Introduction to Transactions

David Toman

School of Computer Science
University of Waterloo

Database Implementation CS448

Basics of Transaction Processing

Query (and update) processing converts requests for *sets of tuples* to requests for reads and writes of physical objects in the database.

database objects (depending on granularity) can be

- individual attributes
- records
- physical pages
- files (only for concurrency control purposes)

Goals

⇒ correct and concurrent execution of queries and updates
⇒ guarantee that acknowledged updates are persistent

Basics of Transaction Processing

Query (and update) processing converts requests for *sets of tuples* to requests for reads and writes of physical objects in the database.

database objects (depending on granularity) can be

- individual attributes
- records
- physical pages
- files (only for concurrency control purposes)

Goals

$\Rightarrow$ correct and concurrent execution of queries and updates
$\Rightarrow$ guarantee that acknowledged updates are persistent

ACID Requirements

Transactions are said to have the ACID properties:

Atomicity: all-or-nothing execution

Consistency: execution preserves database integrity

Isolation: transactions execute independently (as if they were executed in the system alone)

Durability: updates made by a committed transaction will not be destroyed by subsequent failures.

Implementation of transactions in a DBMS comes in two parts:

- **Concurrency Control:** committed transactions do not interfere
- **Recovery Management:** committed transactions are durable, aborted transactions have no effect on the database

ACID Requirements

Transactions are said to have the ACID properties:

Atomicity: all-or-nothing execution

Consistency: execution preserves database integrity

Isolation: transactions execute independently (as if they were executed in the system alone)

Durability: updates made by a committed transaction will not be destroyed by subsequent failures.

Implementation of transactions in a DBMS comes in two parts:

- **Concurrency Control:** committed transactions do not interfere
- **Recovery Management:** committed transactions are durable, aborted transactions have no effect on the database

Concurrency Control: assumptions

1. we fix a database: a set of objects read/written by transactions:
   - ⇒ $r_i[x]$: transaction T_i reads object x
   - ⇒ $w_i[x]$: transaction T_i writes (modifies) object x
2. a transaction T_i is a sequence of operations

$$T_i = r_i[x_1], r_i[x_2], w_i[x_1], \ldots, r_i[x_4], w_i[x_2], c_i$$

c_i is the **commit request** of T_i.

3. for a **set of transactions** $T_1, \ldots, T_k$ we want to produce a *schedule* S of operations such that
   - ⇒ every operation $o_i \in T_i$ appears also in S
   - ⇒ T_i's operations in S are ordered the same way as in T_i

Goal:

produce a *correct schedule* with *maximal parallelism*

Concurrency Control: assumptions

1. we fix a database: a set of objects read/written by transactions:
   - $\Rightarrow$ $r_i[x]$: transaction T_i reads object x
   - $\Rightarrow$ $w_i[x]$: transaction T_i writes (modifies) object x
2. a transaction T_i is a sequence of operations

$$T_i = r_i[x_1], r_i[x_2], w_i[x_1], \ldots, r_i[x_4], w_i[x_2], c_i$$

c_i is the **commit request** of T_i.

3. for a **set of transactions** $T_1, \ldots, T_k$ we want to produce a *schedule* S of operations such that
   - $\Rightarrow$ every operation $o_i \in T_i$ appears also in S
   - $\Rightarrow$ T_i's operations in S are ordered the same way as in T_i

Goal:

produce a *correct schedule* with *maximal parallelism*

Transactions and Schedules

If T_i and T_j are concurrent transactions, then it is always correct to schedule the operations in such a way that:

- T_i will appear to precede T_j meaning that T_j will "see" all 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.

Idea how to define Correctness:

it must appear as if the transactions have been executed sequentially (in some *serial* order).

Serializable Schedules

Definition

An execution of is said to be **serializable** if it is equivalent to a serial execution of the same transactions.

Example:

- An interleaved execution of two transactions:

$$S_a = w_1[x]\ r_2[x]\ w_1[y]\ r_2[y]$$

- An equivalent serial execution (T_1 , T_2):

$$S_b = w_1[x]\ w_1[y]\ r_2[x]\ r_2[y]$$

- An interleaved execution with no equivalent serial execution:

$$S_c = w_1[x]\ r_2[x]\ r_2[y]\ w_1[y]$$

Serializable Schedules

Definition

An execution of is said to be **serializable** if it is equivalent to a serial execution of the same transactions.

Example:

- An interleaved execution of two transactions:

$$S_a = w_1[x]\ r_2[x]\ w_1[y]\ r_2[y]$$

- An equivalent serial execution (T_1 , T_2):

$$S_b = \ w_1[x]\ w_1[y]\ r_2[x]\ r_2[y]$$

- An interleaved execution with no equivalent serial execution:

$$S_c = \ w_1[x]\ r_2[x]\ r_2[y]\ w_1[y]$$

Conflict Equivalence

How do we determine if two schedules are *equivalent*?

$\Rightarrow$ cannot be based on any particular database instance

Conflict Equivalence:

- two operations *conflict* if they
  - (1) belong to different transactions
  - (2) access the same data item x
  - (3) at least one of them is a write operation $w[x]$.
- we require that in two *conflict-equivalent histories* all *conflicting operations* are ordered the same way.
- yields *conflict-serializable* schedules

  $\Rightarrow$ *conflict-equivalent* to a serial schedule

View Equivalence:

allows more schedules, but it is harder (NP-hard) to compute

Conflict Equivalence

How do we determine if two schedules are *equivalent*?

