CS848 Management of Information Systems
Fall 2006

Ken Salem

David R. Cheriton School of Computer Science
University of Waterloo

Problems Caused by Failures

- Update all account balances at a bank branch.

```
Accounts(Anum, CId, BranchId, Balance)

UPDATE Accounts
SET Balance = Balance * 1.05
WHERE BranchId = 12345
```

Problem

If the system crashes while processing this update, some, but not all, tuples with `BranchId = 12345` may have been updated.

Another Failure-Related Problem

- transfer money between accounts:

```
UPDATE Accounts
SET Balance = Balance - 100
WHERE Anum = 8888

UPDATE Accounts
SET Balance = Balance + 100
WHERE Anum = 9999
```

Problem

If the system fails between these updates, money may be withdrawn but not redeposited

Problems Caused by Concurrency

- Application 1:

```
UPDATE Accounts
SET Balance = Balance - 100
WHERE Anum = 8888

UPDATE Accounts
SET Balance = Balance + 100
WHERE Anum = 9999
```

- Application 2:

```
SELECT Sum(Balance)
FROM Accounts
```

Problem

If the applications run concurrently, the total balance returned to Application 2 may be inaccurate.

Another Concurrency Problem

- Application 1:

```
SELECT balance INTO :balance
FROM Accounts
WHERE Anum = 8888
compute :newbalance using :balance
UPDATE Accounts
SET Balance = :newbalance
WHERE Anum = 8888
```

- Application 2: same as Application 1

Problem

If the applications run concurrently, one of the updates may be “lost”.

Transaction Properties

- Transactions are *durable*, *atomic* application-specified units of work.

Atomic:	indivisible, all-or-nothing.
Durable:	effects survive failures.

"ACID" Properties of Transactions

A	tomic: a transaction occurs entirely, or not at all
C	onsistent
I	solated: a transaction's unfinished changes are not visible to others
D	urable: once it is complete, a transaction's changes are permanent

Abort and Commit

- A transaction may terminate in one of two ways:

commit:	When a transaction *commits*, any updates it made become durable, and they become visible to other transactions. A commit is the “all” in “all-or-nothing” execution.
abort:	When a transaction *aborts*, any updates it may have made are undone (erased), as if the transaction never ran at all. An abort is the “nothing” in “all-or-nothing” execution.

- A transaction that has started but has not yet aborted or committed is said to be *active*.

Serializability (informal)

- Concurrent transactions must appear to have been executed sequentially, i.e., one at a time, in some order. If T_i and T_j are concurrent transactions, then either:
  - T_i will appear to precede T_j , meaning that T_j will “see” any updates made by T_i , and T_i will not see any updates made by T_j , or
  - T_i will appear to follow T_j , meaning that T_i will see T_j’s updates and T_j will not see T_i’s.

Serializability: An Example

- An serial execution of two transactions, T_1 and T_2:

$$H_b = \ w_1[x]\, w_1[y]\, r_2[x]\, r_2[y]$$

- An equivalent interleaved execution of T_1 and T_2:

$$H_a = w_1[x]\, r_2[x]\, w_1[y]\, r_2[y]$$

- An interleaved execution of T_1 and T_2 with no equivalent serial execution:

$$H_c = \ w_1[x]\, r_2[x]\, r_2[y]\, w_1[y]$$

H_b is serializable because it is equivalent to H_a , a serial schedule. H_c is not serializable.

Serialization Graphs

The serialization graph $SG(H)$ of a complete execution history H is the directed graph with

- one node for each committed transaction in H
- an directed edge from T_i to T_j iff there are operations $o_i[x]$ and $o_j[x]$ in H such that $o_i[x]$ precedes and conflicts with $o_j[x]$.

Theorem

H is serializable iff $SG(H)$ is acyclic.

Two-Phase Locking

- The rules
  1. Before a transaction may read or write an object, it must have a lock on that object.
     - a *shared lock* is required to read an object
     - an *exclusive lock* is required to write an object
  2. Two or more transactions may not hold locks on the same object unless all hold shared locks.
  3. Once a transaction has released (unlocked) any object, it may not obtain any new locks. (In strict two-phase locking, locks are held until the transaction commits or aborts.)

