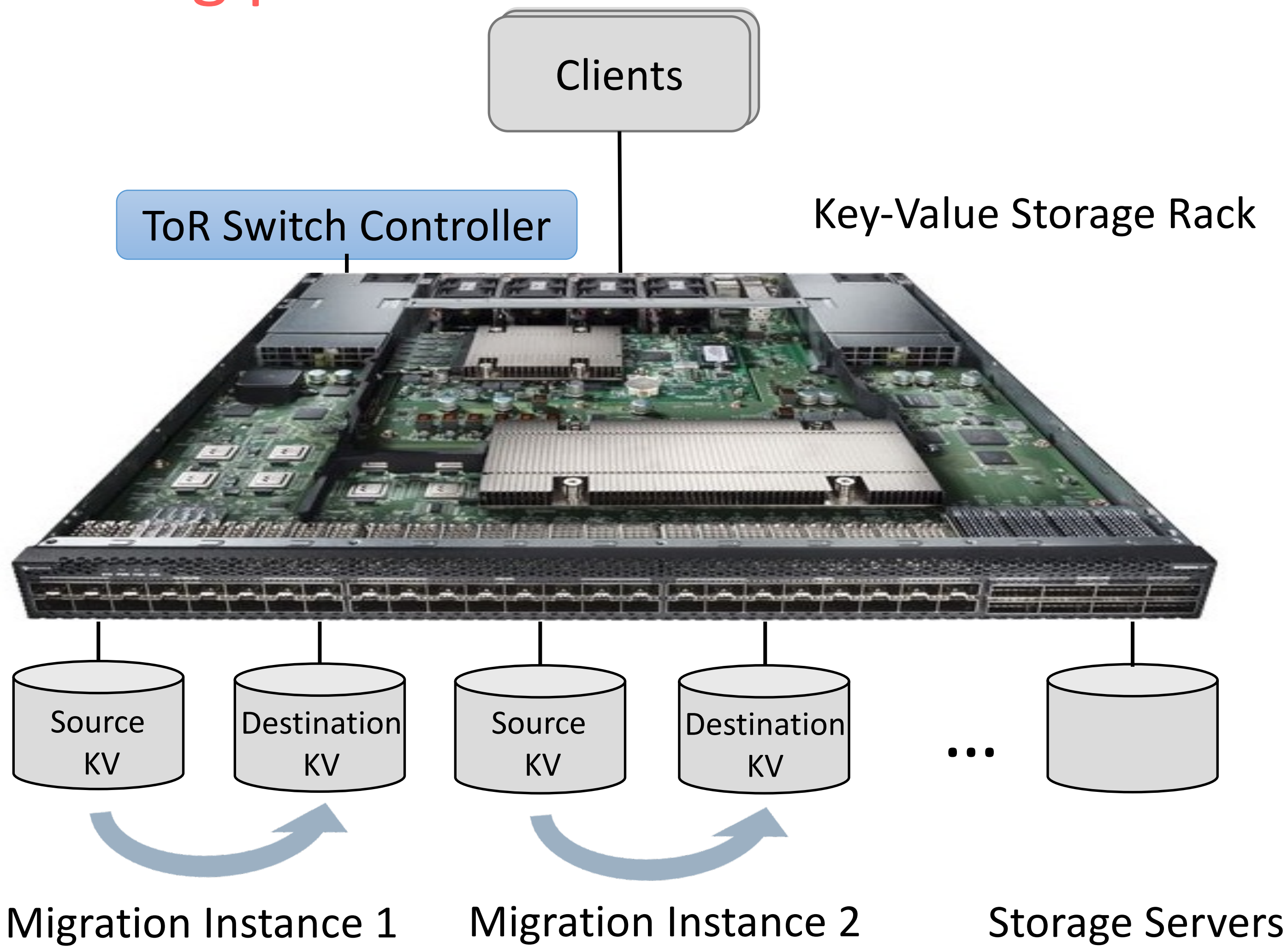
- Minimal query performance impact
- Low extra overhead from migration
- Acceptable and tunable migration time

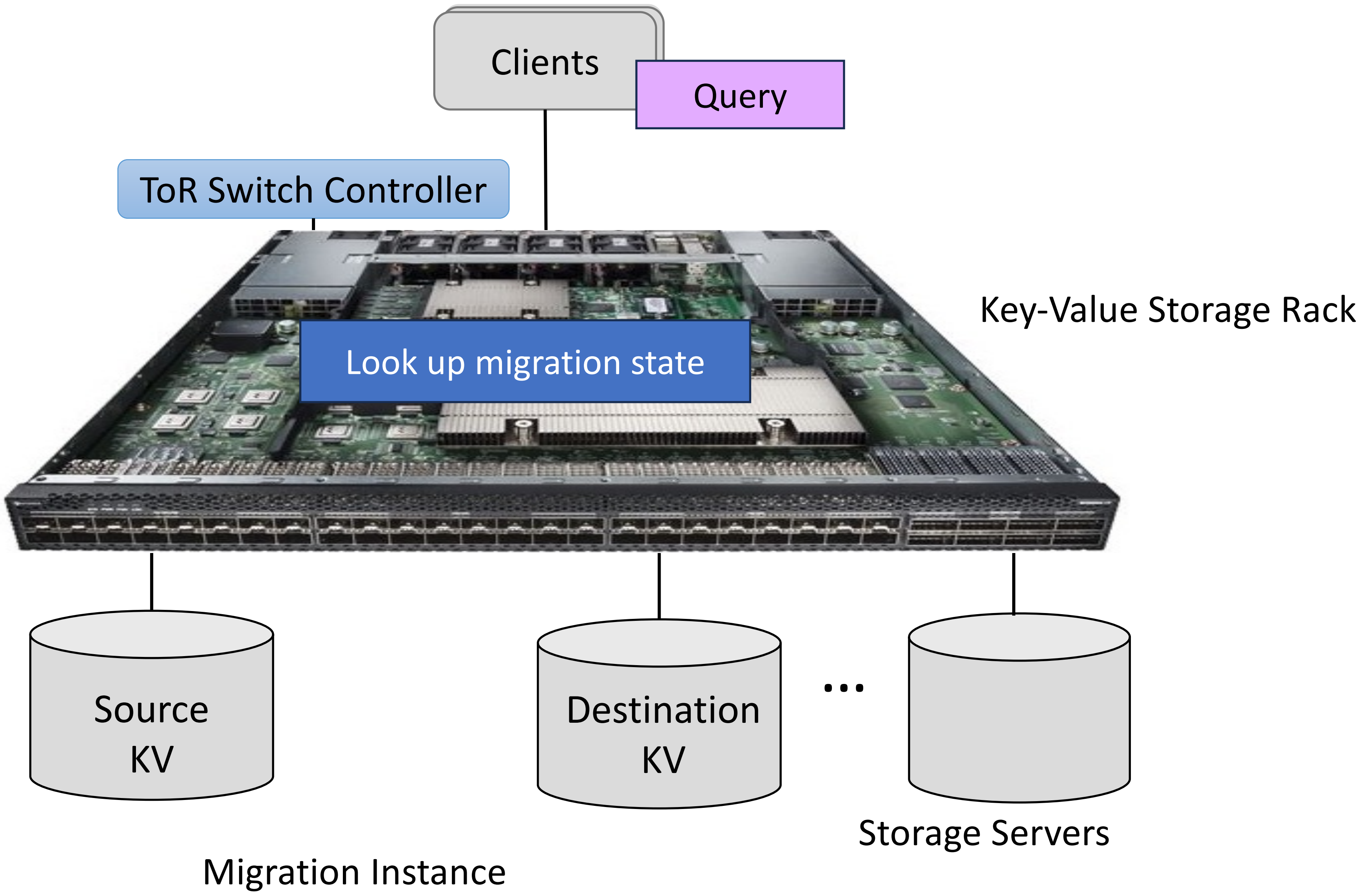
Tradeoff between query performance and migration time

Existing solutions don't know where the data is and pay cost of going to wrong places.

Key Idea: Programmable Top-of-Rack switches to track the migration states.

