Concurrency Control

P.J. McBrien

Imperial College London

Transactions: ACID properties

- **database management systems** (**DBMS**) implements indivisible tasks called **transactions**

Atomicity	all or nothing
Consistency	consistent before $\rightarrow$ consistent after
Isolation	independent of any other transaction
Durability	completed transaction are durable

Transactions: ACID properties

- **database management systems** (**DBMS**) implements indivisible tasks called **transactions**

Atomicity	all or nothing
Consistency	consistent before → consistent after
Isolation	independent of any other transaction
Durability	completed transaction are durable

```
BEGIN TRANSACTION
    UPDATE branch
    SET    cash=cash-10000.00
    WHERE  sortcode=56

    UPDATE branch
    SET    cash=cash+10000.00
    WHERE  sortcode=34
COMMIT TRANSACTION
```

Transactions: ACID properties

- **database management systems** (**DBMS**) implements indivisible tasks called **transactions**

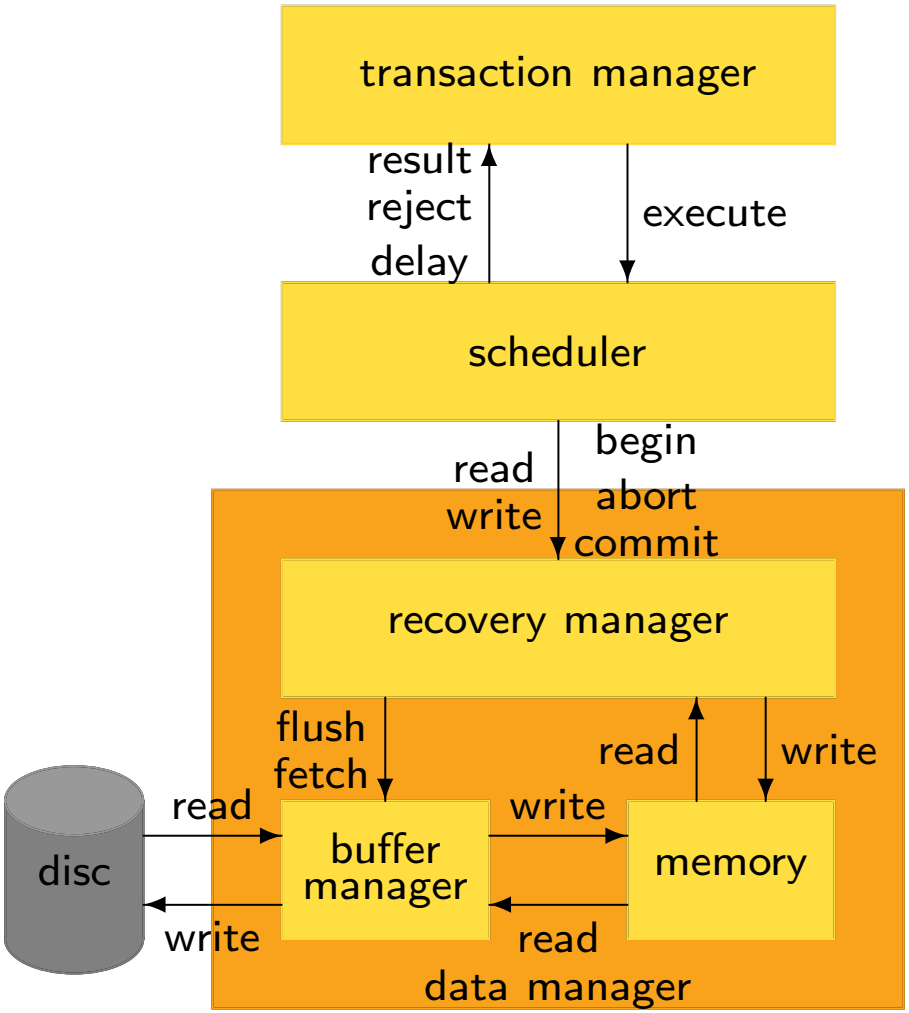
Atomicity	all or nothing
Consistency	consistent before → consistent after
Isolation	independent of any other transaction
Durability	completed transaction are durable

```
BEGIN TRANSACTION
    UPDATE branch
    SET    cash=cash-10000.00
    WHERE  sortcode=56

    UPDATE branch
    SET    cash=cash+10000.00
    WHERE  sortcode=34
COMMIT TRANSACTION
```

Note that if total cash is £137,246.12 before the transaction, then it will be the same after the transaction.

DBMS Architecture

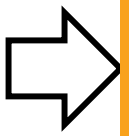


SQL Conversion to Histories

branch		
sortcode	bname	cash
56	'Wimbledon'	94340.45
34	'Goodge St'	8900.67
67	'Strand'	34005.00

```
BEGIN TRANSACTION T1
      UPDATE branch
      SET cash=cash-10000.00
      WHERE sortcode=56

      UPDATE branch
      SET cash=cash+10000.00
      WHERE sortcode=34
COMMIT TRANSACTION T1
```



$H_1 = r_1[b_{56}]$, cash=94340.45,
$w_1[b_{56}]$, cash=84340.45,
$r_1[b_{34}]$, cash=8900.67,
$w_1[b_{34}]$, cash=18900.67, c_1

A history of transaction T_n consists of

1. b_n begin transaction (only given if necessary for discussion)
2. Various read operations on objects $r_n[o_j]$ and write operations $r_n[o_j]$
3. Either c_n for the commitment of the transaction, or a_n for the abort of the transaction

SQL Conversion to Histories

branch		
sortcode	bname	cash
56	'Wimbledon'	84340.45
34	'Goodge St'	18900.67
67	'Strand'	34005.00

```
BEGIN TRANSACTION T2
      UPDATE branch
      SET cash=cash-2000.00
      WHERE sortcode=34

      UPDATE branch
      SET cash=cash+2000.00
      WHERE sortcode=67
COMMIT TRANSACTION T2
```

$H_2 = r_2[b_{34}]$, cash=18900.67,
$w_2[b_{34}]$, cash=16900.67,
$r_2[b_{67}]$, cash=34005.00,
$w_2[b_{67}]$, cash=36005.00, c_2

A history of transaction T_n consists of

1. b_n begin transaction (only given if necessary for discussion)
2. Various read operations on objects $r_n[o_j]$ and write operations $r_n[o_j]$
3. Either c_n for the commitment of the transaction, or a_n for the abort of the transaction

Concurrent Execution

Concurrent Execution of Transactions

- Interleaving of several transaction histories
- Order of operations within each history preserved

$$H_1 = r_1[b_{56}], w_1[b_{56}], r_1[b_{34}], w_1[b_{34}], c_1$$
$$H_2 = r_2[b_{34}], w_2[b_{34}], r_2[b_{67}], w_2[b_{67}], c_2$$

Concurrent Execution

Concurrent Execution of Transactions

- Interleaving of several transaction histories
- Order of operations within each history preserved

$H_1 = r_1[b_{56}], w_1[b_{56}], r_1[b_{34}], w_1[b_{34}], c_1$

$H_2 = r_2[b_{34}], w_2[b_{34}], r_2[b_{67}], w_2[b_{67}], c_2$

Some possible concurrent executions are

$H_x = r_2[b_{34}], r_1[b_{56}], w_1[b_{56}], r_1[b_{34}], w_1[b_{34}], c_1, w_2[b_{34}], r_2[b_{67}], w_2[b_{67}], c_2$

$H_y = r_2[b_{34}], w_2[b_{34}], r_1[b_{56}], w_1[b_{56}], r_1[b_{34}], w_1[b_{34}], r_2[b_{67}], w_2[b_{67}], c_2, c_1$

$H_z = r_2[b_{34}], w_2[b_{34}], r_1[b_{56}], w_1[b_{56}], r_1[b_{34}], w_1[b_{34}], c_1, r_2[b_{67}], w_2[b_{67}], c_2$

Which concurrent executions should be allowed?

Concurrency control → controlling interaction

serialisability

A concurrent execution of transactions should always has the same end result as some serial execution of those same transactions

recoverability

No transaction commits depending on data that has been produced by another transaction that has yet to commit

Quiz 1: Serialisability and Recoverability (1)

$H_x = r_2[b_{34}], r_1[b_{56}], w_1[b_{56}], r_1[b_{34}], w_1[b_{34}], c_1, w_2[b_{34}], r_2[b_{67}], w_2[b_{67}], c_2$

Is H_x

A

Not Serialisable, Not Recoverable

B

Not Serialisable, Recoverable

C

Serialisable, Not Recoverable

D

Serialisable, Recoverable

Quiz 2: Serialisability and Recoverability (2)

$H_y = r_2[b_{34}], w_2[b_{34}], r_1[b_{56}], w_1[b_{56}], r_1[b_{34}], w_1[b_{34}], r_2[b_{67}], w_2[b_{67}], c_2, c_1$

Is H_y

A

Not Serialisable, Not Recoverable

B

Not Serialisable, Recoverable

C

Serialisable, Not Recoverable

D

Serialisable, Recoverable

Quiz 3: Serialisability and Recoverability (3)

$H_z = r_2[b_{34}], w_2[b_{34}], r_1[b_{56}], w_1[b_{56}], r_1[b_{34}], w_1[b_{34}], c_1, r_2[b_{67}], w_2[b_{67}], c_2$

Is H_z

A

Not Serialisable, Not Recoverable

B

Not Serialisable, Recoverable

C

Serialisable, Not Recoverable

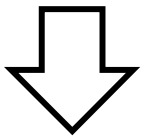
D

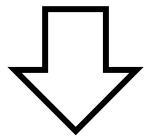
Serialisable, Recoverable

Anomaly 1: Lost update

BEGIN TRANSACTION T1
EXEC move_cash(56,34,10000.00)
COMMIT TRANSACTION T1

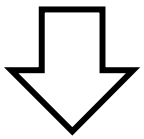
BEGIN TRANSACTION T2
EXEC move_cash(34,67,2000.00)
COMMIT TRANSACTION T2

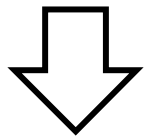




$r_1[b_{56}]$, $w_1[b_{56}]$, $r_1[b_{34}]$, $w_1[b_{34}]$, c_1

$r_2[b_{34}]$, $w_2[b_{34}]$, $r_2[b_{67}]$, $w_2[b_{67}]$, c_2





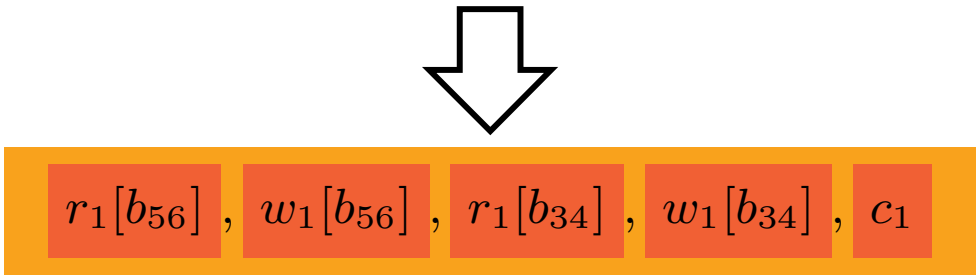
$r_1[b_{56}]$, cash=94340.45, $w_1[b_{56}]$, cash=84340.45, $r_1[b_{34}]$, cash=8900.67,
$r_2[b_{34}]$, cash=8900.67, $w_1[b_{34}]$, cash=18900.67, c_1, $w_2[b_{34}]$, cash=6900.42,
$r_2[b_{67}]$, cash=34005.00, $w_2[b_{67}]$, cash=36005.25, c_2

