Part 11: Updates in SQL

References:

- Elmasri/Navathe:Fundamentals of Database Systems, 3rd Edition, 1999.
  Chap. 8, "SQL — The Relational Database Standard"
- Silberschatz/Korth/Sudarshan: Database System Concepts, 3rd Ed., McGraw-Hill, 1999.
  Chapter 4: "SQL".
- Kemper/Eickler: Datenbanksysteme (in German), 4th Ed., Oldenbourg, 1997.
  Chapter 4: Relationale Anfragesprachen (Relational Query Languages).
- Lipeck: Skript zur Vorlesung Datenbanksysteme (in German), Univ. Hannover, 1996.
- Date/Darwen: A Guide to the SQL Standard, Fourth Edition, Addison-Wesley, 1997.
- van der Lans: SQL, Der ISO-Standard (in German), Hanser, 1990.
- Sunderraman: Oracle Programming, A Primer. Addison-Wesley, 1999.
- Oracle8 SQL Reference, Oracle Corporation, 1997, Part No. A58225-01.
- Oracle8 Concepts, Release 8.0, Oracle Corporation, 1997, Part No. A58227-01.
- Chamberlin: A Complete Guide to DB2 Universal Database. Morgan Kaufmann, 1998.
- Microsoft SQL Server Books Online: Accessing and Changing Data.
- H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, P. O'Neil:
  A critique of ANSI SQL isolation levels. In Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, 1–10, 1995.

Objectives

After completing this chapter, you should be able to:

- use `INSERT`, `UPDATE`, and `DELETE` commands in SQL.
- use `COMMIT` and `ROLLBACK` in SQL.
- explain the concept of a transaction.

  Mention a typical example and explain the ACID-properties.
- explain what happens when several users access the database concurrently.

  Explain locks and possibly multi-version concurrency control.
  When does one need to add "`FOR UPDATE`" to a query?

Overview

Update Commands

- SQL has three commands to change the DB state:
  - ◇ `INSERT`: For inserting new rows in a table.
  - ◇ `UPDATE`: For changing values in existing rows.
  - ◇ `DELETE`: For deleting rows from a table.
- In addition, SQL has two commands for ending transactions:
  - ◇ `COMMIT`: Successful end, make changes durable.
  - ◇ `ROLLBACK`: Transaction failed, undo all changes.

Example Database

STUDENTS

<u>SID</u>	FIRST	LAST	EMAIL
101	Ann	Smith	···
102	Michael	Jones	(null)
103	Richard	Turner	···
104	Maria	Brown	···

EXERCISES

<u>CAT</u>	<u>ENO</u>	TOPIC	MAXPT
H	1	Rel. Algeb.	10
H	2	SQL	10
M	1	SQL	14

RESULTS

<u>SID</u>	<u>CAT</u>	<u>ENO</u>	POINTS
101	H	1	10
101	H	2	8
101	M	1	12
102	H	1	9
102	H	2	9
102	M	1	10
103	H	1	5
103	M	1	7

INSERT Command

- The `INSERT`-command has two forms:
  - ◇ For inserting a single row with new data.
  - ◇ For inserting the result of a query.
- The second form can e.g. be used to copy a table.

  But one still has to first define the goal table with `CREATE TABLE`. Oracle has `CREATE TABLE ... AS SELECT ...`.
- In SQL-92, there is only one general `INSERT` command: "`VALUES`" (first form) and "`SELECT`" (second form) are both table expressions.

