More Concurrency Control

Databases, Aarhus University

Ira Assent

Intended learning outcomes

- Be able to
  - Describe and use optimistic concurrency control
  - Discuss and apply certify and multigranularity locking

ira@cs.au.dk

Recap: Concurrency Control

- `SET TRANSACTION ISOLATION LEVEL`
  - `SERIALIZABLE` (full ACID), `REPEATABLE READ` (no dirty reads, but more rows may appear), `READ COMMITTED` (no dirty reads, but not necessarily the same value every time!), `READ UNCOMMITTED` (dirty reads possible)
- Recoverable schedule: no commit until all transactions that have written an item that is read have committed also
- Cascadeless schedule: read only items written by committed transactions
  - Avoids cascading rollback where transactions that read from aborted transaction must rollback
- Strict Schedule: neither read or write item until the last transaction that wrote it has committed
- Timestamp-based concurrency control:
  - Each transaction is given a timestamp when entering system
  - Access to a data item must be in timestamp order
    - If T wants to read X, X's write-timestamp must precede T's
    - If T wants to write X, X's read- and write-timestamp must precede T's
    - Update X's timestamp if applicable
  - No locking!

ira@cs.au.dk

Recap: multiversion concurrency control

- Multiversion
  - Maintain older versions of data items
  - When reading, allocate the right version to the read operation transaction → never rejected!
  - Storage (RAM and disk) is required to maintain multiple versions

- When T reads a version of Q, that version must be written by a transaction with an earlier timestamp than that of T (and the oldest among all that qualify)
- When T writes a version of Q, that version must not have been written or read by a transaction with a younger timestamp
  - Otherwise abort T if out of order (occurs only for write attempts)

ira@cs.au.dk

Multiversion timestamps

- Assume $X_1, X_2, \ldots, X_n$ are the versions of a data item X created by a write operation of transactions
  - With each X_i a read_TS (read timestamp) and a write_TS (write timestamp) are associated
- **read_TS(X_i)**: The read timestamp of X_i is the largest of all the timestamps of transactions that have successfully read version X_i (= most recent read)
- **write_TS(X_i)**: The write timestamp of X_i that wrote the value of version X_i (= the write that created this version)

➢ A new version of X_i is created only by a write operation

ira@cs.au.dk

Rules in multiversion timestamping

- To ensure serializability, the following two rules are used
- Rule 1: reject T if it **attempts to overwrite a version** (with the most recent timestamp that is still earlier than its own timestamp, so in the right creation order) if that was **already read by a younger** T' (meaning that T' would then have read out of order)
  1. If transaction T issues write_item(X) and version X_i has the highest write_TS(X_i) of all versions of X that is also less than or equal to TS(T), and read _TS(X_i) > TS(T), then abort and roll-back T; otherwise create a new version X_j and read_TS(X_j) = write_TS(X_j) = TS(T)
- Rule 2 guarantees that a **read will never be rejected**
  2. If transaction T issues read_item(X), find the version X_i that has the highest write_TS(X_i) of all versions of X that is also less than or equal to TS(T), then return the value of X_i to T, and set the value of read _TS(X_i) to the largest of TS(T) and the current read_TS(X_i)

ira@cs.au.dk

What happens in regular or multiversion timestamping?

A. T3 aborts in both.
B. T3 reads in both.
C. T3 aborts in regular only.
D. T3 reads in regular only.

T1	T2	T3	T4	A
150	200	175	225	RT=0
				WT=0
R1(A)				RT=150
W1(A)				WT=150
	R2(A)			RT=200
	W2(A)			WT=200
		R3(A)		?
			R4(A)	?

ira@cs.au.dk

Validation (Optimistic) Concurrency Control

- Core idea: just go ahead and "do the work", but **on local copies only**; before committing, check if there are any potential issues; if so, abort and restart; otherwise, write changes to database.
  - Optimistic: usually, no issues
  - Validation: we have to check before committing

