AEROSPIKE

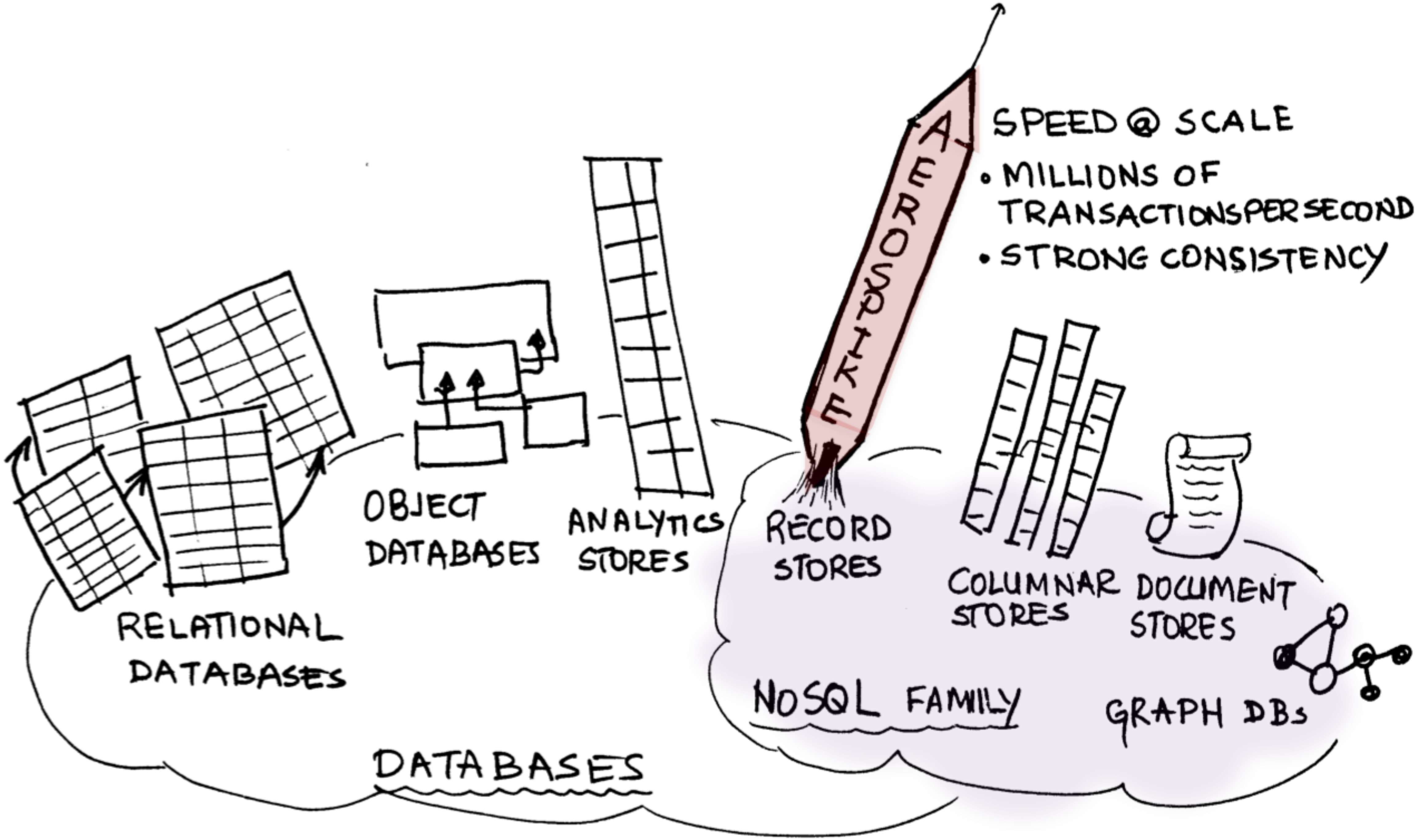
SUMMIT '19

AEROSPIKE

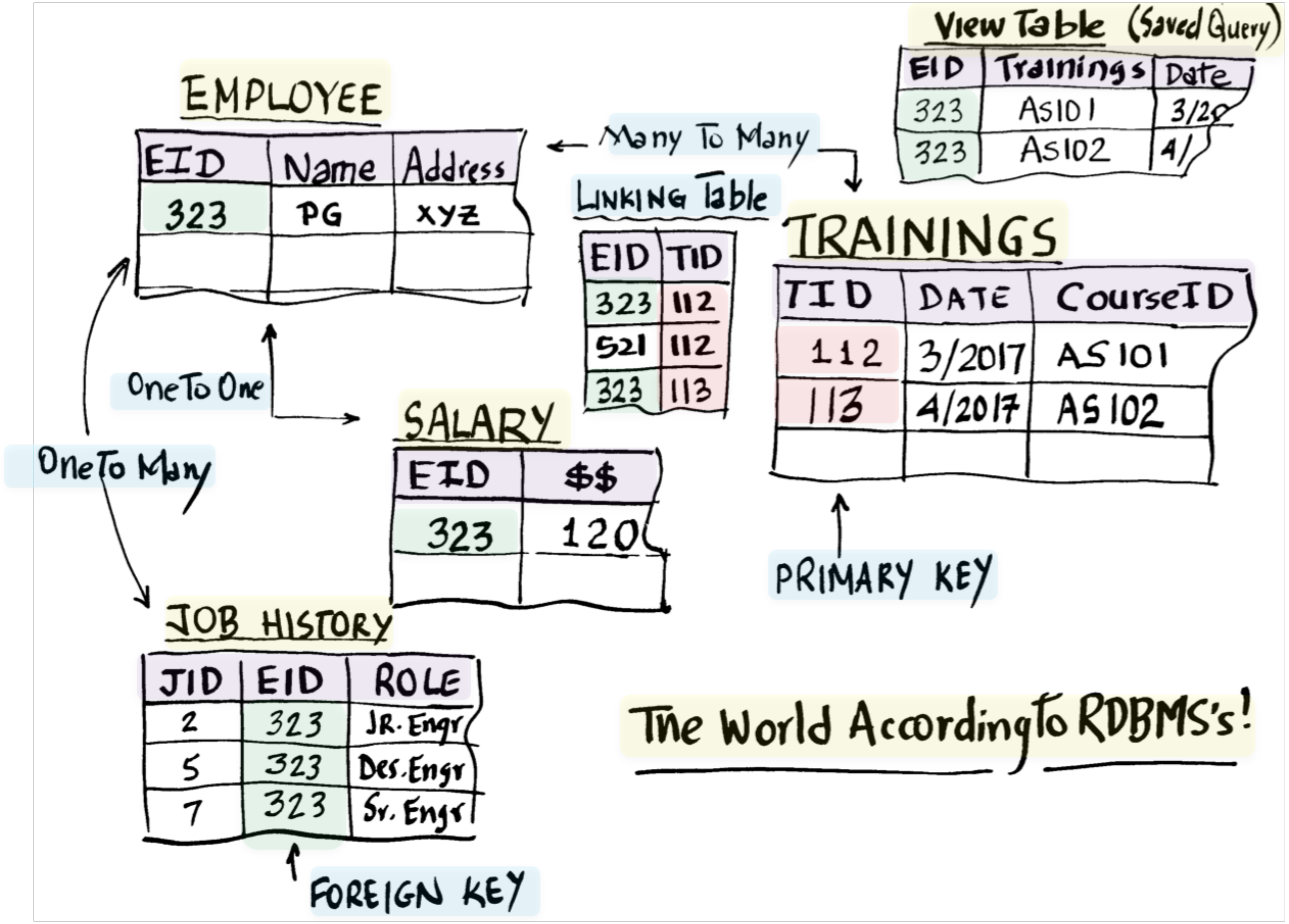
Key Data Modeling Techniques

Piyush Gupta
Director Customer Enablement
Aerospike

Aerospike is a "Record Centric", Distributed, NoSQL Database.



