

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

Hengfeng Wei, Yu Huang, Jian Lu

Nanjing University, China

September 26, 2017



Parameterized and Runtime-tunable Snapshot Isolation

RVSI: Relaxed Version Snapshot Isolation

- 1 Motivation for RVSI
- 2 Definition of RVSI
- 3 CHAMELEON Prototype
- 4 Experimental Evaluation
- 5 Related Work
- 6 Conclusion

Parameterized and Runtime-tunable Snapshot Isolation

RVSI: Relaxed Version Snapshot Isolation

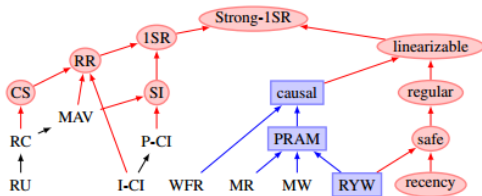
- 1 Motivation for RVSI
- 2 Definition of RVSI
- 3 CHAMELEON Prototype
- 4 Experimental Evaluation
- 5 Related Work
- 6 Conclusion

Distributed key-value stores:



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

Transactions are performed on a group of keys in an “all-or-none” way.



Transactional consistency models (from [Bailis@VLDB'14])

Snapshot isolation (SI [Berenson@SIGMOD'95], [Adya@Thesis'99]):

- ▶ Each transaction **reads** from the “latest” snapshot as of the time it started.
- ▶ If multiple concurrent transactions **write** to the same data item, at most one of them can commit. (WCF: *write-conflict freedom*)

Reading the “latest” in a distributed setting
often requires intensive coordinations.

¹GSI: Generalized Snapshot Isolation [Elnikety@SRDS'05]

²NMSI: Non-Monotonic Snapshot Isolation [Ardekani@SRDS'13]

³PL-FCV: Forward Consistent View [Aday@Thesis'99]

⁴PSI: Parallel Snapshot Isolation [Sovran@SOSP'11]

Reading the “latest” in a distributed setting often requires intensive coordinations.

Relaxed variants of (distributed) SI:

GSI¹: allows to read from “older” snapshots

NMSI²: allows to observe non-monotonically ordered snapshots

PL-FCV³: allows a transaction to observe the updates of transactions that commit after it started

PSI⁴: causal ordering of transactions across sites

¹GSI: Generalized Snapshot Isolation [Elnikety@SRDS'05]

²NMSI: Non-Monotonic Snapshot Isolation [Ardekani@SRDS'13]

³PL-FCV: Forward Consistent View [Aday@Thesis'99]

⁴PSI: Parallel Snapshot Isolation [Sovran@SOSP'11]

Two possible drawbacks:

1. Unbounded inconsistency

- ▶ no specification of the severity of the anomalies w.r.t SI

Two possible 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

The idea of “parameterized and runtime-tunable snapshot isolation”.

¹<https://github.com/hengxin/chameleon-transactional-kvstore>

²<http://www.aliyun.com/>

The idea of “**parameterized** and **runtime-tunable** snapshot isolation”.

RVSI: Relaxed Version Snapshot Isolation

k_1 -BV: k_1 -version bounded *backward* view

k_2 -FV: k_2 -version bounded *forward* view

k_3 -SV: k_3 -version bounded *snapshot* view

¹<https://github.com/hengxin/chameleon-transactional-kvstore>

²<http://www.aliyun.com/>

The idea of “**parameterized** and **runtime-tunable** snapshot isolation”.

RVSI: Relaxed Version Snapshot Isolation

k_1 -BV: k_1 -version bounded *backward* view

k_2 -FV: k_2 -version bounded *forward* view

k_3 -SV: k_3 -version bounded *snapshot* view

CHAMELEON¹: a prototype distributed transactional key-value store

- ▶ Achieves RVSI
- ▶ Allows each transaction to tune its consistency level at runtime

¹<https://github.com/hengxin/chameleon-transactional-kvstore>

²<http://www.aliyun.com/>

The idea of “**parameterized** and **runtime-tunable** snapshot isolation”.

RVSI: Relaxed Version Snapshot Isolation

k_1 -BV: k_1 -version bounded *backward* view

k_2 -FV: k_2 -version bounded *forward* view

k_3 -SV: k_3 -version bounded *snapshot* view

CHAMELEON¹: a prototype distributed transactional key-value store

- ▶ Achieves RVSI
- ▶ Allows each transaction to tune its consistency level at runtime
- ▶ Deployed on Alibaba Cloud (Aliyun)²
- ▶ Evaluates the impacts of RVSI on the transaction abort rates

¹<https://github.com/hengxin/chameleon-transactional-kvstore>

²<http://www.aliyun.com/>

Parameterized and Runtime-tunable Snapshot Isolation

RVSI: Relaxed Version Snapshot Isolation

- 1 Motivation for RVSI
- 2 Definition of RVSI
- 3 CHAMELEON Prototype
- 4 Experimental Evaluation
- 5 Related Work
- 6 Conclusion

Transaction T_i : $s_i \ (r_i/w_i)^+ \ c_i/a_i$

s_i : start operation

r_i/w_i : read/write operation

c_i/a_i : commit/abort operation

Transaction T_i : $s_i \ (r_i/w_i)^+ \ c_i/a_i$

s_i : start operation

r_i/w_i : read/write operation

c_i/a_i : commit/abort operation

x_i : version i of data item x written by T_i

$r_i(x_j)$: transaction T_i reading x_j

$w_i(x_i)$: transaction T_i writing x_i

Transaction T_i : $s_i \ (r_i/w_i)^+ \ c_i/a_i$

s_i : start operation

r_i/w_i : read/write operation

c_i/a_i : commit/abort operation

x_i : version i of data item x written by T_i

$r_i(x_j)$: transaction T_i reading x_j

$w_i(x_i)$: transaction T_i writing x_i

History: modelling an execution of a transactional key-value store

- ▶ *time-precedes partial order* \prec_h over all operations

A history h is in *snapshot isolation* iff it satisfies [Adya@Thesis'99]

Snapshot Read: Each transaction reads data from the “latest” snapshot as of the time it started.

$$\begin{aligned} &\forall r_i(x_{j \neq i}), w_{k \neq j}(x_k), c_k \in h : \\ &\quad (c_j \in h \wedge c_j \prec_h s_i) \wedge (s_i \prec_h c_k \vee c_k \prec_h c_j). \end{aligned}$$

A history h is in *snapshot isolation* iff it satisfies [Adya@Thesis'99]

Snapshot Read: Each transaction reads data from the “latest” snapshot as of the time it started.

$$\begin{aligned} \forall r_i(x_{j \neq i}), w_{k \neq j}(x_k), c_k \in h : \\ (c_j \in h \wedge c_j \prec_h s_i) \wedge (s_i \prec_h c_k \vee c_k \prec_h c_j). \end{aligned}$$

Snapshot Write: No concurrent committed transactions may write the same data item. (WCF: write-conflict freedom)

$$\forall w_i(x_i), w_{j \neq i}(x_j) \in h \implies (c_i \prec_h s_j \vee c_j \prec_h s_i).$$

Principles of RVSI:

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

¹RC: Read Committed isolation.