Theorem

If all transactions use two-phase locking, the resulting execution history will be serializable.

Snapshot Isolation (informal)

- each transaction T has a start time ($start(T)$) and a commit time ($commit(T)$) - unless it aborts.
- each transacation T “sees” a snapshot of the database that include all updates of transactions that commit before $start(T)$ and no updates of transactions that commit after $start(T)$, except . . .
- . . . that T sees its own updates.
- If two transactions T_i and T_j are concurrent, then T_i and T_j are not permitted to update the same object.

Properties of SI

SI provides each transaction with a consistent view of the database, and avoids “lost updates”.

One-Copy Serializability (1SR)

- A 1SR history:

  $H_a = r_2[x_0]\, r_2[y_0]\, r_1[x_0]\, w_2[y_2]\, c_2\, r_1[y_2]\, w_1[x_1]\, c_1\, r_3[x_1]\, r_3[y_2]\, c_3$
- An SI (1SR?) history:

  $H_b = r_1[x_0]\, r_1[y_0]\, r_2[x_0]\, r_2[y_0]\, w_1[x_1]\, c_1\, r_3[x_1]\, r_3[y_0]\, c_3\, w_2[y_2]\, c_2$

1SR vs. SI

- 1SR $\Rightarrow$ SI
- SI $\not\Rightarrow$ 1SR

Freshness and Strong Serializability

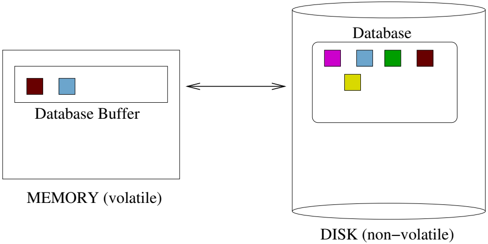
- Consider the following history:

  $H_c = r_2[x_0]\, r_2[y_0]\, r_1[x_0]\, w_2[y_2]\, c_2\, r_3[x_0]\, r_3[y_0]\, c_3\, r_1[y_2]\, w_1[x_1]\, c_1$
- H_c is 1SR, but T_3 sees a stale copy of y (y_0 instead of y_2)
- An execution history H is strongly serializable (strongly 1SR) if H is serializable (1SR) and all pairs of non-concurrent transactions in H can be serialized in the order in which they execute.
- H_c is not strongly 1SR, because T_3 follows T_2 but is serialized before T_2.

Freshness vs. Consistency

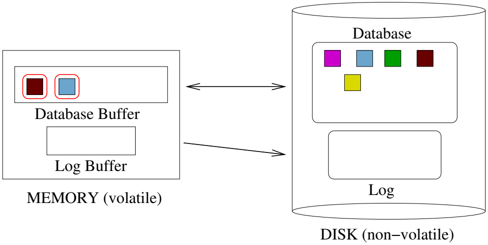
Freshness and consistency are orthogonal issues.

Aborts and Failures



- If a transaction aborts, any changes that it made must be eliminated from volatile and non-volatile storage.
- A system failure destroys the contents of volatile memory.
- To recover, the system must ensure that:
  - committed changes are reflected in non-volatile memory
  - aborted changes are not reflected in non-volatile memory
  - active transactions are either resumed or cleanly aborted.

Committing Transactions



- Suppose individual blocks can be written atomically from volatile memory to non-volatile disk.
- Suppose a transaction T updates two blocks. How to commit T safely?
- Common solution: use a log

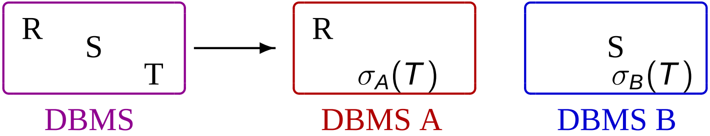
Write-Ahead Logging

- Before performing an update, write redo and undo information into the log.
- Before writing a data page from volatile to non-volatile storage, ensure that all log entries for that page are non-volatile.
- To commit a transaction, write a commit record into the log. Ensure that the commit record is non-volatile before acknowledging the commit to the application.

