

IF3140 – Sistem Basis Data

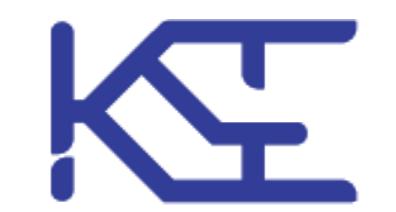
Concurrency Control:

- Multiversion scheme
- Insert/Delete Operations
- Weak levels of consistency



KNOWLEDGE & SOFTWARE ENGINEERING

Multiversion Schemes



KNOWLEDGE & SOFTWARE ENGINEERING

operasi read akan selalu berhasil

tiap kali mau write pasti
membuat versi baru

Multiversion Schemes

- Multiversion schemes keep old versions of data item to increase concurrency. Several variants:
 - **Multiversion Timestamp Ordering**
 - **Multiversion Two-Phase Locking**
 - **Snapshot isolation**
- Key ideas:
 - Each successful **write** results in the creation of a new version of the data item written.
 - Use timestamps to label versions.
 - When a **read(Q)** operation is issued, select an appropriate version of Q based on the timestamp of the transaction issuing the read request, and return the value of the selected version.
 - **reads** never have to wait as an appropriate version is returned immediately.



KNOWLEDGE & SOFTWARE ENGINEERING

©Silberschatz et.al. (2019)

[modified by Tim Pengajar IF3140 Semester 1 2024/2025]

Multiversion Timestamp Ordering

- Each data item Q has a sequence of versions $\langle Q_1, Q_2, \dots, Q_m \rangle$. Each version Q_k contains three data fields:
 - **Content** – the value of version Q_k .
 - **W-timestamp**(Q_k) – timestamp of the transaction that created (wrote) version Q_k
 - **R-timestamp**(Q_k) – largest timestamp of a transaction that successfully read version Q_k



KNOWLEDGE & SOFTWARE ENGINEERING

Multiversion Timestamp Ordering (Cont)

- Suppose that transaction T_i issues a **read(Q)** or **write(Q)** operation. Let Q_k denote the version of Q whose write timestamp is the largest write timestamp less than or equal to $TS(T_i)$.
 1. If transaction T_i issues a **read(Q)**, then
 - the value returned is the content of version Q_k
 - If $R\text{-timestamp}(Q_k) < TS(T_i)$, set $R\text{-timestamp}(Q_k) = TS(T_i)$,
 2. If transaction T_i issues a **write(Q)**
 1. if $TS(T_i) < R\text{-timestamp}(Q_k)$, then transaction T_i is rolled back.
 2. if $TS(T_i) = W\text{-timestamp}(Q_k)$, the contents of Q_k are overwritten
 3. Otherwise, a new version Q_i of Q is created
 - $W\text{-timestamp}(Q_i)$ and $R\text{-timestamp}(Q_i)$ are initialized to $TS(T_i)$.



KNOWLEDGE & SOFTWARE ENGINEERING

Another Example Under TSO

A partial schedule for several data items for transactions with timestamps 1, 2, 3, 4, 5, with all R-TS and W-TS = 0 initially

T_1	T_2	T_3	T_4	T_5	
	read (Y)			read (X)	$T_S(X_0) = (5, 0)$
read (Y)					$T_S(Y_0) = (2, 0)$
	read (Z)	write (Y) write (Z)		read (Z)	$T_S(Y_0) = (2, 0)$
	abort				$T_S(Y_3) = (3, 3)$
read (X)		read (W)			$T_S(Z_3) = (3, 3)$
		write (W) abort			$T_S(Z_3) = (5, 3)$
			read (W)		$T_S(Z_0) = (2, 0)$
				write (Y) write (Z)	$T_S(X_0) = (5, 0)$
					$T_S(W_0) = (4, 0)$
					rollback



Multiverion Timestamp Ordering

(Cont)

- Observations
 - Reads always succeed
 - A write by T_i is rejected if some other transaction T_j that (in the serialization order defined by the timestamp values) should read T_i 's write, has already read a version created by a transaction older than T_i .
- Protocol guarantees serializability



KNOWLEDGE & SOFTWARE ENGINEERING

©Silberschatz et.al. (2019)

[modified by Tim Pengajar IF3140 Semester 1 2024/2025]