Principles of RVSI:

- ▶ Using parameters (k_1, k_2, k_3) to control the severity of the anomalies w.r.t SI
- ▶ $RC^1 \supset RVSI(k_1, k_2, k_3) \supset SI$
- ▶ $RVSI(\infty, \infty, \infty) = RC$ $RVSI(1, 0, *) = SI$

¹RC: Read Committed isolation.

Each transaction reads data from the “latest” snapshot as of the time the transaction started.

– The “Snapshot Read” property of SI

RVSI relaxes “Snapshot Read” in three ways:

Each transaction reads data from the “latest” snapshot as of the time the transaction started.

– *The “Snapshot Read” property of SI*

RVSI relaxes “Snapshot Read” in three ways:

k_1 -BV (Backward View): “stale” data versions

staleness $\leq k_1$

Each transaction reads data from the “latest” snapshot as of the time the transaction started.

– The “Snapshot Read” property of SI

RVSI relaxes “Snapshot Read” in three ways:

k_1 -BV (Backward View): “stale” data versions

staleness $\leq k_1$

k_2 -FV (Forward View): “forward” data versions

forward level $\leq k_2$

Each transaction reads data from the “latest” snapshot as of the time the transaction started.

– The “Snapshot Read” property of SI

RVSI relaxes “Snapshot Read” in three ways:

- k_1 -BV (Backward View): “stale” data versions staleness $\leq k_1$
- k_2 -FV (Forward View): “forward” data versions forward level $\leq k_2$
- k_3 -SV (Snapshot View): “non-snapshot” data versions distance $\leq k_3$

$(k_1\text{-BV})$

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

 $(k_2\text{-FV})$

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

 $(k_3\text{-SV})$

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

$(k_1\text{-BV})$

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

 $(k_2\text{-FV})$

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

 $(k_3\text{-SV})$

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

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

Parameterized and Runtime-tunable Snapshot Isolation

RVSI: Relaxed Version Snapshot Isolation

- 1 Motivation for RVSI
- 2 Definition of RVSI
- 3 CHAMELEON Prototype**
- 4 Experimental Evaluation
- 5 Related Work
- 6 Conclusion

CHAMELEON:

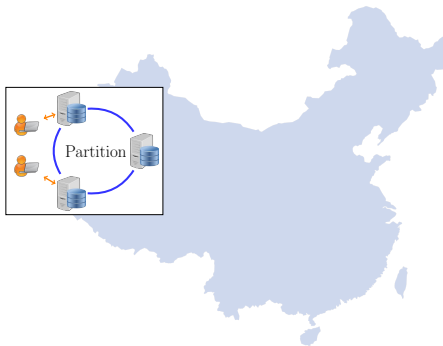
A prototype **partitioned replicated**
distributed transactional **key-value** store

CHAMELEON:

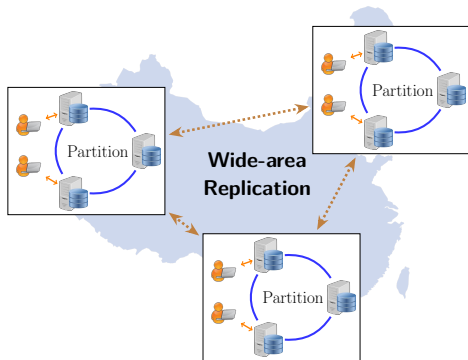
A prototype **partitioned replicated**
distributed transactional **key-value** store

Key: (row key, column key)

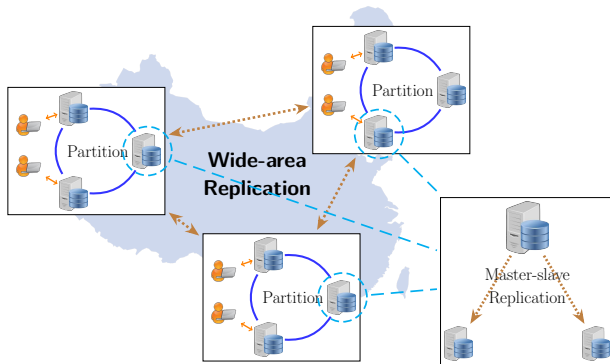




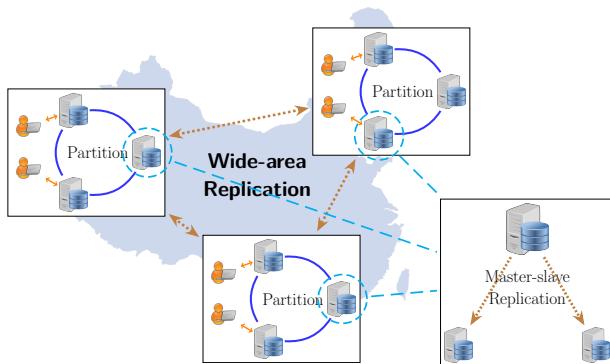
Keys are **partitioned** within a single datacenter.



