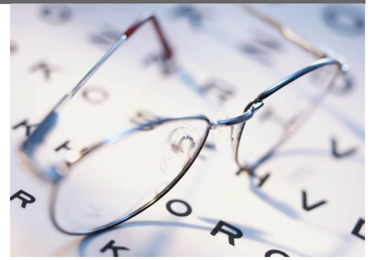
Concurrency Control: 2PL
Concurrency Control: Introduction to Schedules and Serializability

PCSD, Marcos Vaz Salles

Do-it-yourself Recap: Techniques for Performance

- What is the meaning of the following performance metrics: throughput, latency, overhead, utilization, capacity?

- Why can concurrency improve throughput and latency? How does that related to modern hardware characteristics?



What should we learn today?

- Identify the multiple interpretations of the property of atomicity
- Implement methods to ensure before-or-after atomicity, and argue for their correctness
- Explain the variants of the two-phase locking (2PL) protocol, in particular the widely-used Strict 2PL
- Discuss definitions of serializability and their implications, in particular conflict-serializability and view-serializability
- Apply the conflict-serializability test using a precedence graph to transaction schedules

Read-Write Systems

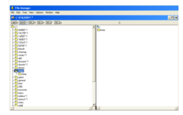
- On-Line Transaction Processing (**OLTP**)
    - Process multiple, but relatively simple, application functions

- **Examples**
    - Order processing, e.g., Amazon
    - Item buy/sell in computer games, e.g., EVE Online
    - High-performance trading
    - Updates on social networks, e.g., Facebook

Atomicity vs. Performance is the fundamental trade-off

- Last week's property: **Strong Modularity**

- This week: **Atomicity (before-or-after flavor)**

Transaction

- Reliable unit of work against memory abstraction
  - In the next lectures, we will use "memory state" and "database" interchangeably!

- **ACID Properties**
  - Atomicity: transactions are all-or-nothing
  - Consistency: transaction takes database from one consistent state to another
  - Isolation: transaction executes as if it were the only one in the system (aka before-or-after atomicity)
  - Durability: once transaction is done ("committed"), results are persistent in the database

Examples of Transactions in SQL

Transaction T1: `TRANSFER`

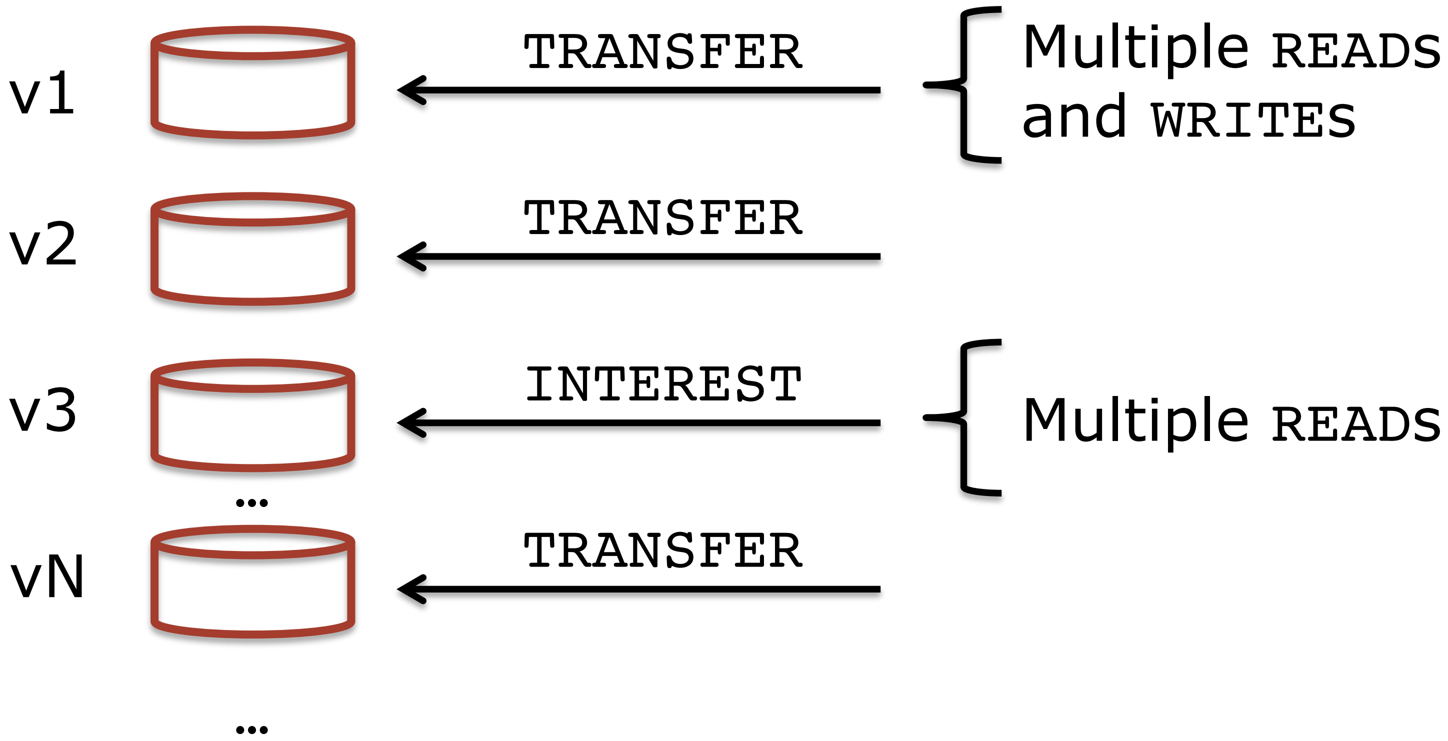
```
BEGIN
  UPDATE account
  SET bal = bal + 100
  WHERE account_id = 'A';
  --
  UPDATE account
  SET bal = bal – 100
  WHERE account_id = 'B';
COMMIT
```