INSERT, 1st Form (1)

- Example:

```
INSERT INTO STUDENTS
VALUES (105, 'Nina', 'Brass', NULL);
```

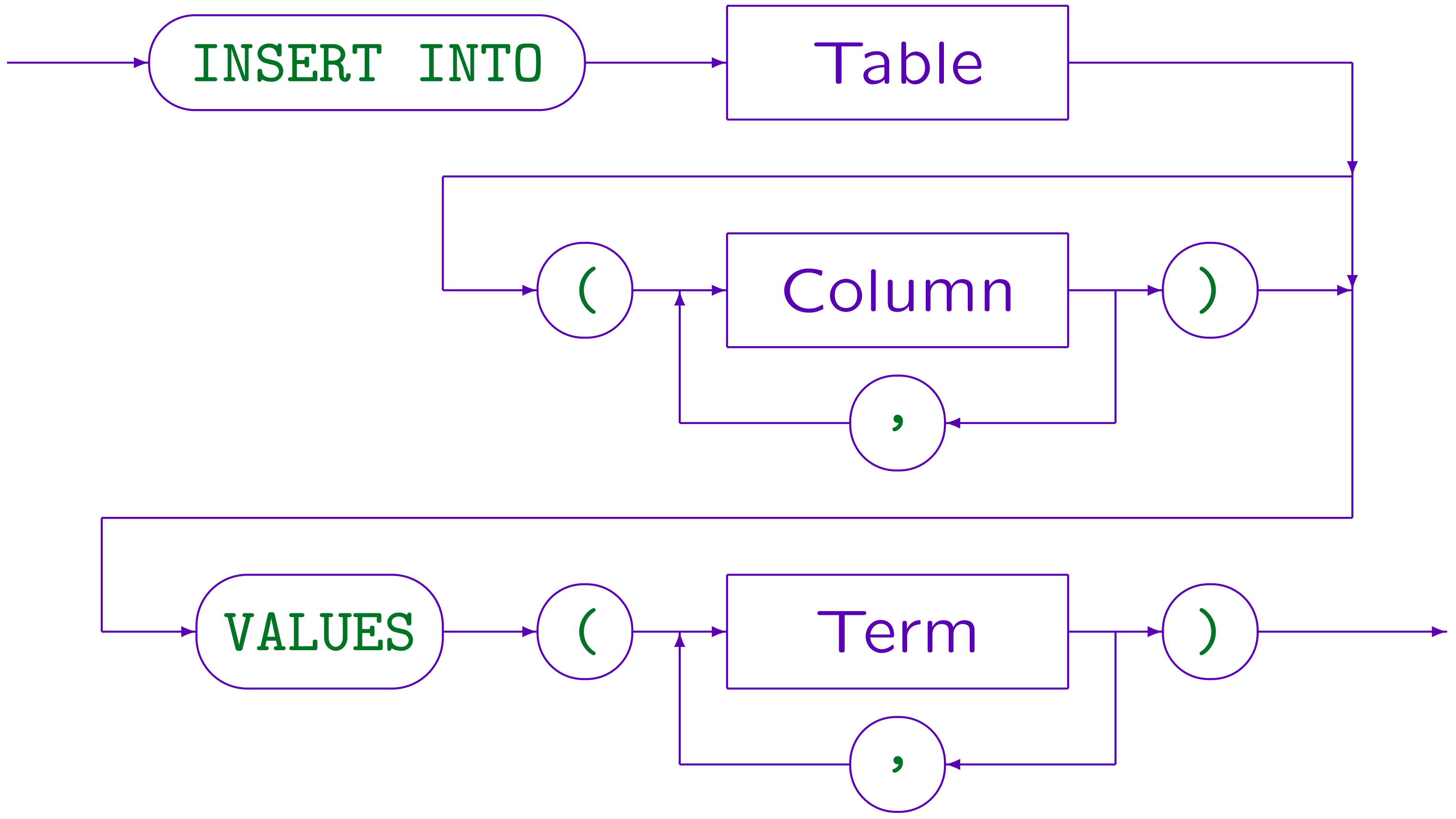
- Possible values are: constants, the keywords "`NULL`", "`DEFAULT`", any term like "`100+5`", "`SYSDATE`", etc.

- One can specify values for only a subset of the columns:

```
INSERT INTO STUDENTS(SID, FIRST, LAST)
VALUES (105, 'Nina', 'Brass')
```

  The default value is inserted in the other column "`EMAIL`" (e.g. `NULL`).