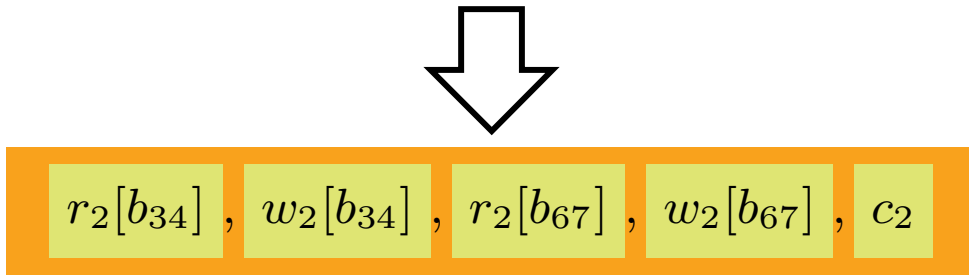
− serialisable + recoverable

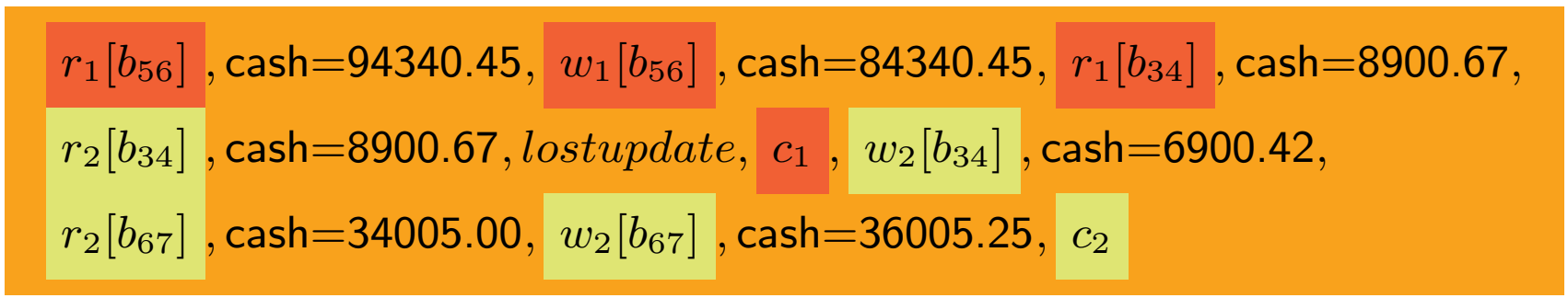
Anomaly 1: Lost update

BEGIN TRANSACTION T1
EXEC move_cash(56,34,10000.00)
COMMIT TRANSACTION T1

BEGIN TRANSACTION T2
EXEC move_cash(34,67,2000.00)
COMMIT TRANSACTION T2





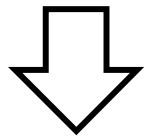


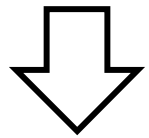
− serialisable + recoverable

Anomaly 2: Inconsistent analysis

BEGIN TRANSACTION T1
EXEC move_cash(56,34,10000.00)
COMMIT TRANSACTION T1

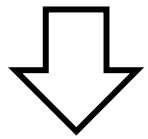
BEGIN TRANSACTION T4
SELECT SUM(cash) FROM branch
COMMIT TRANSACTION T4

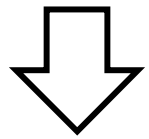




$r_1[b_{56}]$, $w_1[b_{56}]$, $r_1[b_{34}]$, $w_1[b_{34}]$, c_1

$H_4 = r_4[b_{56}]$, $r_4[b_{34}]$, $r_4[b_{67}]$, c_4





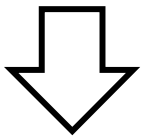
$r_1[b_{56}]$, cash=94340.45, $w_1[b_{56}]$, cash=84340.45, $r_4[b_{56}]$, cash=84340.45,
$r_4[b_{34}]$, cash=8900.67, $r_4[b_{67}]$, cash=34005.00, $r_1[b_{34}]$, cash=8900.67,
$w_1[b_{34}]$, cash=18900.67, c_1, c_4

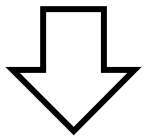
− serialisable + recoverable

Anomaly 3: Dirty Reads

BEGIN TRANSACTION T1
EXEC move_cash(56,34,10000.00)
COMMIT TRANSACTION T1

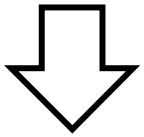
BEGIN TRANSACTION T2
EXEC move_cash(34,67,2000.00)
COMMIT TRANSACTION T2

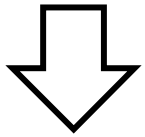




$r_1[b_{56}]$, $w_1[b_{56}]$, $r_1[b_{34}]$, $w_1[b_{34}]$, c_1

$r_2[b_{34}]$, $w_2[b_{34}]$, $r_2[b_{67}]$, $w_2[b_{67}]$, c_2





$r_1[b_{56}]$, cash=94340.45, $w_1[b_{56}]$, cash=84340.45, $r_2[b_{34}]$, cash=8900.67,
$w_2[b_{34}]$, cash=6900.42, $r_1[b_{34}]$, cash=6900.67, $w_1[b_{34}]$, cash=16900.67, c_1 ,
$r_2[b_{67}]$, cash=34005.00, $w_2[b_{67}]$, cash=36005.25, a_2

\+ serialisable

− recoverable

Quiz 4: Anomalies (1)

$$H_x = r_2[b_{34}], r_1[b_{56}], w_1[b_{56}], r_1[b_{34}], w_1[b_{34}], c_1, w_2[b_{34}], r_2[b_{67}], w_2[b_{67}], c_2$$

Which anomaly does H_x suffer?

A

None

B

Lost Update

C

Inconsistent Analysis

D

Dirty Read

Quiz 5: Anomalies (2)

$$H_z = r_2[b_{34}], w_2[b_{34}], r_1[b_{56}], w_1[b_{56}], r_1[b_{34}], w_1[b_{34}], c_1, r_2[b_{67}], w_2[b_{67}], c_2$$

Which anomaly does H_z suffer?

A
None

B
Lost Update

C
Inconsistent Analysis

D
Dirty Read

Worksheet: Anomalies

`rental_charge`

$H_1 = r_1[d_{1000}], w_1[d_{1000}], r_1[d_{1001}], w_1[d_{1001}], r_1[d_{1002}], w_1[d_{1002}]$

`transfer_charge`

$H_2 = r_2[d_{1000}], w_2[d_{1000}], r_2[d_{1002}], w_2[d_{1002}]$

`total_charge`

$H_3 = r_3[d_{1000}], r_3[d_{1001}], r_3[d_{1002}]$

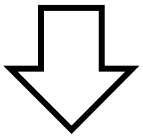
Account Table

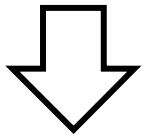
account				
no	type	cname	rate	sortcode
100	'current'	'McBrien, P.'	NULL	67
101	'deposit'	'McBrien, P.'	5.25	67
103	'current'	'Boyd, M.'	NULL	34
107	'current'	'Poulovassilis, A.'	NULL	56
119	'deposit'	'Poulovassilis, A.'	5.50	56
125	'current'	'Bailey, J.'	NULL	56

Anomaly 4: Dirty Writes

```
BEGIN TRANSACTION T5
    UPDATE account
    SET rate=5.5
    WHERE type='deposit'
COMMIT TRANSACTION T5
```

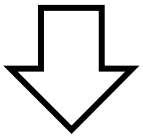
```
BEGIN TRANSACTION T6
    UPDATE account
    SET rate=6.0
    WHERE type='deposit'
COMMIT TRANSACTION T6
```

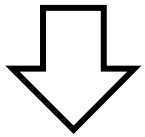




$H_5 = w_5[a_{101}]$, rate=5.5, $w_5[a_{119}]$, rate=5.5, c_5

$H_6 = w_6[a_{101}]$, rate=6.0, $w_6[a_{119}]$, rate=6.0, c_6





$w_6[a_{101}]$, rate=6.0, $w_5[a_{101}]$, rate=5.5, $w_5[a_{119}]$, rate=5.5, $w_6[a_{119}]$, rate=6.0, c_5, c_6

− serialisable

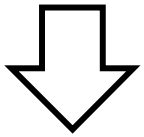
+ recoverable

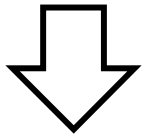
Anomaly 5: Phantom reads

```
BEGIN TRANSACTION T7
   UPDATE account
   SET    rate=rate+0.25
   WHERE  type='deposit'
   AND    rate<5.5

   UPDATE account
   SET    rate=rate+0.25
   WHERE  type='deposit'
COMMIT TRANSACTION T7
```

```
BEGIN TRANSACTION T8
      INSERT INTO account
      VALUES (126,'deposit','Boyd,M.',5.25,34)
COMMIT TRANSACTION T8
```





$r_7[a_{101}]$, rate=5.25, $w_7[a_{101}]$, rate=5.50, $r_7[a_{119}]$, rate=5.50,
$ins_8[a_{126}]$, rate=5.25, c_8, $r_7[a_{101}]$, rate=5.50, $w_7[a_{101}]$, rate=5.75,
$r_7[a_{119}]$, rate=5.50, $w_7[a_{119}]$, rate=5.75, $r_7[a_{126}]$, rate=5.25,
$w_7[a_{126}]$, rate=5.50, c_7

− serialisable + recoverable

Movement and Account Tables

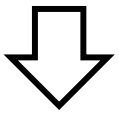
movement			
mid	no	amount	tdate
1000	100	2300.00	5/1/1999
1001	101	4000.00	5/1/1999
1002	100	-223.45	8/1/1999
1004	107	-100.00	11/1/1999
1005	103	145.50	12/1/1999
1006	100	10.23	15/1/1999
1007	107	345.56	15/1/1999
1008	101	1230.00	15/1/1999
1009	119	5600.00	18/1/1999

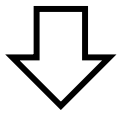
account				
no	type	cname	rate	sortcode
100	'current'	'McBrien, P.'	NULL	67
101	'deposit'	'McBrien, P.'	5.25	67
103	'current'	'Boyd, M.'	NULL	34
107	'current'	'Poulovassilis, A.'	NULL	56
119	'deposit'	'Poulovassilis, A.'	5.50	56
125	'current'	'Bailey, J.'	NULL	56

Anomaly 6: Write skew

```
BEGIN TRANSACTION T9
  INSERT INTO movement
  SELECT  1011,w_account.no,-7000.00,'21-jan-1999'
  FROM    movement NATURAL JOIN account
          JOIN account w_account
          ON account.cname=w_account.cname
  WHERE   w_account.no=101
  GROUP BY w_account.no
  HAVING SUM(amount)>=7000.00;
COMMIT TRANSACTION T9
```

```
BEGIN TRANSACTION T10
  INSERT INTO movement
  SELECT  1012,w_account.no,-2200.00,'21-jan-1999'
  FROM    movement NATURAL JOIN account
          JOIN account w_account
          ON account.cname=w_account.cname
  WHERE   w_account.no=100
  GROUP BY w_account.no
  HAVING SUM(amount)>=2200.00;
COMMIT TRANSACTION T10
```



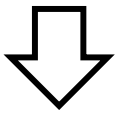


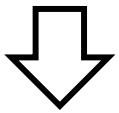
$$r_9[a_{101}], r_9[a_{100}], r_9[m_{1000}], r_9[m_{1001}], r_9[m_{1002}], r_9[m_{1006}], r_9[m_{1008}],$$
$$r_{10}[a_{100}], r_{10}[a_{101}], r_{10}[m_{1000}], r_{10}[m_{1001}], r_{10}[m_{1002}], r_{10}[m_{1006}], r_{10}[m_{1008}],$$
$$ins_9[m_{1011}], ins_{10}[m_{1012}], c_9, c_{10}$$

