Parameterized and Runtime-tunable Snapshot Isolation in Distributed Transactional Key-value Stores

Hengfeng Wei, Yu Huang, Jian Lu

Nanjing University, China

September 14, 2017

RVSI: Relaxed Version Snapshot Isolation

1. Motivation for RVSI
2. Definition of RVSI
3. CHAMELEON Prototype and RVSI Protocol

RVSI: Relaxed Version Snapshot Isolation

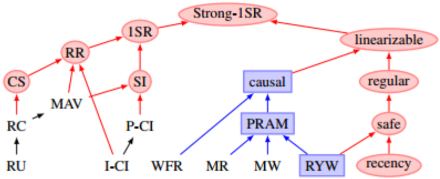
Distributed key-value stores

`put(K key, V val)`   `get(K key)`

Transactional semantics

existential consistency atomic visibility example

Transactional consistency models



Snapshot isolation (SI):

- Read from the latest consistent snapshot of all data items
- No write conflicts among concurrent transactions

Distributed SI

- Costs of SI
- Relaxed variants of SI:
  - GSI [1]
  - NMSI [2]
  - PL-FCV [3]
  - PSI [4]

[1] GSI: Generalized Snapshot Isolation
[2] NMSI: Non-Monotonic Snapshot Isolation
[3] PL-FCV: Forward Consistent View
[4] PSI: Parallel Snapshot Isolation

Two drawbacks

1. Unbounded inconsistency
   - no specification of the severity of the anomalies w.r.t SI
2. Untunable at runtime
   - determined at the system design phase
   - remain unchanged once the system is deployed

A Motivating Example

The *Books* table.

Title	Authors	Publisher	Sales	Inventory	Ratings	Reviews	$\cdots$

Customer:

Bookstore Clerk:

Sales Analyst:

Contributions

RVSI: Relaxed Version Snapshot Isolation

Principle of RVSI

- Parameters (k_1, k_2, k_3) to control the severity of the anomalies w.r.t SI

[5] RC: Read Committed

Principle of RVSI

- Parameters (k_1, k_2, k_3) to control the severity of the anomalies w.r.t SI
- RC [5] $\supset$ RVSI$(k_1, k_2, k_3) \supset$ SI
- RVSI$(\infty, \infty, \infty) =$ RC RVSI$(1, 0, *) =$ SI

[5] RC: Read Committed

Principle of RVSI

. . .

– *"Snapshot Read" property of SI*

1. "stale" data versions

Principle of RVSI

. . .

– *"Snapshot Read" property of SI*

1. "stale" data versions
2. "concurrent" data versions

Principle of RVSI

. . .

– *"Snapshot Read" property of SI*

1. "stale" data versions
2. "concurrent" data versions
3. "non-snapshot" data versions

Principle of RVSI

...

– *"Snapshot Read" property of SI*

1. "stale" data versions — bounded staleness
2. "concurrent" data versions
3. "non-snapshot" data versions

Principle of RVSI

. . .

– *"Snapshot Read" property of SI*

1. "stale" data versions — bounded staleness
2. "concurrent" data versions — bounded concurrency level
3. "non-snapshot" data versions

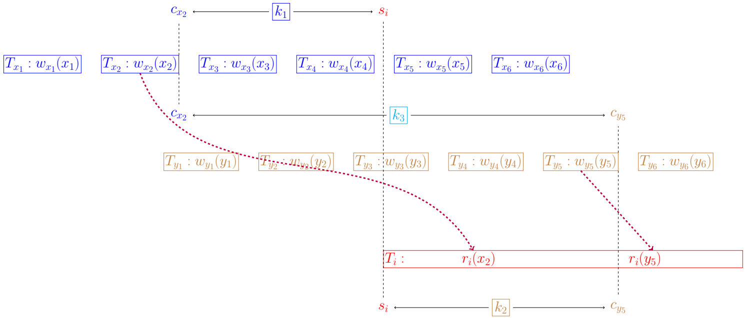
Principle of RVSI

. . .

– *"Snapshot Read" property of SI*

1. "stale" data versions — bounded staleness
2. "concurrent" data versions — bounded concurrency level
3. "non-snapshot" data versions — bounded distance

Illustration of RVSI



Definition of RVSI

(k_1-BV)

$$\forall r_i(x_j), w_k(x_k), c_k \in h : \left(c_j \in h \land \bigwedge_{k=1}^{m} (c_j \prec_h c_k \prec_h s_i) \right) \Rightarrow m < k_1,$$

(k_2-FV)

$$\forall r_i(x_j), w_k(x_k), c_k \in h : \left(c_j \in h \land \bigwedge_{k=1}^{m} (s_i \prec_h c_k \prec_h c_j) \right) \Rightarrow m \leq k_2,$$