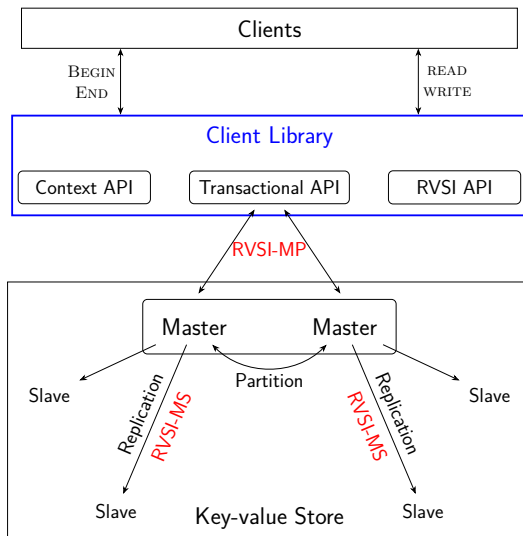
Each key is **replicated** across datacenters



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



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



Client library

Code snippet for writing RVSI transactions:

```
// Initialize keys (ck1 and ck2) here
...

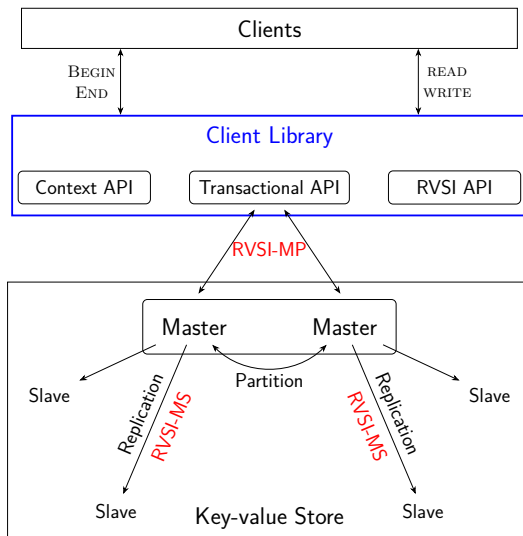
ITx tx = new RVSITx(** context **);

tx.begin();

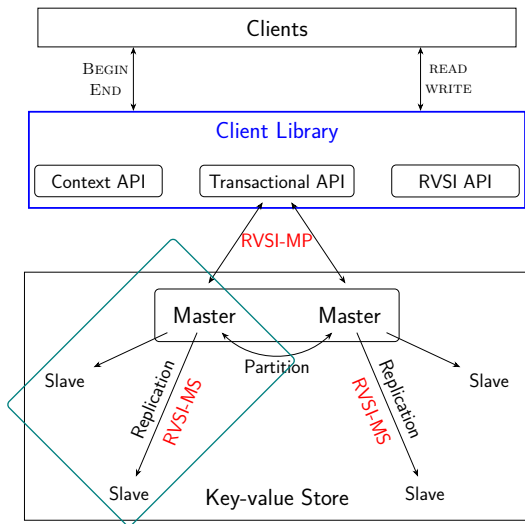
// Read and write here ...

// Specify RVSI specs. (e.g., SVSpec)
RVISpec sv = new SVSpec();
sv.addSpec({ck1, ck2}, 2);
tx.collectRVISpec(sv);

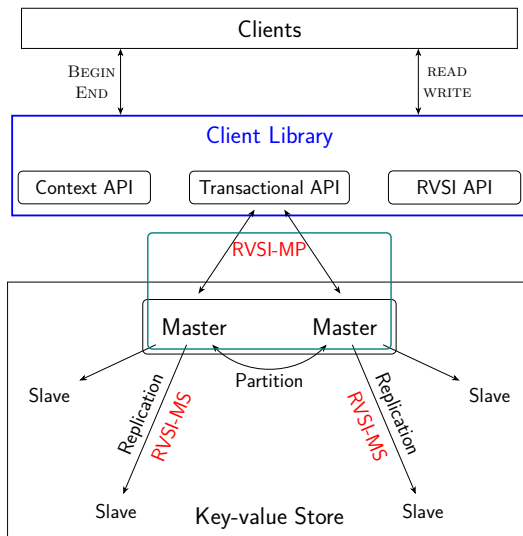
boolean committed = tx.end();
```



RVSI protocol: RVSI-MS + RVSI-MP

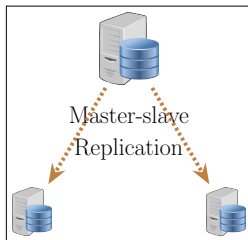


RVSI-MS: RVSI for Master-Slave replication

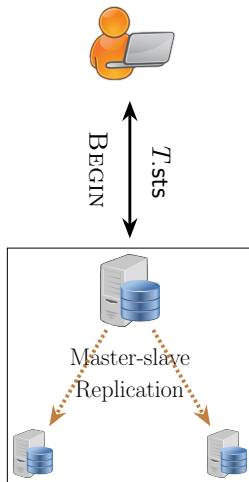


RVSI-MP: RVSI for Multiple Partitions

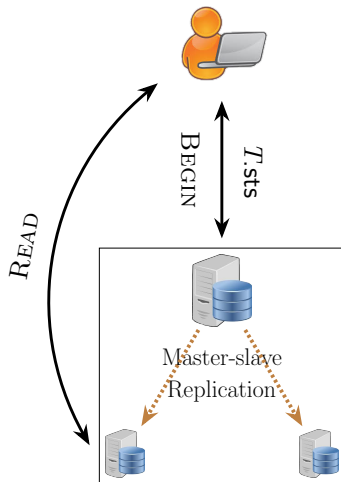
RVSI-MS: RVSI protocol for Master-Slave replication



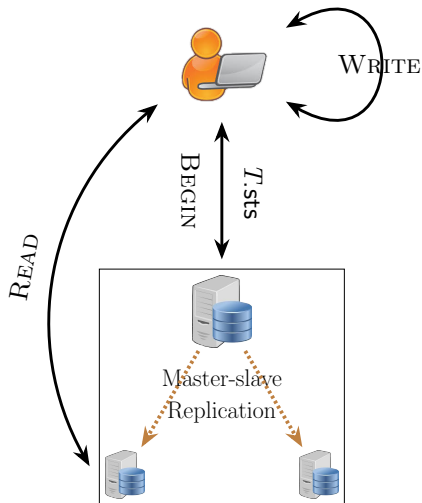
RVSI-MS: RVSI protocol for Master-Slave replication



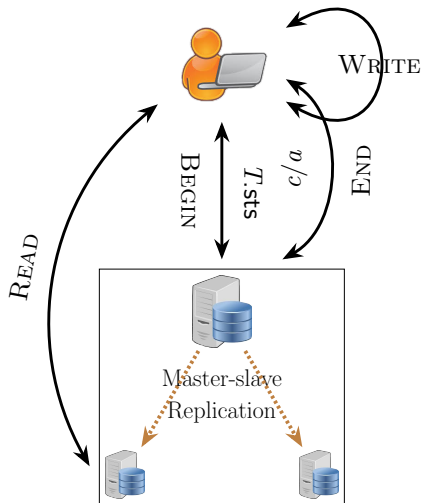
RVSI-MS: RVSI protocol for Master-Slave replication



