1. Ponturi db sem 3 - ponturi db
2. Ce multi sunteti aici sa moara bibi
3. **ACID**
    a. Atomicity: transactions are atomic (all or nothing)
    b. Consistency: the consistency of the DB is preserved after executing a trans, all the constraints are still satisfied (<=> a trans is a correct program) the DB might be in a wrong state at some point during the trans, but it only has to be correct at the end
    c. Isolation: a trans is protected from the effects of scheduling other trans concurrently
    d. Durability: the changes of a successful trans will persist even if there's a crash before writing all changes to disk
        i. Write-ahead log: changes are written to a log on the disk before being reflected in the DB
4. If 2 trans are just reading an object => no conflicts (exec order is not important)
5. If 2 trans are r/w different objects => no conflicts (...)
6. 2 trans r/w on the same object => possible conflicts (order is important)
    a. WR (**dirty reads**, reading uncommitted data) - T2 reads sth written by T1
    b. RW (**unrepeatable reads** - T2 writes sth already read by T1
    c. WW (overwriting uncommitted data, *blind write*) - T2 writes sth already written by T1
7. **Serial schedules** - transactions are not interleaved
8. Serializable schedules iff their effect is identical to the effect of a serial schedule
9. Schedules are conflict equivalent (S1 ≡c S2, the order matters) iff conflict(S1) = conflict(S2) ⇔ S1, S2 have the same op on the same objects + every pair of conflicting ops is ordered in the same way in S1 and S2
10. S is conflict serializable iff its precedence graph is acyclic
11. conflict serializable sch ≡c serializable sch
12. Conflict relation = set of pairs (op1, op2); op1 preceds op2 and in conflict
13. **Precedence graph**: one node for each commited trans in S + one arc Ti -> Tj if an op in Ti preceds and conflicts with an op in Tj
14. S1, S2 are **view equivalent** (S1 ≡v S2) iff each transaction performs the same computation in S1 and S2 and S1, S2 produce the same final DB state

15. Serial schedules ⊂ conflict serializable sch ⊂ view serializable sch ⊂ serializable sch ⊂ all schedules
16. <u>Recoverable</u> sch - T commits only after all the transactions whose changes T reads have committed
    a. Avoid cascading aborts => recoverable sch
17. **Lock** - prevents a trans from accessing an object while another trans is accessing that object
18. Locks + transaction protocols => allow interleaved executions
19. Basically we can have multiple readers but just 1 writer at a time

Can you get a -> lock if there is a ↓ lock	Shared lock	Exclusive lock
Shared lock	yes	no
Exclusive lock	no	no

20. Lock table - for each object, entries of the form (no. of trans with a lock on that obj, lock type, pointer to queue of lock requests)
21. Transactions table - for each trans, the list of locks held by it
22. Strict 2PL - acquires s/x lock before r/w; locks released when trans ends => <u>only serializable sch</u>
23. (Normal) 2PL - acquires…; once releasing a lock, it cannot request another one
24. Strict schedules - recoverable, no cascading aborts, if a trans is aborted, its ops can be undone;
    a. Strict 2PL only allows strict sch
25. **Deadlock** - 2 trans wait for each other to free a resource
    a. Usually the oldest trans has priority
    b. Prevention policies => deadlocks cannot occur
        i. **Wait-Die**: if T1 has higher priority, it waits, else is aborted
        ii. **Wound-Wait**: if T1 has higher priority, T2 is aborted, else it waits
    c. Waits-for graph: 1 node / active trans; arc T1->T2 if T1 is waiting for T2 to release a lock; cycles => deadlocks; periodically checked by the DBMS
    d. Timeout mechanism - if a trans waits too long for a lock to be released, a deadlock is assumed to exist => trans is terminated
26. Dirty writes are not allowed under any iso levels; X locks are <u>always</u> acquired for reading and <u>always</u> released at the end of the trans
27. **Read uncommitted**
    a. A trans can read uncommitted data (dirty reads)
    b. No S locks when reading

