Transaction Processing

Outline

- TranX Applications and their Characteristics
- Correctness issue: database consistency
- Simplified Model of TranX
- Desirable Properties of TranX: ACID
- Schedules, recoverability, Serializability
  - Test
  - usage

Basic TransX Concepts

- **Transaction** = a business deal (e.g. a sale)
  - logical unit of database processing that includes one or more access operations (read -retrieval, write - insert or update, delete).
    - stand-alone specified in a high level language like SQL submitted interactively
    - may be embedded within a program.
  - An **application program** may contain several transactions separated by the **Begin** and **End** transaction boundaries.

OLTP
(Online Transaction Processing)

- Transaction Processing = book-keeping about transactions
  - E-commerce: amazon.com, e-bay, ...
  - Airlines: researvation system
  - Banking: ATM, electronic fund transfer, Home banking
  - Securities: NY Stock Exchange, NASDAQ: 500 Million/day
  - ...Chicago board of trade (futures) - 200 Million/day
  - Telecommunications: call billing, operator services
  - Point-of-sale and Retail: credit cards/checks, sales
  - Manufacturing: Just-In-Time inventory control
  - Student Access System, SAS, (500,000 calls/week), ..

Characteristics

- Consistent on-line database
- ...despite failures, interleavings etc.
- Small work units - e.g. transfer, deposit, withdraw
- High throughput = # transactions per second (TPS)
- High Concurrency = # units overlapping a given unit

Eg. List work units, throughput and concurency for NYSE and SAS:

- NYSE: 200M/day, 5 sec. each :: > 2000 TPS, 10K overlaps/unit
- SAS: 100K/day, 1 minute each:: > 1 TPS, 50 overlaps each unit

Simplified Model of Transactions

- Transaction = execution of a user program accessing database
  - multiuser database, interleaved concurrent user programs
- Transaction: operations and states
  - 21.1.2 Database Operations: Eg. Fig. 21.2
    - ...read_item(X): read database item X into program variable X
    - ...write_item(X), begin-trans, ...
  - 21.2.1 Transaction states and other operations (Fig. 21.4)
    - ...States: active, partially committed, committed, failed, terminated
    - ...Ops: begin/end transaction, commit, abort (rollback), read/write

Need for Concurrency Control and Recovery

- 21.1.3 Why Concurrency Control is needed?
  - Problems due to interleaved concurrent transactions:
  - Lost update, Temporary update (Fig. 17.3, pp 614)
  - Unrepeatable read, incorrect summary, ...
- 21.1.4 Why Recovery is needed? (Avoid partial execution)
  - Goal: either all actions in a transaction are completed or has no effect on database or other transactions
  - Failure types: Catastrophe, system (HW / disk / OS),
  - Transaction error/exception, rollback due to concurrency control
- 21.2: Use system log, commit points

The System Log

- **Log or Journal :** The log keeps track of all transaction operations that affect the values of database items.
  - to permit recovery from transaction failures
  - kept on disk, not affected by any type of failure except for disk or catastrophic failure
  - In addition, the log is periodically backed up to archival storage (tape)
- Types of Log Records:
  1. [start_transaction,T]: Records that transaction T has started execution.
  2. [write_item,T,X,old_value,new_value]: Records that transaction T has changed the value of database item X from old_value to new_value.
  3. [read_item,T,X]: Records that transaction T has read the value of database item X.
  4. [commit,T]: Records that transaction T has completed successfully, and affirms that its effect can be committed (recorded permanently) to the database.
  5. [abort,T]: Records that transaction T has been aborted.

Recovery Using Log Records

- **undo:** Similar to rollback except that it applies to a single operation rather than to a whole transaction.
  - log contains a record of every *write* operation that changes the value of some database item,
  - **undo** the effect of these write operations of a transaction T by tracing backward through the log and resetting all items changed by a write operation of T to their old_values.
- **Redo:** This specifies that certain *transaction operations* must be *redone* to ensure that all the operations of a committed transaction have been applied successfully to the database.
  - **redo** the effect of the write operations of a transaction T by tracing forward through the log and setting all items changed by a write operation of T (that did not get done permanently) to their new_values

Commit Point of a Transaction

- **Commit Point**: when all transX's operations that access the database have been executed successfully and the effect of all transX operations on the database has been recorded in the log.
  - Beyond the commit point, the transaction is said to be committed, and its effect is assumed to be *permanently recorded* in the database. The transaction then writes an entry [commit,T] into the log.
- **Rollback** of transactions: needed for transactions that have a [start_transaction,T] entry into the log but no commit entry [commit,T] into the log.
- **Redoing** transactions: Transactions that have written their commit entry in the log must also have recorded all their write operations in the log; otherwise they would not be committed, so their effect on the database can be *redone* from the log entries.
- **Force writing** a log: Hence, *before* a transaction reaches its commit point, any portion of the log that has not been written to the disk yet must now be written to the disk. This process is called force-writing the log file before committing a transaction.

