近代的トランザクション処理技法

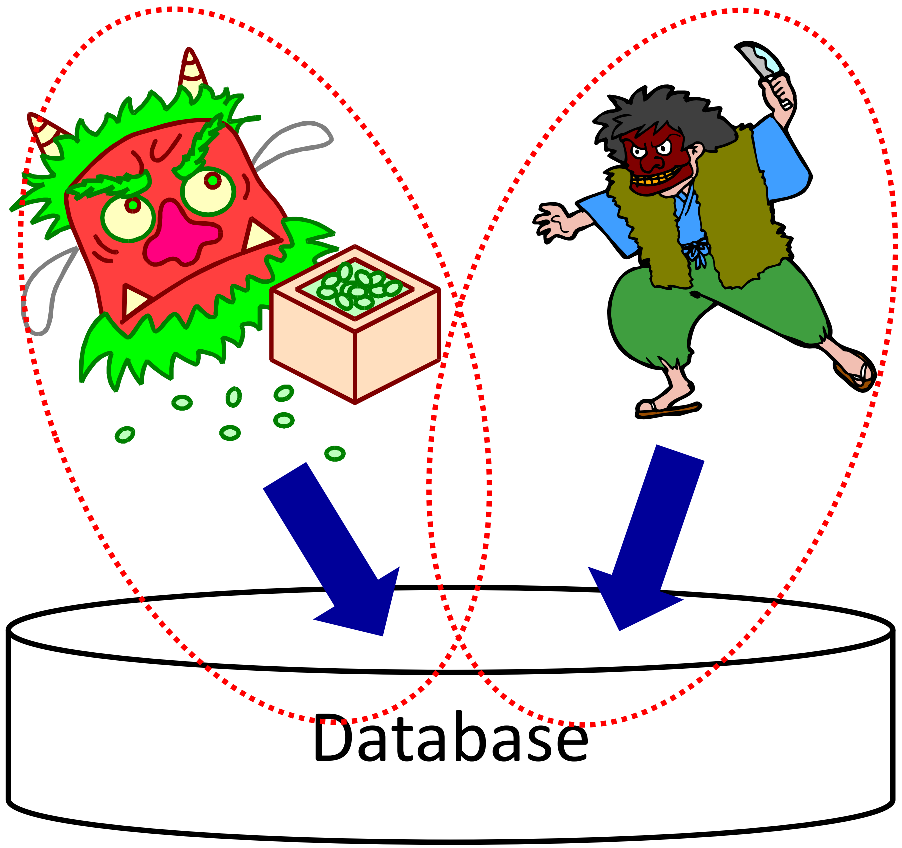
Feb. 27, 2022

Hideyuki Kawashima

Agenda

- Transaction
- Silo CC
- Concurrent index
- Silo recovery
- Modern CC and future directions

Transaction Processing System = Concurrency Control + Recovery



Applications: bank, credit card, distributed file system (spotify), blockchain…

Why transaction?
ATM: Your DB is 30,000 first.

Company

```
Read(DB, x);
x := x + 10,000;



Write(DB, x);
```

You

```
Read(DB, y);
y := y – 10,000;
Write(DB, y);
```

Final value:
40,000?

Lost Update

T2 write has gone…

$$r1[x] \ldots w2[x] \ldots w1[x] \ldots c1$$

T1	T2
Read(x)	Read (x)
	Write(x)
Write(x)	
Commit	

Isolation Levels & Anomalies

Table 4. Isolation Types Characterized by Possible Anomalies Allowed.								
Isolation level	**P0** Dirty Write	**P1** Dirty Read	**P4C** Cursor Lost Update	**P4** Lost Update	**P2** Fuzzy Read	**P3** Phantom	**A5A** Read Skew	**A5B** Write Skew
READ UNCOMMITTED == Degree 1	Not Possible	Possible	Possible	Possible	Possible	Possible	Possible	Possible
READ COMMITTED == Degree 2	Not Possible	Not Possible	Possible	Possible	Possible	Possible	Possible	Possible
Cursor Stability	Not Possible	Not Possible	Not Possible	Sometimes Possible	Sometimes Possible	Possible	Possible	Sometimes Possible
REPEATABLE READ	Not Possible	Not Possible	Not Possible	Not Possible	Not Possible	Possible	Not Possible	Not Possible
Snapshot	Not Possible	Not Possible	Not Possible	Not Possible	Not Possible	Sometimes Possible	Not Possible	Possible
ANSI SQL SERIALIZABLE == Degree 3 == Repeatable Read Date, IBM, Tandem, ...	Not Possible	Not Possible	Not Possible	Not Possible	Not Possible	Not Possible	Not Possible	Not Possible

Hal Berenson, Philip A. Bernstein, Jim Gray, Jim Melton, Elizabeth J. O'Neil, Patrick E. O'Neil: A Critique of ANSI SQL Isolation Levels. SIGMOD Conference 1995: 1-10

Serializability

--Equivalent to Serial Exec?--

T1
R(a)
W(a)
R(b)
W(b)

T2
R(a)
W(a)
R(b)
W(b)

T1
R(a)
W(a)
R(b)
W(b)

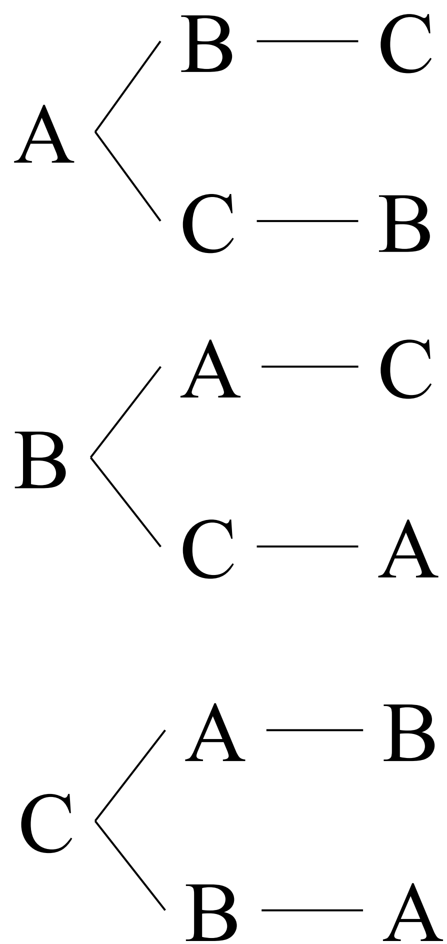
T2
R(a)
W(a)
R(b)
W(b)

T2
R(a)
W(a)
R(b)
W(b)

T1
R(a)
W(a)
R(b)
W(b)

Quiz
How many serial orders?

- Number of transactions
  - 2
  - 3
  - 4
  - 5
  - N



Serializable schedules

T1
R(a)
W(a)
R(b)
W(b)
Commit

T2
R(a)
W(a)
R(b)
W(b)
Commit

T1	T2
R(a)	
W(a)	
R(b)	R(a)
W(b)	W(a)
Commit	R(b)
	W(b)
	Commit

T1	T2
	R(a)
	W(a)
R(a)	
	R(b)
W(a)	W(b)
R(b)	Commit
W(b)	
Commit	

T1 operations →T2 operations

T2 operations →T1 operations

WR Conflicts
(Reading Uncommitted Data)

T1
R(a)
a = a – 100;
W(a)
R(b)
b = b + 100;
W(b)
Commit

T2
R(a)
a = a * 1.06;
W(a)
R(b)
b = b * 1.06;
W(b)
Commit

T1	T2
R(a)	
W(a)	
	R(a)
	W(a)
	R(b)
	W(b)
	Commit
R(b)	
W(b)	
Commit	

T1 should access b first.

T1: Transfter $100 from A to B
T2: Increment A & B by 6 %

Ideal:
A: 200->100->106
B: 200->300->318

Real:
A: 200->100->106
B: 200->212->312

B loss

Serializable

(Conflict Serializable (Final State, View))

$$r_1(A); w_1(A); r_2(A); w_2(A); r_1(B); w_1(B); r_2(B); w_2(B);$$

Transaction ID（1） Data ID（A）

No!

$2\ ops\ in\ a$ TX: $r_i(X); w_i(Y);$

WW to the same data item: $w_i(X); w_j(X);$

RW to the same data item: $r_i(X); w_j(X);$

WR to the same data item: $w_j(X); r_i(X);$

$$r_1(A); w_1(A); r_2(A); w_2(A); r_1(B); w_1(B); r_2(B); w_2(B);$$

$$r_1(A); w_1(A); r_2(A); r_1(B); w_2(A); w_1(B); r_2(B); w_2(B);$$

$$r_1(A); w_1(A); r_1(B); r_2(A); w_2(A); w_1(B); r_2(B); w_2(B);$$

$$r_1(A); w_1(A); r_1(B); r_2(A); w_1(B); w_2(A); r_2(B); w_2(B);$$

$$r_1(A); w_1(A); r_1(B); w_1(B); r_2(A); w_2(A); r_2(B); w_2(B);$$

Quiz: Serializable?

- Quiz-1

$$r_1(x); r_2(x); r_1(z); w_1(x); w_2(y); r_3(z); w_3(y); w_3(z);$$

- Quiz-2

$$r_1(x); r_3(w); r_2(y); w_1(y); w_1(x); w_2(x); w_2(z); w_3(x);$$

- Quiz-3

$$r_1(x); r_2(x); w_2(y); w_1(x);$$

1, 2: Serializable

Concurrency Control Protocols

- Pessimistic
  - Conflict…(;_;)
  - Acquire lock→DB access→Release lock

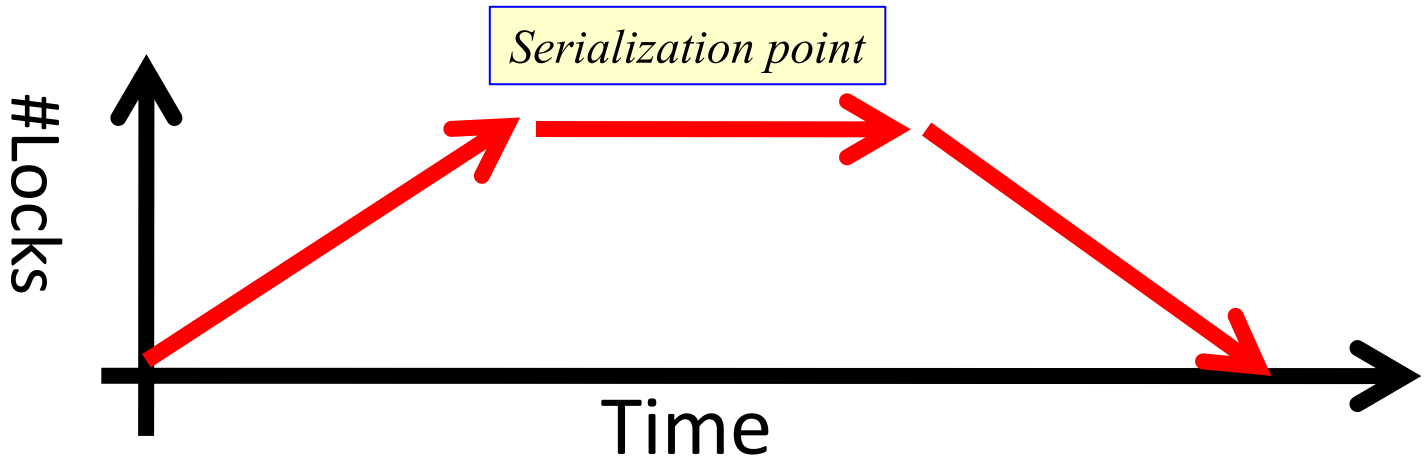
- Optimistic
  - No conflict　(@^^)/~~~
  - Read data→Validate→Acquire lock→DB access→Release lock

- Multi-version
  - Using multiple versions.

Strict 2 Phase Locking (S2PL)

- 2 rules
  - Acquire locks before accessing DB
    - Read-read is not blocked (rigt table)
  - Release all locks at the end
- Growing phase & shrinking phase

	rl	wl
rl	Y	N
wl	N	N



Serializable schedules

T1
R(a)
W(a)
R(b)
W(b)
Commit

T2
R(a)
W(a)
R(b)
W(b)
Commit

T1	T2
R(a)	
W(a)	
R(b)	R(a)
W(b)	W(a)
Commit	R(b)
	W(b)
	Commit

Serialization point

T1	T2
	R(a)
	W(a)
R(a)	
	R(b)
W(a)	W(b)
R(b)	Commit
W(b)	
Commit	

Serialization point

T1 operations → T2 operations

T2 operations → T1 operations

Concurrency Control Protocols

- Pessimistic
  - Conflict…(;_:)
  - Acquire lock→DB access→Release lock

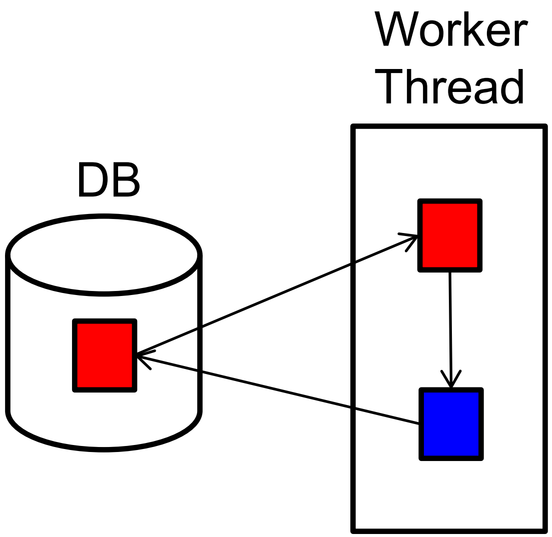
- Optimistic
  - No conflict　(@^^)/~~~
  - Read data→Validate→Acquire lock→DB access→Release lock

- Multi-version
  - Using multiple versions.

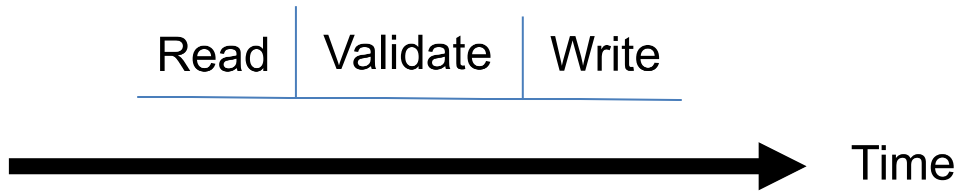
Kung-Robinson Model

3 phases: read, validate, write

- READ **(no read lock!)**
  - Copy from DB to thread
  - Update local data
- VALIDATE
  - Check conflicts
- WRITE
  - If no conflicts, write back to DB



H. T. Kung, John T. Robinson: On Optimistic Methods for Concurrency Control. ACM Trans. Database Syst. 6(2): 213-226 (1981)

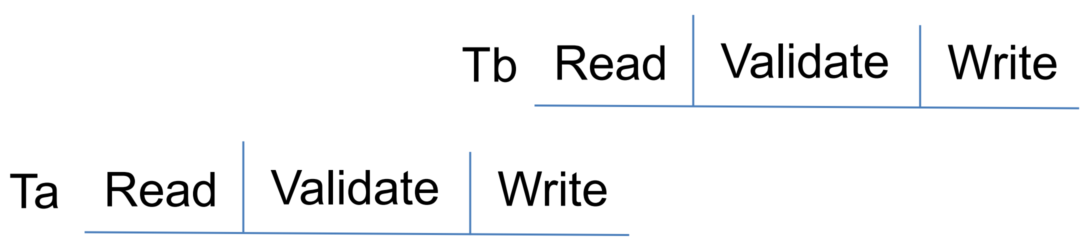


Validation Case 1

Ta completes its write phase before Tb starts its read phase

Validation Case 2

The write set of Ta does not intersect the read set of Tb, and Ta completes its write phase before Tb starts its write phase.



$TX_{start} \leftarrow TX_{global}$

READ

Acquire GiantLock

$TX_{end} \leftarrow TX_{global}$

valid $\leftarrow$ true

For t from $(TX_{start} + 1)$ to (TX_{end})

    if (writeset of t intersects my readset)

        valid $\leftarrow$ false

if (valid = true)

    writephase

    $TX_{global} \leftarrow TX_{global} + 1$

    $TX_{mine} \leftarrow TX_{global}$

Release GiantLock

VALIDATE
WRITE

$TX_{start} \leftarrow TX_{global}$ Who is finished when I start?

READ

Acquire GiantLock My turn!

$TX_{end} \leftarrow TX_{global}$ This is the end of checking scope

valid $\leftarrow$ true I wish I win…..!

For t from $(TX_{start} + 1)$ to (TX_{end}) Scope, case 1 is out of scope

  if (writeset of t intersects my readset) Check the case 2

    valid $\leftarrow$ false Sigh…

if (valid = true) Yeah!

  writephase Time to say "write"

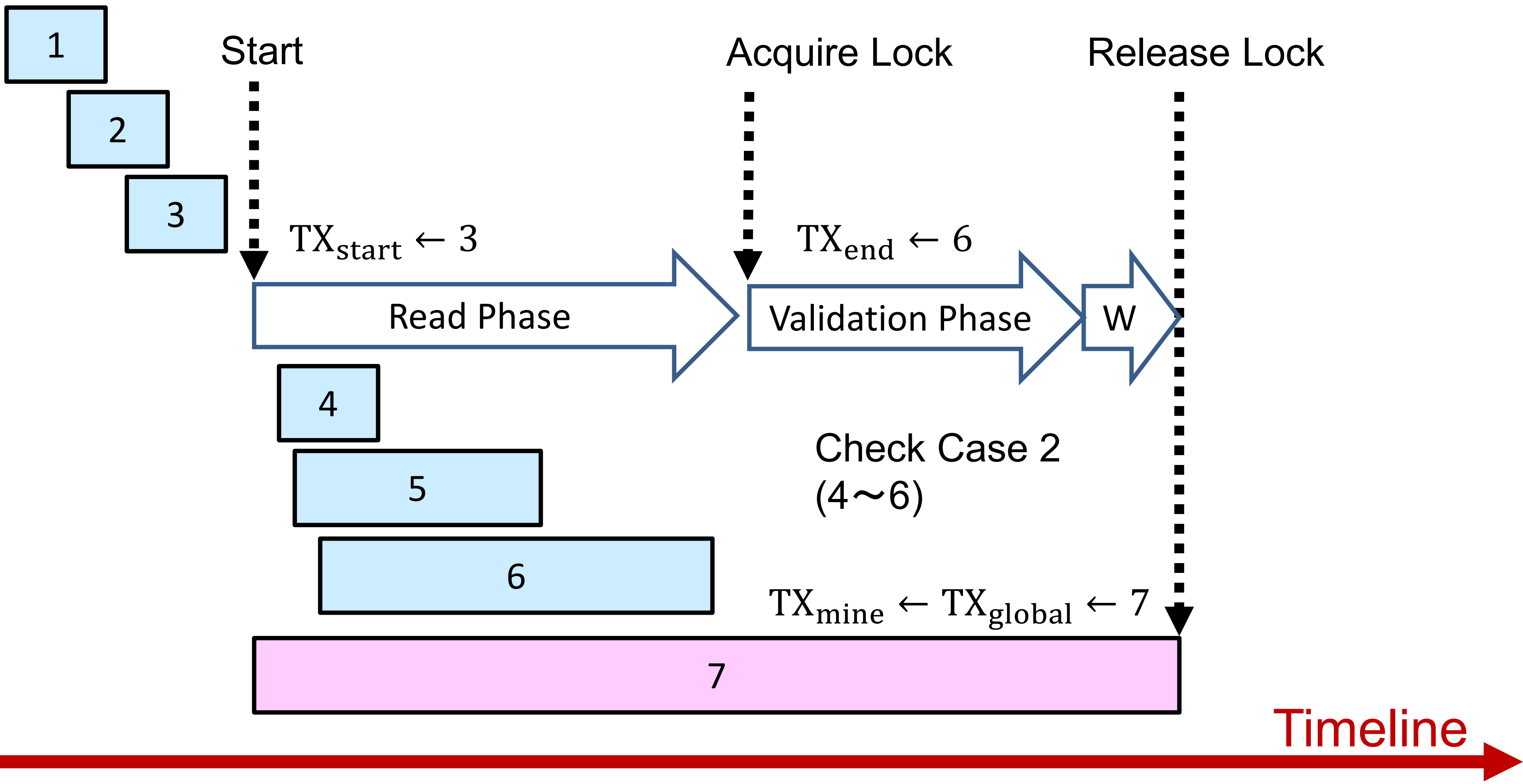
  $TX_{global} \leftarrow TX_{global} + 1$ Increment succeeded TX, which is me!

  $TX_{mine} \leftarrow TX_{global}$ This is my ID, will be checked later by another TX

Release GiantLock Bye!