Multiversion Two-Phase Locking

- Differentiates between read-only transactions and update transactions
- **Update transactions** acquire read and write locks, and hold all locks up to the end of the transaction. That is, update transactions follow rigorous two-phase locking.
 - **Read of a data item** returns the latest version of the item
 - The first **write** of Q by T_i results in the creation of a new version Q_i of the data item Q written
 - $W\text{-timestamp}(Q_i)$ set to ∞ initially → untuk write pertama thd Q
 - When update transaction T_i completes, commit processing occurs:
 - Value **ts-counter** stored in the database is used to assign timestamps
 - **ts-counter** is locked in two-phase manner → kalo yg lain mau cb commit gabisa
 - Set $TS(T_i) = \text{ts-counter} + 1$
 - Set $W\text{-timestamp}(Q_i) = TS(T_i)$ for all versions Q_i that it creates
 - **ts-counter = ts-counter + 1** → +1 krn read only baca versi yg udh dihasilkan
setelah itu lock dibuka, commit selesai



KNOWLEDGE & SOFTWARE ENGINEERING

Multiversion Two-Phase Locking (Cont.)

- **Read-only transactions**

- are assigned a timestamp = **ts-counter** when they start execution
- follow the multiversion timestamp-ordering protocol for performing reads
 - Do not obtain any locks
- Read-only transactions that start after T_i increments **ts-counter** will see the values updated by T_i .
- Read-only transactions that start before T_i increments the **ts-counter** will see the value before the updates by T_i .
- Only serializable schedules are produced.

selalu berhasil, tinggal cari versi yg sesuai aja

versi yg dibaca = ts-counter atau versi max tp < ts-counter



KNOWLEDGE & SOFTWARE ENGINEERING

A partial schedule for several data items for transactions T_1, T_2, T_3, T_4, T_5

with ts-counter and TS of all data = 0 initially				
T_1	T_2	T_3	T_4	T_5
read(A)				
	read(B)			
read(B)				
write(A)				
	read(A)			
		read(A)		
commit				
	commit			
			read(A)	
		write(A)		
			read(B)	
		commit		
				read(A)



MVCC: Implementation Issues

- Creation of multiple versions increases storage overhead
 - Extra tuples → byk versi → buat nyimpen timestamp
 - Extra space in each tuple for storing version information
 - Versions can, however, be garbage collected
 - E.g., if Q has two versions Q5 and Q9, and the oldest active transaction has timestamp > 9, than Q5 will never be required again
 - Issues with
 - primary key and foreign key constraint checking
 - Indexing of records with multiple versions
- See textbook for details



KNOWLEDGE & SOFTWARE ENGINEERING

Snapshot Isolation

- Motivation: Decision support queries that read large amounts of data have concurrency conflicts with OLTP transactions that update a few rows
 - Poor performance results
- Solution 1: Use multiversion 2-phase locking
 - Give logical “snapshot” of database state to read only transaction
 - Reads performed on snapshot
 - Update (read-write) transactions use normal locking
 - Works well, but how does system know a transaction is read only?
- Solution 2 (partial): Give snapshot of database state to every transaction
 - Reads performed on snapshot
 - Use 2-phase locking on updated data items
 - Problem: variety of anomalies such as lost update can result
 - Better solution: snapshot isolation level (next slide)



KNOWLEDGE & SOFTWARE ENGINEERING

Snapshot Isolation

- A transaction T1 executing with Snapshot Isolation
 - Takes snapshot of committed data at start
 - Always reads/modifies data in its own snapshot
 - Updates of concurrent transactions are not visible to T1
 - Writes of T1 complete when it commits
 - First-committer-wins rule:** *transaksi yg pertama kali commit akan menang*
 - Commits only if no other concurrent transaction has already written data that T1 intends to write.

Concurrent updates not visible

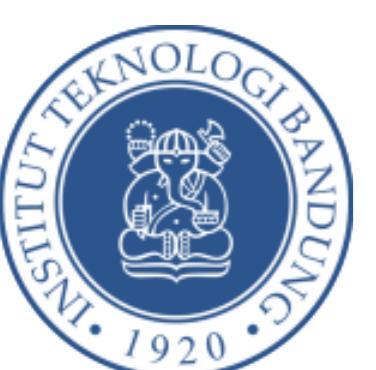
Own updates are visible

Not first-committer of X

Serialization error, T2 is rolled back

T1	T2	T3
W(Y := 1) Commit ✓		
	Start R(X) → 0 R(Y) → 1 <i>sesuai dgn penulisan T1</i>	
		W(X:=2) W(Z:=3) Commit ✓

Diagram illustrating the execution of three transactions (T1, T2, T3) under Snapshot Isolation. Transaction T1 is shown committing with a checkmark. Transaction T2 starts and reads X=0, Y=1. Transaction T3 starts and reads X=0, Y=1. Handwritten notes indicate that T1's writes are visible to T2 and T3. T3's writes are visible to T2. A 'Commit-Req' message is sent from T3 to T2. A 'Abort' message is sent from T2 to T3, indicating a serialization error because T2 is rolled back.



