Data Management
09 Transaction Processing

Matthias Boehm, **Arnab Phani**

Graz University of Technology, Austria
Computer Science and Biomedical Engineering
Institute of Interactive Systems and Data Science
BMK endowed chair for Data Management

Last update: May 09, 2022

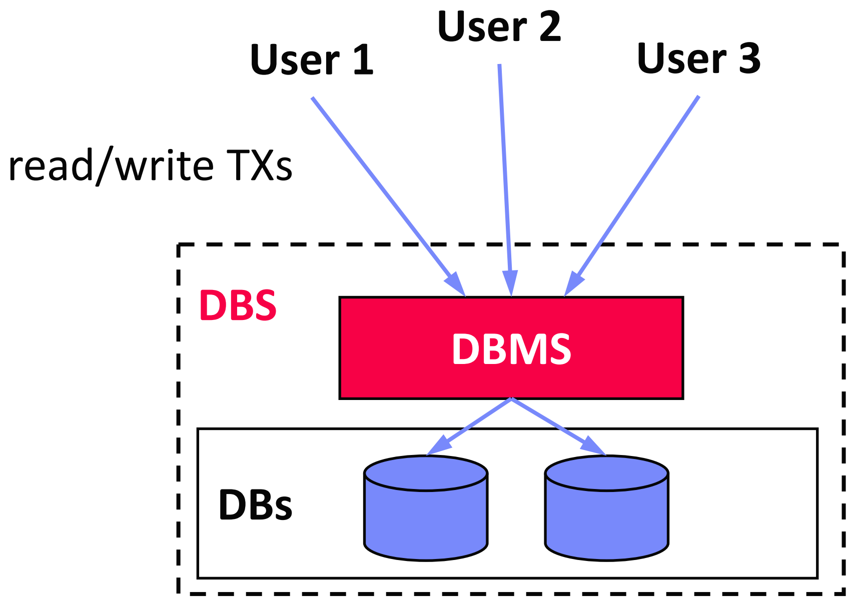
Announcements/Org

- **#1 Video Recording**
  - Link in **TUbe** & **TeachCenter** (lectures will be public)
  - Optional attendance (independent of COVID)
  - **Virtual lectures** (recorded) until end of the year
    https://tugraz.webex.com/meet/m.boehm

- **#2 Exercise Submissions**
  - **Grading Exercise 1:** uploaded, **Exercise 2:** end of May
  - **Exercise 3** (already published, discussed today) due May 31 + 7 late days

- **#3 CS Talks**
  - Eva Galperin (Director of Cybersecurity at EFF):
    Who Deserves Cybersecurity
  - Aula Alte Technik; **Jun 07, 5.30pm**

Transaction (TX) Processing



#1 Multiple users
➔ Correctness?

#2 Various failures
(TX, system, media)
➔ Reliability?

Deadlocks

Constraint violations

Network failure

Crash/power failure

Disk failure

- **Goal: Basic Understanding of Transaction Processing**
  - Transaction processing from user perspective
  - Locking and concurrency control to ensure **#1 correctness**
  - Logging and recovery to ensure **#2 reliability**

Agenda

- **Overview Transaction Processing**
- **Locking and Concurrency Control**
- **Logging and Recovery**
- **Exercise 3: Tuning and Transactions**

Additional Literature:

[**Jim Gray**, Andreas Reuter: Transaction Processing: Concepts and Techniques. **Morgan Kaufmann 1993**]

[Gerhard Weikum, Gottfried Vossen: Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery. **Morgan Kaufmann 2002**]

Overview Transaction Processing

Terminology of Transactions

- **Database Transaction**
  - A transaction (TX) is a **series of steps** that brings a database from a **consistent state** into another (not necessarily different) **consistent state**
  - **ACID properties** (atomicity, consistency, isolation, durability)

- **Terminology by Example**

#2 Start/**begin of TX** (BOT/BT)

#1 Isolation level (defined by addressed anomalies)

```
START TRANSACTION ISOLATION LEVEL SERIALIZABLE;
  UPDATE Account SET Balance=Balance-100
     WHERE AID = 107;
  UPDATE Account SET Balance=Balance+100
     WHERE AID = 999;

  SELECT Balance INTO lbalance
     FROM Account WHERE AID=107;
  IF lbalance < 0 THEN
     ROLLBACK TRANSACTION;
  END IF
COMMIT TRANSACTION;
```

#3 Reads and writes of data objects

#6 Savepoints (checkpoint for partial rollback)

#4 Abort/rollback TX (unsuccessful end of transaction, EOT/ET)

#5 Commit TX (successful end of transaction, EOT/ET)

Example OLTP Benchmarks

- **Online Transaction Processing (OLTP)**
  - Write-heavy database workloads, primarily with point lookups/accesses
  - **Applications:** financial, commercial, travel, medical, and governmental ops
  - **Benchmarks:** e.g., **TPC-C**, **TPC-E**, AuctionMark, SEATS (Airline), **Voter**

- **Example TPC-C**
  - 45% New-Order
  - 43% Payment
  - 4% Order Status
  - 4% Delivery
  - 4% Stock Level

```
New Order Transaction:
1) Get records describing a
   warehouse (tax), customer, district
2) Update the district to increment
   next available order number
3) Insert record into Order and NewOrder
4) For All Items
   a) Get item record (and price)
   b) Get/update stock record
   c) Insert OrderLine record
5) Update total amount of order
```

