Concurrency Control

Peter M^{c}Brien

Dept. of Computing, Imperial College London

Transactions: ACID properties

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.

Example Data

branch		
<u>sortcode</u>	bname	cash
56	'Wimbledon'	94340.45
34	'Goodge St'	8900.67
67	'Strand'	34005.00

movement			
<u>mid</u>	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				
<u>no</u>	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

key branch(sortcode)
key branch(bname)
key movement(mid)
key account(no)
movement(no) $\stackrel{fk}{\Rightarrow}$ account(no)
account(sortcode) $\stackrel{fk}{\Rightarrow}$ branch(sortcode)

Transaction Properties: Atomicity

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

CRASH

Suppose that the system crashes half way through processing a cash transfer, and the first part of the transfer has been written to disc

- The database on disc is left in an inconsistent state, with £10,000 'missing'
- A DBMS implementing **Atomicity** of transactions would on restart UNDO the change to branch 56

Transaction Properties: Consistency

```
BEGIN TRANSACTION
   DELETE FROM branch
   WHERE sortcode=56

   INSERT INTO account
   VALUES (100,'Smith, J','deposit',5.00,34)
END TRANSACTION
```

Suppose that a user deletes branch with sortcode 56, and inserts a deposit account number 100 for John Smith at branch sortcode 34

- The database is left in an inconsistent state for two reasons
  - it has three accounts recorded for a branch that appears not to exist, and
  - it has two records for account number 100, with different details for the account
- A DBMS implementing **Consistency** of transactions would forbid both of these changes to the database

Transaction Properties: Isolation

Transaction 1	Transaction 2
BEGIN TRANSACTION	BEGIN TRANSACTION
UPDATE branch SET cash=cash −10000.00 WHERE sortcode=56	
	SELECT SUM(cash) AS net_cash FROM branch
UPDATE branch SET cash=cash +10000.00 WHERE sortcode=34	
END TRANSACTION	END TRANSACTION

Suppose that the system sums the cash in the bank in one transaction, half way through processing a cash transfer in another transaction

- The result of the summation of cash in the bank erroneously reports that £10,000 is missing
- A DBMS implementing **Isolation** of transactions ensures that transactions always report results based on the values of committed transactions

Transaction Properties: Durability

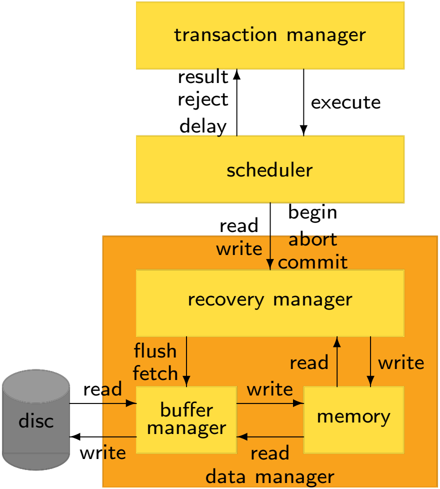
```
BEGIN TRANSACTION
   UPDATE branch
   SET    cash=cash -10000.00
   WHERE  sortcode=56

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

CRASH

Suppose that the system crashes after informing the user that it has committed the transfer of cash, but has not yet written to disc the update to branch 34

- The database on disc is left in an inconsistent state, with £10,000 'missing'
- A DBMS implementing **Durability** of transactions would on restart complete the change to branch 34 (or alternatively never inform a user of commitment with writing the results to disc).

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

history of transaction T_n

1. Begin transaction b_n (only given if necessary for discussion)
2. Various read operations on objects $r_n[o_j]$ and write operations $w_n[o_j]$
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

history of transaction T_n

1. Begin transaction b_n (only given if necessary for discussion)
2. Various read operations on objects $r_n[o_j]$ and write operations $w_n[o_j]$
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

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

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

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

$r_1[b_{56}]$, $w_1[b_{56}]$, $r_1[b_{34}]$, $w_1[b_{34}]$, 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, $lost update$, 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 2: Inconsistent analysis

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

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

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

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

$r_1[b_{56}]$, 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)

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

Which anomaly does H_y suffer?

A	B
None	Lost Update

C	D
Inconsistent Analysis	Dirty Read

Quiz 6: Anomalies (3)

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

Which anomaly does H_z suffer?

A
None

B
Lost Update

C
Inconsistent Analysis

D
Dirty Read

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

Patterns of operations associated with Anomalies

Anomaly	Pattern
Dirty Write	$w_1[o] \prec w_2[o] \prec e_1$
Dirty Read	$w_1[o] \prec r_2[o] \prec e_1$
Inconsistent Analysis	$r_1[o_a] \prec w_2[o_a],\ w_2[o_b] \prec r_1[o_b]$
Lost Update	$r_1[o] \prec w_2[o] \prec w_1[o]$

Notation

- e_i means either c_i or a_i occurring
- $op_a \prec op_b$ mean op_a occurs before op_b in a history

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 T11
  UPDATE account
  SET    rate=max_rate
  FROM   (SELECT MAX(rate) AS max_rate
          FROM account) AS max_data
  WHERE  rate<max_rate
COMMIT TRANSACTION T11
```

```
BEGIN TRANSACTION T12
  UPDATE account
  SET    rate=min_rate
  FROM   (SELECT MIN(rate) AS min_rate
          FROM account) AS min_data
  WHERE  rate>min_rate
COMMIT TRANSACTION T12
```