- In this technique serializability only checked at commit time
  - transactions are aborted in case of non-serializable schedules
- Three phases:
  1. Read phase (where work happens; **both read and write!**)
     - Any changes are made in local workspace (copy of relevant data from database) only
  2. Validation phase (check serializability) – actual **concurrency control**
     - Idea: assign timestamps when starting validation, check other running transactions, **read-write/write-write conflicts only allowed from older to younger**
  3. Write phase (where local changes are actually written to the database if validation successful)

ira@cs.au.dk

Example Validation CC

T_1	T_2
BEGIN	BEGIN
READ	
R(A)	READ
	R(A)
	VALIDATE
	WRITE
	COMMIT
W(A)	
VALIDATE	
WRITE	
COMMIT	

- T_1 and T_2 just read and write (**read phase**)
  - Please note confusing naming
    - Read phase, because we only **read/write** on local copy, but **do not change** database
- For all transactions, maintain their read_set and write_set
  - Set of items being used
  - E.g. read_set(T_1)={A}, write_set(T_1)={A}, read_set(T_2)={A}, write_set(T_2)={}
- T_2 is done, enters **validation phase**
  - Check for read or write conflicts
    - If none, done (**write phase**)
      - Write phase, because now the **changes** from the local copy are applied to the database
    - When changes written to database, T_2 leaves system
- T_1 is done, enters validation phase…

ira@cs.au.dk

Validation Phase

- ”Did I do anything wrong?”
- Transaction T_i assigned timestamp $TS(T_i)$ at beginning of its validation phase
- Check the timestamp ordering with other transactions T_j that recently committed, i.e., in **write** phase, or in **validation** phase
  - Check that the reads and writes are in timestamp order
- For all other such recent $TS(T_j) < TS(T_i)$, then one of the following three conditions in (upcoming slides) must hold…

  if the first holds, easy, done right away

  if not, check second, if it holds, ok, done here

  if not, check last, if it holds, phew, done finally

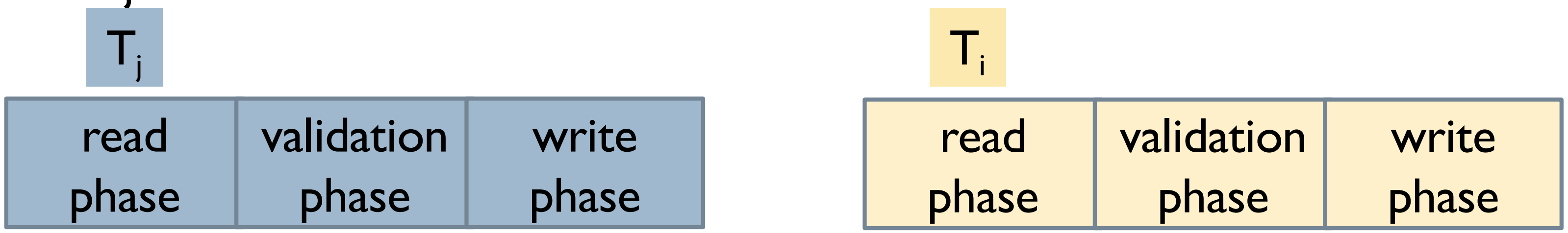
  else, abort
  - these are in increasing complexity – so often the simpler ones work, else need to go to most complex last one
- So, **for all** other such transactions, **at least one** condition needs to hold
  - There cannot be conflict with any other transaction (so all transactions need to be checked)
  - Any passed check condition suffices to show there is no conflict with that particular transaction

ira@cs.au.dk

1. Validation Condition for T_i

For all other recent $TS(T_j) < TS(T_i)$

- T_j completes all three phases before T_i begins



allows T_i to see T_j's changes, but execute in serial order: first T_j, then T_i

ira@cs.au.dk

1. Validation Condition for T_i

- T_j completes all three phases before T_i begins

allows T_i to see T_j's changes, but execute in serial order: first T_j, then T_i