[http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-c_v5.11.0.pdf]

ACID Properties

[Theo Härder, Andreas Reuter: Principles of Transaction-Oriented Database Recovery. **ACM Comput. Surv. 15(4) 1983**]

- **Atomicity**
  - A transaction is executed atomically (**completely or not at all**)
  - If the transaction fails/aborts no changes are made to the database (**UNDO**)
- **Consistency**
  - A successful transaction ensures that all **consistency constraints are met** (referential integrity, semantic/domain constraints)
- **Isolation**
  - Concurrent transactions are executed in isolation of each other
  - **Appearance of serial transaction execution**
- **Durability**
  - **Guaranteed persistence** of all changes made by a successful transaction
  - In case of system failures, the database is recoverable (**REDO**)

Anomalies – Lost Update

TA1 updates points for Exercise 1

```
SELECT Pts INTO :points
   FROM Students WHERE Sid=789;

points += 23.5;

UPDATE Students SET Pts=:points
   WHERE Sid=789;
COMMIT TRANSACTION;
```

TA2 updates points for Exercise 2

```
SELECT Pts INTO :points
   FROM Students WHERE Sid=789;

points += 24.0;

UPDATE Students SET Pts=:points
   WHERE Sid=789;
COMMIT TRANSACTION;
```

Time

Student received 24 instead of 47.5 points
(lost update 23.5)

- **Problem: Write-write dependency**
- **Solution: Exclusive lock on write**

Anomalies – Dirty Read

TA1 updates points for Exercise 1

```
UPDATE Students SET Pts=100
   WHERE Sid=789;

ROLLBACK TRANSACTION;
```

TA2 updates points for Exercise 2

```
SELECT Pts INTO :points
   FROM Students WHERE Sid=789;

points += 24.0;

UPDATE Students SET Pts=:points
   WHERE Sid=789;
COMMIT TRANSACTION;
```

Time

Student received 124 instead of 24 points

- **Problem: Write-read dependency**
- **Solution: Read only committed changes; otherwise, cascading abort**

Anomalies – Unrepeatable Read

TA1 updates points for Exercise 1

TA2 runs statistics for Exercise 1

```
SELECT Pts INTO :p1
   FROM Students WHERE Sid=789;
```

```
START TRANSACTION;
  UPDATE Students SET Pts=Pts+23.5
    WHERE Sid=789;
COMMIT TRANSACTION;
```

```
...
```

modified value

```
SELECT Pts INTO :p2
   FROM Students WHERE Sid=789;
...
COMMIT TRANSACTION;
```

Time

TA2 sees only committed data but analysis corrupted as p1!=p2

- **Problem:** **Read-write dependency**
- **Solution:** **TA works on consistent snapshot of touched records**

Anomalies – Phantom

TA1 inserts missing student

TA2 runs statistics for Exercise 1

```
SELECT Avg(Pts) INTO :p1
   FROM Students WHERE Sid<1000;
```

```
START TRANSACTION;
  INSERT INTO Students
    VALUES (999, ..., 0);
COMMIT TRANSACTION;
```

```
...
```

added row
(harder to track because new database object)

```
SELECT Avg(Pts) INTO :p2
   FROM Students WHERE Sid<1000;
...
COMMIT TRANSACTION;
```

Time

TA2 sees only committed data but analysis corrupted as p1!=p2

- **Similar to non-repeatable read but at set level**
  (snapshot of accessed data objects not sufficient)

Isolation Levels

- **Different Isolation Levels**
  - **Tradeoff Isolation vs performance** per session/TX
  - SQL standard requires **guarantee against lost updates** for all

```
SET TRANSACTION
  ISOLATION LEVEL
  READ COMMITTED
```

- **SQL Standard Isolation Levels**

Isolation Level	Lost Update	Dirty Read (P1)	Unrepeatable Read (P2)	Phantom Read (P3)
READ UNCOMMITTED	No*	Yes	Yes	Yes
READ COMMITTED	No*	No	Yes	Yes
REPEATABLE READ	No*	No	No	Yes
[SERIALIZABLE]	No*	No	No	No

  - Serializable w/ highest guarantees (**pseudo-serial execution**)

* Lost update potentially w/ different semantics in standard

- **How can we enforce these isolation levels?**
  - **User:** set default/transaction isolation level (mixed TX workloads possible)
  - **System:** dedicated concurrency control strategies + scheduler

Excursus: A Critique of SQL Isolation Levels

[Hal Berenson, Philip A. Bernstein, Jim Gray, Jim Melton, Elizabeth J. O'Neil, Patrick E. O'Neil: A Critique of ANSI SQL Isolation Levels. **SIGMOD 1995**]

- **Summary**
  - **Criticism:** SQL standard isolation levels are ambiguous (strict/broad interpretations)
  - Additional anomalies: dirty write, cursor lost update, fuzzy read, read skew, write skew
  - Additional isolation levels: **cursor stability** and **snapshot isolation**
- **Snapshot Isolation (< Serializable)**
  - **Type of optimistic concurrency control** via multi-version concurrency control
  - TXs reads data from a snapshot of committed data when TX started
  - **TXs never blocked on reads**, other TXs data invisible
  - TX **T1 only commits if no other TX wrote the same data items** in the time interval of T1
- **Current Status?**