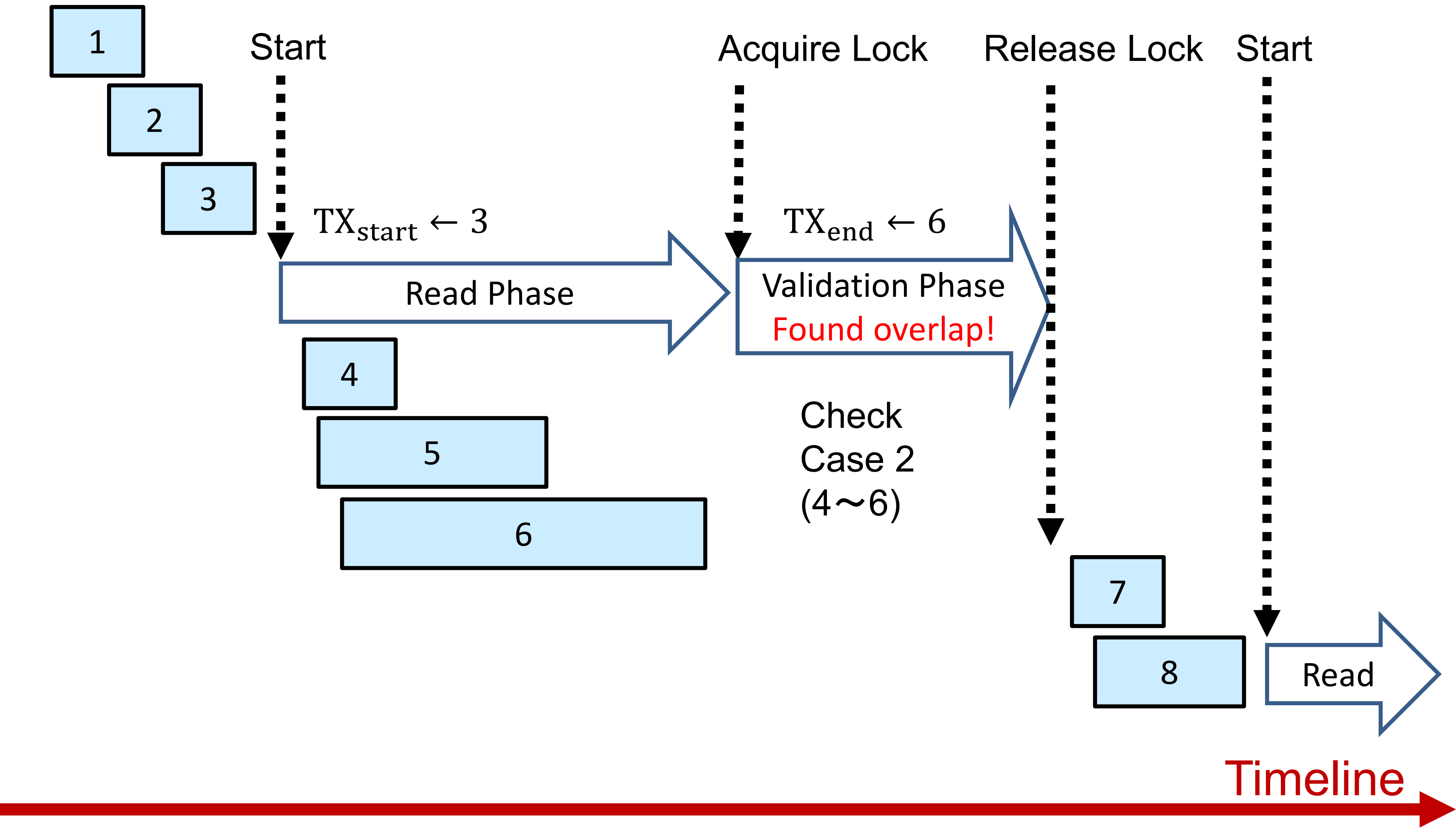
VALIDATE WRITE

An Example (1/2)



1. Only 1 TX can enter Validation phase (Critical Section)
2. Read phases can be conducted by multiple TXs.
3. “Serial Validation” (Section 4) in K&R paper is a reference

An Example (2/2)



4. Validation fails, then retry. This overhead is the problem.

Agenda

- Transaction
- Silo CC
- Concurrent index
- Silo recovery
- Modern CC and future directions

Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, Samuel Madden: Speedy transactions in multicore in-memory databases. SOSP 2013: 18-32

```
Data: read set R, write set W, node set N,
      global epoch number E

// Phase 1
for record, new-value in sorted(W) do
    lock(record);
compiler-fence();
e ← E;                          // serialization point
compiler-fence();

// Phase 2
for record, read-tid in R do
    if record.tid ≠ read-tid or not record.latest
            or (record.locked and record ∉ W)
    then abort();
for node, version in N do
    if node.version ≠ version then abort();
commit-tid ← generate-tid(R, W, e);

// Phase 3
for record, new-value in W do
    write(record, new-value, commit-tid);
    unlock(record);
```

Lock

Validate

Write

Production Example:
LineairDB: https://lineairdb.github.io/LineairDB/html/index.html

Silo Concurrency Control Protocol: 3 phases

```
Data: read set R, write set W
// Phase 0 (read and write)
R ← value and pointer of read(); W ← pointer of write();
// Phase 1 (lock)
for record, new-value in sorted(W)
    lock(record);
// Phase 2 (validate)
for record, read-tid in R
    if record.tid ≠ read-tid or record.locked
        abort();
commit-tid ← generate-tid(R, W);
// Phase 3 (write)
for record, new-value in W
    write(record, new-value, commit-tid);
    unlock(record);
```

Silo Concurrency Control Protocol

Data: $\boxed{\text{read set } R\text{, write set } W}$

// Phase 0 (read and write)

$R \leftarrow$ value and pointer of read(); $W \leftarrow$ pointer of write();

// Phase 1 (lock)

On "commit", it starts

for *record*, *new-value* **in** sorted(W)

    lock(*record*);

Lock records (DB). Sorting is for deadlock avoidance.

// Phase 2 (validate)

for *record*, *read-tid* **in** R

Record-TID and RS-TID are different? abort.

    if *record.tid* $\neq$ *read-tid* **or** *record.locked*

        abort();

Record is locked, abort.

commit-tid $\leftarrow$ generate-tid(R, W);

Generage new TID.

// Phase 3 (write)

for *record*, *new-value* **in** W

    write(*record*, *new-value*, *commit-tid*);

    unlock(*record*);

Update TID, write data, unlock

Silo commit protocol

$$r_1(A);\ r_2(B);\ w_1(B);\ w_2(A);\ c_1;\ c_2;$$

Quiz: serializable?

T1: readset

T1: writeset

T2: readset

T2: writeset

Header	Key	Value
	A	10
	B	20

Silo commit protocol

$$r_1(A);\ r_2(B);\ w_1(B);\ w_2(A);\ c_1;\ c_2;$$

T1: readset

A:10:ptr

T1: writeset

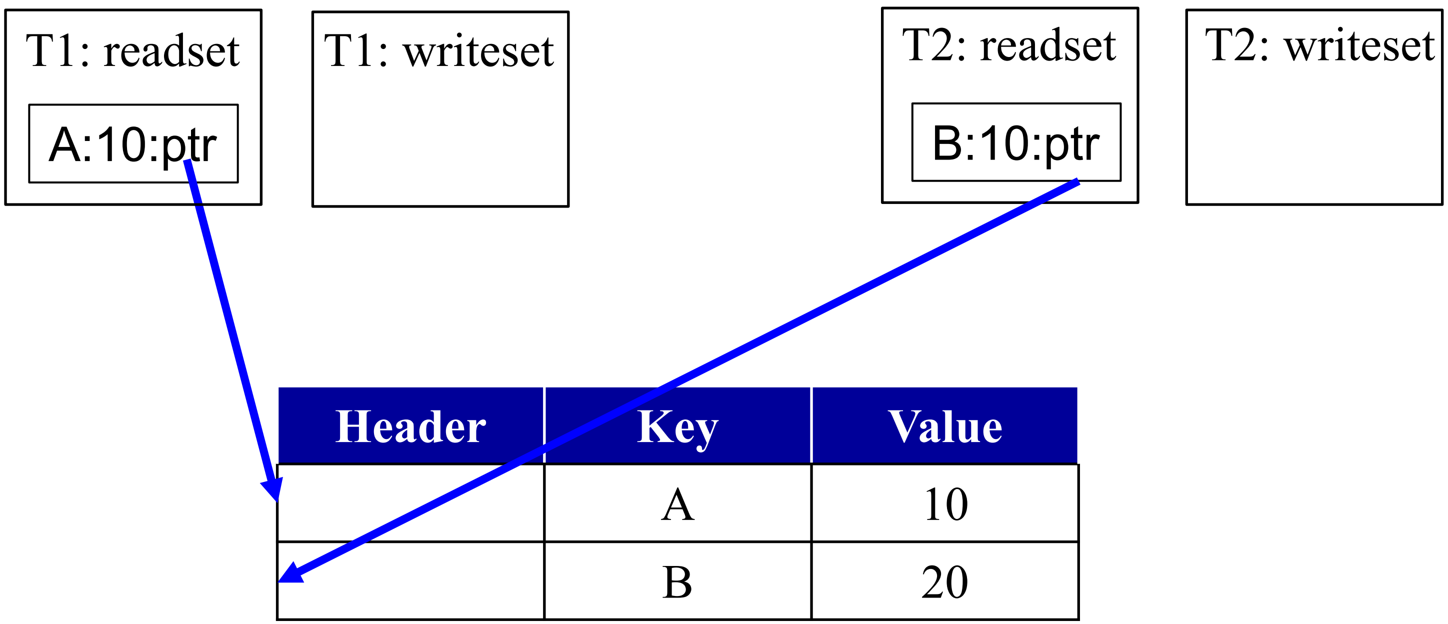
T2: readset

T2: writeset

Header	Key	Value
	A	10
	B	20

Silo commit protocol

$$r_1(A);\ r_2(B);\ w_1(B);\ w_2(A);\ c_1;\ c_2;$$



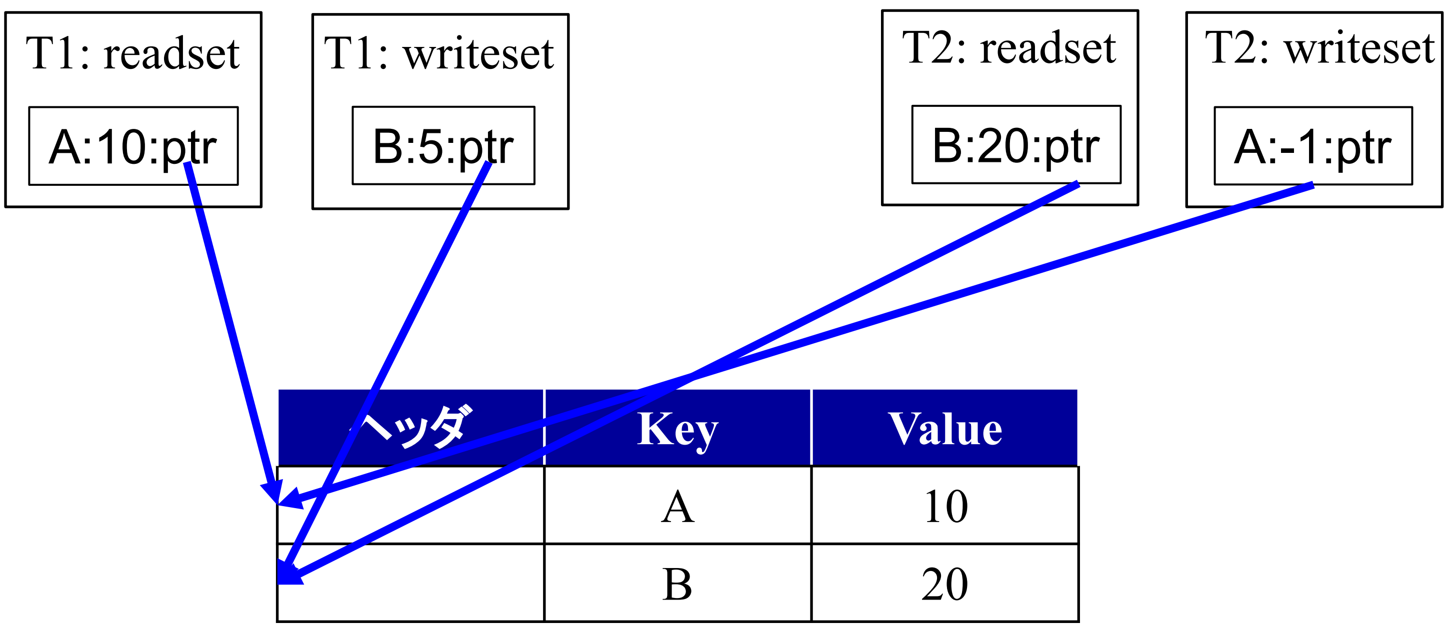
Silo commit protocol

$$r_1(A);\ r_2(B);\ w_1(B);\ w_2(A);\ c_1;\ c_2;$$

Header	Key	Value
	A	10
	B	20

Silo commit protocol

$$r_1(A);\ r_2(B);\ w_1(B);\ w_2(A);\ c_1;\ c_2;$$

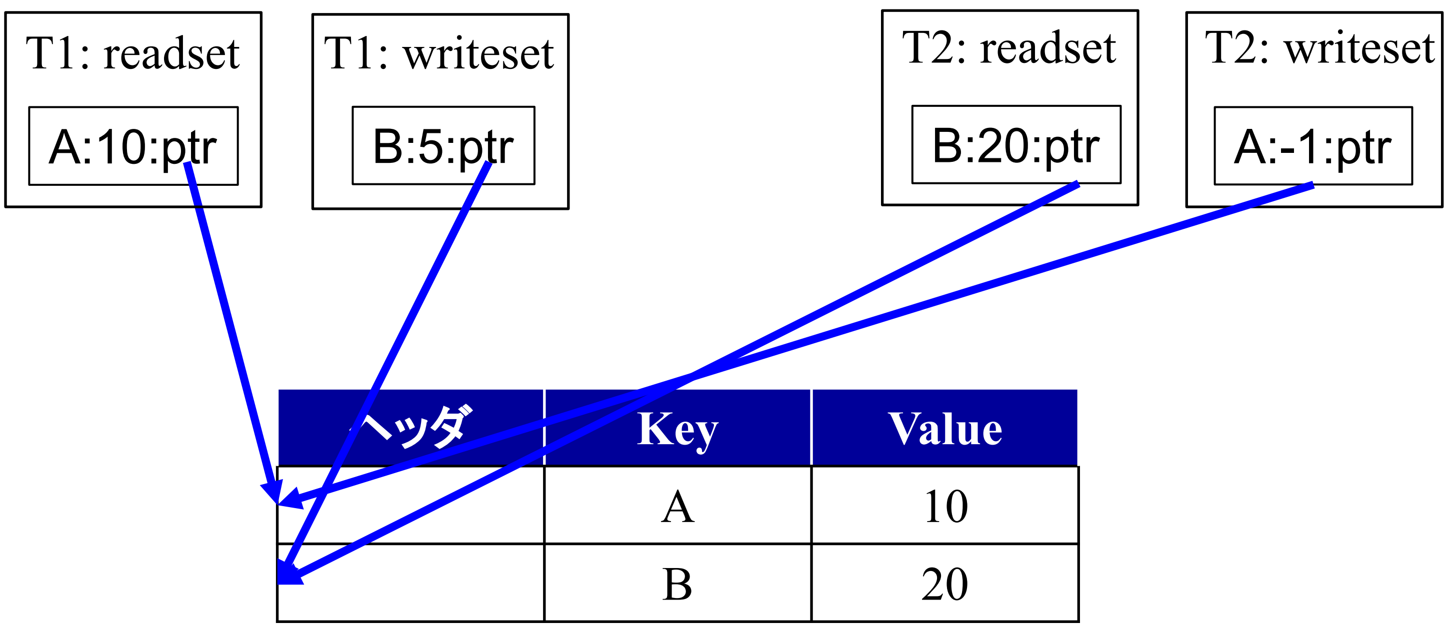


ヘッダ	Key	Value
	A	10
	B	20

Silo commit protocol

$$r_1(A);\ r_2(B);\ w_1(B);\ w_2(A);\boxed{c_1;\ c_2;}$$

May run concurrently



Silo Concurrency Control Protocol

3 Phases

Data: read set R, write set W

// Phase 0 (read and write)

$R \leftarrow$ value and pointer of read(); $W \leftarrow$ pointer of write();

// Phase 1 (lock)

On “commit”, it starts

for *record, new-value* **in** sorted(W)

    lock(*record*);

Lock records (DB). Sorting is for deadlock avoidance.

// Phase 2 (validate)

for *record, read-tid* **in** R

Record-TID and RS-TID are different? abort.

    if *record.tid* $\neq$ *read-tid* **or** *record.locked*

        abort();

Record is locked, abort.

commit-tid $\leftarrow$ generate-tid(R, W);

Generage new TID.

// Phase 3 (write)

for *record, new-value* **in** W

    write(*record, new-value, commit-tid*);

    unlock(*record*);

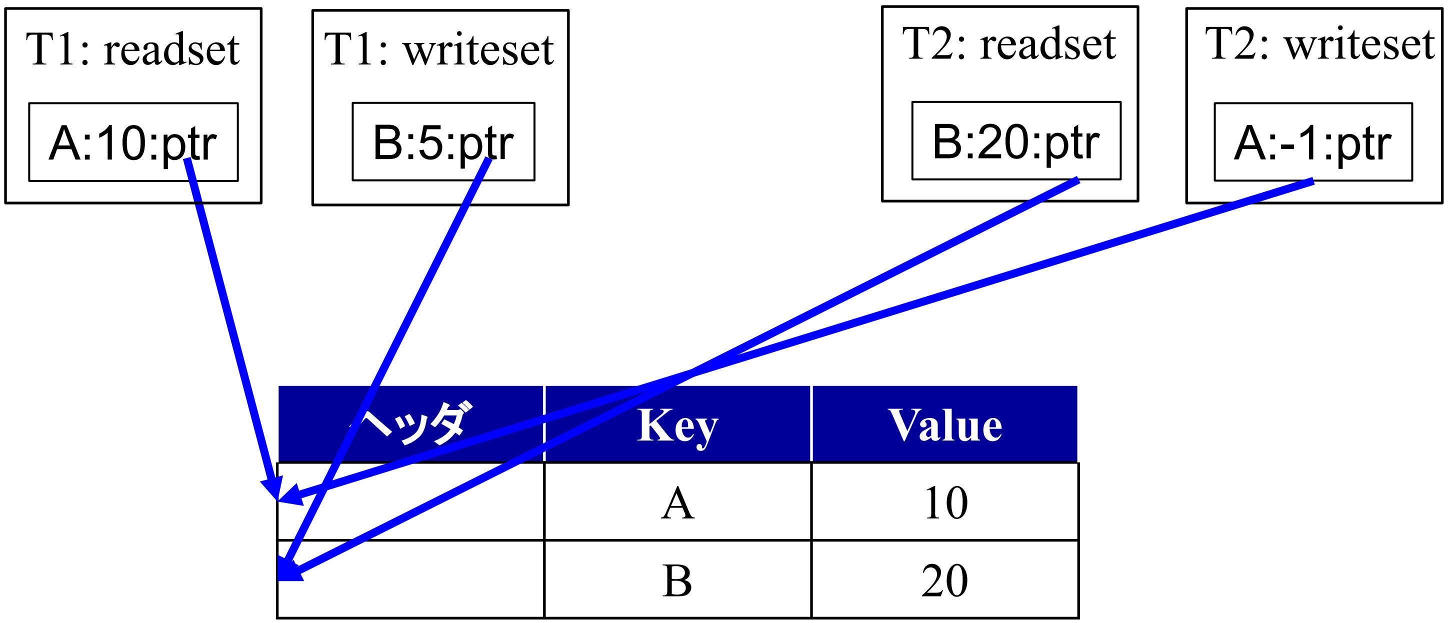
Update TID, write data, unlock

Silo commit protocolの例

$$r_1(A);\ r_2(B);\ w_1(B);\ w_2(A);\boxed{c_1;\ c_2;}$$

B != B'

$$c_1;\ c_2;\ L_1(B);\ V_1\{A\};\ W_1(B');\ U_1(B');\ L_2(A);\boxed{V_2\{B\};}$$



Silo commit protocolの例

$$r_1(A);\ r_2(B); w_1(B); w_2(A); c_1; c_2;$$

T1 ok $\quad B != B'$

$c_1; c_2;\ L_1(B); V_1\{A\}; W_1(B'); U_1(B');\ L_2(A); \boxed{V_2\{B\};}$

T1 ok

$c_1; c_2;\ L_1(B); V_1\{A\}; W_1(B');\ L_2(A); U_1(B'); \boxed{V_2\{B\};}$

T1 ok

$c_1; c_2;\ L_1(B); V_1\{A\}; L_2(A); W_1(B'); U_1(B'); \boxed{V_2\{B\};}$

$A\ is\ locked$

$c_1; c_2;\ L_1(B); V_1\{A\}; L_2(A); \boxed{V_2\{B\};}$ ……T1 ok

$B\ is\ locked$

$c_1; c_2;\ L_1(B); L_2(A); \boxed{V_1\{A\};}$ ……T1 ok

Silo Concurrency Control Protocol

Data: read set R, write set W

Why sort?

// Phase 0 (read and write)

$R \leftarrow$ value and pointer of read(); $W \leftarrow$ pointer of write();

// Phase 1 (lock)

On "commit", it starts

for *record, new-value* **in** sorted(W)

    lock(*record*);

Lock records (DB). Sorting is for deadlock avoidance.

