MRVs: Enforcing Numeric Invariants in Parallel Updates to Hotspots with Randomized Splitting[(*)]

Nuno Faria José Pereira

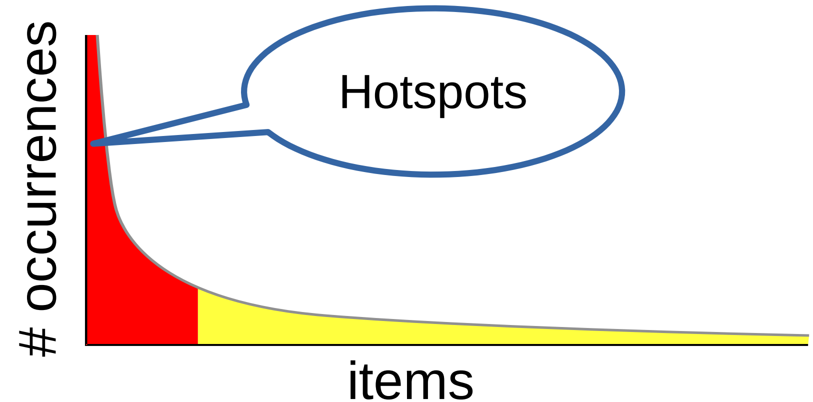
{nuno.f.faria,jose.o.pereira}@inesctec.pt

Universidade do Minho

[(*)] To appear in SIGMOD 2023

Motivation: Update hotspots

- Read data access distribution is usually highly skewed
- Consequences:
  - Delays (pessimistic c.c.)
  - Rollbacks (optimistic c.c.)
  - Unbalanced shards



(image from Wikipedia)

Motivation: Update hotspots

- Focus on numeric values with a lower bound
- Operations:
  - Add to value
  - Subtract from value if result >= 0
  - Read current value
  - (Overwrite value)
- Part of larger, multi-operation, transactions
- Applications: Pre-paid account, finite stock, ...

Goal

- Mechanism that mitigates the impact of hotspots in numeric values
- Layered on existing transactional system
  - No changes to underlying c.c.
  - Preserves isolation criteria
  - No additional coordination -> decentralized

Related work

- Escrow locking [O'Neil, 1986]
- CRDTs [Balegas et al., 2015]
- RedBlue [Li et al., 2012]
- <u>Reservations</u> [Barbará-Milla et al., 1994]
- <u>Splitting</u> [Narula et al., 2014]

- Proposal: **Multi-Record Values (MRVs)** are a form of <u>reservations / splitting</u>

Challenge

- How to assign records to operations without knowledge of cores / nodes / ... ?
- What if the chosen record is not enough?
- How to dynamically change the number of records for each value?

Assumptions

- Multi-item transactions
  - Repeatable Read, Snapshot Isolation, (Serializability)
- Dynamic tree-structured index
  - Range queries and concurrent updates
- (Query processing, views, rules, ...)
- Examples:
  - PostgreSQL, MongoDB, MySQL GR, Google Spanner, ...

Step 1: Randomization

- Split each contended value to multiple database records
- Each operation accesses a random record *i* out of *n*
- In contrast to:
  - phase reconciliation: 1 partition / core
  - reservations: 1 partition / node
- The probability of conflict can be made arbitrarily small by increasing the number of partitions

Step 2: Range traversal

- Use record *i+1*, *i+2* -> not random again!
  - wrap around at *n* to *0* -> circular structure
  - stop at *i-1*
- In contrast to:
  - transitioning to “joined phase”
  - contacting “home” node
- Easy to detect termination
- Avoids deadlocks

Step 3: Sparse keys

- Assign a random key to each of *n* records out of *N* (*N* >> *n*)
  - lookup lowest with key >= *i*
    - as efficient as lookup of *i* in tree structured indexes
- Insertion and removal of records does not conflict with updates of other records
- In contrast to:
  - transition to “joined phase” and back
  - centralized coordination by “home” node

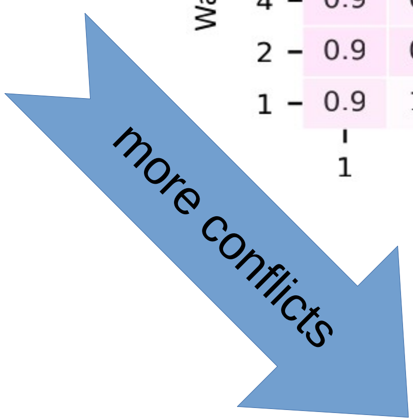
Implementation

- Maintenance of records:
  - Adjusting the number of records to the workload
  - Balancing the value in records
- Can be done in the background by concurrent worker threads
- Can be approximate and decentralized

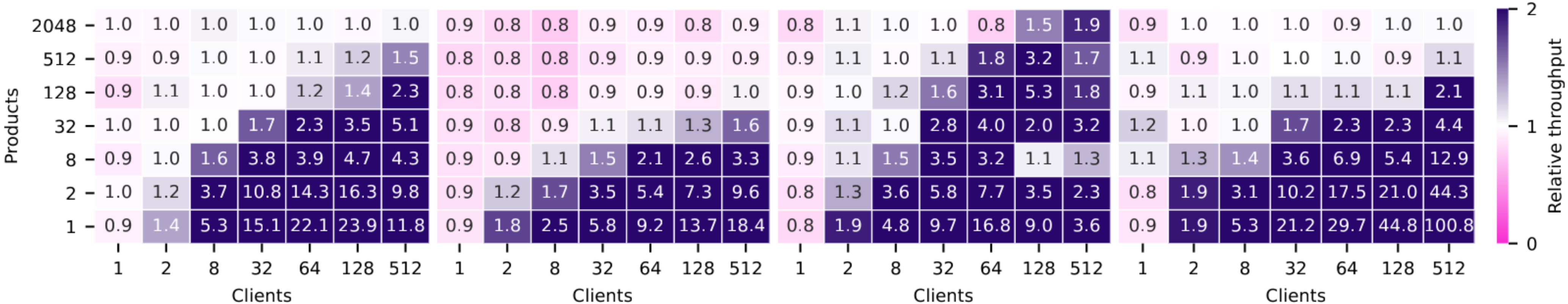
Results: Workloads

(b) *TPC-C*

(c) *STAMP Vacation (optimized)*



Results: DBMSs



(a) ***Single-writer SQL*** **(PostgreSQL)**

(b) ***Single-writer NoSQL*** **(MongoDB)**

(c) ***Multi-writer SQL*** **(MySQL Group Replication)**

(d) ***Multi-writer cloud-native NewSQL*** **(Google Spanner)**

Conclusions

- New take on a classical problem, motivated by a new generation of database systems (NewSQL)
- Part of an ongoing project to rethink distributed database systems architectures