  [http://dbmsmusings.blogspot.com/2019/05/introduction-to-transaction-isolation.html]

  - "SQL standard that **fails to accurately define database isolation levels** and database vendors that attach liberal and non-standard semantics"

Excursus: Isolation Levels in Practice

- **Default and Maximum Isolation Levels for "ACID" and "NewSQL" DBs** [as of 2013]
  - 3/18 SERIALIZABLE by default
  - 8/18 did not provide SERIALIZABLE at all

Database	Default	Maximum
Actian Ingres 10.0/10S [1]	S	S
Aerospike [2]	RC	RC
Akiban Persistit [3]	SI	SI
Clustrix CLX 4100 [4]	RR	RR
Greenplum 4.1 [8]	RC	S
IBM DB2 10 for z/OS [5]	CS	S
IBM Informix 11.50 [9]	Depends	S
MySQL 5.6 [12]	RR	S
MemSQL 1b [10]	RC	RC
MS SQL Server 2012 [11]	RC	S
NuoDB [13]	CR	CR
Oracle 11g [14]	RC	SI
Oracle Berkeley DB [7]	S	S
Oracle Berkeley DB JE [6]	RR	S
Postgres 9.2.2 [15]	RC	S
SAP HANA [16]	RC	SI
ScaleDB 1.02 [17]	RC	RC
VoltDB [18]	S	S

RC: read committed, RR: repeatable read, SI: snapshot isolation, S: serializability, CS: cursor stability, CR: consistent read

[Peter Bailis, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, Ion Stoica: HAT, Not CAP: Towards Highly Available Transactions. **HotOS 2013**]

Beware of defaults, even though the SQL standard says **SERIALIZABLE** is the default

Locking and Concurrency Control

(Consistency and Isolation)

Overview Concurrency Control

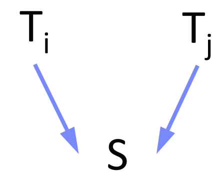
- **Terminology**
  - **Lock:** logical synchronization of TXs access to database objects (row, table, etc)
  - **Latch:** physical synchronization of access to shared data structures
- **#1 Pessimistic Concurrency Control**
  - Locking schemes (lock-based database scheduler)
  - Full serialization of transactions
- **#2 Optimistic Concurrency Control (OCC)**
  - Optimistic execution of operations, check of conflicts (validation)
  - Optimistic and timestamp-based database schedulers
- **#3 Mixed Concurrency Control (e.g., PostgreSQL)**
  - Combines locking and OCC
  - Might return **synchronization errors**

```
ERROR: could not serialize access
       due to concurrent update
ERROR: deadlock detected
```

Serializability Theory

- **Operations of Transaction T_j**
  - **Read and write operations** of A by T_j: **$r_j(A)$ $w_j(A)$**
  - **Abort of transaction** T_j: **a_j** (unsuccessful termination of T_j)
  - **Commit of transaction** T_j: **c_j** (successful termination of T_j)
- **Schedule S**
  - Operations of a transaction **T_j are executed in order**
  - Multiple transactions may be executed concurrently

  → **Schedule describes the total ordering of operations**



- **Equivalence of Schedules S1 and S2**
  - Read-write, write-read, and write-write dependencies on data object A executed in same order:

$$r_i(A) <_{S1} w_j(A) \Leftrightarrow r_i(A) <_{S2} w_j(A)$$
$$w_i(A) <_{S1} r_j(A) \Leftrightarrow w_i(A) <_{S2} r_j(A)$$
$$w_i(A) <_{S1} w_j(A) \Leftrightarrow w_i(A) <_{S2} w_j(A)$$

Serializability Theory, cont.

- **Example Serializable Schedules**
  - Input TXs
    T1: BOT $r_1(A)$ $w_1(A)$ $r_1(B)$ $w_1(B)$ c_1
    T2: BOT $r_2(C)$ $w_2(C)$ $r_2(A)$ $w_2(A)$ c_2
  - Serial execution $r_1(A)$ $w_1(A)$ $r_1(B)$ $w_1(B)$ c_1 $r_2(C)$ $w_2(C)$ $r_2(A)$ $w_2(A)$ c_2
  - Equivalent schedules
    $r_1(A)$ $r_2(C)$ $w_1(A)$ $w_2(C)$ $r_1(B)$ $r_2(A)$ $w_1(B)$ $w_2(A)$ c_1 c_2
    $r_1(A)$ $w_1(A)$ $r_2(C)$ $w_2(C)$ $r_1(B)$ $w_1(B)$ $r_2(A)$ $w_2(A)$ c_1 c_2
  - Wrong schedule $r_1(A)$ $r_2(C)$ $w_2(C)$ $r_2(A)$ $w_1(A)$ $r_1(B)$ $w_1(B)$ $w_2(A)$ c_1 c_2

- **Serializability Graph (conflict graph)**
  - Operation dependencies (read-write, write-read, write-write) aggregated
  - **Nodes:** transactions; **edges:** transaction dependencies
  - **Transactions are serializable** (via topological sort) **if the graph is acyclic**
  - **Beware:** Serializability Theory considers only successful transactions, which disregards anomalies like dirty read that might happen in practice

BREAK (and Test Yourself)

