B0B36DBS, BD6B36DBS: **Database Systems**

http://www.ksi.mff.cuni.cz/~svoboda/courses/172-B0B36DBS/

Lecture 9

# Database Transactions

Authors: **Tomáš Skopal**, **Irena Holubová**

Lecturer: **Martin Svoboda**, martin.svoboda@fel.cvut.cz

20. 2. 2018

**Czech Technical University in Prague**, Faculty of Electrical Engineering

# Today’s lecture outline

- motivation and the ACID properties
- schedules („interleaved“ transaction execution)
  - serializability
  - conflicts
  - (non)recoverable schedule
- locking protocols
  - 2PL, strict 2PL, conservative 2PL
  - deadlock and prevention
  - phantom
- alternative protocols

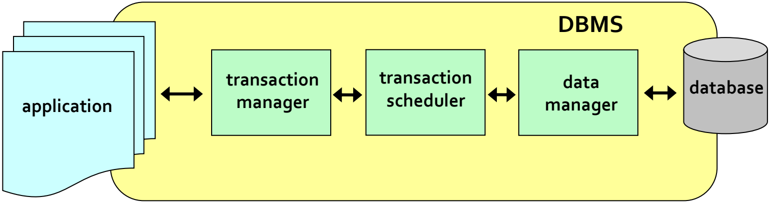
# Motivation

- problem: we need to execute complex database operations
  - e.g., stored procedures, triggers, etc.
  - in a multi-user and parallel environment
- database transaction
  - **sequence of actions** on database objects (+ others like arithmetic, etc.)
- example:
  - Let us have a bank database with table ***Accounts*** and the following transaction to transfer the money (pseudocode):

```
transaction PaymentOrder(amount, fromAcc, toAcc)
    {
     SELECT Balance INTO X FROM Accounts WHERE accNr = fromAcc;
     if (X < amount) AbortTransaction("Not enough money!");
     UPDATE Accounts SET Balance = Balance – amount WHERE accNr = fromAcc;
     UPDATE Accounts SET Balance = Balance + amount WHERE accNr = toAcc;
     CommitTransaction;
     }
```

# Transaction management in DBMS

- application launches transactions
- transaction manager executes transactions
- scheduler dynamically schedules the parallel transaction execution, producing a **schedule** (history)
- data manager executes partial operation of transactions



# Transaction management in DBMS

- transaction termination
  - **successful** – terminated by **COMMIT** command in the transaction code
    - the performed actions are confirmed
  - **unsuccessful** – transaction is cancelled
    1. termination by the transaction code **– ABORT** (or **ROLLBACK**) command
       - user can be notified
    2. system abort – DBMS aborts the transaction
       - some integrity constraint is violated – user is notified
       - by transaction scheduler (e.g., a deadlock occurs) – user is not notified
    3. system failure – HW failure, power loss – transaction must be restarted
- main objectives of transaction management
  - enforcement of **ACID properties**
  - maximal performance (throughput)
    - parallel/concurrent execution of transactions

# ACID – desired properties of transaction management

- **A**tomicity – partial execution is not allowed (all or nothing)
  - prevents from incorrect transaction termination (or failure)

  = consistency at the DBMS level
- **C**onsistency

  ?
  - any transaction will bring the database from one **consistent** (valid) state to another

  = consistency at application level
- **I**solation
  - transactions executed in parallel do not "see" effects of each other unless committed
  - parallel/concurrent execution is necessary to achieve high throughput
- **D**urability
  - once a transaction has been committed, it will remain so, even in the event of power loss, crashes, or errors
  - logging necessary (log/journal maintained)

# Transaction

- an executed transaction is a sequence of actions

$$T = <A_T^1,\ A_T^2,\ \ldots\ ,\ \text{COMMIT or ABORT}>$$

- basic database actions (operations)
- for now consider a **static database** (no inserts/deletes, just updates), let **A** be a database object (table, row, attribute in row)
  - we omit other actions such as control construct (if, for), etc.
- **READ**(A) – reads A from database
- **WRITE**(A) – writes A to database
- **COMMIT** – confirms executed actions as valid, terminates transaction
- **ABORT** – cancels executed actions, terminates transaction (with error)
- SQL commands **SELECT**, **INSERT**, **UPDATE**, could be viewed as transactions implemented using the basic actions (in SQL command **ROLLBACK** is used instead of abort)