Anomaly 6: Write skew

```
BEGIN TRANSACTION T9
  INSERT INTO movement
  SELECT  1011,w_account.no,-7000.00,'21-jan-1999'
  FROM    movement NATURAL JOIN account
          JOIN account w_account
          ON account.cname=w_account.cname
  WHERE   w_account.no=101
  GROUP BY w_account.no
  HAVING SUM(amount)>=7000.00;
COMMIT TRANSACTION T9
```

```
BEGIN TRANSACTION T10
  INSERT INTO movement
  SELECT  1012,w_account.no,-2200.00,'21-jan-1999'
  FROM    movement NATURAL JOIN account
          JOIN account w_account
          ON account.cname=w_account.cname
  WHERE   w_account.no=100
  GROUP BY w_account.no
  HAVING SUM(amount)>=2200.00;
COMMIT TRANSACTION T10
```





$$r_9[Match(101)], r_{10}[Match(100)], ins_9[m_{1011}], ins_{10}[m_{1012}], c_9, c_{10}$$

note the conflicts

$$r_9[Match(101)] \rightarrow ins_{10}[m_{1012}]$$

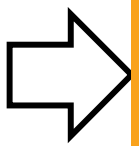
$$r_{10}[Match(100)] \rightarrow ins_9[m_{1011}]$$

Serialisable Transaction Execution

- Solve anomalies $\rightarrow$ $H \equiv$ serial execution
- Only interested in the **committed projection**

$H_c =$

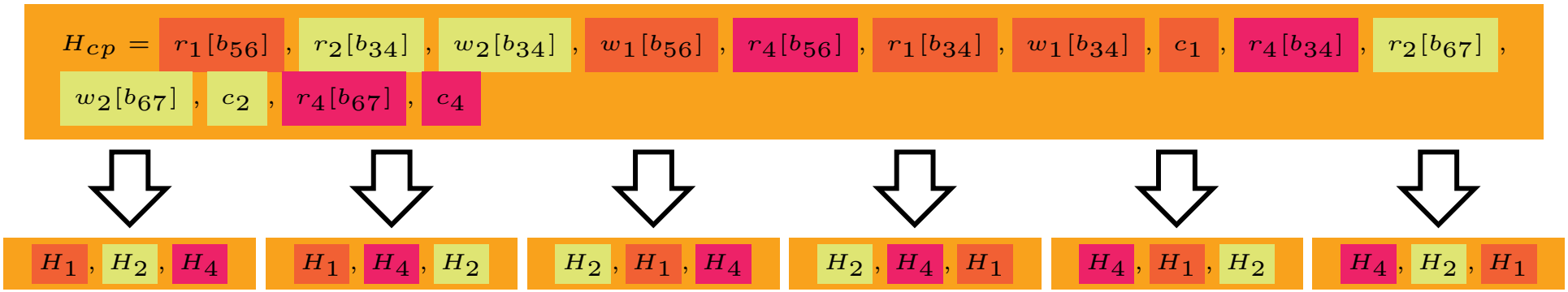
$r_1[b_{56}]$, $r_2[b_{34}]$, $w_2[b_{34}]$,
$r_3[m_{1000}]$, $r_3[m_{1001}]$, $r_3[m_{1002}]$,
$w_1[b_{56}]$, $r_4[b_{56}]$,
$r_3[m_{1003}]$, $r_3[m_{1004}]$, $r_3[m_{1005}]$,
$r_1[b_{34}]$, a_3 , $w_1[b_{34}]$, c_1 , $r_4[b_{34}]$,
$r_2[b_{67}]$, $w_2[b_{67}]$, c_2 , $r_4[b_{67}]$, c_4



$C(H_c) =$

$r_1[b_{56}]$, $r_2[b_{34}]$, $w_2[b_{34}]$,
$w_1[b_{56}]$, $r_4[b_{56}]$,
$r_1[b_{34}]$, $w_1[b_{34}]$, c_1 , $r_4[b_{34}]$,
$r_2[b_{67}]$, $w_2[b_{67}]$, c_2 , $r_4[b_{67}]$, c_4

Possible Serial Equivalents



- how to determine that histories are equivalent?
- how to check this during execution?

View Equivalence & View Serialisable

Want database to end up in the same final state as if serial execution had occurred

View Equivalence & View Serialisable

Want database to end up in the same final state as if serial execution had occurred

View Equivalence

Two histories H_i, H_j will leave the database in the same final state if, at each commit

If the above are met, we say the histories are **view equivalent** $H_i \equiv_{VE} H_j$

View Equivalence & View Serialisable

Want database to end up in the same final state as if serial execution had occurred

View Equivalence

Two histories H_i, H_j will leave the database in the same final state if, at each commit

1. H_i, H_j have same set of operations

If the above are met, we say the histories are **view equivalent** $H_i \equiv_{VE} H_j$

View Equivalence & View Serialisable

Want database to end up in the same final state as if serial execution had occurred

View Equivalence

Two histories H_i, H_j will leave the database in the same final state if, at each commit

1. H_i, H_j have same set of operations
2. $w_t[o] \rightarrow r_s[o]$ in both H_i, H_j or in neither

If the above are met, we say the histories are **view equivalent** $H_i \equiv_{VE} H_j$

View Equivalence & View Serialisable

Want database to end up in the same final state as if serial execution had occurred

View Equivalence

Two histories H_i, H_j will leave the database in the same final state if, at each commit

1. H_i, H_j have same set of operations
2. $w_t[o] \rightarrow r_s[o]$ in both H_i, H_j or in neither
3. Last committed write on o is $w_t[o]$ in both H_i, H_j or in neither

If the above are met, we say the histories are **view equivalent** $H_i \equiv_{VE} H_j$

View Equivalence & View Serialisable

Want database to end up in the same final state as if serial execution had occurred

View Equivalence

Two histories H_i, H_j will leave the database in the same final state if, at each commit

1. H_i, H_j have same set of operations
2. $w_t[o] \rightarrow r_s[o]$ in both H_i, H_j or in neither
3. Last committed write on o is $w_t[o]$ in both H_i, H_j or in neither

If the above are met, we say the histories are **view equivalent** $H_i \equiv_{VE} H_j$

View Serialisable Histories

a history H is **view serialisable** (**VSR**) if $C(H) \equiv_{VE}$ a serial history

Testing for View Equivalence

$$H_{cp} = r_1[b_{56}], r_2[b_{34}], w_2[b_{34}], w_1[b_{56}], r_4[b_{56}], r_1[b_{34}], w_1[b_{34}], c_1, r_4[b_{34}], r_2[b_{67}], w_2[b_{67}], c_2, r_4[b_{67}], c_4$$

$$\equiv$$

$$H_2, H_1, H_4 = r_2[b_{34}], w_2[b_{34}], r_2[b_{67}], w_2[b_{67}], c_2, r_1[b_{56}], w_1[b_{56}], r_1[b_{34}], w_1[b_{34}], c_1, r_4[b_{56}], r_4[b_{34}], r_4[b_{67}], c_4$$

1. H_{cp} and H_2, H_1, H_4 same set of operations
2. reads are $w_2[b_{34}] \rightarrow r_1[b_{34}]$, $w_2[b_{67}] \rightarrow r_4[b_{67}]$, $w_1[b_{34}] \rightarrow r_4[b_{34}]$, $w_1[b_{56}] \rightarrow r_4[b_{56}]$
3. last committed writes are $w_1[b_{34}]$, $w_1[b_{56}]$, and $w_2[b_{67}]$

$$H_2, H_1, H_4 \equiv_{VE} H_{cp} \quad \rightarrow \quad H_{cp} \in VSR$$

Conflicts: Potential For Problems

Difficult to test for VSR at runtime: need something easier to test at runtime

conflict

A **conflict** occurs when there is a interaction between two transactions

- $r_x[o]$ and $w_y[o]$ are in H where $x \neq y$
  or
- $w_x[o]$ and $w_y[o]$ are in H where $x \neq y$

conflicts

$H_x = r_2[b_{34}], r_1[b_{56}], w_1[b_{56}], r_1[b_{34}], w_1[b_{34}], c_1, w_2[b_{34}], r_2[b_{67}], w_2[b_{67}], c_2$

$H_y = r_2[b_{34}], w_2[b_{34}], r_1[b_{56}], w_1[b_{56}], r_1[b_{34}], w_1[b_{34}], r_2[b_{67}], w_2[b_{67}], c_2, c_1$

$H_z = r_2[b_{34}], w_2[b_{34}], r_1[b_{56}], w_1[b_{56}], r_1[b_{34}], w_1[b_{34}], c_1, r_2[b_{67}], w_2[b_{67}], c_2$

Conflicts

- $w_2[b_{34}] \rightarrow r_1[b_{34}]$ T1 reads from T2 in H_y, H_z
- $w_1[b_{34}] \rightarrow w_2[b_{34}]$ T2 writes over T1 in H_x
- $r_2[b_{34}] \rightarrow w_1[b_{34}]$ T1 writes after T2 reads in H_x

Quiz 6: Conflicts

$H_w =$

$r_2[a_{100}]$, $w_2[a_{100}]$, $r_2[a_{107}]$, $r_1[a_{119}]$, $w_1[a_{119}]$, $r_1[a_{107}]$, $w_1[a_{107}]$, c_1, $w_2[a_{107}]$, c_2

Which of the following is not a conflict in H_w?

A

$r_2[a_{107}] \rightarrow r_1[a_{107}]$

B

$r_2[a_{107}] \rightarrow w_1[a_{107}]$

C

$r_1[a_{107}] \rightarrow w_2[a_{107}]$

D

$w_1[a_{107}] \rightarrow w_2[a_{107}]$

Conflict Equivalence and Conflict Serialisable

Conflict Equivalence

Two histories H_i and H_j are **conflict equivalent** if:

1. Contain the same set of operations
2. Order conflicts (of non-aborted transactions) in the same way.

Conflict Serialisable

a history H is **conflict serialisable** (**CSR**) if $C(H) \equiv_{CE}$ a serial history

Failure to be conflict serialisable

$H_x = r_2[b_{34}], r_1[b_{56}], w_1[b_{56}], r_1[b_{34}], w_1[b_{34}], c_1, w_2[b_{34}], r_2[b_{67}], w_2[b_{67}], c_2$

Contains conflicts $r_2[b_{34}] \rightarrow w_1[b_{34}]$ and $w_1[b_{34}] \rightarrow w_2[b_{34}]$ and so is not conflict equivalence to H_1, H_2 nor H_2, H_1, and hence is not conflict serialisable.

$CSR \subset VSR$

Testing for Conflict Equivalence

$$H_{cp} = r_1[b_{56}], r_2[b_{34}], w_2[b_{34}], w_1[b_{56}], r_4[b_{56}], r_1[b_{34}], w_1[b_{34}], c_1, r_4[b_{34}], r_2[b_{67}], w_2[b_{67}], c_2, r_4[b_{67}], c_4$$

$$\equiv$$

$$H_2, H_1, H_4 = r_2[b_{34}], w_2[b_{34}], r_2[b_{67}], w_2[b_{67}], c_2, r_1[b_{56}], w_1[b_{56}], r_1[b_{34}], w_1[b_{34}], c_1, r_4[b_{56}], r_4[b_{34}], r_4[b_{67}], c_4$$

1. H_{cp} and H_2, H_1, H_4 contain the same set of operations
2. conflicting pairs are
   $w_2[b_{34}] \rightarrow r_1[b_{34}]$, $w_2[b_{67}] \rightarrow r_4[b_{67}]$,
   $w_1[b_{34}] \rightarrow r_4[b_{34}]$, $w_1[b_{56}] \rightarrow r_4[b_{56}]$
3. $H_2, H_1, H_4 \equiv_{CE} H_{cp} \quad \rightarrow \quad H_{cp} \in CSR$

Serialisation Graph

Serialisation Graph

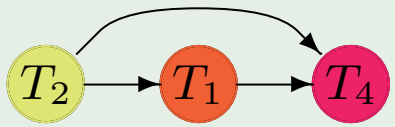
A **serialisation graph** $SG(H)$ contains a node for each transaction in H, and an edge $T_i \rightarrow T_j$ if there is some object o for which a conflict $rw_i[o] \rightarrow rw_j[o]$ exists in H. If $SG(H)$ is acyclic, then H is conflict serialisable.