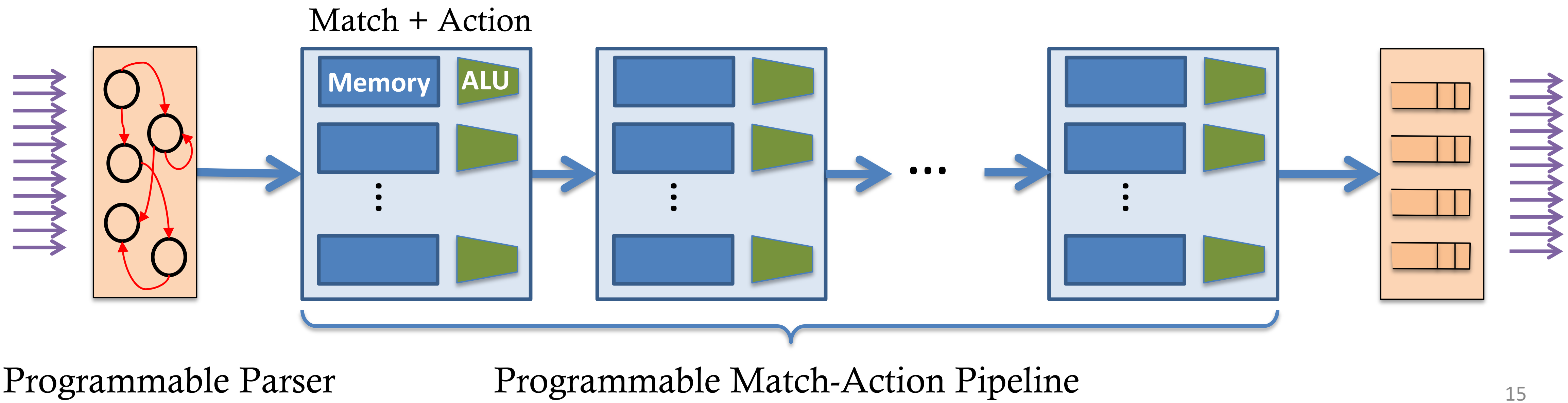
- Centralized view of all data movement
- Real-time information of who owns the data




Clients
Query
ToR Switch Controller
Key-Value Storage Rack
Look up migration state
Source KV
Destination KV
...
Storage Servers
Migration Instance

A Typical Programmable Switch Architecture

- Flexible programmability
  - ➢ Parse, read and update custom fields **at line rate**
- Registers
  - ➢ Store data
- High line-rate packet processing 12.4 Tbps



Design Challenges of NetMigrate

- Challenge #1: How to track fine-grained migration states?
  - On-switch resources are limited (e.g., 64MB SRAM vs. Millions of KV pairs)
- Challenge #2: How to query during migration?
  - Maintain data consistency during migration.
    - Read-After-Write, Write-After-Read, Write-After-Write.
- Challenge #3: How to support dynamic migration policies?

Design Challenges of NetMigrate

- Challenge #1: How to track fine-grained migration states?
  - On-switch resources are limited (e.g., 64MB SRAM vs. Millions of KV pairs)

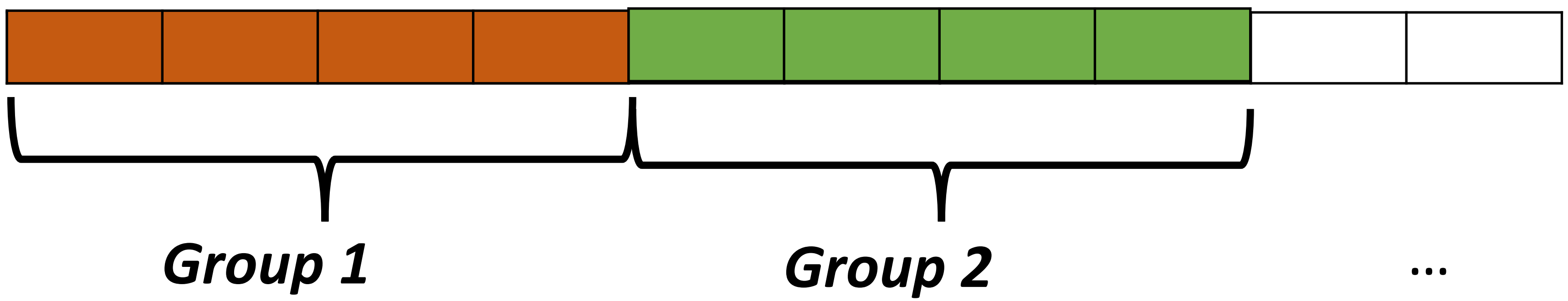
- Challenge #2: How to query during migration?
  - Maintain data consistency during migration.
    - Read-After-Write, Write-After-Read, Write-After-Write.

- Challenge #3: How to support dynamic migration policies?

Shrink Record Granularity for Limited Switch Resources

On-switch resources are limited (e.g., 64MB SRAM vs. Millions of KV pairs)