$$r_{11}[a_{101}], r_{11}[a_{119}], r_{12}[a_{101}], r_{12}[a_{119}], w_{11}[a_{101}], w_{12}[a_{119}], c_{11}, c_{12}$$

Worksheet: Anomalies

Serialisable Transaction Execution

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

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

$\Rightarrow$

$C(H_c) =$
$r_1[b_{56}], r_2[b_{34}], w_2[b_{34}],$
$w_1[b_{56}], r_4[b_{56}],$
$r_1[b_{34}], w_1[b_{34}], c_1, r_4[b_{34}],$
$r_2[b_{67}], w_2[b_{67}], c_2, r_4[b_{67}], c_4$

Possible Serial Equivalents

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

- H_1, H_2, H_4
- H_1, H_4, H_2
- H_2, H_1, H_4
- H_2, H_4, H_1
- H_4, H_1, H_2
- H_4, H_2, H_1

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

Conflicts: Potential For Problems

conflict

A **conflict** occurs when there is an 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 $x \neq y$

Only consider pairs where there is no third operation $rw_z[o]$ between the pair of operations that conflicts with both

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 7: 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.

Serialisation Graph

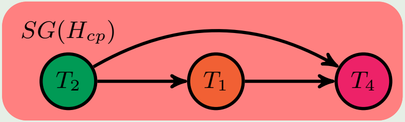
Serialisation Graph

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



$SG(H_{cp})$ is acyclic, therefore H_{cp} is CSR. Serialisation order T_2, T_1, T_4