// Phase 2 (validate)

for *record, read-tid* **in** R

Record-TID and RS-TID are different? abort.

    if *record.tid* $\neq$ *read-tid* **or** *record.locked*

        abort();

Record is locked, abort.

commit-tid $\leftarrow$ generate-tid(R, W);

Generage new TID.

// Phase 3 (write)

for *record, new-value* **in** W

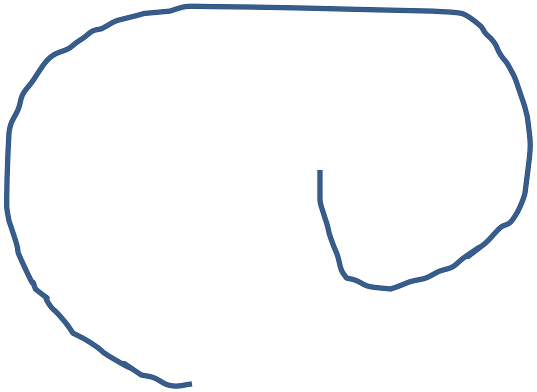
    write(*record, new-value, commit-tid*);

    unlock(*record*);

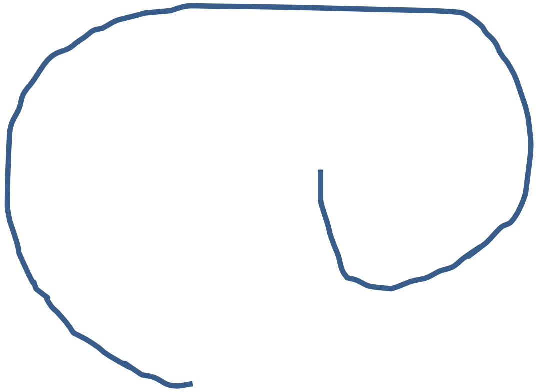
Update TID, write data, unlock

Deadlock

alice



bob



x, y

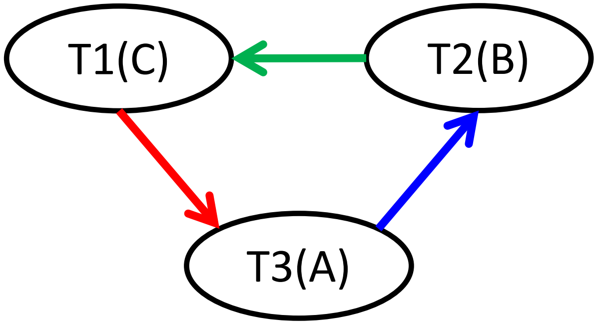
y, x

$$wl_a(x); wl_b(y); wl_a(y); wl_b(x);$$

Deadlock Detection

- Wait graph
  - Create a node N(Ti) for Ti
  - When Ti waits for Tj, N(Ti)→N(Tj)
  - If cycled, deadlock…

	1	2	3	4	5	6
T1	X(C)			X(A)		
T2		X(B)			X(C)	
T3			X(A)			X(B)



Deadlock avoidance

- Conservative S2PL
  - If you fail to acquire a lock, abort & retry
- Sort access order before accessing DB [MOCC]
  - $A \rightarrow B \rightarrow C$

[MOCC] Tianzheng Wang, Hideaki Kimura: Mostly-Optimistic Concurrency Control for Highly Contended Dynamic Workloads on a Thousand Cores. PVLDB 10(2): 49-60 (2016)

Silo Concurrency Control Protocol

Data: read set R, write set W

Why?

// Phase 0 (read and write)

$R \leftarrow$ value and pointer of read(); $W \leftarrow$ pointer of write();

// Phase 1 (lock)

On "commit", it starts

for *record, new-value* **in** sorted(W)

lock(*record*);

Lock records (DB). Sorting is for deadlock avoidance.

// Phase 2 (validate)

for *record, read-tid* **in** R

Record-TID and RS-TID are different? abort.

if *record.tid* $\neq$ *read-tid* **or** *record.locked*

abort();

Record is locked, abort.

commit-tid $\leftarrow$ generate-tid(R, W);

Generage new TID.

// Phase 3 (write)

for *record, new-value* **in** W

write(*record, new-value, commit-tid*);

unlock(*record*);

Update TID, write data, unlock

Silo commit protocol

$$r_1(A);\ r_2(B);\ w_1(B);\ w_2(A);\ c_1;\ c_2;$$

T1 ok $\quad B \,!\!= B'$

$$c_1; c_2;\ L_1(B); V_1\{A\}; W_1(B'); U_1(B');\ L_2(A); \boxed{V_2\{B\};}$$

T1 ok

$$c_1; c_2;\ L_1(B); V_1\{A\}; W_1(B');\ L_2(A); U_1(B'); \boxed{V_2\{B\};}$$

T1 ok

$$c_1; c_2;\ L_1(B); V_1\{A\}; L_2(A); W_1(B'); U_1(B'); \boxed{V_2\{B\};}$$

B is locked

$$c_1; c_2;\ L_1(B); V_1\{A\}; L_2(A); \boxed{V_2\{B\};}$$ ……T1 ok

A is locked

$$c_1; c_2;\ L_1(B); L_2(A); \boxed{V_1\{A\};}$$ ……T1 ok

Conflict (RW, WR, WW)

$r_1(A); w_1(A); r_2(A); w_2(A); r_1(B); w_1(B); r_2(B); w_2(B);$

Transaction ID（1） Data ID（A）

No Switch

$2\ ops\ in\ a$ TX: $r_i(X); w_i(Y);$

WW to the same data item: $w_i(X); w_j(X);$

RW to the same data item: $r_i(X); w_j(X);$

WR to the same data item: $w_j(X); r_i(X);$

$r_1(A); w_1(A); r_2(A); w_2(A); r_1(B); w_1(B); r_2(B); w_2(B);$

$r_1(A); w_1(A); r_2(A); r_1(B); w_2(A); w_1(B); r_2(B); w_2(B);$

$r_1(A); w_1(A); r_1(B); r_2(A); w_2(A); w_1(B); r_2(B); w_2(B);$

$r_1(A); w_1(A); r_1(B); r_2(A); w_1(B); w_2(A); r_2(B); w_2(B);$

$r_1(A); w_1(A); r_1(B); w_1(B); r_2(A); w_2(A); r_2(B); w_2(B);$

Silo commit protocol

$$r_1(A);\ r_2(B);\ w_1(B);\ w_2(A);\ c_1;\ c_2;$$

T1: readset	T1: writeset	T2: readset	T2: writeset
A:10:80	B:5:90	B:20:90	A:-1:80

TID	Lock	Key	Value
80	1	A	10
91	0	B	5

$$c_1;\ c_2;\ L_1(B); V_1\{A\}; W_1(B'); U_1(B');\ L_2(A)\boxed{; V_2\{B\}}$$

T2 : I read B. I found B is concurrently updated by T1.
T2 : Oh, I found I was conflicted with T1 on B…
T2 : T1 updated B before my access. Then, the schedule should be **T1->T2.**
Thus, I need to read the result of T1.
T2 : But I read result of T0…This is inconsistent. Abort sayonara.

Transaction requires serializablity. T1->T2 or T2 -> T1

Silo Concurrency Control Protocol

Data: read set R, write set W

Why?

// Phase 0 (read and write)

$R \leftarrow$ value and pointer of read(); $W \leftarrow$ pointer of write();

// Phase 1 (lock)

On "commit", it starts

for *record, new-value* **in** sorted(W)

lock(*record*);

Lock records (DB). Sorting is for deadlock avoidance.

// Phase 2 (validate)

for *record, read-tid* **in** R

Record-TID and RS-TID are different? abort.

if *record.tid* $\neq$ *read-tid* **or** *record.locked*

abort();

Record is locked, abort.

commit-tid $\leftarrow$ generate-tid(R, W);

Generage new TID.

// Phase 3 (write)

for *record, new-value* **in** W

write(*record, new-value, commit-tid*);

unlock(*record*);

Update TID, write data, unlock

Silo commit protocol

$$r_1(A);\ r_2(B);\ w_1(B);\ w_2(A);\ c_1;\ c_2;$$

T1 ok $\quad B != B'$

$$c_1;\ c_2;\ L_1(B);\ V_1\{A\};\ W_1(B');\ U_1(B');\ L_2(A);\ \boxed{V_2\{B\};}$$

T1 ok

$$c_1;\ c_2;\ L_1(B);\ V_1\{A\};\ W_1(B');\ L_2(A);\ U_1(B');\ \boxed{V_2\{B\};}$$

T1 ok

$$c_1;\ c_2;\ L_1(B);\ V_1\{A\};\ L_2(A);\ W_1(B');\ U_1(B');\ \boxed{V_2\{B\};}$$

$B\ is\ locked$

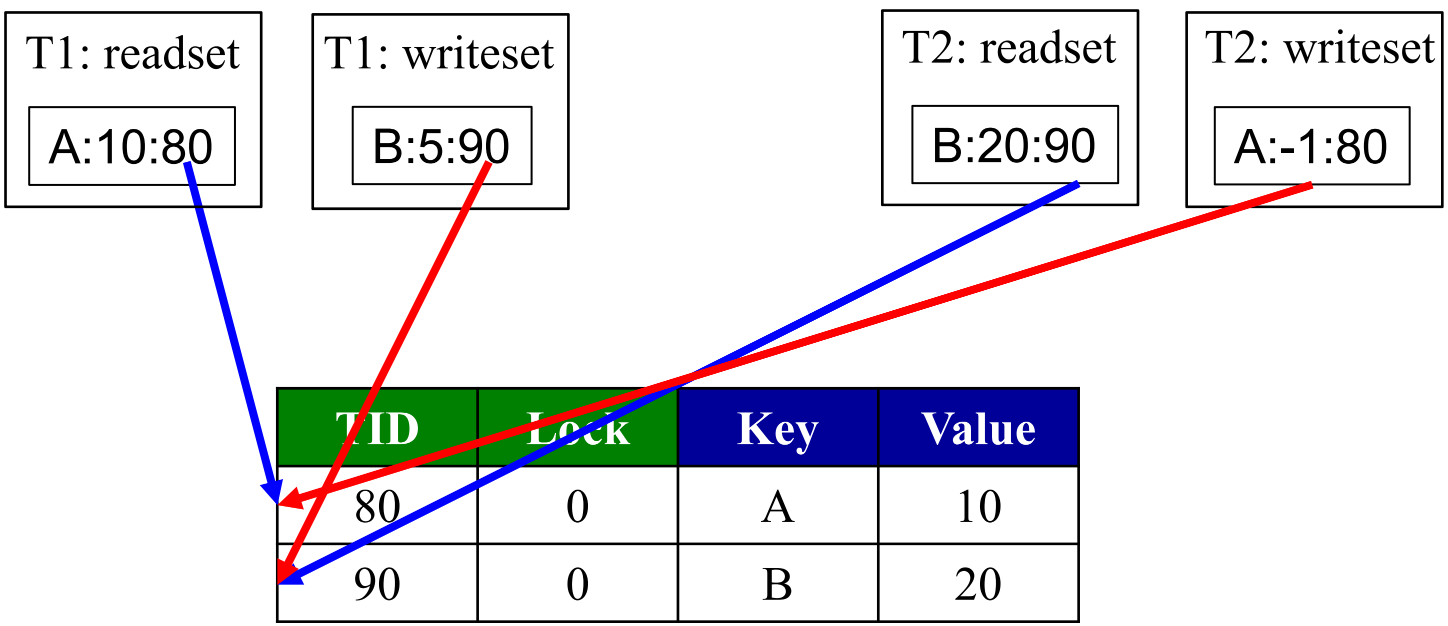
$$\underline{c_1;\ c_2;\ L_1(B);\ V_1\{A\};\ L_2(A);\ \boxed{V_2\{B\};}}$$ ……T1 ok

$A\ is\ locked$

$$c_1;\ c_2;\ L_1(B);\ L_2(A);\ \boxed{V_1\{A\};}$$ ……T1 ok

Silo commit protocol

$$r_1(A);\ r_2(B);\ w_1(B);\ w_2(A);\ c_1;\ c_2;$$



TID	Lock	Key	Value
80	0	A	10
90	0	B	20

Silo commit protocol

$$r_1(A);\ r_2(B);\ w_1(B);\ w_2(A);\ c_1;\ c_2;$$

T1: readset	T1: writeset	T2: readset	T2: writeset
A:10:80	B:5:90	B:20:90	A:-1:80

TID	Lock	Key	Value
80	0	A	10
90	1	B	20

	rl	wl
rl	Y	N
wl	N	N

$$c_1;\ c_2;\ L_1(B);\ V_1\{A\};\ L_2(A);\ \boxed{V_2\{B\};}$$

T2: I read B, B is locked by another one.
T2: As you know, if an item is write-locked, then we cannot obtain read lock…
T2: I believed that I succeeded to acquire read lock, but it was just my dream…
T2: Since I failed to acquire a read lock, I abort. Sayonara!

Silo Concurrency Control Protocol

Done

Data: read set R, write set W

// Phase 0 (read and write)

$R \leftarrow$ value and pointer of read(); $W \leftarrow$ pointer of write();

// Phase 1 (lock)

On "commit", it starts

for *record, new-value* **in** sorted(W)

    lock(*record*);

Lock records (DB). Sorting is for deadlock avoidance.

// Phase 2 (validate)

for *record, read-tid* **in** R

Record-TID and RS-TID are different? abort.

    if *record.tid* $\neq$ *read-tid* **or** *record.locked*

        abort();

Record is locked, abort.

commit-tid $\leftarrow$ generate-tid(R, W);

Generage new TID.

// Phase 3 (write)

for *record, new-value* **in** W

    write(*record, new-value, commit-tid*);

    unlock(*record*);

Update TID, write data, unlock

Quiz

- Can a CC prevent following?
  - S2PL
  - Silo

Non-Serializable Schedule

$$r_1(A);\ r_2(B);\ w_2(A);\ w_1(B);\ c_1 c_2;$$

Answer

$$r_1(A);\ r_2(B);\ w_2(A);\ w_1(B);\ c_1 c_2;$$

$$lr_1(A) lw_1(B) lr_2(B)\ r_1(A) w_1(B)\ ul_1(A) ul_1(B);\ c_1;$$
$$\boxed{lw_2(A)} r_2(B); w_2(A);\ ul_2(B) ul_2(A);\ c_2;$$

T2 is block by here. If conservative, T2 aborts.

$$r_1(A);\ r_2(B);\ w_2(A); w_1(B); c_1; c_2;$$
$$L_2(A); V_2(B); L_1(B); \boxed{V_1(A);}$$

T2 may start earlier.

T1 can start earlier.

T1 aborts since it is locked

Quiz

- Can Silo pass the following?

Serializable Schedule

$$r_1(A);\ w_2(A);\ w_1(B); c_2; c_1;$$

$$r_1(A);\ w_2(A); w_1(B); c_2;\ L_2(A); W_2(A); U_2(A);\ c_1;\ L_1(B);\ V_1(A);$$

$$r_1(A);\ w_2(A); w_1(B); c_2;\ L_2(A);\ c_1;\ L_1(B);\ V_1(A);$$

Serializable but…. False negative.

Why is Silo speedy?

1. Short blocking
2. Native latch
3. Invisible read

1. Shorter blocking time. Silo locks *after* commit

$$r_1(A);\ r_2(B);\ w_2(A);\ w_1(B);\ c_1 c_2;$$

$$lr_1(A)lw_1(B)lr_2(B)\ r_1(A)w_1(B)\ ul_1(A)ul_1(B);\ c_1;$$
$$\boxed{lw_2(A)}r_2(B); w_2(A);\ ul_2(B)ul_2(A);\ c_2;$$

T2 is blocked by here. If conservative, T2 aborts.

$$r_1(A);\ r_2(B);\ w_2(A); w_1(B); c_1; c_2;$$
$$L_2(A); V_2(B); L_1(B); \boxed{V_1(A);}$$

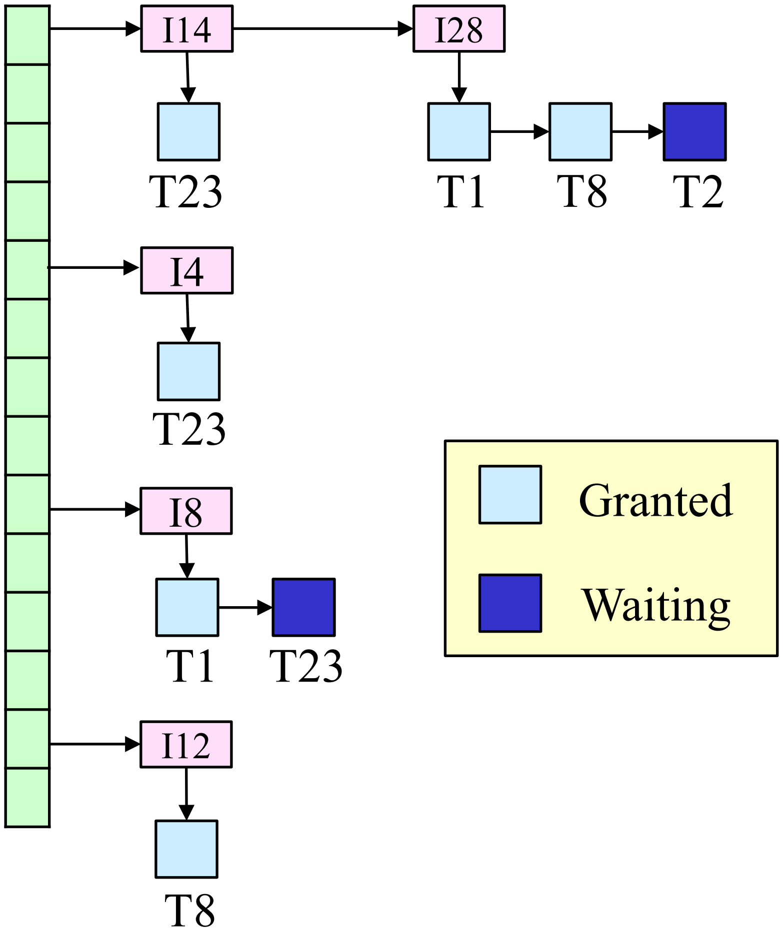
T2 may start earlier.

T1 can start earlier.

T1 aborts since it is locked

2. Native latch
Centralized lock manager is slow

- Disadvantage
  - Locking may be necessary for hash.
  - Locking is necessary for accessing an item.
  - Many pointer accesses.



Silo does have a lock manager.
Latches are decentralized.

Agenda

- Transaction
- Silo CC
- Concurrent index
- Silo recovery
- Modern CC and future directions

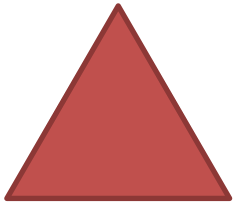
Data: read set R, write set W, node set N,
global epoch number E

// Phase 1
for *record*, *new-value* **in** sorted(W) **do**
  lock(*record*);
compiler-fence();
$e \leftarrow E$; *// serialization point*
compiler-fence();

If you insert a new data,
a leaf version is updated, then abort
Leaf node: bottom level node.