Snapshot Read



masing-masing membaca versinya sendiri

Concurrent updates invisible to snapshot read

$$X_0 = 100, Y_0 = 0$$

T_1 deposits 50 in Y	T_2 withdraws 50 from X
$r_1(X_0, 100)$ $r_1(Y_0, 0)$ $w_1(Y_1, 50)$ $r_1(X_0, 100)$ (update by T_2 not seen) $r_1(Y_1, 50)$ (can see its own updates)	$r_2(Y_0, 0)$ $r_2(X_0, 100)$ $w_2(X_2, 50)$ $r_2(Y_0, 0)$ (update by T_1 not seen)

$$X_2 = 50, Y_1 = 50$$



Snapshot Write: First Committer Wins

$X_0 = 100$	
T_1 deposits 50 in X	T_2 withdraws 50 from X
$r_1(X_0, 100)$	$r_2(X_0, 100)$ $w_2(X_2, 50)$
$w_1(X_1, 150)$ $commit_1$	
	$commit_2$ (Serialization Error T_2 is rolled back)

$X_1 = 150$

- Variant: “**First-updater-wins**”
 - Check for concurrent updates when write occurs by locking item
 - But lock should be held till all concurrent transactions have finished
 - (Oracle uses this plus some extra features)
 - Differs only in when abort occurs, otherwise equivalent
- kalau transaksi mau write,
dia lock dulu item yg
mou diwrite
- yg menang adlh yg
pertama kali update

Benefits of SI

- Reads are never blocked, *semua read pasti berhasil*
 - and also don't block other txns activities
- Performance similar to Read Committed
- Avoids several anomalies
 - No dirty read, i.e. no read of uncommitted data
 - No lost update
 - I.e., update made by a transaction is overwritten by another transaction that did not see the update)
 - No non-repeatable read *kerja di snapshot sendiri*
 - I.e., if read is executed again, it will see the same value
- Problems with SI
 - SI does not always give serializable executions
 - Serializable: among two concurrent txns, one sees the effects of the other
 - In SI: neither sees the effects of the other
 - Result: Integrity constraints can be violated



KNOWLEDGE & SOFTWARE ENGINEERING

Snapshot Isolation

- Example of problem with SI
 - Initially $A = 3$ and $B = 17$
 - Serial execution: $A = ??, B = ??$
 - if both transactions start at the same time, with snapshot isolation: $A = ??, B = ??$
- Called **skew write**
- Skew also occurs with inserts
 - E.g:
 - Find max order number among all orders
 - Create a new order with order number = previous max + 1
 - Two transaction can both create order with same number
 - Is an example of phantom phenomenon

serial T_i baru $T_j \rightarrow A=17, B=17$
 T_j baru $T_i \rightarrow A=3, B=3$

SI \rightarrow **A = 17, B = 3** :: ga serializable

T_i	T_j
read(A)	
read(B)	
	read(A)
	read(B)
	$A=B$
	$B=A$
	write(A)
	write(B)



Snapshot Isolation Anomalies

- SI breaks serializability when transactions modify *different* items, each based on a previous state of the item the other modified
 - Not very common in practice
 - E.g., the TPC-C benchmark runs correctly under SI
 - when txns conflict due to modifying different data, there is usually also a shared item they both modify, so SI will abort one of them
 - But problems do occur
 - Application developers should be careful about write skew
- SI can also cause a read-only transaction anomaly, where read-only transaction may see an inconsistent state even if updaters are serializable
 - We omit details
- Using snapshots to verify primary/foreign key integrity can lead to inconsistency
 - Integrity constraint checking usually done outside of snapshot



KNOWLEDGE & SOFTWARE ENGINEERING

©Silberschatz et.al. (2019)

[modified by Tim Pengajar IF3140 Semester 1 2024/2025]

Serializable Snapshot Isolation

- **Serializable snapshot isolation (SSI)**: extension of snapshot isolation that ensures serializability
- Snapshot isolation tracks write-write conflicts, but does not track read-write conflicts
 - Where T_i writes a data item Q, T_j reads an earlier version of Q, but T_j is serialized after T_i
- Idea: track read-write dependencies separately, and roll-back transactions where cycles can occur
 - Ensures serializability
 - Details in book
- Implemented in PostgreSQL from version 9.1 onwards
 - PostgreSQL implementation of SSI also uses index locking to detect phantom conflicts, thus ensuring true serializability