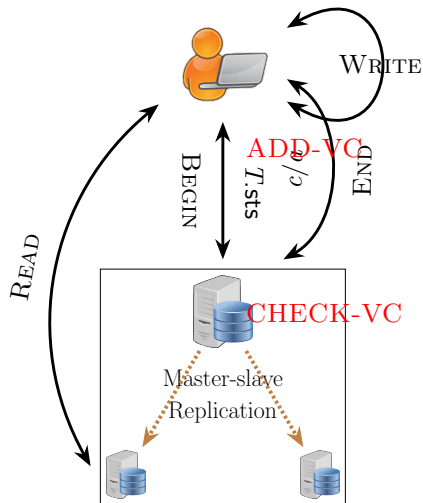
RVSI-MS: RVSI protocol for Master-Slave replication



RVSI-MS: RVSI protocol for Master-Slave replication



RVSI-MS: RVSI protocol for Master-Slave replication



$\mathcal{O}_x(t)$ = version NO. of x before time t

$$r_i(x_j) \in T_i$$

The version actually observed vs. The version just before T_i starts:

k_1 -BV:

$$\mathcal{O}_x(T_i.sts) - \mathcal{O}_x(T_j.cts) < k_1$$

k_2 -FV:

$$\mathcal{O}_x(T_j.cts) - \mathcal{O}_x(T_i.sts) \leq k_2$$

$\mathcal{O}_x(t)$ = version NO. of x before time t

$$r_i(x_j) \in T_i$$

The version actually observed vs. The version just before T_i starts:

k_1 -BV:

$$\mathcal{O}_x(T_i.sts) - \mathcal{O}_x(T_j.cts) < k_1$$

k_2 -FV:

$$\mathcal{O}_x(T_j.cts) - \mathcal{O}_x(T_i.sts) \leq k_2$$

$$r_i(x_j), r_i(y_l) \in T_i \quad (\text{Assume } T_j.cts < T_l.cts)$$

k_3 -SV:

The snapshot x_j is born in vs. The snapshot y_l is born

$$\mathcal{O}_x(T_l.cts) - \mathcal{O}_x(T_j.cts) \leq k_3$$

RVSI-MP: RVSI protocol for Multiple Partitions

Distributed transactions spanning multiple masters
need to be committed atomically.

RVSI-MP: RVSI protocol for Multiple Partitions

Distributed transactions spanning multiple masters
need to be committed atomically.

Using the two-phase commit (2PC) protocol [[Bernstein@Book'87](#)].

RVSI-MP: RVSI protocol for Multiple Partitions

Distributed transactions spanning multiple masters
need to be committed atomically.

Using the two-phase commit (2PC) protocol [[Bernstein@Book'87](#)].

We have two issues to address.

Assumes a global timestamp oracle [Peng@OSDI'10]:

Client: asks for the start-timestamp in BEGIN

Coordinator: asks for the commit-timestamp in COMMIT

Split the RVSI version constraints according to partitions:

$$r_i(x_j) \in T_i$$

k_1 -BV:

$$\mathcal{O}_x(T_i.sts) - \mathcal{O}_x(T_j.cts) < k_1$$

k_2 -FV:

$$\mathcal{O}_x(T_j.cts) - \mathcal{O}_x(T_i.sts) \leq k_2$$

$$r_i(x_j), r_i(y_l) \in T_i$$

k_3 -SV:

$$\mathcal{O}_x(T_l.cts) - \mathcal{O}_x(T_j.cts) \leq k_3$$

Each version constraint involves only one data item.

Parameterized and Runtime-tunable Snapshot Isolation

RVSI: Relaxed Version Snapshot Isolation

- 1 Motivation for RVSI
- 2 Definition of RVSI
- 3 CHAMELEON Prototype
- 4 Experimental Evaluation**
- 5 Related Work
- 6 Conclusion

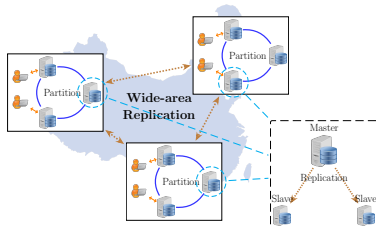
Impacts of RVSI specification on the *transaction abort rates* in various scenarios

Impacts of RVSI specification
on the *transaction abort rates* in various scenarios

Performance is *not* reported in this work
because it is *not* sensitive to the parameters k_1 , k_2 or k_3 .

CHAMELEON on Alibaba Cloud:

- ▶ 3 datacenters ¹
- ▶ 3 nodes in each datacenter
- ▶ Partition & Replication
- ▶ Clients in our lab ²



¹ Located in East China, North China, and South China, respectively.

² Located in East China.

Workload parameters for the experiments on Alibaba Cloud.

Parameter	Value	Explanation
#keys	$25 = 5 \text{ (rows)} \times 5 \text{ (columns)}$	
#clients	5, 10, 15, 20, 25, 30	
#txs/client	1000	
#ops/tx	$\sim \text{Binomial}(20, 0.5)$	
rwRatio	1:2, 1:1, 4:1	#reads/#writes
zipfExponent	1	parameter for Zipfian distribution
minInterval	0ms	min/max/mean inter-transaction time
maxInterval	10ms	
meanInterval	5ms	
(k_1, k_2, k_3)	(1,0,0) (1,1,0) (1,1,1) (2,0,0) (2,0,1) (2,1,1)	

Overview:

1. Transaction abort rates because of violating the RVSI version constraints (“vc-aborted”) are quite *sensitive* to different values of k_1 , k_2 , or k_3 .

¹<https://github.com/hengxin/chameleon-transactional-kvstore>

Overview:

1. Transaction abort rates because of **violating the RVSI version constraints** (“vc-aborted”) are quite *sensitive* to different values of k_1 , k_2 , or k_3 .
2. In the **Alibaba Cloud scenarios**, most transactions have been aborted because of violating k_2 -FV.