Transaction T2: `INTEREST`

```
BEGIN
  UPDATE account
  SET bal = bal * 1.06;
COMMIT
```

Under the hood, we know it all translates to calls to `READ` and `WRITE`

Conceptual Model: Version Histories



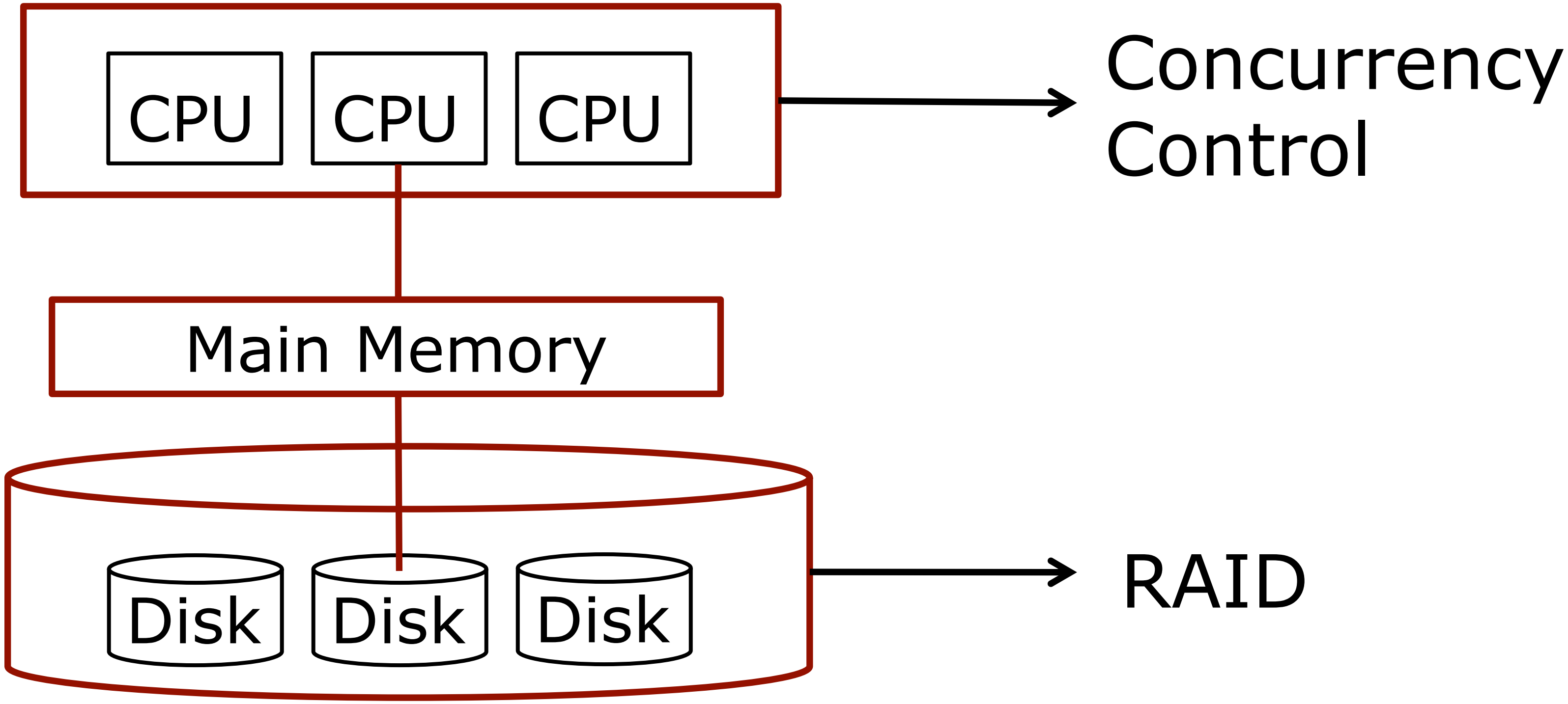
The many faces of atomicity

- **Atomicity** is strong modularity mechanism!
  - Hides that one high-level action is actually made of many sub-actions

- **Before-or-after** atomicity
  - == Isolation
  - Cannot have effects that would only arise by interleaving of parts of transactions

- **All-or-nothing** atomicity
  - == Atomicity (+ Durability)
  - Cannot have partially executed transactions
  - Once executed and confirmed, transaction effects are visible and not forgotten

Scaling Up



- **Problem**: Ensure automatically that all interactions leave data consistent

Goal of Concurrency Control

- Transactions should be executed so that it is *as though* they executed in some serial order
  - Also called Isolation or Serializability or Before-or-after atomicity

- Weaker variants also possible
  - Lower “degrees of isolation”

Example

- Consider again our two transactions (*Xacts*):

```
T1:   BEGIN   A=A+100,   B=B-100    END
T2:   BEGIN   A=1.06*A,  B=1.06*B   END
```

- T1 transfers $100 from B's account to A's account
- T2 credits both accounts with 6% interest