    c. Dirty reads, nonrepeatable reads, phantoms
28. **Read committed (default)**
    a. Trans can only read committed data, but the read data can still be modified by another trans
    b. S locks when reading, released asap
    c. Nonrepeatable reads, phantoms
29. **Repeatable read**
    a. Trans can only read committed data, the read data cannot be modified by another trans
    b. S locks when reading, released at the end of the trans
    c. Just phantoms
30. **Serializable**
    a. RR + if the trans reads a set of objects based on a search predicate, that set cannot be modified by another trans (key-range locks)
    b. Locks released at the end
    c. No phantoms
31. **Snapshot - ?**
32. **Recovery manager** - ensures atomicity (uncommitted trans are undone) and durability (committed trans survive crashes)
33. Trans failure causes: system failure, app error, action by the transaction manager (ex. because of deadlock resolution), self-abort
34. **Steal** = the changes made by a trans <u>can</u> be written to disk before the commit (the BM wants to remove the frame of t1 to make space for a frame for t2 => t2 "steals" a frame from t1)
35. **No-steal** = changes <u>cannot</u> be written... => we don't have to undo changes in case of failure - but the BM needs to be able to hold all the changed pages, not always the case
36. **Force** = the changes are written to disk right when commiting => no need to redo trans after crashes, because they have been written to disk - but if a page is commited many times by some transactions, it's written to disk that many times; no-force writes it only once
37. **No-force** = the changes might remain in the BP for a short while
38. Steal + no force -> most used
39. **ARIES** - steal + no force; after a crash, it does:
    a. Analysis - determine active trans + dirty pages
    b. Redo - reapply all changes, even those of aborted trans (bring the db to the state from when the crash occurred), in order
    c. Undo - undo uncommitted trans, in reverse order

    d. Write-Ahead Logging - changes are first recorded in a log; the log is written to stable storage before the change is written to disk

40. **Log** (journal)
    a. Log tail = most recent fragment; periodically forced to stable storage
    b. LSN = log sequence number = id of a log record
    c. Page LSN = the LSN of the most recent record describing a change to P
    d. Log record = prev LSN, transID, type (update, commit etc)
    e. A record is written for: update page, commit, abort, end, undo update (=> compensation log record CLR)
41. Transaction table - 1 entry / active trans
    a. Trans ID, status (**in progress, committed, aborted**), lastLSN
42. Dirty page table - 1 entry / dirty page in BP
    a. pageID, recLSN (LSN of log that first dirtied the page)
43. Checkpointing in ARIES
    a. Write a begin_checkpoint record
    b. Write an end_checkpoint record
    c. After the end_checkpoint is written to stable storage, write a master record to stable storage
44. Crash -> restart -> restore from the most recent checkpoint
45. **Analysis**
    a. dirty page table gets a copy of the DirtyPageTable from the end checkpoint (most recent); same for trans table (TT)
    b. end log record => remove the trans from TT
    c. any other log record => add T to the TT if it's not there already, set its last LSN as the LSN of the crt log record;
    d. if it's a commit => set T's status (in the TT) to C, otherwise U
    e. if the page is not in DPT, add it; recLSN = crt LSN
    f. otherwise just update recLSN
46. **Redo**
    a. reapply the updates and CLRs, unless one of these conditions happen
        i. the page (P) is not in DPT (usually their changes have already been written to disk)
        ii. P in DPT, but P.recLSN > crtLSN
        iii. P.pageLSN (from the actual DB on the disk) >= crtLSN
    b. the actual redo: reapply the changes; P.pageLSN = crtLSN
47. **Undo**
    a. loser transaction = active at the end of the crash

    b. toUndo = set of last LSNs of losers; empty => the undos have been completed
    c. if L is a CLR, check undoNextLSN;
        i. if not null => add undoNextLSN to toUndo
        ii. else => do the redo?
    d. if L is an update - write CLR, undo action, add L.prevLSN to toUndo

48. Security = protecting data against unauthorized users; integrity = .. authorized users
49. Access request = requested obj + requested operation + requesting user
50. Discretionary control - allowed operations are explicitly specified, anything else if forbidden
51. Mandatory control - each object has a classification level, each user has a clearence level
    a. Bell and La Padula rules - x can retrieve y if clearance(x) >= classification(y); x can update y if clearance(x) == classification(y)
52. Unless specified otherwise, every command is a transaction
53. There exists an update lock (U) - deadlock avoidance mechanism
54. Key-range locking - lock existing data and data which doesn't exist (yet) based on a search predicate
55. SQL server uses deadlock detection; + the trans that is the least expensive to rollback is terminated
56. Algorithms for operators, based on
    a. Iteration
    b. Indexing
    c. Partition
        i. Sorting
        ii. Hashing
57. **Access path** = way of retrieving tuples from a relation (file scan or index + condition)
    a. Condition C matches index I if I can be used to retrieve just the tuples satisfying C
    b. Hash index - used for equality selections; C has to have one term for each sk of I (C with <a, b> is not matched by I with <a, b, c>)
    c. Tree-based index - used for all (<, >, ..), including equality; terms of C have to be a prefix of those of I
    d. Most selective access path - retrieves the fewest pages (data retrieval cost is minimized)
    e. General selection - the condition has to be in CNF; L7, P16