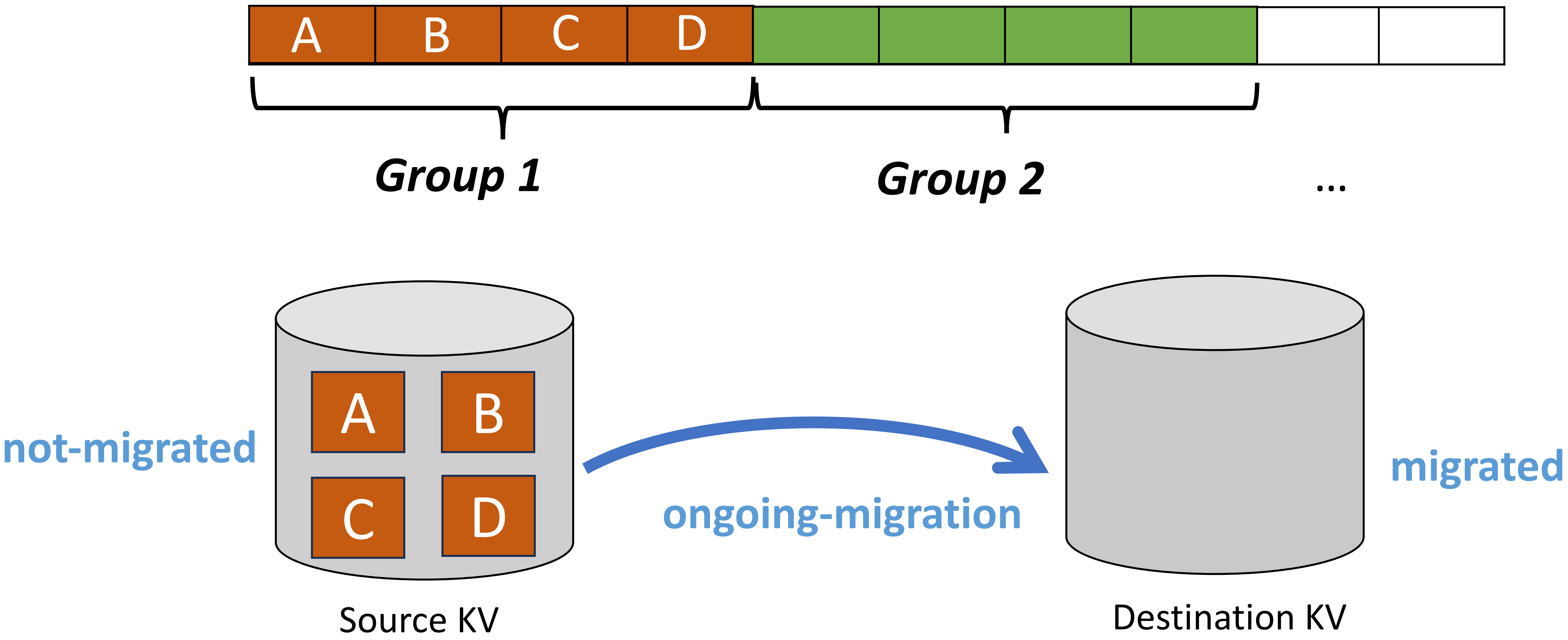
KVS data structure: hash table



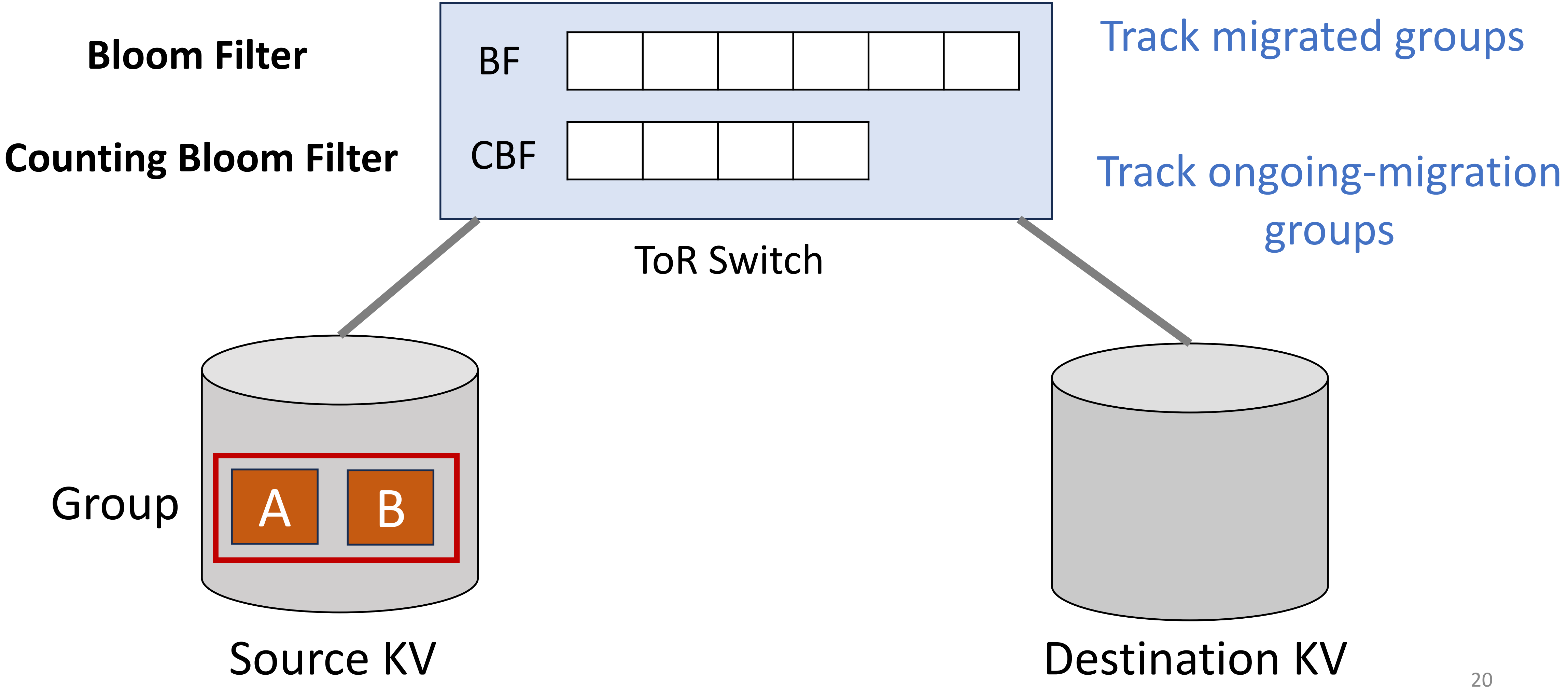
Track migration in a coarser record granularity

Three States to Understand Data Location

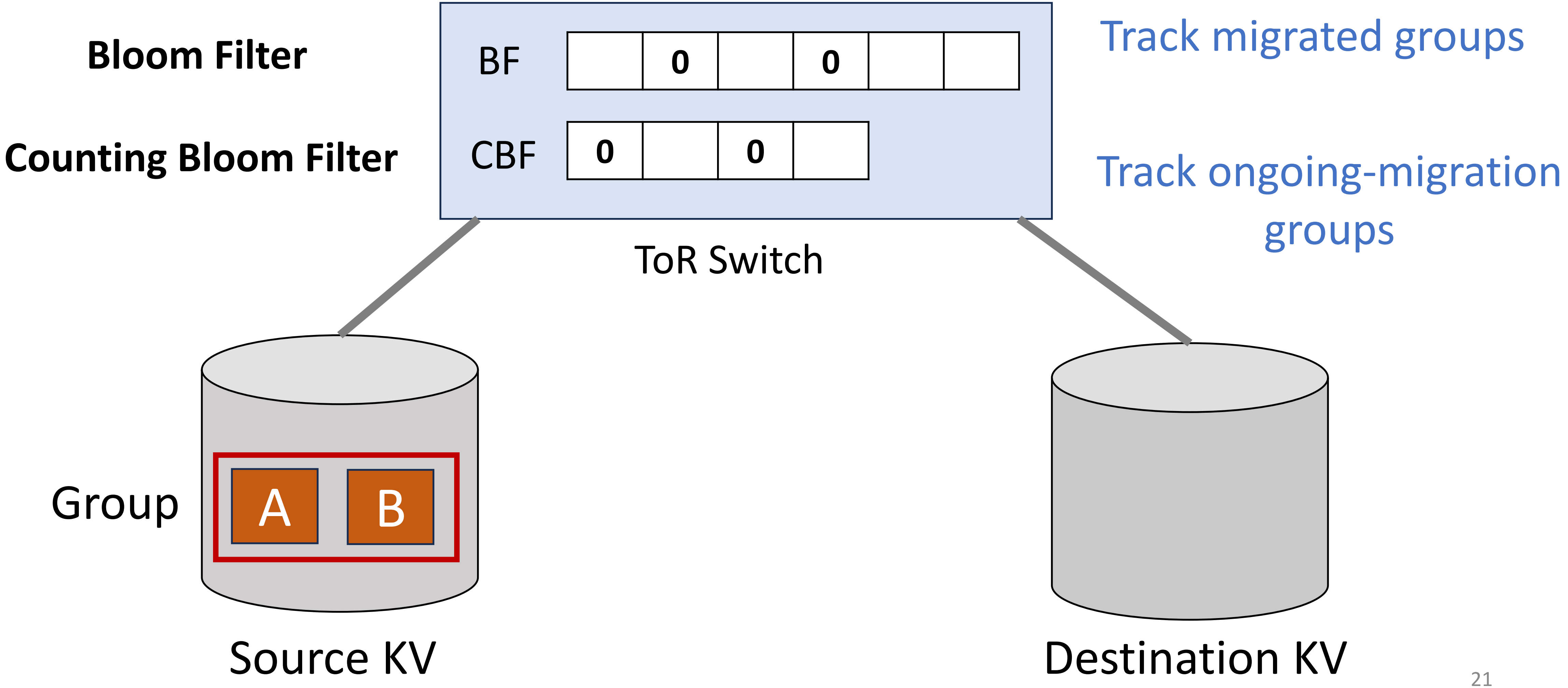
Group migration states: migrated, ongoing-migration, not-migrated