- If submitted concurrently, net effect should be equivalent to Xacts running in some serial order
  - No guarantee that T1 "logically" occurs before T2 (or vice-versa) – but one of them is true

Source: Ramakrishnan & Gehrke (partial)

Solution 1

1) Get exclusive lock on entire database
2) Execute transaction
3) Release exclusive lock

- Transactions execute in *critical section*
- Serializability guaranteed because execution is serial!

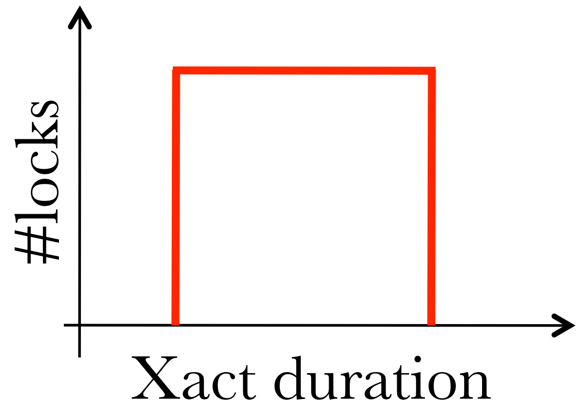
- Problems?

Solution 2

1) Get exclusive locks on *accessed* data items
2) Execute transaction
3) Release exclusive locks

- Greater concurrency

- Problems?

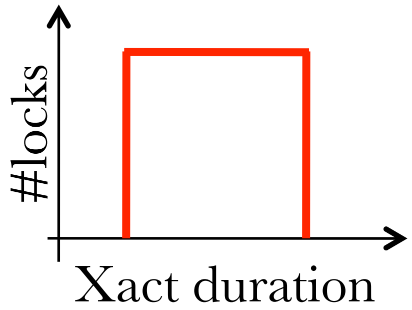


Source: Ramakrishnan & Gehrke (partial)

Solution 3

1) Get exclusive locks on data items that are *modified*; get shared locks on data items that are only *read*
2) Execute transaction
3) Release all locks

- Greater concurrency
- Conservative Strict Two Phase Locking (2PL)
- Problems?

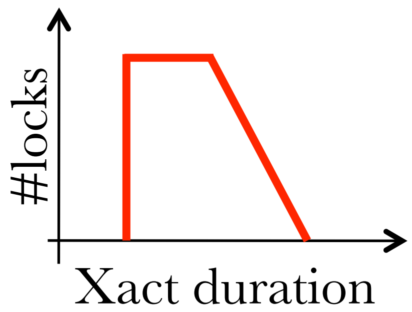


	S	X
S	Yes	No
X	No	No

Solution 4

1) Get exclusive locks on data items that are modified and get shared locks on data items that are read
2) Execute transaction and release locks on objects no longer needed *during execution*

- Greater concurrency
- Conservative Two Phase Locking (2PL)
- Problems?