(k_3-SV)

$$\forall r_i(x_j), r_i(y_l), w_k(x_k), c_k \in h : \left(\bigwedge_{k=1}^{m} (c_j \prec_h c_k \prec_h c_l) \right) \Rightarrow m \leq k_3.$$

Definition of RVSI

$$h \in \mathsf{RVSI} \iff h \in k_1\text{-}\mathsf{BV} \cap k_2\text{-}\mathsf{FV} \cap k_3\text{-}\mathsf{SV} \cap \mathsf{WCF}.$$

Theorem

$$\mathsf{RVSI}(1, 0, *) = \mathsf{SI}.$$

RVSI: Relaxed Version Snapshot Isolation

Chameleon Protocol

Chameleon:

A prototype **partitioned replicated**

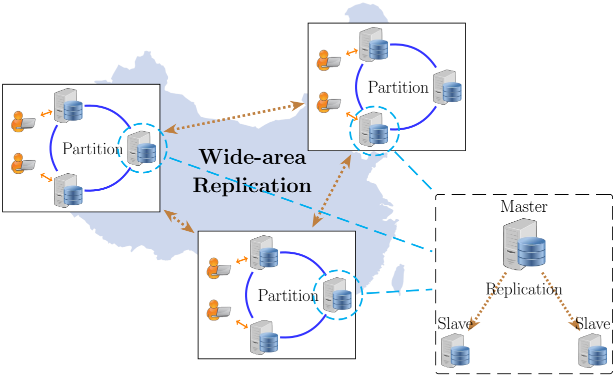
distributed transactional **key-value** store

Chameleon Prototype

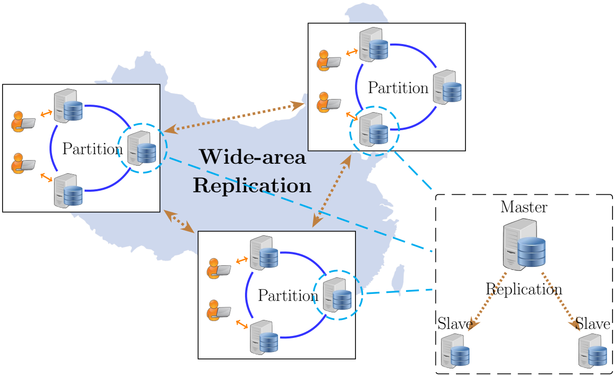
Classic **key-value** data model

Key: (row key, column key)

Chameleon Prototype

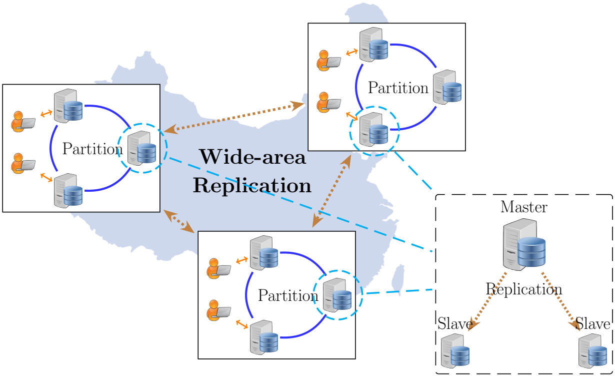


Chameleon Prototype



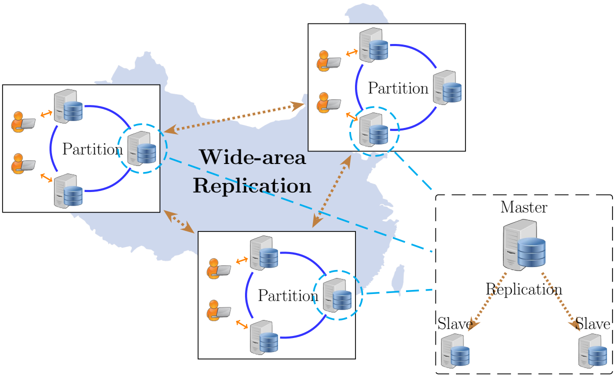
Keys are **partitioned** within a single datacenter.

CHAMELEON Prototype



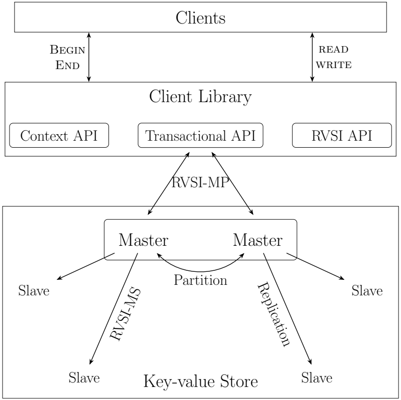
Each key is **replicated** in a master-slave manner across datacenters.

