Concurrency Control and Recovery

Oliver Flatt and Winston Jodjana

DBMS Overview

DataBase Management System

Goals

1. Protect the data in the database
2. Query of the data correctly despite **concurrent** access and **failures**

Core components to achieve these goals

1. Concurrency Control
2. Recovery

Transactions

Ratul covered this.

- Main principal: A~~C~~ID
  - Atomicity - "All or nothing"
  - ~~Consistency - Any changes maintains constraints~~
  - Isolation - Each transaction if they ran alone
  - Durability - After COMMIT, survives failures

Example of a Transaction

```
BEGIN

UPDATE accounts SET balance = balance - 50 WHERE id = 1;

UPDATE accounts SET balance = balance + 50 WHERE id = 2;

COMMIT;
```

1. Concurrency Control (i.e. the "I")

Serializability - a schedule is said to be serializable if and only if it is equivalent to some **serial schedule**

Equivalance - same operations, same ordering of conflicting operations

Serial schedule - operations **appear** consecutively

Provides guaranteed complete **Isolation**

Expensive and difficult in practice

1. Concurrency Control (cont.)

Conflict Serializability - a schedule is said to be conflict serializable if and only if it can be transformed into a serial schedule by swapping non-**conflict**ing operations.

Conflict operations - 2 inter-transaction operations that operate on the same data where one of them is a write operation

Implies **Serializability**

Simpler and cheaper

(e.g. Cycle in precedence graph = abort and retry)

1.1 Pessimistic Solution: Two Phase Locking (2PL)

Locks are a common concurrency control strategy

Two types of locks - Shared and Exclusive

(For reads) (For writes)

In 2PL, in every transaction, all acquires must happen before release

Acquire Phase -> Release Phase

(Growing) (Shrinking)

2PL guarantees **conflict serializability**

1.2 2PL Deadlocks

When different transactions block on each other and can’t make progress

Deadlock = Cycle in precedence graph on lock

1. Periodically run cycle detection
2. If cycle detected,
    a. Rollback transaction(s)
    b. (Hopefully) make progress
    c. Retry the rolled back transaction(s)

2PL also has problems with recoverability -> Solution is Strict 2PL

1.3 Common Concurrency Control Problems

- Non-Atomic Operations
- Lost Update
- Dirty/Inconsistent Read
- Unrepeatable Read
- Phantom Read

1.3 Example: Phantom Read

Same query produces different (unexpected) results within a transaction

e.g.

Thread 1	**Thread 2**
	`read(x) # 100`
`update(x, 200)`	
	`read(x) # 200`

Violates **Isolation**

1.4 Isolation Levels

READ UNCOMMITED

READ COMMITED

REPEATABLE READ

SERIALIZABLE

SNAPSHOT ISOLATION (MVCC)

Database	Default Isolation	Maximum Isolation
Actian Ingres 10.0/10S	S	S
Aerospike	RC	RC
Akiban Persistit	SI	SI
Clustrix CLX 4100	RR	?
Greenplum 4.1	RC	S
IBM DB2 10 for z/OS	CS	S
IBM Informix 11.50	Depends	RR
MySQL 5.6	RR	S
MemSQL 1b	RC	RC
MS SQL Server 2012	RC	S
NuoDB	CR	CR
Oracle 11g	RC	SI
Oracle Berkeley DB	S	S
Oracle Berkeley DB JE	RR	S
Postgres 9.2.2	RC	S
SAP HANA	RC	SI
ScaleDB 1.02	RC	RC
VoltDB	S	S
Legend	*RC: read committed, RR: repeatable read, S: serializability, SI: snapshot isolation, CS: cursor stability, CR: consistent read*	

1.5 Optimistic Concurrency Control

- Assume schedule is serializable
- If conflict, abort
- Workloads with low levels of contention

Some Solutions

- Timestamping
  - The timestamp order defines the serialization order of the transaction
- Validation
  - Read Phase, Validate Phase, Write Phase
- **Snapshot Isolation**
  - Combines techniques: Timestamps, Multiversion, Validation

2. Fault Tolerance: Recovery (i.e. the "A" and "D")

How to deal with crashes that can happen at any moment? How about corrupted data?

