TRANSACTION MANAGEMENT

RELATIONAL DATABASES

TRANSACTIONS

- TRANSACTIONS ARE A FUNDAMENTAL CONCEPT OF ALL DATABASE SYSTEMS
- TRANSACTIONS ALLOW USERS TO MAKE CHANGES TO DATA AND THEN DECIDE WHETHER TO SAVE OR DISCARD THE WORK
- DATABASE TRANSACTIONS BUNDLE MULTIPLE STEPS INTO ONE LOGICAL UNIT OF WORK

TRANSACTIONS

- A TRANSACTION CONSISTS OF ONE OF THE FOLLOWING:
    - DML STATEMENT WHICH CONSTITUTE ONE CONSISTENT CHANGE TO THE DATA
    - INSERT, UPDATE, DELETE AND MERGE
    - ONE DDL STATEMENT SUCH AS CREATE, ALTER, DROP, RENAME, OR TRUNCATE
    - ONE DCL STATEMENT SUCH AS GRANT OR REVOKE

TRANSACTIONS

- A BANK DATABASE CONTAINS BALANCES FOR VARIOUS CUSTOMER ACCOUNTS, AS WELL AS DEPOSIT BALANCES FOR OTHER BRANCHES
- A CUSTOMER WANTS TO WITHDRAW AND TRANSFER MONEY FROM HIS ACCOUNT AND DEPOSIT IT INTO ANOTHER CUSTOMER'S ACCOUNT AT A DIFFERENT BRANCH
- SEVERAL STEPS INVOLVED, BOTH BRANCHES WANT TO ENSURE ALL STEPS HAPPEN, OR NONE OF THEM HAPPEN AND IF THE SYSTEM CRASHES THE TRANSACTION IS NOT PARTIALLY COMPLETE.
- GROUPING THE WITHDRAWAL AND DEPOSIT STEPS INTO ONE TRANSACTION PROVIDES THIS GUARANTEE

TRANSACTIONS

- TRANSACTIONS ARE CONTROLLED USING THE FOLLOWING STATEMENTS:
  - COMMIT: REPRESENTS THE POINT IN TIME WHERE THE USER HAS MADE ALL THE CHANGES HE/SHE WANTS TO HAVE LOGICALLY GROUPED TOGETHER, AND BECAUSE NO MISTAKES HAVE BEEN MADE, THE USER IS READY TO SAVE THE WORK. WHEN A COMMIT STATEMENT IS ISSUED THE CURRENT TRANSACTION ENDS, MAKING ALL PENDING CHANGES PERMANENT.
  - ROLLBACK: ENABLES THE USER TO DISCARD ALL PENDING CHANGES MADE TO THE DATABASE
  - SAVEPOINT: CREATES A MARKER IN A TRANSACTION, WHICH DIVIDES IT INTO SMALLER PIECES.
  - ROLLBACK TO SAVEPOINT: ALLOWS THE USER TO ROLLBACK TO A SPECIFIED SAVEPOINT, THUS DISCARDING ONLY THOSE CHANGES MADE AFTER THE SAVEPOINT

TRANSACTIONS

- A TRANSACTION BEGINS WITH THE FIRST DML STATEMENT
  - IT ENDS WHEN ONE OF THE FOLLOWING OCCURS:
    - A COMMIT OR ROLLBACK STATEMENT IS ISSUED
    - A DDL STATEMENT IS ISSUED
    - A DCL STATEMENT IS ISSUED
    - A USER EXITS NORMALLY FROM THE DATABASE UTILITY, CAUSING THE CURRENT TRANSACTION TO BE IMPLICITLY COMMITTED.
- A DDL STATEMENT OR DCL STATEMENT IS AUTOMATICALLY COMMITTED THEREFORE IMPLICITLY ENDS A TRANSACTION
- EVERY DATA CHANGE MADE DURING A TRANSACTION IS TEMPORARY UNTIL THE TRANSACTION IS COMMITTED.