Relational Data Modeling – Table Centric Schema, 3rdNF

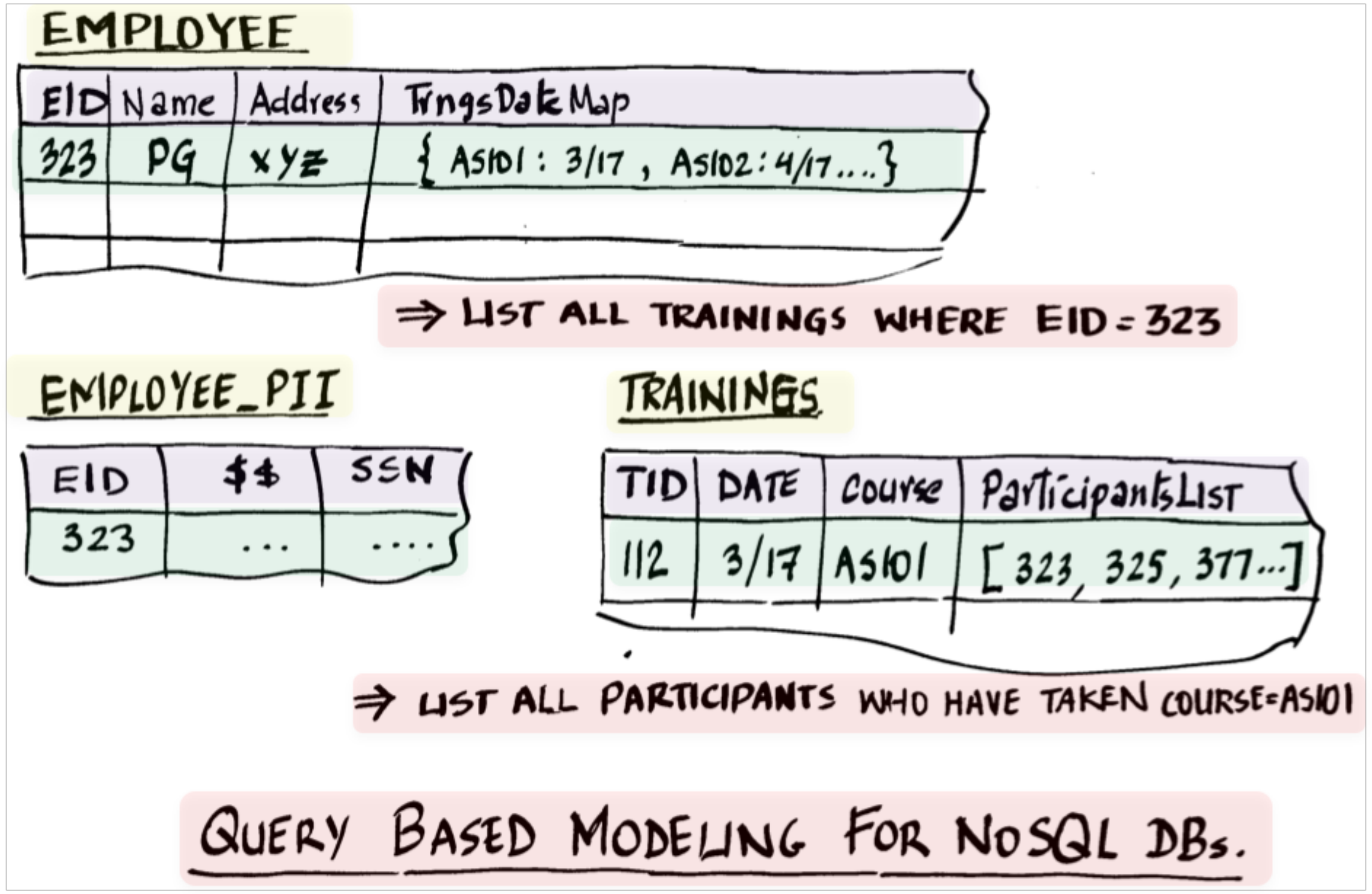


AEROSPIKE

AEROSPIKE SUMMIT '19

NoSQL Modeling: Record Centric Data Model

- De-normalization implies duplication of data
  - Queries required dictate Data Model
  - No “Joins” across Tables (No View Table generation)
- Aggregation (Multiple Data Entry) vs Association (Single Data Entry)
  - “**Consists of**” vs “**related to**”



 I Proprietary & Confidential I

AEROSPIKE

AEROSPIKE SUMMIT ‘19

Before jumping into modeling your data ...

What do you want to **achieve?**

- Speed at Scale.
- Need Consistency & Multi-Record Transactions?
- Know your **traffic.**
- Know your **data.**

Model your data:

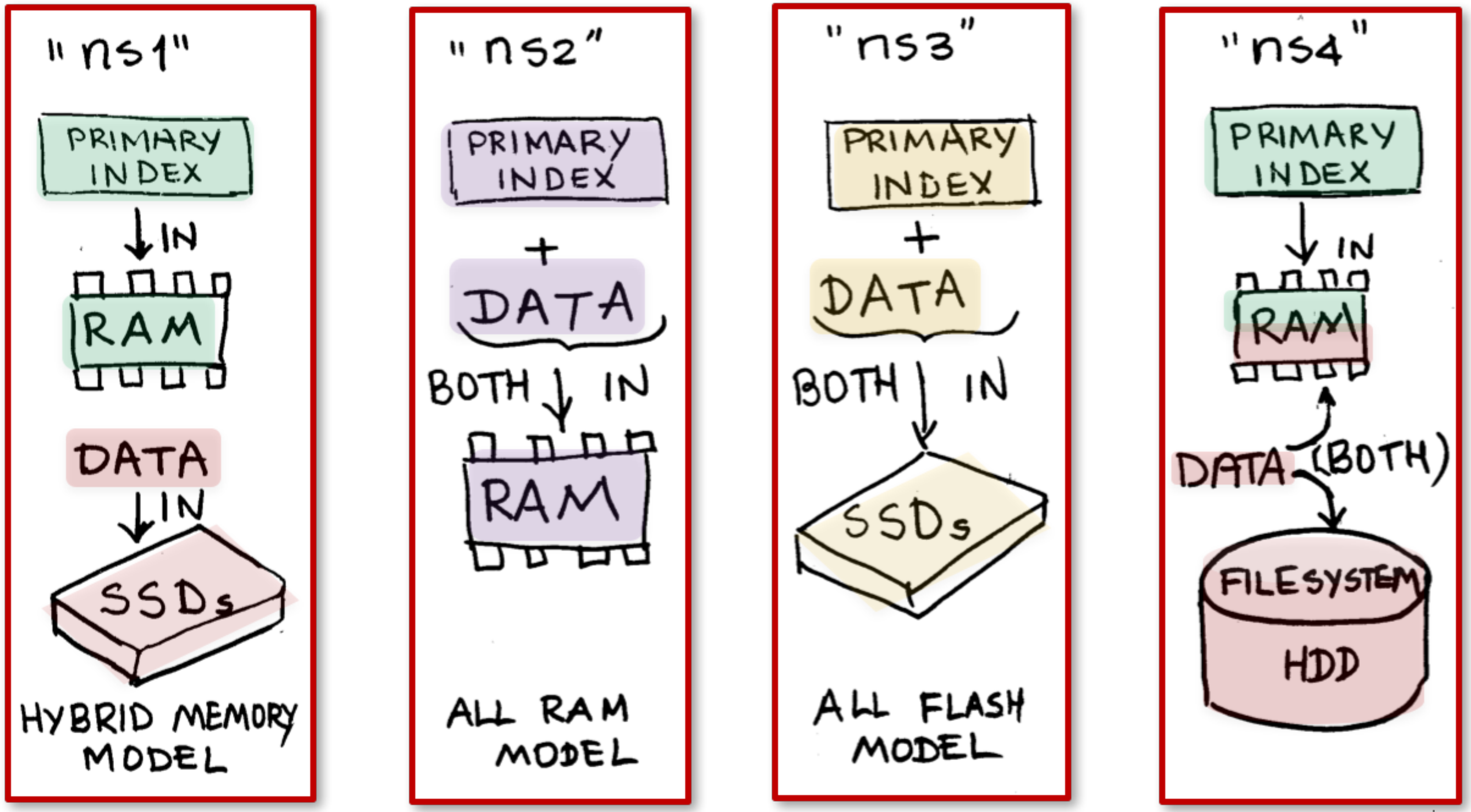
- Even a simple key-value lookup model can be optimized to significantly reduce TCO.
- Will you need secondary indexes?
- List your Queries upfront.
- Design de-normalized data model to address the queries.

Data Modeling is tightly coupled with reducing the Total Cost of Operations.

Aerospike Architecture Related Decisions

Namespaces – Select one or more. [Data Storage Options]

- Storage: RAM– fastest, File storage slowest. SSDs: RAM like performance.
- ALL FLASH: TCO advantage for petabyte stores/small size records. Latency penalty.



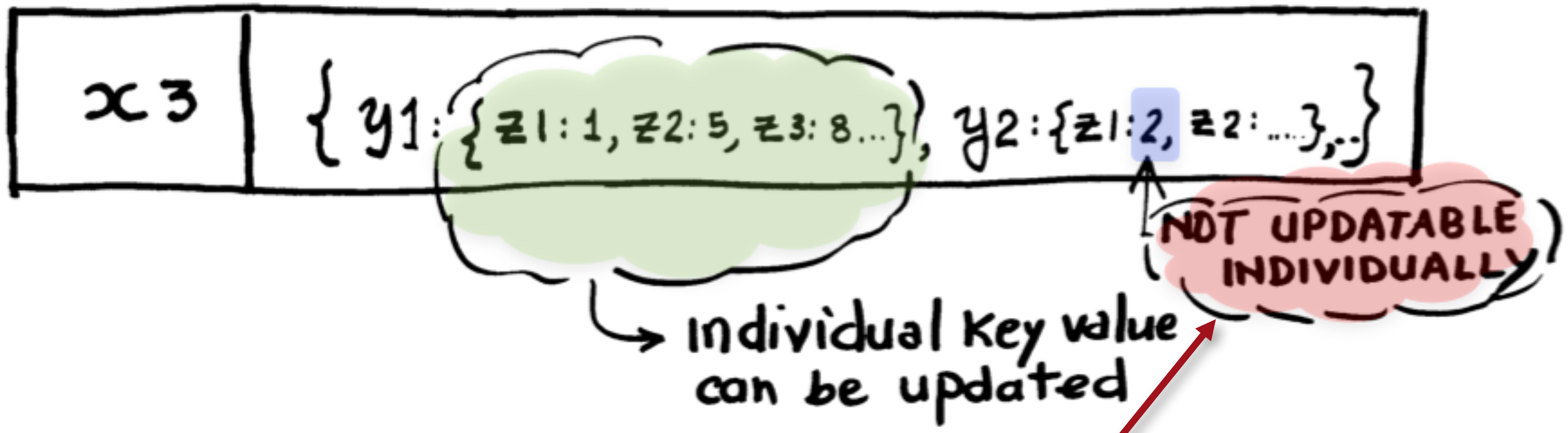
Aerospike API Features for Data Modeling

API Features to exploit for Data Modeling:

- **Write policy** - GEN_EQUAL for Compare-and-Set (Read-Modify-Write).
- **Write Policy flags**: e.g. UPDATE_ONLY (don't create),
- **Complex Data Types** (CDT) – Maps and Lists – offer rich set of APIs.
- **Map Write Flag**: CREATE_ONLY (fail if MapKey exists).
- **Operate()** – update a record and read back in the same record lock.
- **Other Features**: Secondary Index Queries, Scans, Predicate Filtering.

Maps & Lists

- Maps are items stored as key:value pairs, in key order.
- Value may be any scalar data type allowed in Aerospike, including lists and maps.



- API access to nested Maps and Lists coming soon!

Limitations:

- Map's individual K:V pairs do not have a separate "time-to-live". TTL is always at record level.
- UDFs have limited ability to modify Maps.

Maps: Terms & Nomenclature

Size of the map = number of Key:Value pairs in the map.