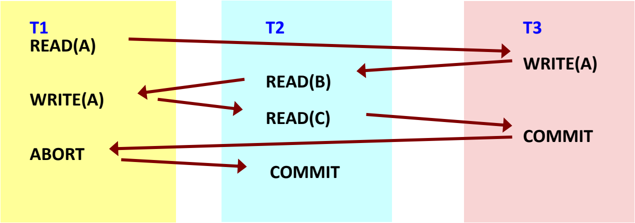
Example:

Subtract 5 from A (some attribute), such that A>0.

```
T = <READ(A),                    // action 1
if (A ≤ 5) then ABORT
else WRITE(A – 5),               // action 2
COMMIT>                          // action 3
or
T = <READ(A),                    // action 1
if (A ≤ 5) then ABORT            // action 2
else … >
```

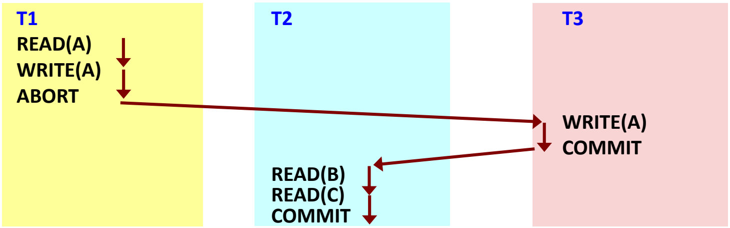
# Transaction programs vs. schedules

- **database program**
  - "design-time" (not running) piece of code (that will be executed as a transaction)
  - i.e., nonlinear – branching, loops, jumps
- **schedule** (history) is a sorted list of actions coming from several transactions (i.e., transactions as interleaved)
  - „runtime" history **of already concurrently executed** actions of **several** transactions
  - i.e., linear – sequence of primitive operations, w/o control constructs



# Serial schedules

- specific schedule, where all actions of a transaction are coupled together
  - no action interleaving
- given a set *S* of transactions, we can obtain |S|! serial schedules
  - from the definition of ACID properties, all the schedules are equivalent – it does not matter if one transaction is executed before or after another one
    - if it matters, they are not independent and so they should be merged into single transactions
- example:



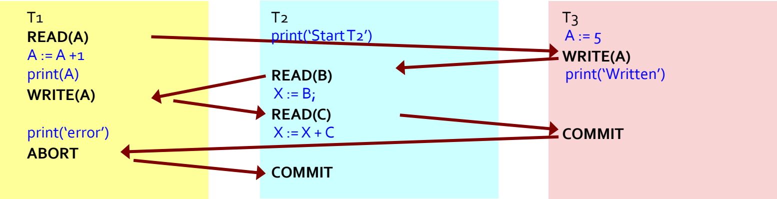
T1
READ(A)
WRITE(A)
ABORT

T2
READ(B)
READ(C)
COMMIT

T3
WRITE(A)
COMMIT

# Why to interleave transactions?

- every schedule leads to interleaved **sequential** execution of transactions (there is no parallel execution of database operations)
  - simplified model justified by single storage device
- Question: So why to interleave transactions when the number of steps is the same as in a serial schedule?
- two reasons
  - parallel execution of non-database operations with database operations
  - response proportional to transaction complexity (e.g., OldestEmployee vs. ComputeTaxes)
- example



T1
**READ(A)**
A := A +1
print(A)
**WRITE(A)**

print('error')
**ABORT**

T2
print('Start T2')

**READ(B)**
X := B;
**READ(C)**
X := X + C

**COMMIT**

T3
A := 5
**WRITE(A)**
print('Written')