READ CONSISTENCY

- READ CONSISTENCY GUARANTEES A CONSISTENT VIEW OF THE DATA BY ALL USERS AT ALL TIMES
- READERS DO NOT VIEW DATA THAT IS IN THE PROCESS OF BEING CHANGED
- WRITERS ARE ENSURED THAT THE CHANGES TO THE DATABASE ARE DONE IN A CONSISTENT WAY
- CHANGES MADE BY ONE WRITER DO NOT DESTROY OR CONFLICT WITH CHANGES ANOTHER WRITER IS MAKING

CONCURRENCY

- READ CONSISTENCY IS AN AUTOMATIC IMPLEMENTATION
- THIS IS TO ENSURE THAT IN A MULTI USER ENVIRONMENT SUCH AS RELATIONAL DATABASE THAT TRANSACTIONS THAT ARE RUNNING AT THE SAME TIME WORKING ON THE SAME PIECES OF DATA DO NOT INTERFERE WITH EACHOTHER.

CONCURRENCY

- A PARTIAL COPY OF THE DATABASE IS KEPT IN UNDO SEGMENTS.
- WHEN USER A ISSUES AN INSERT, UPDATE OR DELETE OPERATION TO THE DATABASE, THE ORACLE SERVER TAKES A SNAPSHOT (COPY) OF THE DATA BEFORE IT IS CHANGED AND WRITES IT TO THE UNDO (ROLLBACK) SEGMENT.
- USER B STILL SEES THE DATABASE AS IT EXISTED BEFORE THE CHANGES STARTED; HE VIEWS THE UNDO SEGMENT'S SNAPSHOT OF THE DATA.
- BEFORE CHANGES ARE COMMITTED TO THE DATABASE, ONLY THE USER MAKING THE CHANGE SEES THEM, EVERYONE ELSE SEES THE SNAPSHOT.
- THIS GUARANTEES THAT READERS OF THE DATA SEE CONSISTENT DATA THAT IS NOT CURRENTLY UNDERGOING CHANGE.

CONCURRENCY

- WHEN A DML IS COMMITTED, THE CHANGE MADE TO THE DATABASE BECOMES VISIBLE TO ANYONE EXECUTING A SELECT STATEMENT.
- IF THE TRANSACTION IS ROLLED BACK, THE CHANGES ARE UNDONE:
  - THE ORIGINAL, OLDER VERSION OF DATA IN THE UNDO SEGMENT IS WRITTEN BACK TO THE TABLE
  - ALL USERS SEE THE DATABASE AS IT EXISTED BEFORE THE TRANSACTION BEGAN.

CONCURRENCY PROBLEMS

- LOST UPDATE PROBLEM:
  - WHEN TWO TRANSACTIONS ARE UPDATING THE SAME DATA ELEMENT AND ONE OF THE UPDATES IS LOST (OVERWRITTEN BY THE OTHER TRANSACTION)
  - IF TRANSACTION A READS THE DATA VALUE BEFORE TRANSACTION B COMMITS THEN THE CHANGE TRANSACTION B MAKES IS LOST

Time	Transaction	Step	Stored value
1	T1	Read prod_qty	35
2	T2	Read prod_qty	35
3	T1	Prod_qty=35+100	135
4	T2	Prod_qty=35-30	5
5	T1	Write(prod_qty)	135 (lost update
6	T2	Write(prod_qty)	5

CONCURRENCY PROBLEMS

- UNCOMMITTED DATA:
  - THIS OCCURS WHEN TWO TRANSACTIONS ARE EXECUTED AT THE SAME TIME AND ONE ROLLS BACK AFTER THE OTHER TRANSACTION ALREADY ACCESSED THE UNCOMMITTED DATA.

Time	Transaction	Step	Stored value
1	T1	Read prod_qty	35
2	T1	Prod_qty=35+100	
3	T1	Write(prod_qty)	135
4	T2	Read prod_qty	135(read uncommitted data)
5	T2	Prod_qty=135-30	105
6	T1	Rollback	35
7	T2	Write(prod_qty)	105

CONCURRENCY PROBLEMS

- INCONSISTENT RETRIEVALS:
  - THIS OCCURS WHEN A TRANSACTION ACCESSES DATA BEFORE AND AFTER ANOTHER TRANSACTION(S) FINISHES WORKING WITH DATA.
  - T1 IS KEEPING A RUNNING TOTAL OF PRODUCT QUANTITIES
  - T2 IS ADDING TO THE QUANTITIES OF PRODUCTS