// Phase 2
for *record*, *read-tid* **in** R **do**
  if *record.tid* $\neq$ *read-tid* **or not** *record.latest*
    or (*record.locked* **and** *record* $\notin W$)
  then abort();
for *node*, *version* **in** N **do**
  if *node.version* $\neq$ *version* **then** abort();
commit-tid $\leftarrow$ generate-tid(R, W, e);

// Phase 3
for *record*, *new-value* **in** W **do**
  write(*record*, *new-value*, *commit-tid*);
  unlock(*record*);



What is this?
Phantom avoidance
using index

Phantom

T1: more results at 2nd read?

$$r1[P] \ldots w2[y\ in\ P] \ldots c2 \ldots r1[P] \ldots c1$$

P: Predicate

Ex: read <=10

T1	T2
Read -> x	
	Insert (z) Commit
Read -> x, z Commit	

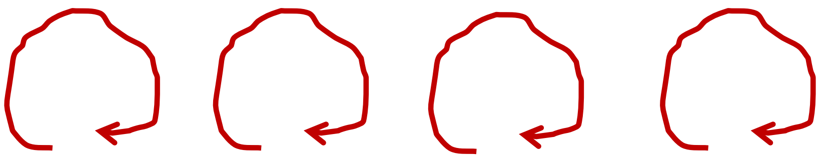
x (10), y (30)

x (10), y (30), z(3)

Concurrent tree

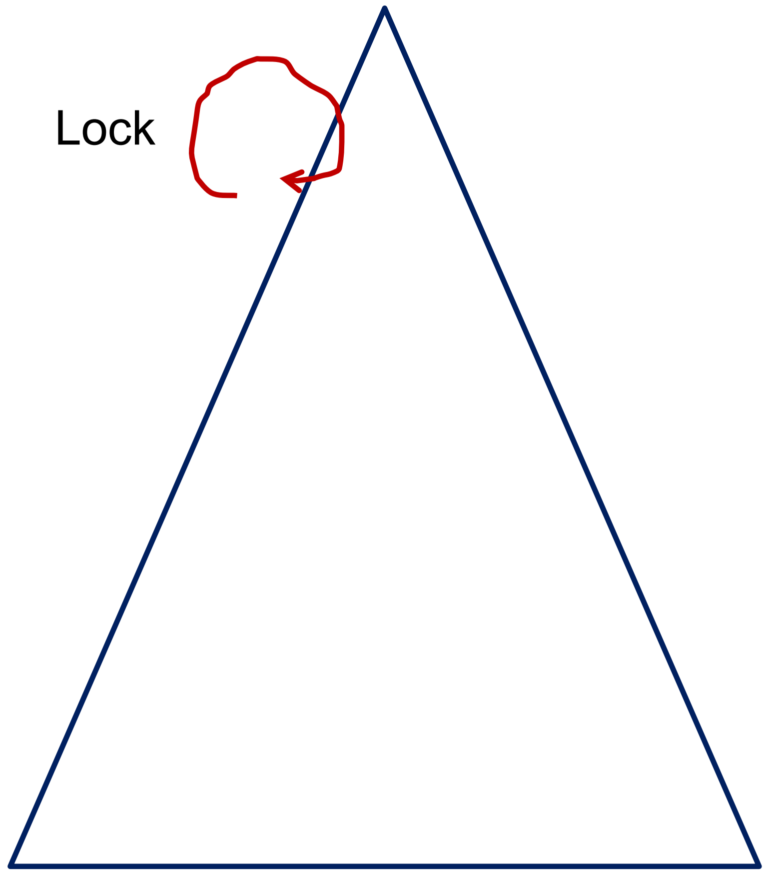
Tree-Locking

- B+-tree for serial updates



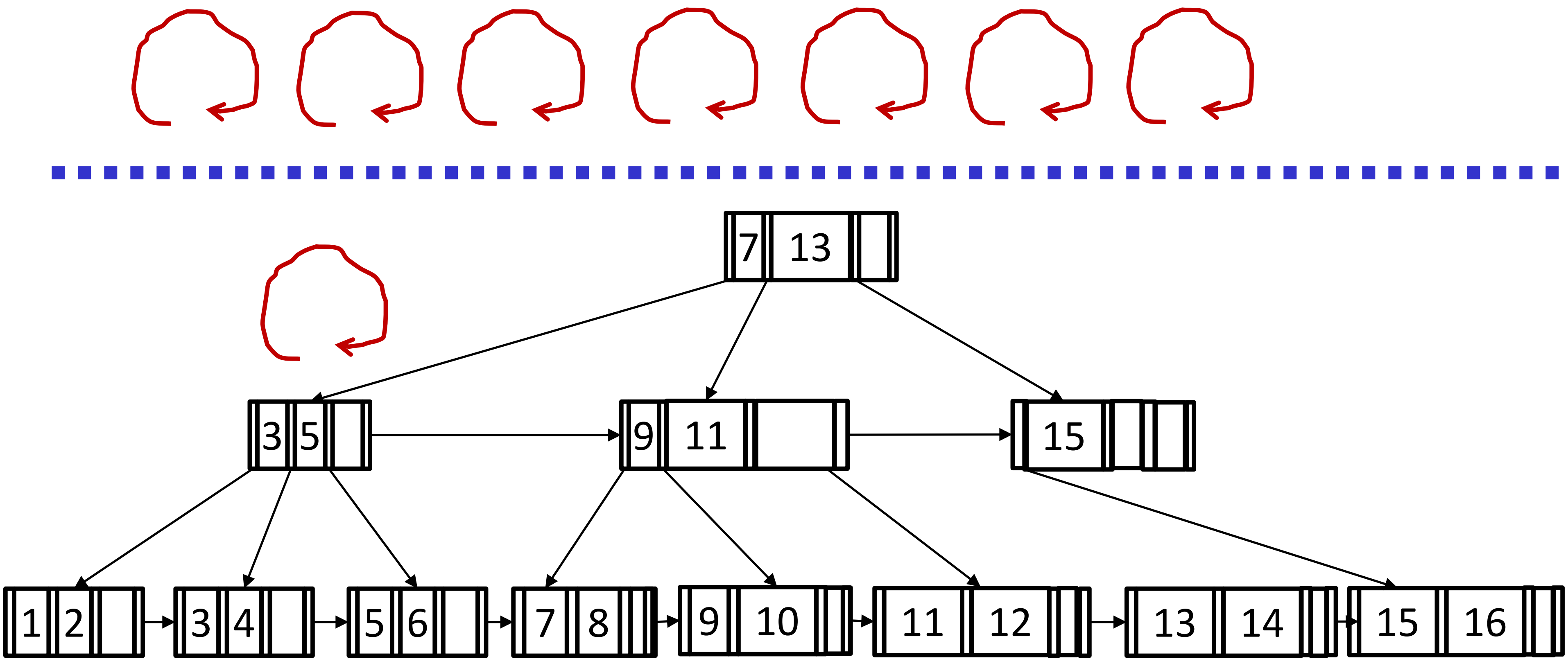
Waiting on a queue…

- Only one can access the tree
- No Concurrency



Philip L. Lehman, S. Bing Yao: Efficient Locking for Concurrent Operations on B-Trees. ACM Trans. Database Syst. 6(4): 650-670 (1981)

Example (Tree-Locking)



Concurrent tree

Lock-Coupling

- B+-tree for concurrent updates
- Top to bottom
- Locking area is narrower

$While\ (m \neq leaf)\ \{$

$\quad Lock(m)$

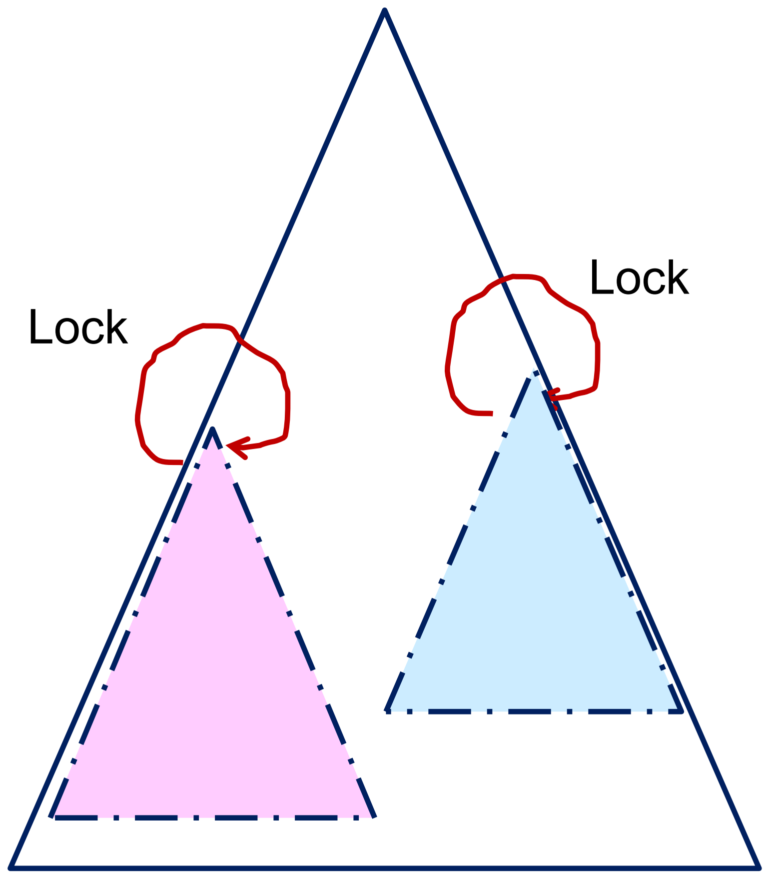
$\quad if\ (child\ is\ full)\ wait$

$\quad n \leftarrow m.child$

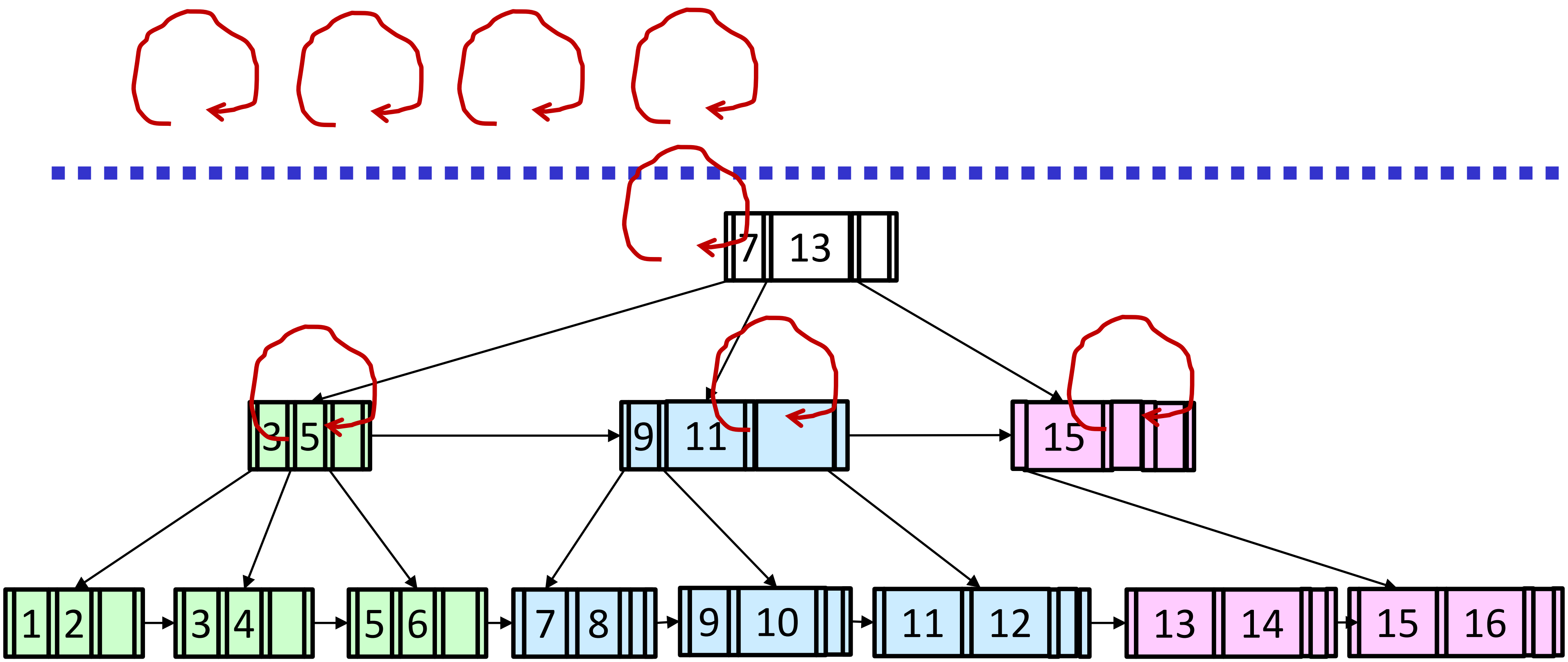
$\quad Lock(n)$

$\quad Unlock(m)$ Unlock parent

$\}$



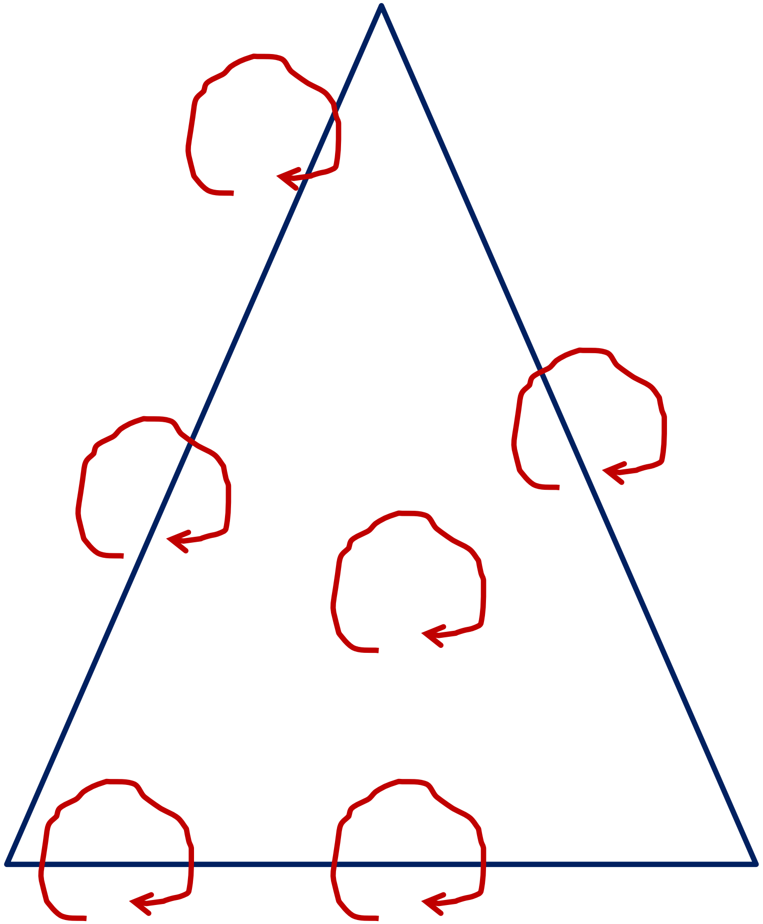
Example (Lock-Coupling)



Concurrent tree

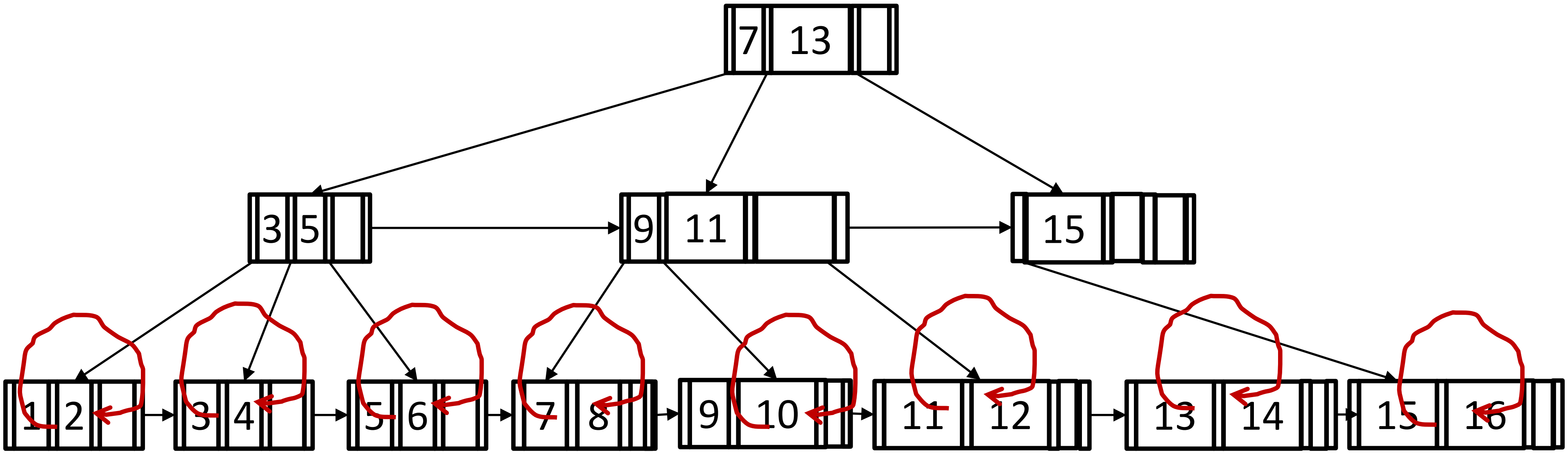
B-link-tree

- B+-tree for concurrent updates
  – Multithreading
- Additional data structure
  – high_key
  – Right_link
- Features
  1. Deadlock free
  2. Sorted keys
  3. PostgreSQL



Philip L. Lehman, S. Bing Yao: Efficient Locking for Concurrent Operations on B-Trees. ACM Trans. Database Syst. 6(4): 650-670 (1981)

Example (B-link-tree)



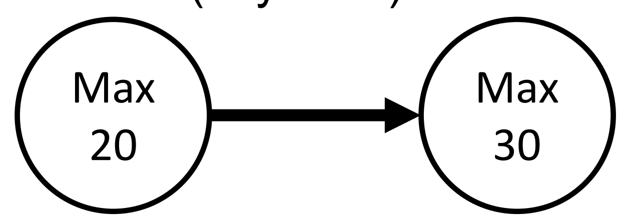
insert

```
leaf = find_leaf (key);    No locks!
lock(leaf);   Lock is acquired at leaf
leaf = move_right (leaf, key);
If (safe) {
    insert_in_leaf(leaf, key, data);
    unlock(leaf);
} else { // split
    split();
    insert_in_parent(leaf, leaf.high_key,
right);
}
```

move_right

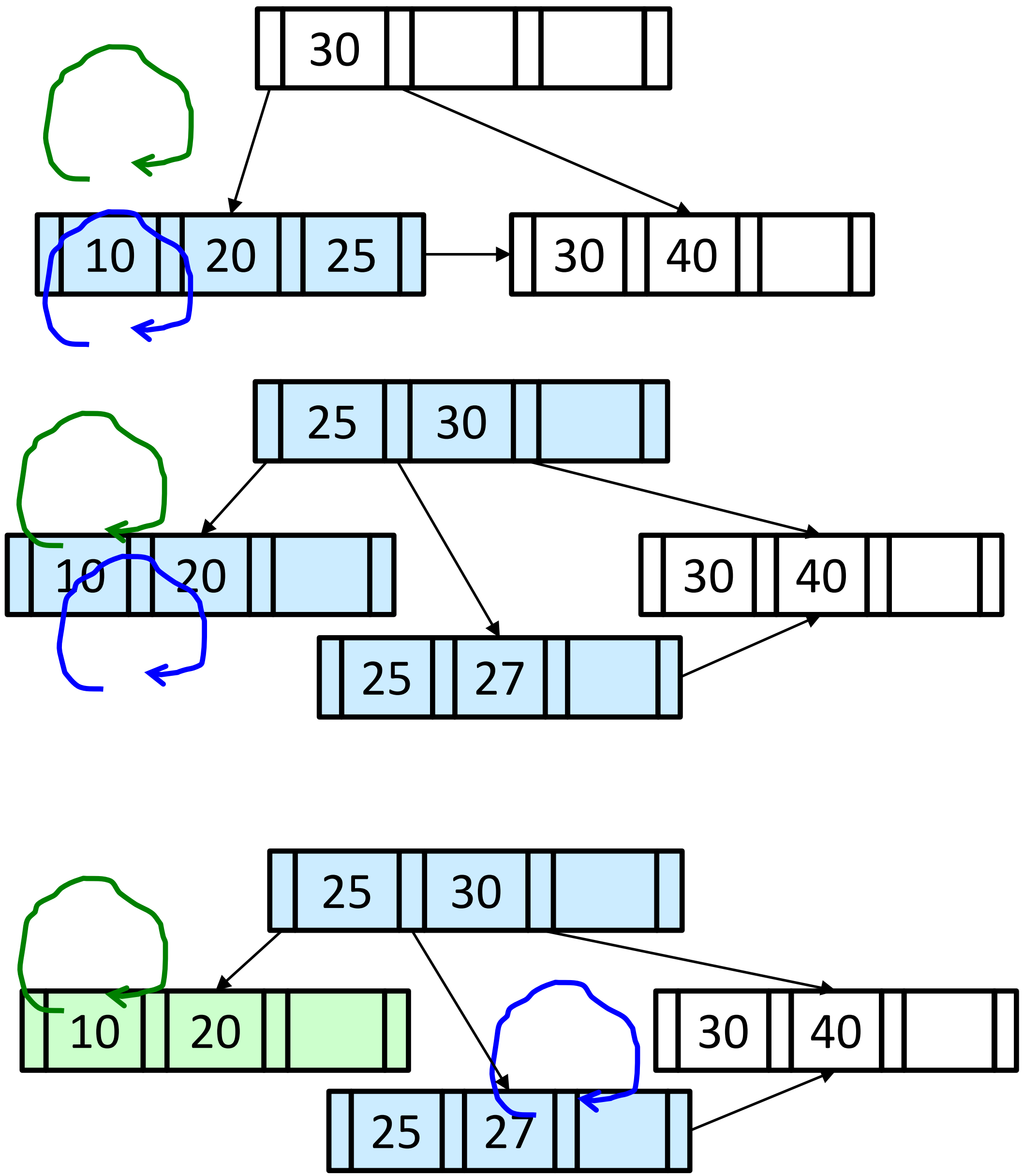
```
while (n.high_key < key)  {
     lock(n.right_link);
     unlock(n);
     n = n.right_link;
 }
 return n;
```

1. Find leaf (key = 25)   3. move_right



2. 20 < 25   4. ~~30 < 25~~

TX1: insert (27) TX2: search (25)



TX1: root

TX1: left

TX1: lock left

TX2: root

TX2: left leaf

TX2: stop at left (blocked)

TX1: Split

TX1: lock root

TX1: unlock left

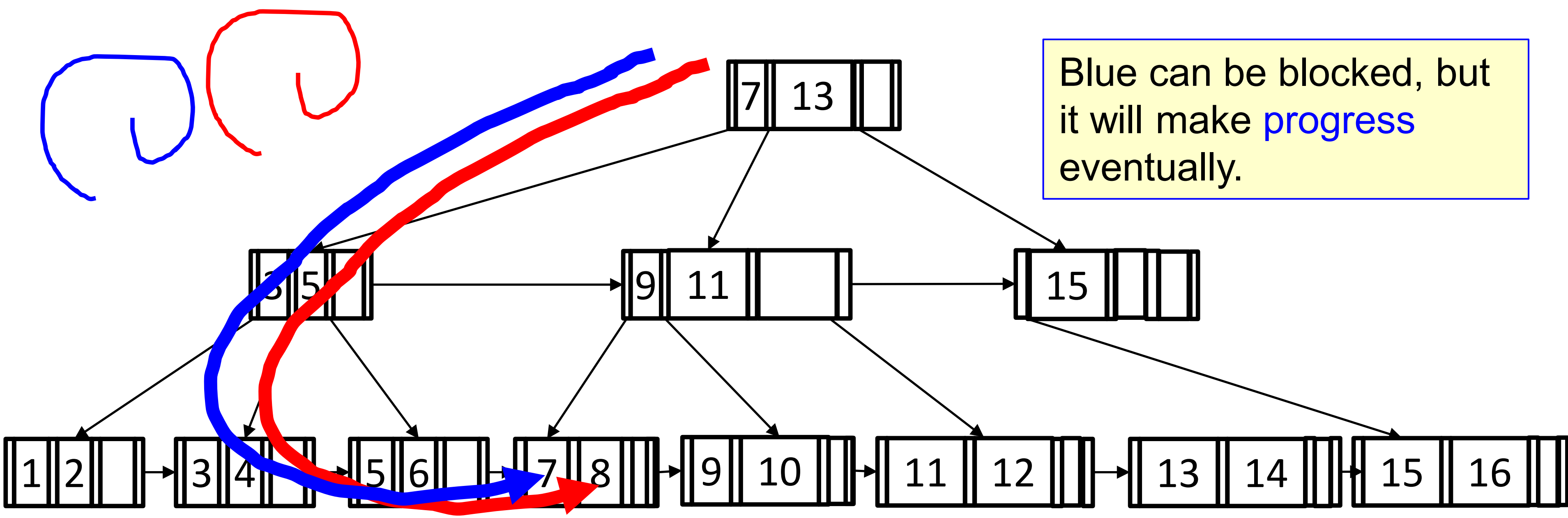
TX2: lock left

TX2: what’s? -> move_right

Why deadlock? Two directions.

$$wl_a(x); wl_b(y); wl_a(y); wl_b(x);$$

One direction in B-link



Blue can be blocked, but it will make progress eventually.