¹<https://github.com/hengxin/chameleon-transactional-kvstore>

Overview:

1. Transaction abort rates because of violating the RVSI version constraints (“vc-aborted”) are quite *sensitive* to different values of k_1 , k_2 , or k_3 .
2. In the Alibaba Cloud scenarios, most transactions have been aborted because of violating k_2 -FV.
3. In controlled experiments, the impacts of k_1 -BV emerge when the “issueDelay” gets shorter.

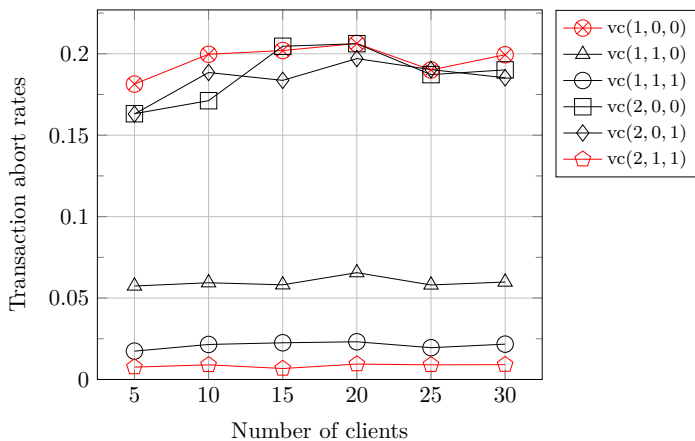
¹<https://github.com/hengxin/chameleon-transactional-kvstore>

Overview:

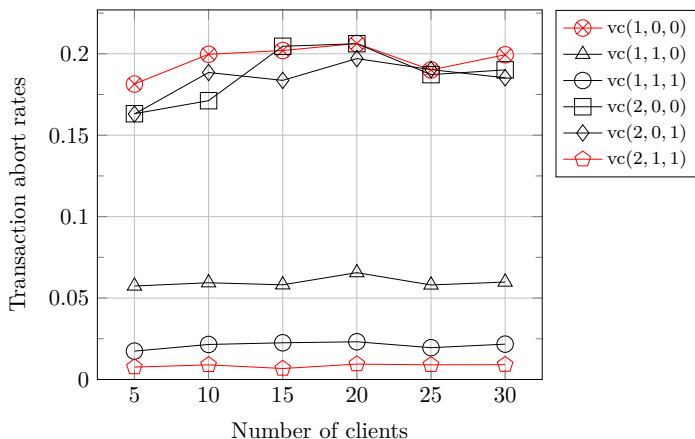
1. Transaction abort rates because of **violating the RVSI version constraints** (“vc-aborted”) are quite **sensitive** to different values of k_1 , k_2 , or k_3 .
2. In the **Alibaba Cloud scenarios**, most transactions have been aborted because of violating **k_2 -FV**.
3. In **controlled experiments**, the impacts of **k_1 -BV** emerge when the “issueDelay” gets shorter.

We report the results under the read-frequent ($\#rwRatio = 4:1$) workloads ¹.

¹<https://github.com/hengxin/chameleon-transactional-kvstore>

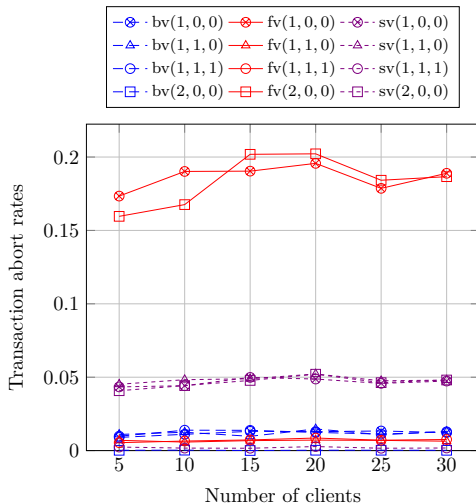


The transaction abort rates due to “vc-aborted”



The transaction abort rates due to “vc-aborted” can be **greatly reduced** by **slightly** increasing the values of k_1 , k_2 , or k_3 :

$$vc(1, 0, 0) = 0.1994 \implies vc(2, 1, 1) = 0.0091 \quad (\#clients = 30)$$



Most “vc-aborted” transactions abort because of violating k_2 -FV.

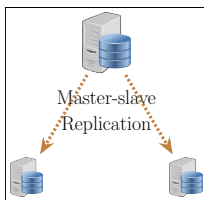
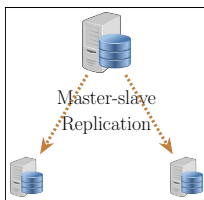
$$fv(1, 0, 0) = 0.1889 \implies fv(2, 0, 0) = 0.1866 \implies fv(1, \mathbf{1}, 0) = 0.0064$$

In the Alibaba Cloud scenarios, we saw *little* impacts of k_1 -BV.

In the Alibaba Cloud scenarios, we saw *little* impacts of k_1 -BV.

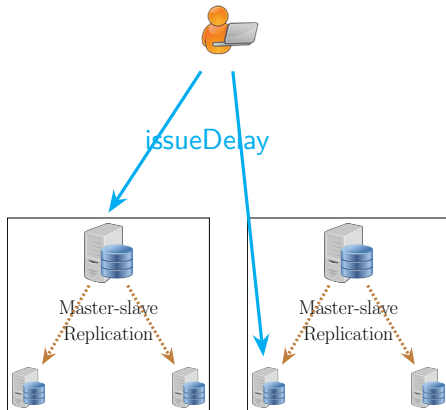
We therefore explore the impacts of k_1 -BV in controlled experiments.

Three types of delays for **controlled experiments** on local hosts.



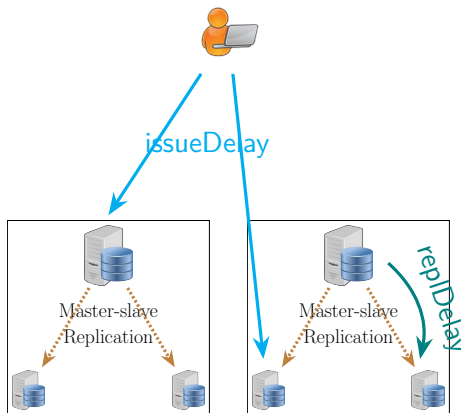
Types	Values (ms)	Explanation
-------	-------------	-------------

Three types of delays for **controlled experiments** on local hosts.