Demonstrating a History is CSR

Given H_{cp}= $r_1[b_{56}]$, $r_2[b_{34}]$, $w_2[b_{34}]$, $w_1[b_{56}]$, $r_4[b_{56}]$, $r_1[b_{34}]$, $w_1[b_{34}]$, c_1, $r_4[b_{34}]$, $r_2[b_{67}]$, $w_2[b_{67}]$, c_2, $r_4[b_{67}]$, c_4

Conflicts are $w_2[b_{34}] \rightarrow r_1[b_{34}]$, $w_2[b_{67}] \rightarrow r_4[b_{67}]$, $w_1[b_{34}] \rightarrow r_4[b_{34}]$, $w_1[b_{56}] \rightarrow r_4[b_{56}]$

Then serialisation graph is



$SG(H_{cp})$ is acyclic, therefore H_{cp} is CSR

Worksheet: Serialisability

$H_1 = r_1[o_1], w_1[o_1], w_1[o_2], w_1[o_3], c_1$

$H_2 = r_2[o_2], w_2[o_2], w_2[o_1], c_2$

$H_3 = r_3[o_1], w_3[o_1], w_3[o_2], c_3$

$H = r_1[o_1], w_1[o_1], r_2[o_2], w_2[o_2], w_2[o_1], c_2, w_1[o_2], r_3[o_1], w_3[o_1], w_3[o_2], c_3, w_1[o_3], c_1$

Recoverability

- Serialisability necessary for isolation and consistency of committed transactions
- Recoverability necessary for isolation and consistency when there are also aborted transactions

Recoverable execution

A **recoverable** (**RC**) history H has no transaction committing before another transaction from which it read

Execution avoiding cascading aborts

A history which **avoids cascading aborts** (**ACA**) does not read from a non-committed transaction

Strict execution

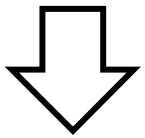
A **strict** (**ST**) history does not read from a non-committed transaction nor write over a non-committed transaction

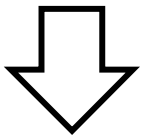
$ST \subset ACA \subset RC$

Non-recoverable executions

```
BEGIN TRANSACTION T1
      UPDATE branch
      SET cash=cash-10000.00
      WHERE sortcode=56
      UPDATE branch
      SET cash=cash+10000.00
      WHERE sortcode=34
COMMIT TRANSACTION T1
```

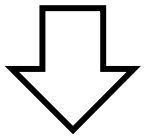
```
BEGIN TRANSACTION T4
      SELECT SUM(cash) FROM branch
COMMIT TRANSACTION T4
```

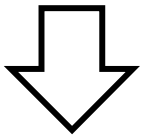




$H_1 = r_1[b_{56}], w_1[b_{56}], a_1$

$H_4 = r_4[b_{56}], r_4[b_{34}], r_4[b_{67}], c_4$





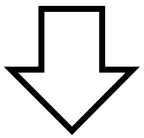
$H_c = r_1[b_{56}]$, cash=94340.45, $w_1[b_{56}]$, cash=84340.45, $r_4[b_{56}]$, cash=84340.45, $r_4[b_{34}]$, cash=8900.67, $r_4[b_{67}]$, cash=34005.00, c_4, a_1

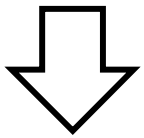
$H_c \notin RC$

Cascading Aborts

```
BEGIN TRANSACTION T1
        UPDATE branch
        SET cash=cash-10000.00
        WHERE sortcode=56
        UPDATE branch
        SET cash=cash+10000.00
        WHERE sortcode=34
COMMIT TRANSACTION T1
```

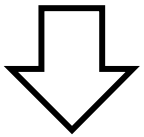
```
BEGIN TRANSACTION T4
        SELECT SUM(cash) FROM branch
COMMIT TRANSACTION T4
```

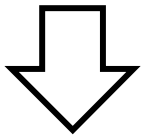




$H_1 = r_1[b_{56}], w_1[b_{56}], a_1$

$H_4 = r_4[b_{56}], r_4[b_{34}], r_4[b_{67}], c_4$



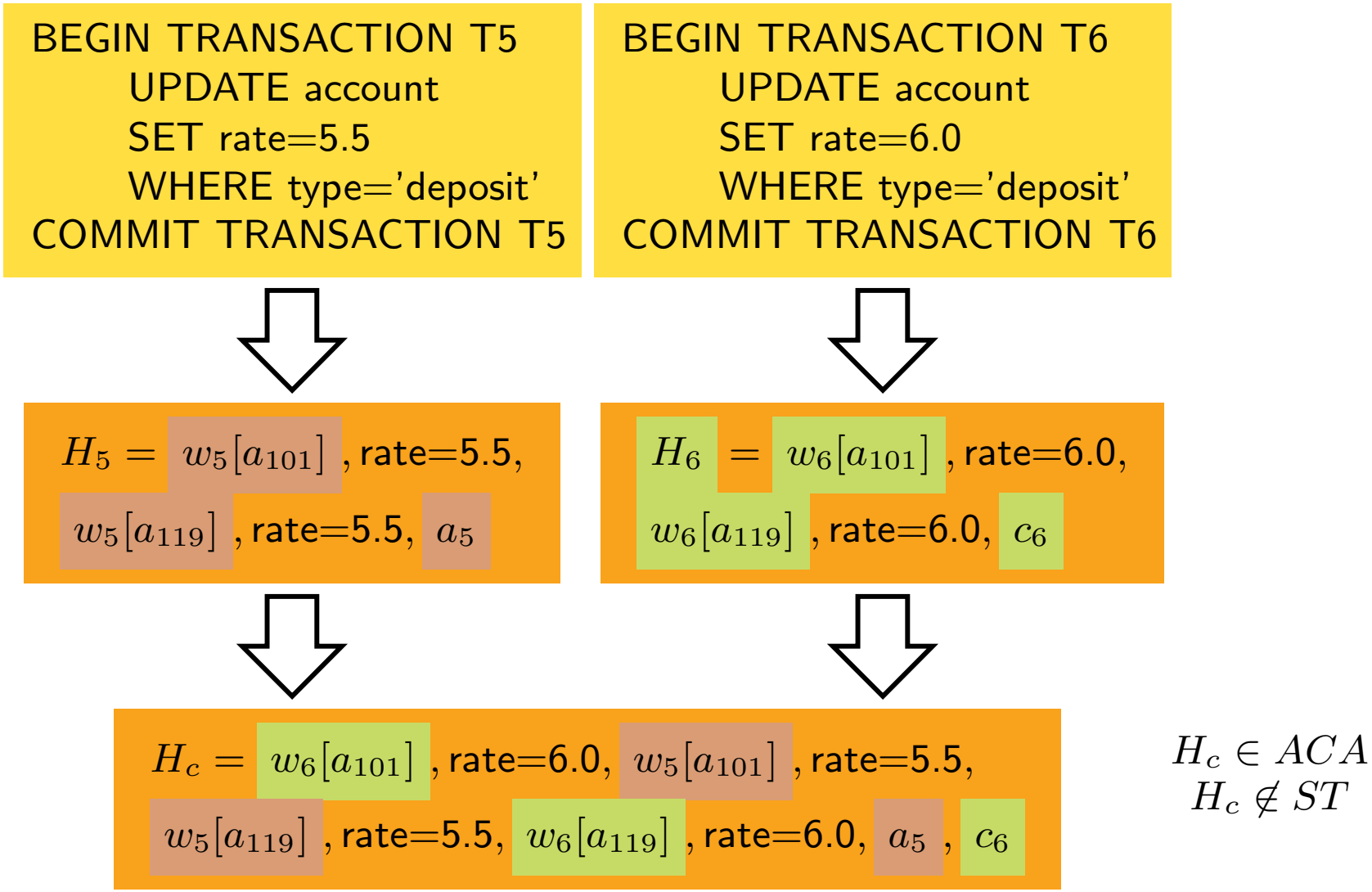


$H_c = r_1[b_{56}]$, cash=94340.45, $w_1[b_{56}]$, cash=84340.45, $r_4[b_{56}]$, cash=84340.45, $r_4[b_{34}]$, cash=8900.67, $r_4[b_{67}]$, cash=34005.00, a_1, a_4

$H_c \in RC$

$H_c \notin ACA$

Strict Execution



$H_c \in ACA$
$H_c \notin ST$

Quiz 7: Recoverability

$H_z = r_2[b_{34}], w_2[b_{34}], r_1[b_{56}], w_1[b_{56}], r_1[b_{34}], w_1[b_{34}], c_1, r_2[b_{67}], w_2[b_{67}], c_2$

Which describes the recoverability of H_z?

A
Non-recoverable

B
Recoverable

C
Avoids Cascading Aborts

D
Strict

Worksheet: Recoverability

$H_w = r_2[o_1], r_2[o_2], w_2[o_2], r_1[o_2], w_2[o_1], r_2[o_3], c_2, c_1$

$H_x = r_2[o_1], r_2[o_2], w_2[o_1], w_2[o_2], w_1[o_1], w_1[o_2], c_1, r_2[o_3], c_2$

$H_y = r_2[o_1], r_2[o_2], w_2[o_2], r_1[o_2], w_2[o_1], c_1, r_2[o_3], c_2$

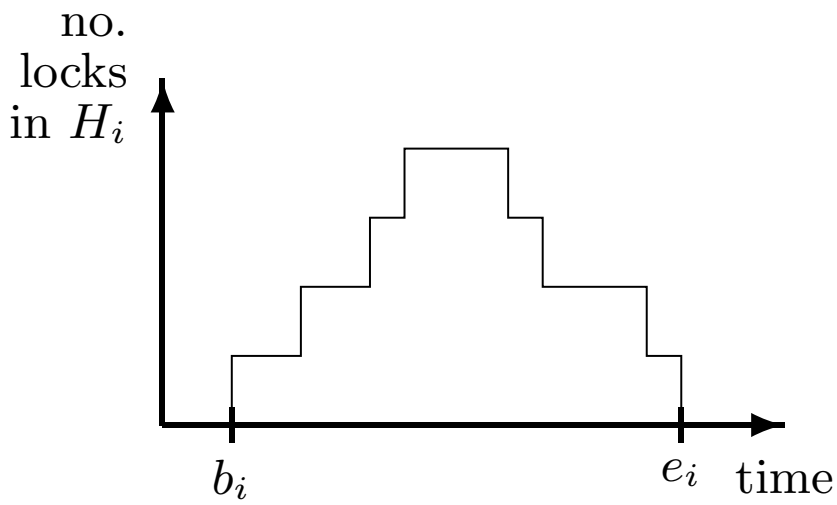
$H_z = r_2[o_1], w_1[o_1], r_2[o_2], w_2[o_2], r_2[o_3], c_2, r_1[o_2], w_1[o_2], w_1[o_3], c_1$

Maintaining Serialisability and Recoverability

- **two-phase locking (2PL)**
  - conflict based
  - uses **locks** to prevent problems
  - common technique
- **time-stamping**
  - add a timestamp to each object
  - write sets timestamp to that of transaction
  - may only read or write objects with earlier timestamp
  - abort when object has new timestamp
  - common technique
- **optimistic concurrency control**
  - do nothing until commit
  - at commit, inspect history for problems
  - good if few conflicts

The 2PL Protocol

1. read locks $rl[o], \ldots, r[o], \ldots, ru[o]$
2. write locks $wl[o], \ldots, w[o], \ldots, wu[o]$
3. Two phases
   i. **growing phase**
   ii. **shrinking phase**
4. refuse $rl_i[o]$ if $wl_j[o]$ already held
   refuse $wl_i[o]$ if $rl_j[o]$ or $wl_j[o]$ already held
5. $rl_i[o]$ or $wl_i[o]$ refused $\rightarrow$ delay T_i



Quiz 8: Two Phase Locking (2PL)

Which history is not valid in 2PL?