Atomic Commitment

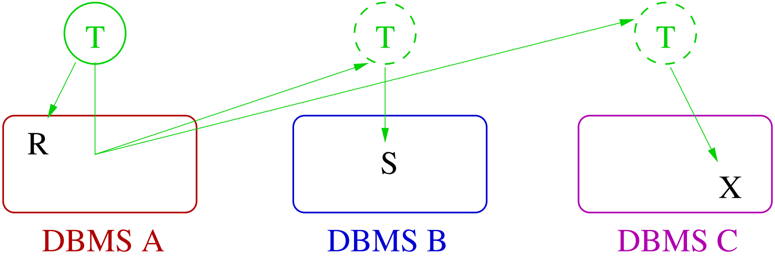
Moving the commit record into non-volatile storage commits the transaction.

Data Partitioning



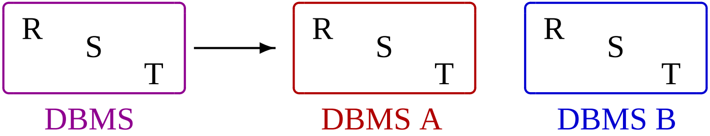
- transactions may span sites (distributed queries, distributed transactions)
- physical design: which data at each site?
- adding/removing sites involves data redistribution

Two Phase Commit



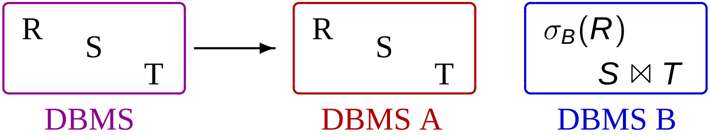
1. `UPDATE R`
2. `UPDATE S`
3. `UPDATE X`
4. `COMMIT`
    - 2PC phase 1
    - 2PC phase 2

Data Replication



- transactions execute at one site (possible exception: synchronization)
- synchronization: how to keep copies consistent?
- replicas are redundant, require extra space
- simple (though expensive) to add sites, simple to remove sites

Materialized Views and Query Caching



- transactions execute at one or two sites
- synchronization: how to keep views consistent
- matierialized views are redundant, require extra space
- simple to add/remove sites

Replication Techniques: Eager [GHOS96]

- to read R, read local replica of R
- to update R, update all replicas of R
- global concurrency control
  - each local site has a local concurrency controller which locks local replicas
  - global (multi-site) update transactions consist of sub-transactions at each site
- use 2PC to atomically commit transaction updates

Global Serializability

Local strict two-phase locking $+$ 2PC for commit coordination is sufficient to ensure global 1SR.

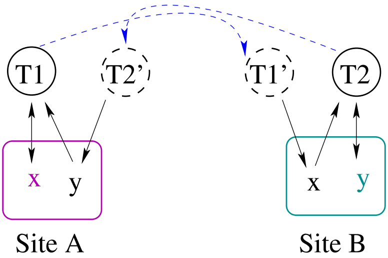
Replication Techniques: Lazy/Group [GHOS96]

- to read R, read local replica of R
- to update R, update local replica of R
- propagate updates lazily to other sites, where they are applied by separate local transactions at those sites
- Problems:
  - no guarantee of 1SR
  - conflicting updates
  - (manual) reconciliation of resulting inconsistencies

Replication Techniques: Lazy/Master [GHOS96]

- to read *R*, read local replica of *R*
- to update *R*, update master replica of *R*
- propagate updates lazily to other replicas, where they are applied by separate local transactions at those sites
- Problems:
  - no guarantee of 1SR
  - (manual) reconciliation of resulting inconsistencies

Lazy/Master Example



- Site *A* has master copy of *x*, replica of *y*
- Site *B* has master copy of *y*, replica of *x*
- Transaction T_1 at site *A*: $r_1[x]\, r_1[y]\, w_1[x]\, c_1$
- Transaction T_2 at site *B*: $r_2[x]\, r_2[y]\, w_2[y]\, c_2$
- Propagation transaction T_1' for T_1 (at site *B*): $w_1[x]\, c_1$
- Propagation transaction T_2' for T_2 (at site *A*): $w_2[y]\, c_2$
- Global execution is not 1SR.