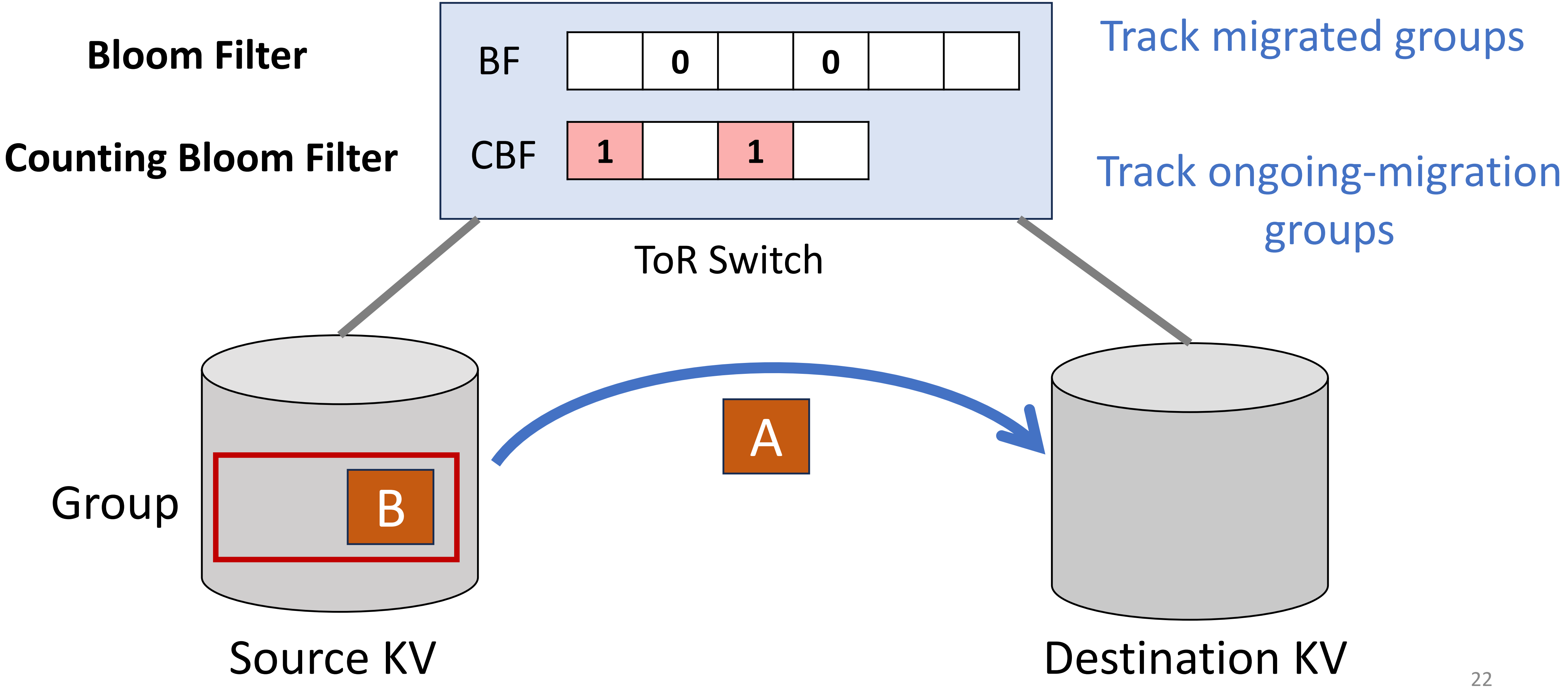
Probabilistic Ownership Tracking



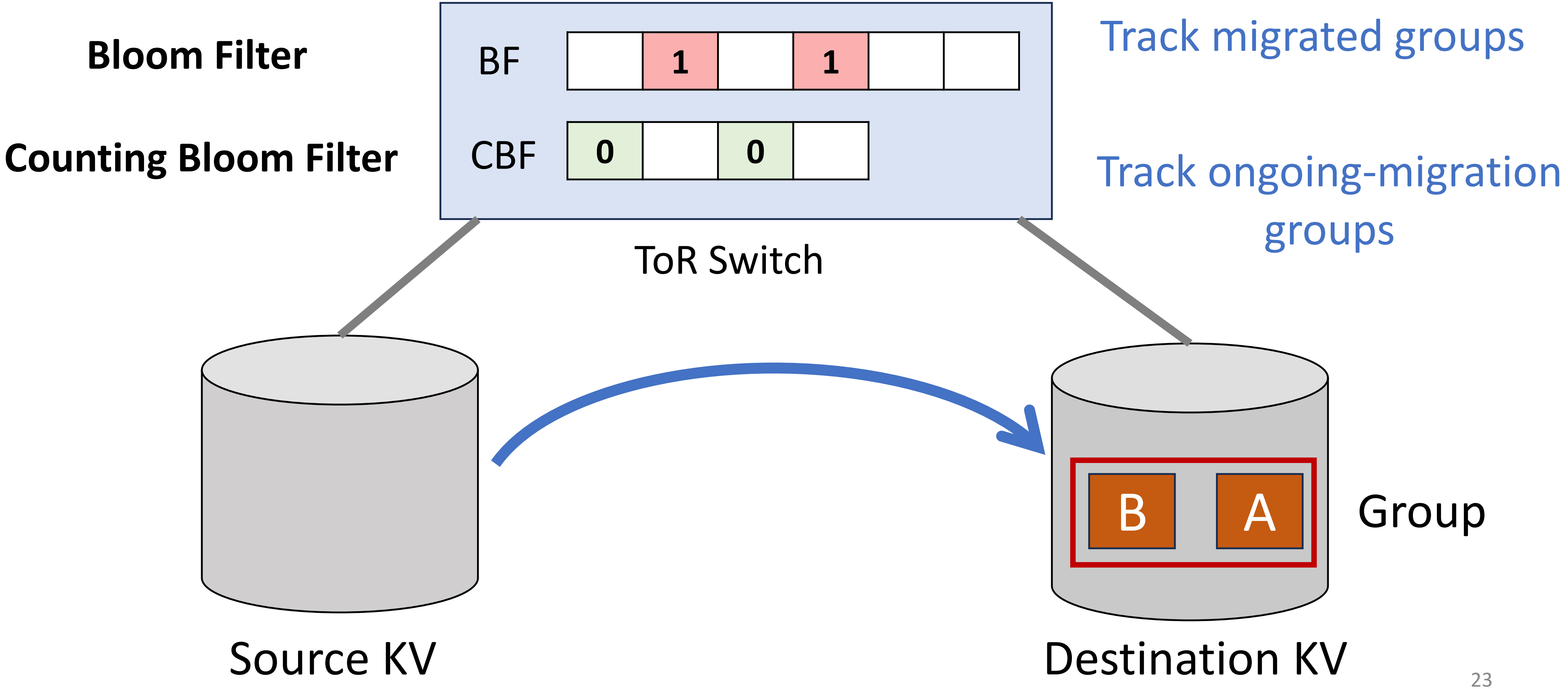
Not Started Migration



Ongoing Migration



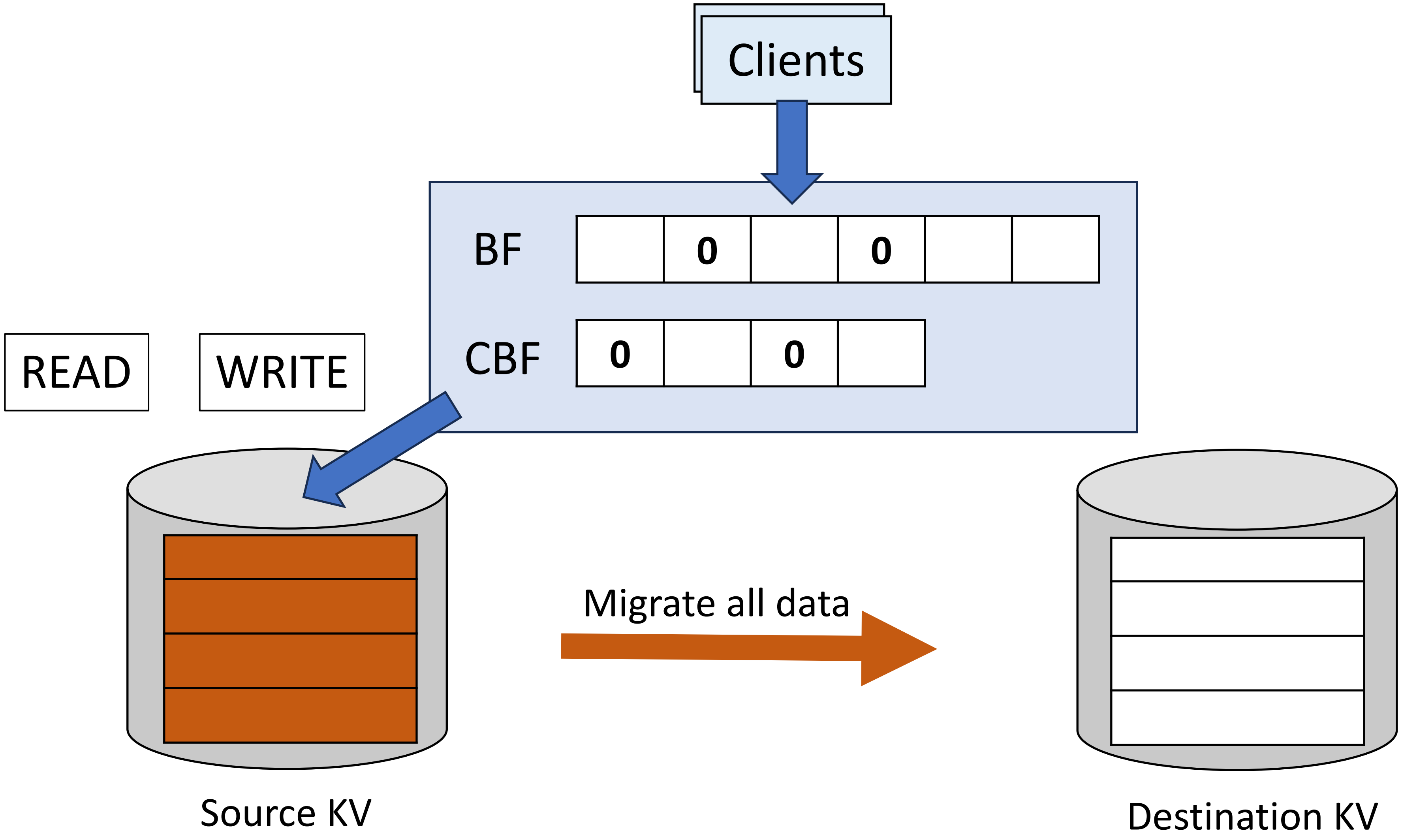
Finished Migration



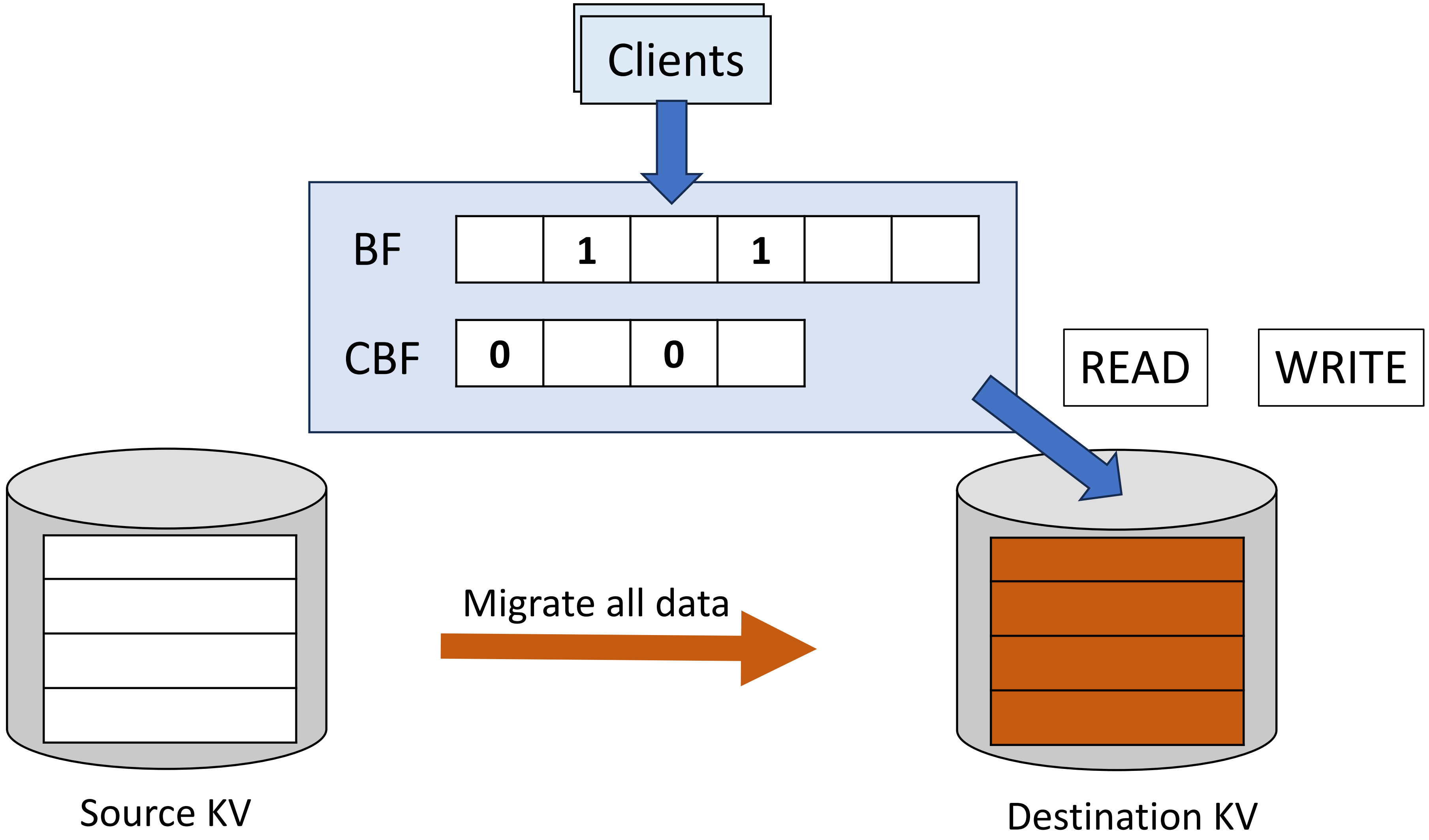
Design Challenges of NetMigrate

- Challenge #1: How to track fine-grained migration states?
  - On-switch resources are limited (e.g., 64MB SRAM vs. Millions of KV pairs)