58. Joins

a. E (outer rel) - M pages; pe records/ page; S = N pages; ps records/ page

b. Iteration

  i. **Simple nested loops join**; cost = M + pe * M * N I/Os

```
foreach tuple e ∈ E do
    foreach tuple s ∈ S do
        if e_i == s_j then add <e, s> to the result
```

  ii. **Page-Oriented nested loops join**; cost = M + M * N

```
foreach page pe ∈ E do
    foreach page ps ∈ S do
        if e_i == s_j then add <e, s> to the result
```

  iii. **Block nested loops join -** uses the buffer pages more effectively

```
foreach block be ∈ E do
   foreach page ps ∈ S do
     {
        for all pairs of tuples <e, s> that meet the join
          condition, where e ∈ be and s ∈ ps,
               add <e, s> to the result
```

    1. B pages in the buffer pool => B - 2 pages in a block (input / output each have 1 page)
    2. Number of blocks = M / (B - 2) (rounded up)
    3. Cost = M + no. blocks * N

c. Indexing

  i. Index nested loops join

```
foreach tuple e in E do
    foreach tuple s in S where e_i == s_j
        add <e, s> to the result
```

    1. Cost of examining the index on S = 2-4 for B+ tree index, 1.2 for hash index;
    2. 1 extra I/O for reading a record in S, if clustered index; at most 1 if nonclustered
    3. Cost: M + M * pe * (examine cost + read cost)

d. Partitioning

  i. Sort-merge join

1. Sort each relation (ex. with external merge sort)
2. Merge, with cost = M + N if the inner partition is scanned only once; M * N worst case
3. Cost = sorting costs + merge cost

ii. Hash join
1. Partitioning phase: 1 buffer page = input, B-1 pages = output; the hash function distributes tuples to the B-1 pages
2. **Partition** = collection of tuples with the same hash value, can be stored on 1+ pages when on disk
3. Probing phase: for each partition of the smaller relation, scan the other one for matching tuples; 1 page for input, 1 for output, B - 2 to read the partitions of the smaller relation
4. If a partition does not fit in memory => partition overflow => apply hash join recursively
5. Cost = 3 * (M + N)

59. If the data fits in main memory => internal sorting (ex. quicksort), else => external sorting

60. **Simple 2-way merge sor**t

$$2 * N * (\lceil log_2 N \rceil + 1) \text{ I/Os}$$

a. Pass 0 -> read each page, sort it, save it back to disk => 1-page runs
b. Pass 1+ -> use just 3 buffer pages, read + merge previous runs => runs twice as long
c. read + write each page at each pass (=> that first 2 I/O)

61. **External merge sort**: N = pages in input file; B = buffer pages => cost =

$$2 * N * \left(\left\lceil log_{B-1} \left\lceil \frac{N}{B} \right\rceil \right\rceil + 1 \right) \text{ I/Os}$$

a. Pass 0 -> read B pages at once, sort, write to disk

62. RLV (row-level versioning) - useful when we need committed data, but not necessarily the most recent version
a. **XSN** = transaction sequence number; each row version is marked with ^ of the trans that changed the row

b. The versions are kept in a linked list
c. **Read committed snapshot isolation** - operations see the most recent committed data as of the **beginning of their execution**; consistent reads at the command level
d. **Full snapshot isolation** - …**beginning of their transaction**; consistent.. transaction level
e. Increased concurrency, but update operations are slower and we also ‘use’ a *tempdb*; also, there is still the issue of simultaneous writers (update conflict)