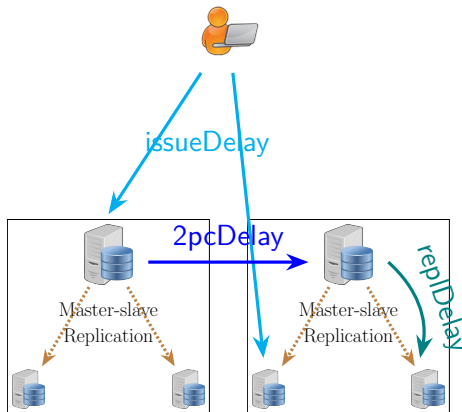
Types	Values (ms)	Explanation
issueDelay	5, 8, 10, 12, 15, 20	delays between clients and replicas

Three types of delays for **controlled experiments** on local hosts.

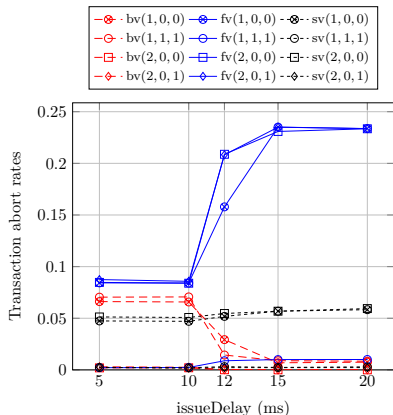


Types	Values (ms)	Explanation
replDelay	5, 10, 15, 20, 30	delays between masters and slaves

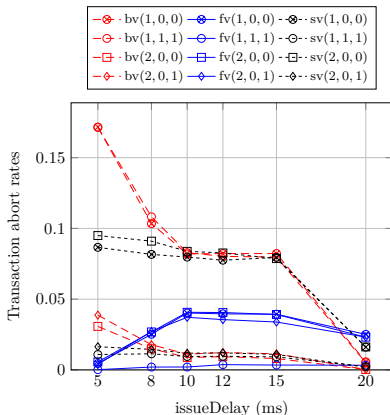
Three types of delays for **controlled experiments** on local hosts.



Types	Values (ms)	Explanation
2pcDelay	10, 20, 30, 40, 50	delays among masters



(Under read-frequent workloads.)



(Under write-frequent workloads.)

When the “**issueDelay**” gets shorter,
the impacts of k_1 -BV have begun to emerge.

What about the impacts of k_3 -SV?

- ▶ k_3 -SV involves multiple data items
- ▶ Complex and challenging
- ▶ Have not found any simple yet significant patterns

Parameterized and Runtime-tunable Snapshot Isolation

RVSI: Relaxed Version Snapshot Isolation

- 1 Motivation for RVSI
- 2 Definition of RVSI
- 3 CHAMELEON Prototype
- 4 Experimental Evaluation
- 5 Related Work**
- 6 Conclusion

The idea of “**bounded transactional inconsistency**” is partly inspired by the work on

- ▶ Relaxed Currency and Consistency (C&C) semantics [Guo@SIGMOD'04]
- ▶ Relaxed Currency Serializability (RC-SR) [Bernstein@SIGMOD'06]

Two main differences:

- ▶ Serializability (SR) vs. SI
- ▶ Currency in real-time vs. Versions in order

Bounded transactional inconsistency (others):

Epsilon-SR inconsistency introduced by concurrent update transactions [Pu@SIGMOD'91] [Ramamritham@TKDE'95]

- ▶ uncommitted vs. RC (for RVSI)

N-ignorant System ignorant of $\leq K$ “prior” transactions [Krishnakumar@PODS'91]

- ▶ SR vs. SI (for RVSI)

Dynamic consistency choices:

Parameterized ESR, N -ignorant, RC-SR, C&C semantics, **RVSI**

Pileus strong, intermediate, and eventual consistency
[Kotla@MSR-TR'2013]

SIEVE a tool automating the choice of consistency levels
[Li@ATC'14]
(based on the theory of RedBlue consistency [Li@OSDI'12])

Salt combining ACID and BASE transactions [Xie@OSDI'14]

Multi-level a transaction model supporting four consistency levels
[Tripathi@BigData'15]

Parameterized and Runtime-tunable Snapshot Isolation

RVSI: Relaxed Version Snapshot Isolation

- 1 Motivation for RVSI
- 2 Definition of RVSI
- 3 CHAMELEON Prototype
- 4 Experimental Evaluation
- 5 Related Work
- 6 Conclusion

The idea of “parameterized and runtime-tunable snapshot isolation”.

RVSI: Relaxed Version Snapshot Isolation

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

CHAMELEON: a prototype distributed transactional key-value store

- ▶ Allows each transaction to tune its consistency level at runtime
- ▶ Evaluates the impacts of RVSI on the transaction abort rates

Two possible future work:

- ▶ To evaluate the impacts of k_3 -SV on transaction abort rates, probably with data mining technologies
- ▶ To study the impacts/anomalies of RVSI from the perspectives of developers

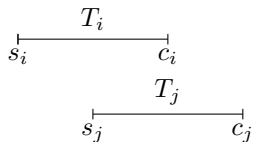


hfwei@nju.edu.cn



Two transactions are *concurrent* if

$$s_i \prec_h c_j \wedge s_j \prec_h c_i$$



An online bookstore application ¹ for motivating
“bounded inconsistency” and “runtime-tuable”:

Title	Authors	Sales	Inventory	Ratings	Reviews	...
-------	---------	-------	-----------	---------	---------	-----

Customer (T_1): Obtaining the basic info. about a book
▶ *out-of-date* reviews

¹Adapted from [Guo@SIGMOD'04] and [Bernstein@SIGMOD'06].

An online bookstore application ¹ for motivating
“bounded inconsistency” and “runtime-tuable”:

Title	Authors	Sales	Inventory	Ratings	Reviews	...
-------	---------	-------	-----------	---------	---------	-----

Customer (T_1): Obtaining the basic info. about a book

- ▶ *out-of-date* reviews

Bookstore Clerk (T_2): Checking the inventory of a book

- ▶ updated by concurrent transactions that commit *after* T_2 starts

¹Adapted from [Guo@SIGMOD'04] and [Bernstein@SIGMOD'06].

