Transactions and their Properties

Olaf Hartig

David R. Cheriton School of Computer Science
University of Waterloo

CS 640
Principles of Database Management and Use
Winter 2013

These slides are based on slide sets
provided by M. T. Öszu and
by R. Ramakrishnan and J. Gehrke

Notes

Outline

Notes

Why We Need Transaction Management

- A database is a shared resource accessed by many users and processes concurrently.
  - Both queries and modifications
- Not managing this concurrent access to a shared resource will cause problems (not unlike in operating systems)
  - Problems due to concurrency
  - Problems due to failures

Notes

Problems Caused by Concurrency

```
Accounts(AccountNumber, CustID, BranchID, Balance)
```

- Application 1: You are depositing money to your bank account.

```
update Accounts
set Balance = Balance + 100
where AccountNumber = 9999
```

- Application 2: The branch is calculating the balance of the accounts.

```
select Sum(Balance)
from Accounts
```

Problem – Inconsistent reads

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

Notes

Another Concurrency Problem

- Application 1: You are depositing money to your bank account at an ATM.

```
update Accounts
set Balance = Balance + 100
where AccountNumber = 9999
```

- Application 2: Your partner is withdrawing money from the same account at another ATM.

```
update Accounts
set Balance = Balance - 50
where AccountNumber = 9999
```

Problem – Lost Updates

If the applications run concurrently, one of the updates may be "lost", and the database may be inconsistent.

Notes

Yet Another Concurrency Problem

- Application 1:

```
update Employee
set Salary = Salary + 1000
where WorkDept = 'D11'
```

- Application 2:

```
select * from Employee
where WorkDept = 'D11'

select * from Employee
where Lastname like 'A%'
```

Problem – Non-Repeatable Reads

If there are employees in D11 with surnames that begin with "A", Application 2's queries may see them with different salaries.

Notes

High-Level Lesson

We need to worry about interaction between two applications when

- one reads from the database while the other writes to (modifies) the database;
- both write to (modify) the database.

We do not worry about interaction between two applications when both only read from the database.

Notes

Problems Caused by Failures

- Update all account balances at a bank branch.

```
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` (i.e., some account balances) may have been updated.

Problem

If the system crashes after this update is processed but before all of the changes are made permanent (updates may be happening in the buffer), the changes may not survive.

Notes

Another Failure-Related Problem

- transfer money between accounts:

```
update Accounts
set Balance = Balance - 100
where AccountNumber = 8888

update Accounts
set Balance = Balance + 100
where AccountNumber = 9999
```

Problem

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

Notes

High-Level Lesson

We need to worry about partial results of applications on the database when a crash occurs.

We need to make sure that when applications are completed their changes to the database survive crashes.

Notes

Transactions

Definition (Transaction)
An application-specified *atomic* and *durable* unit of work (a *process*).

- Concurrency transparency
- Failure transparency

Notes

Transaction Termination

COMMIT: Any updates a transaction has made become permanent and visible to other transactions. Before COMMIT, changes are tentative.

ABORT: Any updates a transaction may have made are undone (erased), as if the transaction never ran at all.

ABORTED BY SYSTEM: Same effect as ABORT by application.

- Happens in case of problems that may only be detected by the DBMS (e.g. timeout, deadlock)

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

Notes

Transactions in SQL

- A new transaction is begun when an application first executes an SQL command.
- Two SQL commands are available to terminate a transaction:
  - **commit**: commits the transaction
  - **rollback**: abort the transaction
- A new transaction begins with the application's next SQL command after **commit** or **rollback**.

Notes

Example Transaction – Single Statement

The start of a new SQL expression (SELECT, UPDATE, INSERT, DELETE, CREATE) automatically starts a transaction – no explicit command required, but the termination needs to be specified.

```
SELECT *
FROM Employee
WHERE WorkDept = 'D11'
COMMIT
```

```
UPDATE Employee
SET Salary = Salary + 1000
WHERE WorkDept = 'D11'
COMMIT
```


Notes

Outline

Notes

Properties of Transactions

Atomic: a transaction occurs entirely, or not at all

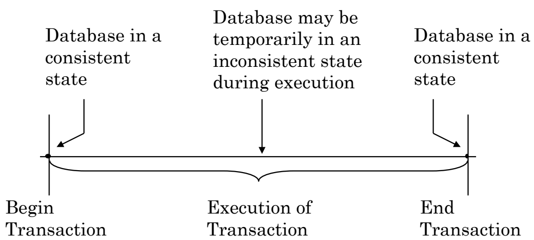
Consistency: each transaction preserves the consistency of the database

Isolated: concurrent transactions do not interfere with each other

Durable: once committed *successfully*, a transaction's changes are permanent

Notes

Consistency



Notes

How Do Transactions Help?

- Application 1: You are depositing money to your bank account at an ATM.