KNOWLEDGE & SOFTWARE ENGINEERING

SI Implementations

- Snapshot isolation supported by many databases
 - Including Oracle, PostgreSQL, SQL Server, IBM DB2, etc
 - Isolation level can be set to snapshot isolation
- Oracle implements “first updater wins” rule (variant of “first committer wins”)
 - Concurrent writer check is done at time of write, not at commit time
 - Allows transactions to be rolled back earlier
- **Warning:** even if isolation level is set to serializable, Oracle actually uses snapshot isolation
 - Old versions of PostgreSQL prior to 9.1 did this too
 - Oracle and PostgreSQL < 9.1 do not support true serializable execution



KNOWLEDGE & SOFTWARE ENGINEERING

Working Around SI Anomalies

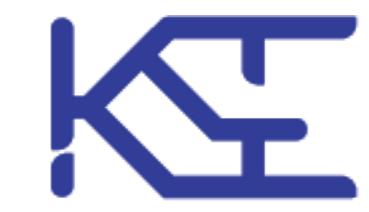
- Can work around SI anomalies for specific queries by using **select .. for update** (supported e.g. in Oracle)
 - Example
 - **select max(orderno) from orders for update**
 - read value into local variable maxorder
 - insert into orders (maxorder+1, ...)
 - **select for update (SFU) clause** treats all data read by the query as if it were also updated, preventing concurrent updates
 - Can be added to queries to ensure serializability in many applications
 - Does not handle phantom phenomenon/predicate reads though



KNOWLEDGE & SOFTWARE ENGINEERING



Insert/Delete Operations



KNOWLEDGE & SOFTWARE ENGINEERING

Insert/Delete Operations and Predicate Reads

→ konflik dgn read & write

- Locking rules for insert/delete operations
 - An exclusive lock must be obtained on an item before it is deleted
 - A transaction that inserts a new tuple into the database automatically given an X-mode lock on the tuple
- Ensures that
 - reads/writes conflict with deletes
 - Inserted tuple is not accessible by other transactions until the transaction that inserts the tuple commits



KNOWLEDGE & SOFTWARE ENGINEERING

Phantom Phenomenon

konflik terhadap tuple yg
sebenarnya tidak ada

- Example of **phantom phenomenon**.
 - A transaction T1 that performs **predicate read** (or scan) of a relation
 - **select count(*)
from instructor
where dept_name = 'Physics'**
 - and a transaction T2 that inserts a tuple while T1 is active but after predicate read
 - **insert into instructor values ('11111', 'Feynman', 'Physics', 94000)**
(conceptually) conflict in spite of not accessing any tuple in common.
- If only tuple locks are used, non-serializable schedules can result
 - E.g. the scan transaction does not see the new instructor, but may read some other tuple written by the update transaction



KNOWLEDGE & SOFTWARE ENGINEERING

©Silberschatz et.al. (2019)

[modified by Tim Pengajar IF3140 Semester 1 2024/2025]

Phantom Phenomenon

- Example of **phantom phenomenon**.
 - A transaction T1 that performs **predicate read** (or scan) of a relation
 - `select count(*)
from instructor
where dept_name = 'Physics'`
 - and a transaction T2 that inserts a tuple while T1 is active but after predicate read
 - `insert into instructor values ('11111', 'Feynman',
'Physics', 94000)`
 - (conceptually) conflict in spite of not accessing any tuple in common.
- If only tuple locks are used, non-serializable schedules can result
 - E.g. the scan transaction does not see the new instructor, but may read some other tuple written by the update transaction

T1	T2
Read(instructor where dept_name='Physics')	
	Insert Instructor in Physics
	Insert Instructor in Comp. Sci.
	Commit
	Read(instructor where dept_name='Comp. Sci.')

Can also occur with updates
E.g. update Wu's department from Finance to Physics



Insert/Delete Operations and Predicate Reads

- **Another Example:** T1 and T2 both find maximum instructor ID in parallel, and create new instructors with $ID = \text{maximum ID} + 1$
 - Both instructors get same ID, not possible in serializable schedule



KNOWLEDGE & SOFTWARE ENGINEERING

©Silberschatz et.al. (2019)

[modified by Tim Pengajar IF3140 Semester 1 2024/2025]

Handling Phantoms

- There is a conflict at the data level
 - The transaction performing predicate read or scanning the relation is reading information that indicates what tuples the relation contains
 - The transaction inserting/deleting/updating a tuple updates the same information.
 - The conflict should be detected, e.g. by locking the information.
- One solution:
 - Associate a data item with the relation, to represent the information about what tuples the relation contains.
 - Transactions scanning the relation acquire a shared lock in the data item,
 - Transactions inserting or deleting a tuple acquire an exclusive lock on the data item. (Note: locks on the data item do not conflict with locks on individual tuples.)
- Above protocol provides very low concurrency for insertions/deletions.