- {1:1, 3:6, 5:3, 6:8, 7:1} : Size = N = 5

Index: Position of the key:value pair in the map.

- {1:1, 3:6, 5:3, 6:8, 7:1} :Pair 1:1 has index **0**. Pair 3:6 has index **1**.

Negative indexing (REVERSE_INDEX): Index of the last item in this is: -1, second last is -2

- {1:1, 3:6, 5:3, 6:8, 7:1} : Pair 6:8 has index **3** and is also index **-2**. Pair 7:1 has index **4** and is also index **-1**.

Rank: Order of the **value** of the key:value pair items. [RANK 0 = Minimum Value, RANK -1 = Maximum Value]

  - Negative Indexing applies to Ranks too.
- {1:1, 3:6, 5:3, 6:8, 7:1} : Rank 2 = Value 3, pair 5:3, Rank 4 = Value 8, pair 6:8, which is also Rank -1.

Lists: Terms & Nomenclature

Size of the list = number of elements in the list.

- [1, 4, 6, 1, 3, 8] : Size = N = 6

Index: Position of the value in a list.

- [**1**, 4, **6**, 1, 3, 8] : Value 1 has index 0. Value 6 has index 2.

Negative index (aka REVERSE_INDEX): Index of the last item.

  - Last item is: -1, second last is -2

- [1, 4, 6, 1, **3**, **8**] : Value 8 has index 5 & also index -1. Value 3 has index 4 & also index -2.

Rank: It is the Order by Value.   [RANK 0 = Minimum Value, RANK -1 = Maximum Value]

  - Negative Indexing applies to Ranks too.

- [**1**, 4, 6, **1**, 3, **8**] : Rank 0 = Value 1, Rank 1 = Value 1, Rank 2 = Value 3, Rank 3 = Value 4, Rank 4 = Value 6, Rank 5 = Value 8, which is also Rank -1.
- Lists may be ORDERED (by value) or UNORDERED (default). Lists may also be SORTED.

Rich Set of Map APIs allow creative Data Models

- add(), add_items(), increment(), decrement()
- get or remove ... _by_key(), _by_index(), _by_rank()
- get or remove ... _by_key_interval(), _by_index_range()
- get or remove ... _by_value_interval(), _by_rank_range(), _all_by_value()
- get or remove ... _all_by_key_list(), _all_by_value_list()
- clear() - remove all items from the map
- size() - number of K:V pairs in the map

List APIs

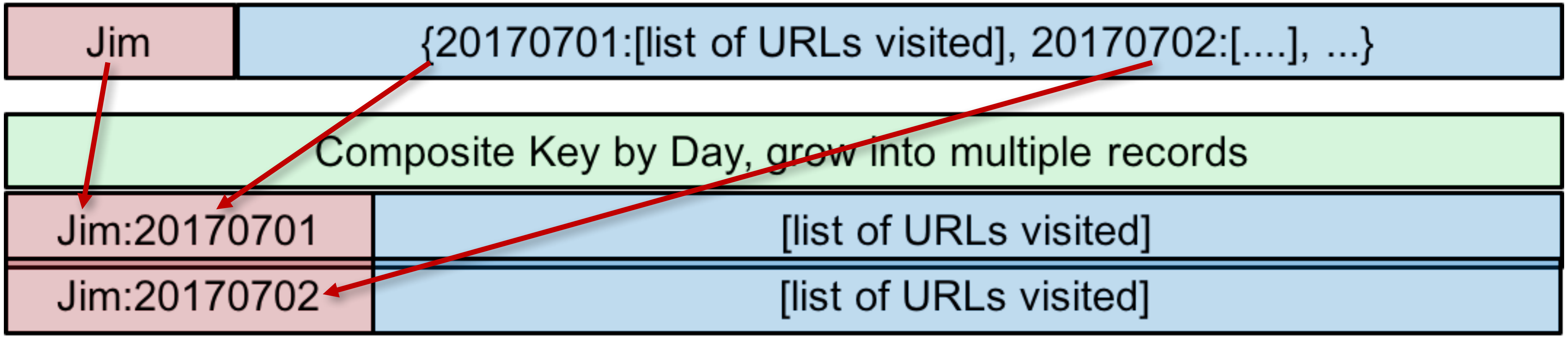
- append(), insert(), insert_items(), add_items()
- set() - to replace or set value at an index
- get or remove ... _by_index(), _by_index_range()
- get or remove ... _by_rank(), _by_rank_range()
- get or remove ... _by_value(), _by_value_interval(), _all_by_value(), _all_by_value_list()
- increment(), sort()
- clear(), size()

Modeling Tips and Tricks

Composite keys

Composite key created with 2 pieces of related data, easily derived in application.

- Breaks a long record into multiple records by using an attribute as part of the primary key



- Helps with 8MB record size limit on SSD.

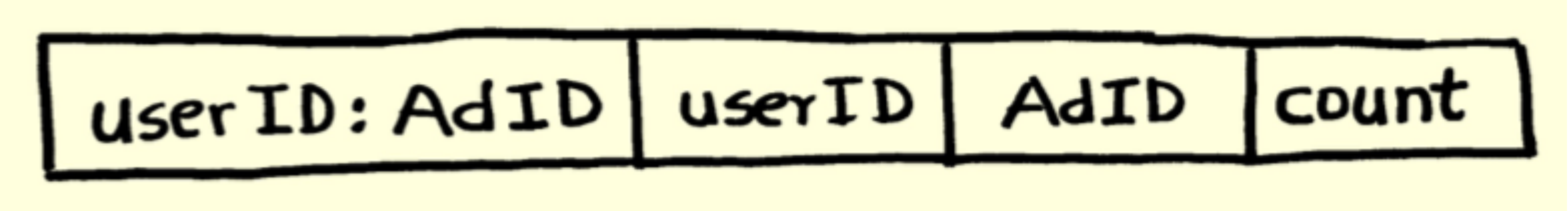
Using Composite Keys & Counting on the fly.

Problem:

- Count every time user-X sees Ad-Y.
- Provide list of all unique users who have seen Ad-Y.

Solution:

- Use composite key: user-X:Ad-Y with bin1=userid, bin2=AdID and bin3=count.



- When user-X sees Ad-Y, enter userID, adID and increment count.
- Run Secondary Index query on AdID bin to return list of bin=userID.

Using Hash - The Reverse Lookup Problem

Problem: Reverse lookup userID by any userIDAttr i.e. given email, find userID.

userID	userData	userIDAttr
user235	{k1:v1, k2:v2 ...}	["email:xyz", "phone:123", ...]

- Create reverse lookup set using data-in-index with userIDAttr as Primary Key?
- Or Secondary Index on userIDAttr LIST values?
- **Cons**: RAM required for Primary Index or Secondary Index is exorbitant.
- **Pros**: Maintains record level atomicity.

Reverse Lookup – Sizing the RAM

- Consider 1 billion records, average 2 userIDAttr per user
- A) **RAM for Data-in-index**: $64*2*10^9/(1024^3)$ = 119GB
- B) **SI on userIDAttr** LIST values - RAM estimate
- RAM for SI = 28.44*1.5*[K + R] = 28.44 * 1.5 * [2,000,000,000 + 1,000,000,000] $/(1024^3)$= 119GB

lookupKey	userdIDAttrMap
Hash26LSBs	{"email:xyz":user235, "phone:567":user566, ...}

Hash Based Reverse Lookup Records:

- *Primary key* = 26 bits of RIPEMD160 Hash of "email:xyz" values. (2^{26} = ~67 million)
- Each map bin will have approx. 1000*2/67 = *30 key value pairs*.
- *PI RAM*: $67,108,864*64/(1024^3)$ = 4 GB (Significant - ~30x - RAM savings)
- *Cons:* Any record entry or update requires lookup table update also. (Multi-record update).

Reverse Lookup - Managing Multi-Record Update

Update Sequence for avoiding hung pointers:

- 1 - Update Lookup Table entry of new or updated userIDAttr
- 2 - Update the userID – data / userIDAttr list record
- 3 - Update Lookup Table to delete stale userIDAttr entry (if applicable in case userIDAttr was modified)

Benefit of using Lookup Table method

- Significant reduction in RAM usage.
- On AWS, i3 instances are 31:1 ratio, SSD to RAM.
- Adding more instances just for RAM gets very expensive.
- This technique can significantly reduce TCO.
- **Alternate:** Use ALL FLASH option (namespace selection).

Using Hash - Modeling Tiny Objects

Problem:

- Our object size is very small, say 12 bytes of data – 64 bytes of Primary Index per record is causing high RAM usage.

- We have:
- k1:v1
- k2:v2
- k3:v3 ....
- where v1, v2 ...etc are very small size.

Primary Key	id	name	ver
"id1"	"id1"	"name1"	1
"id2"	"id2"	"name2"	2
…			
"id5"	"id5"	"name5"	5

Primary Key	records
0x00000011	{"id1":{"id":"id1", "name":"name1","ver":1}, "id2":{…}, "id5":{..}

Using Hash - Modeling Tiny Objects (cont.)

- Aggregate small objects as Key:Value Maps into larger objects
  - Take RIPEMD160 Hash of (id1) → 20 bytes of digest.
  - Bitwise AND to keep desired number of significant bits.
  - For eg: Consider a 1 byte hash example (256 unique keys) :
    - hash(id1) = 0x11010011
    - hash(id2) = 0x10101011
    - hash(id3) = 0x11010001
    - hash(id4) = 0x11110101
    - hash(id5) = 0x11110011

Primary Key	records
0x00000011	{"id1":{"id":"id1", "name":"name1","ver":1}, "id2":{...}, "id5":{..}

    - **0x11010011 & 0x00000111 → 8 unique keys**
    - Keys id1, id2, id5 – end in same large record, whose Primary Key will be: 0x0000011
  - Key=0x00000011 : Bin = { id1:v1, id2:v2, id5:v5} ... Use Aerospike Map Type
  - In the above limited example, we compressed 256 records into 8 records.