Two important operations:

- Undo
  - Removing the effects of an incomplete or aborted transaction
- Redo
  - Reinstating the effects of a committed transaction

ARIES: Page-oriented REDO-logical UNDO log

Strategies for Recovery

	STEAL- allowed to mutate storage during transaction	NO-STEAL- not allowed to mutate actual storage until transaction finishes
NO-FORCE- don’t need to finish writing transaction to commit		
FORCE- ensure all updates are in storage before commit		

Strategies for Recovery

	STEAL- allowed to mutate storage during transaction	NO-STEAL- not allowed to mutate actual storage until transaction finishes
NO-FORCE- don’t need to finish writing transaction to commit	**Good performance**, difficult recovery	Write all of the transaction after it’s been committed
FORCE- ensure all updates are in storage before commit	Write all the transactions to storage, then commit	**Bad performance**, easy recovery Need detailed **log**

Logging

Data page:

Data1 = 0
Data2 = 3
Data3 = 10
Log number -> 0

Log:

Logging

Data page:

Data1 = 0
Data2 = 3
Data3 = 11 - updated!
Log number -> 4

Log:

Log Sequence Number: 4
Updated Data3 which had value 10

What needs to be logged?

For STEAL?

For NO-FORCE?

What needs to be logged?

For STEAL?

- The previous values for anything overwritten

For NO-FORCE?

What needs to be logged?

For STEAL?

- The previous values for anything overwritten

For NO-FORCE?

- The new values that still need to be written

What needs to be logged?

For STEAL?

- The previous values for anything overwritten

For NO-FORCE?

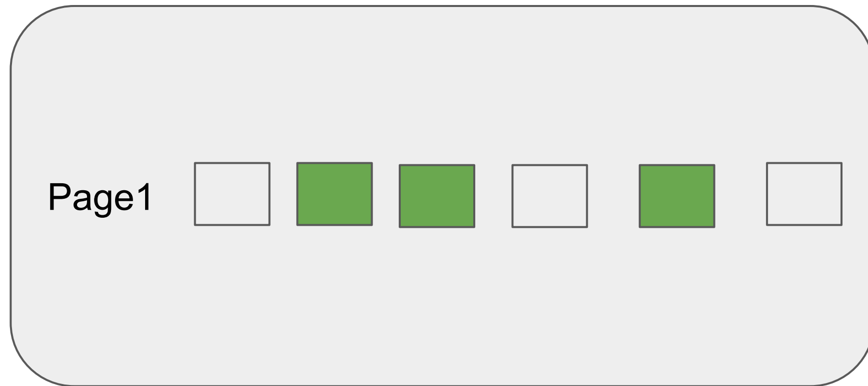
- The new values that still need to be written

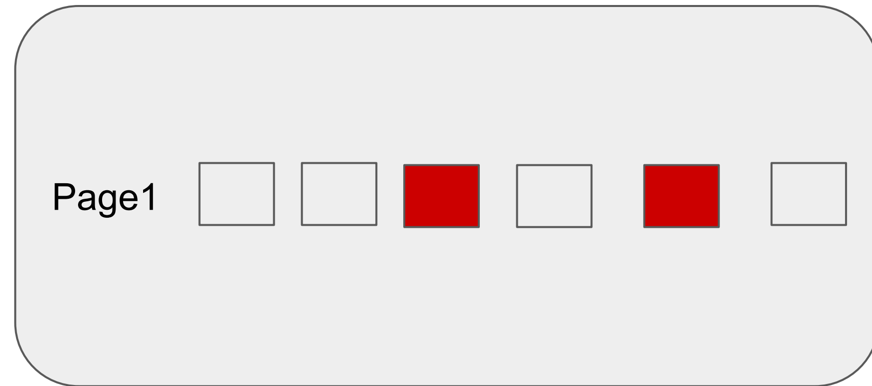
Logical logging involves logging a more complex operation that requires multiple low-level writes

Example: Insert a tuple into a database, requiring updating the database index among other operations

Hierarchical locking

How many locks do I need?