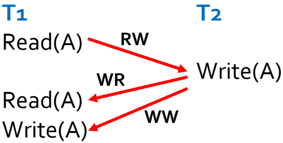
**COMMIT**

# Serializability

- a schedule is **serializable** if its execution leads to consistent database state, i.e., if the schedule is **equivalent to any serial schedule**
  - for now we consider only committed transactions and a static database
  - note that non-database operations are not considered so that consistency cannot be provided for non-database state (e.g., print on console)
  - it does not matter which serial schedule is equivalent (independent transactions)
- strong property
  - secures the Isolation and Consistency in ACID
- **view serializability** extends serializability by including aborted transactions and dynamic database
  - however, testing is NP-complete, so it is not used in practice
  - instead, **conflict serializability** + other techniques are used

# “Dangers” caused by interleaving

- to achieve serializability (i.e., consistency and isolation), the action of interleaving cannot be arbitrary
- there exist 3 types of local dependencies in the schedule, so-called conflict pairs
- four possibilities of reading/writing the same resource in schedule
  - read-read – ok, by reading the transactions do not affect each other
  - write-read (WR) – T1 writes, then T2 reads – reading uncommitted data
  - read-write (RW) – T1 reads, then T2 writes – unrepeatable reading
  - write-write (WW) – T1 writes, then T2 writes – overwrite of uncommitted data



T1 | T2
--- | ---
Read(A) |
 | Write(A)
Read(A) |
Write(A) |

# Conflicts (WR)

- reading uncommitted data (**write-read conflict**)
  - transaction T2 reads A that was earlier updated by transaction T1, but T1 did not commit so far, i.e., T2 reads potentially inconsistent data
    - so-called **dirty read**

Example: T1 transfers 1000 USD from account A to account B (A = 12000, B = 10000)
T2 adds 1% per account

| **T1** | **T2** |
|---|---|
| **R(A)** // A = 12000 | |
| A := A – 1000 | |
| **W(A)** // database is now inconsistent – account B still contains the old balance | |
| | **R(A)** // uncommitted data is read |
| | **R(B)** |
| | A := 1.01*A |
| | B := 1.01*B |
| | **W(A)** |
| | **W(B)** |
| | **COMMIT** |
| **R(B)** // B = 10100 | |
| B := B + 1000 | |
| **W(B)** | |
| **COMMIT** // inconsistent database, A = 11110, B = 11100 | |

# Conflicts (RW)

- unrepeatable read (**read-write conflict**)
  - transaction T2 writes A that was read earlier by T1 that didn't finish yet
  - T1 cannot repeat the reading of A (A now contains another value)
    - so-called **unrepeatable read**

Example: T1 transfers 1000 USD from account A to account B (A = 12000, B = 10000)
T2 adds 1% per account

| **T1** | | **T2** | |
|---|---|---|---|
| **R(A)** | // A = 12000 | | |
| | | **R(A)** | |
| | | **R(B)** | |
| | | A := 1.01*A | |
| | | B := 1.01*B | |
| | | **W(A)** | // update of A |
| | | **W(B)** | |
| | | **COMMIT** | |
| // database now contains A = 12120 | | | |
| **R(B)** | | | |
| A := A – 1000 | | | |
| **W(A)** | | | |
| B := B + 1000 | | | |
| **W(B)** | | | |
| **COMMIT** | | // inconsistent database, A = 11000, B = 11100 | |

# Conflicts (WW)

- overwrite of uncommitted data (**write-write conflict**)
  - transaction T2 overwrites A that was earlier written by T1 that still runs
  - loss of update (original value of A is lost)
    - so-called **blind write** (update of unread data)

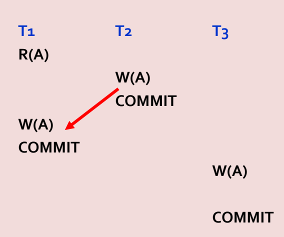
Example: Set the same price to all DVDs.
*(let’s have two instances of this transaction, one setting price to 10 USD, second 15 USD)*

| **T1** | **T2** | |
|---|---|---|
| | DVD1 := 15 | |
| | **W(DVD1)** | |
| DVD2 := 10 | | |
| **W(DVD2)** | | |
| | DVD2 := 15 | |
| | **W(DVD2)** | // overwrite of uncommitted data |
| | **COMMIT** | |
| DVD1 := 10 | | |
| **W(DVD1)** | | |
| **COMMIT** | // inconsistent database, DVD1 = 10, DVD2 = 15 | |