- **Given two transactions T_1 and T_2, which pairs of the following three schedules are equivalent? Explain for each pair (S_1-S_2, S_1-S_3, S_2-S_3) why they are equivalent or non-equivalent.** [5 points]
  - $T_1 = \{r_1(a), r_1(c), w_1(a), w_1(c)\}$
  - $T_2 = \{r_2(b), w_2(b), r_2(c), w_2(c)\}$

- **Schedules**
  - $S_1 = \{r_1(a), r_1(c), w_1(a), w_1(c), r_2(b), w_2(b), r_2(c), w_2(c)\} = \{T_1, T_2\}$

    → $S_1 \equiv S_2$ (equivalent, because $r_2(b)$, $w_2(b)$ independent of T_1)

  - $S_2 = \{r_1(a), r_2(b), r_1(c), w_1(a), w_2(b), w_1(c), r_2(c), w_2(c)\}$

    → $S_2 \not\equiv S_3$ (non-equivalent, because $w_1(c)$, $r_2(c)$ of c in different order)

  - $S_3 = \{r_1(a), r_2(b), r_1(c), w_1(a), w_2(b), r_2(c), w_1(c), w_2(c)\}$

→ $S_1 \not\equiv S_3$ (transitive)

Locking Schemes

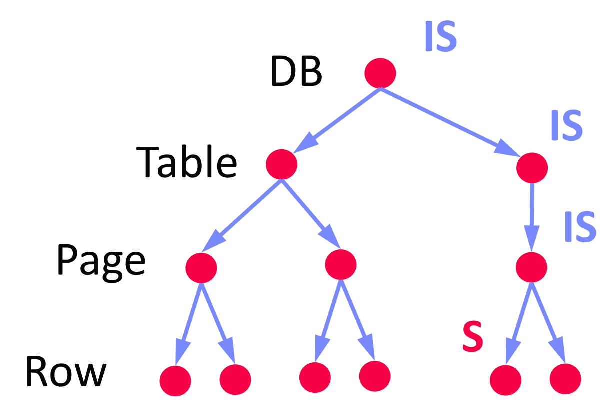
- **Compatibility of Locks**
  - X-Lock (exclusive/write lock)
  - S-Lock (shared/read lock)

Existing Lock

Requested Lock	None	S	X
S	Yes	Yes	No
X	Yes	No	No

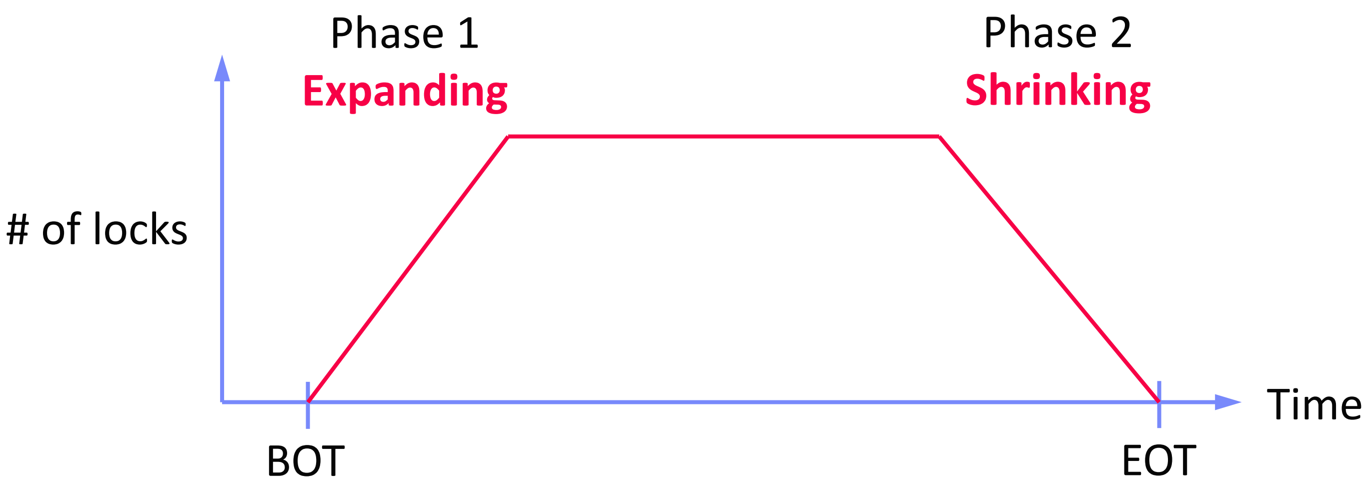
- **Multi-Granularity Locking**
  - Hierarchy of DB objects
  - Additional intentional IX and IS locks

	None	S	X	IS	IX
S	Yes	Yes	No	Yes	No
X	Yes	No	No	No	No
IS	Yes	Yes	No	Yes	Yes
IX	Yes	No	No	Yes	Yes