Characterizing Transactions and Schedules(21.3-21.4)

- Desirable Properties (**ACID**)
  - **Atomicity**: performed all actions or none
  - **Consistency** Preserving: moves database across consistent states
  - **Isolation**: T's updates are not visible to other Ts before commit
  - **Durability**: committed changes to DB are permanent
- Schedule of concurrently executing interleaved transactions
  - Trace from database
  - Def: schedule S (T_1, ..., T_n) = an **ordering** of operations of T_i, s.t.
  - order of operations of T_i in S = order of operations in T_i forall T_i.
  - ordering could be total or partial.
  - Ex. Fig. 17.3(a), (b) :: shorthand $r_i(X)$, $w_j(Y)$, c_i, a_j, ...
  - Committed projection: C(S) = operations from committed Ts

Exercises

- Q1? How many possible schedules for
  - Transactions in (a) Fig. 21.2 ?
  - Transactions in (b) Fig. 21.8?
- Q2? Count number of schedules for T1, ..., Tn,
  - Where Ni = number of operations in Ti
  - N = N1 + N2 + ... + Nn

Exercises

- Answer 1:
  - (a) (9!)/(6!*3!) = 84
  - (b) (13!)/(5!*4!*4!) = 90900
- Answer 2:
  - The ordering of Ni actions within Ti preserved = >
  - lose (Ni!) permutations for each valid schedule
  - #schedules S(T1, ..., Tn) = N! / (N1! * N2! * ... Nn!)

Recoverability Properties of Schedules (21.4.2)

- **Strict schedule(S) :** iff (wj(X) < cj < ri(X)), (wj(X) < cj < wi(X))
  - no concurrent read/write on any data item!
  - recovery is easy = > restore before image of X
- **Recoverable** with cascaded_rollback
  - iff forall Ti,Tj in S: reads_from(Ti,Tj,S) = > (cj < ci)
  - commit(Ti) waits for commit(Tj)
  - Ti self-aborts if Tj has aborted
  - Assumes abort(Tj) can not rollback commited transactions.
- No cascaded rollback
  - Cascadeless(S) iff reads_from(Ti,Tj,S) = > (cj < ri(X))
  - No dirty read (strong condition to avoid cascaded rollbacks)

Serializability of Schedules (21.5)

- Why is serializability interesting?
  - It is the characterization of "correctness", legal interleavings
- Serial, Non-serial and serializable schedules (Fig. 21.5)
  - Serial S: no interleaving, i.e. all actions Ti together in S
  - Nonserial S: interleaving, for some Ti, all actions are not together
  - Serializable S, if S is eqv. to some serial schedule of same Ts
- ...used as correctness measure

Serializability of Schedules (Cont.)

- Conflict Serializable(S) iff conflict-equivalent(S, a serial schedule)
- Equivalence of schedules S1, S2
  - Result eqv.: if they produce the same final state of DB
    - not enough for correctness, see Fig. 21.6
  - Conflict eqv. order of any two conflicting operation is identical
    - < Wi(X),..., Wj(X) > or < Wi(X), ...,Rj(X) > or < Ri(X), ...,Wj(X) >

Testing Conflict serializability
(Algo. 21.1, pp 763 in Chap 21.5.2)

- Look at only ReadItem (X) and WriteItem (X) operations
- Constructs a precedence graph
  - Nodes = transactions T1, T2, ..., Tn
  - Edges = conflists (read/write, write/read, write/write)
  - ...(Ti -- > Tj) if before(wi(X),rj(X)) in S
  - ...(Ti -- > Tj) if before(ri(X),wj(X)) in S
  - ...(Ti -- > Tj) if before(wi(X),wj(X)) in S
- No cycle in predence graph(S) IFF conflict serializable(S)

Exercises

- Ex. Consider the schedule in Fig. 21.5 (pp760)
  - Classify schedules in Fig. 21.5 as serial, serializable, nonserial
  - Find eqv. schdules under conflict eqv.
- Ex. Fig. 21.5 : check schedules C and D for serializability.
- Ex. Fig. 21.8(a-c) (pp 766-767) check serilizability of schedules E and F

Global Serializability

- Distributed transaction
- Ensure not only the subtransactions are serializable at each sites
- But also global serialization over all sites