# Conflict serializability

- two schedules are **conflict equivalent** if they share the set of conflict pairs
- a schedule is **conflict serializable** if it is conflict-equivalent to some serial schedule, i.e., there are no "real" conflicts
  - more restrictive than serializability (defined only by consistency preservation)
- conflict serializability alone does not consider:
  - cancelled transactions
    – ABORT/ROLLBACK, so the schedule could be **unrecoverable**
  - dynamic database (inserting / deleting database objects)
    – so-called **phantom** may occur
  - hence, conflict serializability is not sufficient condition to provide ACID (**view serializability** is ultimate condition)

Example: schedule, that is **serializable** (serial schedule <T1, T2, T3>), but **is not conflict serializable** (writes in T1 and T2 are in wrong order)



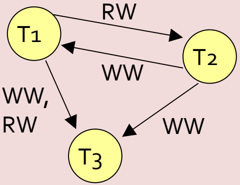
| T1 | T2 | T3 |
|---|---|---|
| R(A) | | |
| | W(A) | |
| | COMMIT | |
| W(A) | | |
| COMMIT | | |
| | | W(A) |
| | | COMMIT |

# Detection of conflict serializability

- **precedence graph** (also serializability graph) on a schedule
  - nodes $T_i$ are **committed** transactions
  - edges represent RW, WR, WW conflicts in the schedule
- schedule is conflict serializable
  if its precedence graph is **acyclic**

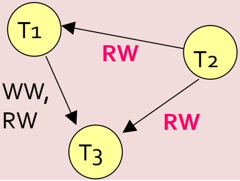
Example: not conflict serializable

| T1 | T2 | T3 |
|---|---|---|
| **R(A)** | | |
| | **W(A)** | |
| | **COMMIT** | |
| **W(A)** | | |
| **COMMIT** | | |
| | | **W(A)** |
| | | **COMMIT** |



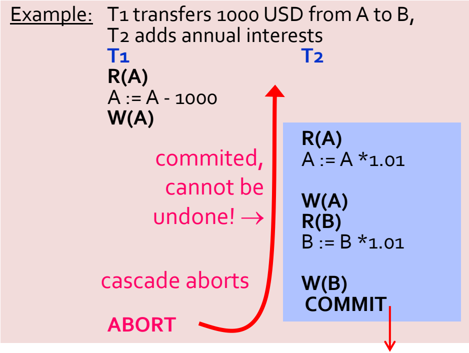
Example: conflict serializable

| T1 | T2 | T3 |
|---|---|---|
| **R(A)** | | |
| | **R(A)** | |
| | **COMMIT** | |
| **W(A)** | | |
| **COMMIT** | | |
| | | **W(A)** |
| | | **COMMIT** |



# Unrecoverable schedule

- at this moment we extend the transaction model by ABORT which brings another “danger” – **unrecoverable schedule**
  - one transaction aborts so that undos of every write must be done, however, this cannot be done for already committed transactions that read changes caused by the aborted transaction
    - durability property of ACID
- in **recoverable schedule** a transaction T is committed after all other transactions that affected T commit (i.e., they changed data later read by T)
- if reading changed data is allowed only for committed transactions, we also avoid **cascade aborts of transactions**

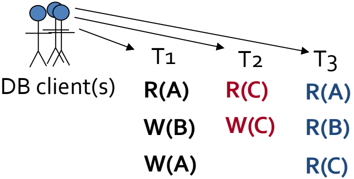


Example: T1 transfers 1000 USD from A to B, T2 adds annual interests

| T1 | T2 |
|---|---|
| **R(A)**<br>A := A - 1000<br>**W(A)** | |
| | **R(A)**<br>A := A *1.01<br>**W(A)**<br>**R(B)**<br>B := B *1.01<br>**W(B)**<br>**COMMIT** |
| **ABORT** | |