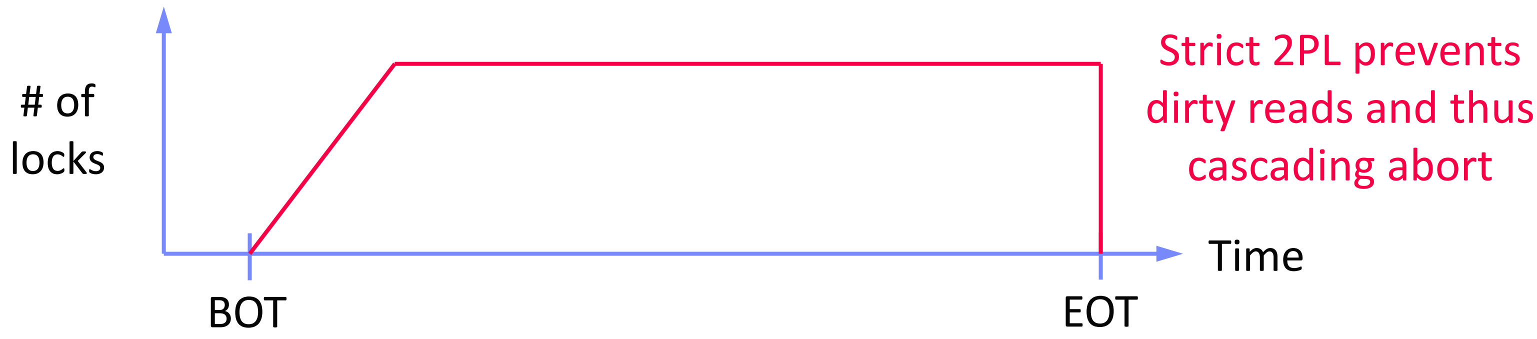
Two-Phase Locking (2PL)

- **Overview**
  - 2PL is a concurrency protocol that guarantees **SERIALIZABLE**
  - **Expanding phase:** acquire locks needed by the TX
  - **Shrinking phase:** release locks acquired by the TX
    (can only start if all needed locks acquired)

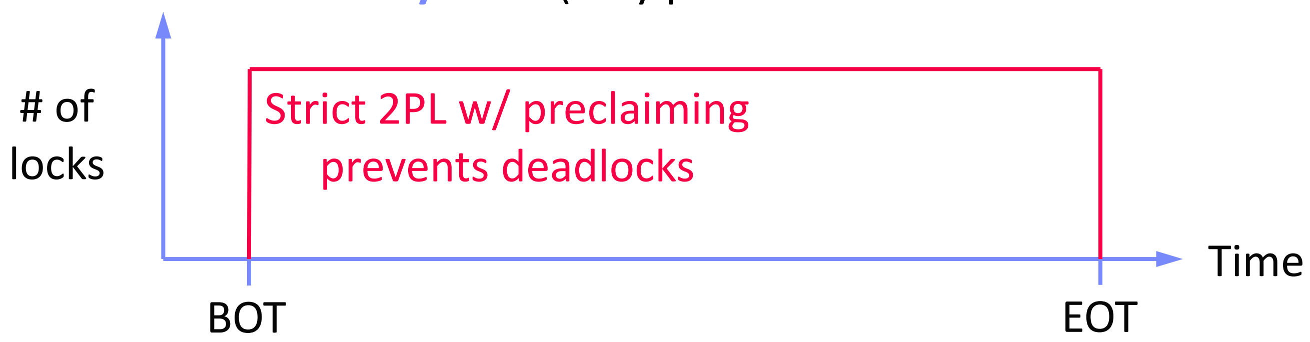


Two-Phase Locking, cont.

- **Strict 2PL (S2PL) and Strong Strict 2PL (SS2PL)**
  - **Problem:** Transaction rollback can cause (**Dirty Read**)
  - Release all X-locks (S2PL) or X/S-locks (SSPL) **at end of transaction (EOT)**

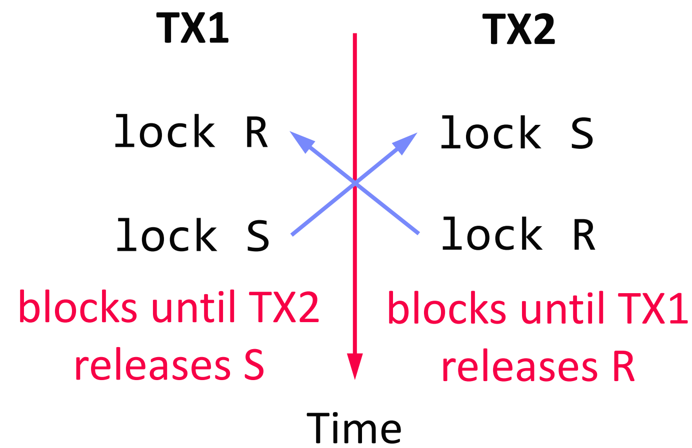


- **Strict 2PL w/ pre-claiming (aka conservative 2PL)**
  - Problem: incremental expanding can cause deadlocks for interleaved TXs
  - **Pre-claim all necessary locks** (only possible if entire TX known + **latches**)



Deadlocks

- **Deadlock Scenario**
  - Deadlocks of concurrent transactions
  - Deadlocks happen due to **cyclic dependencies without pre-claiming** (wait for exclusive locks)



DEADLOCK, as this will never happen

- **#1 Deadlock Prevention**
  - Guarantee that deadlocks can't happen
  - E.g., **via pre-claiming** (but overhead and not always possible)
- **#2 Deadlock Avoidance**
  - Attempts to avoid deadlocks before acquiring locks via timestamps per TX
  - **Wound-wait** (T1 locks something held by T2 → if T1<T2, restart T2)
  - **Wait-die** (T1 locks something held by T2 → if T1>T2, abort T1 but keep TS)
- **#3 Deadlock Detection**
  - Maintain a wait-for graph of blocked TX (similar to serializability graph)
  - Detection of cycles in graph (on timeout) → abort one or many TXs