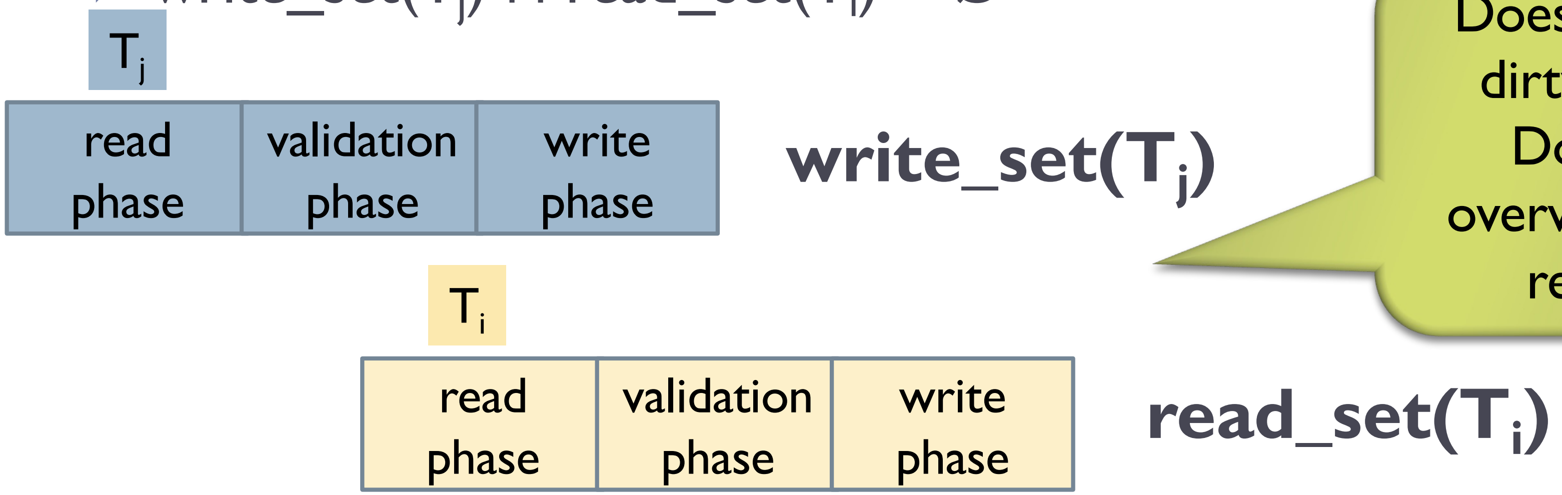
T_1	T_2
BEGIN	
READ	
R(A)	
W(A)	
VALIDATE	
WRITE	
COMMIT	
	BEGIN
	READ
	W(A)
	VALIDATE
	WRITE
	COMMIT

ira@cs.au.dk

2. Validation Condition for T_i

For all other such recent $TS(T_j) < TS(T_i)$

- T_j completes its write phase before T_i starts its write phase, and T_j does not change any item read by T_i
  - $write\_set(T_j) \cap read\_set(T_i) = \varnothing$



Does T_i read dirty data?
Does T_j overwrite T_i's reads?

- allows T_i to read items while T_j is still modifying items, but T_i does not read any item modified by T_j
- although T_i might overwrite items written by T_j, all of T_j's writes precede all of T_i's writes (timestamp order ok)

2. Validation Condition for T_i

- allows T_i to read items while T_j is still modifying items, but T_i does not read any item modified by T_j
- although T_i might overwrite items written by T_j, all of T_j's writes precede all of T_i's writes (timestamp order ok)

For all other such recent $TS(T_j) < TS(T_i)$

- T_j completes its write phase before T_i starts its write phase, and T_j does not write to any item read by T_i
  - $write\_set(T_j) \cap read\_set(T_i) = \varnothing$

T_1	T_2
BEGIN	BEGIN
READ	
R(A)	
W(A)	
	READ
	R(A)
	VALIDATE
	WRITE
	COMMIT
VALIDATE	
WRITE	
COMMIT	

VALID

Safe to commit T_1 because we know that T_2 does not write A

Database

Item	Value	W-TS
A	123	0
-	-	-

T_1 Workspace

Item	Value	W-TS
A	456	∞
-	-	-

T_2 Workspace

Item	Value	W-TS
A	123	0
-	-	-

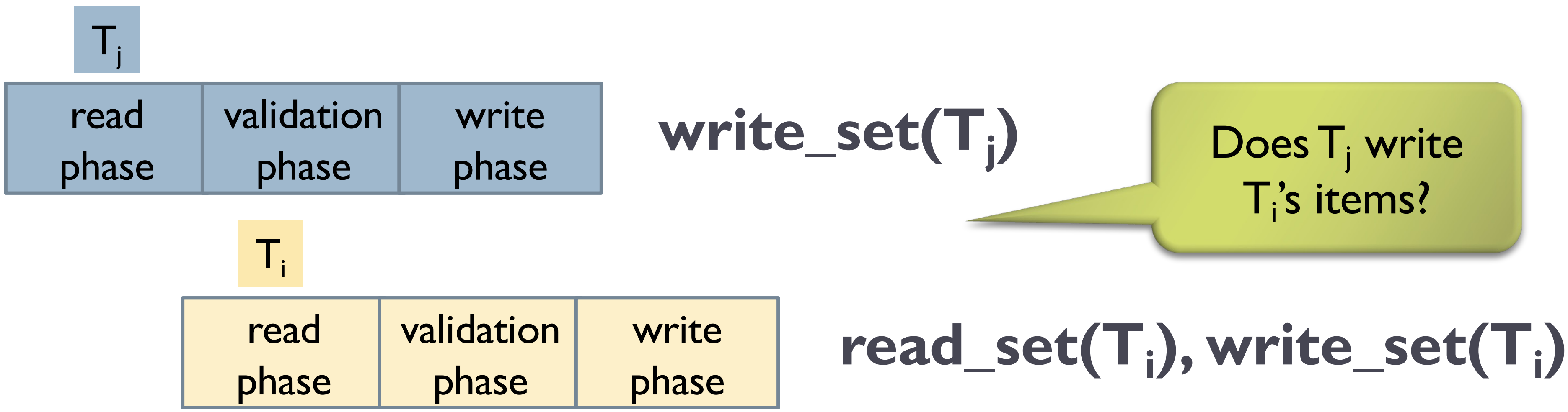
When T_1 validates, T_2 has finished write phase and its write_set is empty, so no overlap with what T_1 reads