Hard to implement…
Why are you there?

- Cause
  - Inappropriate traversal
- How to fix?
  - Approach: pencil
    - No GDB or valgrind…
  - Update order
    - Pointers from right to left
    - Keys from right to left

Ideal
1, 2, 3

Real…
1, 3, 2

PostgreSQL Code

Lehman and Yao don't require read locks, but assume that in-memory copies of tree pages are unshared. Postgres shares in-memory buffers among backends.

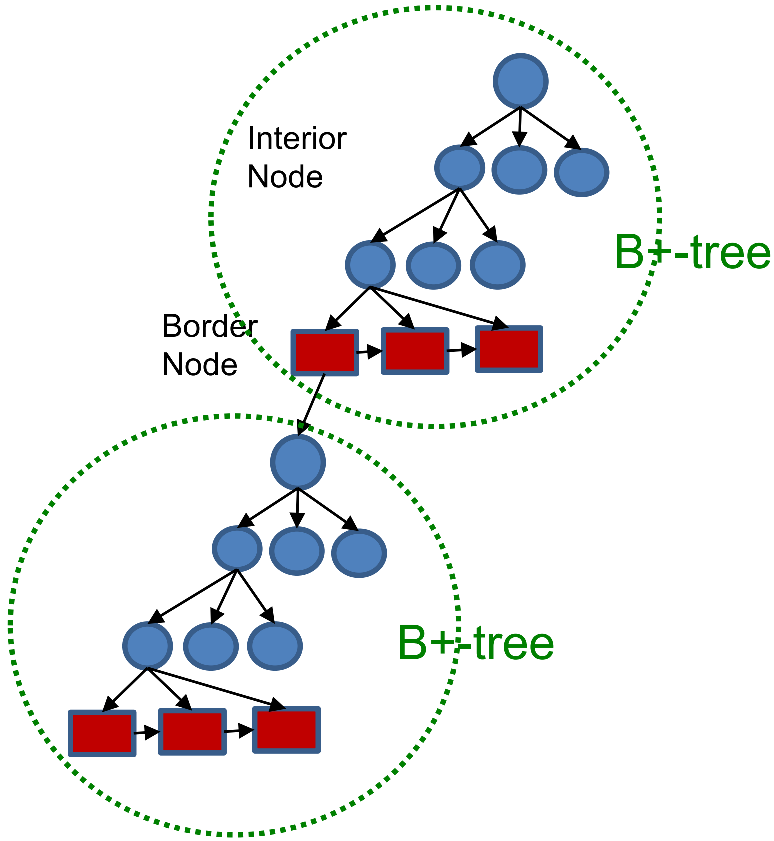
As a result, we do page-level read locking on btree pages in order to guarantee that no record is modified while we are examining it.

This reduces concurrency but guarantees correct behavior. An advantage is that when trading in a read lock for a write lock, we need not re-read the page after getting the write lock. Since we're also holding a pin on the shared buffer containing the page, we know that buffer still contains the page and is up-to-date.

https://github.com/postgres/postgres/tree/master/src/backend/access/nbtree

Concurrent trees

- OLFIT
  - S. K. Cha, S. Hwang, K. Kim, and K. Kwon. Cache conscious concurrency control of main-memory indexes on shared-memory multiprocessor systems. PVLDB’01.
- PALM
  - J. Sewall, J. Chhugani, C. Kim, N. Satish, and P. Dubey. PALM: Parallel architecture-friendly latch-free modifications to B+ trees on many-core processors. PVLDB’11.
- Masstree
  - Y. Mao,, et, al. Cache craftiness for fast multicore key-value storage. EuroSys’12. **SILO.**
- Bw-tree
  - J. Levandoski, et, al. The Bw-Tree: A B-tree for new hardware platforms. ICDE’13. **MS SQL Server (Hekaton)**
  - Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, Michael Kaminsky, David G. Andersen: Building a Bw-Tree Takes More Than Just Buzz Words. SIGMOD’18.



Data: read set R, write set W, node set N,
global epoch number E

// Phase 1

for *record*, *new-value* **in** sorted(W) **do**
    lock(*record*);
compiler-fence();
$e \leftarrow E$; *// serialization point*
compiler-fence();

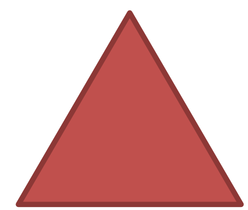
// Phase 2

for *record*, *read-tid* **in** R **do**
    if *record.tid* $\neq$ *read-tid* **or not** *record.latest*
        or (*record.locked* **and** *record* $\notin W$)
    then abort();
for *node*, *version* **in** N **do**
    if *node.version* $\neq$ *version* **then** abort();
commit-tid $\leftarrow$ generate-tid(R, W, e);

// Phase 3

for *record*, *new-value* **in** W **do**
    write(*record*, *new-value*, *commit-tid*);
    unlock(*record*);

If one inserts a data item,
a leaf version is updated, then abort
Leaf node: bottom level node.

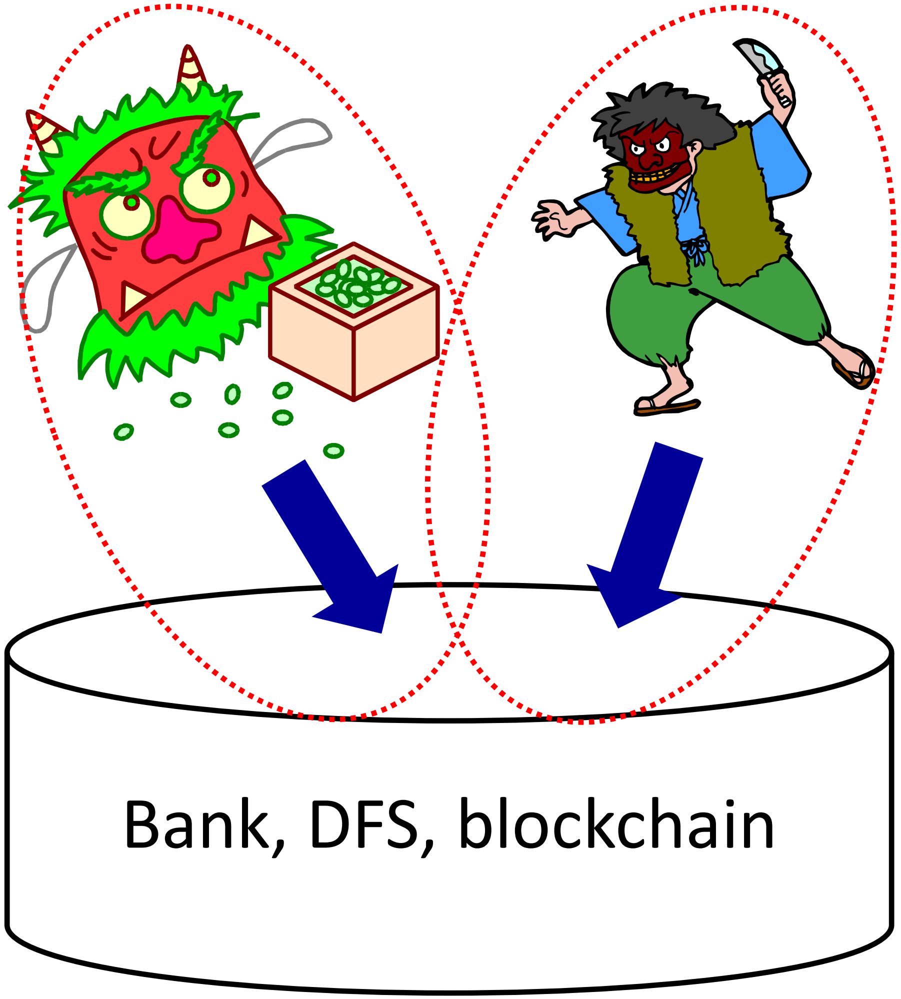


What is this?
Phantom avoidance
using index

Agenda

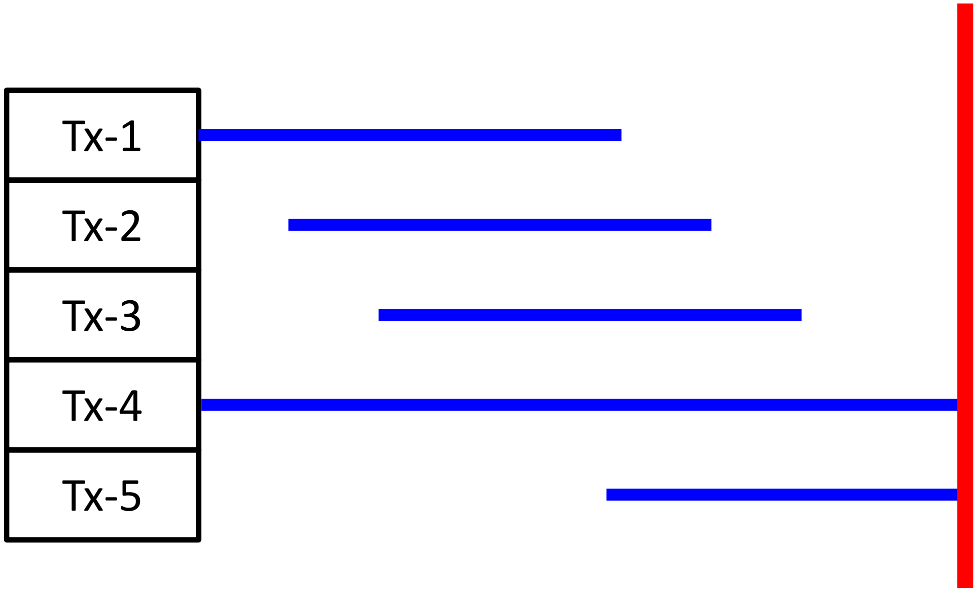
- Transaction
- Silo CC
- Concurrent index
- Silo recovery
- Modern CC and future directions

Transaction Processing System = Concurrency Control + Recovery



Motivation

- Atomicity
  - Transactions may abort ("Rollback")
- Durability
  - What if DBMS stops running ?
- Preferred
  - T1, T2, T3: durable
  - T4, T5: abort



Write Log records on normal process.

Repeat logs on crash recovery.

ARIES Algorithm (2/2)

- Analysis
- Redo (repeating the history)
  - Repeat all activities starting from the appropriate point in the log. And the state of the database is returned to the time of failure.
- Undo
  - Eliminate uncommitted transactions (UNDO). That is, it reflects only the activity of the committed transaction.

Data: read set R, write set W, node set N,
global epoch number E

// Phase 1
for *record*, *new-value* **in** sorted(W) **do**
  lock(*record*);
compiler-fence();
$e \leftarrow E$; *// serialization point*
compiler-fence();

Epoch is necessary for recovery

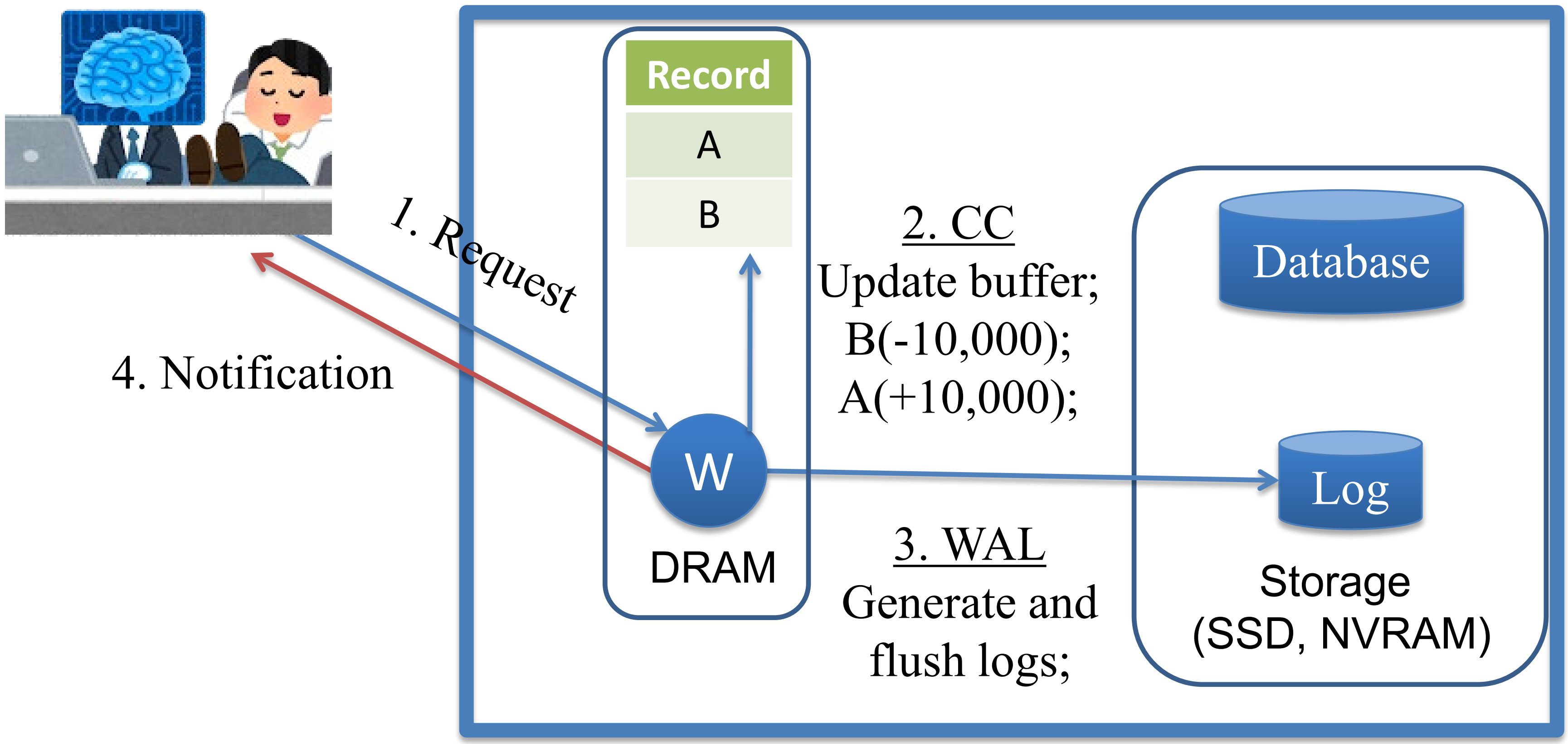
// Phase 2
for *record*, *read-tid* **in** R **do**
  if *record.tid* $\neq$ *read-tid* **or not** *record.latest*
    or (*record.locked* **and** *record* $\notin W$)
  then abort();
for *node*, *version* **in** N **do**
  if *node.version* $\neq$ *version* **then** abort();
commit-tid $\leftarrow$ generate-tid(R, W, e);

// Phase 3
for *record*, *new-value* **in** W **do**
  write(*record*, *new-value*, *commit-tid*);
  unlock(*record*);

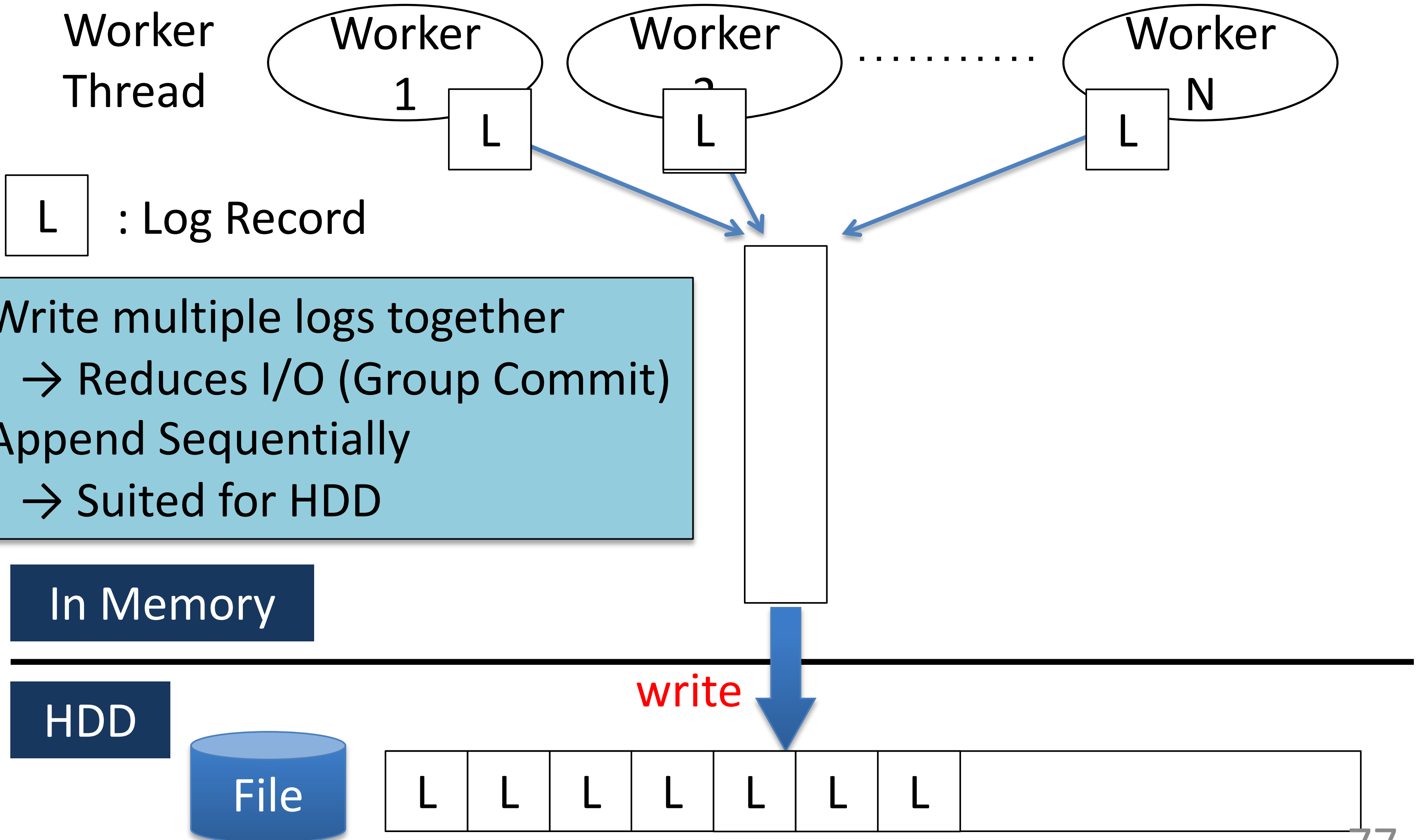
Review: Recovery for Write Ahead Logging (WAL)

