Lecture 16
Concurrency Control with Timestamps

Chapter 18.2-18.4 (except 18.3.2): Elmasri & Navathe, 5ED
Chapter 17.6: Ramakrishnan & Gehrke, 3ED

Demetris Zeinalipour
http://www.cs.ucy.ac.cy/~dzeina/courses/epl446
Concurrency Control in DBMSs
(Έλεγχος Ταυτοχρονίας σε ΣΔΒΔ)

• In the previous lecture we explained how a real DBMS enforces (επιβάλει) Serializability and Recoverability (Strict 2PL) in its transaction schedules using Locking.

• We will now see another class of protocols based on Timestamps (though not widely utilized in real DBMSs, they have a theoretical interest).

• Concurrency Control with Timestamps (without Locking)
  – Timestamp Ordering (Έλεγχος Ταυτοχρονίσμος με Διάταξη Χρονόσημων): Ensure serializability using the ordering of timestamps generated by the DBMS.
  – Multiversion CC (Έλεγχος Ταυτοχρονίσμος Με Πολλαπλές Εκδόσεις): Use multiple versions of items to enforce serializability.
  – Optimistic CC (Αισιόδοξος (Οπτιμιστικός) Έλεγχος Ταυτοχρονίσμού): No checking done during execution of a Transaction but post-execution validation (επικύρωση) enforces serializability.
Timestamp based CC: Definitions
(Έλεγχος Ταυτοχρονίας με Χρονόσημα: Ορισμοί)

- **Timestamp (Χρονόσημο)**
  - A monotonically increasing variable (integer) indicating the age of an operation or a transaction.
  - A larger timestamp indicates a more recent transaction
    - Timestamps are assigned in our context during Xact creation.
    - **Using date timestamps** (e.g., a long integer that represents the number of seconds that have elapsed from 1/1/1970)
      - TS1: 1237917600 (2009-03-24 18:00:00)
      - TS2: 1237917610 (2009-03-24 18:00:10)
    - **Using a counter timestamp** (e.g., using a counter stored inside the Operating System kernel as a semaphore)

---

**Diagram:**
- Time
  - TS1 (Older Transaction)
  - TS2 (Younger Transaction)
  - 10 sec
  - Now
• Assume a collection of **data items** that are accessed, with read and write operations, by transactions.

• **For each data item** X the DBMS maintains the following values:
  
  – **RTS(X):** The Timestamp on which object X was **last read** (by some transaction T_i, i.e., RTS(X):=TS(T_i))
  
  – **WTS(X):** The Timestamp on which object X was **last written** (by some transaction T_j, i.e., WTS(X):=TS(T_j))

• **For the following algorithms we use the following assumptions:**
  
  – A data item X in the database has a **RTS(X)** and **WTS(X)** (recorded when the object was last accessed for the given action)
  
  – A transaction T attempts to perform some action (read or write) on data item X on timestamp **TS(T)**
    
    • **Problem:** We need to decide whether T has to be aborted or whether T can continue execution.
We shall now present the first algorithm, coined **Basic Timestamp Ordering (TO)**, that utilizes Timestamps to guarantee **serializability** of concurrent transactions.

**Timestamp Ordering (TO) Rule**

- If $p_a(x)$ and $q_b(x)$ are conflicting operations, of xacts $T_a$ and $T_b$ for item $x$, then $p_a(x)$ is processed before $q_b(x)$ iff ($\Rightarrow \Leftarrow$) $ts(T_a) < ts(T_b)$

- **Main Idea**: Conflicts are only allowed from older transactions (with smaller ts) to a younger transaction $T$ (with larger ts)

- **Main Idea Example**:

  ![Diagram](image.png)

  Note that the conflict moves only to the right not to the left!