INSERT, 1st Form (2)



INSERT, 2nd Form (1)

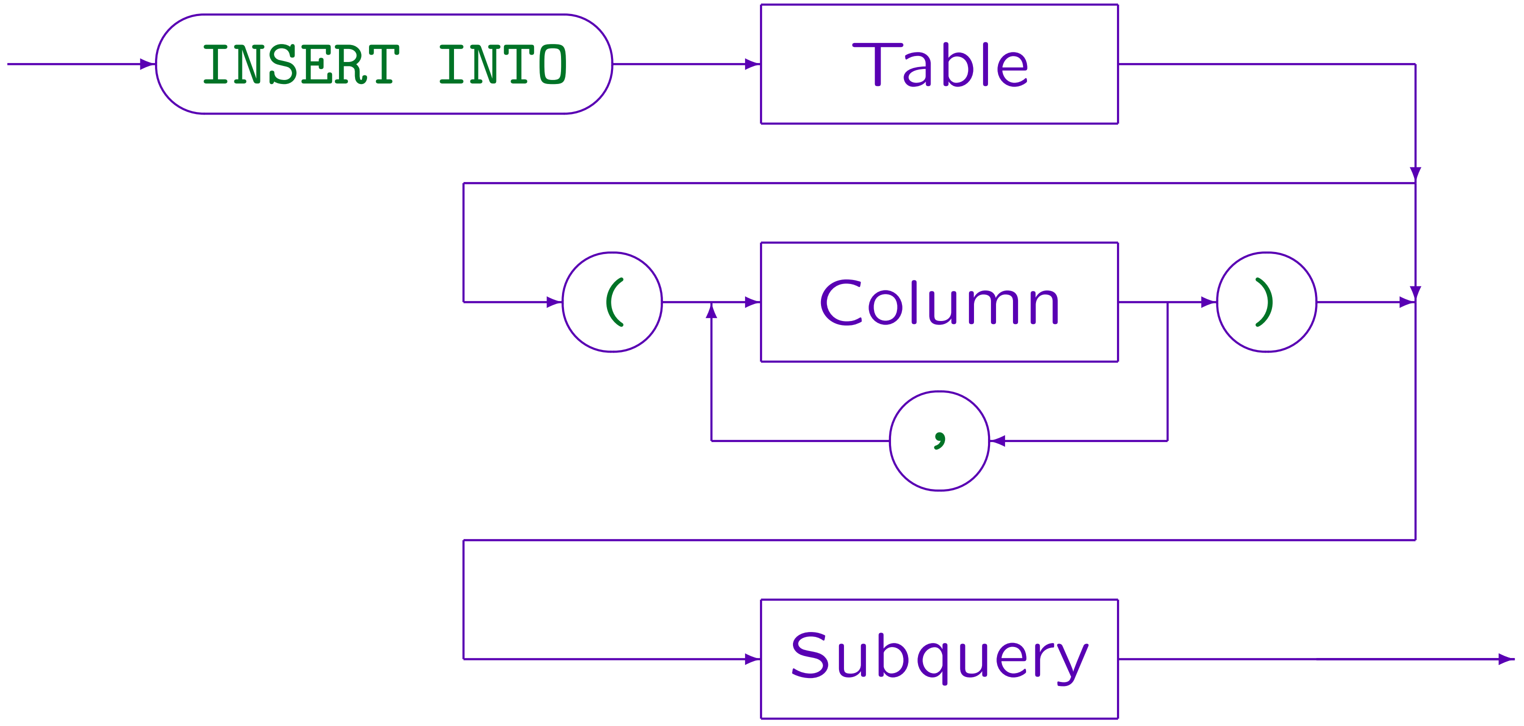
- Example:

```
INSERT INTO FINAL_EXAMS(TERM, ENO, TOPIC, PCT)
SELECT '2002', E.ENO, E.TOPIC,
       AVG(R.POINTS/E.MAXPT)*100
FROM   EXERCISES E, RESULTS R
WHERE  E.CAT='F' AND R.CAT='F' AND E.ENO=R.ENO
GROUP  BY E.ENO, E.TOPIC
```

- The subquery will first be fully evaluated before tuples are inserted.