An online bookstore application ¹ for motivating
“bounded inconsistency” and “runtime-tuable”:

Title	Authors	Sales	Inventory	Ratings	Reviews	...
-------	---------	-------	-----------	---------	---------	-----

Customer (T_1): Obtaining the basic info. about a book

- ▶ *out-of-date* reviews

Bookstore Clerk (T_2): Checking the inventory of a book

- ▶ updated by concurrent transactions that commit *after* T_2 starts

Sales Analyst (T_3): Studying sales vs. ratings of a book

- ▶ sales and ratings from *separate snapshots*

¹Adapted from [Guo@SIGMOD'04] and [Bernstein@SIGMOD'06].

Applicability

$$T_{x_1} : w_{x_1}(x_1)$$

$$T_{x_2} : w_{x_2}(x_2)$$

$$T_{x_3} : w_{x_3}(x_3)$$

$$T_{x_4} : w_{x_4}(x_4)$$

$$T_{x_5} : w_{x_5}(x_5)$$

$$T_{x_6} : w_{x_6}(x_6)$$

$$T_{y_1} : w_{y_1}(y_1)$$

$$T_{y_2} : w_{y_2}(y_2)$$

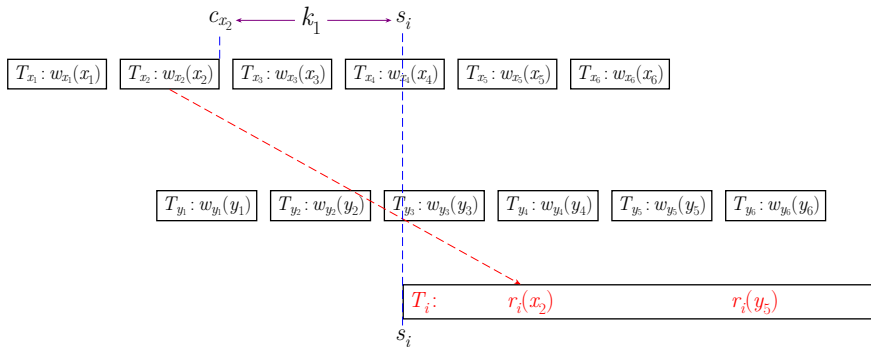
$$T_{y_3} : w_{y_3}(y_3)$$

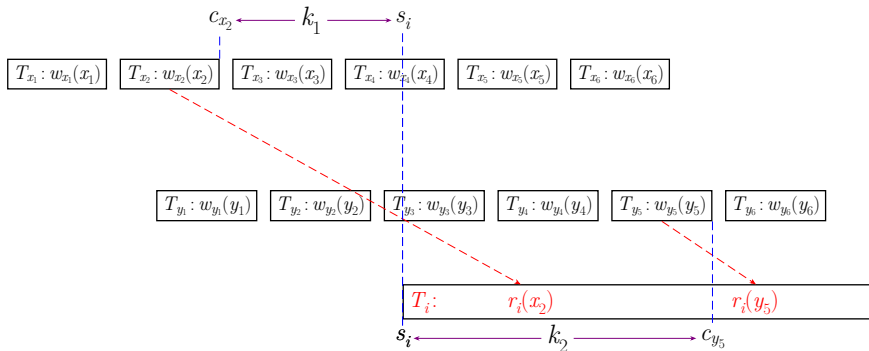
$$T_{y_4} : w_{y_4}(y_4)$$

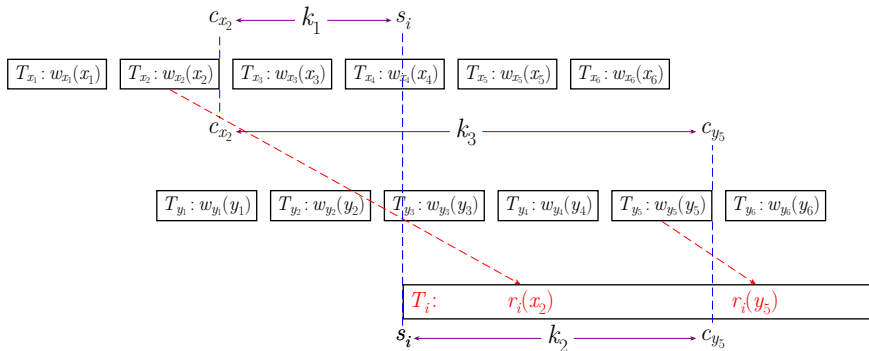
$$T_{y_5} : w_{y_5}(y_5)$$

$$T_{y_6} : w_{y_6}(y_6)$$

$$T_i : \quad r_i(x_2) \quad r_i(y_5)$$







For convenience, the definition of RVSI specifies the bounds k_1 , k_2 , and k_3 globally w.r.t a history.

However, they can be easily generalized to support dynamic bounds per transaction or even w.r.t each individual read operation or every pair of them.

In terms of *event* generation and handling:

Clients: BEGIN, READ, WRITE, END

Master: START, COMMIT, SEND

Slaves: RECEIVE

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

```
1: procedure BEGIN()  
2:    $T.sts \leftarrow \text{rpc-call START}()$  at master  $\mathcal{M}$   
3: procedure READ( $x$ )  
4:    $x.ver \leftarrow \text{rpc-call READ}(x)$  at any site  
5: procedure WRITE( $x, v$ )  
6:   add  $(x, v)$  to  $T.writes$   
7: procedure END( $T$ )  
8:    $T.vc \leftarrow \text{ADD-VC}()$   
9:    $c/a \leftarrow \text{rpc-call COMMIT}(T.writes, T.vc)$  at  $\mathcal{M}$ 
```

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

$\mathcal{M}.ts$: for start-timestamps and commit-timestamps

$\{x.ver = (x.ts, x.ord, x.val)\}$: set of versions of x

```
1: procedure START()
2:   return ++ $\mathcal{M}.ts$ 
3: procedure READ( $x$ )
4:   return the latest  $x.ver$  installed
5: procedure COMMIT( $T.writes, T.vc$ )
6:   if CHECK-VC( $T.vc$ ) && write-conflict freedom then
7:      $T.cts \leftarrow ++\mathcal{M}.ts$ 
8:     ▷ apply  $T.writes$  locally and propagate it
9:      $T.upvers = \emptyset$                                 ▷ collect updated versions
10:    for  $(x, v) \in T.writes$  do
11:       $x.new-ver \leftarrow (T.cts, ++x.ord, v)$ 
12:      add  $x.new-ver$  to  $\{x.ver\}$  and  $T.upvers$ 
13:    broadcast  $\langle PROP, T.upvers \rangle$  to slaves
14:    return  $c$  denoting “committed”
15:  return  $a$  denoting “aborted”
```