Time	Transaction	Step	Stored value	Sum
1	T1	Read prod_qty for id='73FG'	35	35
2	T1	Read prod_qty for id='23TY'	55	90
3	T2	Read prod_qty for id='11BA'	10	
4	T2	Prod_qty('11BA')=10+15	25	
5	T2	Write(prod_qty for id='11BA')	25	
6	T1	Read prod_qty for id='11BA'	25	115(read after)
7	T1	Read prod_qty for id='55KK'	50	165(read before)
8	T2	Read prod_qty for id='55KK'	50	
9	T2	Prod_qty('55KK')=50+20	70	
10	T2	Write(prod_qty for id='55KK') Commit	70	
11	T1	Read prod_qty for id='99UH'	5	170

CONCURRENCY CONTROL TECHNIQUES

- TWO BASIC CONCURRENCY CONTROL TECHNIQUES ARE:
  - LOCKING
  - TIMESTAMPING

- BOTH ARE CONSERVATIVE (PESSIMISTIC) APPROACHES: DELAY TRANSACTIONS IN CASE THEY CONFLICT WITH OTHER TRANSACTIONS.

- OPTIMISTIC METHODS ASSUME CONFLICT IS RARE AND ONLY CHECK FOR CONFLICTS AT COMMIT.
  - VERSIONING

LOCKING

- TRANSACTION USES LOCKS TO DENY ACCESS TO OTHER TRANSACTIONS AND SO PREVENT INCORRECT UPDATES.
- MOST WIDELY USED APPROACH TO ENSURE READ CONSISTENCY.
- GENERALLY, A TRANSACTION MUST CLAIM A SHARED (READ) OR EXCLUSIVE (WRITE) LOCK ON A DATA ITEM BEFORE READ OR WRITE.
- LOCK PREVENTS ANOTHER TRANSACTION FROM MODIFYING ITEM OR EVEN READING IT, IN THE CASE OF A WRITE LOCK.

LOCKING - BASIC RULES

- IF TRANSACTION HAS **SHARED** LOCK ON ITEM, IT CAN READ BUT NOT UPDATE ITEM.
- IF TRANSACTION HAS **EXCLUSIVE** LOCK ON ITEM, IT CAN BOTH READ AND UPDATE ITEM.
- READS CANNOT CONFLICT, SO MORE THAN ONE TRANSACTION CAN HOLD SHARED LOCKS SIMULTANEOUSLY ON SAME ITEM.
- EXCLUSIVE LOCK GIVES TRANSACTION EXCLUSIVE ACCESS TO THAT ITEM (I.E. THE ITEM CANNOT EVEN BE READ BY OTHER TRANSACTIONS).

LOCKING - BASIC RULES

- SOME SYSTEMS ALLOW TRANSACTION TO UPGRADE READ LOCK TO AN EXCLUSIVE LOCK, OR DOWNGRADE EXCLUSIVE LOCK TO A SHARED LOCK.

LOCKING

- LOCKING DOES NOT GUARANTEE READ CONSISTENCY
- IF A TRANSACTION RELEASES THE LOCK PRIOR TO A COMMIT THEN ANOTHER TRANSACTION COULD READ THE DATA AND THE ORIGINAL TRANSACTION END UP ROLLING BACK RATHER THAN COMMITTING.
- UNCOMMITTED DATA PROBLEM

TWO-PHASE LOCKING (2PL)

- TRANSACTION FOLLOWS 2PL PROTOCOL IF ALL LOCKS ARE HELD UNTIL THE TRANSACTION ENDS (EITHER COMMITTED OR ROLLED BACK)

- TWO PHASES FOR TRANSACTION:
  - GROWING PHASE - ACQUIRES ALL LOCKS BUT CANNOT RELEASE ANY LOCKS.
  - SHRINKING PHASE - RELEASES LOCKS BUT CANNOT ACQUIRE ANY NEW LOCKS.

PREVENTING LOST UPDATE PROBLEM USING 2PL

Time	T_1	T_2	bal_x
t_1		begin_transaction	100
t_2	begin_transaction	write_lock(bal_x)	100
t_3	write_lock(bal_x)	read(bal_x)	100
t_4	WAIT	$bal_x = bal_x + 100$	100
t_5	WAIT	write(bal_x)	200
t_6	WAIT	commit/unlock(bal_x)	200
t_7	read(bal_x)		200
t_8	$bal_x = bal_x - 10$		200
t_9	write(bal_x)		190
t_{10}	commit/unlock(bal_x)		190