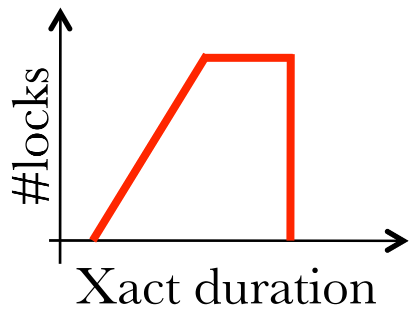
	S	X
S	Yes	No
X	No	No

Source: Ramakrishnan & Gehrke (partial)

Solution 5

1) Get exclusive locks on data items that are modified and get shared locks on data items that are read, but do this *during execution* of transaction (as needed)
2) Release all locks

- Greater concurrency
- Strict Two Phase Locking (2PL)
- Problems?



	S	X
S	Yes	No
X	No	No

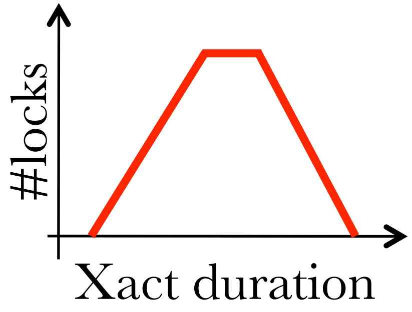
Source: Ramakrishnan & Gehrke (partial)

Solution 6

1) Get exclusive locks on data items that are modified and get shared locks on data items that are read, but do this during execution of transaction (as needed)
2) Release locks on objects no longer needed during execution of transaction

3) Cannot acquire locks once any lock has been released

- ***Hence two-phase (acquiring phase and releasing phase)***



- Greater concurrency
- Two Phase Locking (2PL)

- Problems?

Source: Ramakrishnan & Gehrke (partial)

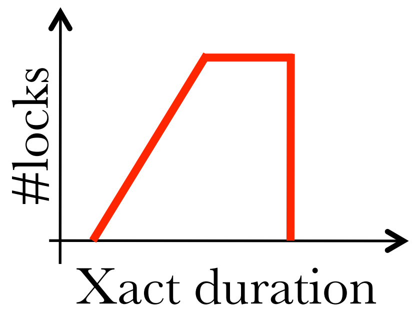
Summary of Alternatives

- Conservative Strict 2PL
  - No deadlocks, no cascading aborts
  - But need to know objects a priori, least concurrency
- Conservative 2PL
  - No deadlocks, more concurrency than Conservative Strict 2PL
  - But need to know objects a priori, when to release locks, cascading aborts
- Strict 2PL
  - No cascading aborts, no need to know objects a priori or when to release locks, more concurrency than Conservative Strict 2PL
  - But deadlocks
- 2PL
  - Most concurrency, no need to know object a priori
  - But need to know when to release locks, cascading aborts, deadlocks

Source: Ramakrishnan & Gehrke (partial)

Method of Choice

- Strict 2PL
  - No cascading aborts, no need to know objects a priori or when to release locks, more concurrency than Conservative Strict 2PL
  - But deadlocks
- Reason for choice
  - Cannot know objects a priori, so no Conservative options
  - Thus only 2PL and Strict 2PL left
  - 2PL needs to know when to release locks (main problem), and has cascading aborts
  - Hence Strict 2PL
- Implication
  - Need to deal with deadlocks!



	S	X
S	Yes	No
X	No	No

Lock Management

- Lock/unlock requests handled by lock manager
- Lock table entry:
  - Number of transactions currently holding a lock
  - Type of lock held (shared or exclusive)
  - Pointer to queue of lock requests
- Locking and unlocking have to be atomic operations
- **Lock upgrade**: transaction that holds a shared lock can be upgraded to hold an exclusive lock

Source: Ramakrishnan & Gehrke (partial)

Questions so far?

Is Strict 2PL correct?