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

```
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, a_5

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

$H_c = w_6[a_{101}]$, rate=6.0, $w_5[a_{101}]$, rate=5.5, $w_5[a_{119}]$, rate=5.5, $w_6[a_{119}]$, rate=6.0, a_5, c_6

$H_c \in ACA$

$H_c \notin ST$

Quiz 8: 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: Serialisability and Recoverability

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

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

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

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

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

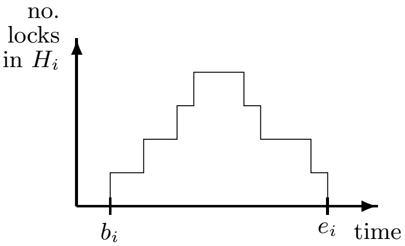
$$H_z = r_3[o_1], w_3[o_1], r_1[o_1], w_3[o_2], w_1[o_1], w_1[o_2], r_2[o_2], w_2[o_2], w_1[o_3], w_2[o_1], c_3, c_1, c_2$$

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
   1. **growing phase**
   2. **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 9: 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

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

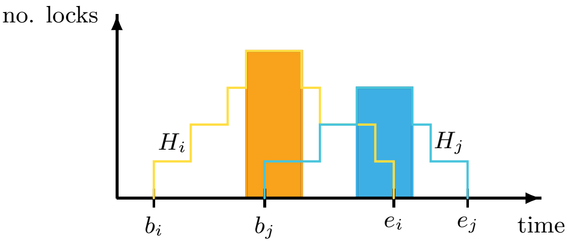
C

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

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 → 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

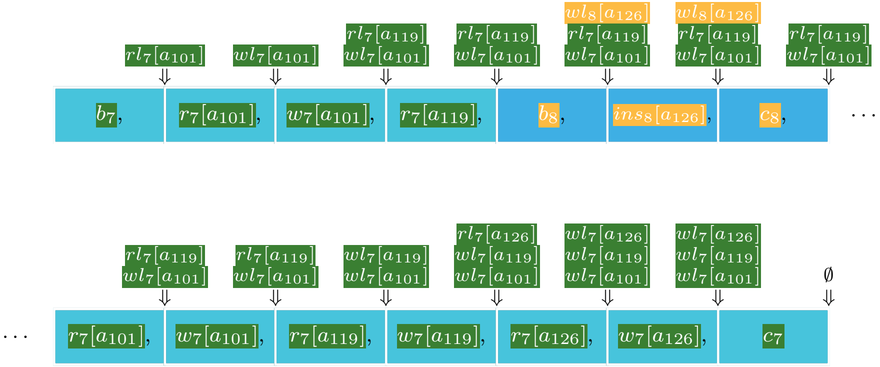
```
BEGIN TRANSACTION T7
   UPDATE account
   SET    rate=rate+0.25
   WHERE  type='deposit'
   AND    rate<5.5

   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}?
  - predicate type='deposit' AND rate<5.5

Solution 1: Table Locks

- Problem with phantom reads is due to changing data matching query
- Read lock table when performing a 'scan' of the table
- ✗ Can produce needless conflicts
- ✓ Can be efficient if large parts of the table are being updated

Query Requiring Table Lock

```
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
```

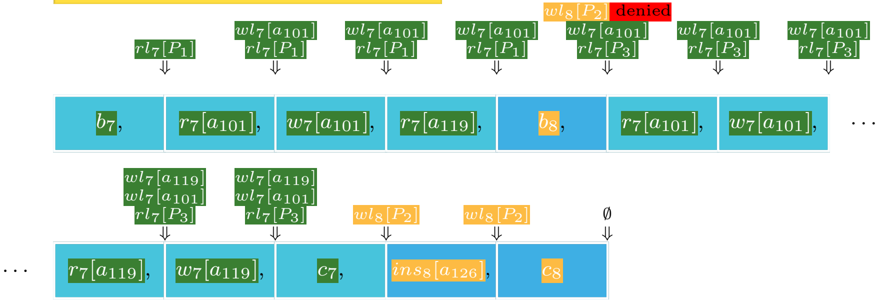
H_7 uses $wl_7[a]$ instead of $wl_7[a_{101}]$, $wl_7[a_{119}]$

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

Quiz 10: Predicate Locks

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

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

key branch(sortcode)
key branch(bname)
key account(no)
account(sortcode) $\overset{fk}{\Rightarrow}$ branch(sortcode)

Which SQL query requires a predicate lock in order to prevent phantom reads by any transaction in which it is placed?

A

```
SELECT  *
FROM    account
WHERE   no=101
```

B

```
SELECT  *
FROM    branch
WHERE   name='Wimbledon'
```

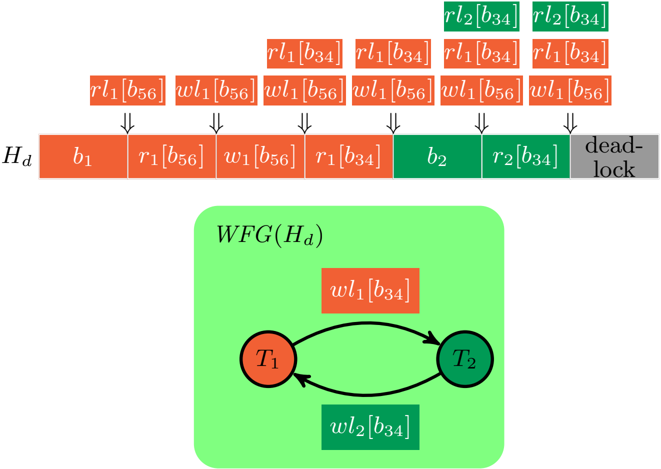
C

```
SELECT  *
FROM    branch
        JOIN account
        USING (sortcode)
WHERE   branch.sortcode=56
```