Transfer A to B 10000 yen

Write log before modifying DB

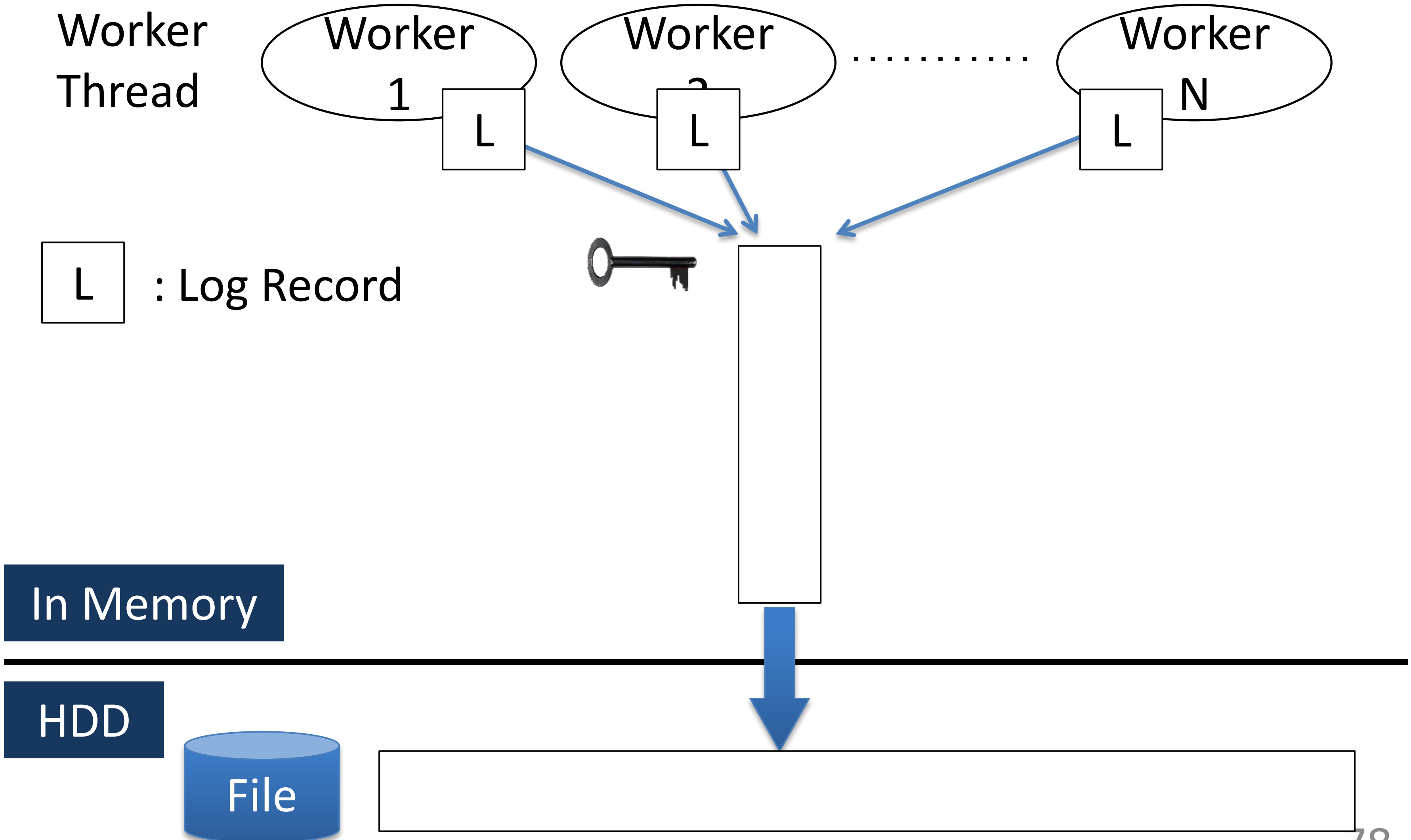


Problem of WAL (1)
Storage Access



Write multiple logs together
→ Reduces I/O (Group Commit)
Append Sequentially
→ Suited for HDD

Problem of WAL (2)
Mutual Execution on Log Insert



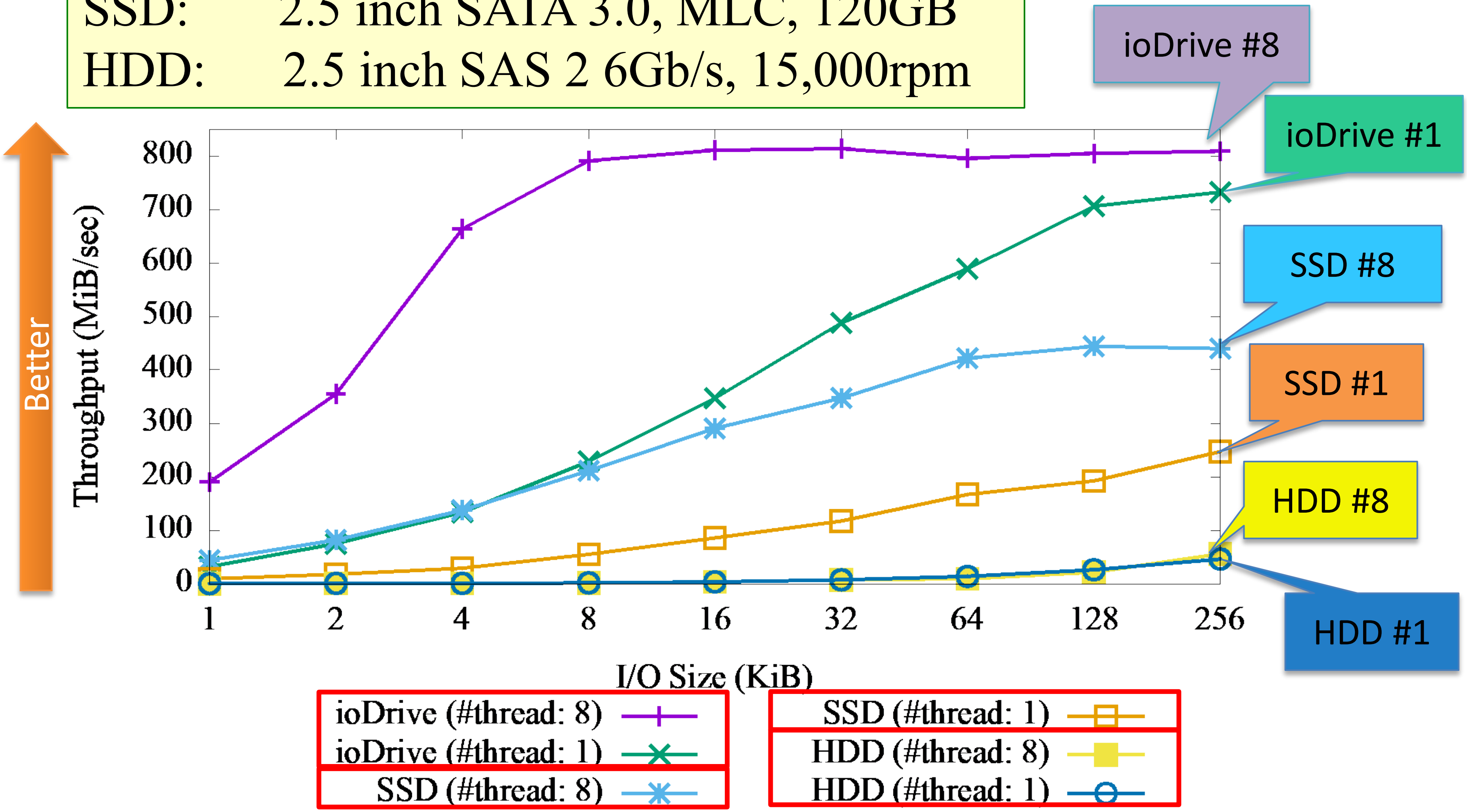
Change of Architecture

- No more CPU frequency
  - Shift to many-cores
- Development of Flash/NVRAM
  - Parallel access with multiple channel provides high performance.

Serial WAL is out-of-date for today's architecture

Performance of Parallel Write

ioDrive: PCI Express 2.0 x8, SLC, 160GB
SSD: 2.5 inch SATA 3.0, MLC, 120GB
HDD: 2.5 inch SAS 2 6Gb/s, 15,000rpm



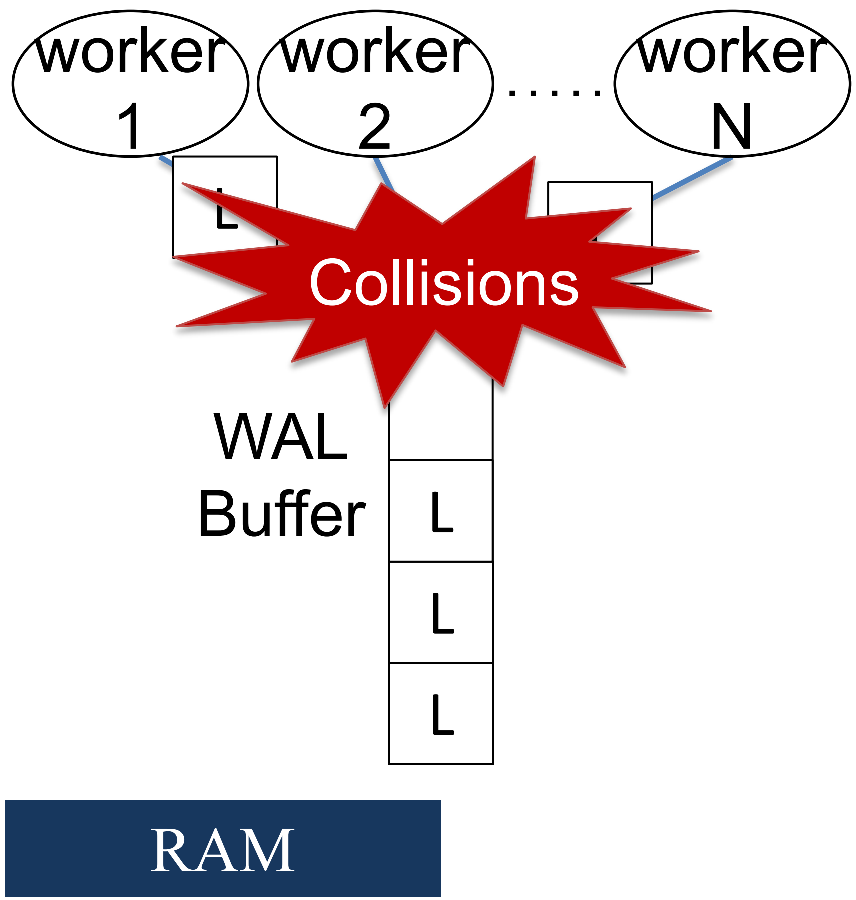
Today:
Parallel Write Ahead Logging

1. Multiple WAL buffers
2. Multiple WAL files
3. Early Lock Release

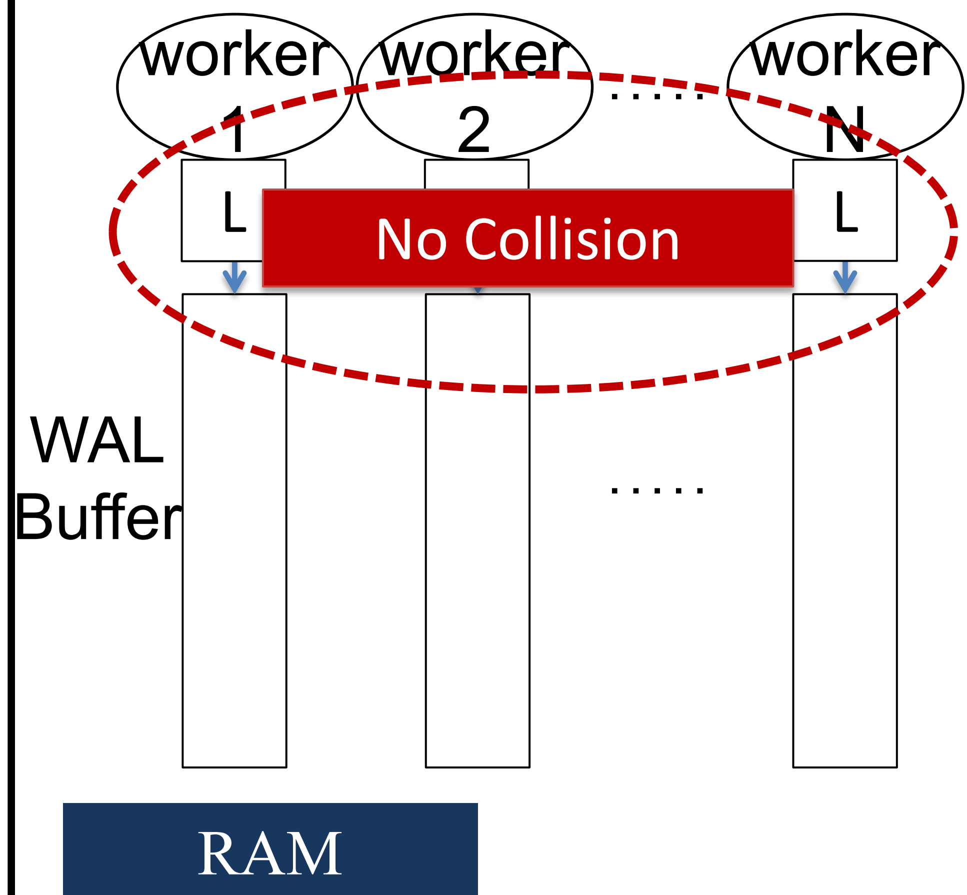
神谷，星野，川島，建部：並列ログ先行書き込み手法P-WAL，IPSJ（TOD），2017.
Hideaki Kimura: FOEDUS: OLTP Engine for a Thousand Cores and NVRAM. SIGMOD’15
Stephen Tu, et al.: Speedy transactions in multicore in-memory databases. SOSP’13
Tianzheng Wang, et al.: Scalable Logging through Emerging Non-Volatile Memory. PVLDB’14

P-WAL (1/3)
Multiple WAL buffers

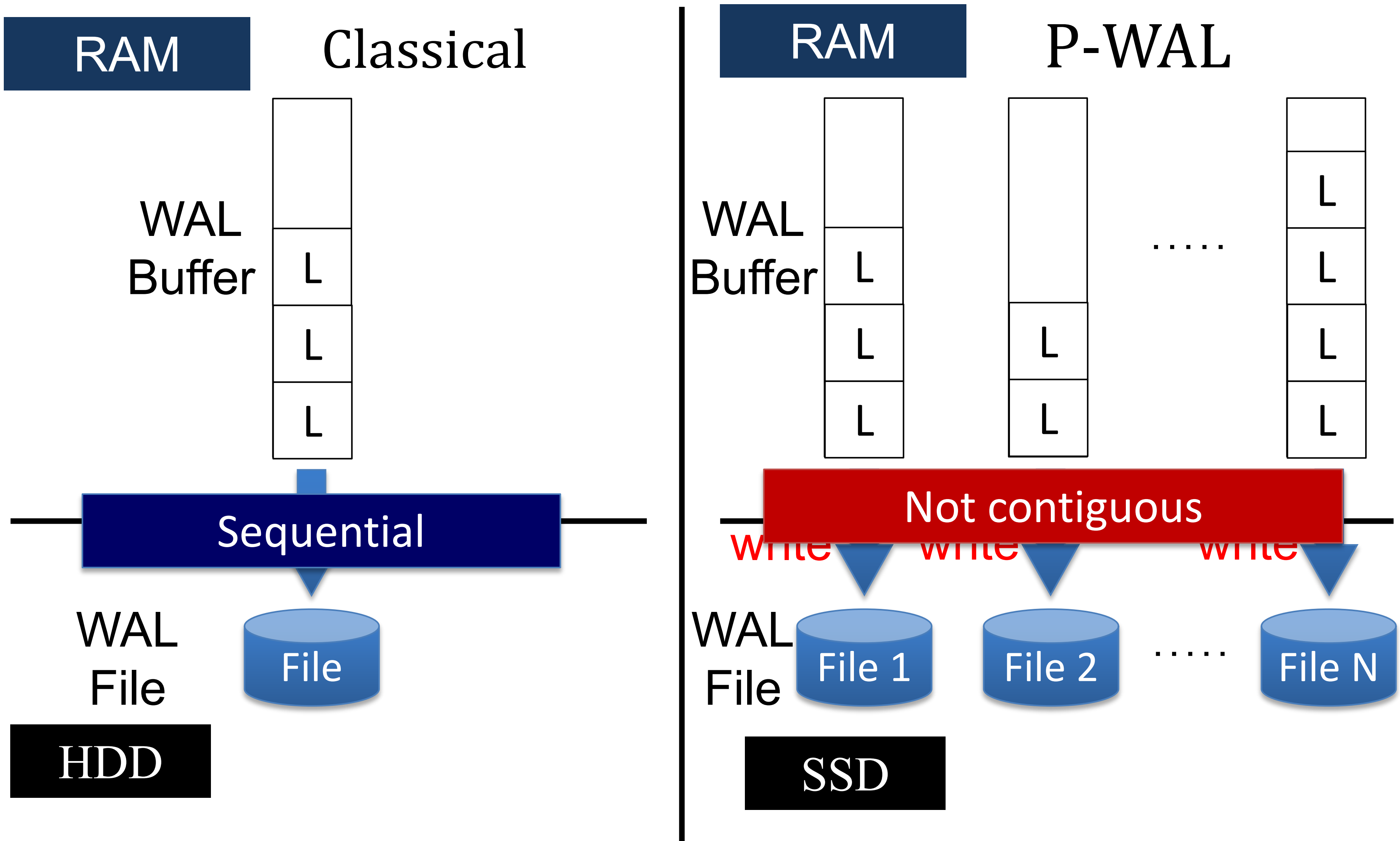
Conventional WAL



P-WAL



P-WAL (2/3)
Multiple WAL files



P-WAL (3/3)
Early Lock Release

- Algorithm
  - Unlock data items before writing log records
- Pros
  - Reduce locking time, provides better throughput
- Cons
  - Worsen latency

D1 D2 D3

1. Lock data
2. Modify data
3. Generate log
4. Write log
5. Unlock data

ELR

1. Lock data
2. Modify data
3. Generate Log
4. Unlock data
5. Write log

Ryan Johnson, et al.: Aether: A Scalable Approach to Logging. PVLDB 2010

Early Lock Release

- Why do we conduct ELR?
  - It improves throughput
  - Because blocking time by lock waiting is reduced
- Dirty read anomaly?
  - T1 may abort after T2 reads x

T1	T2
Write(x)	
	Read(x)
Abort	
	Commit

Coping with dirty read: *notification control*
Threads read data in secret to users…

T1	T2
Write(x)	
	Read(x)
Commit or Abort	
	Write(x)
	Commit

1. Here, future of T1 is unknown. Let's go! (optimistic)
2. If T1 pre-commits (CC ok), T2 pre-commits (CC ok). Flush logs. Go 4.
3. Else T1 aborts, also T2 (cascading abort (T3,..,Tn))
4. Wait until history completes because we are not sure whether prev. logs are all written to storage.(== no lack of log)

並列ログ先行書き込み手法P-WAL, IPSJ（TOD）10(1), 24-39, 2017-03-22

Q. Does P-WAL require a centralized counter (LSN)?

Coping with dirty read: *notification control*

Threads read data in secret to users…

T1	T2
Write(x)	
	Read(x)
Commit or Abort	
	Write(x)
	Commit

1. Here, future of T1 is unknown. Let's go! (optimistic)
2. If T1 pre-commits (CC ok), T2 pre-commits (CC ok). Flush logs. Go 4.
3. Else T1 aborts, also T2 (cascading abort (T3,..,Tn))
4. Wait until history completes or prev. logs are all written to storage.(== no lack of log)

(MIN_MAX_F_LSN=5)