Modeling Tiny Objects

Benefit:

- Significant reduction in RAM usage.
  - Fewer Primary Index entries.
  - Map Type storage is more compact.
- Can significantly reduce TCO.

Cons:

- XDR is no longer shipping individual records.
- TTL – Best if using "Live-for-Ever" – you loose per record TTL granularity.

Good solution for single entity very high read access pattern

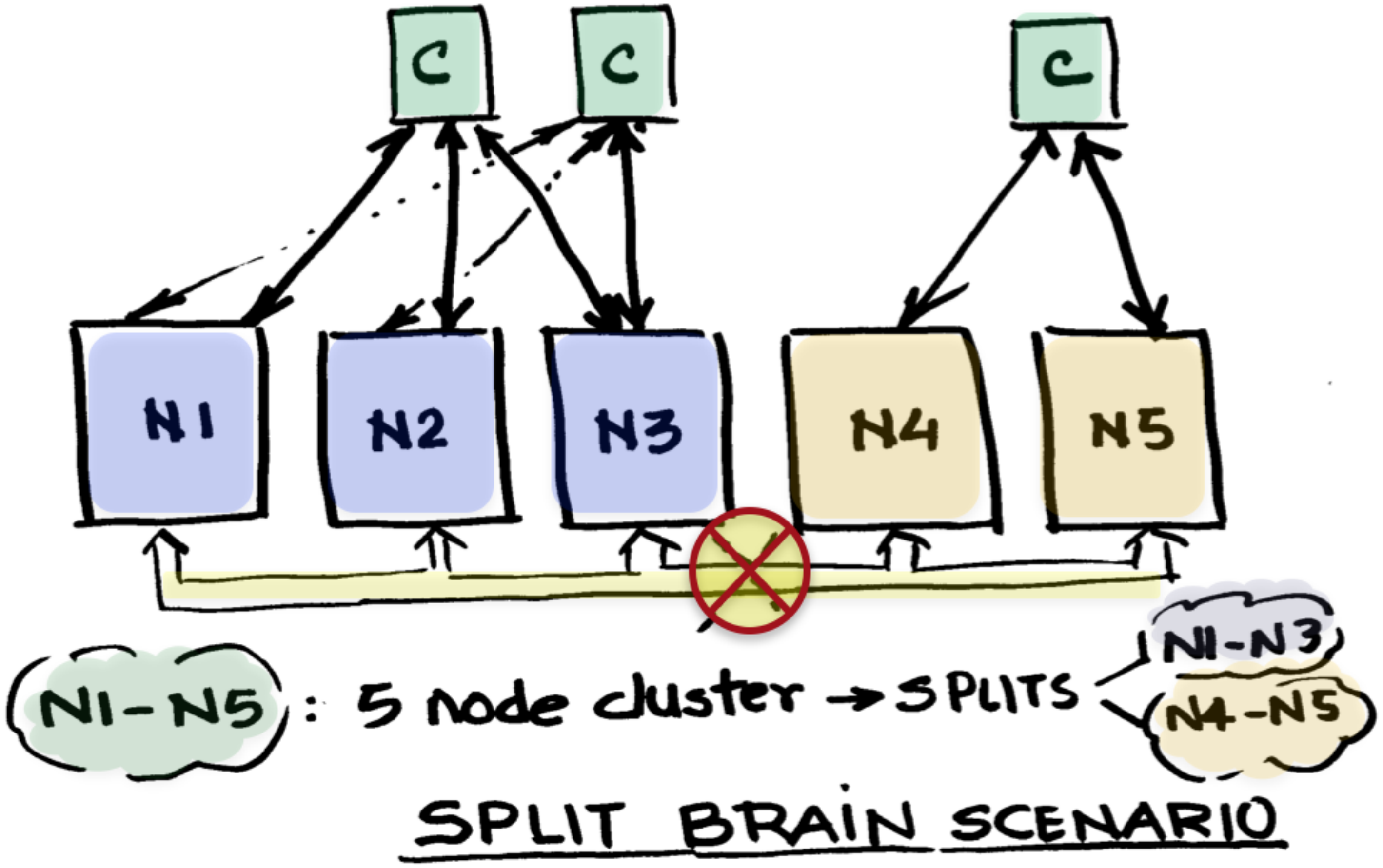
Multi-record Transactions

Multi-Record Transactions (MRTs)– Make Records Stateful!

- Implemented at Application Level.
- Aerospike locks one record at a time. MRTs not offered at server level.
- Must deal with client failing in the middle of an MRT.
- Many multi-record problems can be modeled using co-operative locking.
- Must have a robust rollback scheme in case of failure.
- Proposed scheme uses a UDF – recognize UDF performance limitations.
- *Alternate:* Use expiration time & lock bin and poll. (Adaptive Map Example)
- Use Strong Consistency Mode of Operation to address split-brain concerns.
- *Strong Consistency mode guarantee:*
  - Successful writes or updates are never lost.
  - Reads can be configured to be never stale.
  - No Dirty reads (Reading data that is not fully committed).

Split Brain

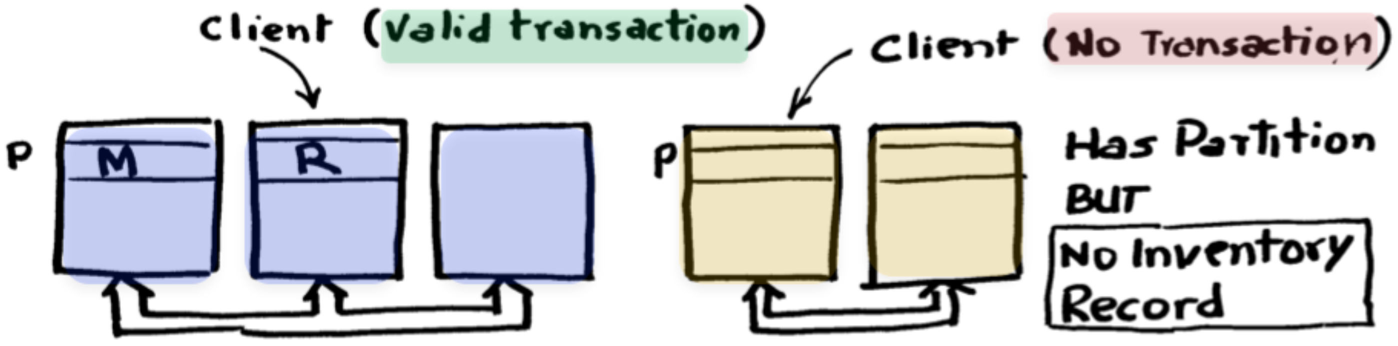
- Split Brain for the purposes of our discussion is defined as a scenario where a cluster splits into two or more separate clusters, each thinking it is **the** cluster.
- This happens rarely, but is a possibility.
- Take into consideration when modeling any transactions on distributed databases.



Split Brain in AP Mode – Not an issue in Strong Consistency Mode.

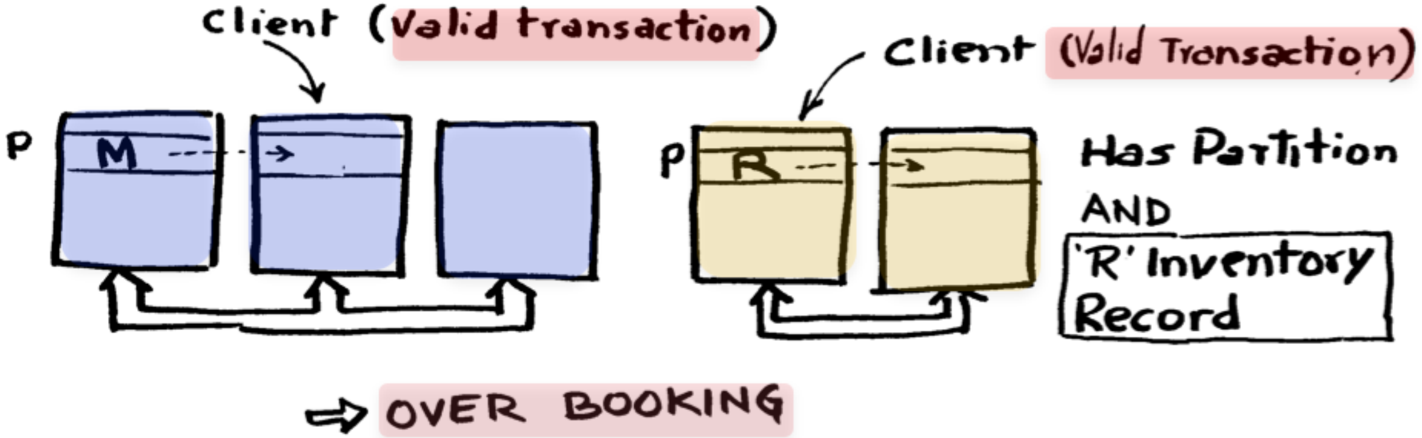
- Case 1: Master & Replica on same split, no inventory record on the other split.

➔ Non-Availability



- Case 2: Master & Replica on split clusters can create valid over-bookings.