PREVENTING UNCOMMITTED DATA PROBLEM USING 2PL

Time	T_3	T_4	bal_x
t_1		begin_transaction	100
t_2		write_lock(bal_x)	100
t_3		read(bal_x)	100
t_4	begin_transaction	$bal_x = bal_x + 100$	100
t_5	write_lock(bal_x)	write(bal_x)	200
t_6	WAIT	rollback/unlock(bal_x)	100
t_7	read(bal_x)		100
t_8	$bal_x = bal_x - 10$		100
t_9	write(bal_x)		90
t_{10}	commit/unlock(bal_x)		90

PREVENTING INCONSISTENT ANALYSIS

Time	T_5	T_6	$\mathbf{bal_x}$	$\mathbf{bal_y}$	$\mathbf{bal_z}$	sum
t_1		begin_transaction	100	50	25	
t_2	begin_transaction	sum = 0	100	50	25	0
t_3	write_lock($\mathbf{bal_x}$)		100	50	25	0
t_4	read($\mathbf{bal_x}$)	read_lock($\mathbf{bal_x}$)	100	50	25	0
t_5	$\mathbf{bal_x} = \mathbf{bal_x} - 10$	WAIT	100	50	25	0
t_6	write($\mathbf{bal_x}$)	WAIT	90	50	25	0
t_7	write_lock($\mathbf{bal_z}$)	WAIT	90	50	25	0
t_8	read($\mathbf{bal_z}$)	WAIT	90	50	25	0
t_9	$\mathbf{bal_z} = \mathbf{bal_z} + 10$	WAIT	90	50	25	0
t_{10}	write($\mathbf{bal_z}$)	WAIT	90	50	35	0
t_{11}	commit/unlock($\mathbf{bal_x}$, $\mathbf{bal_z}$)	WAIT	90	50	35	0
t_{12}		read($\mathbf{bal_x}$)	90	50	35	0
t_{13}		sum = sum + $\mathbf{bal_x}$	90	50	35	90
t_{14}		read_lock($\mathbf{bal_y}$)	90	50	35	90
t_{15}		read($\mathbf{bal_y}$)	90	50	35	90
t_{16}		sum = sum + $\mathbf{bal_y}$	90	50	35	140
t_{17}		read_lock($\mathbf{bal_z}$)	90	50	35	140
t_{18}		read($\mathbf{bal_z}$)	90	50	35	140
t_{19}		sum = sum + $\mathbf{bal_z}$	90	50	35	175
t_{20}		commit/unlock($\mathbf{bal_x}$, $\mathbf{bal_y}$, $\mathbf{bal_z}$)	90	50	35	175

DEADLOCK

- AN IMPASSE THAT MAY RESULT WHEN TWO (OR MORE) TRANSACTIONS ARE EACH WAITING FOR LOCKS HELD BY THE OTHER TO BE RELEASED.

Time	T_{17}	T_{18}
t_1	begin_transaction	
t_2	write_lock(**bal$_x$**)	begin_transaction
t_3	read(**bal$_x$**)	write_lock(**bal$_y$**)
t_4	**bal$_x$** = **bal$_x$** − 10	read(**bal$_y$**)
t_5	write(**bal$_x$**)	**bal$_y$** = **bal$_y$** + 100
t_6	write_lock(**bal$_y$**)	write(**bal$_y$**)
t_7	WAIT	write_lock(**bal$_x$**)
t_8	WAIT	WAIT
t_9	WAIT	WAIT
t_{10}	⋮	WAIT
t_{11}	⋮	⋮

DEADLOCK

- ONLY ONE WAY TO BREAK DEADLOCK: ABORT ONE OR MORE OF THE TRANSACTIONS.
- DEADLOCK SHOULD BE TRANSPARENT TO USER, SO DBMS SHOULD RESTART TRANSACTION(S).
- THREE GENERAL TECHNIQUES FOR HANDLING DEADLOCK:
  - TIMEOUTS.
  - DEADLOCK PREVENTION.
  - DEADLOCK DETECTION AND RECOVERY.

TIMEOUTS

- TRANSACTION THAT REQUESTS LOCK WILL ONLY WAIT FOR A SYSTEM-DEFINED PERIOD OF TIME.
- IF LOCK HAS NOT BEEN GRANTED WITHIN THIS PERIOD, LOCK REQUEST TIMES OUT.
- IN THIS CASE, DBMS ASSUMES TRANSACTION MAY BE DEADLOCKED, EVEN THOUGH IT MAY NOT BE, AND IT ABORTS AND AUTOMATICALLY RESTARTS THE TRANSACTION.