Timestamp Ordering

Great, low overhead scheme if conflicts are rare (no hot spots)

- **Synchronization Scheme**
  - Transactions get timestamp (or version number) $TS(T_j)$ at BOT
  - Each data object A has **readTS(A)** and **writeTS(A)**
  - Use timestamp comparison to validate access, otherwise abort
  - No locks but latches (physical synchronization)
- **Read Protocol $T_j(A)$**
  - If $TS(T_j)$ >= writeTS(A): allow read, set readTS(A) = max($TS(T_j)$, readTS(A))
  - If $TS(T_j)$ < writeTS(A): **abort T_j** (older than last modifying TX)
- **Write Protocol $T_j(A)$**
  - If $TS(T_j)$ >= readTS(A) AND $TS(T_j)$ >= writeTS(A): allow write, set writeTS(A)=$TS(T_j)$
  - If $TS(T_j)$ < readTS(A): **abort T_j** (older than last reading TX)
  - If $TS(T_j)$ < writeTS(A): **abort T_j** (older than last modifying TX)
- **BEWARE:** Timestamp Ordering requires additional handling of dirty reads, and concurrent transactions in general (e.g., via abort or versions)

Optimistic Concurrency Control (OCC)

- **Read Phase**
  - Initial reads from DB, **repeated reads and writes into TX-local buffer**
  - Maintain **ReadSet(T_j)** and **WriteSet(T_j)** per transaction T_j
  - TX seen as read-only transaction on database
- **Validation Phase**
  - Check read/write and write/write conflicts, **abort on conflicts**
  - BOCC (Backward-oriented concurrency control) – check all older TXs T_i that finished (EOT) while T_j was running ($EOT(T_i) \geq BOT(T_j)$)
    - **Serializable:** if $EOT(T_i) < BOT(T_j)$ or $WSet(T_i) \cap RSet\left(T_j\right) = \emptyset$
    - **Snapshot isolation:** $EOT(T_i) < BOT(T_j)$ or $WSet(T_i) \cap WSet\left(T_j\right) = \emptyset$
  - FOCC (Forward-oriented concurrency control) – check running TXs
- **Write Phase**
  - Successful TXs: propagate TX-local buffer into the database and log
  - Unsuccessful TXs: discard the TX-local buffer

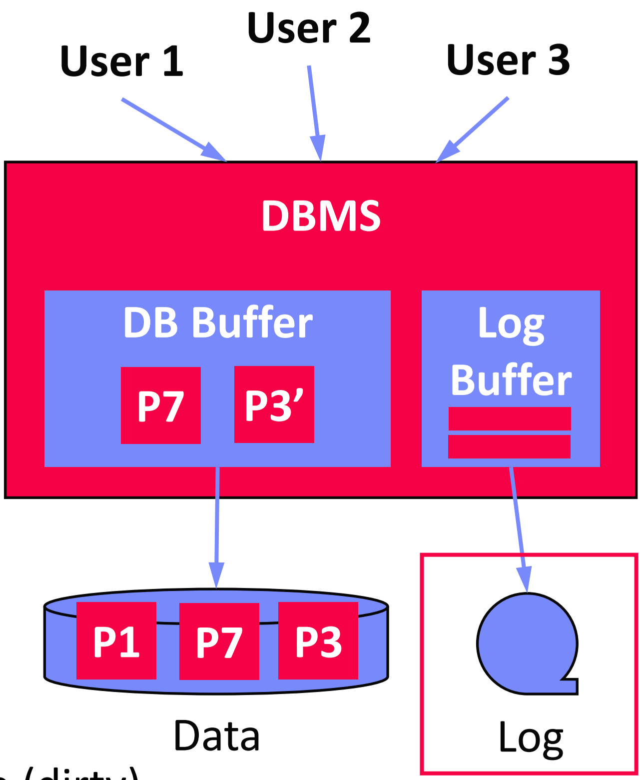
Logging and Recovery

(Atomicity and Durability)

Failure Types and Recovery

- **Transaction Failures**
  - E.g., Violated integrity constraints, abort
  - → **R1-Recovery: partial UNDO** of this uncommitted TX
- **System Failures** (soft crash)
  - E.g., HW or operating system crash, power outage
  - Kills all in-flight transactions, but does not lose persistent data
  - → **R2-Reovery: partial REDO** of all committed TXs
  - → **R3-Recovery: global UNDO** of all uncommitted TXs
- **Media Failures** (hard crash)
  - E.g., disk hard errors (non-restorable)
  - Loses persistent data → need backup data (checkpoint)
  - → **R4-Recovery: global REDO** of all committed TXs

Database (Transaction) Log



- **Database Architecture**
  - Page-oriented storage on disk and in memory (DB buffer)
  - Dedicated **eviction algorithms**
  - Modified in-memory pages marked as dirty, flushed by cleaner thread
  - Log: append-only TX changes
  - Data/log often placed on different devices and periodically archived (backup + truncate)
- **Write-Ahead Logging (WAL)**
  - The log records representing changes to some (dirty) data page must be on **stable storage before the data page** (UNDO - atomicity)
  - **Force-log on commit** or full buffer (REDO - durability)
  - **Recovery:** forward (REDO) and backward (UNDO) processing
  - Log sequence number (LSN)