Views

```
Books (BookId, Title, Author, Subject, Year)
Holdings (BookId, LibraryId)

CREATE VIEW CSBooks AS
 SELECT * FROM Books WHERE Subject = ’CS’

CREATE VIEW UWHoldings AS
 SELECT Title FROM Books B, Holdings H
 WHERE B.BookId = H.BookId AND
       LibraryId = ’UW’
```

Views

Views are named queries that can be used much like regular tables.

Materialized Views

- materialized views are views for which the result of the underlying query has been computed and stored
- issues:

transparency: is the application aware of the materialized views?

synchronization: what happens to the stored query result when the underlying database is updated?

Full replication is a special case of view materialization.

Transparency

```
Books (BookId, Title, Author, Subject, Year)

CREATE VIEW CSBooks AS
 SELECT * FROM Books WHERE Subject = ’CS’
```

- non-transparent:

```
SELECT * FROM CSBooks
```

- transparent:

```
SELECT Title FROM Books
WHERE Topic = ’CS’ AND Year = 2006
```

View Matching Problem

Is a given view relevant to a given query?

Views and Updates

```
CREATE VIEW CSBooks AS
 SELECT * FROM Books WHERE Subject = ’CS’

CREATE VIEW UWHoldings AS
 SELECT Title FROM Books B, Holdings H
 WHERE B.BookId = H.BookId AND LibraryId = ’UW’
```

- Changes (INSERT, DELETE, UPDATE) to Books may change the result of the query that defines CSBooks.
- Changes to Holdings may change the result of the query that defines UWHoldings.

Update Relevance

An update is relevant to a view if that update could change the result of the view’s underlying query.

Synchronization

timing: when relevant updates occur, when is the materialized view updated?

- immediate: view is updated within the transaction that updates the underlying table
- deferred: view updated occurs after the underlying table is updated

mechanism: how is the materialized view updated?

- full refresh: recompute the view after the underlying table is updated
- incremental refresh: compute the view changes that result from the update, and apply them to the old materialized view

Incremental Refresh

```
Books (BookId, Title, Author, Subject, Year)

CREATE VIEW CSBooks AS
 SELECT * FROM Books WHERE Subject = ’CS’
```

Suppose tuple *t* is inserted into `Books`. Incremental maintenance of `CSBooks` involves:

1. test whether *t*.`Subject = ’CS’`
2. if so, insert *t* into `CSBooks`

Incremental Refresh (cont'd)

```
Books (BookId, Title, Author, Subject, Year)
Holdings (BookId, LibraryId)
CREATE VIEW UWHoldings AS
 SELECT Title FROM Books B, Holdings H
 WHERE B.BookId = H.BookId AND LibraryId = ’UW’
```

Suppose tuple *t* is inserted into `Holdings`. Incremental maintenance of `UWHoldings` involves:

1. test whether *t*.`LibraryId = ’UW’`
2. join *t* with `Books` on *t*.`BookId = Books.BookId`
3. insert the resulting `Title` into `UWHOldings`

Self-Maintainability

`UWHoldings` is not self-maintainable wrt inserts into `Holdings`.

Query Caching

- materialize query results and use them to answer subsequent queries more quickly
- like view materialization:
  - dynamic set of materialized queries
  - transparent to applications
    - exact matching based on query text
    - more general or partial matching
  - sychronization
    - incremental refresh
    - invalidation

Bibliography

Jim Gray, Pat Helland, Patrick E. O'Neil, and Dennis Shasha.
The dangers of replication and a solution.
In *Proc. ACM SIGMOD International Conference on Management of Data (SIGMOD'96)*, pages 173–182, 1996.

Paper Presentations and Reviews

- Presentations
  - 25 minutes
  - focus on the most interesting and significant material: no need to be comprehensive
  - try to put the material in context: how does it relate to papers or lecture topics covered in this course
  - try to raise issues for discussion
- Reviews
  - pretend you are reviewing for a conference
  - follow the instructions on the review form, including length limits