➔ Inconsistency



AEROSPIKE

AEROSPIKE SUMMIT '19

Co-operative Locking for Multi-record Updates

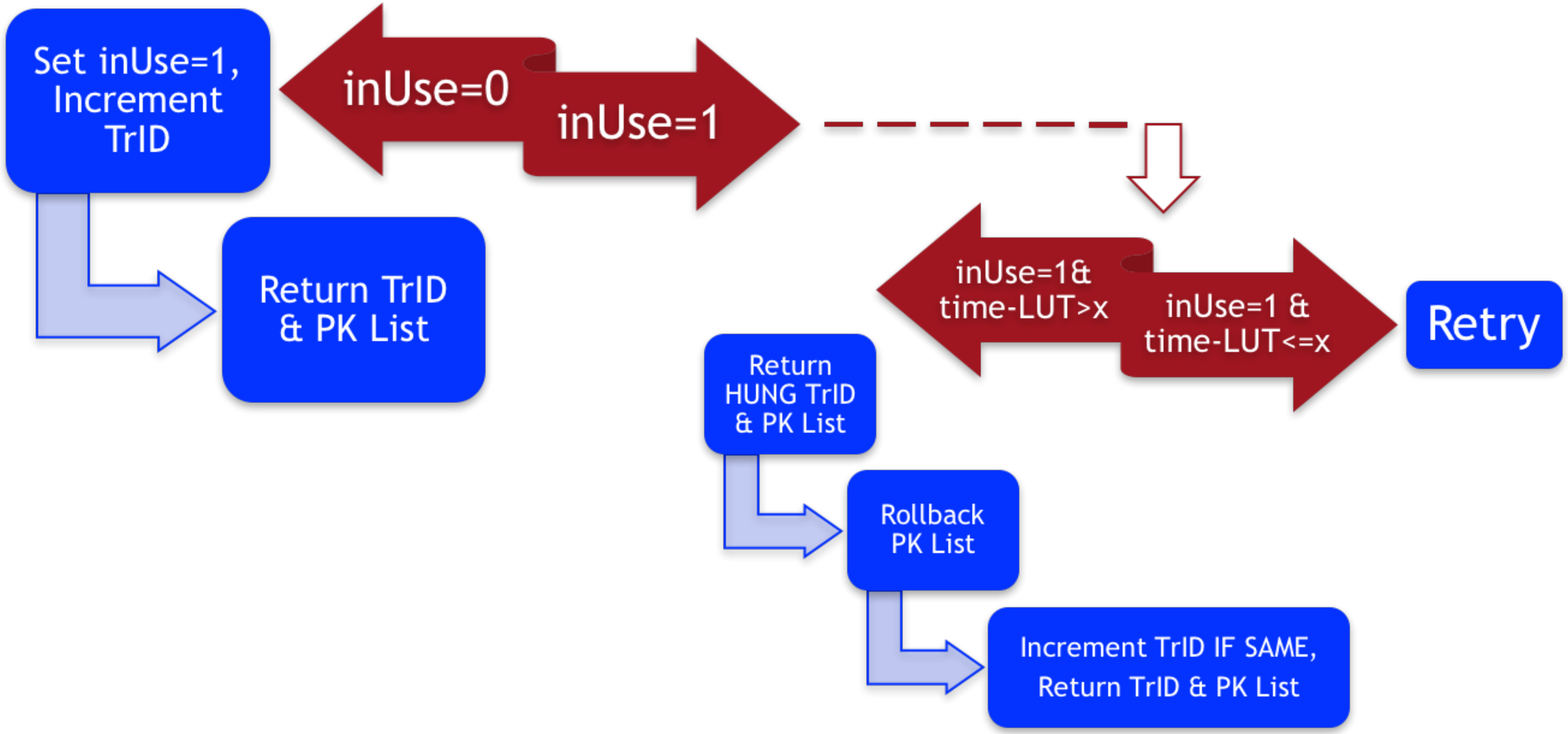
- Implementing co-operative locking via 'inUse' bin (flag or lock) and TransactionID.
- Allocate reasonable time 'x' for doing all updates.
- Use a RecordUDF to manage inUse bin. *(Alternate, expiration with polling discussed later.)*

Detect hangs in RecordUDF:

  - IF: inUse = 1, AND if currTime - LUT > 'x' (else retry) return the "hung" Transaction ID and list of pk1..etc.
  - Rollback partially updated transaction records (pk1...etc) using hung TrID.
  - Return to PK_lock record. If TrID is same, increment it and update the record. Return new TrID and List of PKs.
  - ELSE (normal case): If inUse = 0, set inUse=1, increment TransactionID, update the record and return list of PKs to modify and new TransactionID.
  - **Assumption:** Records listed in bin1 that participate in this transaction are exclusive to this transaction. e.g. Self-sharding Adaptive Map.

PK_lock	inUse = 1 or 0	TrID=344	bin1=[pk1, pk2, pk3,...]

Co-operative Locking for Multi-record Updates



PK_lock	inUse = 1 or 0	TrID=344	bin1=[pk1, pk2, pk3,...]

pk1	i_Trid=204	i_bin1="abc"	f_TrID=304	f_bin2="xyz"
pk2	i_Trid=304	i_bin1="def"	f_TrID=344	f_bin2="vwx"

Co-operative Locking for Multi-record Updates

Implementing Rollbacks in transaction records (pk1 ... etc)

- Use a RecordUDF with hung TrID to update each transaction record.
- *Each bin has initial state value and final state value.*
- *Initial state and final state have associated transaction ids.*
- Use hungTransaction ID (eg 344) to roll back partially updated records (pk1 ... etc)
- If pk1 was successfully updated, f_TrID will be "344", otherwise "304"
- When rolling back, (if f_TrID = 344) move i_TrID and i_bin1 to f_TrID and f_bin1.
- **Note:** Do not execute new transaction till all previous hung transactions records (pk1...etc) have been restored to initial state. (Avoid multiple sequential hangs error.)
- **Normal Update:** Move f_TrID and f_bin1 to i_TrID and i_bin1 and new value to f_bin1 and f_TrID.

pk1	i_Trid=204	i_bin1="abc"	f_TrID=304	f_bin2="xyz"

Using Polling and Error Flags for Locking Records – Alternate to RecordUDF Approach

AEROSPIKE

Data Modeling for Bins with Varying Expiration – Polling Example

- Requirement: Build a User Profile Store with each Attribute having a TTL associated with it

ISP User Profile Store

userID	topSearchItem	mostVisitedSite	city
u42	laptop	bestbuy.com	San Jose
	TTL = 1 day	TTL = 5 days	TTL = 5 years

- User may have 10 to 100 attributes, each may have different TTL

Data Modeling for Bins with Varying Expiration – Polling example

- Solution: [userID I Map{attr:value} I Map{attr:TTL}]

userID	attrValue Map	attrTTLMap
u42	{topSearchItem: laptop, mostVisitedSite : bestbuy.com, city: SanJose }	{topSearchItem : 1d, mostVisitedSite: 5d, city: 5y }

- TTL = future timestamp to expire attribute (i.e. 1d ➔ future timestamp value)
- K-V sorted map policy on Map{attr:TTL}
- Map type allows updating any single key:value pair in a bin
- Periodically, use sorted Map API to find lowest TTL :

```
timestamp = get_by_rank(attrTTL,0,VALUE)
if timestamp < current_time,
  key = get_by_rank(attrTTL, 0, KEY)
  remove attrValue:key and attrTTL:key
```

Pros: Single record read/write operations

Cons:

- Must scan through all records to delete expired bins ➔ Use scan UDF
- Use client side logic on read to delete expired attribute value and TTL (intra scan deletes)
- Record size limitation – 8MB (ver4.2+) on SSD