  So the table to be modified can be used in the subquery with a defined result and without the risk of endless loops.

INSERT, 2nd Form (2)



DELETE (1)

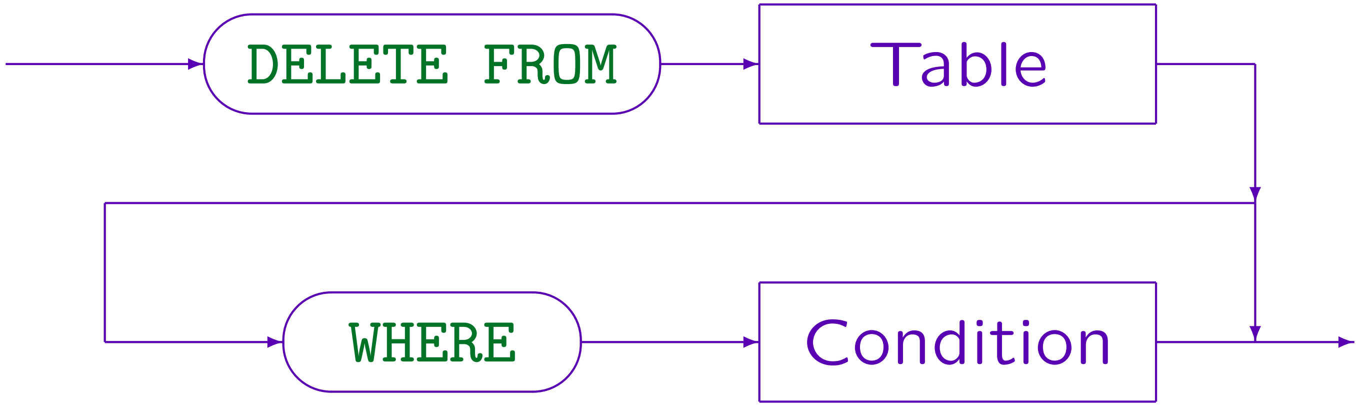
- E.g.: Delete all results for Ann Smith:

```
DELETE FROM RESULTS
WHERE  SID IN (SELECT SID
               FROM   STUDENTS
               WHERE  FIRST = 'Ann'
               AND    LAST  = 'Smith')
```

- Be careful: Without the WHERE-condition, all tuples are deleted!

  It might be be possible to use "ROLLBACK" if something went wrong. In order to use ROLLBACK, one must find the error before the transaction is ended. So look at the table after the change whether it is really what you expected. Some SQL interfaces immediately commit any change (autocommit), then there is no possibility for an undo.

DELETE (2)



TRUNCATE (1)

- Oracle, SQL Server, and MySQL (but not DB2 and Access) have a command

  `TRUNCATE TABLE ⟨Table Name⟩`

  which deletes all rows from the table and frees the disk space occupied by the table.

- This is similar to a `DROP TABLE`, but the table definition (schema information) remains in the system, only the table data is deleted.

  Thus, references in grants (access rights), views, triggers, stored procedures etc. are not affected.

TRUNCATE (2)

- In contrast to `DELETE`, `TRUNCATE` cannot be rolled back (undone). Therefore it is much faster.

  Also, at least in Oracle `DELETE` would not actually free any disk space. Another problem is that if one tries to `DELETE` all rows of a large table, the rollback segment (storage space for the undo information) might be too small (in Oracle). Then "`DELETE FROM` ⟨`Table Name`⟩" gives an error message and nothing is deleted.