- Challenge #2: How to query during migration?
  - Maintain data consistency during migration.
    - Read-After-Write, Write-After-Read, Write-After-Write.
- Challenge #3: How to support dynamic migration policies?

Data is Consistent When Not Started Migration

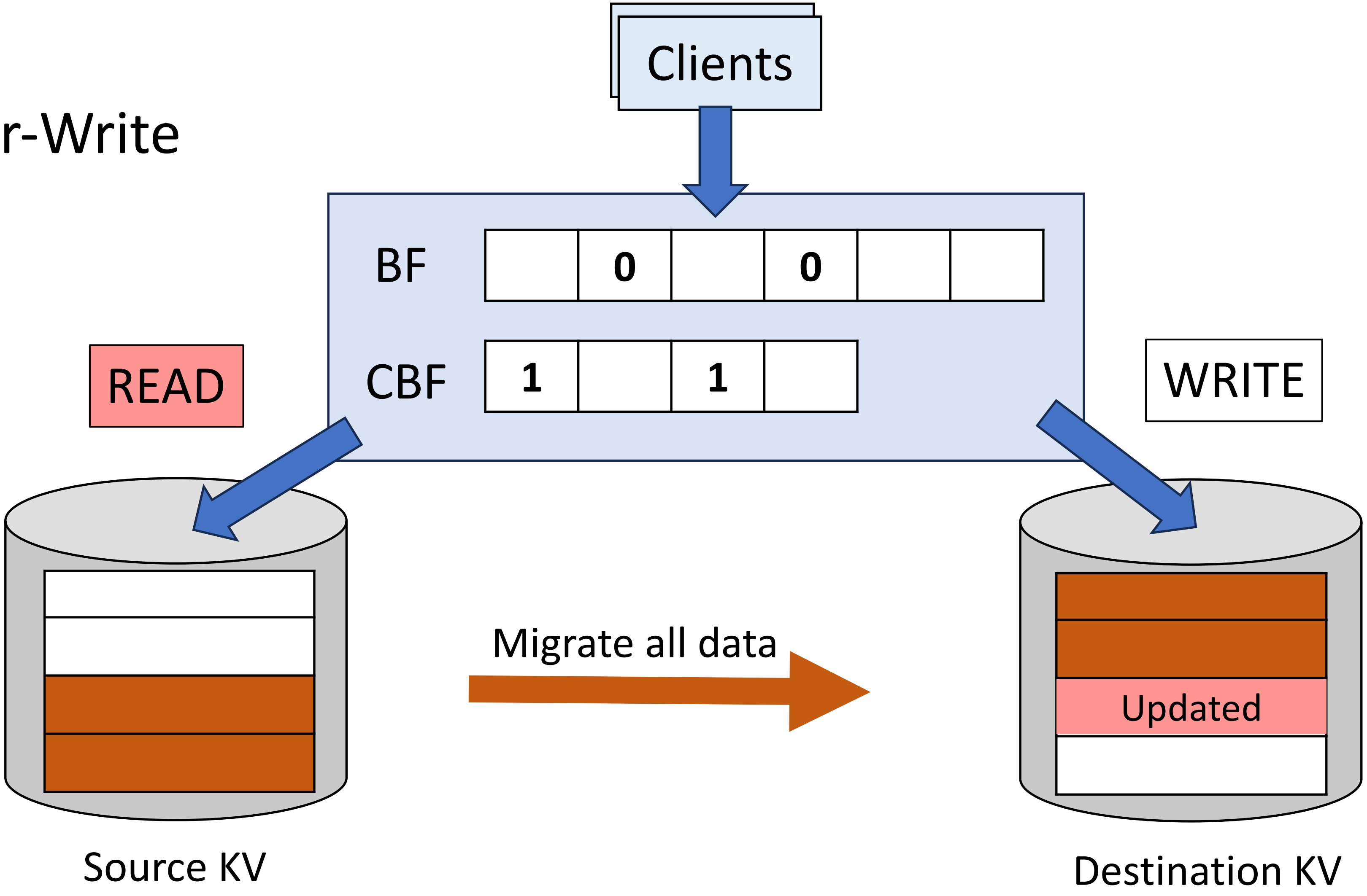


Data is Consistent When Finished Migration



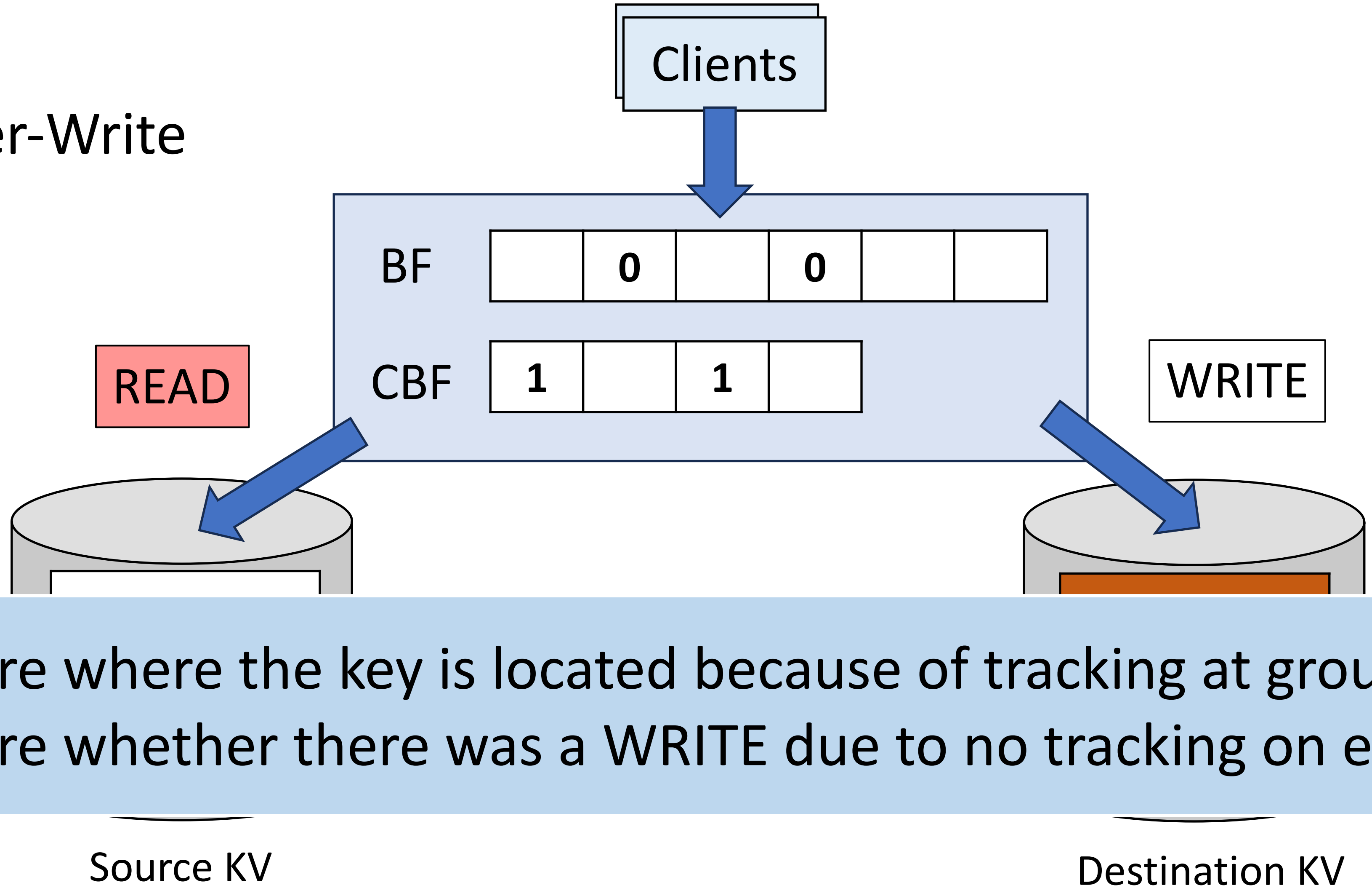
An Inconsistent Example When Ongoing Migration