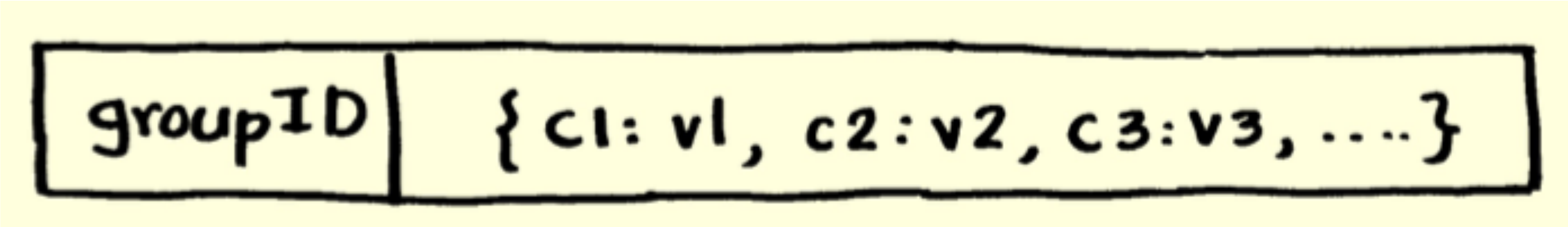
AEROSPIKE

AEROSPIKE SUMMIT '19

Adaptive Map – a Working Multi-Record Transaction Example

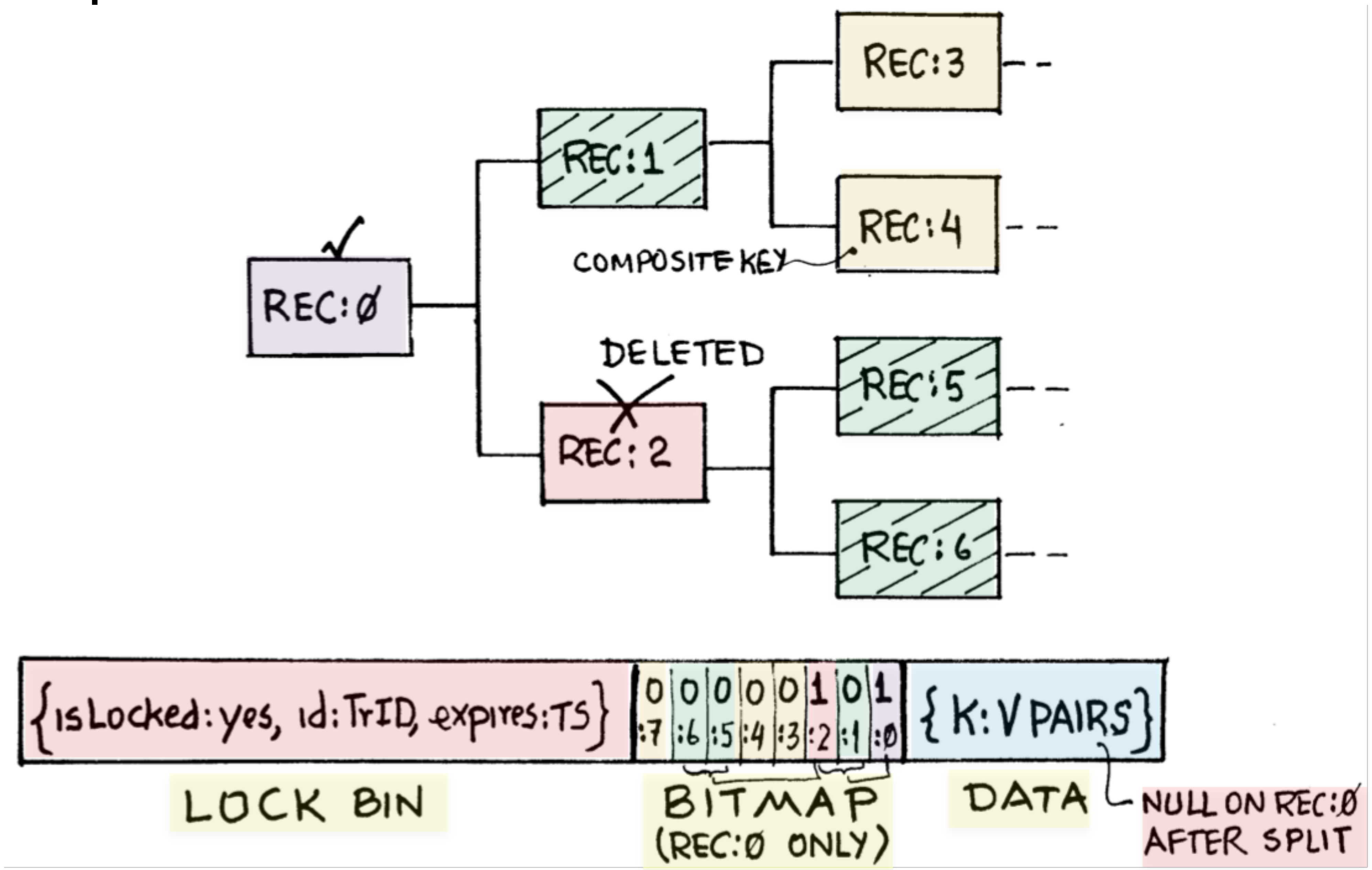
- **Requirement:** Build a self-sharding adaptive map storing key value pairs.
- Append only, no rollbacks needed.

➢ Multi-record data model using Co-operative Locking with Polling



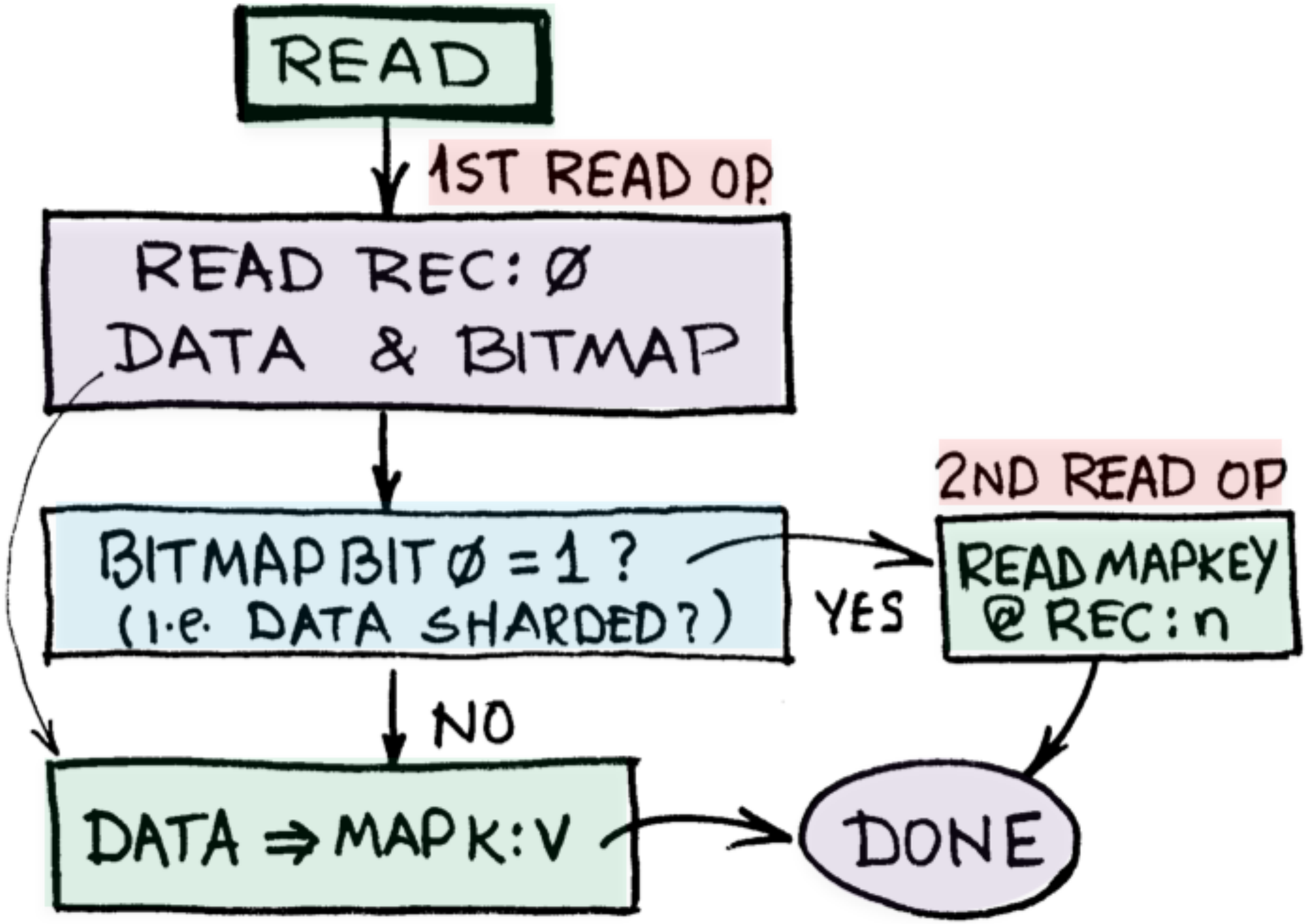
- ***Aerospike Constraints:*** Single Record Level Locking, Max 8MB record size.
- Starts with Record:0 and automatically self shards and grows till namespace capacity.
- If not sharded yet, Writes or Reads complete in single operation ➔ No performance penalty.
- After sharding, need two or more record operations for read, write or sharding.
- Cannot have stale reads or lost writes ➔ Must use Strong Consistency Mode.
- *Reach to your Aerospike Solution Architect if you have a use case for this source code.*

Adaptive Map – Data Structure



BITMAP (Stored only on REC:0) : 1 ➔ Record has split, 0 ➔ Has Data if Root or Parent has Split.

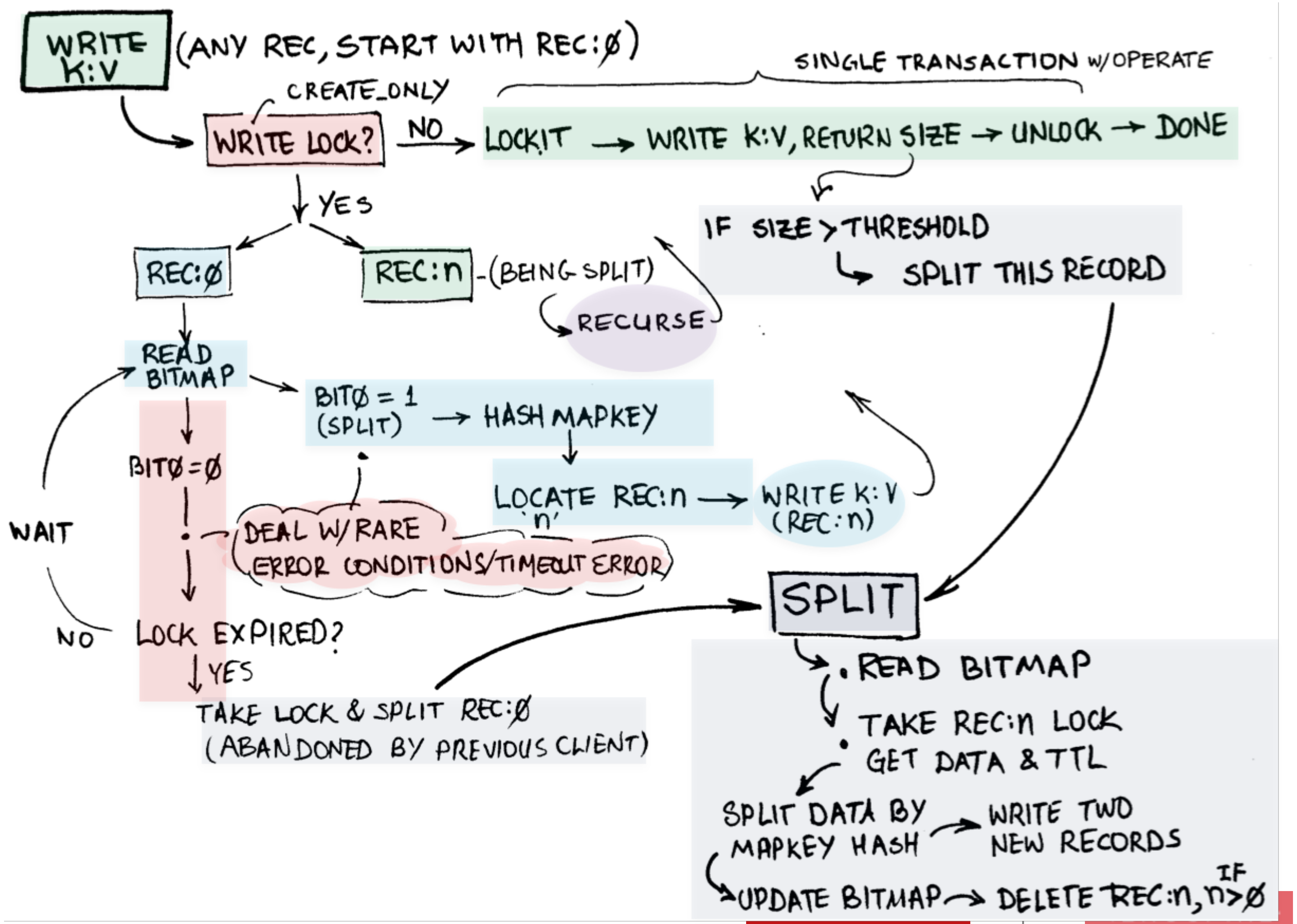
Adaptive Map – Reading a MapKey Value.