[C. Mohan, Donald J. Haderle, Bruce G. Lindsay, Hamid Pirahesh, Peter M. Schwarz: ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. **TODS 1992**]

Logging Types and Recovery

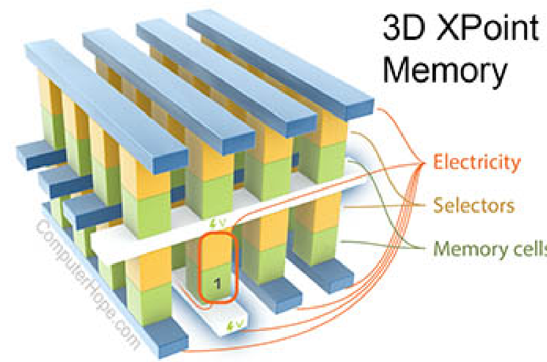
- **#1 Logical (Operation) Logging**
  - REDO: log operation (not data) to construct after state
  - UNDO: inverse operations (e.g., increment/decrement), not stored
  - Non-determinism cannot be handled, more flexibility on locking
- **#2 Physical (Value) Logging**
  - REDO: log REDO (after) image of record or page
  - UNDO: log UNDO (before) image of record or page
  - Larger space overhead (despite page diff) for set-oriented updates

```
UPDATE Emp
  SET Salary=Salary+100
  WHERE Dep='R&D';
```

- **Restart Recovery (ARIES)**
  - Conceptually: take database checkpoint and replay log since checkpoint
  - Operation and value locking; stores log seq. number (LSN, PageID, PrevLSN)
  - Phase 1 Analysis: determine winner and loser transactions
  - Phase 2 Redo: replay all TXs in order [repeating history] → state at crash
  - Phase 3 Undo: replay uncommitted TXs (losers) in reverse order

Excursus: Recovery on Storage Class Memory

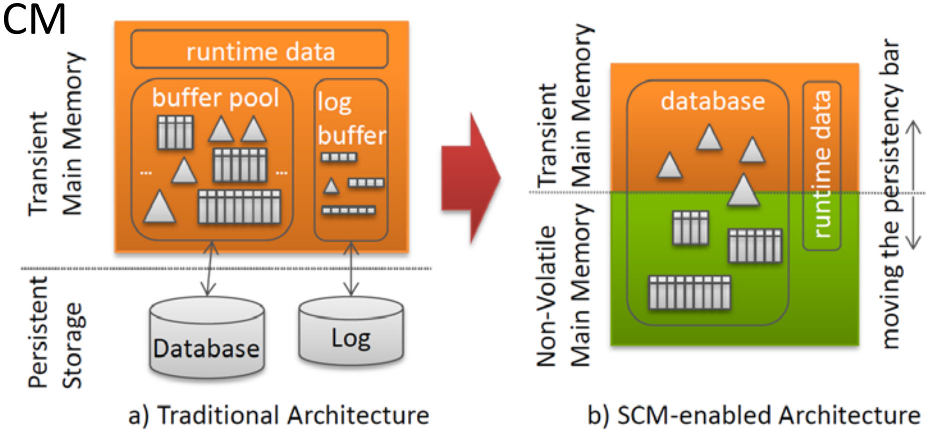
- **Background: Storage Class Memory (SCM)**
  - **Byte-addressable, persistent memory** with higher capacity, but latency close to DRAM
  - **Examples:** Resistive RAM, Magnetic RAM, Phase-Change Memory (e.g., **Intel 3D XPoint**)



[**Credit:** https://computerhope.com]

- **SOFORT: DB Recovery on SCM**
  - Simulated DBMS prototype on SCM
  - Instant recovery by trading TX throughput vs recovery time (**% of data structures on SCM**)

[Ismail Oukid, Wolfgang Lehner, Thomas Kissinger, Thomas Willhalm, Peter Bumbulis: Instant Recovery for Main Memory Databases. **CIDR 2015**]



- **Write-Behind Logging** (for hybrid SCM)
  - Update persistent data (SCM) on commit, log change metadata + timestamps ➔ **1.3x**

[Joy Arulraj, Matthew Perron, Andrew Pavlo: Write-Behind Logging. **PVLDB 2016**]

Exercise 3: Tuning and Transactions

Published: May 09, 2022

Deadline: May 31, 2022

Task 3.1 Query Rewriting and Tuning

5/25 points

- **a) Query Processing** (3 points)
  - **Q09:** Find districts with >1500 inhabitants holding citizenships != 'Austria' as of '2022-01-01'. (return district name, country name, population count).
  - `T1/Q09.sql` and EXPLAIN plan output `T1/Q09.json`

Expected Result in Slide Notes

- **b) Indexing** (3 points)
  - Create one or many indexes on attributes of your choosing in order to reduce costs of Q09
  - `Index.sql` and EXPLAIN plan output `T1/Q09WithIndex.json`

```
ANALYZE Participant, Locale;

EXPLAIN (FORMAT JSON) SELECT *
  FROM Participant AS R, Locale AS S
  WHERE R.LID=S.LID;
```

```
"[{"Plan": {
 "Node Type": "Hash Join", ...
 "Startup Cost": 1.25,
 "Total Cost": 2.47
 "Plans": [ {...
  "Node Type": "Seq Scan",
  "Relation Name": "participant",
  },{
  "Node Type": "Hash",
  "Plans": [{...
    "Node Type": "Seq Scan",
    Relation Name": "locale",
}]}]}}]"
```