**Theorem**: If the TO rule is enforced in a schedule then the schedule is (conflict) serializable.

- **Why?** Because cycles are not possible in the Conflict Precedence Graph (Γράφος Προτεραιότητας Συγκρούσεων)!
Basic Timestamp Ordering Algorithm
(Basic Timestamp Ordering Algorithm)

Case 1 (Read): Transaction T issues a read(X) operation

A. If \(TS(T) < WTS(X)\), then read(X) is rejected (as the TO rule is violated).
   T has to *abort* and be *rejected*.

   Happens if T started earlier than T′ (that wrote X) … see example on slide 16.9

   \[\text{Reject T (TO rule violated)}\]

B. If \(WTS(X) \leq TS(T)\), then execute read(X) of T and update RTS(X).
   \[\text{Accept T (TO ok)}\]

   \(R/R\) not conflicting action so RTS(X) ≤ TS(T) not investigated

Case 2 (Write): Transaction T issues a write(X) operation

A. If \(TS(T) < RTS(X)\) or if \(TS(T) < WTS(X)\), then write is rejected.
   \[\text{Reject T (TO rule violated)}\]

B. If \(RTS(X) \leq TS(T)\) or \(WTS(X) \leq TS(T)\), then execute write(X) of T and update WTS(X).
   \[\text{Accept T (TO ok)}\]
Basic TO Algorithm Example
(Παράδειγμα Βασικός Αλγόριθμος Διάταξης Χρονοσήμων)

• Consider the following scenario:
  – Two transactions T1 and T2
  – Initially RTS=0 and WTS=0 for data items X, Y
  – Timestamps are as follows: TS(T1)=10 and TS(T2)=20

T1(10)
1. A1 = Read(X)
2. A1 = A1 − k
3. Write(X, A1)
4. A2 = Read(Y)
5. A2 = A2 + k
6. Write(Y, A2)

T2(20)
1. A1 = Read(X)
2. A1 = A1 * 1.01
3. Write(X, A1)
4. A2 = Read(Y)
5. A2 = A2 * 1.01
6. Write(Y, A2)
Basic TO Algorithm Example

(Παράδειγμα Βασικός Αλγόριθμος Διάταξης Χρονοσήμων)

• Is the schedule serializable?

  – Utilize the Basic TO Algorithm to justify your answer (otherwise the precedence graph could have been used to answer this question)

  **T1(10)**

1. \( A1 = \text{Read}(X) \)
2. \( A1 = A1 - k \)
3. \( \text{Write}(X, A1) \)

**T2(20)**

1. \( A1 = \text{Read}(X) \)
2. \( A1 = A1 \ast 1.01 \)
3. \( \text{Write}(X, A1) \)

RTS(X): 10
WTS(X): 10
RTS(Y): 0
WTS(Y): 0

RTS(X): 20
WTS(X): 20
RTS(Y): 0
WTS(Y): 0

Yes! The schedule is serializable!
This can be confirmed by the precedence graph which is acyclic
Basic TO Algorithm Example

(Παράδειγμα Βασικός Αλγόριθμος Διάταξης Χρονοσήμων)

- Is the schedule serializable?
  - Utilize the Basic TO Algorithm to justify your answer

T1(10)
1. A1 = Read(X)
2. A1 = A1 - k
3. Write(X, A1)

T2(20)
1. A1 = Read(X)
2. A1 = A1 * 1.01
3. Write(X, A1)
4. A2 = Read(Y)
5. A2 = A2 * 1.01
6. Write(Y, A2)

So far we discussed only about serializability. How about recoverability? We will discuss this in a while.

NO! The schedule is NOT serializable
- this is confirmed with the precedence graph which is cyclic
Advantages/Disadvantages of Basic TO

(Πλεονεκ./Μειονεκ. του Βασικού Αλγ. Διατ. Χρον.)

• **Basic TO Remark**
  – Note that there is no notion of RR-conflict
  If \( TS(T) < RTS(X) \), then execute \( \text{read}(X) \) of \( T \) and update \( RTS(X) \).

• **Advantages of Basic TO Algorithm**
  – Schedules are serializable (like 2PL protocols)
  – No waiting for transaction, thus, no deadlocks!

• **Disadvantages**
  – Schedule may not be recoverable (read uncomit. data)
    • **Solution**: Utilize Strict TO Algorithm (see next)
  – **Starvation** is possible (if the same transaction is continually aborted and restarted)
    • **Solution**: Assign new timestamp for aborted transaction
Strict Timestamp Ordering
(Αστηρός Αλγόριθμος Διάταξης Χρονοσήμων)

- The **Basic T.O.** algorithm guarantees **serializability** but not **recoverability** (επαναφερσιμότητα)
- The **Strict T.O.** algorithms introduces **recoverability**.
  - (Revision) **Strict Schedule**: A transaction can neither **read** or **write** an uncommitted data item X.