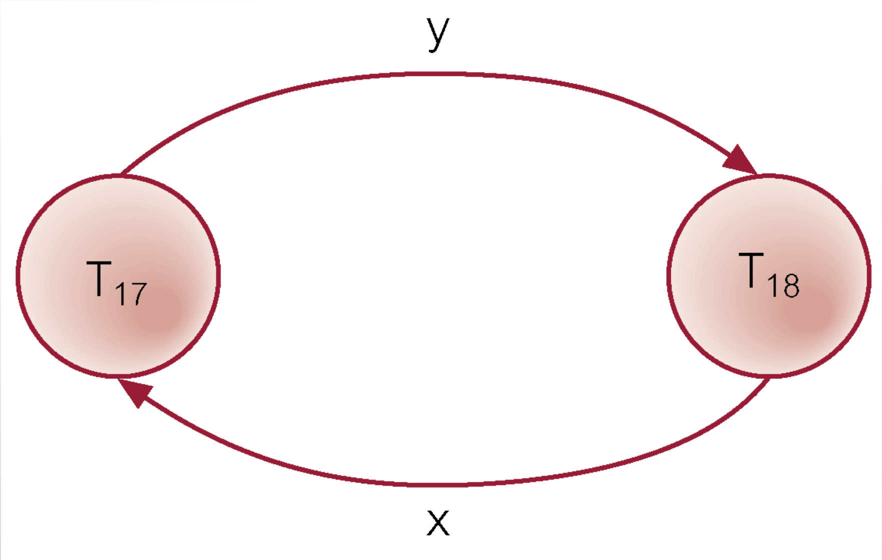
DEADLOCK PREVENTION

- DBMS LOOKS AHEAD TO SEE IF TRANSACTION WOULD CAUSE DEADLOCK AND NEVER ALLOWS DEADLOCK TO OCCUR.
- COULD ORDER TRANSACTIONS USING TRANSACTION TIMESTAMPS:
  - WAIT-DIE: ONLY AN OLDER TRANSACTION CAN WAIT FOR YOUNGER ONE, OTHERWISE TRANSACTION IS ABORTED (DIES) AND RESTARTED WITH SAME TIMESTAMP.
  - WOUND-WAIT: ONLY A YOUNGER TRANSACTION CAN WAIT FOR AN OLDER ONE. IF OLDER TRANSACTION REQUESTS LOCK HELD BY YOUNGER ONE, YOUNGER ONE IS ABORTED (WOUNDED).

DEADLOCK DETECTION AND RECOVERY

- DBMS ALLOWS DEADLOCK TO OCCUR BUT RECOGNIZES IT AND BREAKS IT.
- USUALLY HANDLED BY CONSTRUCTION OF WAIT-FOR GRAPH (WFG) SHOWING TRANSACTION DEPENDENCIES:
  - CREATE A NODE FOR EACH TRANSACTION.
  - CREATE EDGE TI -> TJ, IF TI WAITING TO LOCK ITEM LOCKED BY TJ.
- DEADLOCK EXISTS IF AND ONLY IF WFG CONTAINS CYCLE.
- WFG IS CREATED AT REGULAR INTERVALS.

EXAMPLE - WAIT-FOR-GRAPH (WFG)



RECOVERY FROM DEADLOCK DETECTION

- SEVERAL ISSUES:
  - CHOICE OF DEADLOCK VICTIM;
  - HOW FAR TO ROLL A TRANSACTION BACK;
  - AVOIDING STARVATION.

TIMESTAMPING

- TRANSACTIONS ORDERED GLOBALLY SO THAT OLDER TRANSACTIONS, TRANSACTIONS WITH SMALLER TIMESTAMPS, GET PRIORITY IN THE EVENT OF CONFLICT.
- CONFLICT IS RESOLVED BY ROLLING BACK AND RESTARTING TRANSACTION.
- NO LOCKS SO NO DEADLOCK.

TIMESTAMPING

- TIMESTAMP
  - A UNIQUE IDENTIFIER CREATED BY DBMS THAT INDICATES RELATIVE STARTING TIME OF A TRANSACTION.
- CAN BE GENERATED BY USING SYSTEM CLOCK AT TIME TRANSACTION STARTED, OR BY INCREMENTING A LOGICAL COUNTER EVERY TIME A NEW TRANSACTION STARTS.