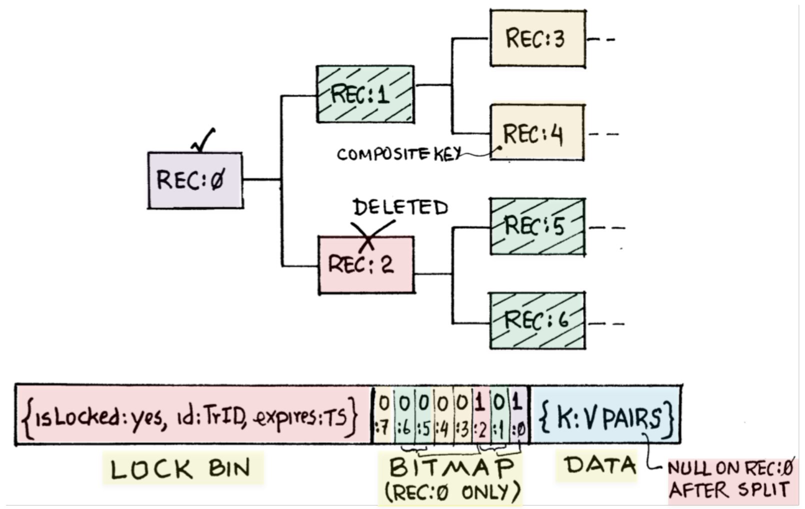


- If REC:0 is sharded, BITMAP will read 1 in Bit 0. (Otherwise DATA has MapKey:Value.)
- Find the actual REC:n containing the data using the BITMAP.
- Read the MapKey in second read operation.

Adaptive Map – Writing a MapKey Value.

- All sharded records uniquely tied to root record. [REC:0]. Once REC:0 is sharded, leave it write locked.
- Poll for TIMEOUT errors & abandoned lock, test for expiration timestamp. (Alternative to UDF approach.)





Using Error Flags for "locking".

- **Key Idea:** Set and Infer state of record (locked or not) using write error flags such as MAPKEY CREATE_ONLY or BIN_TYPE (increment a string).

- Adaptive Map uses CREATE_ONLY feature of Map K:V insert operation to check the lock or take and complete the write in the same Aerospike record lock.
- If CREATE_ONLY fails, we know the lock is taken.
- Read the lock clientID and expiration time.
- If expired, overwrite the expiration time and clientId and acquire the lock OTHERWISE poll periodically (every 1 ms).

- Alternate FLAG used in other models: Key Type Error – increment an integer bin, to lock the first record in a multi-record operation, write a string in this bin. When done, re-write an integer.

Record UDF Performance

- A given UDF module, on any given node after >128 concurrent invocations will exhibit performance degradation.
- A record UDF attached to a scan will yield better performance than a invoking a UDF to modify on each record from the client side.
- A simple record UDF may **not** perform better at scale than CAS using generation especially if the collisions are infrequent and number of concurrent calls to the UDF increases.
- If possible, characterize performance for specific data model and cluster implementation and decide best course.

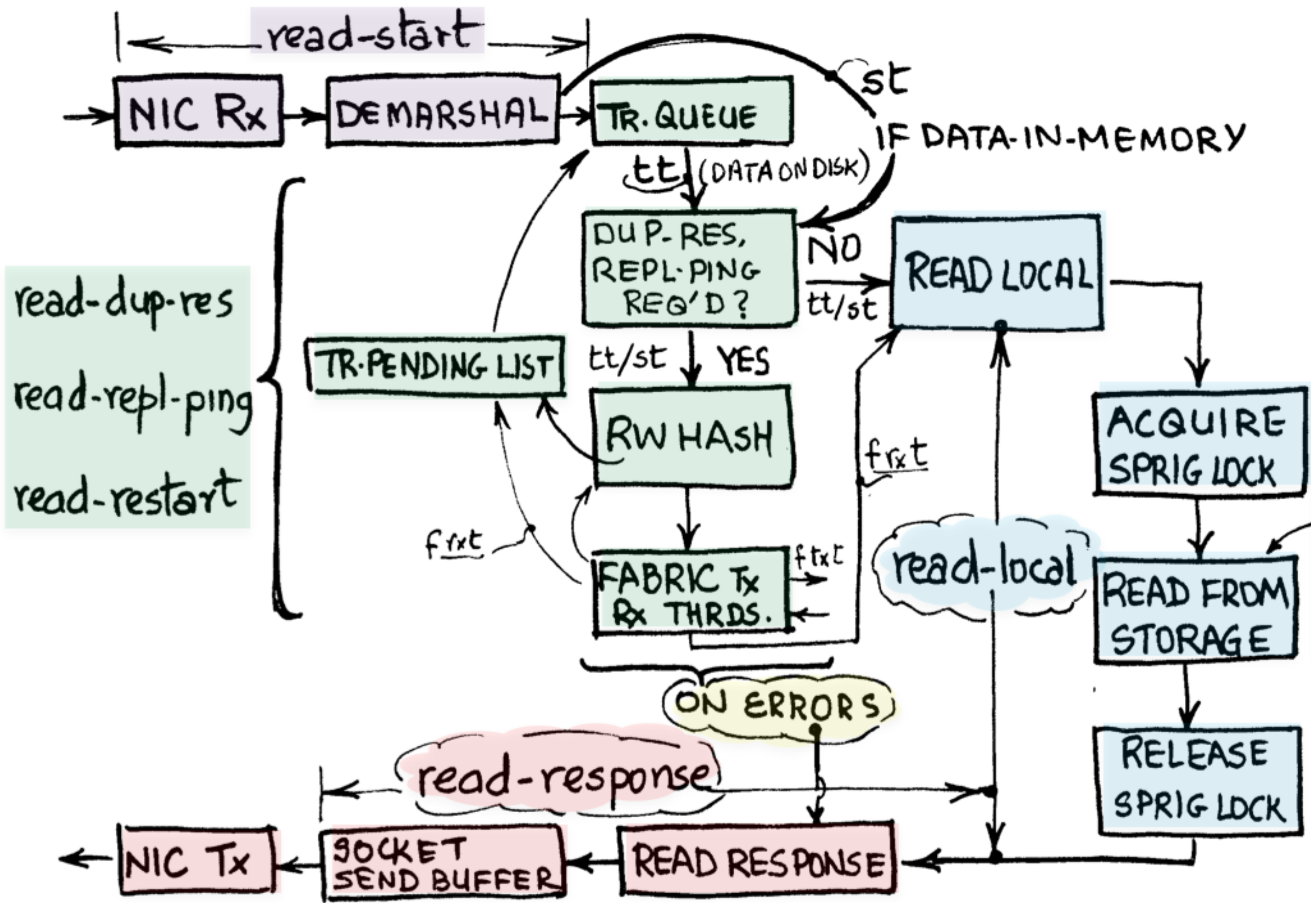
Need Lowest Possible Latency? Understand Transaction Flow ...

Read Transaction Flow

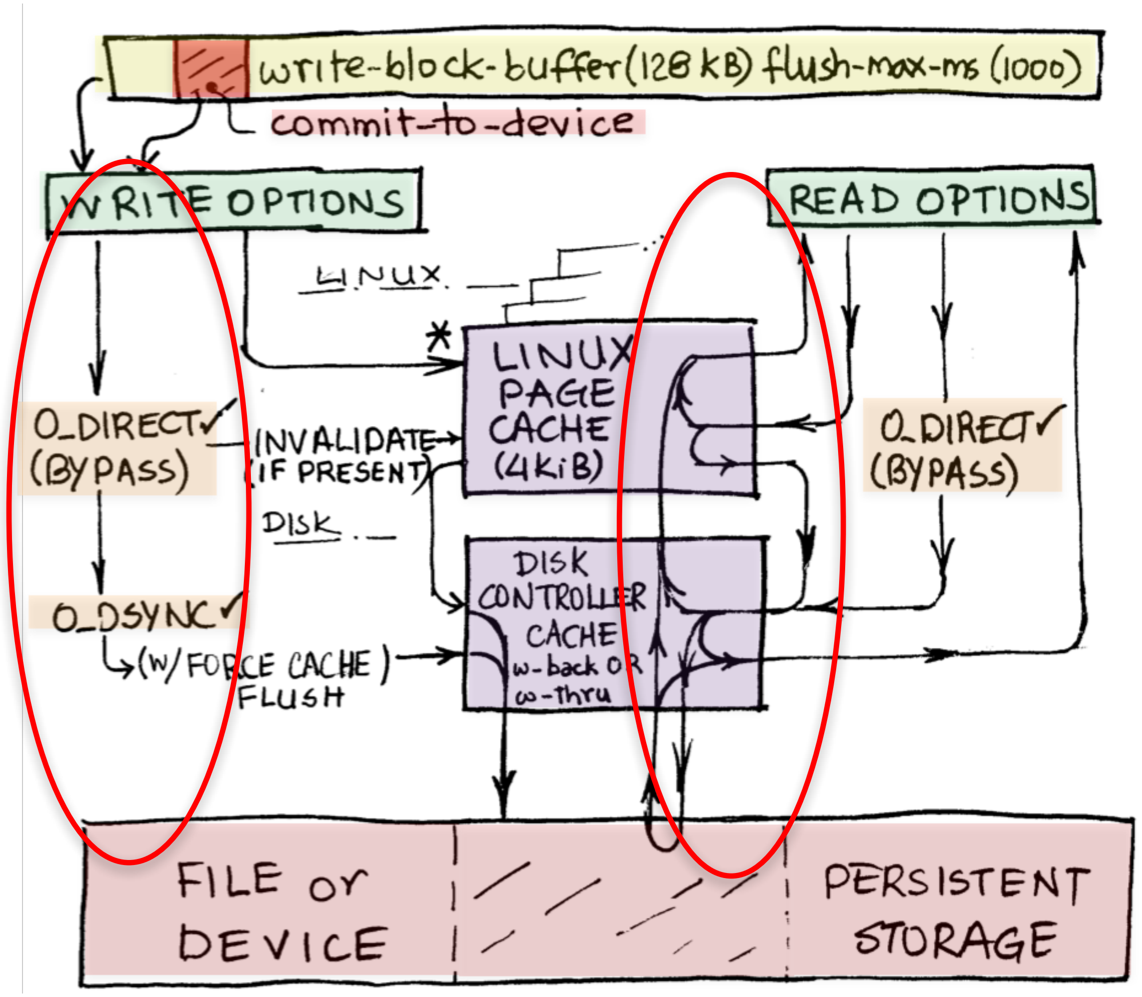
- Read transactions that exchange msgs/data via the fabric, or go to disk, or park in a queue ... we see latency impact, and in that order.