D

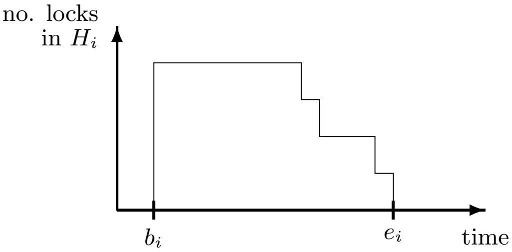
```
SELECT  *
FROM    branch
        JOIN account
        USING (sortcode)
WHERE   no=101
```

Deadlock Detection: WFG with Cycle = Deadlock



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

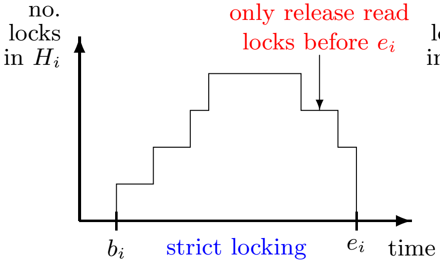
Conservative Locking

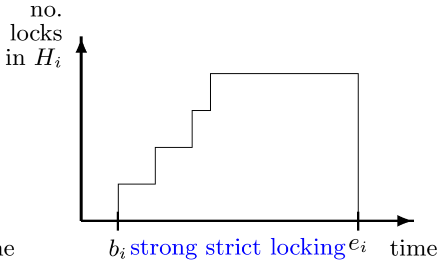


Conservative Locking

- prevents deadlock
- when to release locks problem
- not recoverable

Strict Locking





Strict Locking

- prevents write locks being released before transaction end
- allows deadlocks
- no dirty reads/writes $\rightarrow$ recoverable

Strong Strict Locking

In addition to strict locking properties

- prevents read locks being released before transaction end
- simple to implement
- suitable for distributed transactions (using atomic commit)

2PL and the Prevention of Anomalies

- Define e_i to mean either c_i or a_i occurring
- Define $op_a \prec op_b$ to mean op_a occurs before op_b in a history

Anomaly	Pattern	Prevented by
Dirty Write	$w_1[o] \prec w_2[o],\ w_2[o] \prec e_1$	Strict 2PL
Dirty Read	$w_1[o] \prec r_2[o],\ r_2[o] \prec e_1$	Strict 2PL
Inconsistent Analysis	$r_1[o_a] \prec w_2[o_a],\ w_2[o_b] \prec r_1[o_b]$	2PL
Lost Update	$r_1[o] \prec w_2[o],\ w_2[o] \prec w_1[o]$	2PL
Simple Write Skew	$r_1[o_a] \prec w_2[o_b],\ r_1[o_b] \prec w_2[o_a]$	2PL
Write Skew	$r_1[P_1] \prec w_2[x \in P_2],\ r_2[P_2] \prec w_1[y \in P_1]$	2PL with Predicate Locks
Phantom Read	$r_1[P_1] \prec w_2[x \in P_1],\ w_2[y \in P_2] \prec r_1[P_2]$	2PL with Predicate Locks

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 allows you to vary the 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
- Pre Postgres 9.1 and Oracle 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

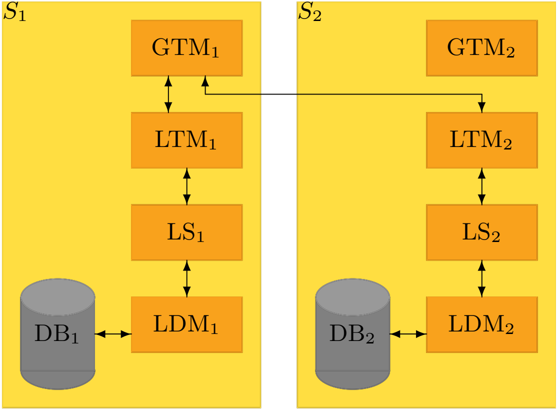
Anomaly	Possible
Dirty Write	N
Dirty Read	N
Lost Update	N
Inconsistent Analysis	N
Phantom	Y
Write Skew	N