ira@cs.au.dk

3. Validation Condition for T_i

For all other such recent $TS(T_j) < TS(T_i)$

- T_j completes its read phase before T_i completes its read phase
- And T_j does not write to any item that is either read or written by T_i:
  - $write\_set(T_j) \cap read\_set(T_i) = \varnothing$
  - $write\_set(T_j) \cap write\_set(T_i) = \varnothing$



allows T_i and T_j to read and write items at the same time, but does not allow T_i to read or write items written by T_j

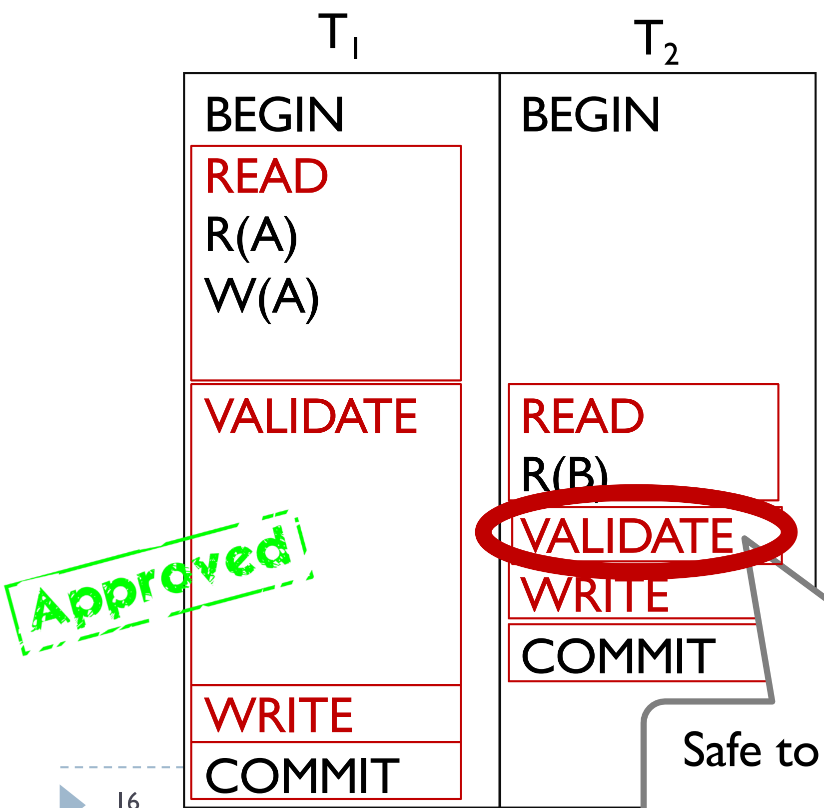
ira@cs.au.dk

3. Validation Condition for T_i

allows T_i and T_j to read and write items at the same time, but does not allow T_i to read or write items written by T_j

For all other such recent $TS(T_j) < TS(T_i)$

- T_j completes its read phase before T_i completes its read phase
- And T_j does not write to any item that is either read or written by T_i:
  - $write\_set(T_j) \cap read\_set(T_i) = \emptyset$
  - $write\_set(T_j) \cap write\_set(T_i) = \emptyset$