F_LSN=8 F_LSN=5 F_LSN=9

6,7

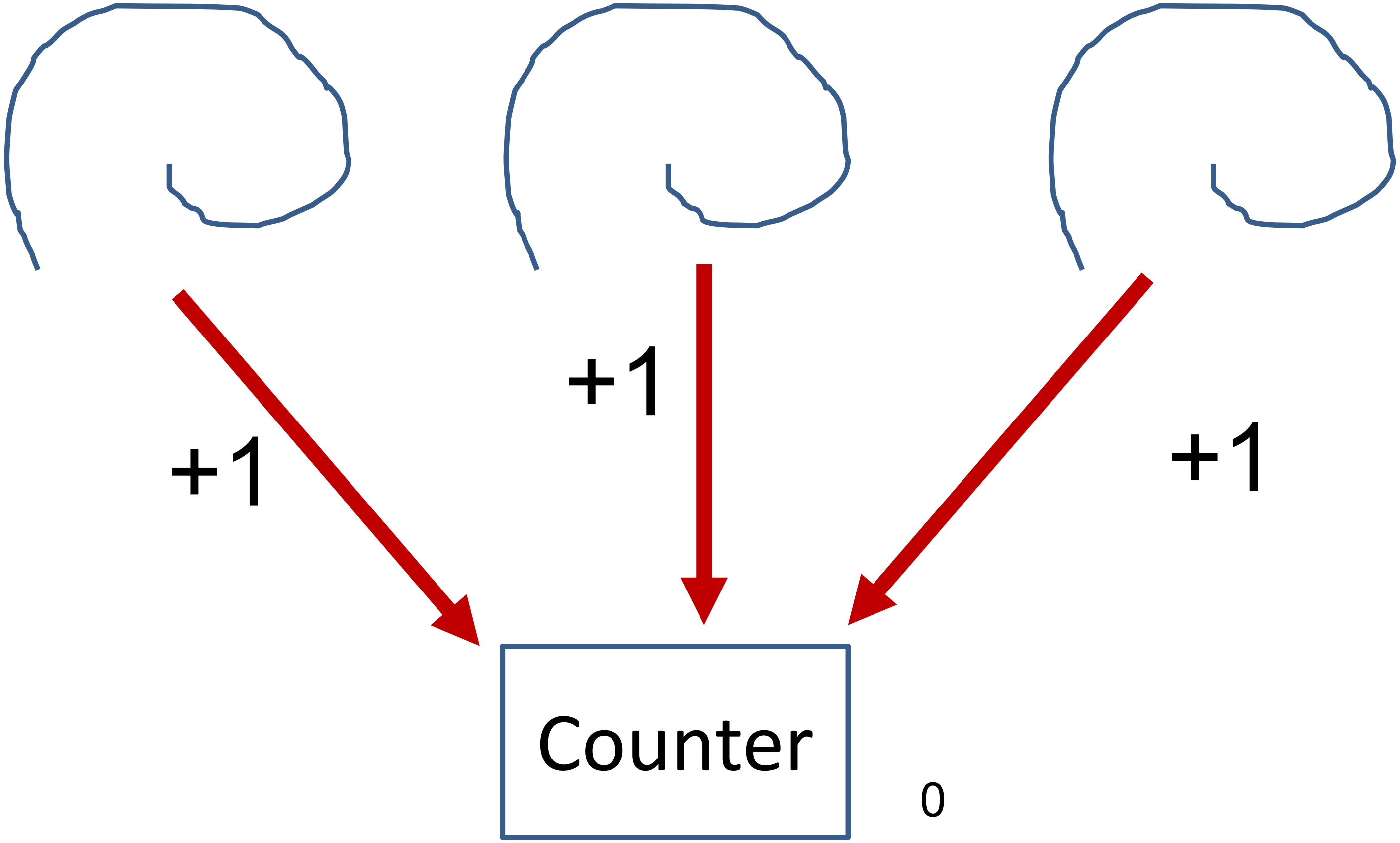
TX-1	TX-2	TX-3
TX-4	TX-5	TX-9
TX-8		

Complete history by 5
Can notify by 5

並列ログ先行書き込み手法P-WAL，IPSJ（TOD）10(1), 24-39, 2017-03-22

Q. Does P-WAL require a centralized counter (LSN)?

Thinking about Locking



Lock library

```
void *
worker(void *arg)
{
  for (uint i = 0; i < NbLoop; i++) {
    if ((pthread_mutex_lock(&Lock)) == -1) ERR;
    Counter++;
    if ((pthread_mutex_unlock(&Lock)) == -1) ERR;
  }

  return NULL;
}
```

Any problem?

counter_pthread_lock.cc

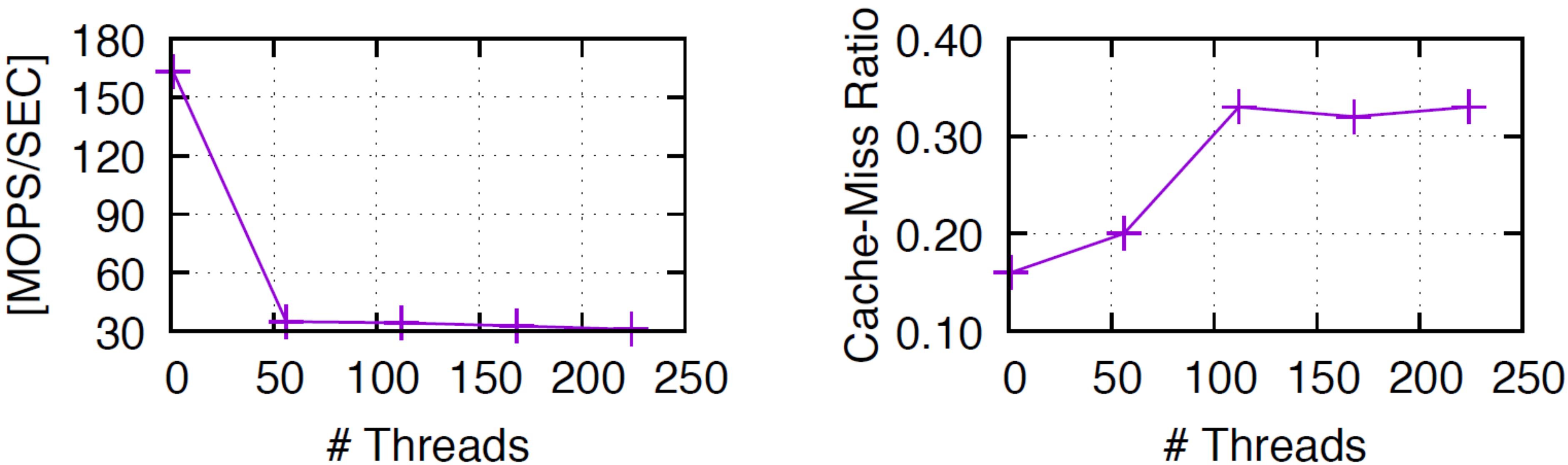
% time ./counter_pthread_lock

Atomic Add

```
void *
worker(void *arg)
{
  for (uint i = 0; i < NbLoop; i++) {
    atomic_fetch_add(&Counter, 1);
  }
  return NULL;
}
```

% time ./counter_fetch_and_add

Atomic Add at Many-Core Era



(a) Throughput. (b) Cache-miss ratio.

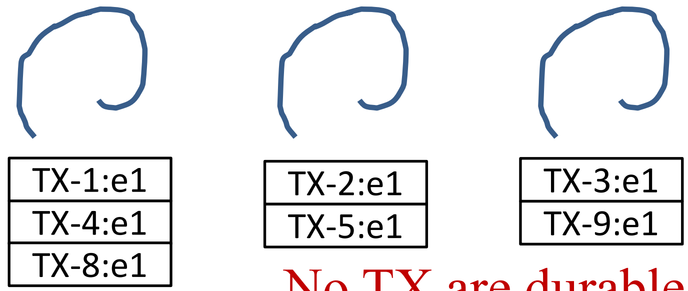
Figure 6: Scalability of fetch_add

Epoch based synchronization

All TXs in an epoch become durable at the same time.

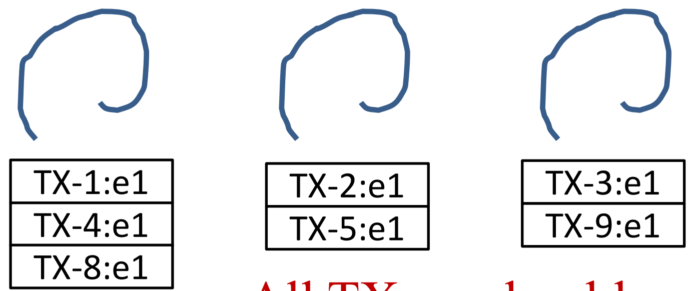
- Epoch
  - A centralized counter, updated only by a special thread.
  - Epoch *e*
  - Durable epoch *d*
- Threads
  - Worker
    - On commit, reads *e*. No update to *e*. No lock is necessary.
    - Writes logs with *e*.
  - Notifier
    - Wait until *d* is less than *e* in TX.

e = 1, d = 0



No TX are durable.

e = 2, d = 1



All TX are durable.

All the logs at epoch=Ek is persisted at the same time.

Q. Recovery order and CC order are not the same?
CC: [1->4->8 or 2->5 or 3->9], Rec: [1,4,8,2,5,3,9] same.

CC order <= Rec. order (Never inverse order)

Data: read set R, write set W, node set N,
global epoch number E

// Phase 1

for *record*, *new-value* **in** sorted(W) **do**
lock(*record*);
compiler-fence();
$e \leftarrow E$; *// serialization point*
compiler-fence();

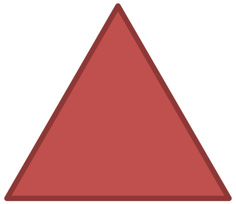
Read epoch here.

// Phase 2

for *record*, *read-tid* **in** R **do**
if *record.tid* $\neq$ *read-tid* **or not** *record.latest*
or (*record.locked* **and** *record* $\notin W$)
then abort();
for *node*, *version* **in** N **do**
if *node.version* $\neq$ *version* **then** abort();
commit-tid $\leftarrow$ generate-tid(R, W, e);

// Phase 3

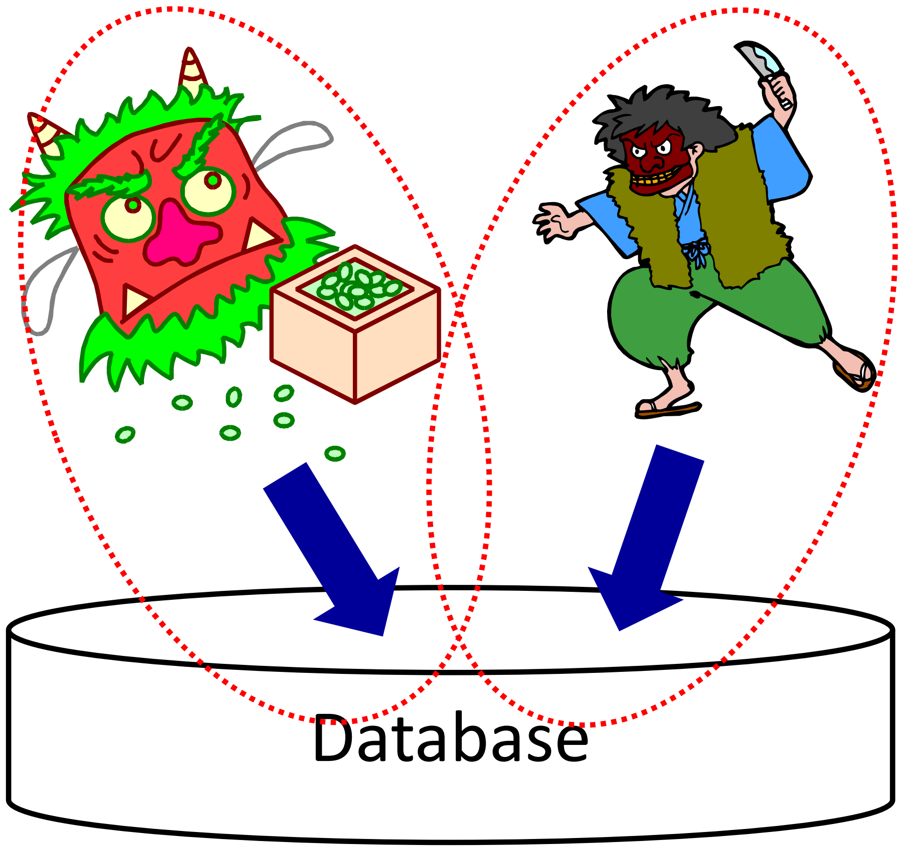
for *record*, *new-value* **in** W **do**
write(*record*, *new-value*, *commit-tid*);
unlock(*record*);



What is this?
Phantom avoidance using index

Q. Why does a thread read epoch at this place?

Transaction Processing System = Concurrency Control + Recovery



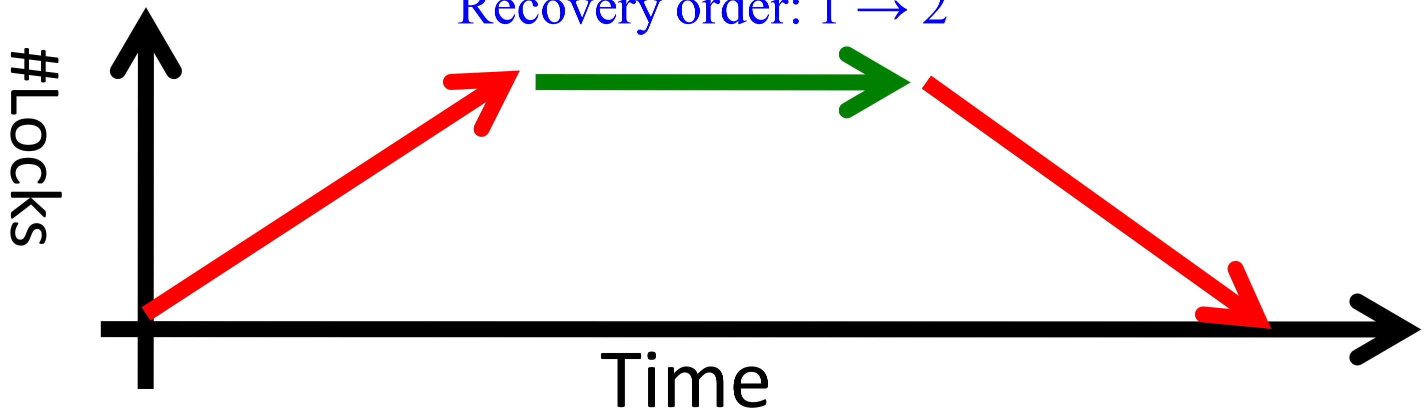
Concurrency Control

Recovery

Applications: bank, credit card, distributed file system (spotify), blockchain…

Place of the Epoch

Serialization point of 2PL
For r1(x)w2(x),
CC order: 1 → 2
Recovery order: 1 → 2



Place of the Epoch

r1(x);

w2(x);

Data: read set R, write set W, node set N,
global epoch number E

// Phase 1
for *record, new-value* **in** sorted(W) **do**
lock(*record*);
compiler-fence();

compiler-fence();

// Phase 2
for *record, read-tid* **in** R **do**
if $record.tid \neq read\text{-}tid$ **or not** *record.latest*
or ($record.locked$ **and** $record \notin W$)
then abort();

Why not here?

for *node, version* **in** N **do**
if $node.version \neq version$ **then** abort();
$commit\text{-}tid \leftarrow$ generate-tid(R, W, e);

// Phase 3
for *record, new-value* **in** W **do**
write(*record, new-value, commit-tid*);
unlock(*record*);

r1(x); w2(x);c1;c2

V1(x); L2(x);
Then, CC order is determined to 1->2
Because read lock by T1 is ok, then
Write lock by T2 is ok.

T2: epoch = 10
epoch++
T1: epoch = 11

CC order: T1 → T2
Recovery order: T2 → T1

No, Inverse Order!

Place of the Epoch

Data: read set R, write set W, node set N,
global epoch number E

// Phase 1
for *record*, *new-value* **in** sorted(W) **do**
lock(*record*);
compiler-fence();

compiler-fence();
// Phase 2
for *record*, *read-tid* **in** R **do**
if *record.tid* $\neq$ *read-tid* **or not** *record.latest*
or (*record.locked* **and** *record* $\notin W$)
then abort();
Why not here?
for *node*, *version* **in** N **do**
if *node.version* $\neq$ *version* **then** abort();
commit-tid $\leftarrow$ generate-tid(R, W, e);
// Phase 3
for *record*, *new-value* **in** W **do**
write(*record*, *new-value*, *commit-tid*);

r1(x);

w2(x);

r1(x); w2(x);c1;c2

V1(x); L2(x);
Then, CC order is determined to 1->2
Because read lock by T1 is ok, then
Write lock by T2 is ok.

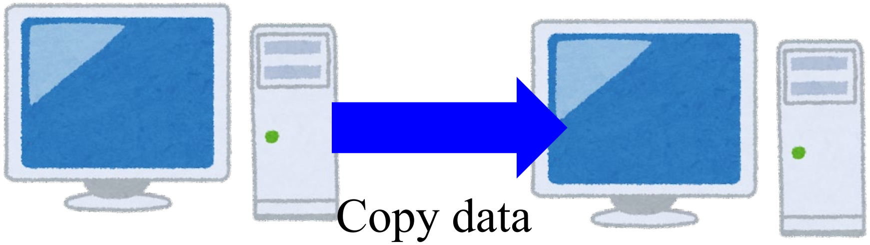
T2: epoch = 10
epoch++
T1: epoch = 11

CC order: T1 → T2
Recovery order: T2 → T1

No, Inverse Order!

Q. Is inverse order a problem? T1 is read only, it is not logged. Any problem?

Hybrid transaction/analytical processing (HTAP)



OLTP (transaction)
Update-intensive

OLAP (analytics)
Read-only

- Distributed computing for performance.
  - DC can be used for availability too.
- OLTP is update only, OLAP is read only for high-speed analysis.
- OLTP update is copied to OLAP node in real-time.
- If OLTP and OLAP are done in a single node, many aborts…
- This "HTAP" scheme is often used in real business (e.g. SAP HANA)

r1(x);

w2(x);

r1(x); w2(x);c1;c2

V1(x); L2(x);
Then, CC order is determined to 1->2
Because read lock by T1 is ok, then
Write lock by T2 is ok.

T2: epoch = 10
epoch++
T1: epoch = 11

CC order: T1 → T2
Recovery order: T2 → T1

No, Inverse Order!

Q. Is inverse order a problem? T1 is read only, it is not logged. Any problem?

r1(x) may read w2(x) instead of wk(x)

Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, Samuel Madden: Speedy transactions in multicore in-memory databases. SOSP 2013: 18-32

Data: read set R, write set W, node set N,
global epoch number E

// Phase 1
for *record, new-value* **in** sorted(W) **do**
lock(*record*);
compiler-fence();
$e \leftarrow E$; *// serialization point*
compiler-fence();

Lock

// Phase 2
for *record, read-tid* **in** R **do**
if *record.tid* $\neq$ *read-tid* **or not** *record.latest*
or (*record.locked* **and** *record* $\notin W$)
then abort();
for *node, version* **in** N **do**

Validate

if *node.version* $\neq$ *version* **then** abort();
commit-tid $\leftarrow$ generate-tid(R, W, e);

// Phase 3
for *record, new-value* **in** W **do**
write(*record, new-value, commit-tid*);
unlock(*record*);

Write

Production Example:
LineairDB: https://lineairdb.github.io/LineairDB/html/index.html

Agenda

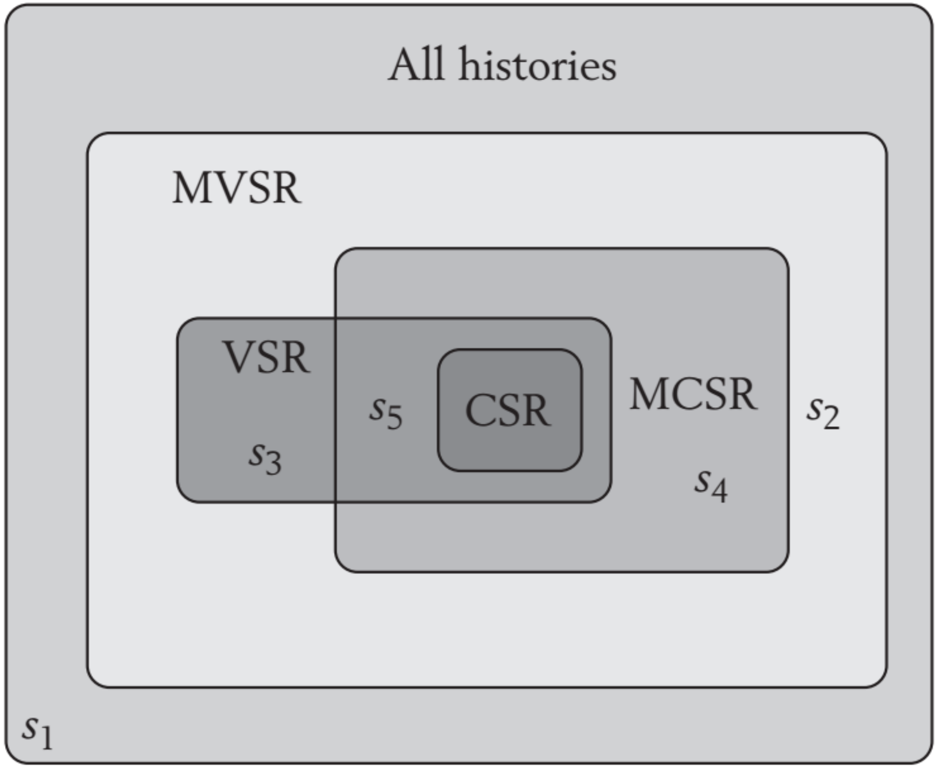
- Transaction
- Silo CC
- Concurrent index
- Silo recovery
- Modern CC and future directions

Modern CC

Method	Year	Conference	Features
Polyjuice	2021	OSDI	Optimistic Machine Learning
Bamboo	2021	SIGMOD	PCC, dirty read
Cicada	2017	SIGMOD	Optimistic Multi-Version
SSN	2017	VLDBJ	Multi-Version
MOCC	2016	VLDB	Optimistic Pessimistic
TicToc	2016	SIGMOD	Optimistic
Silo	2013	SOSP	Optimistic
SI	1995	SIGMOD	Multi-Version
2 Phase Lock	1976	Comm. ACM	Pessimistic

Two Mysteries…

- Really fast?
  - Using different platform? [Cicada]
  - Demonstrating only strong points?
- Why fast?
  - Each paper proposes a variety of fancy techs.
  - CC performance depends on both theory and engineering.
  - What is essential for performance?