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

Distributed Concurrency Control



Distributed 2PL

- Fragmentation and replication imply coordination of transaction commit
- Fragmentation implies locks go to relevant fragments
- Replication implies replication of locks

Deadlock in Centralised DBMS

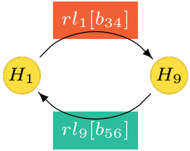
BEGIN TRANSACTION T1
    EXEC move_cash(56,34,10000.00)
COMMIT TRANSACTION T1

BEGIN TRANSACTION T9
    EXEC move_cash(34,56,2000.00)
COMMIT TRANSACTION T9

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

$H_9 = r_9[b_{34}], w_9[b_{34}], r_9[b_{56}], w_9[b_{56}], c_9$

$H_c = r_1[b_{56}], w_1[b_{56}], r_9[b_{34}], w_9[b_{34}], deadlock$



Distribution of Histories

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

BEGIN TRANSACTION T9
EXEC move_cash(34,56,2000.00)
COMMIT TRANSACTION T9

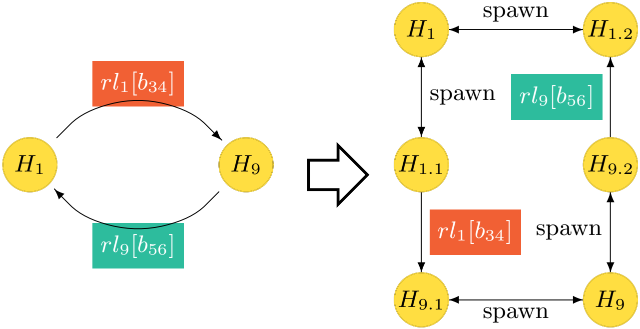
$H_1 = r_1[b_{56}], w_1[b_{56}], r_1[b_{34}], w_1[b_{34}], c_1$

$H_9 = r_9[b_{34}], w_9[b_{34}], r_9[b_{56}], w_9[b_{56}]$

S_1:
$H_{1.1} = r_1[b_{34}], w_1[b_{34}], c_1$
$H_{9.1} = r_9[b_{34}], w_9[b_{34}], c_9$

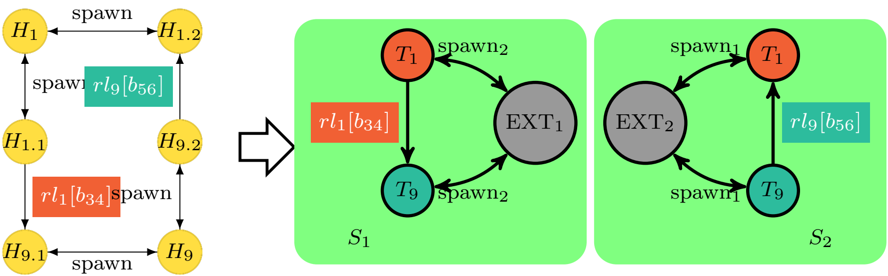
S_2:
$H_{1.2} = r_1[b_{56}], w_1[b_{56}], c_1$
$H_{9.2} = r_9[b_{56}], w_9[b_{56}], c_9$

Local WFG → Sub-Transactions



$H_c = r_1[b_{56}], w_1[b_{56}], w_9[b_{34}], w_9[b_{34}]$, deadlock

Sub-transactions → EXT nodes+DWFG



- When local cycle appears, fetch remote WFG

Quiz 11: Deadlock detection in DWFGs

Which of the following is correct?

A

Deadlock has occurred once a cycle has appeared at any node executing a distributed transaction.

B

Deadlock might have occurred once a cycle has appeared at any node executing a distributed transaction.

C

Deadlock has occurred once a cycle has appeared at all nodes executing a distributed transaction.