A

$rl_1[a_{107}]$, $r_1[a_{107}]$, $wl_1[a_{107}]$, $w_1[a_{107}]$, $wu_1[a_{107}]$, $ru_1[a_{107}]$

B

$wl_1[a_{107}]$, $wl_1[a_{100}]$, $r_1[a_{107}]$, $w_1[a_{107}]$, $r_1[a_{100}]$, $w_1[a_{100}]$, $wu_1[a_{100}]$, $wu_1[a_{107}]$

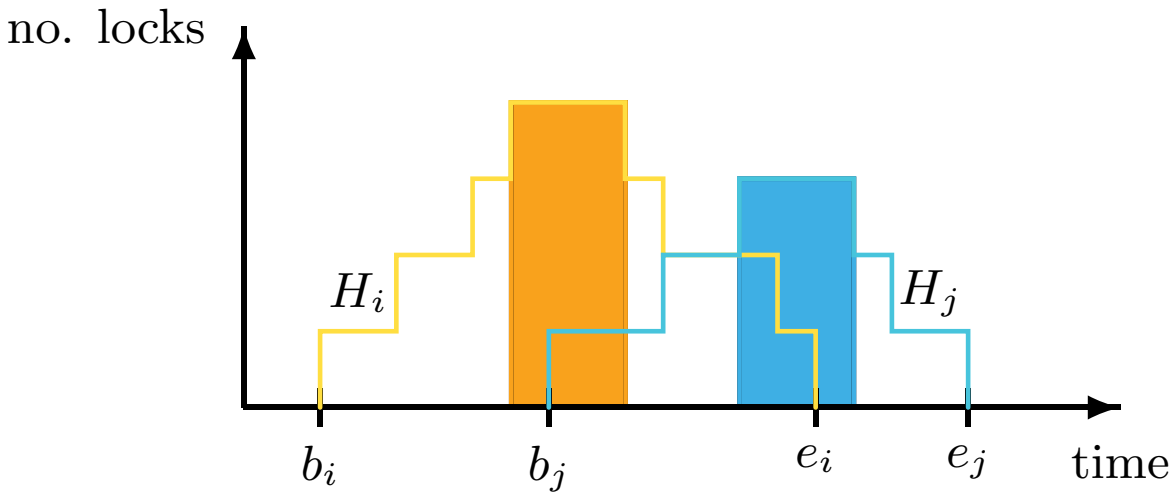
C

$wl_1[a_{107}]$, $r_1[a_{107}]$, $w_1[a_{107}]$, $wu_1[a_{107}]$, $wl_1[a_{100}]$, $r_1[a_{100}]$, $w_1[a_{100}]$, $wu_1[a_{100}]$

D

$wl_1[a_{107}]$, $r_1[a_{107}]$, $w_1[a_{107}]$, $wl_1[a_{100}]$, $r_1[a_{100}]$, $wu_1[a_{107}]$, $w_1[a_{100}]$, $wu_1[a_{100}]$

Why does 2PL Work?



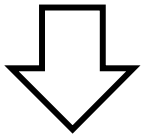
- two-phase rule $\rightarrow$ maximum lock period
- can re-time history so all operations take place during maximum lock period
- CSR since *all* conflicts prevented during maximum lock period

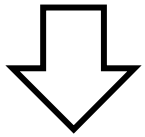
Anomaly 5: Phantom reads

```
BEGIN TRANSACTION T7
   UPDATE account
   SET    rate=rate+0.25
   WHERE  type='deposit'
   AND    rate<5.5

   UPDATE account
   SET    rate=rate+0.25
   WHERE  type='deposit'
COMMIT TRANSACTION T7
```

```
BEGIN TRANSACTION T8
      INSERT INTO account
      VALUES (126,'deposit','Boyd,M.',5.25,34)
COMMIT TRANSACTION T8
```

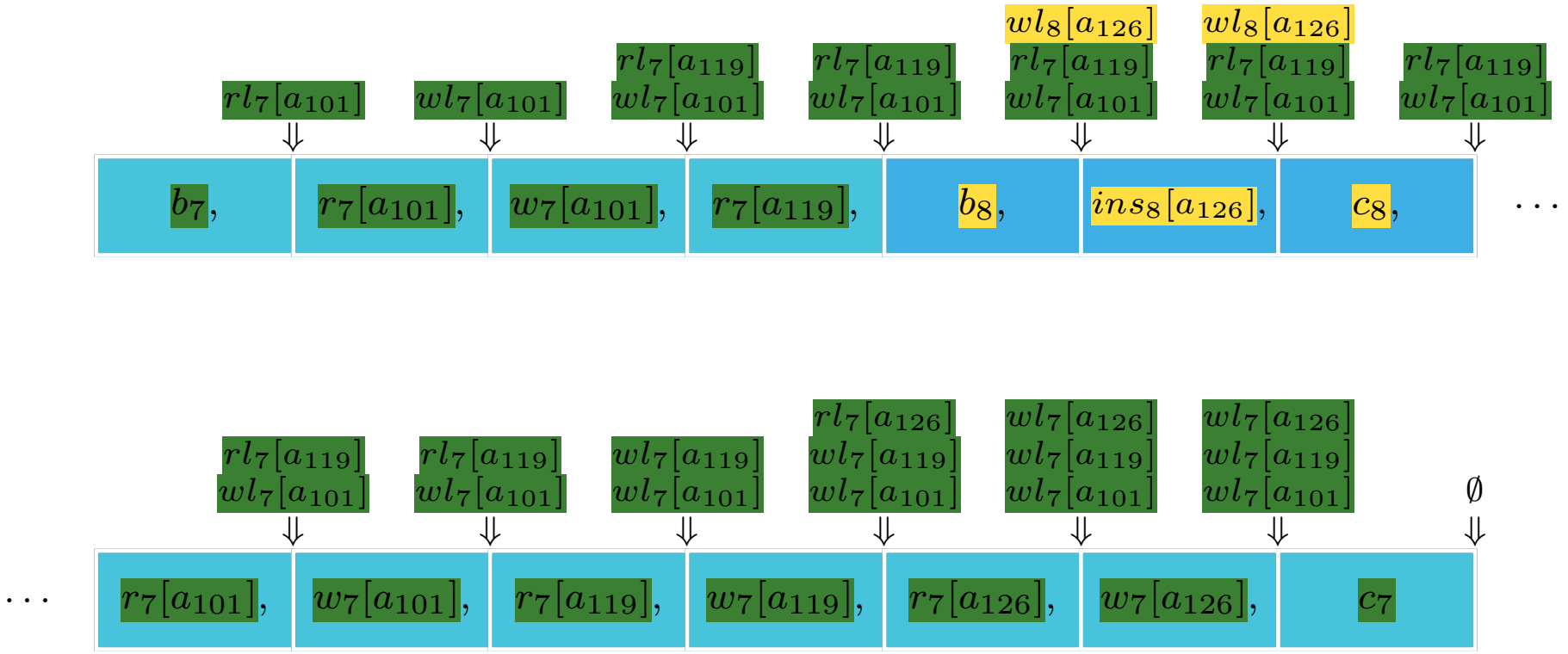




$r_7[a_{101}]$, rate=5.25, $w_7[a_{101}]$, rate=5.50, $r_7[a_{119}]$, rate=5.50,
$ins_8[a_{126}]$, rate=5.25, c_8, $r_7[a_{101}]$, rate=5.50, $w_7[a_{101}]$, rate=5.75,
$r_7[a_{119}]$, rate=5.50, $w_7[a_{119}]$, rate=5.75, $r_7[a_{126}]$, rate=5.25,
$w_7[a_{126}]$, rate=5.50, c_7

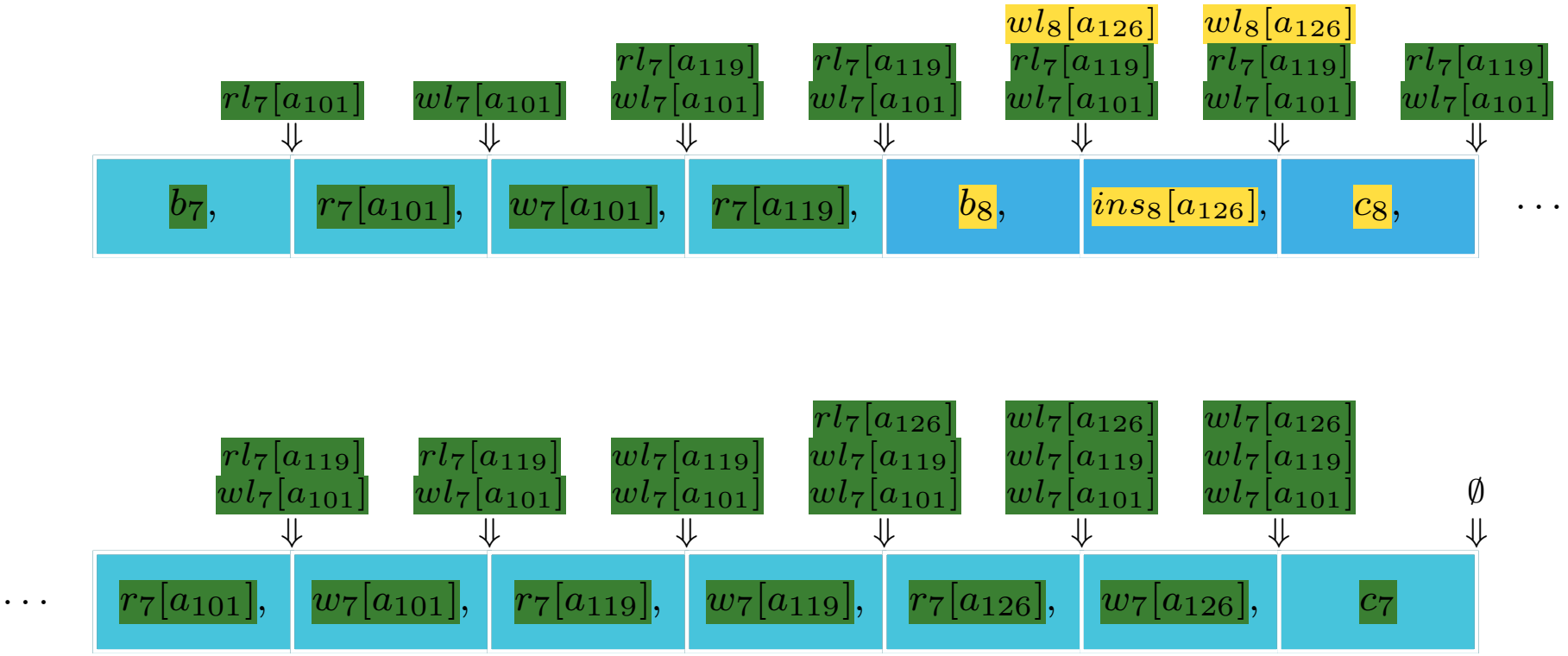
− serialisable + recoverable

Naive 2PL of Insert



- What is being locked?
  - objects a_{101} and a_{119}?