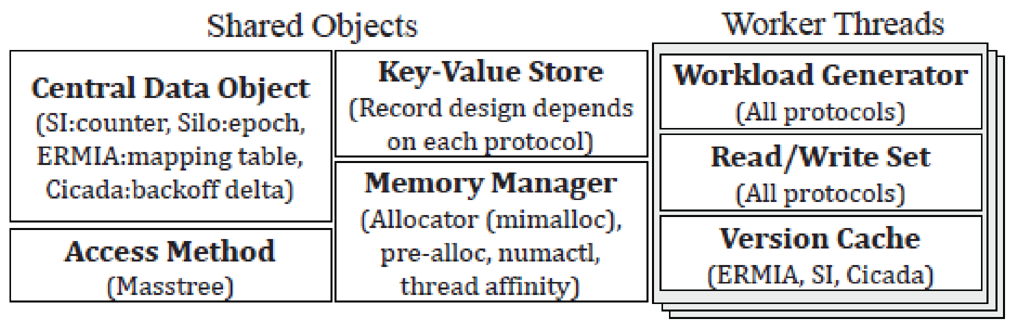
$$s_1 = r_1(x)r_2(x)w_1(x)w_2(x)c_1c_2$$
$$s_2 = w_1(x)c_1r_2(x)r_3(y)w_3(x)w_2(y)c_2c_3$$
$$s_3 = w_1(x)c_1r_2(x)r_3(y)w_3(x)w_2(y)c_2c_3w_4(x)c_4$$
$$s_4 = r_1(x)w_1(x)r_2(x)r_2(y)w_2(y)r_1(y)w_1(y)c_1c_2$$
$$s_5 = r_1(x)w_1(x)r_2(x)w_2(y)c_2w_1(y)w_3(y)c_1c_3$$

From Transactional Information Systems, Weikum, Vossen

Analysis system: CCBench

https://github.com/thawk105/ccbench

1. Shared modules.
2. Careful tuning for performance.
3. Masstree



Performance Factor	CPU Cache		Delay by Conflict			Version Lifetime	
Optimization Method	Decentralized Ordering	Invisible Reads	NoWait or Wait	Adaptive Backoff	ReadPhase Extension(α)	AssertiveVersion Reuse(δ)	Rapid GC
2PL [60]	Org, CCB	—	Org,CCB	CCB	—	—	—
Silo [54]	Org, CCB	Org, CCB	CCB	CCB	CCB	—	—
MOCC [58]	Org, CCB	Org, CCB (β)	—	CCB	CCB	—	—
TicToc(ϵ) [65]	Org, CCB	—	CCB (γ)	CCB	CCB	—	—
SI [32]	—	—	—	CCB	—	CCB	CCB
ERMIA [33]	—	—	—	CCB	—	CCB	CCB
Cicada(ζ) [36]	Org, CCB	—	—	Org, CCB	CCB	CCB	Org, CCB

1. Can we apply opt-method X proposed in protocol A to another protocol B?
2. What is the meaning of the optimization methods.

T. Wang and H. Kimura. Mostly-Optimistic Concurrency Control for Highly Contended Dynamic Workloads on a Thousand Cores. PVLDB, 10(2):49–60, 2016.

Really fast? (MOCC)

Reproducible

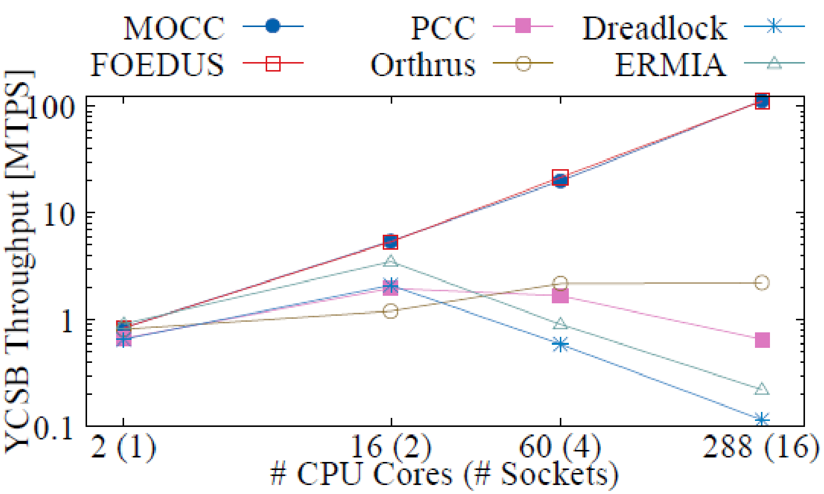
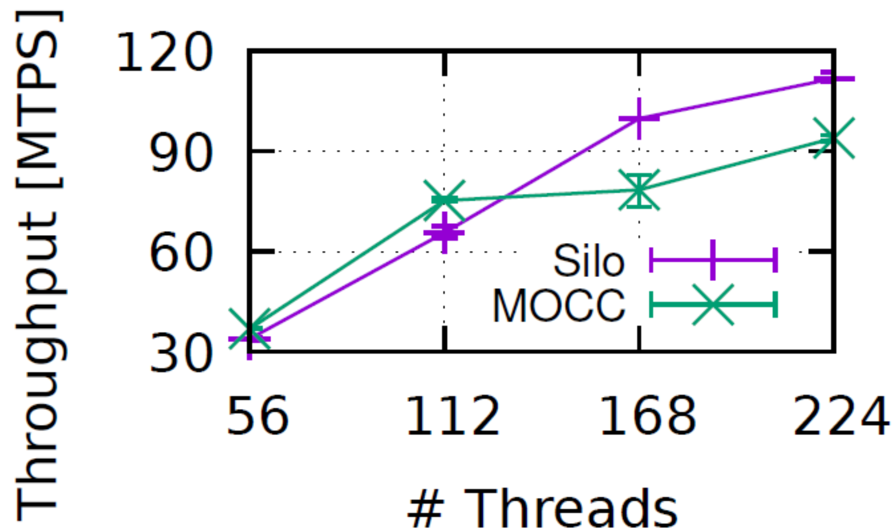


Figure 6: Throughput of a read-only YCSB workload with high contention and no conflict on four machines with different scales. MOCC adds no overhead to FOEDUS (OCC), performing orders of magnitude faster than the other CC schemes.



(a) Read Only Workload.

Yes, reproducible

T. Wang and H. Kimura. Mostly-Optimistic Concurrency Control for Highly Contended Dynamic Workloads on a Thousand Cores. PVLDB, 10(2):49–60, 2016.

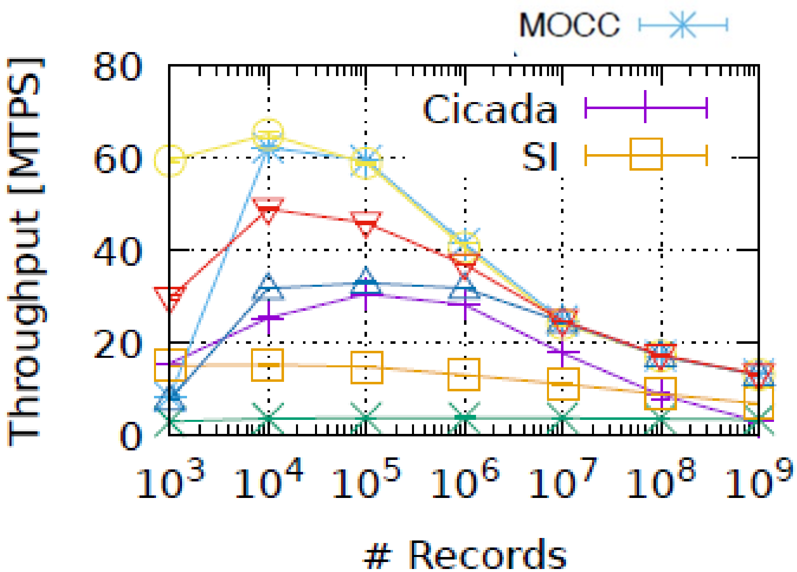
Why fast?

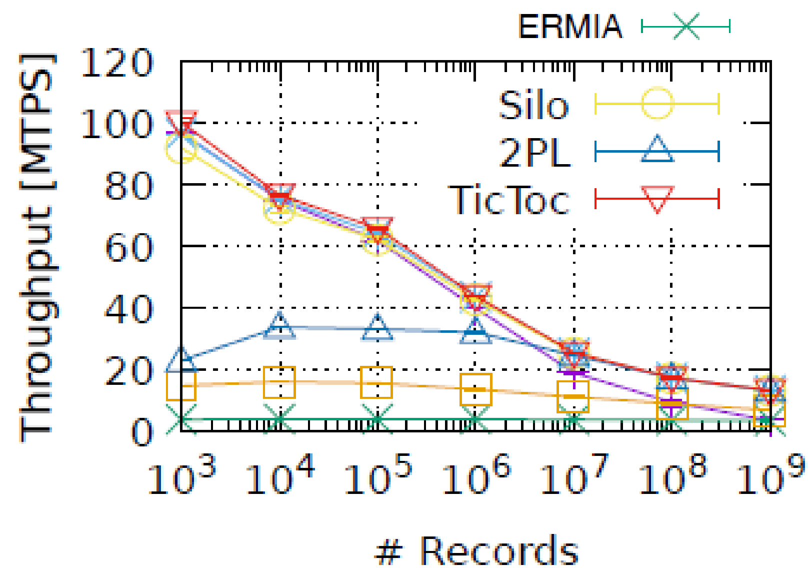
224 threads analysis (others scalability only)

	Cache		Delay		Version
Figure Number	F5	F7	F9	F10	F11
Skew	0	0	0.8	0.9	ζ
Cardinality	α	β	10^8	10^8	10^8
Payload (byte)	4	4	4	ϵ	4
Xact size	10	10	δ	10	10
Read ratio (%)	0,5	γ	50,95	50	50,95
Thread count	Always 224 except from reproduction				
Read modify write	Always off except from reproduction				

Answer : Cache, Delay, Version

Cache-Line-Conflicts

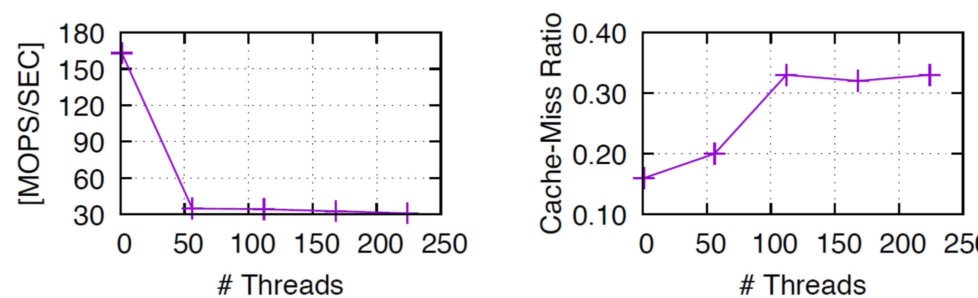




Read=95%, Write=5%

Read=100%, Write=0%

1. SI's scheduling space is largest, but slow
2. Silo/MOCC are nice
3. Centralized ordering is bad



(a) Throughput (b) Cache miss ratio

Figure 6: Scalability of fetch_add

Phantom

T1: more results at 2nd read?

$$r1[P] \ldots w2[y\ in\ P] \ldots c2 \ldots r1[P] \ldots c1$$

P: Predicate

Ex: read <= 10

T1	T2
Read -> x	
	Insert (z) Commit
Read -> x, z Commit	

x (10), y (30)

x (10), y (30), z(3)

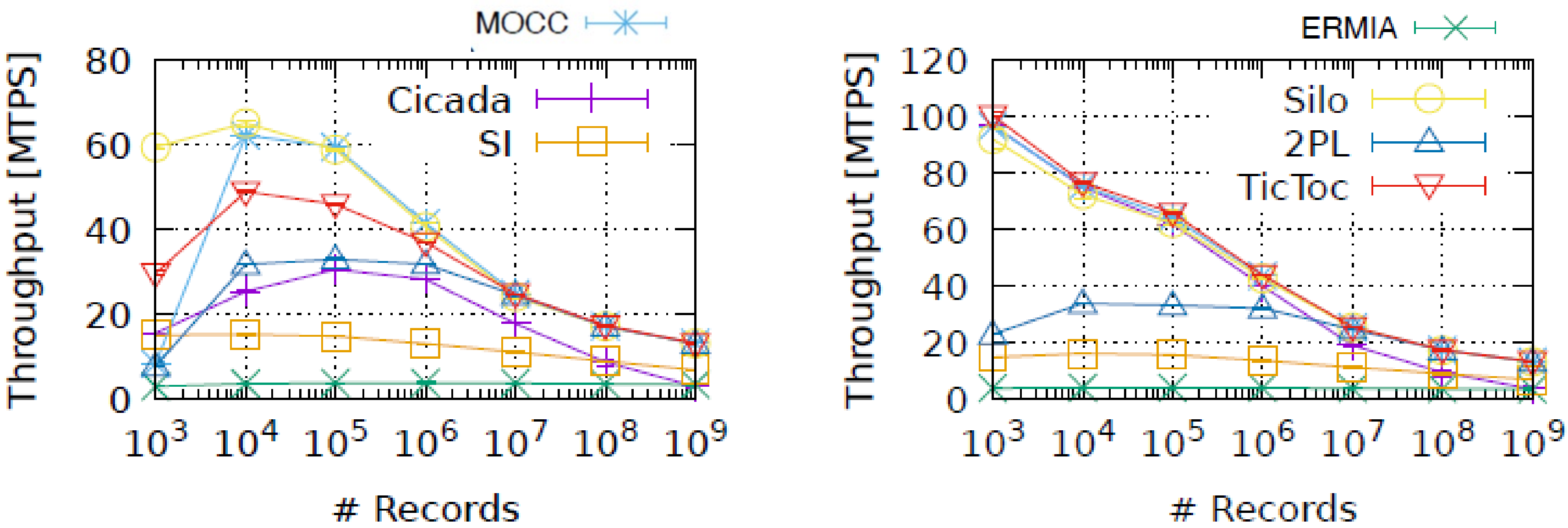
- A weaker isolation level should have a wider scheduling space.
- Wider scheduling space should be faster…

Table 4. Isolation Types Characterized by Possible Anomalies Allowed.

Isolation level	**P0** Dirty Write	**P1** Dirty Read	**P4C** Cursor Lost Update	**P4** Lost Update	**P2** Fuzzy Read	**P3** Phantom	**A5A** Read Skew	**A5B** Write Skew
READ UNCOMMITTED == Degree 1	Not Possible	Possible	Possible	Possible	Possible	Possible	Possible	Possible
READ COMMITTED == Degree 2	Not Possible	Not Possible	Possible	Possible	Possible	Possible	Possible	Possible
Cursor Stability	Not Possible	Not Possible	Not Possible	Sometimes Possible	Sometimes Possible	Possible	Possible	Sometimes Possible
REPEATABLE READ	Not Possible	Not Possible	Not Possible	Not Possible	Not Possible	Possible	Not Possible	Not Possible
Snapshot	Not Possible	Not Possible	Not Possible	Not Possible	Not Possible	Sometimes Possible	Not Possible	Possible
ANSI SQL SERIALIZABLE == Degree 3 == Repeatable Read Date, IBM, Tandem, ...	Not Possible	Not Possible	Not Possible	Not Possible	Not Possible	Not Possible	Not Possible	Not Possible

Hal Berenson, Philip A. Bernstein, Jim Gray, Jim Melton, Elizabeth J. O'Neil, Patrick E. O'Neil: A Critique of ANSI SQL Isolation Levels. SIGMOD Conference 1995: 1-10

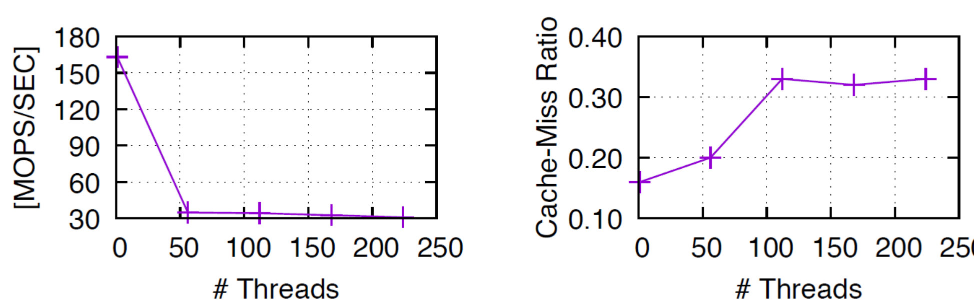
Cache-Line-Conflicts



Read=95%, Write=5%

Read=100%, Write=0%

1. SI's scheduling space is largest, but slow
2. Silo/MOCC are nice
3. Centralized ordering is bad

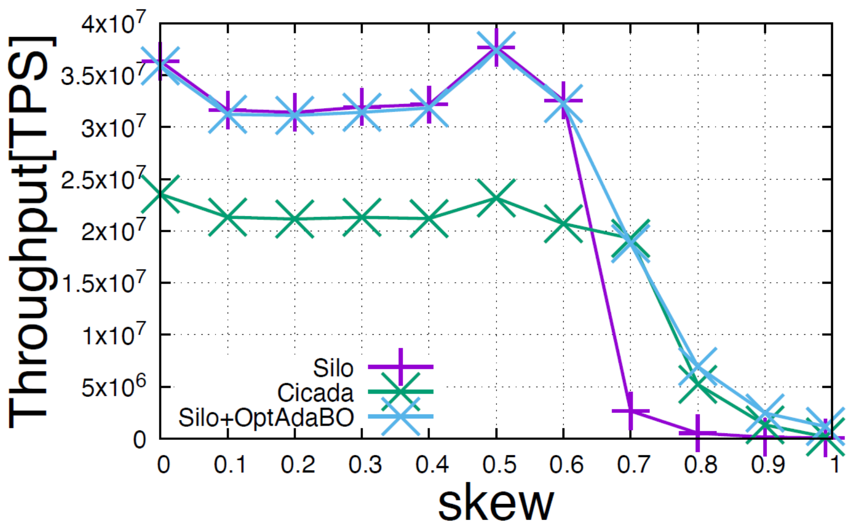


(a) Throughput (b) Cache miss ratio

Figure 6: Scalability of fetch_add

Modern CC Research

- Assumption
  - Short
  - Stored procedure (No NW)
- Target Workloads
  - Whole-sale (TPC-C)
  - Web (YCSB)



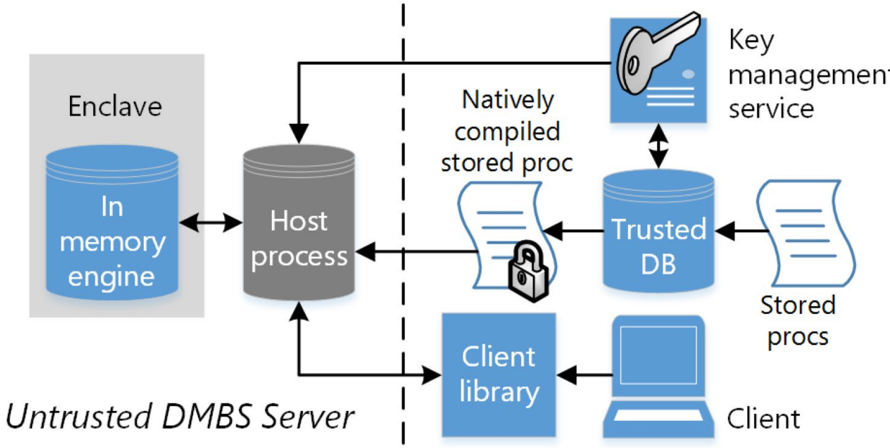
Method	Year	Venue	Features
Polyjuice	2021	OSDI	Learning
Bamboo	2021	SIGMOD	PCC, dirty read
Cicada	2017	SIGMOD	OCC MVCC
SSN	2017	VLDBJ	MVCC
MOCC	2016	VLDB	OCC/PCC
TicToc	2016	SIGMOD	OCC
Silo	2013	SOSP	OCC
SI	1995	SIGMOD	MVCC
2PL	1976	Comm. ACM	PCC

Is throttling enough?

Masumura, Hoshino, Kawashima, Accelerating Concurrency Control with Active Thread Adjustment, BigComp'22, Best Paper (2nd)

Future Directions

- Security (SGX 1TB) [1]
- Adaptivity [2]
  - Self-management/driving
- Distributed database [3]
  - Blockchain, ledger
- Embedded (CPS) [4,5]
  - Autonomous car/robot
- Batch (long) ???



[1] より抜粋

ROS
Robot Operating System

[1] EnclaveDB: A Secure Database using SGX, IEEE S&P 2018
[2] CMU 15-799 - Spring 2022, Special Topics: Self-Driving Database Management Systems
[3] Scalar DLT: https://scalar-labs.com/product/
[4] Superfast Subsystem Cooperation, A ROS Centric Database System, ROS World'21.
[5] Making ROS TF Transactional: Ogiwara, Kawashima, et al, ICCPS'22 (WiP, accepted)

Summary

- Transaction

Ordering

- Silo CC
- Concurrent index
- Silo recovery

Decentralization

Many topics

- Modern CC and future directions
  - Security, Finance, CPS, batch