# Protocols for concurrent transaction scheduling

- transaction scheduler works under some **protocol** that allows to guarantee the ACID properties and maximal throughput
- **pessimistic control** (highly concurrent workloads)
  - locking protocols
  - time stamps
- **optimistic control** (not very concurrent workloads)
- why protocol?
  - the scheduler cannot create the entire schedule beforehand
  - scheduling is performed in local time context – dynamic transaction execution, branching parts in code



DB client(s)

| T1 | T2 | T3 |
|---|---|---|
| R(A) | R(C) | R(A) |
| W(B) | W(C) | R(B) |
| W(A) | | R(C) |

*Schedule*

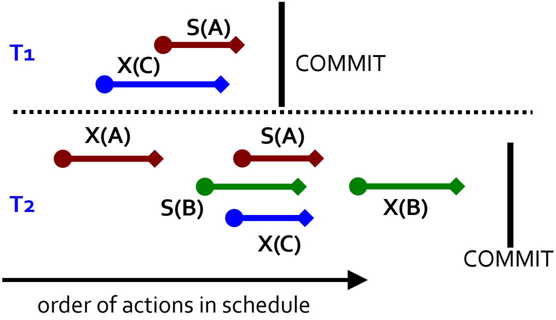
| T1 | T2 | T3 |
|---|---|---|
| R(A) | | |
| | R(C) | |
| | | R(A) |
| | W(C) | |
| | | R(B) |
| W(B) | | |
| W(A) | | |
| | | R(C) |

# Locking protocols

- locking of database entities can be used to control the order of reads and writes and so to secure the conflict serializability
- **exclusive locks**
  - **X(A)** locks A so that reads and writes of A are allowed only to the lock owner/creator
  - can be granted to just one transaction
- **shared locks**
  - **S(A)** – only reads of A are allowed
  - can be granted to (shared by) multiple transactions
- **unlocking** by **U(A)**
- if a lock that is not available is required for a transaction, the transaction execution is suspended and waits for releasing the lock
  - in the schedule, the lock request is denoted, followed by empty rows of waiting
- the un/locking code is added by the transaction scheduler
  - i.e., operation on locks appear just in the schedules, not in the original transaction code

# Example: schedule with locking

| T1 | T2 |
|---|---|
| | *X(A)* |
| *X(C)* | |
| | W(A) |
| R(C) | |
| W(C) | |
| | *U(A)* |
| *S(A)* | |
| | *S(B)* |
| R(A) | |
| | R(B) |
| *U(C)* | |
| | *X(C)* |
| *U(A)* | |
| | *S(A)* |
| COMMIT | |
| | W(C) |
| | R(A) |
| | *U(B)* |
| | *U(C)* |
| | *U(A)* |
| | *X(B)* |
| | W(B) |
| | *U(B)* |
| | COMMIT |



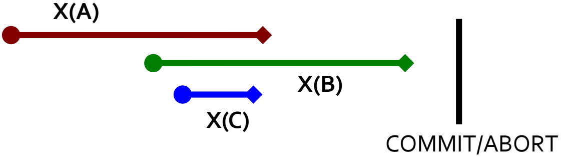
# Two-phase locking protocol (2PL)

**2PL protocol** applies two rules for building the schedule:

1) if a transaction wants to read (write) an entity A, it must first acquire a shared (exclusive) lock on A
2) transaction **cannot requests a lock**, if it already released one (regardless of the locked entity)

Two obvious phases – locking and unlocking

Example: 2PL adjustment of the second transaction in the previous schedule

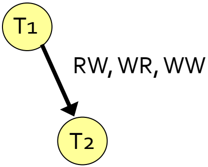


# Properties of 2PL

- the 2PL restriction of schedule ensures that the precedence graph is acyclic, i.e., the schedule is **conflict serializable**
- 2PL does **not guarantee recoverable schedules**