Naive 2PL of Insert



- What is being locked?
  - objects a_{101} and a_{119}?
  - predicate type='deposit' AND rate<5.5

Solution 1: EOF Marker

- Problem with inserts and phantom reads is due to changing size of file
- Introduce **end of file** (**EOF**) marker
- Read lock EOF when performing a 'scan' of the file

  H_7 uses $rl_7[b_{EOF}]$
- Insert lock when inserting new records

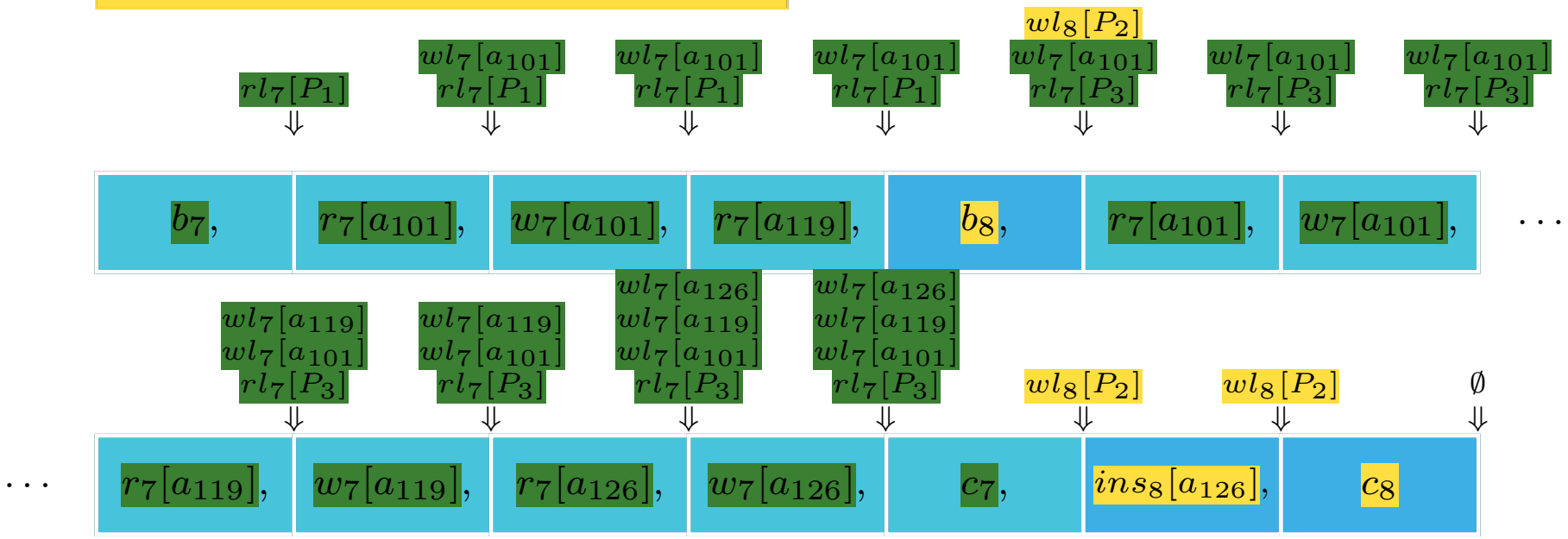
  H_8 uses $il_8[b_{EOF}]$
- Conflict arises when both insert lock and read lock are requested at same time
- Can produce needless conflicts

Solution 2: Predicate Locking

$P_1 : \sigma_{type=deposit \wedge rate \leq 5.50}(account)$

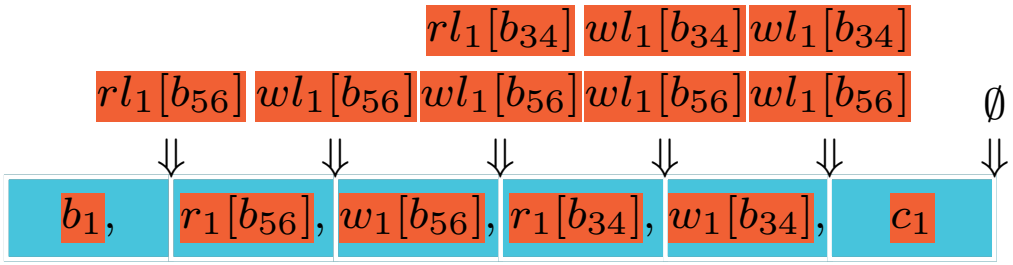
$P_2 : \sigma_{no=126 \wedge type=deposit \wedge cname=Boyd,M. \wedge rate=5.25 \wedge branch=34}(account)$

$P_3 : \sigma_{type=deposit}(account)$



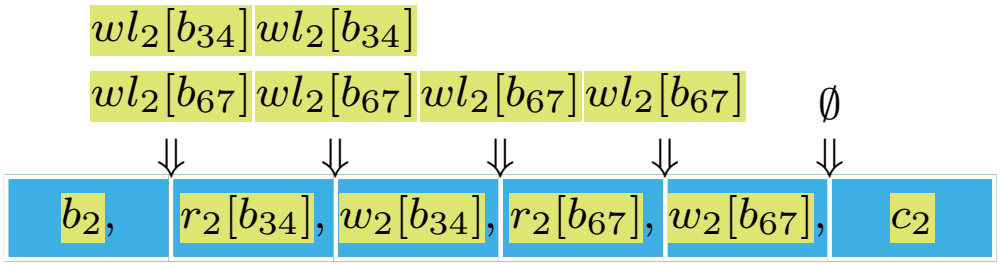
- lock the predicate that the transaction uses
- difficult to implement

When to lock: Aggressive Scheduler



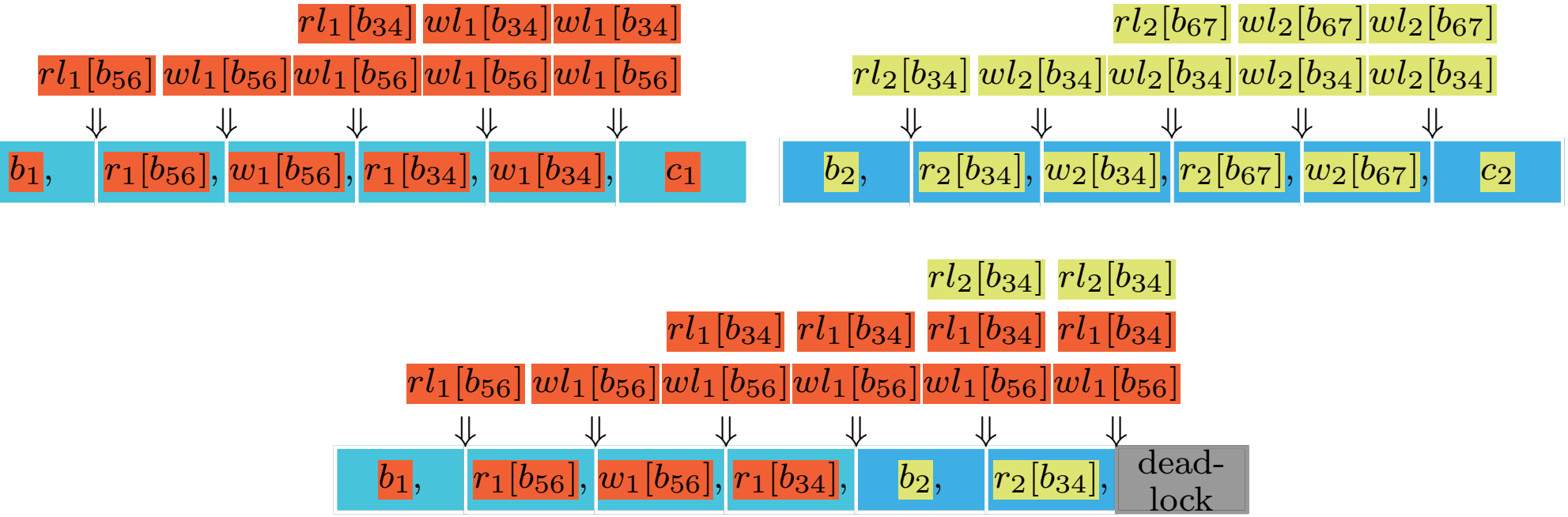
- delay taking locks as long as possible
- maximises concurrency
- might suffer delays later on

When to lock: Conservative Scheduler



- take locks as soon as possible
- removes risks of delays later on
- might refuse to start

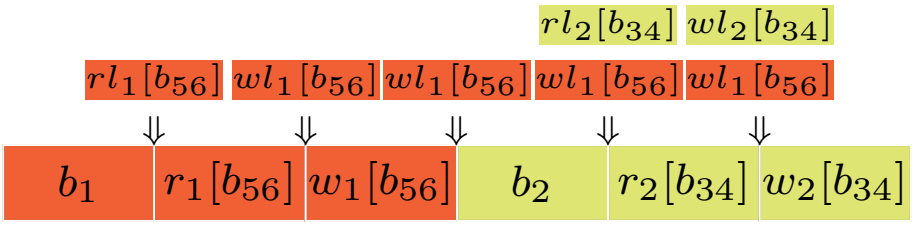
Aggressive Schedulers Might Deadlock



Issues to Address

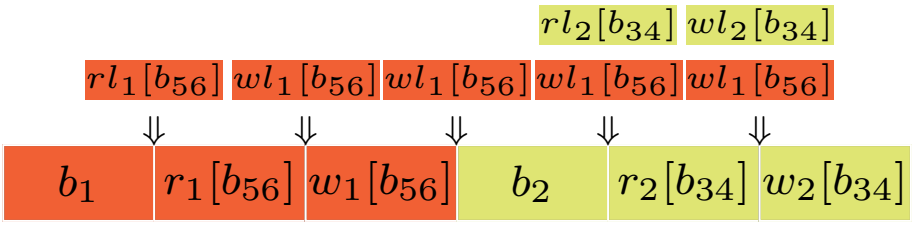
1. **deadlock**
2. shrinking requires advanced knowledge of what the remainder of the history will contain
3. the recoverability of transactions is not ensured (commit with dirty reads allowed)
4. **livelock**

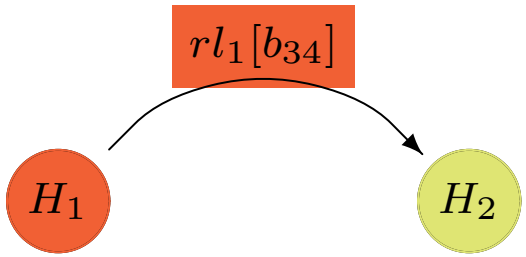
Deadlock Detection: WFG with No Cycle = No Deadlock



- **waits-for graph (WFG)**
- describes which transactions waits for others

Deadlock Detection: WFG with No Cycle = No Deadlock

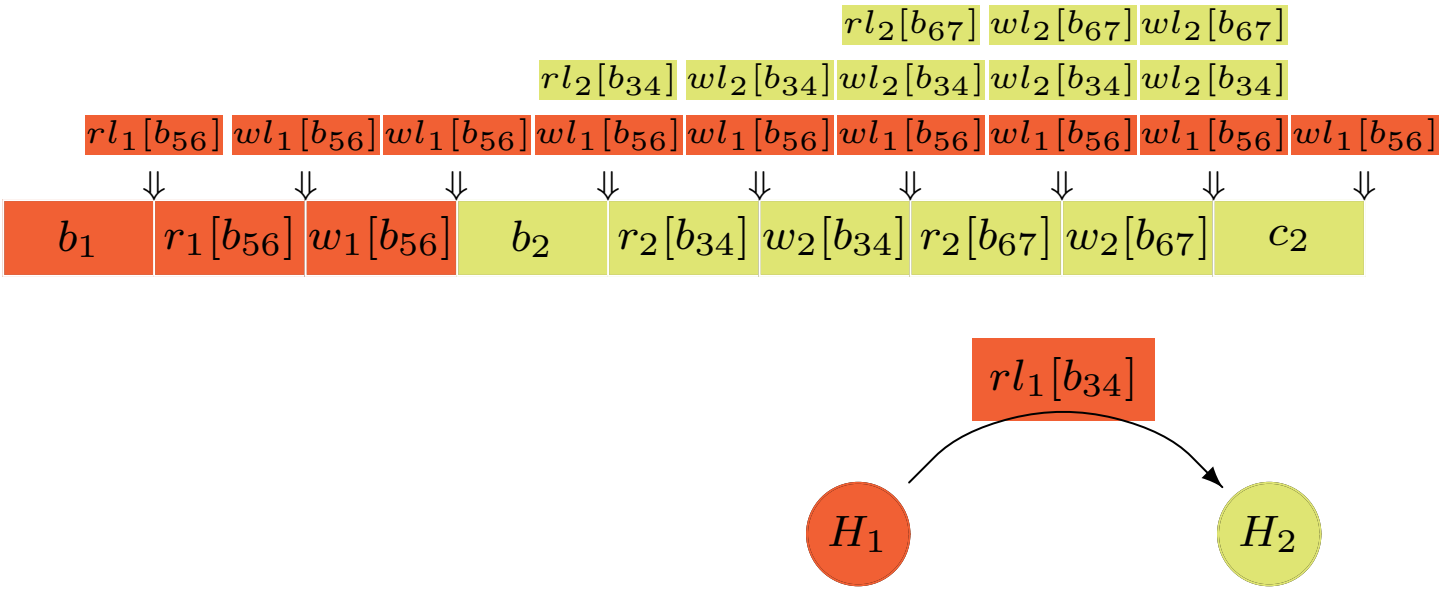




H_1 attempts $r_1[b_{34}]$, but is refused since H_2 has a write-lock, and so is put on WFG