D

Deadlock has occurred once a cycle has appeared at all nodes in the distributed database.

Worksheet: Distributed WFG

$T_1 = r_1[b_{56}], w_1[b_{56}], r_1[b_{34}], w_1[b_{34}]$

$T_2 = r_2[b_{34}], w_2[b_{34}], r_2[b_{67}], w_2[b_{67}]$

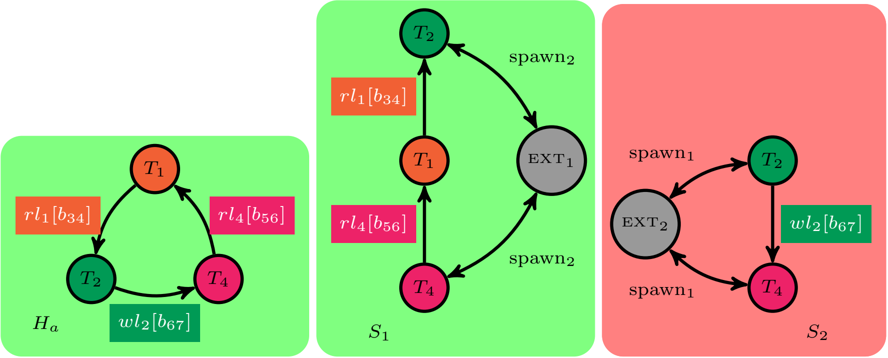
$T_4 = r_4[b_{67}], r_4[b_{56}], r_4[b_{34}]$

Worksheet: Distributed WFG

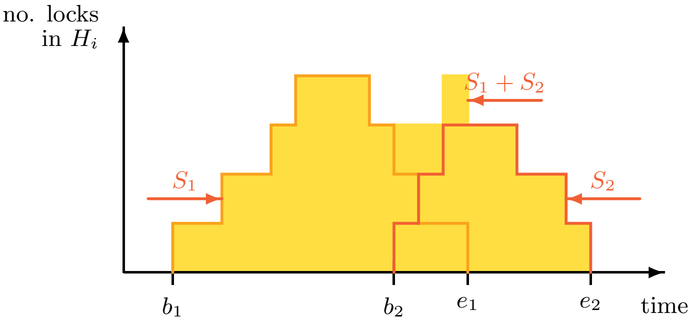
Consider the conflicts $w_1[b_{56}] \rightarrow r_4[b_{56}]$, $w_2[b_{34}] \rightarrow r_1[b_{34}]$, $r_4[b_{67}] \rightarrow w_2[b_{67}]$

These can give a deadlock state:

$H_a = r_1[b_{56}], w_1[b_{56}], r_2[b_{34}], w_2[b_{34}], r_2[b_{67}], r_4[b_{67}]$, deadlock