63. Projection, based on
    a. **Sorting** - select only the required columns, sort this => a temporary table, from which we remove adjacent duplicate tuples
        i. Cost: M + T (T is O(M)) + sort cost + T
        ii. Last T only if we remove duplicates
        iii. Sort cost: see external merge sort / 2way merge, but it’s O(M log M)
    b. Hashing
        i. Partitioning phase - 1 input page, B - 1 output => hash into B - 1 partitions, no unwanted fields
        ii. Duplicate elimination phase - one partition at a time; build a new hash table for each to detect duplicates
        iii. Cost: M + 2 * T (T = no. of partitions)
64. Intersection = join with condition of equality on all fields
65. Cross-product = join with no condition
66. Union, based on
    a. Sorting - sort the relations + merge them, eliminating duplicates
    b. Hashing - partition the relations with a hash function; build another hash table for each partition for one of the relations; scan the other relation, while comparing with the values in the second hash table
67. Aggregate operations
    a. Without group by - maintain some *running info* about tuples (ex. current max, sum etc); cost = scan cost
    b. With group by - sort the relation on the grouping attr, then scan the relation to get the aggregates for each group
68. **Optimizer** - generates plans for a query, tries to select the best one (but that’s not always possible); optimizes one block at a time
    a. Convention - the outer relation is the left left child of the join op (in an algebra tree)

69. Statistics maintained by the DBMS - updated periodically (R = relation, I = index)
    a. <u>NTuples</u>(R), <u>NPages</u>(R)
    b. <u>NKeys</u>(I) (distinct key values), <u>INPages</u>(I) (page count or, if b+ tree index -> leaf pages); <u>IHeight</u>(I) (just for tree indexes); <u>ILow</u>(I), <u>IHigh</u>(I) (min/ max key value)
70. Estimating result sizes
    a. **Reduction factor (RF)** - estimation of how much each term in a WHERE eliminates candidate rows
    b. The estimation = product of the relation cardinalities * product of RFs
    c. Condition: column = value
        i. Index on column => RF = 1 / NKeys(I)
        ii. No index => RF = 1 / 10
    d. Condition: column1 = column2
        i. One index on each => RF = 1 / MAX(nkeys(i1), nkeys(i2))
        ii. Just one index (on either column) => RF = 1 / nkeys(i)
        iii. No index => RF = 1 / 10
    e. Condition: column > value
        i. Index => RF = (ihigh(i) - value) / (ihigh - ilow)
        ii. No index => a random val < 0.5
    f. Condition: column IN (value list) => RF = (RF for column = value) * list length
    g. Condition: NOT => RF = 1 - RF(condition)
71. <u>Enumeration of alternative plans</u>
    a. Query with 1 relation in FROM
        i. No joins / cross-products
    b. … several relations in FROM
72. DMV = dynamic management view
73. Query fine-tuning:
    a. Identify waits - sys.dm_os_wait_stats
    b. Correlate waits with queues - sys.dm_os_performance_counters
    c. Drill down to DB / file level - sys.dm_io_virtual_file_stats
    d. Drill down to process level
74. **Distributed DB**
    a. Distributed data independence - when writing a query, the user doesn’t know / care that the data is distributed
    b. Distributed transaction atomicity - a trans that operates on different sites is still atomic; again, the user doesn’t know / care that the data is distributed

c. **Homogeneous** - same DBMS at every site; **heterogeneous** - different DBMSs
d. <u>Fragmentation</u> - split the data into fragments, stored across several sites
    i. Horizontal - basically selection => disjoint subsets of rows; reconstruct -> union
    ii. Vertical - basically projection => disjoint subsets of columns; reconstruct -> natural join
    iii. Hybrid - combination of both ^
e. <u>Replication</u> - same fragment in multiple sites => increased data availability (in case one location fails, we still have others) + faster queries (no communication costs)
    i. **Synchronous** = all data copies are up to date before the trans commits
        1. **Voting** - when writing, a trans modifies a majority of its copies; when reading, a trans has to read at least (n - the value of the majority)
            a. Ex: n = 10; T1 modifies 7 copies => T2 has to read 4, to guarantee that at least 1 was modified by T1
        2. **Read-any write-all** - when writing, a trans modifies all the copies; when reading, it has to read just 1
            a. Better because reading is more frequent than writing, and when voting, reading is quite expensive
            b. However, this method will require x locks from all the other sites, which could be slow
    ii. **Asynchronous** = synchronized periodically => time periods when some copies are outdated -> the user will have to take data distribution into account
        1. **Peer to peer**
            a. A hierarchy of copies, only master copies can be changed => all their successors are changed
            b. preferred when there are no conflicts / data copies are disjoint
        2. **Primary site** - just one master copy, with secondary copies at other sites; change propagation in 2 steps:
            a. Capture - ..all the changes made by a trans to the master copy

            i. Log based => a ‘change data table’ - which will contain only the update log records of committed trans
            ii. Procedural - a procedure is automatically invoked (ex. trigger); uses snapshots of the primary copy
        b. Apply - propagate changes to secondary copies; either the primary site continually sends the data (CDT / snapshot) to the secondaries, or the secondaries periodically request it
        c. Log + continuous apply => minimum delay
        d. Procedure + request => most flexible