Item	Value	W-TS
A	456	1
B	XYZ	0

T1 Workspace

Item	Value	W-TS
A	456	∞

T2 Workspace

Item	Value	W-TS
B	XYZ	0

Safe to commit T_2 because T_2 does not use T_1s changes

ira@cs.au.dk

Validating conditions summarized

- When validating T_i, then for each T_j committed or in validation phase
  - 1. condition simplest: check first; if false then 2. is checked and if 2. is false then 3. is checked
  - If none of these conditions holds, the validation fails and T_i is aborted
- Conditions all try to check if transactions worked in a conflict-free manner
  - 1. condition allows T_i to see T_j's changes, but execute in validation timestamp order
  - 2. condition allows T_i to read items while T_j is still modifying items, but T_i does not read any item modified by T_j.; although T_i might overwrite items written by T_j, all of T_j's writes precede all of T_i's writes
  - 3. condition allows T_i and T_j to write itemss at the same time, and thus have even more overlap in time, but sets of items used by T_i cannot contain changes by T_j
  - Checking requires maintaining lists of items read and written by each transaction
  - While validating, no other transaction may commit; otherwise, may miss conflicts with respect to newly committed transaction

ira@cs.au.dk

Validation CC in practice

- Record read set and write set while transactions are running and write into private workspace
- Execute Validation and Write phase inside a protected critical section at the end of transaction lifetime
- Works well when number of conflicts is low
  - For large database and balanced workload, low probability of conflict, so locking wasteful
- High overhead for copying data locally
- Validation/Write phase bottlenecks
- Aborts are more wasteful because they only occur after a transaction has already executed

ira@cs.au.dk

VCC: allowed?

A. Yes
B. Yes, if $TS(T_1) < TS(T_2)$
C. Yes, if $TS(T_1) > TS(T_2)$
D. Yes, if T_1 ends its write phase before T_2 ends its write phase
E. Yes, if T_2 ends it write phase before T_1 ends its write phase
F. No

T_1	T_2
$R_1(Y)$	
	$R_2(Y)$
	$W_2(Y)$
	$R_2(X)$
	$W_2(X)$
$R_1(X)$	

(1) T_i completes all three phases before T_j begins
(2) T_j completes its write phase before T_i starts its write phase, and T_j does not write to any item read by T_i : $write\_set(T_j) \cap read\_set(T_i) = \emptyset$
(3) T_j completes its read phase before T_i completes its read phase and T_j does not write to any item that is either read or written by T_i :
- $write\_set(T_j) \cap read\_set(T_i) = \emptyset$
- $write\_set(T_j) \cap write\_set(T_i) = \emptyset$