Read-After-Write



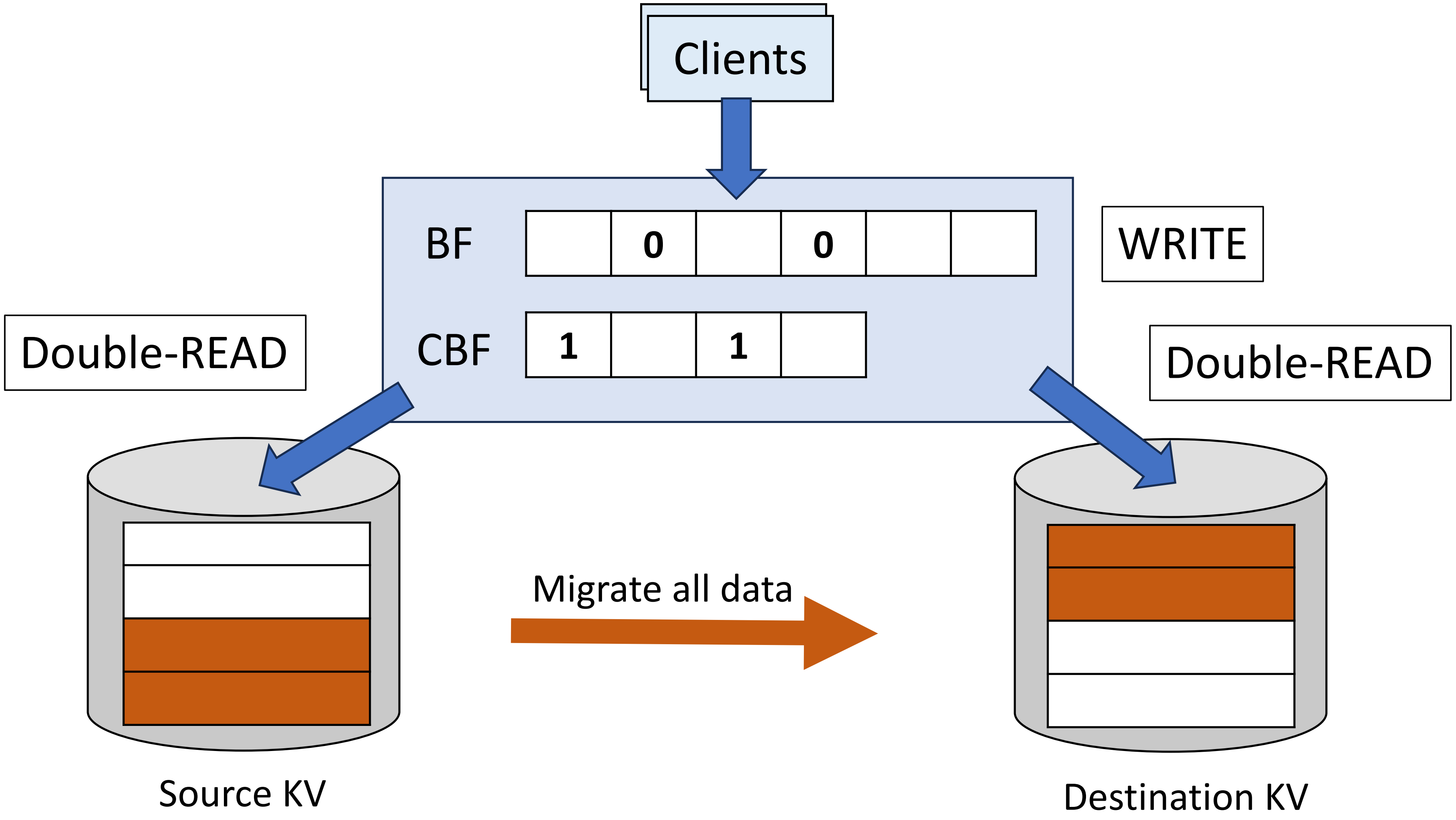
An Inconsistent Example When Ongoing Migration

Read-After-Write

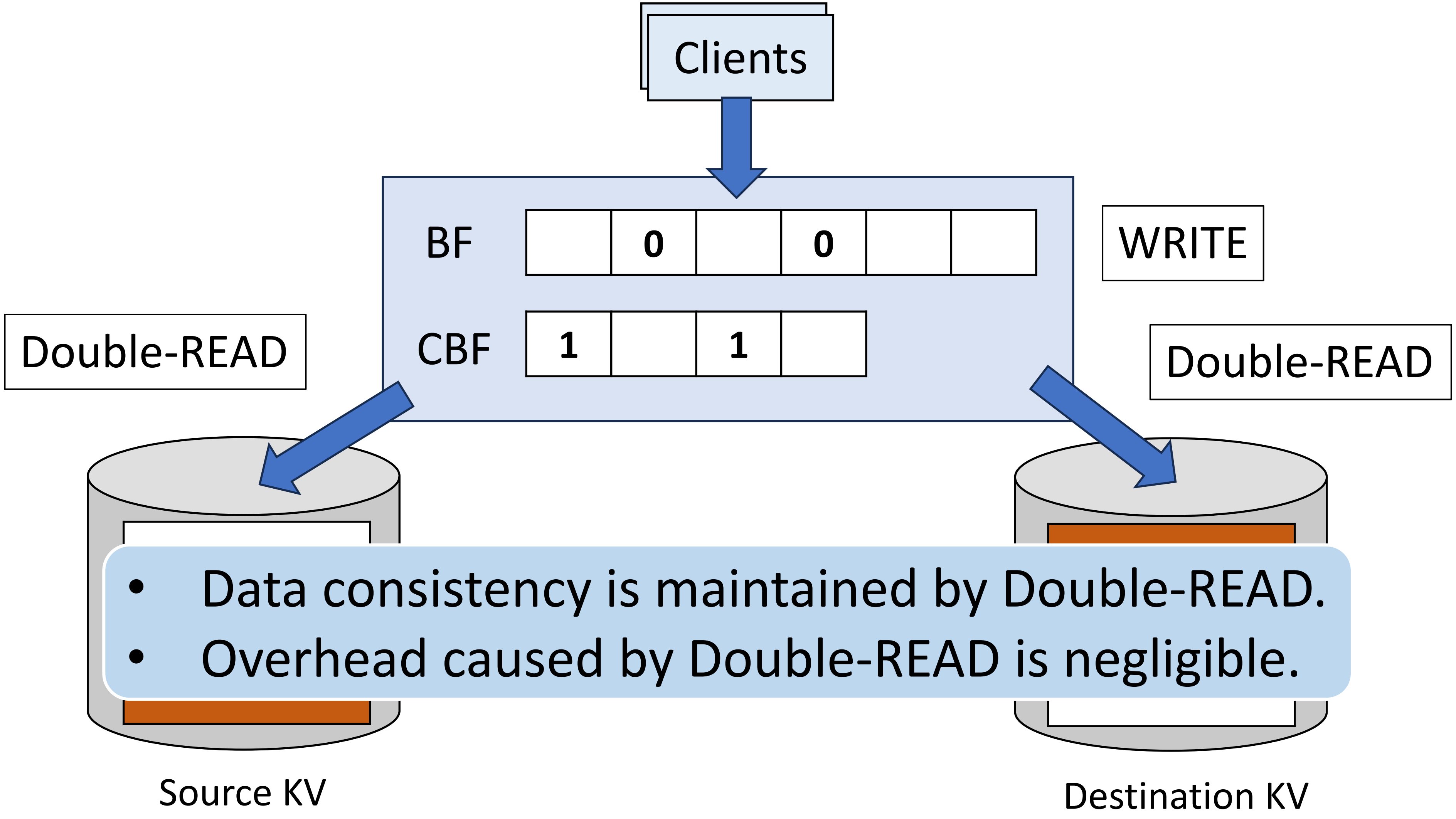


- Not sure where the key is located because of tracking at group level.
- Not sure whether there was a WRITE due to no tracking on every WRITE.