CHAMELEON Prototype

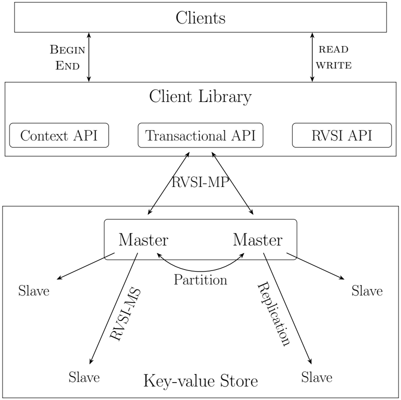


Transactions are first executed and committed on the masters, and are then asynchronously propagated to slaves.

Chameleon Protocol

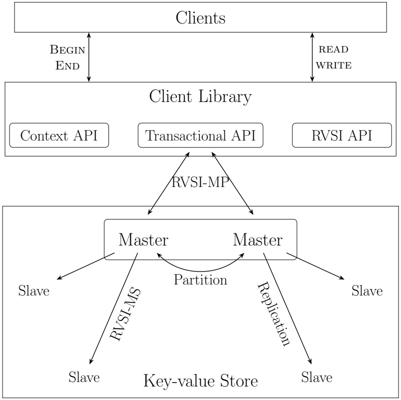


Chameleon Protocol



Partitioned replicated transactional key-value store

CHAMELEON Protocol



Client library

CHAMELEON Protocol

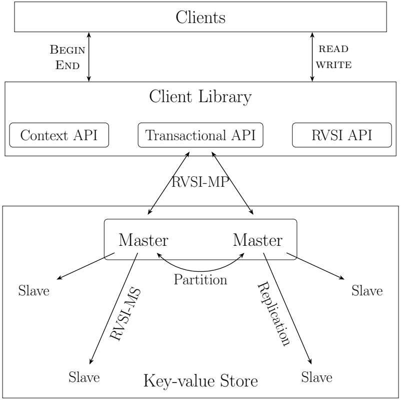
```
// Initialize keys (ck, ck1, and ck2) here
ITx tx = new RVSITx(/** context **/);

tx.begin();

// Read and write
ITsCell tsCell = tx.read(ck);
ITsCell tsCell1 = tx.read(ck1);
tx.write(ck1, new Cell("R1C1"));
ITsCell tsCell2 = tx.read(ck2);

// Specify RVSI specs. (e.g., SVSpec)
RVSISpec sv = new SVSpec();
sv.addSpec({ck, ck1, ck2}, 2);
tx.collectRVSISpec(sv);
```

Chameleon Protocol



RVSI protocol: RVSI-MS + RVSI-MP

RVSI-MS Protocol

RVSI-MS: RVSI protocol for master-slave replication

In terms of *event* generation and handling:

Clients: Begin, Read, Write, End

Master: Start, Commit, Send

Slaves: Receive

RVSI-MS Protocol

TikZ overlay for RVSI-MS

RVSI-MS Protocol

Calculating version constraints:

RVSI-MP Protocol

Distributed transactions spanning multiple masters need to be committed atomically.

RVSI-MP Protocol

Distributed transactions spanning multiple masters need to be committed atomically.

Using the two-phase commit (2PC) protocol.

RVSI-MP Protocol

Assumes a timestamp oracle:

Clients:

Masters:

RVSI-MP Protocol

RVSI version constraints in 2PC protocol:

k_1-BV:

k_2-FV:

k_3-SV:

RVSI-MS Protocol

Algorithm 1 RVSI-MS: RVSI Protocol for Replication (for Executing Transaction T).

Client-side methods:

```
procedure BEGIN()
    T.sts ← rpc-call START() at master 𝓜
procedure READ(x)
    x.ver ← rpc-call READ(x) at any site
procedure WRITE(x, v)
    add (x, v) to T.writes
procedure END(T)
    T.vc ← ADD-VC()
    c/a ← rpc-call COMMIT(T.writes, T.vc) at 𝓜
```

RVSI-MP Protocol

Algorithm 2 RVSI-MP: RVSI Protocol for Partition (for Executing Transaction T).

Client-side methods:

procedure $\textsc{Begin}()$
  return rpc-call $\textsc{GetTS}()$ at $\mathcal{T}$
procedure $\textsc{End}()$
  $T.vc \leftarrow \textsc{Add-VC}()$
  $c/a \leftarrow$ **rpc-call** $\textsc{C-Commit}(T.writes, T.vc)$ at $\mathcal{C}$

Timestamp oracle methods:

$\mathcal{T}.ts$: for start-timestamps and commit-timestamps
procedure $\textsc{GetTS}()$
  return $++\mathcal{T}.ts$