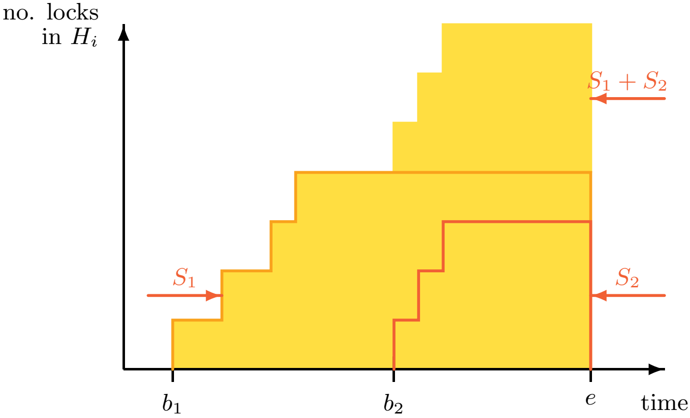


Incorrect Global 2PL



- Can not just execute 2PL at each site

Correct Global 2PL with Strong Strict Locking



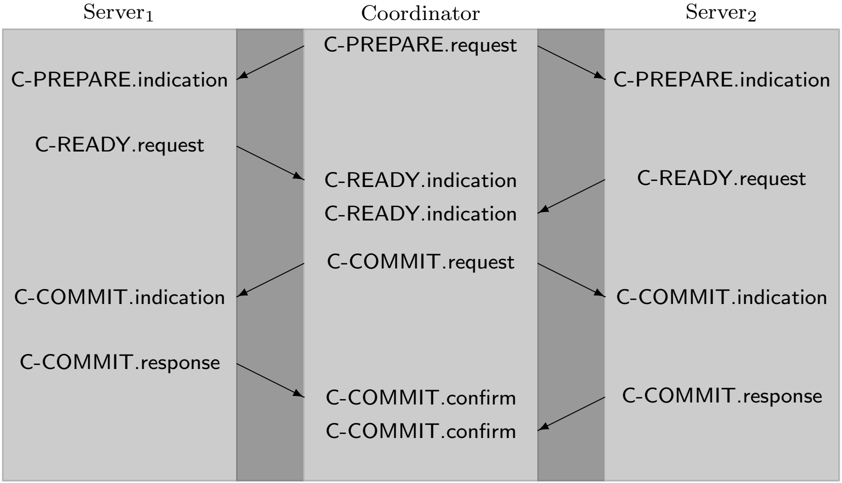
- Execute Strong Strict 2PL at each site
- Use global atomic commit to end transaction

Two-Phase Commit (2PC)

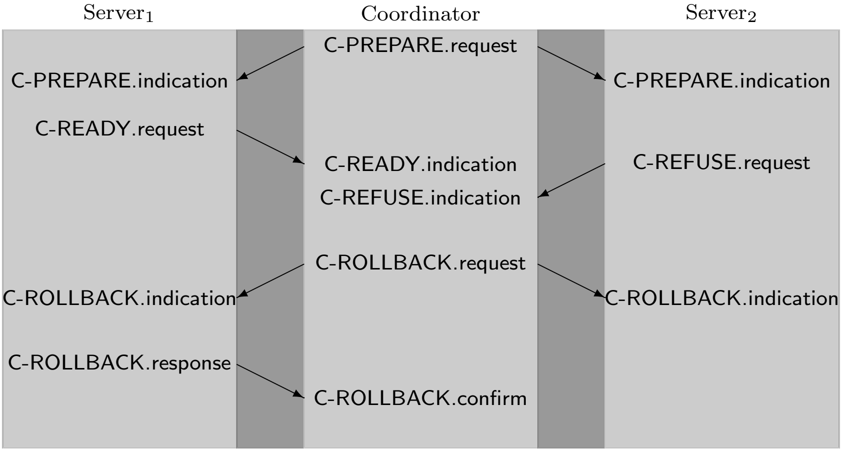
service element	source	semantics
C-PREPARE	coordinator	get ready to commit
C-READY	server	ready to commit
C-REFUSE	server	not ready to commit
C-COMMIT	coordinator	commit the transaction
C-ROLLBACK	server	rollback the transaction
C-RESTART	either	try to return to start of transaction

- OSI model application layer **commitment, concurrency, and recovery** (**CCR**) service
- .NET System.Transactions namespace, Java Transaction API (JTA)
- Commonly available for commercial DBMSs