Data is Consistent When Ongoing Migration

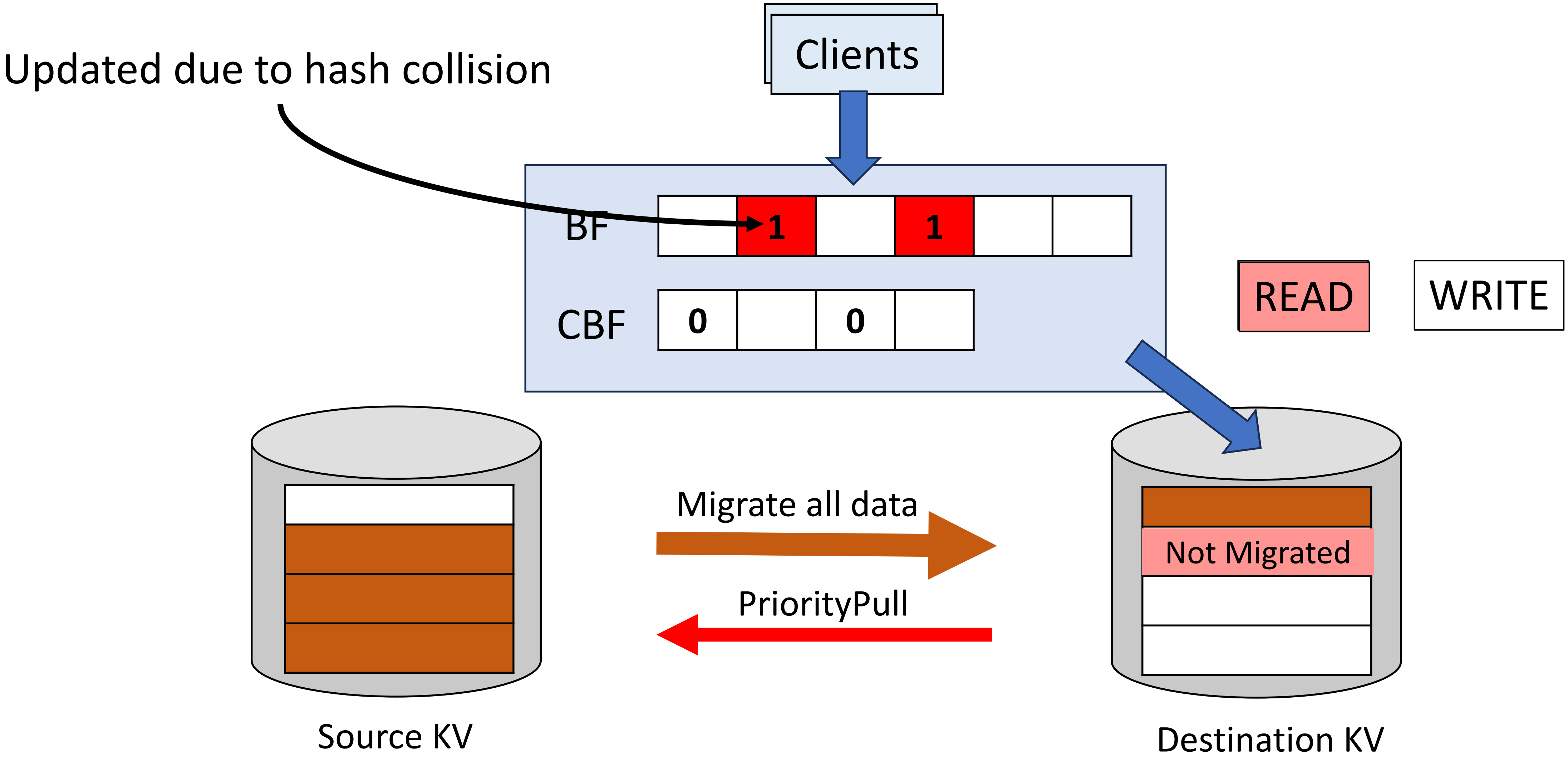


Data is Consistent When Ongoing Migration



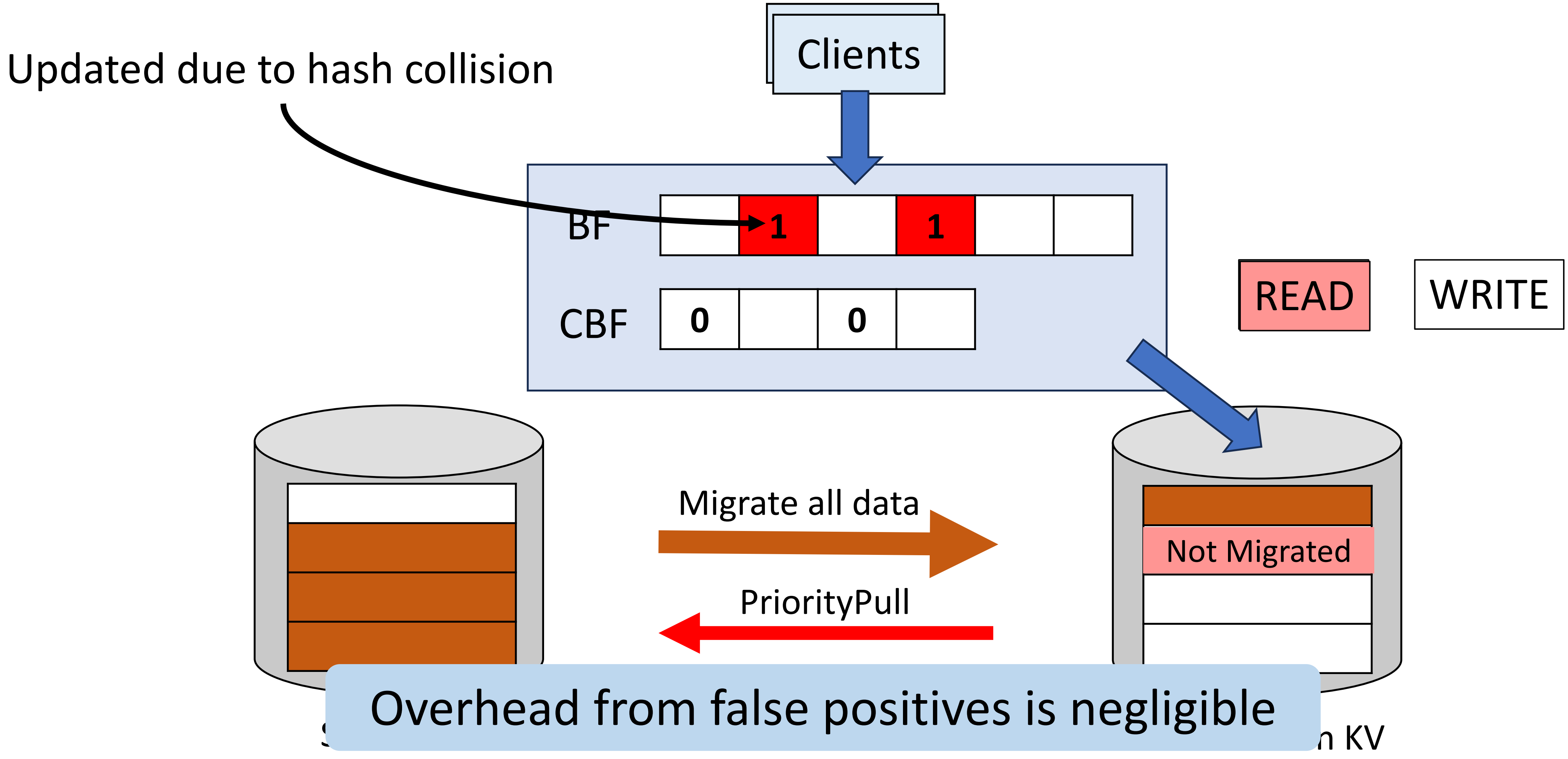
- Data consistency is maintained by Double-READ.
- Overhead caused by Double-READ is negligible.