TIMESTAMPING

- READ/WRITE PROCEEDS ONLY IF LAST READ/WRITE ON THAT DATA ITEM WAS CARRIED OUT BY AN OLDER TRANSACTION.
- OTHERWISE, TRANSACTION REQUESTING READ/WRITE IS RESTARTED AND GIVEN A NEW TIMESTAMP.
- ALSO TIMESTAMPS FOR DATA ITEMS:
  - READ-TIMESTAMP - TIMESTAMP OF LAST TRANSACTION TO READ ITEM;
  - WRITE-TIMESTAMP - TIMESTAMP OF LAST TRANSACTION TO WRITE ITEM.

TIMESTAMPING EXAMPLE

John	Time	Marsha	Bal_x	Data Item Timestamp
read (bal_x)	t1		100	Last read by John at t1
	t2	read (bal_x)	100	Last read by Marsha at t2
$bal_x = bal_x - 50$	t3		50	
write (bal_x) *	t4		50	
roll back	t5	$bal_x = bal_x – 10$	90	
	t6	write (bal_x)	90	Last updated by Marsha at t6
read (bal_x)	t7		90	Last read by John at t7
	t8		90	
$bal_x = bal_x - 50$	t9		40	
write (bal_x) **	t10		40	Last update by John at t10
	t11			
	t12			

TIMESTAMPING EXAMPLE

* PROBLEM OCCURS HERE – JOHN HAS TRIED TO UPDATE A DATA ITEM WHICH WAS LAST READ BY ANOTHER TRANSACTION (MARSHA). THEREFORE HIS TRANSACTION MUST BE ROLLED BACK - ABORTED AND RESTARTED, AND GIVEN A NEW TIMESTAMP. **FROM TIME T6 AND ONWARDS MARSHA IS NOW THE OLDER TRANSACTION, WHEREAS JOHN'S TRANSACTION IS THE NEWER OF THE TWO (HAVING BEEN GIVEN A NEW TIMESTAMP).**

** THIS TIME JOHN'S UPDATE TO THE BALANCE IS ALLOWED TO PROCEED, AS THE DATA ITEM HAS NOT BEEN READ/UPDATED BY ANYONE ELSE SINCE HIS TRANSACTION WAS RESTARTED.

OPTIMISTIC TECHNIQUES

- BASED ON ASSUMPTION THAT CONFLICT IS RARE AND MORE EFFICIENT TO LET TRANSACTIONS PROCEED WITHOUT DELAYS TO ENSURE SERIALIZABILITY.
- AT COMMIT, CHECK IS MADE TO DETERMINE WHETHER CONFLICT HAS OCCURRED.
- IF THERE IS A CONFLICT, TRANSACTION MUST BE ROLLED BACK AND RESTARTED.
- POTENTIALLY ALLOWS GREATER CONCURRENCY THAN TRADITIONAL PROTOCOLS.

VERSIONING

- VERSIONING OF DATA CAN BE USED TO INCREASE CONCURRENCY.
- BASIC TIMESTAMP ORDERING PROTOCOL ASSUMES ONLY ONE VERSION OF DATA ITEM EXISTS, AND SO ONLY ONE TRANSACTION CAN ACCESS DATA ITEM AT A TIME.
- CAN ALLOW MULTIPLE TRANSACTIONS TO READ AND WRITE DIFFERENT VERSIONS OF SAME DATA ITEM.
- IN MULTIVERSION CONCURRENCY CONTROL, EACH WRITE OPERATION CREATES NEW VERSION OF DATA ITEM WHILE RETAINING OLD VERSION.
- NEW VERSIONS ARE LATER MERGED INTO THE DATABASE; CONFLICTS ARE DEALT WITH IF THEY ARISE

GRANULARITY OF DATA ITEMS

- SIZE OF DATA ITEMS CHOSEN AS UNIT OF PROTECTION BY CONCURRENCY CONTROL PROTOCOL.
- RANGING FROM COARSE TO FINE:
  - THE ENTIRE DATABASE.
  - A FILE.
  - A PAGE (OR AREA OR DATABASE SPACED).
  - A RECORD.
  - A FIELD VALUE OF A RECORD.

GRANULARITY OF DATA ITEMS

- TRADEOFF:
  - COARSER, THE LOWER THE DEGREE OF CONCURRENCY;
  - FINER, MORE LOCKING INFORMATION THAT IS NEEDED TO BE STORED.
- BEST ITEM SIZE DEPENDS ON THE TYPES OF TRANSACTIONS.