- `TRUNCATE` is not part of the SQL-92 standard.

  But it does appear in the Oracle Certification exam.

UPDATE (1)

- The UPDATE command is for changing attribute values of selected tuples.
- E.g. give all solutions for Exercise 1 of the midterm exam 2 bonus points:

```
UPDATE RESULTS
SET    POINTS = POINTS + 2
WHERE  CAT = 'M' AND ENO = 1
```

- The right hand side of the assignment can use the old values of all attributes of the selected tuple.

  The WHERE-condition and the terms that define the new values are evaluated for all tuples before any update is done. The right hand side of the assignment can also be the keyword NULL.

UPDATE (2)

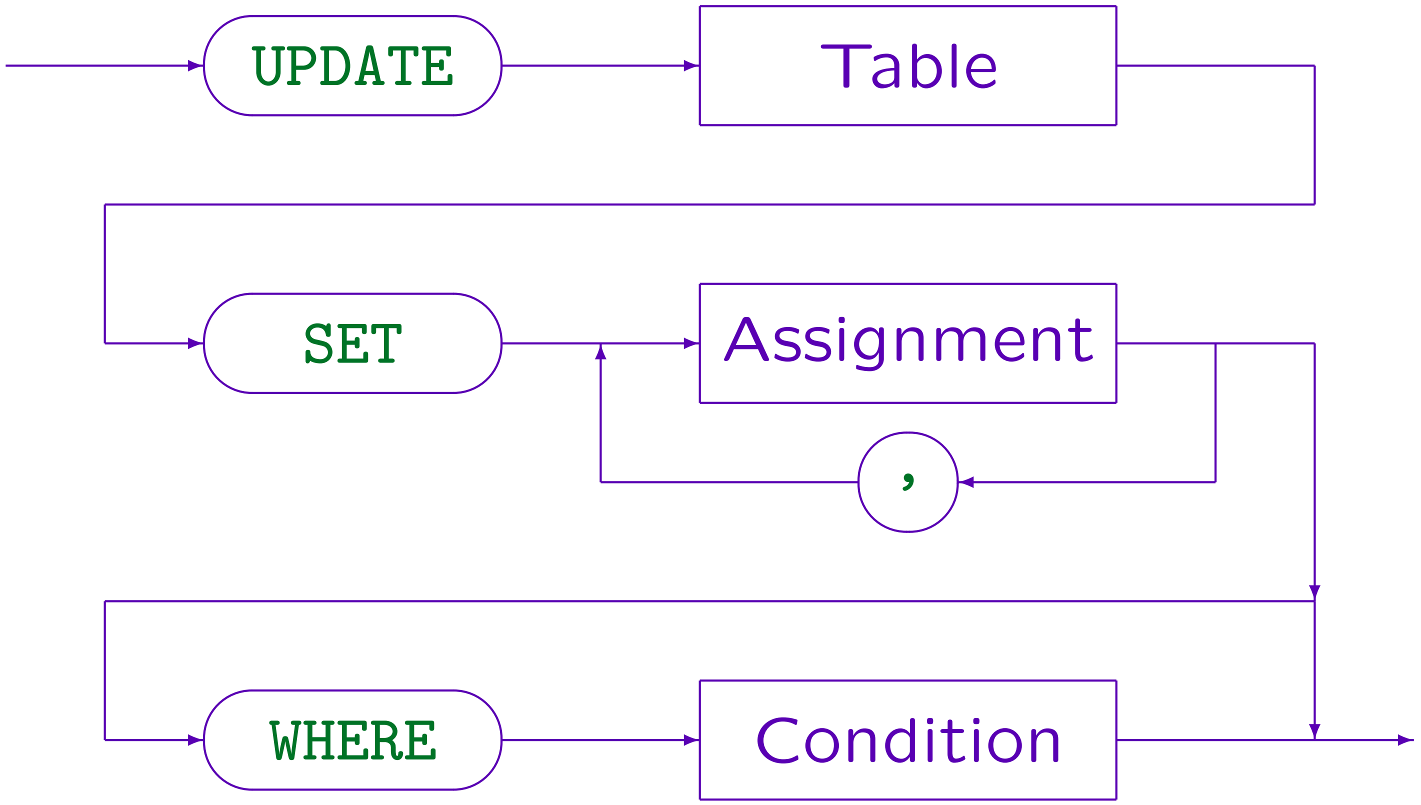
- In SQL-92, Oracle, DB2, and SQL server (but not in SQL-86, MySQL, and Access), a subquery can be used to compute the new value.