ira@cs.au.dk

Multiple Granularity Locking

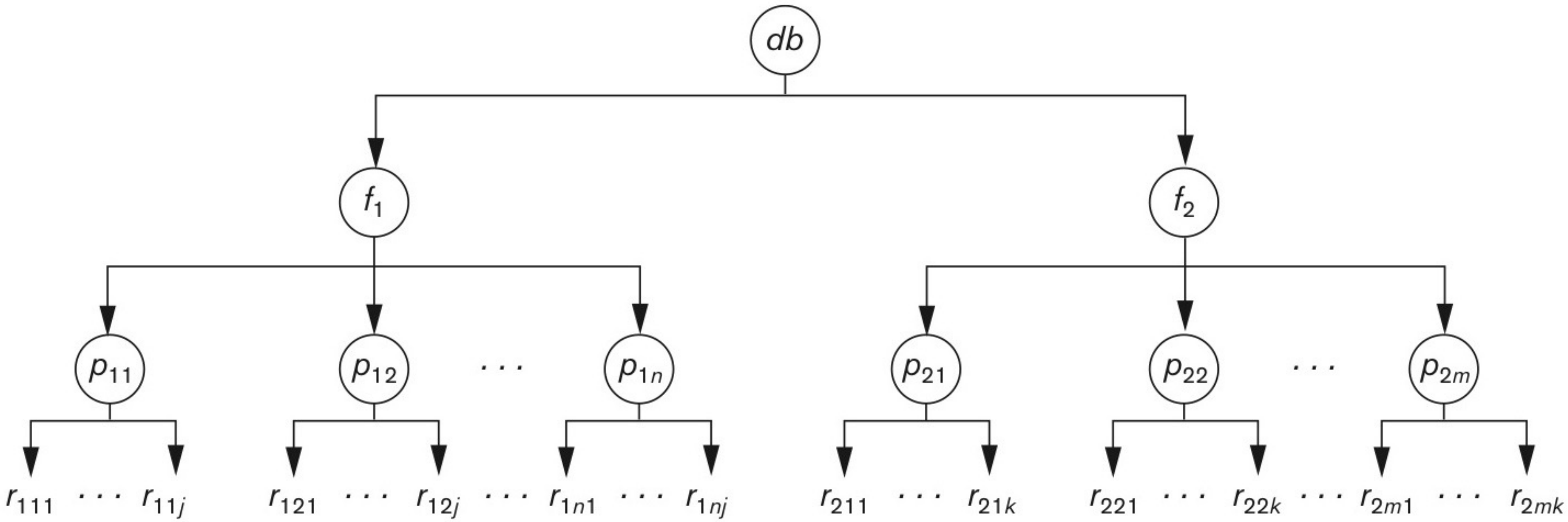
Granularity of data items

- A lockable unit of data defines its granularity
  - granularity can be coarse (entire database) or it can be fine (a tuple or an attribute of a relation)
- Data item granularity significantly affects concurrency control performance
  - Degree of concurrency is low for coarse granularity and high for fine granularity
- Example of data item granularity:
  1. A field of a database record (an attribute of a tuple)
  2. A database record (a tuple or a relation)
  3. A disk block
  4. An entire file
  5. The entire database

ira@cs.au.dk

Example granularity hierarchy

- The following diagram illustrates a hierarchy of granularity from coarse (database) to fine (record)
  - Top (root): all data
  - Bottom-most level (leaves): individual records



ira@cs.au.dk

Lock hierarchy

- To manage such hierarchy, in addition to read and write, three additional locking modes, called intention lock modes are defined:
  - **Intention-shared (IS)**: indicates that a shared lock(s) will be requested on some descendent nodes(s)
  - **Intention-exclusive (IX)**: indicates that an exclusive lock(s) will be requested on some descendent node(s)
  - **Shared-intention-exclusive (SIX)**: indicates that the current node is locked in shared mode but an exclusive lock(s) will be requested on some descendent nodes(s)

Locks compatibility table

- Locks are applied using the following compatibility matrix:

	IS	IX	S	SIX	X
IS	Yes	Yes	Yes	Yes	No
IX	Yes	Yes	No	No	No
S	Yes	No	Yes	No	No
SIX	Yes	No	No	No	No
X	No	No	No	No	No

Intention-shared (IS)
Intention-exclusive (IX)
Shared (S)
Shared-intention-exclusive (SIX)
Exclusive (X)

ira@cs.au.dk

Producing multigranularity schedules

- The set of rules which must be followed for producing serializable schedules using multiple granularity locking are
  1. The lock compatibility must adhered to
  2. The root of the tree must be locked first, in any mode
  3. A node N can be locked by a transaction T in S or IS mode only if the parent node is already locked by T in either IS or IX mode
  4. A node N can be locked by T in X, IX, or SIX mode only if the parent of N is already locked by T in either IX or SIX mode
  5. T can lock a node only if it has not unlocked any node (to enforce 2PL policy)
  6. T can unlock a node, N, only if none of the children of N are currently locked by T

ira@cs.au.dk

VOTE

Right after T_1 starts (before it completes anything), T_2 starts. What happens?

