The Archive, Serializability

CS157B
Chris Pollett
Apr.25, 2005.

Outline

- The Archive
- Serial and Serializable Schedules
- Conflict Serializability

The Archive

- To protect against media failures,we want to keep an archive. That is, we want to maintain a copy of the database separate from the database itself.
- There are two levels of archiving:
  - A *full dump,* in which the entire database is copied.
  - An *incremental dump*, in which only those database elements changed since the previous full or incremental dump are copied.
- Archiving is different from just backing up the log file, since if we did the latter, then over time we would need to store way too much data.
- They are connected though. To restore, we will generally use the most recent full archive together with subsequent incremental dumps and the log file since the last archive.

Non-quiescent Archiving

- As we cannot shut the database down while archiving, a non-quiescent dump tries to capture the database as it was at the start of the dump even though transactions continue to be processed.
- We assume redo or undo/redo logging is being used.
- The steps to do an archive are:
  1. Write a <START DUMP> record.
  2. Perform a check point.
  3. Perform a full or incremental dump of the data disks as desired, copying the data in some fixed order.
  4. Copy enough of the log that the prefix of the log up to and including the checkpoint in item (2) is included.
  5. Write a log record <END DUMP>. We can now throw away old log from last archive to previous checkpoint.

Recovery Using an Archive and a Log

- To restore the database from the archive involves:
  - Finding the most recent full dump and reconstructing the database from it.
  - If there are later incremental dumps, modifying the database according to each, earliest first.
  - Modifying the database using the surviving log. This in turn involves, using the method of recovery suited to the type of logging being used.

Concurrency Control

- Interactions between transactions can cause the database to become inconsistent even when the transactions individually preserve the correctness of the database state.
- This is because transactions can interleave their actions.
- The job of figuring out which operation of which transaction is performed next is done by the database *scheduler*.
- The process of insuring in such a concurrent set up that the database stays in a consistent state is called *concurrency control*.
- We are now going to study conditions which guarantee database consistency.

Schedules

- A *schedule* is a time-ordered sequence of the important actions taken by one or more transactions.
- We are interested in reads and write and not in outputs.
- For example, suppose we had two transactions:
  - T1: R(A,t), t:=t+100, W(A,t), R(B,t), t := t + 100, W(B,t).
  - T2: R(A,s); s:= s*2, W(A,s), R(B,s), s := s*2, W(B,s)
- An example schedule might be:

  $R_1(A,t)$, $(t:=t+100)_1$, $W_1(A,t)$, $R_1(B,t)$, $(t := t + 100)_1$, $W_1(B,t)$, $R_2(A,s)$; $(s:= s*2)_2$, $W_2(A,s)$, $R_2(B,s)$, $(s := s*2)_2$, $W_2(B,s)$.

Serial Schedules

- A schedule is said to be a *serial* schedule if all of its actions consist of all the actions of one transaction, followed by all the actions of another transaction, etc. without interleaving of transaction operations.
- The example of the last slide was a serial schedule.
- If each transaction maps the database from a consistent state to a consistent state, then a serial schedule will map the database from a consistent state to a consistent state.

Serializable Schedules

- Serial schedules don't allow two transactions to be working on the DB at the same time. So we want a better notion of a good schedule so that we can get better concurrency.
- A *serializable* schedule is a schedule whose effect on the database is the same as some serial schedule.
- For example, $R_1(A,t)$, $(t{:=}t+100)_1$, $W_1(A,t)$, $R_2(A,s)$, $(s{:=}\ s*2)_2$, $R_1(B,t)$, $(t := t + 100)_1$, $W_1(B,t)$, $W_2(A,s)$, $R_2(B,s)$, $(s := s*2)_2$, $W_2(B,s)$.

  has the same effect on the database as the schedule a couple slides back.
- Usually, we don't record the local variables of a transaction when we write our schedules to keep them simple. i.e., we write W(A) for W(A,t) and we wouldn't write actions like t:=t+100.

Conflict Serializable

- Serializable is still too general, and it is hard to ensure a schedule is serialiazable. We will next look at a weaker notion, which will imply serializable, allows some concurrency, but maybe allows less concurrency than serializability.
- We say a pair of operations O1,…,O2 in a schedule from two different transactions S and T do not *conflict* if:

1. They are both reads.
2. They are $R_S(X)$ and $W_T(Y)$ and X is not equal to Y.
3. They are $W_S(X)$ and $R_T(Y)$ and X is not equal to Y
4. They are $W_S(X)$ and $W_T(Y)$ and and X is not equal to Y.

- Otherwise, the two actions are said to *conflict*.
- Two schedules are *conflict equivalent* if they can be turned into each other by swapping non-conflicting transactions.
- A transaction is *conflict-serializable* if it is conflict equivalent to a serial schedule.