For low latency, consider:

- Data-in-memory (namespace)
- No Duplicate Resolution (default)
- No Linearize Read (SC only)



"read-page-cache" ON "device" Storage – helps HOTKEY READS

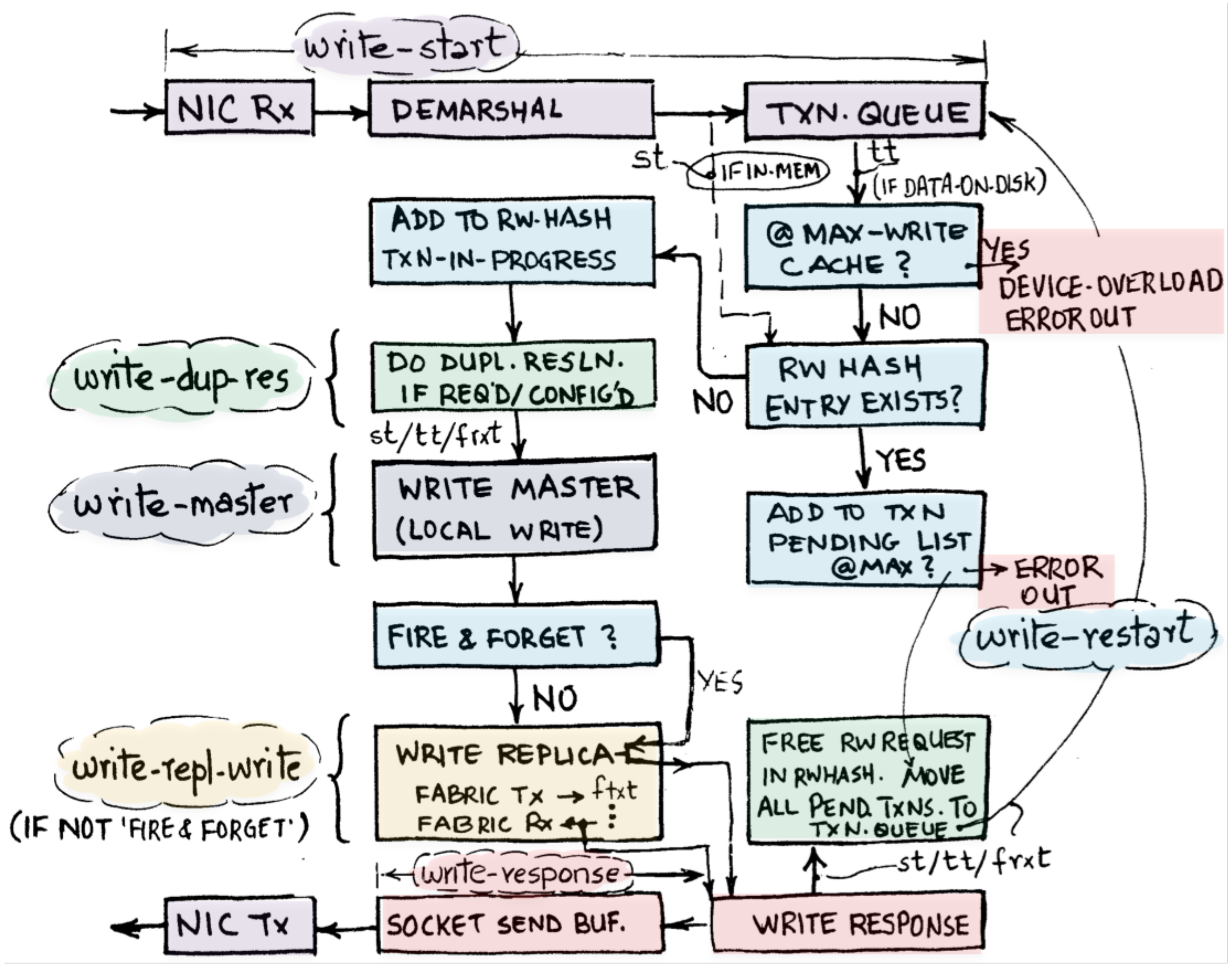


device: read-page-cache – uses O_DIRECT + O_DSYNC for writes only.

- Write transactions bypass linux page cache, immed. flush from disk controller cache to disk.
- Enables caching on reads only.

Write Transaction Flow

- During migrations, writes always Duplicate Resolve by default. (fabric)
- Writes also replicate to another node. (fabric)
- Locally, writes always write to RAM. It is eventually flushed to disk.

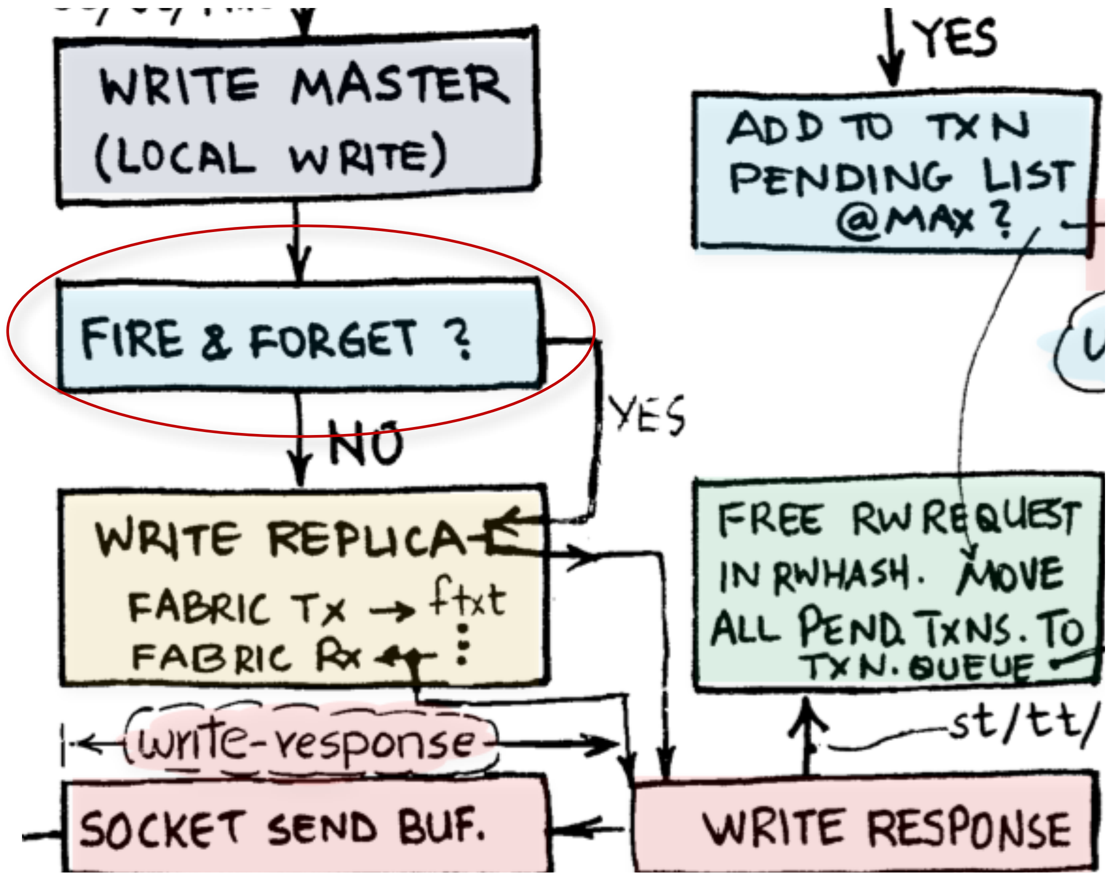


Speeding up Write Transactions – "Fire and Forget"

- Writes in *AP mode*, offer "fire-and-forget" i.e. don't wait for confirmation of replica writes. *Is it a viable option for "hot-key" update data models?*
- Configure namespace (all records): *write-commit-level-override master* OR per record via write policy (safer). *Java: commitLevel COMMIT_MASTER*

Fire-and-Forget:

- Sends Replica Write but does not wait for Replica Write response.
- Sends response to client.
- Frees the rwHash.
- *However, monitor RAM usage for queued up writes on fabric in the Socket Send Buffer.*



Write Transactions – "Fire and Forget"

Data Modeling question to answer:

- If a write fails or has a TIMEOUT, what does the data model demand? Retry?
- State of the record on the server is unknown.
- Depending on where the network or node failure happened, just the master, or, both master and replica, or, none of them could have the write – we just don't know at the client side.

How are you handling TIMEOUT errors? If using increment() – how is TIMEOUT handled?

Does knowing that replication happened, help with the data model?

Can you use "Fire and Forget"? (You can if you don't care about TIMEOUT Errors!)

Q&A?