T_1: `SELECT * FROM R WHERE a="x";`

T_2: `UPDATE R SET b="y" WHERE a="z";`

R	a	b
a1	x	u
a2	x	v
a3	z	w

A. T_1 gets IS lock on R, S lock an a_1, a_2. T_2 gets IX lock on R, waits for T_1 to unlock

B. T_1 gets IS lock on R, S lock on a_1, a_2. T_2 gets IX on R, X lock on a_3

C. T_1 gets IX lock on R, X lock on a_1, a_2. T_2 gets IS lock on R, S lock on a_3

D. T_1 gets IX lock on R, S lock on a_1, a_2. T_2 waits for T_1 to unlock

ira@cs.au.dk

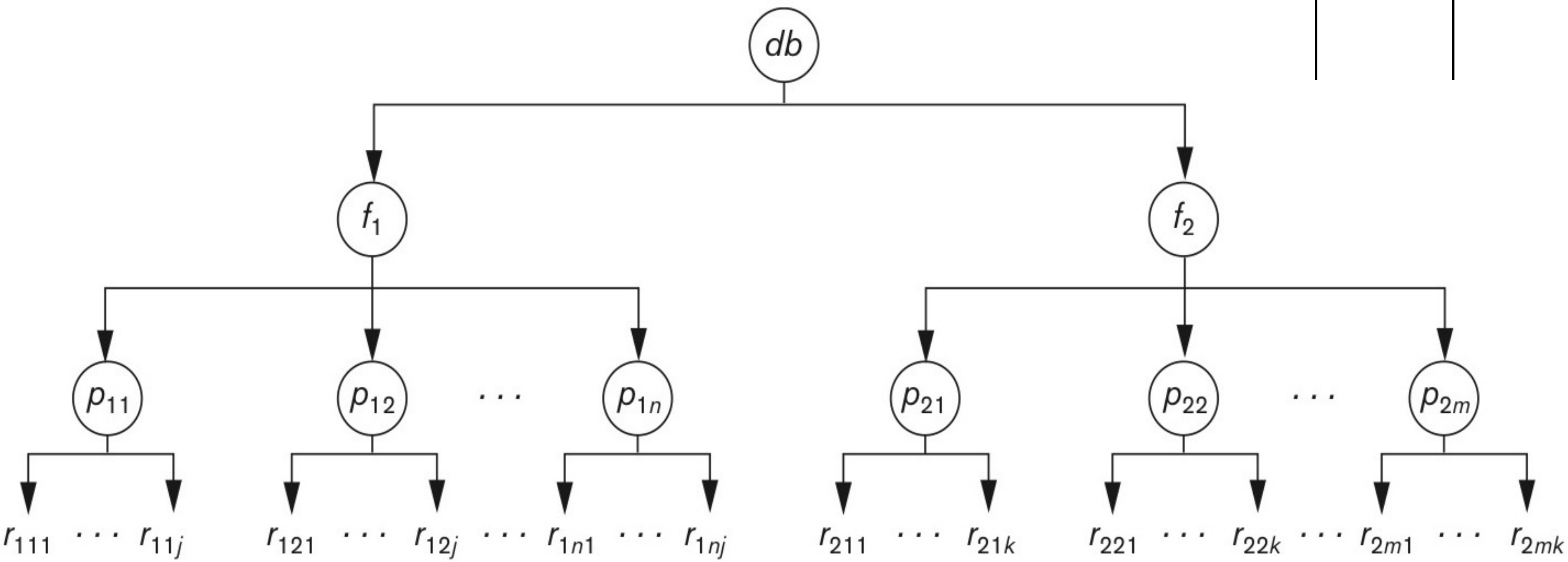
Example

T_1 wants to update record r_{111} and r_{211}

T_2 wants to update all records on page p_{12}

T_3 wants to read record r_{11j} and the entire f_2 file

T_1	T_2	T_3



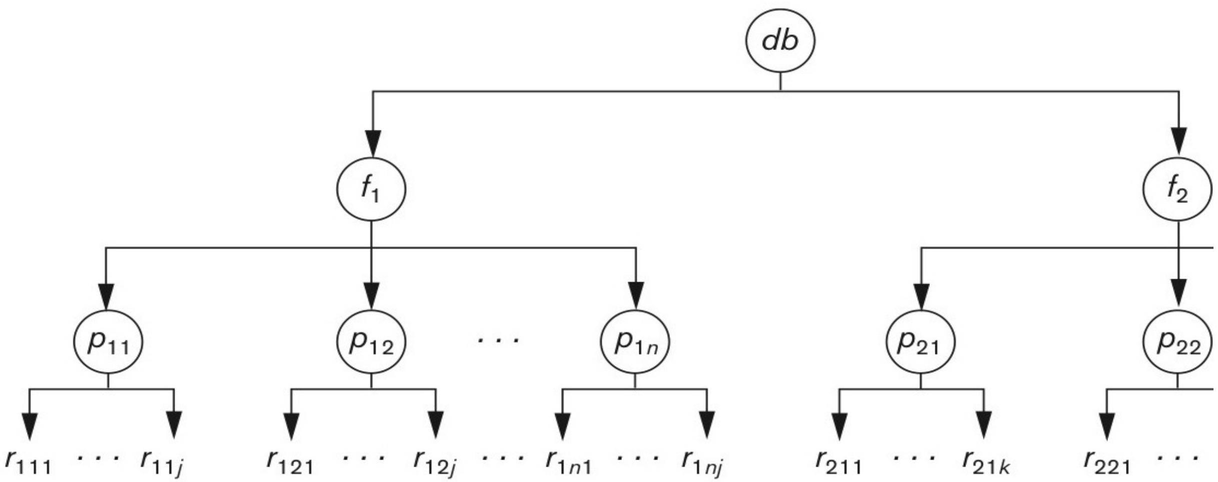
ira@cs.au.dk

Example

T_1 wants to update record r_{111} and r_{211}

T_2 wants to update all records on page p_{12}

T_3 wants to read record r_{11j} and the entire f_2 file



T_1	T_2	T_3
IX(*db*)		
IX(f_1)		
	IX(*db*)	
		IS(*db*)
		IS(f_1)
		IS(p_{11})
IX(p_{11})		
X(r_{111})		
	IX(f_1)	
	X(p_{12})	
		S(r_{11j})
IX(f_2)		
IX(p_{21})		
X(p_{211})		
unlock(r_{211})		
unlock(p_{21})		
unlock(f_2)		
		S(f_2)
	unlock(p_{12})	
	unlock(f_1)	
	unlock(*db*)	
unlock(r_{111})		
unlock(p_{11})		
unlock(f_1)		
unlock(*db*)		
		unlock(r_{11j})
		unlock(p_{11})
		unlock(f_1)
		unlock(f_2)
		unlock(*db*)