**Strict T.O. Main Idea**: Extend the **Accept cases** of the **Basic T.O. algorithm** by adding the requirement that a commit occurs before T proceeds with its operation. i.e.,

**For read()**
\[ WTS(X) \leq TS(T) \]

Accept T (TO ok)

**For write()**
\[ RTS(X) \leq TS(T) \text{ or } WTS(X) \leq TS(T) \]

Accept T (TO ok)

\[ WTS(X) \]
\[ TS(T): \text{Read}(X) \]
\[ \text{commit} \]
\[ \text{Time} \]

\[ WTS(X) \]
\[ TS(T): \text{Write}(X) \]
\[ \text{commit} \]
\[ \text{Time} \]
Strict Timestamp Ordering (Strict T.O.)

- **Case 1**: Transaction T issues a read(X) operation:
  - If $WTS(X) < TS(T)$, then delay T until the transaction $T'$ that wrote or read X has terminated (committed or aborted).

- **Case 2**: Transaction T issues a write(X) operation:
  - If $RTS(X) \leq TS(T)$ or $WTS(X) \leq TS(T)$, then delay T until the transaction $T'$ that wrote or read X has terminated (committed or aborted).
Multiversion Concurrency Control
(Έλεγχος Ταυτοχρονισμού Με Πολλαπλές Εκδόσεις)

- **Multiversion technique based on timestamp ordering** (Έλεγχος Ταυτοχρονισμού Με Πολλαπλές Εκδόσεις)
  - This approach maintains a **number of versions** of a **data item** and allocates the **right version** to a **read operation of a transaction**.
    - Thus unlike other mechanisms a **read operation in this mechanism is never rejected**.

- **Disadvantage:**
  - Significantly **more storage** (RAM and Disk) is required to maintain **multiple versions**.
  - To check **unlimited growth of versions**, a **garbage collection** is run periodically.
Multiversion Concurrency Control
(Έλεγχος Ταυτοχρονισμού Με Πολλαπλές Εκδόσεις)

- **Multiversion technique based on Timestamp Ordering**
  - Assume \( X_1, X_2, \ldots, X_n \) are the version of a data item \( X \) created by a write operation of transactions.
  - Note: New version of \( X_i \) is created only by a write operation.
  - With each \( X_i \) a RTS (read timestamp) and a WTS (write timestamp) are associated.

- **Notation**
  - \( \text{RTS}(X_i) \): The read timestamp of \( X_i \) is the largest of all the timestamps of transactions that have successfully read version \( X_i \).
  - \( \text{WTS}(X_i) \): The write timestamp of \( X_i \) is the largest of all the timestamps of transactions that have successfully written the value of version \( X_i \).

- **Basic Idea**: Works much like Basic TO with the difference that instead of WTS(\( X \)) and RTS(\( X \)) we now utilize the highest WTS(\( X_i \)) and highest RTS(\( X_i \)) respectively.
Multiversion Concurrency Control

(Έλεγχος Ταυτοχρονισμού Με Πολλαπλές Εκδόσεις)

• To ensure serializability, the following rules are used
  – **C1B)** If transaction T issues \texttt{read(X)}, find the version \(i\) of \(X\) that has the highest \(WTS(X_i)\) of all versions of \(X\) that is also less than or equal to \(TS(T)\), then accept the read and update the RTS(\(X\)) respectively.

  \[
  \begin{align*}
  \text{Max}\{WTS(X_i)\} & \quad \text{TS(T): Read(X)} \\
  \text{Max}\{WTS(X_i)\} & \quad \text{TS(T): Read(X)}
  \end{align*}
  \]

  \[
  \begin{align*}
  \text{Accept T (TO ok)} & \quad \text{Max}\{RTS(X_i)\} \\
  \text{Max}\{RTS(X_i)\} & \quad \text{Max}\{WTS(X_i)\}
  \end{align*}
  \]

  \[
  \begin{align*}
  \text{Time} & \quad \text{Time} \\
  \text{Time} & \quad \text{Time}
  \end{align*}
  \]

  \[
  \begin{align*}
  \text{X} & \quad \text{X}
  \end{align*}
  \]

  \[
  \begin{align*}
  \text{Max}\{WTS(X_i)\} & \quad \text{TS(T): Write(X)} \\
  \text{Max}\{RTS(X_i)\} & \quad \text{TS(T): Write(X)}
  \end{align*}
  \]