Data is Consistent even with False Positives



(check more details in our paper)

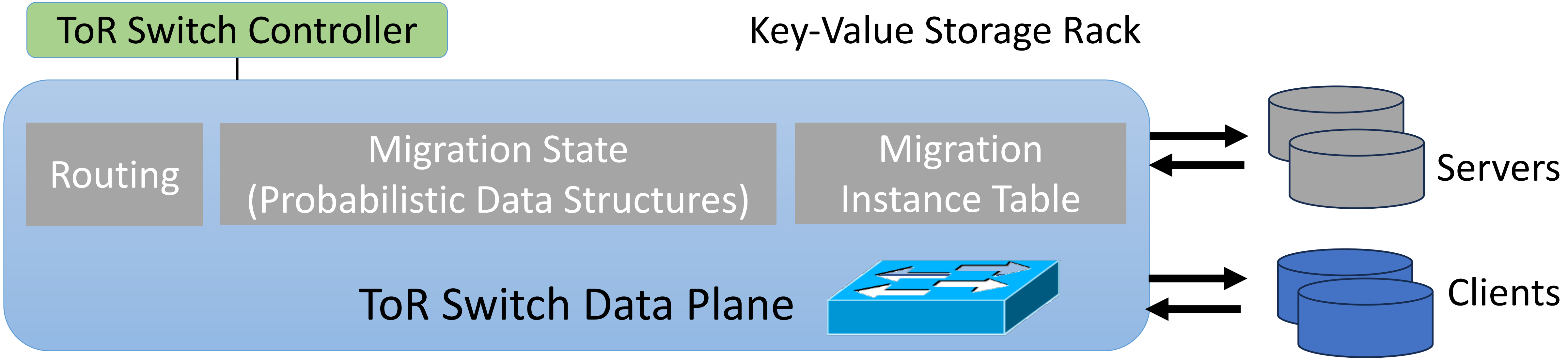
Data is Consistent even with False Positives



Overhead from false positives is negligible

(check more details in our paper)

Putting It Together: NetMigrate



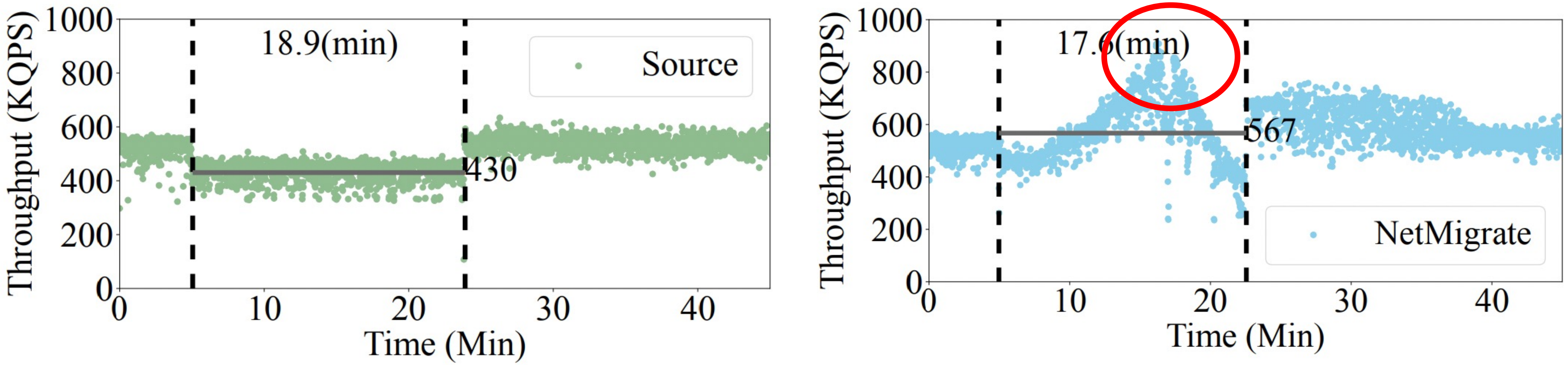
- Leveraging probabilistic data structures on the switch to track three migration states.
- Query protocol guaranteeing consistency.
- The overhead caused by false positives and unsure states is small.

Evaluation

- Testbed
  - 6.5 Tbps Intel Tofino switch
  - 3 servers each with an 8-core CPU, a 40G NIC, and 64GB memory
- Baselines
  - Source-based migration protocol, Rocksteady, Fulva
- Workloads
  - Migrating 256 million KV pairs (~16GB), with 4B key, 64B value
  - YCSB with 0%, 5%, 10%, 20%, 30% write ratio
  - Source CPU budgets: 100%, 70%, 40%

Overall performance -- Throughput

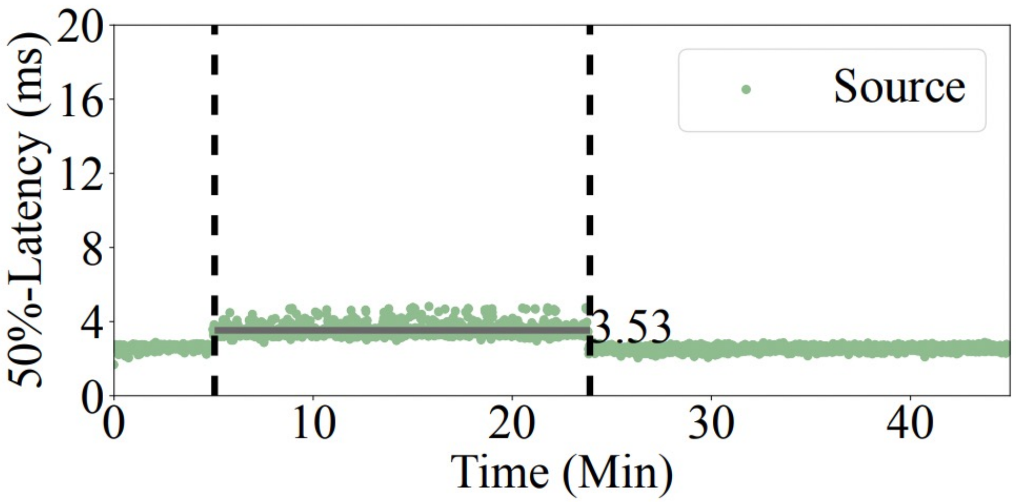
Setting: YCSB-B (5%) write ratio, source node is not overloaded (100%)

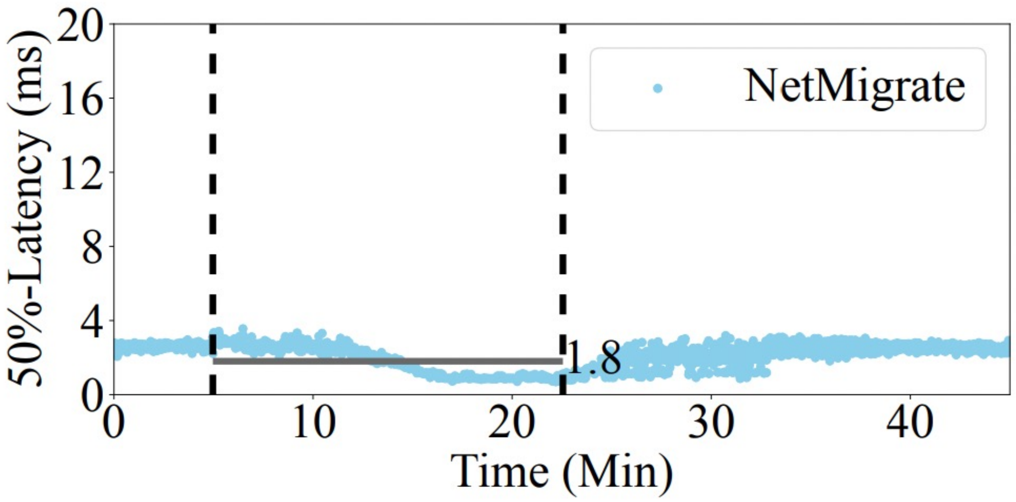


32% to 78% average throughput improvement compared to Source-based, Rocksteady, Fulva with similar migration time.

Overall performance – Median Latency

Setting: YCSB-B (5%) write ratio, source node is not overloaded (100%)





49% to 65% average median latency reduction.
Up to 39% average 99% tail-latency reduction.

Network Overhead

Protocols/Overhead	Client-side	Server-side
Rocksteady	7%~12%	0
Source-based	0	Proportional to write ratio
Fulva	~50%	0
NetMigrate	<0.05%	$<5\times10^{-5}\%$

Extra network bandwidth overhead
between clients and servers (client-side)
or between servers (server-side)

Conclusions

- Existing KV store live migration techniques still suffer from low query-serving performance and high overhead.
- We propose NetMigrate, a network-based hybrid live migration approach.
  - Track fine-grained migration states in programmable data plane.
  - Provide enhanced throughput and low migration overheads.
- Open-sourced at https://github.com/Froot-NetSys/NetMigrate.

ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE

ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL

ARTIFACT EVALUATED usenix ASSOCIATION REPRODUCED

Thank you! Q&A