- **waits-for graph (WFG)**
- describes which transactions waits for others

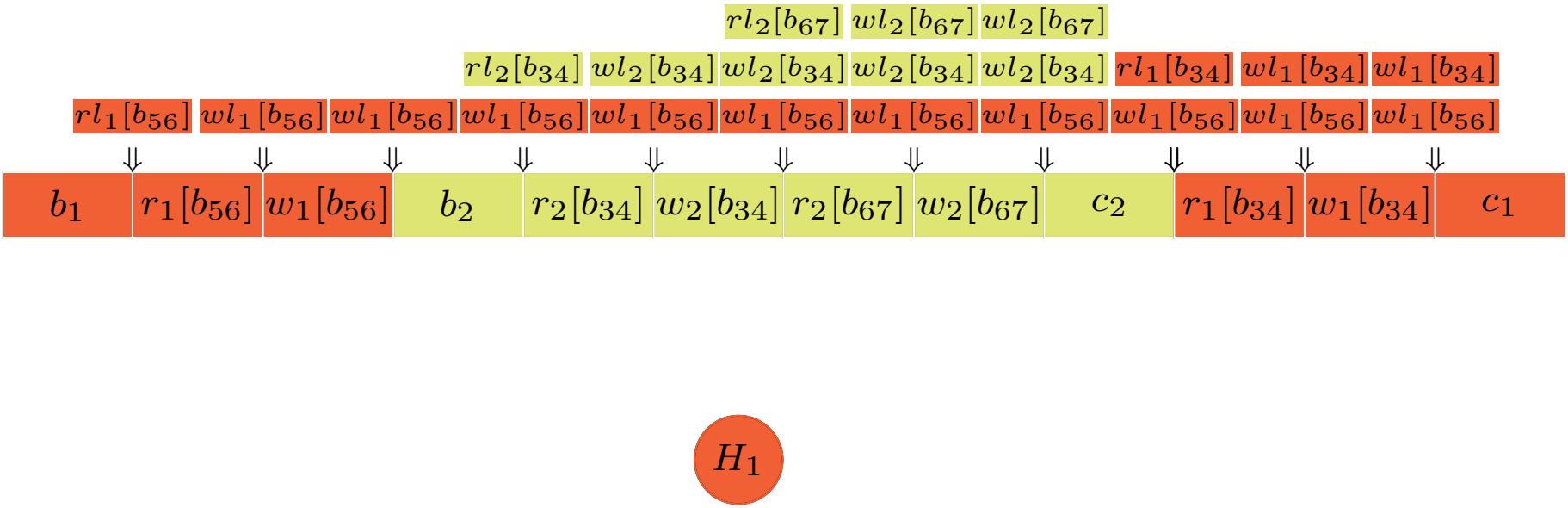
Deadlock Detection: WFG with No Cycle = No Deadlock



H_2 can proceed to complete its execution, after which it will have released all its locks

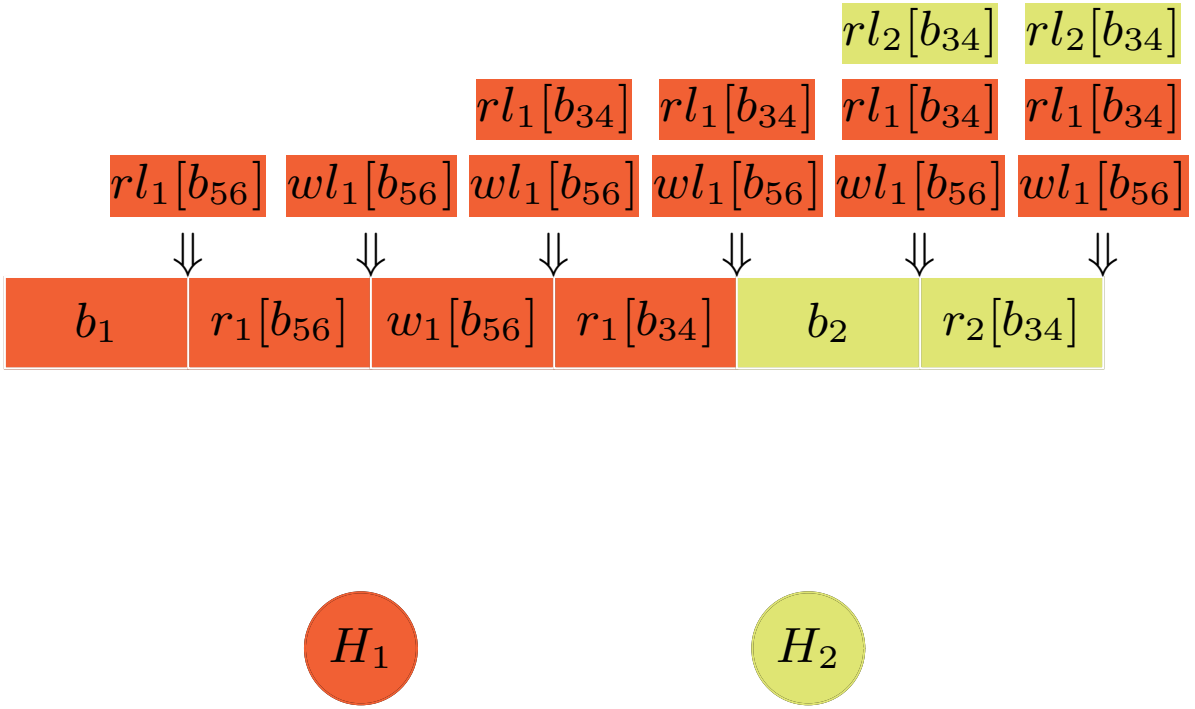
- **waits-for graph (WFG)**
- describes which transactions waits for others

Deadlock Detection: WFG with No Cycle = No Deadlock

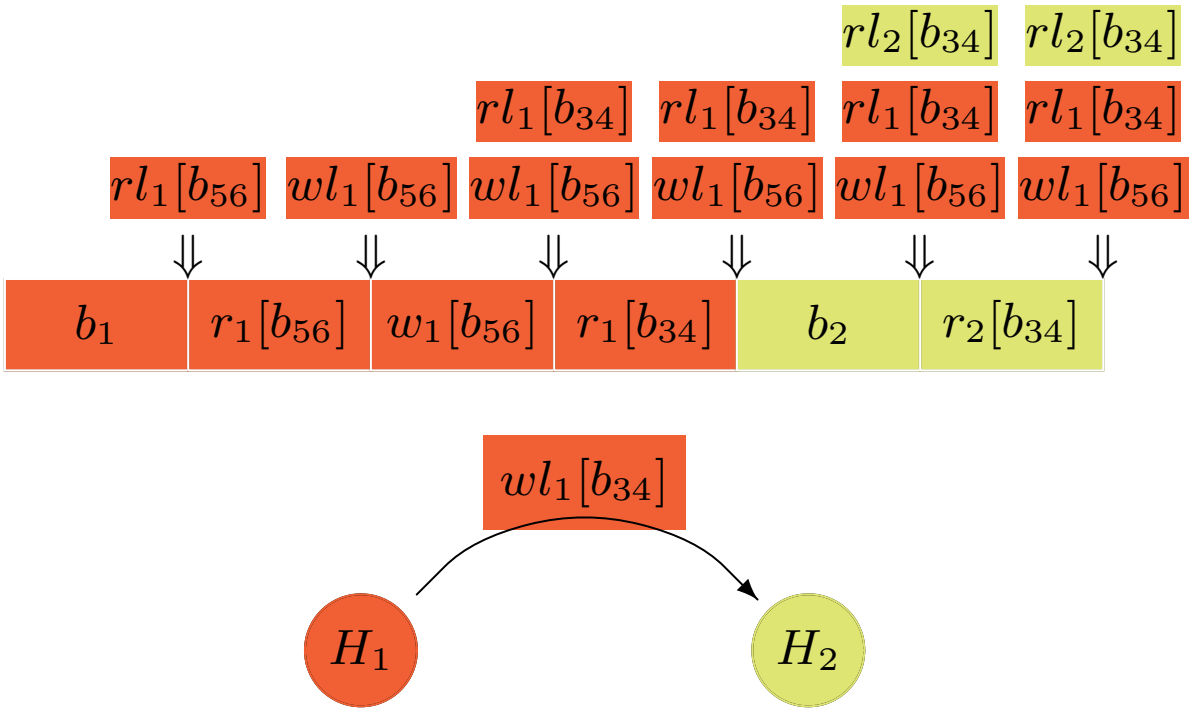


- **waits-for graph (WFG)**
- describes which transactions waits for others

Deadlock Detection: WFG with Cycle = Deadlock

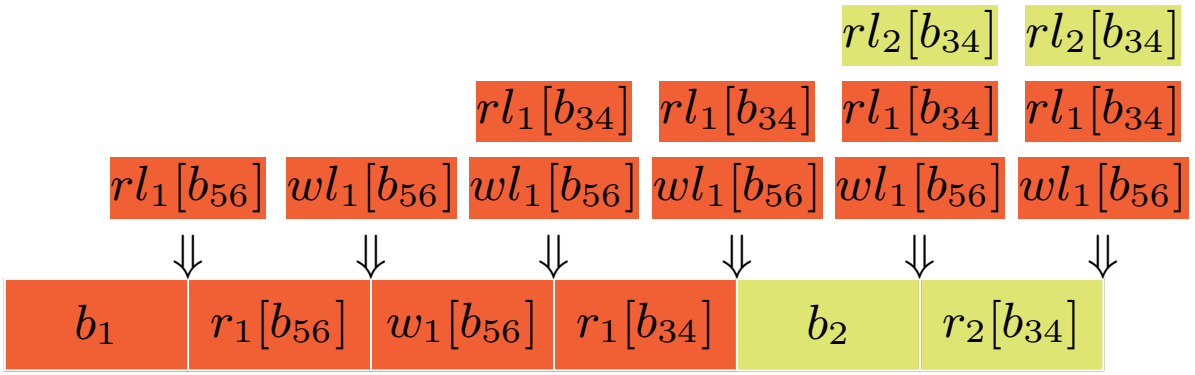


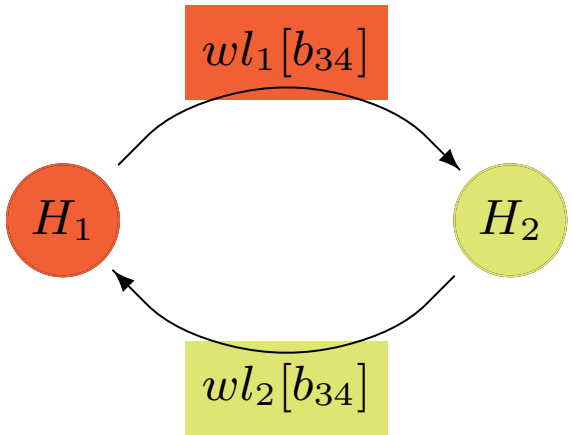
Deadlock Detection: WFG with Cycle = Deadlock



H_1 attempts $w_1[b_{34}]$, but is refused since H_2 has a read-lock, and so is put on WFG