KNOWLEDGE & SOFTWARE ENGINEERING

Index Locking To Prevent Phantoms

yg dilock cukup leaf nodes yg affected

- **Index locking protocol** to prevent phantoms
 - Every relation must have at least one index.
 - A transaction can access tuples only after finding them through one or more indices on the relation → dia hrs akses index
 - A transaction T_i that performs a lookup must lock all the index leaf nodes that it accesses, in S-mode
 - Even if the leaf node does not contain any tuple satisfying the index lookup (e.g. for a range query, no tuple in a leaf is in the range)
 - A transaction T_i that inserts, updates or deletes a tuple t_i in a relation r
 - Must update all indices to r
 - Must obtain exclusive locks on all index leaf nodes affected by the insert/update/delete
 - The rules of the two-phase locking protocol must be observed
- Guarantees that phantom phenomenon won't occur



KNOWLEDGE & SOFTWARE ENGINEERING

Weak Levels of Consistency



KNOWLEDGE & SOFTWARE ENGINEERING

Weak Levels of Consistency

2PL tdk berlaku

- **Degree-two consistency:** differs from two-phase locking in that S-locks may be released at any time, and locks may be acquired at any time
 - X-locks must be held till end of transaction
 - Serializability is not guaranteed, programmer must ensure that no erroneous database state will occur
- **Cursor stability:**
 - For reads, each tuple is locked, read, and lock is immediately released
 - X-locks are held till end of transaction
 - Special case of degree-two consistency



Weak Levels of Consistency in SQL

- SQL allows non-serializable executions
 - **Serializable**: is the default *semua transaksi ekivalen*
 - **Repeatable read**: allows only committed records to be read, and repeating a read should return the same value (so read locks should be retained) *read yg berulang akan menghasilkan data yg sama*
 - However, the phantom phenomenon need not be prevented *menghasilkan data yg sama*
 - T1 may see some records inserted by T2, but may not see others inserted by T2
 - **Read committed**: same as degree two consistency, but most systems implement it as cursor-stability *tdk jamin kl read berulang menghasilkan data yg sama*
 - **Read uncommitted**: allows even uncommitted data to be read
- In most database systems, read committed is the default consistency level
 - Can be changed as database configuration parameter, or per transaction
 - set isolation level serializable



KNOWLEDGE & SOFTWARE ENGINEERING

Concurrency Control across User Interactions

- Many applications need transaction support across user interactions
 - Can't use locking for long durations
 - Application level concurrency control
 - Each tuple has a version number
 - Transaction notes version number when reading tuple
 - **select r.balance, r.version into :A, :version from r where acctId = 23**
 - When writing tuple, check that current version number is same as the version when tuple was read
 - **update r set r.balance = r.balance + :deposit, r.version = r.version+1 where acctId = 23 and r.version = :version**
- partiiun vertinya sesuai sm
 yg dimau
 kl versi lhr udh diwrote sm transactor
 lain, otomatis update gagal



Concurrency Control across User Interactions

- Equivalent to **optimistic concurrency control without validating read set**
 - Unlike SI, reads are not guaranteed to be from a single snapshot.
 - Does not guarantee serializability
 - But avoids some anomalies such as “lost update anomaly”
- Used internally in Hibernate ORM system
- Implemented manually in many applications
- Version numbers stored in tuples can also be used to support first committer wins check of snapshot isolation

tdk menjamin pembacaan dr snapshot yg sama



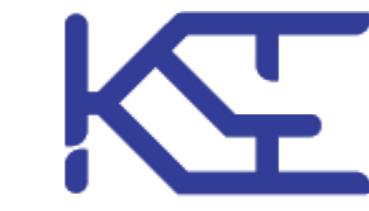
KNOWLEDGE & SOFTWARE ENGINEERING

©Silberschatz et.al. (2019)

[modified by Tim Pengajar IF3140 Semester 1 2024/2025]



End of Topic



KNOWLEDGE & SOFTWARE ENGINEERING

©Silberschatz et.al. (2019)

[modified by Tim Pengajar IF3140 Semester 1 2024/2025]