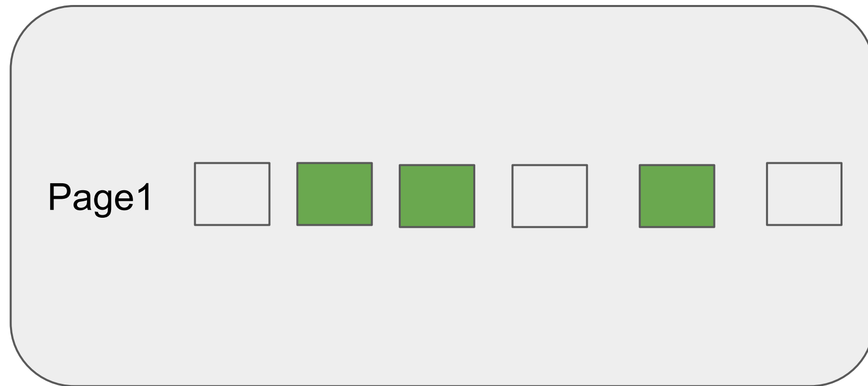


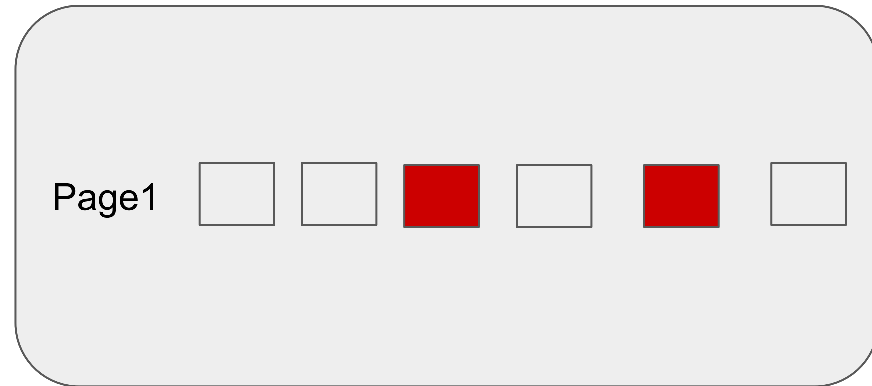
Hierarchical locking

How many locks do I need?

3 locks for the green transaction, 2 locks for the red transaction

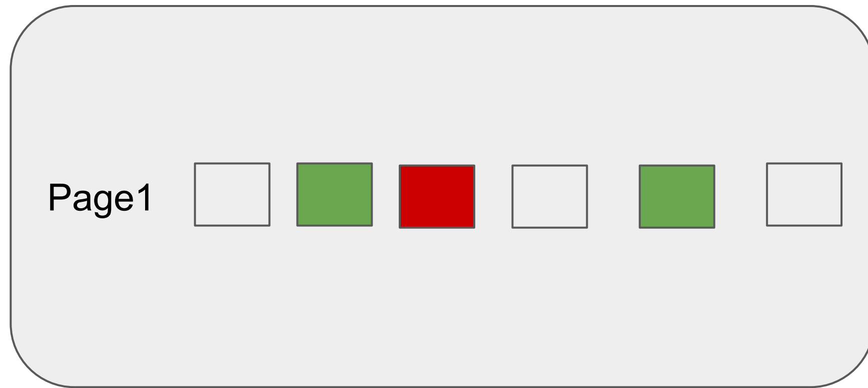
All through the lock manager!

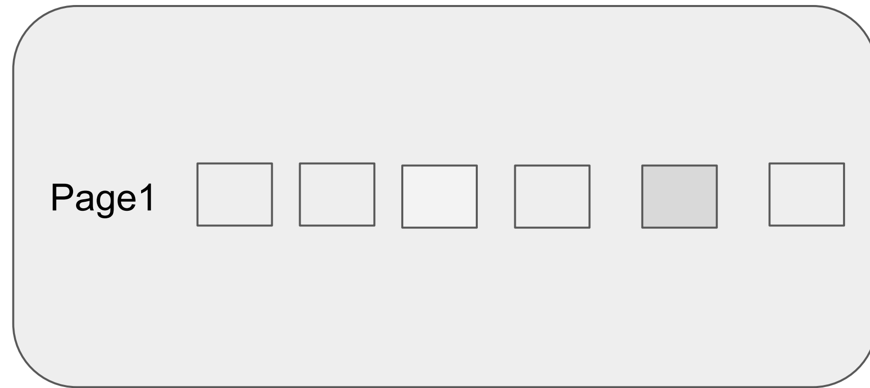




Hierarchical locking

Could lock the whole page, but that's worse in this case:





Hierarchical Locking

- Specify *intent* on what you want to do to a page
  - Read page, write page, or acquire more fine grained permissions
- Can perform runtime analysis to determine a policy
  - Ex: Transaction 3 always acquires a lot of locks in this page, so just give it a lock on the entire page

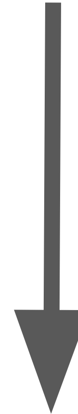
Degrees of Isolation

READ UNCOMMITTED- No guarantee about reads, doesn't even get read locks!

READ COMMITTED- Guarantees that values read are from committed transactions

REPEATABLE READ- Reading data items will result in the same value during the transactions (holds read locks for **long** duration of transaction)

SERIALIZABLE- All locks are long duration, also solves **phantom problem**



Stronger Guarantee

READ COMMITTED is commonly used

But what does this mean for your actually running program?

“Despite the ubiquity of weak isolation, I haven’t found a database architect, researcher, or user who’s been able to offer an explanation of when, and, probably more importantly, why isolation models such as Read Committed are sufficient for correct execution. ... I don’t think we have any real understanding of how so many applications are seemingly (!?) okay running under them.”

http://www.bailis.org/blog/understanding-weak-isolation-is-a-serious-problem/