$\Rightarrow$ cannot be based on any particular database instance

Conflict Equivalence:

- two operations *conflict* if they
  (1) belong to different transactions
  (2) access the same data item x
  (3) at least one of them is a write operation $w[x]$.
- we require that in two *conflict-equivalent histories* all *conflicting operations* are ordered the same way.
- yields *conflict-serializable* schedules

  $\Rightarrow$ *conflict-equivalent* to a serial schedule

View Equivalence:

allows more schedules, but it is harder (NP-hard) to compute

Conflict Equivalence

How do we determine if two schedules are *equivalent*?

$\Rightarrow$ cannot be based on any particular database instance

Conflict Equivalence:

- two operations *conflict* if they
  (1) belong to different transactions
  (2) access the same data item x
  (3) at least one of them is a write operation $w[x]$.
- we require that in two *conflict-equivalent histories* all *conflicting operations* are ordered the same way.
- yields *conflict-serializable* schedules

  $\Rightarrow$ *conflict-equivalent* to a serial schedule

View Equivalence:

allows more schedules, but it is harder (NP-hard) to compute

Serialization Graph

How do we test if a schedule is conflict equivalent to a serial schedule?

- A **serialization graph** $SG(S)$ for a schedule S is a directed graph with nodes labeled by transactions such that

$$T_i \to T_j \in SG(S) \text{ iff } o_i[x] \text{ precedes } o_j[x] \text{ in } S$$

where $o_i[x]$ and $o_j[x]$ are conflicting operations.

Theorem:

A schedule S is serializable if and only if $SG(S)$ is acyclic graph.

Serialization Graph

How do we test if a schedule is conflict equivalent to a serial schedule?

- A **serialization graph** $SG(S)$ for a schedule S is a directed graph with nodes labeled by transactions such that

  $$T_i \to T_j \in SG(S) \text{ iff } o_i[x] \text{ precedes } o_j[x] \text{ in } S$$

  where $o_i[x]$ and $o_j[x]$ are conflicting operations.

Theorem:

A schedule S is serializable if and only if $SG(S)$ is acyclic graph.

Other Properties of Schedules

Serializability guarantees correctness. However, we'd like to avoid other **unpleasant** situations.

Recoverable Schedules: (RC)

transaction T_j *reads* a value T_i has written, T_j succeeds to **commit**, and T_i tries to abort (in this order)

$\Rightarrow$ to abort T_2 we need to *undo* effects of a *committed* transaction T_1.

$\Rightarrow$ commits only in order of the read-from dependency

Cascadeless Schedules (ACA):

if T_j above didn't commit we can abort it: may lead to *cascading aborts* of many transactions

$\Rightarrow$ no reading of uncommitted data

Other Properties of Schedules

Serializability guarantees correctness. However, we'd like to avoid other **unpleasant** situations.

Recoverable Schedules: (RC)

transaction T_j *reads* a value T_i has written, T_j succeeds to **commit**, and T_i tries to abort (in this order)

$\Rightarrow$ to abort T_2 we need to *undo* effects of a *committed* transaction T_1.

$\Rightarrow$ commits only in order of the read-from dependency

Cascadeless Schedules (ACA):

if T_j above didn't commit we can abort it: may lead to *cascading aborts* of many transactions

$\Rightarrow$ no reading of uncommitted data

Other Properties of Schedules

Serializability guarantees correctness. However, we'd like to avoid other **unpleasant** situations.

Recoverable Schedules: (RC)

transaction T_j *reads* a value T_i has written, T_j succeeds to **commit**, and T_i tries to abort (in this order)

$\Rightarrow$ to abort T_2 we need to *undo* effects of a *committed* transaction T_1.

$\Rightarrow$ commits only in order of the read-from dependency

Cascadeless Schedules (ACA):

if T_j above didn't commit we can abort it:
may lead to *cascading aborts* of many transactions

$\Rightarrow$ no reading of uncommitted data

How to Get a Serializable Schedule?

So how do we build schedulers that produce serializable and cascadeless schedules?

The **scheduler** receives requests from the query processor(s). For each operation it chooses one of the following actions:

- execute it (by sending to a lower module),
- delay it (by inserting in some queue), or
- reject it (thereby causing abort of the transaction)
- ignore it (as it has no effect)

Two main kinds of schedulers:

⇒ conservative (favors delaying operations)
⇒ aggressive (favors rejecting operations)

Summary

ACID properties of transactions guarantee correctness of concurrent access to the database and of data storage.

- consistency and isolation based on **serializability**

  $\Rightarrow$ leads to definition of correct **schedulers**

  $\Rightarrow$ responsibility of the **transaction manager**

- durability and atomicity

  $\Rightarrow$ responsibility of the **recovery manager**

  $\Rightarrow$ synchronous writing is too inefficient

  replaced by synchronous writes to a LOG and WAL

Summary

- many ways to implement a correct scheduler:

  $\Rightarrow$ conservative: locking (2PL)

  with deadlock prevention
  with deadlock detection

  $\Rightarrow$ aggressive: timestamps

  $\Rightarrow$ schedulers that *abort* transactions rely on the **recovery manager**

- additional issues:
  1. inserts and deletes?
  2. granularity of concurrency control?
  3. concurrency and data structures?
  4. multiple versions of data items?