  The subquery must return exactly one row (with a single column). If it returns no rows, a null value is used.

- Multiple columns can be changed in one `UPDATE` statement:

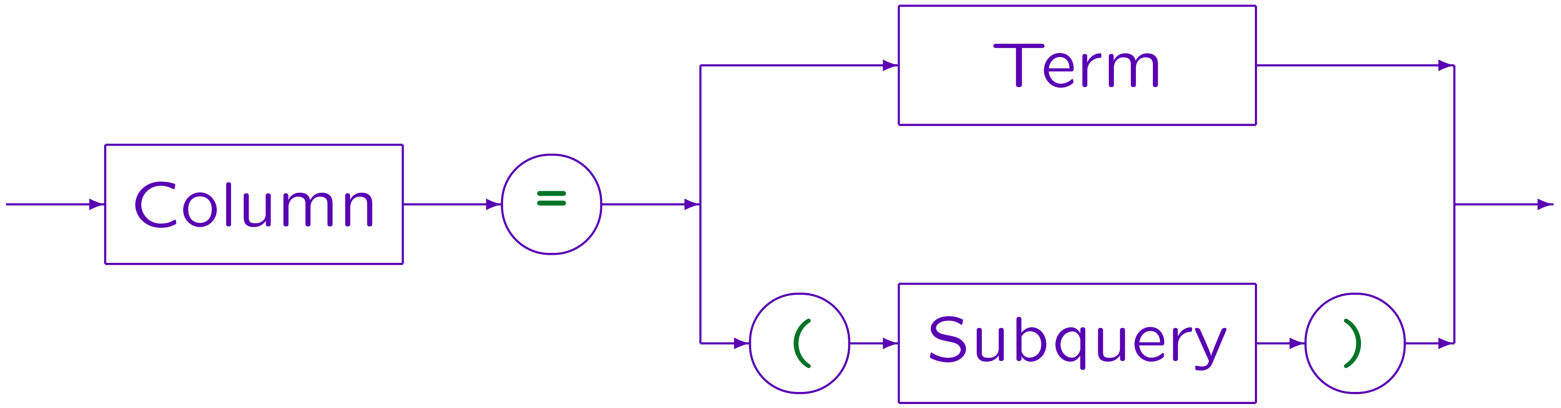
```
UPDATE EXERCISES
SET    TOPIC = 'Advanced SQL',
       MAXPT = 12
WHERE  CAT = 'H' AND ENO = 2
```

UPDATE (3)



UPDATE (4)

Assignment:



- SQL-86, MySQL, and Access do not support subqueries on the right hand side.
- In SQL-92, DB2, and SQL-Server a subquery can be used as a term, so the subquery case is already contained in the first alternative. Only for Oracle 8, the subquery must be explicitly mentioned.

Overview

1. Update commands in SQL
2. Transactions
3. Concurrent Accesses: Examples
4. Introduction to Concurrency Control Theory

Transactions (1)

- A transaction is a sequence of DB commands, especially updates, which the DBMS treats as a unit.
- E.g. a transfer of 50 dollars from account 1 to account 2 consists of
  - ◇ checking the current balance and credit limit of account 1,
  - ◇ decreasing the balance of 1 by 50 (debit),
  - ◇ increasing the balance of 2 by 50 (credit),
  - ◇ writing entries to a banking statement (history of changes) for both accounts.