  \[
  \begin{align*}
  \text{Accept T (TO ok)} & \quad \text{Accept T (TO ok)} \\
  \text{Max}\{RTS(X_i)\} & \quad \text{Max}\{RTS(X_i)\}
  \end{align*}
  \]

  \[
  \begin{align*}
  \text{Time} & \quad \text{Time} \\
  \text{Time} & \quad \text{Time}
  \end{align*}
  \]

  \[
  \begin{align*}
  \text{X} & \quad \text{X}
  \end{align*}
  \]

  \[
  \begin{align*}
  \text{Max}\{RTS(X_i)\} & \quad \text{TS(T): Write(X)} \\
  \text{Max}\{WTS(X_i)\} & \quad \text{TS(T): Write(X)}
  \end{align*}
  \]

  \[
  \begin{align*}
  \text{Accept T (TO ok)} & \quad \text{Reject T (TO rule violated)} \\
  \text{Max}\{RTS(X_i)\} & \quad \text{Max}\{WTS(X_i)\}
  \end{align*}
  \]

  \[
  \begin{align*}
  \text{Time} & \quad \text{Time} \\
  \text{Time} & \quad \text{Time}
  \end{align*}
  \]

  \[
  \begin{align*}
  \text{X} & \quad \text{X}
  \end{align*}
  \]

  \[
  \begin{align*}
  \text{Max}\{RTS(X_i)\} & \quad \text{TS(T): Write(X)} \\
  \text{Max}\{WTS(X_i)\} & \quad \text{TS(T): Write(X)}
  \end{align*}
  \]

  \[
  \begin{align*}
  \text{Accept T (TO ok)} & \quad \text{Reject T (TO rule violated)} \\
  \text{Max}\{RTS(X_i)\} & \quad \text{Max}\{WTS(X_i)\}
  \end{align*}
  \]

  \[
  \begin{align*}
  \text{Time} & \quad \text{Time} \\
  \text{Time} & \quad \text{Time}
  \end{align*}
  \]

  \[
  \begin{align*}
  \text{X} & \quad \text{X}
  \end{align*}
  \]

  \[
  \begin{align*}
  \text{Max}\{RTS(X_i)\} & \quad \text{TS(T): Write(X)} \\
  \text{Max}\{WTS(X_i)\} & \quad \text{TS(T): Write(X)}
  \end{align*}
  \]

  \[
  \begin{align*}
  \text{Accept T (TO ok)} & \quad \text{Reject T (TO rule violated)} \\
  \text{Max}\{RTS(X_i)\} & \quad \text{Max}\{WTS(X_i)\}
  \end{align*}
  \]

  \[
  \begin{align*}
  \text{Time} & \quad \text{Time} \\
  \text{Time} & \quad \text{Time}
  \end{align*}
  \]

  \[
  \begin{align*}
  \text{X} & \quad \text{X}
  \end{align*}
  \]

- **C2A)** If transaction T issues \texttt{write(X)} and version \(i\) of \(X\) has the highest \(WTS(X_i)\) of all versions of \(X\) that is also less than or equal to \(TS(T)\), and \(TS(T) < RTS(X_i)\), then abort and roll-back T;

- **C2B)** otherwise create a new version \(X_i\) and \(\text{read}_\text{TS}(X) = \text{write}_\text{TS}(X_j) = TS(T)\).
Optimistic Concurrency Control
(Οπτιμιστικός Έλεγχος Ταυτοχρονίας)

- Locking and TO are pessimistic (απαισιόδοξοι) ways to handle concurrency (We assume that conflicts will arise)
- When most transactions don’t conflict with the other transactions then Optimistic CC is much more efficient.

(Optimistic) Concurrency Control
(Αισιόδοξος (Οπτιμιστικός) Έλεγχος Ταυτοχρονισμού)

- Basic Idea:
  - In Optimistic CC, a schedule is checked against serializability only at the time of commit (e.g., using timestamp orders or some other mechanism)
  - transactions are aborted in case of non-serializable schedules.

- Three phases:
  1. Read phase (Φάση Ανάγνωσης)
  2. Validation phase (Φάση Επικύρωσης)
  3. Write phase (Φάση Γραφής)