Example: 2PL-compliant schedule, but not recoverable, if T1 aborts

| T1 | T2 |
|---|---|
| **X(A)** | |
| **R(A)** | |
| **W(A)** | |
| **U(A)** | |
| | **X(A)** |
| | **R(A)** |
| | A := A *1.01 |
| | **W(A)** |
| | **X(B)** |
| | **U(A)** |
| | **R(B)** |
| | B := B *1.01 |
| | **W(B)** |
| | **U(B)** |
| | **COMMIT** |
| **ABORT / COMMIT** | |

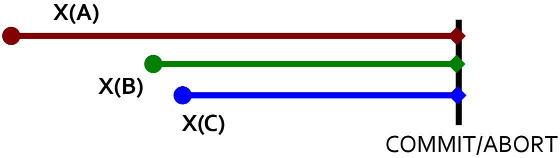


# Strict 2PL

**Strict 2PL protocol** makes the second rule of 2PL stronger, so that both rules become:

1) if a transaction wants to read (write) an entity A, it must first acquire a shared (exclusive) lock on A
2) **all locks are released at the transaction termination**

Example: strict 2PL adjustment of second transaction in the previous example



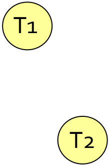
Insertions of U(A) are not needed (implicit at the time of COMMIT/ABORT).

# Properties of strict 2PL

- the 2PL restriction of schedule ensures that the precedence graph is acyclic, i.e., the schedule is **conflict serializable**
- moreover, strict 2PL ensures
  - schedule **recoverability**
  - avoids **cascade aborts**

Example: schedule built using strict 2PL

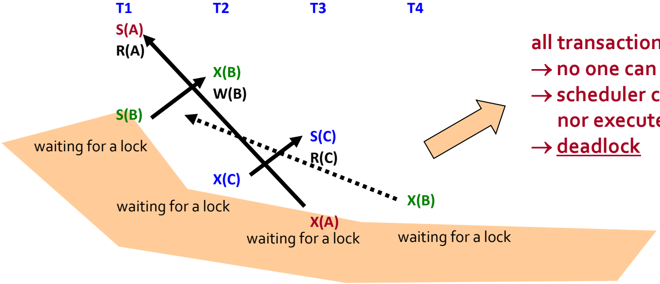
| T1 | T2 |
|---|---|
| S(A) | |
| R(A) | |
| | S(A) |
| | R(A) |
| | X(B) |
| X(C) | |
| R(C) | |
| | R(B) |
| | W(B) |
| | COMMIT |
| W(C) | |
| ABORT / COMMIT | |



# Deadlock

- during transaction execution it may happen that transaction $T_1$ requests a lock that was already granted to $T_2$, but $T_2$ cannot release it because it waits for another lock kept by $T_1$
  - could be generalized to multiple transactions,
    T1 waits for $T_2$, $T_2$ waits for $T_3$, ..., $T_n$ waits for $T_1$
- strict 2PL cannot prevent from deadlock (not speaking about the weaker protocols)

Example:



| T1 | T2 | T3 | T4 |
|---|---|---|---|
| S(A)<br>R(A) | | | |
| | X(B)<br>W(B) | | |
| | | S(C)<br>R(C) | |
| S(B) | | | |
| | X(C) | | |
| | | | X(B) |
| | | X(A) | |

waiting for a lock

waiting for a lock

waiting for a lock

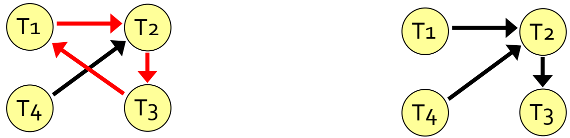
waiting for a lock

all transactions wait for a lock
→ no one can release a lock
→ scheduler cannot schedule
nor execute transactions
→ deadlock

# Deadlock detection

- deadlock can be detected by repeated checking the waits-for graph
- **waits-for graph** is a dynamic graph that captures the waiting of transactions for locks
  - nodes are active transactions
  - an edge denotes waiting of transaction for lock kept by another transaction
  - a cycle in the graph = **deadlock**