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

$x.ver = (x.ts, x.ord, x.val)$: the latest version of x

```
1: procedure READ( $x$ )  
2:   return  $x.ver$   
3: upon RECEIVED( $\langle \text{PROP}, T.upvers \rangle$ )  
4:   for  $(x.ver' = (x.ts', x.ord', x.val')) \in T.upvers$  do  
5:     if  $x.ord' > x.ord$  then  
6:        $x.ver \leftarrow x.ver'$ 
```

Algorithm 2 RVSI-MP for Executing Transaction T (Client).

```
1: procedure BEGIN()  
2:   return rpc-call GETTS() at  $\mathcal{T}$   
3: procedure END()  
4:    $T.vc \leftarrow \text{ADD-VC}()$   
5:    $c/a \leftarrow \text{rpc-call C-COMMIT}(T.writes, T.vc)$  at  $\mathcal{C}$ 
```

Algorithm 2 RVSI-MP for Executing Transaction T (Timestamp Oracle).

$\mathcal{T}.ts$: for start-timestamps and commit-timestamps

- 1: **procedure** GETTS()
 - 2: **return** ++ $\mathcal{T}.ts$
-

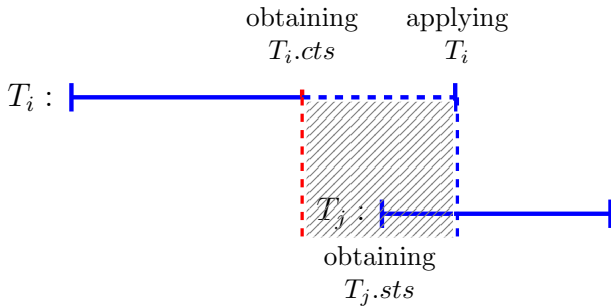
Algorithm 2 RVSI-MP for Executing Transaction T (Coordinator).

```
1: procedure C-COMMIT( $T.writes, T.vc$ )
2:   split  $T.writes$  and  $T.vc$  with the data partitioning strategy
3:    $\triangleright$  the prepare phase:
4:   rpc-call PREPARE( $T.writes, T.vc$ ) at each  $\mathcal{M}$ 
5:    $\triangleright$  the commit phase:
6:   if all PREPARE( $T.writes, T.vc$ ) return true then
7:      $T.cts \leftarrow$  rpc-call GETTS() at  $\mathcal{T}$ 
8:     rpc-call COMMIT( $T.cts, T.writes$ ) at each  $\mathcal{M}$ 
9:   else
10:    rpc-call ABORT() at each  $\mathcal{M}$ 
11:    return  $a$  denoting “aborted”
12:   if all COMMIT( $T.cts, T.writes$ ) return true then
13:     return  $c$  denoting “committed”
14:   else
15:     return  $a$  denoting “aborted”
```

Algorithm 2 RVSI-MP for Executing Transaction T (Master).

```
1: procedure PREPARE( $T.writes, T.vc$ )
2:   return CHECK-VC( $T.vc$ ) && write-conflict freedom
3: procedure COMMIT( $T.cts, T.writes$ )
4:   ▷ apply  $T.writes$  locally and propagate it
5: procedure ABORT()
6:   ▷ abort
```

Atomicity of the commit-timestamps:




Delays

(One-way) delays among nodes ¹:

Within datacenter: 1 ~ 2ms

Across datacenters: 15 ~ 25ms

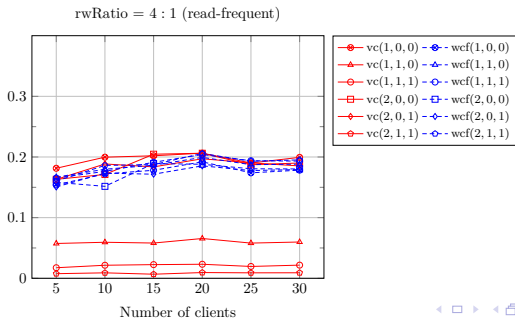
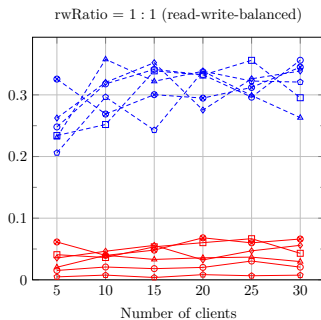
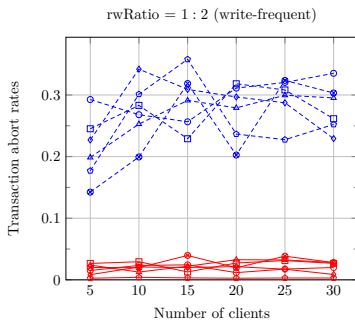
Clients to nodes: 15 ~ 20ms

¹<https://github.com/hengxin/aliyun-ping-traces> 

Benchmarks

- ▶ The TPC-C benchmark is commonly used to benchmark relational databases.
- ▶ The YCSB benchmark [Cooper@SoCC'10] for distributed key-value stores does not support transactions.

We design our own workloads.



issueDelay = 20ms: $bv(1, 0, 0) = 0.0057$ $fv(1, 0, 0) = 0.0251$

issueDelay = 15ms: $bv(1, 0, 0) = 0.08225$ $fv(1, 0, 0) = 0.0393$

issueDelay = 5ms: $bv(1, 0, 0) = 0.1716$ $fv(1, 0, 0) = 0.0045$

issueDelay = 20ms: $bv(1, 0, 0) = 0.0057$ $fv(1, 0, 0) = 0.0251$

issueDelay = 15ms: $bv(1, 0, 0) = 0.08225$ $fv(1, 0, 0) = 0.0393$

issueDelay = 5ms: $bv(1, 0, 0) = 0.1716$ $fv(1, 0, 0) = 0.0045$

larger issueDelay \implies longer transaction

more concurrent transactions

more likely to obtain data versions updated by concurrent transactions

more sensitive to k_2 -FV