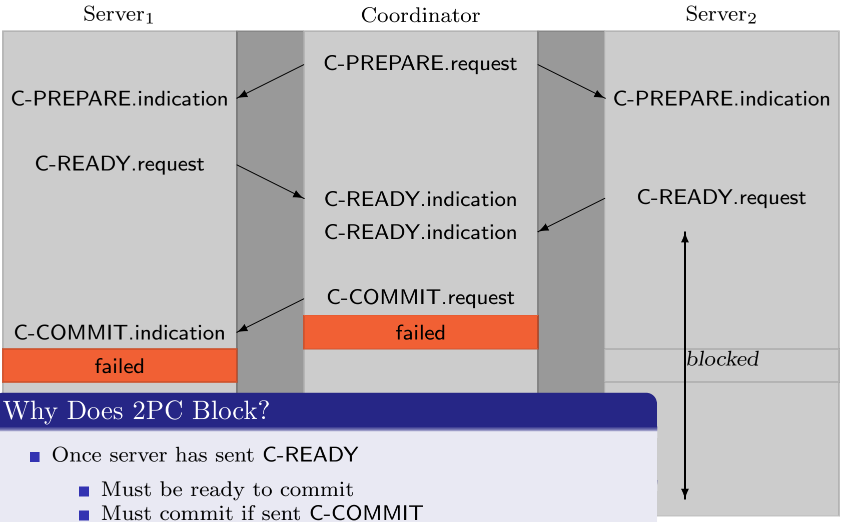
2PC: Normal Commit



2PC: Normal Abort



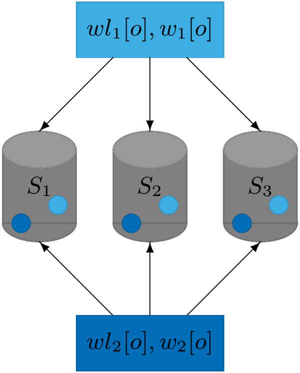
2PC: Blocking



Why Does 2PC Block?

- Once server has sent C-READY
  - Must be ready to commit
  - Must commit if sent C-COMMIT
  - Must abort if sent C-ROLLBACK
- Can prevent problem by separating voting decision from commit command

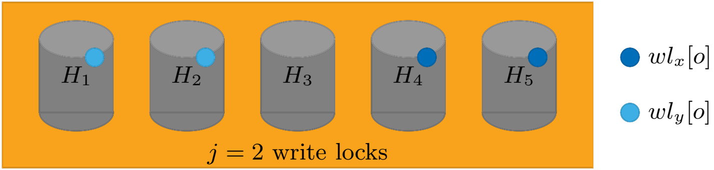
Where to send the write locks?



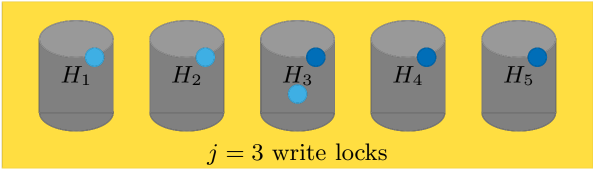
- must send $w_x[o]$ to all hosts
- could send $wl_x[o]$ to all hosts
- conflict detected at all hosts

Write-Write conflicts

write-write conflict missed



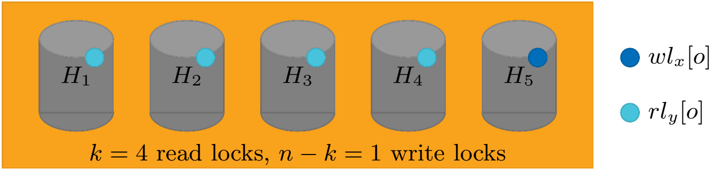
write-write conflict detected



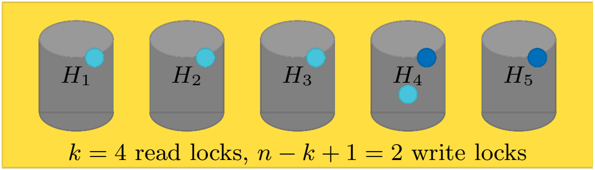
- $j \geq \lceil \frac{n+1}{2} \rceil$

Read-Write conflicts

read-write conflict missed



read-write conflict detected



- $j \geq n - k + 1$

Detecting all conflicts

Must detect both types of conflict

- n hosts
- each read lock sent to k hosts
- each write lock sent to j hosts

- To detect write-write conflicts: $j \geq \lceil \frac{n+1}{2} \rceil$
- To detect read-write conflicts: $j \geq n - k + 1$

Quiz 12: Distributed Locking

Consider a distributed database with data replicated to six sites.
$|rl_x[o]|$ indicates the number of sites to which any read lock is sent.
$|wl_x[o]|$ indicates the number of sites to which any write lock is sent.

Which distributed locking strategy is invalid?

A

$|rl_x[o]| = 1, |wl_x[o]| = 6$

B

$|rl_x[o]| = 2, |wl_x[o]| = 5$

C

$|rl_x[o]| = 3, |wl_x[o]| = 4$

D

$|rl_x[o]| = 4, |wl_x[o]| = 3$