```
update Accounts
set Balance = Balance + 100
where AccountNumber = 9999
```

- Application 2: Your partner is withdrawing money from the same account at another ATM.

```
update Accounts
set Balance = Balance - 50
where AccountNumber = 9999
```

Isolation

If each of these applications run as a transaction, their effects would be isolated from each other – Application 2 can't see Application 1's update until Application 1 completes.

Notes

How Do Transactions Help?

- Update all account balances at a bank branch.

```
update Accounts
set Balance = Balance * 1.05
where BranchID = 12345
```

Atomicity

If the application runs as a transaction, either all the accounts will get updated or none of them will.

Notes

How do DBMSs Guarantee the ACID Properties?

Isolation: Concurrency control algorithms and techniques guarantee concurrent transactions do not interfere with each other and don't see each other's changes until they complete.

- Some sort of mutual exclusion is typically implemented (i.e., locking) but alternatives exist

⇒ Focus of our discussion next week (Gray et al., 1976)

Atomicity & Durability: Recovery management guarantees that committed transactions are durable (despite failures), and that aborted transactions have no effect on the database.

- DBMS logs every action securely so that it can consult the log later to determine what to do.

⇒ Focus of our discussion in two weeks (Haerder and Reuter, 1983)

Notes

Scheduling Transactions for Concurrency Control

Schedule: a sequence of actions from multiple TAs such that the sequence of each TA is preserved

Serial schedule: schedule that does not interleave the actions of different TAs (i.e. the TAs are scheduled one after another)

Equivalence of schedules: Schedules S_1 and S_2 are equivalent if, for any database state, executing S_1 has the same effect as executing S_2 (i.e. the resulting database state is the same).

Serializable schedule: A schedule S is serializable if there exists a serial schedule that is equivalent to S.

Notes

Strict Two-Phase Locking Protocol (Strict 2PL)

Rules:

1. Any TA must obtain a *shared lock* on an object before reading.
2. Any TA must obtain an *exclusive lock* on an object before writing.
3. All locks held by a TA are released at the end of the TA.
4. If a TA holds an exclusive lock on an object, no other TA can get a lock (shared or exclusive) on that object.

Theorem

Strict 2PL generates only serializable schedules.

Notes

More on Concurrency Control

... in our discussion next week

Read:

- J. Gray, R. A. Lorie, G. R. Putzolu, and I. L. Traiger: **Granularity of Locks and Degrees of Consistency in a Shared Data Base**. *IFIP Working Conference on Modelling in Data Base Management Systems* 1976.

Notes

Logging for Recovery

Idea

For all update operations, record REDO / UNDO information in a log.

Log: an ordered list of log records

Log record: represents a single REDO/UNDO action; contains:

- transaction ID
- page ID
- offset
- length
- old data
- new data
- additional control information

Notes

Write-Ahead Logging (WAL) Protocol

Rules:

1. Persist the log record for an update operation before the corresponding data page is written to disk
   - Guarantees atomicity
2. Persist all log records for (all update operations of) a transaction before reporting a successful commit
   - Guarantees durability

Notes

Recovering from a Crash

3 Phases (of the *Aries* recovery algorithm):

1. Analysis phase: Scan the log forward to identify:
   - all TAs that were active at the time of the crash; and
   - all "dirty" pages in the main memory page buffer (i.e. changed but not written) at the time of the crash.
2. Redo phase: Redoes all updates to dirty pages in the buffer (to ensure that all logged updates are in fact carried out and written to disk).
3. Undo phase: Undo the writes of all TAs that were active at the crash (by restoring the old value of any update, as stored in the corresponding log record), working backwards in the log.
   - Some care must be taken to handle the case of another crash that may occur during the recovery.

Notes

More on Recovery

... in our discussion in two weeks

Read:

- T. Haerder and A. Reuter: **Principles of Transaction-Oriented Database Recovery**. *ACM Computing Surveys 15*(4): 287-317 (1983).

Notes