Example: waits-for graph for the previous example



**(a) T3 requests X(A)**

**(b) T3 does not request X(A)**

# Deadlock resolution and prevention

- deadlocks are usually not very frequent, so the **resolution** could be simple
  - abort of the waiting transaction and its restart (user will not notice)
  - testing waits-for graph – if a deadlock occurs, abort and restart a transaction in the cycle
    - such transaction is aborted, that
      - holds the smallest number of locks
      - performed the least amount of work
      - is far from completion
    - an aborted transaction is not aborted again (if another deadlock occurs)
- deadlocks could be **prevented**
  - prioritizing
    - each transaction has a priority (e.g., time stamp); if T1 requests a lock kept by T2, the lock manager chooses between two strategies
      - **wait-die** – if T1 has higher priority, it can wait, if not, it is aborted and restarted
      - **wound-wait** – if T1 has higher priority, T2 is aborted, otherwise T1 waits

# Coffman Conditions

- Deadlocks can arise if all of the following conditions hold simultaneously in a system
  - **Mutual exclusion** – resources can be held in a non-shareable mode
  - **Resource holding** (hold and wait) – additional resources may be requested even when already some resources are held
  - **No preemption** – resources can be released only voluntarily
  - **Circular wait** – transactions can request and wait for resources in cycles

- Unfulfillment of any of these conditions is enough to prevent deadlocks from occurring

# Phantom

- now consider dynamic database
  - allowing inserts and deletes
- if one transaction works with some *set* of data entities, while another transaction changes this set (inserts or deletes), it could lead to inconsistent database (inserializable schedule)
  - Why? T1 locks all entities that at the given moment are relevant
    - e.g., fulfill some WHERE condition of a SELECT command
  - during execution of T1 a new transaction T2 could logically extend the set of entities
    - i.e., at that moment the number of locks defined by WHERE would be larger
    - so that some entities are locked and some are not
- applied also to strict 2PL

# Example – phantom

**T1:** find the oldest male and female employees
(**SELECT * FROM** Employees ...) + **INSERT INTO** Statistics ...

**T2:** insert new employee Phill and delete employee Eve (employee replacement)
(**INSERT INTO** Employees ..., **DELETE FROM** Employees ...)

Initial state of the database: **{[Peter, 52, m], [John, 46, m], [Eve, 55, f], [Dana, 30, f]}**

| **T1** | **T2** |
|---|---|
| ***lock men, i.e.,*** | |
| **S(Peter)** | |
| **S(John)** | |
| **M = max{R(Peter), R(John)}** | |
| | **Insert(Phill, 72, m)** |
| | **X(Eve)** |
| | **Delete(Eve)** |
| | **COMMIT** |
| ***lock women, i.e.,*** | |
| **S(Dana)** | |
| **F = max{R(Dana)}** | |
| **Insert(M, F) // result is inserted into table Statistics** | |
| **COMMIT** | |

phantom

a new male employee can be inserted, although **all men** should be locked

Although the schedule is **strict 2PL** compliant, the result **[Peter, Dana]** is not correct as it does not follow the serial schedule T1, T2, resulting in **[Peter, Eve]**, nor T2, T1, resulting **[Phill, Dana]**.

# Phantom – prevention

- if there do not exist indexes, everything relevant must be locked
  - e.g., entire table or even multiple tables must be locked
- if there exist indexes (e.g., B$^+$-trees) on the entities defined by the „lock condition“, it is possible to “watch for phantom“ at the index level – **index locking**
  - external attempt for the set modification is identified by the index locks updated
  - as an index usually maintains just one attribute, its applicability is limited
- generalization of index locking is **predicate locking**, when the locks are requested for the logical sets, not particular data instances
  - however, this is hard to implement and so not used much in practice

# Optimistic (not locking) protocols

- if concurrently executed transactions are not often in conflict (not competing for resources), the locking overhead is unnecessarily large
- 3-phase optimistic protocol
  1. **Read**: transaction reads data from database but writes into its private local data space
  2. **Validation**: if the transaction wants to commit, it forwards the private data space to the transaction manager (i.e., request on database update)
     - the transaction manager decides if the update is in conflict with another transaction
       - if there is a conflict, the transaction is aborted and restarted
       - if not, the last phase takes place:
  3. **Write**: the private data space is copied into the database