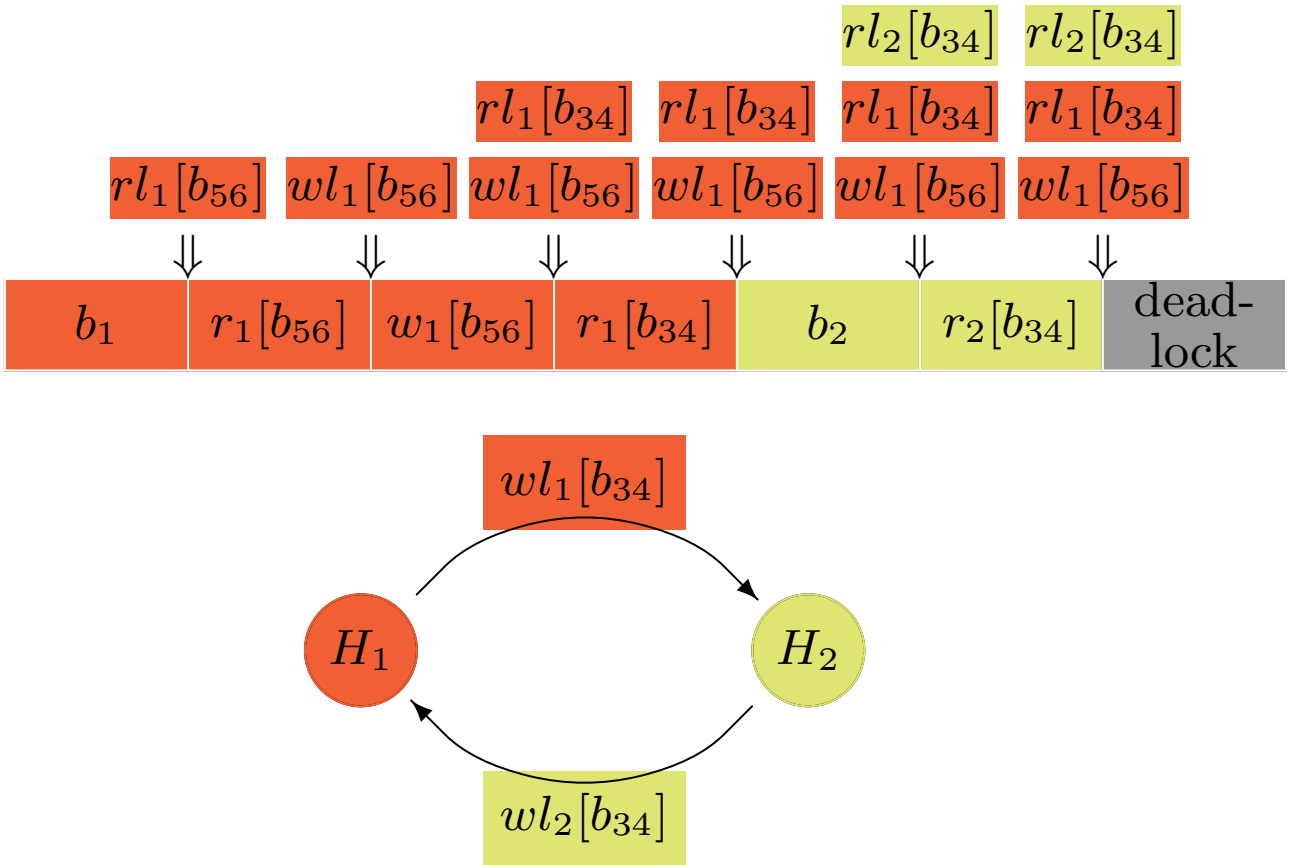
Deadlock Detection: WFG with Cycle = Deadlock





H_2 attempts $w_2[b_{34}]$, but is refused since H_1 has a read-lock, and so is put on WFG

Deadlock Detection: WFG with Cycle = Deadlock



Cycle in WFG means DB in a deadlock state, must abort either H_1 or H_2

Quiz 9: Resolving Deadlocks in 2PL

$H_1 = r_1[p_1], r_1[p_2], r_1[p_3], r_1[p_4], r_1[p_5], r_1[p_6]$

$H_2 = r_2[p_5], w_2[p_5], r_2[p_1], w_2[p_1]$

$H_3 = r_3[p_6], w_3[p_6], r_3[p_2], w_3[p_2]$

$H_4 = r_4[p_4], r_4[p_5], r_4[p_6]$

Suppose the transactions above have reached the following deadlock state

$H_d = r_1[p_1], r_1[p_2], r_1[p_3], r_1[p_4], r_2[p_5], w_2[p_5], r_2[p_1],$
$r_3[p_6], w_3[p_6], r_3[p_2], r_4[p_4]$

Which transaction should be aborted?

A	B	C	D
H_1	H_2	H_3	H_4

Quiz 10: Avoiding Deadlocks in 2PL

Which scheduling method for 2PL avoids deadlocks?

A

Gain all the locks for a transaction at the start, and then release locks as they are no longer required during transaction execution.

B

Gain locks only as required during transaction execution, and release locks only at the end of the transaction.

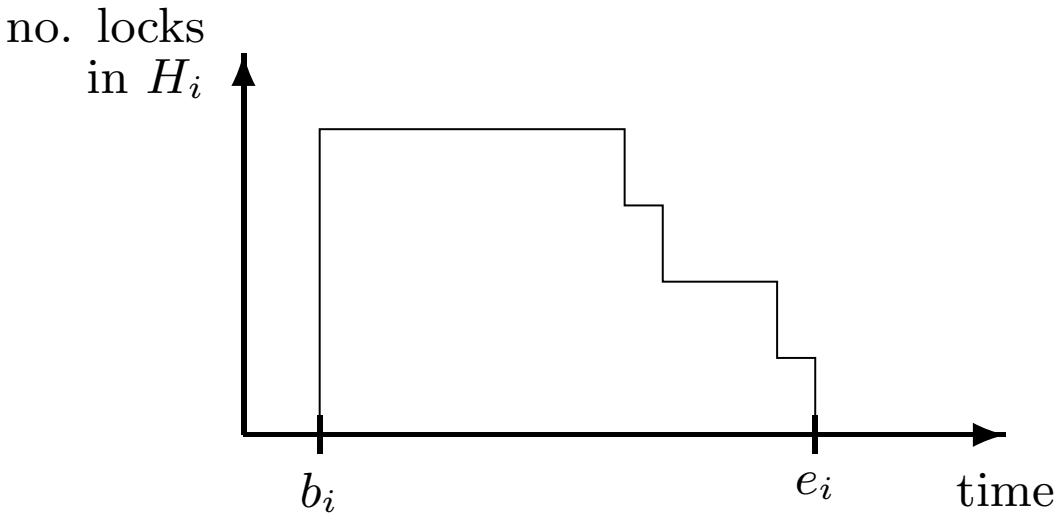
C

Gain locks only as required during transaction execution, and release locks as soon as possible.

D

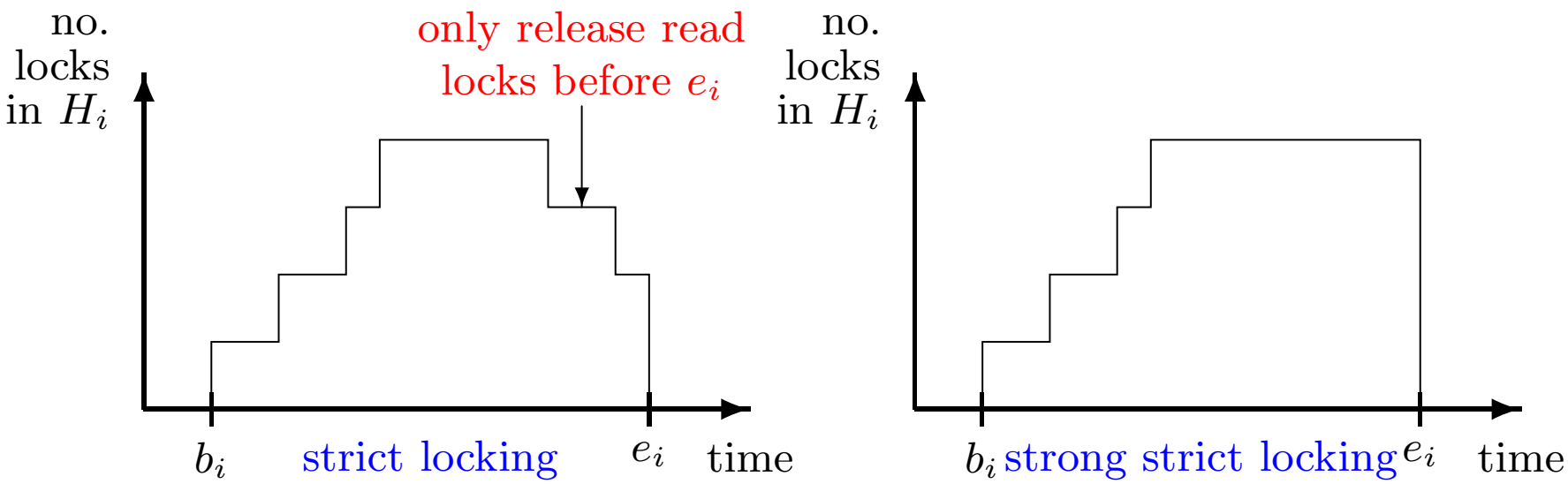
Gain locks only as required during transaction execution, except if a read lock is being obtained, first check to see if the transaction will later require a write lock on the same object, and if so, obtain a write lock. Release locks as soon as possible.

Conservative Locking



- prevents deadlock
- release problem harder
- still not recoverable

Strict Locking



- strict locking prevents write locks being released before transaction end
- recoverable
- allows deadlocks
- strong strict locking means no locks released before end
- no problem determining when to release

Livelock Prevention

- livelocks caused by transaction waiting indefinite time for lock
- associate priority with transaction
- increase priority with time
- at some point transaction must reach maximum priority

Transaction Isolation Levels

- Do we always need ACID properties?

```
BEGIN TRANSACTION T3
        SELECT DISTINCT no
        FROM movement
        WHERE amount>=1000.00
COMMIT TRANSACTION T3
```

- Some transactions only need 'approximate' results
  e.g. Management overview
  e.g. Estimates
- May execute these transactions at a 'lower' level of concurrency control

SQL: READ UNCOMMITTED

- Set by executing SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED
- The weakest level, only prevents dirty writes
- Allows transactions to read uncommitted data
  Hence allows Dirty reads

Anomaly	Possible
Dirty Write	N
Dirty Read	Y
Lost Update	Y
Inconsistent Analysis	Y
Phantom	Y
Write Skew	Y

SQL: READ COMMITTED

- Allows transactions to only read committed data
- Recoverable; but may suffer inconsistent analysis

Anomaly	Possible
Dirty Write	N
Dirty Read	N
Lost Update	Y
Inconsistent Analysis	Y
Phantom	Y
Write Skew	Y

SQL: SNAPSHOT

- Transactions behave as if read committed version of data at start of transaction, and write all data at end of transaction
- Not standard SQL. Available in SQL-Server 2005
- Postgres SERIALIZABLE is infact SNAPSHOT

Anomaly	Possible
Dirty Write	N
Dirty Read	N
Lost Update	N
Inconsistent Analysis	N
Phantom	N
Write Skew	Y

SQL: REPEATABLE READ

- Allows inserts to tables already read
- Allows phantom reads
- Prevents write skew

Anomaly	Possible
Dirty Write	N
Dirty Read	N
Lost Update	N
Inconsistent Analysis	N
Phantom	Y
Write Skew	N

SQL: SERIALIZABLE

- Execution equivalent to a serial execution
- no anomalies of any kind (not just those listed)

Anomaly	Possible
Dirty Write	N
Dirty Read	N
Lost Update	N
Inconsistent Analysis	N
Phantom	N
Write Skew	N