Task 3.2 B-Tree Insertion and Deletion

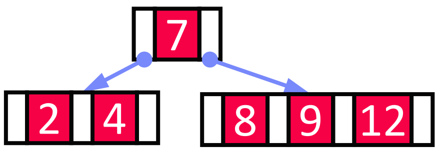
6/25 points

- **Setup**
  - Generate a sequence S of numbers as follows and materialize as `Input.txt`

```
SELECT SETSEED(1.0/(SELECT MOD(X,8)+1));
SELECT * FROM generate_series(1,16) ORDER BY random();
```

- **B-Tree Insertion (k=2)**
  - Draw a B-tree after sequentially inserting S in the defined order

- **B-Tree Deletion**
  - Draw the B-tree again after deleting the sequence [8,14) in order of keys (del 8, del 9, …, del 13)




```
a: (b) 7 (c)
b: 2 () 4
c: 8 () 9 () 12
```

- **Note: Text Format**

```
node_id: (child_node_id 1) key (child_node_id 2) ... (child_node_id n)
```

Task 3.3 Transaction Processing

6/25 points

- **a) Setup** (2 points)
  - Create the following tables w/ meaningful data types → `TXSetup.sql`

    ```
    Vendors(VID, Name, Profit)
    Products(PID, Name, Price, Stock)
    Sales(SDate, VID, PID, Quantity)
    ```

  - Insert the following tuples

    ```
    (9,’V1’, 0.0) into Customer
    (1, ’P1’, 25, 100) into Products
    ```

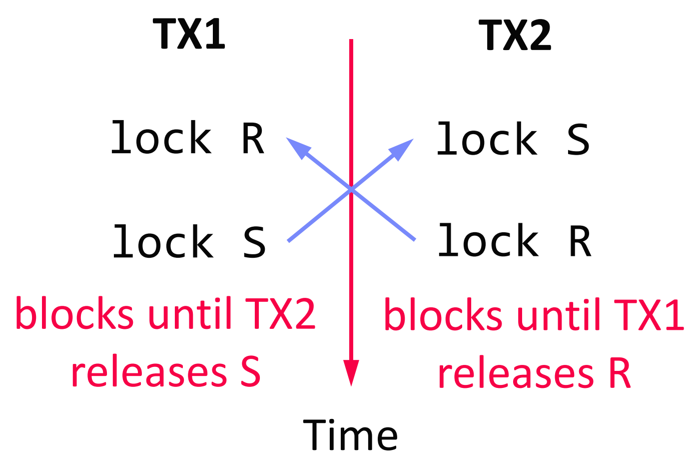
- **b) Atomic Transaction** (2 points)
  - Write a SQL transaction a placing a new order → `TXNewOrder.sql`
    - Add order for 10 x product P1, by vendor V1 as of 2022-04-07
    - Modify product stock / vendor profit accordingly

Task 3.3 Transaction Processing, cont.

6/25 points

- **c) Deadlock** (2 points)
  - Simulate a Deadlock on the populated tables via two TXs and explain how the operations should be interleaved

  → `Deadlock.sql`



Task 3.4 Iterator Model and Operators

8/25 points

- **a) Operator Implementations** (**Extra Credit** for ICE course 'Databases')
  - Pick your favorite programming language (e.g., Python, Java, C# or C++)
  - open(), next(), close() iterator model (base class)
  - Implement **table scan**, **cmp selection** (ge/eq), and **hash join**
- **b) Implement Query Q10** (on new provided data)

```
SELECT *
  FROM Districts D,
       PopByGender PG
  WHERE D.DKey = PG.DKey
    AND D.Population >= 30000
    AND PG.PopDate = '2022-01-01'
```

[https://mboehm7.github.io/teaching/ss22_dbs/DataExport.zip]

```
./runQuery10.sh ./Districts.csv \
  ./PopByGender.csv ./out.csv
```

```
D = new TblScan(args[1]) //Districts
PG = new TblScan(args[2]) //PopGender
D2 = new CmpSelect(D, attr=3,
       type='ge', value=30000))
PG2 = new CmpSelect(PG, attr=3,
       type='eq', value='2022-01-01'))
q = new HashJoin(D2, PG2, 1, 1)

q.open()
while( (t = q.next()) != EOF )
  out.write(t)
q.close()
```

Expected Result in Slide Notes

Summary and Q&A

- **Summary 09 Transaction Processing**
  - Overview transaction processing
  - Locking and concurrency control
  - Logging and recovery

- **Summary Part A: Database Systems**
  - Databases systems primarily from user perspective
  - End of lectures for Databases (but +1 ECTS if you attend entire course)

- **Next Lectures (Part B: Modern Data Management)**
  - **10 NoSQL (key-value, document, graph)** [May 23]
  - **11 Distributed Storage and Data Analysis** [May 30]
  - **Jun 06:** Whit Monday (Pfingstmontag)
  - **12 Data Stream Processing Systems and Q&A** [Jun 13, Patrick]