75. Distributed query processing
    a. There is a <u>*shipping cost*</u> - we transfer pages between sites
    b. Nonjoin queries
    c. Join queries
        i. Fetch as needed - for each tuple in R, retrieve the corresponding tuples in A
        ii. Ship to one site - ship the entire R to a location, do the join there
        iii. Semijoin
            1. project R onto the join columns and send the projection to a location
            2. we do the join there (“reduction of A with respect to R”)
            3. ship the reduction of A back to the first location, where we join R with the reduction of A
            4. Useful if there is a selection on one of the relations
        iv. Bloomjoin
            1. Hash the tuples in R; use a bit vector to store the hash results, send the bit vector to another location
            2. Same hash but for A, discard the tuples whose hash value don’t exist in the other bit vector => *“reduction of A with respect to R”*, send the reduction to the first location
            3. Join R with the reduction of A

76. Distributed catalog management
    a. Centralized system catalog - one site contains all the info; vulnerable to **single-site failures(SSF)**

b. Global system catalog at each site - every site has all the info; no longer vulnerable to ssf^, but local autonomy is compromised (local changes have to be propagated to all other sites)
c. Local catalog at each site - each site has only local data; a catalog keeps track of all fragments / replicas of its relations - if we move / create a new replica => have to update the original catalog

77. Distributed transaction management
    a. Distributed concurrency control
        i. **Lock management**
            1. Centralized - one site does all the locking, vulnerable to SSF
            2. Primary copy
                a. each obj has a primary copy stored at some site with some lock manager
                b. lock requests of all copies of the obj are handled by that lock manager
                c. not vulnerable to SSF
                d. when reading another copy of the same obj, stored at a different site, we need to communicate with both sites
            3. Fully distributed - each obj has a copy…; but here the lock req are handled by the lock manager of the site where the copy is stored => no longer need to communicate with 2 sites
        ii. **Distributed deadlocks**
            1. Each site has a local waits-for graph, but there could exist global deadlocks even if there are no cycles in the local graphs
            2. <u>Centralized detection algorithm</u> - all the local graphs are periodically sent to a single site, where they are united => global graph
            3. <u>Hierarchical</u> .. - there is a hierarchy of sites; each site sends the graph to its parent
            4. <u>Based on timeout</u> - trans is aborted if it waits too long; could be the only option in heterogeneous systems (different dbms => the graphs might not be compatible)

        5. Phantom deadlock - a trans aborts before the global graph is updated => global thinks there is a deadlock and might abort another transaction

78. Distributed recovery
    a. (for a trans T) coordinator = transaction manager at the site where T originated; subordinates = .. sites where T's subtransactions execute
    b. **Two-phase commit protocol** (2PC)
        i. 2 rounds of messages, both init by the coord (voting / termination)
        ii. The user decides to commit T => commit command sent to T's coordinator => start 2PC
            1. Coord sends a "prepare" msg to each subordinate
            2. The subordinate decides whether to commit / abort and responds "yes" / "no"
            3. Coord receives "yes" => sends "commit" msg to all subord; ...at least a "no" or no msg at all from a subord => coord sends "abort"...
            4. Subord receives "abort" => writes an abort log record, responds with an "ack" msg, aborts the trans; … receives "commit" => commit log, "ack" msg, commit trans
            5. Coord receives all "ack" msg => write end log
        iii. Msg are sent only after the log has been forced to stable storage
    c. Restart after failure
        i. If there is a commit / abort log => must redo / undo T; coord periodically sends commit / abort msg to subord until it receives all "ack" responses, then write and end log
        ii. If there is a prepare log but no commit / abort => subord contact their coord until they get the trans status
        iii. If there are no commit / abort / prepare logs => abort T, undo T
    d. Link and remote site failures: remote site R doesn't respond
        i. If the crt site S is coord => S should abort T
        ii. If S is a subord and hasn't voted yet(?) => S should abort T
        iii. .. subord and voted "yes" => S blocked until T's coord responds