Transactions (2)

Transactions are characterized by the ACID-Properties:

- Atomicity

  A transaction is either executed completely, or not at all.

- Consistency

  A transaction does not lead to a violation of the constraints.

- Isolation

  Transactions of concurrent users do not interfere with each other.

- Durability

  Data stored by committed transactions is safe.

Transactions (3)

Atomicity:

- Modern DBMS guarantee that a transaction is
  - ◇ either executed in total,
  - ◇ or not at all ("all or nothing" principle).
- If the transaction cannot be executed until the end (e.g. because of a power failure or system crash), the database state before the transaction has begun will be restored when the DBMS is started the next time.

Transactions (4)

- Atomicity gives the user powerful undo features:
  - ◇ As long as the transaction has not been declared as complete (with `COMMIT`), all changes can be undone (with `ROLLBACK`).
  - ◇ In most DBMS, one can only undo the complete transaction (not only the last command).
  - ◇ However, in Oracle and SQL Server it is possible to set “savepoints” within a transaction and undo all changes after such a savepoint.
  - ◇ After `COMMIT`, no undo is possible.

Transactions (5)

Durability:

- When a DBMS acknowledges the successful end of a transaction, the changes are guaranteed durable.
- The changes are stored on disk — they are not lost even if there is a power failure one second later.

  In operating systems, one often cannot be sure whether the data is on disk or still in a buffer.
- Larger DBMS have powerful backup and recovery mechanisms: Even if a disk fails, no data is lost.

  In contrast, OS utilities typically create only one backup per day.

Transactions (6)

- Atomicity and durability together mean that there is one point in time which lies between
  - ◇ the user telling the system that the transaction is complete (`COMMIT`), and
  - ◇ the system telling the user that this command was successfully processed,

  when all changes become effective.

  If the system crashes before this point, the database state is not changed. If the system crashes after this point, the database state contains all changes which the transaction has executed.

Transactions (7)

Isolation:

- A larger DBMS supports concurrent accesses of multiple users.
- Without control, this could have strange effects including lost updates.
- However, a DBMS tries to create the impression that every transaction runs in isolation, i.e. has exclusive access to the complete database.
- Usually, a DBMS automatically manages locks on DB objects (e.g. tuples, tables) for this purpose.

Transactions (8)

Consistency:

- User and system can be sure that the current state is the result of a sequence of completely executed transactions.
- The user must ensure that each transaction, if applied fully and in isolation to a consistent state (i.e. one that satisfies the constraints), will produce a consistent state.

  Modern DBMS offer some support: Keys, foreign keys, `NOT NULL` and `CHECK` constraints can be specified declaratively. For more complex constraints, there are triggers.

Transactions (9)

- The consistency is in part consequence of the other three properties and in part user responsibility.
- However, it should be appreciated that all data structures are kept consistent.

  If the users stores redundant data, he/she is responsible for updating them in the same transaction that changed the original data. But then the system ensures that even if there is a power failure in between, the two never get out of sync. This is also important for the internal data structures of the DBMS: E.g. it has to manage indexes (redundant data structures to quickly find rows with a given column value). It is very important that all rows in the table are also represented in the index, otherwise the query behaviour would get unpredictable.

Transaction Management (1)

- SQL has no command for beginning a transaction.

  A transaction begins when one logs into the database and every time after a transaction is finished.

- A transaction is finished with
  - ◇ `COMMIT [WORK]`: Makes changes durable.
  - ◇ `ROLLBACK [WORK]`: Undoes changes.

  Some commands, such as `DROP TABLE`, will automatically execute a `COMMIT` in Oracle, so they cannot be undone.

- The "multi user" features of a DBMS (e.g. Oracle) can be tried by starting the SQL interpreter several times concurrently in different windows.

Transaction Management (2)

- If one works with the database for a long time, one should COMMIT the work from time to time.