Certify Locks Concept

- Multiversion Two-Phase Locking Using Certify Locks
  - Allow a transaction T' to read a data item X while it is write-locked by a conflicting transaction T
  - This is accomplished by maintaining two versions of each data item X where one version must have been written by some committed transaction
    - The second "local version" created when a transaction acquires a write lock
    - This means a write operation always creates a new version of X
  - So, three *modes* of locks: read, write, certify
    - Write lock is no longer fully exclusive, reads possible
    - Certify lock new: fully exclusive

ira@cs.au.dk

The steps in Certify Locks Concurrency Control

1. X is committed version of a data item
2. When T wishes to write X, T creates a second version X' after obtaining write lock on X
3. Other transactions continue to read X
4. When T is ready to commit, it obtains certify lock on X'
5. The committed version X becomes X'
6. T releases its certify lock on X', which is new committed version of X now

ira@cs.au.dk

Lock compatibility compared

- In standard locking, compatibility:
  - if an item is read locked, then other read locks can be granted, else denied
- In certify locking,
  - If an item is read or write locked then other read locks can be granted; but not additional writes (limits number of versions)
  - Certify locks are exclusive and are a form of lock upgrade from the write lock

Lock compatibility table for read/write locking scheme

	Read	Write
Read	Yes	No
Write	No	No

Lock compatibility table for read/write/certify locking scheme

	Read	Write	Certify
Read	Yes	Yes	No
Write	Yes	No	No
Certify	No	No	No

ira@cs.au.dk

Certify locking pros and cons?

1. Deadlocks cannot occur
2. Read and write operations on same data item by different transactions cannot be processed concurrently
3. Transactions may have to wait when ready to commit
4. Cascading aborts may occur

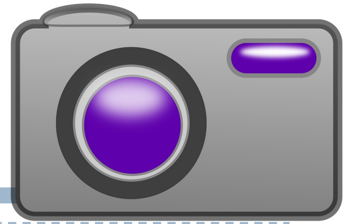
ira@cs.au.dk

Certify locks: pros and cons

- In multiversion 2PL with certify locks
  - read and write operations from conflicting transactions can be processed concurrently
    - Improves concurrency
  - But may delay transaction when it is ready to commit because of obtaining certify locks on all its writes
  - Avoids cascading abort but like strict two phase locking scheme conflicting transactions may get deadlocked

ira@cs.au.dk

Snapshot isolation



- Transactions see data items based on committed values of the items in the database snapshot (or database state) when the transaction starts
- Any changes after transaction start not seen
- No read locks, only write locks
- Writes create older versions in temporary version store (tempstore) along with creation timestamp
  - So that other transactions that started before can read the respective older version
  - Usually implemented using pointers from item to list of recent versions
- No phantom reads, dirty read, or nonrepeatable read as only committed values are seen
  - Rare anomalies can occur (not covered here, complex to detect)
    - Either ignore these issues or
    - Resolve in program (cumbersome) or
    - Use variant: serializable snapshot isolation (SSI) e.g. in PostGRES
  - Tradoff between runtime performance and accepting rare anomalies

ira@cs.au.dk

Summary

- Intended learning outcomes
  - Be able to
    - Discuss and apply certify and multigranularity locking
    - Describe and use optimistic concurrency control
    - Apply logging and basic recovery techniques

Acknowledgements: includes slide material by Faloutsos and Pavlo, CMU

What was this all about?

Guidelines for your own review of today's session

- Multiversion concurrency control is motivated by the observation that…
  - Multiversion means that the following information is maintained…
  - The rules applied based on this information are…
- Validation based concurrency control assumes that…
  - It validates when…
  - It makes use of the following checks…
- Multigranularity locking is better at…
  - We have to decide the levels of…
  - When accessing data, locks are granted as follows…
  - Compatibility means… and is summarized as…
- Certify locking takes the approach of…
  - The main steps are…
  - It's pros and cons are…

ira@cs.au.dk