- We will formalize now **serializability** and argue that Strict 2PL is correct
  - Full proof is left as homework ☺

- Strict 2PL can however deadlock
  - We will see how to handle deadlock automatically

Schedules

- Consider a possible interleaving (*schedule*):

```
T1:    A=A+100,                              B=B-100
T2:                 A=1.06*A, B=1.06*B
```

- The system's view of the schedule:

```
T1:    R(A),W(A),                            R(B),W(B)
T2:                 R(A),W(A),R(B),W(B)
```

Scheduling Transactions

- *Serial schedule:* Schedule that does not interleave the actions of different transactions.
- *Equivalent schedules*: For any database state
  - The effect (on the set of objects in the database) of executing the schedules is the same
  - The values read by transactions is the same in the schedules
    - Assume no knowledge of transaction logic
- *Serializable schedule*: A schedule that is equivalent to some serial execution of the transactions.

(Note: If each transaction preserves consistency, every serializable schedule preserves consistency.)

Source: Ramakrishnan & Gehrke (partial)

Anomalies with Interleaved Execution

- Reading Uncommitted Data (WR Conflicts, "dirty reads"):

```
T1:   R(A), W(A),                        R(B), W(B), Abort
T2:                R(A), W(A), C
```

- Unrepeatable Reads (RW Conflicts):

```
T1:   R(A),                              R(A), W(A), C
T2:             R(A), W(A), C
```

Anomalies (contd.)

- Overwriting Uncommitted Data (WW Conflicts):

```
T1:   W(A),                    W(B), C
T2:         W(A), W(B), C
```

Source: Ramakrishnan & Gehrke (partial)

Conflict Serializable Schedules

- Two schedules are conflict equivalent if:
  - Involve the same actions of the same transactions
  - Every pair of conflicting actions is ordered the same way

- Schedule S is conflict serializable if S is conflict equivalent to some serial schedule

Example

- A schedule that is not conflict serializable:

```
T1:     R(A), W(A),                                 R(B), W(B)
T2:                    R(A), W(A), R(B), W(B)
```

Precedence graph

- The cycle in the graph reveals the problem. The output of T1 depends on T2, and vice-versa.

Source: Ramakrishnan & Gehrke (partial)

Precedence Graph

- *Precedence graph*: One node per Xact; edge from *Ti* to *Tj* if operation in T*j* conflicts with earlier operation in T*i.*
- Theorem: Schedule is conflict serializable if and only if its precedence graph is acyclic
- Strict 2PL only allows conflict serializable schedule
  - Precedence graph is always acyclic

Are the following schedules conflict-serializable?

- Build the precedence graph for each of the following transaction schedules

```
T1: R(A)                W(B)                           C
T2:         R(B)                R(A)             R(C)     C
T3:                 R(B)                W(C)                   C
```

Note: C stands for commit

```
T1: R(A)        W(B)                                   C
T2:         R(B)                R(A)             R(C)     C
T3:                     R(B)             W(C)                 C
```

Returning to Definition of Serializability

- A schedule S is serializable if there exists a serial order SO such that:
  - The **state of the database** after S is the **same** as the state of the database after SO
  - The **values read** by each transaction in S are the **same** as that returned by each transaction in SO
    - Database does not know anything about the internal structure of the transaction programs
- **Under this definition, certain serializable executions are not conflict serializable!**

Are these schedules serializable?

```
T1: R(A)          W(A)
T2:          W(A)
T3:                       W(A)
```

```
T1: R(A),W(A)
T2:                    W(A)
T3:                            W(A)
```

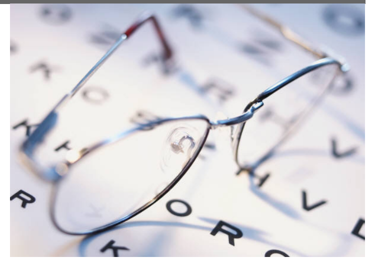
Source: Ramakrishnan & Gehrke (partial)

View Serializability

- Schedules S1 and S2 are view equivalent if:
  - If Ti reads initial value of A in S1, then Ti also reads initial value of A in S2
  - If Ti reads value of A written by Tj in S1, then Ti also reads value of A written by Tj in S2
  - If Ti writes final value of A in S1, then Ti also writes final value of A in S2

```
T1: R(A)        W(A)
T2:        W(A)
T3:                   W(A)
```

```
T1: R(A),W(A)
T2:                W(A)
T3:                      W(A)
```



What should we learn today?

- Identify the multiple interpretations of the property of atomicity
- Implement methods to ensure before-or-after atomicity, and argue for their correctness
- Explain the variants of the two-phase locking (2PL) protocol, in particular the widely-used Strict 2PL
- Discuss definitions of serializability and their implications, in particular conflict-serializability and view-serializability
- Apply the conflict-serializability test using a precedence graph to transaction schedules