  If there should be a power failure etc., only the changes after the last COMMIT are lost. The DBMS also locks tuples for the transaction until the user decides to COMMIT or ROLLBACK. Long transactions might hinder other users.

- If one leaves Oracle SQL*Plus normally, changes are automatically committed (so nothing is lost when one forgets to use COMMIT).
- However, if one simply closes the window or logs out of the operating system a ROLLBACK is done.

Transaction Management (3)

- Some systems offer an "autocommit mode": Then a commit is automatically done after every update. But this means that changes cannot be undone.

  In SQL*Plus this mode can be selected with "`set autocommit on`" (it is off by default). SQL Server by default runs in autocommit mode, but "`BEGIN TRANSACTION`" gives the normal mode. In DB2, `COMMIT` and `ROLLBACK` work as usual. MySQL basically has only autocommit mode, except when one uses special table types that support transactions. Access automatically commits all changes and does not understand `COMMIT` and `ROLLBACK`.

- Some commands (such as `CREATE TABLE`) may automatically commit the transaction: They cannot be undone (and all previous updates get committed).

Overview

Goal: Isolation

- Every user should have the impression that he/she has exclusive access to the database for the duration of the entire transaction.
- All other transactions must appear as if completed before his/her transaction, or started after it.
- What users see and the changes they perform must be equivalent to a serial schedule of transactions (as if there were only one terminal from which the DB can be accessed).

Goal: Performance

- While one transaction waits for the disk or user input, the DBMS should work on another transaction (instead of simply being idle).
- A long running transaction must be interrupted from time to time to allow shorter transactions to complete.

  Overall this provides quicker response times: Otherwise many short transactions will queue up after one long one.
- Concurrent transactions might make use of parallel hardware.

Problems (1)

- The two goals are in conflict with each other: 100% isolation results in very little parallelism — often entire tables must be locked.
- SQL has no “begin transaction” command. As long as there are only queries, it is not clear whether they form one big transaction or are each a separate transaction.
- DBMS guarantee “some isolation” and offer the mechanisms to reach full isolation, but need help from the programmer.

Problems (2)

- Application programming is simplified, because the programmer normally does not have to worry about the possibility of concurrent transactions.
- However, programmers must be aware of the few cases where special commands must be used.
- Errors due to concurrent execution will not be noticed during normal debugging — real system load is needed and even then it may take months until a fatal situation occurs.

Locks (1)

- Most systems use locks for concurrency control.
- Locks can be used for objects of different granularity: Tables, disk blocks, tuples, attributes.
- If Transaction A holds a lock on an object, and another Transaction B also wants to lock this object, B must wait.

  B is suspended from execution (put to sleep). The lock manager has for each lock a list of waiting transactions. So when Transaction A takes the lock off, the lock manager can wake B up. Then B can acquire the lock.

Locks (2)

Transaction A	Transaction B

`UPDATE EMP`
`SET SAL = SAL * 2`
`WHERE ENAME = 'JAMES'`
⟶ 1 row updated.	

	`UPDATE EMP`
	`SET SAL = SAL / 2`
	`WHERE ENAME = 'JAMES'`
	⟶ (no reaction)

`COMMIT`	
	⟶ 1 row updated.

Locks (3)

- Often DBMS have (at least) two kinds of locks:
  - ◇ Exclusive locks (X) are used for write accesses.

    They exclude any other access (read or write).
  - ◇ Shared locks (S) are used for read accesses.

    They exclude write accesses, but allow read accesses by other transactions.
- This is shown in a "lock compatibility matrix":

Requested Lock	Existing Lock		
	None	S	X
S	+	+	−
X	+	−	−

Deadlocks (1)

- It is possible that two transactions wait on locks which are held by the other transaction:

Transaction A	Transaction B

`UPDATE EMP SET ...`
`WHERE ENAME = 'JAMES'`	

	`UPDATE EMP SET ...`
	`WHERE ENAME = 'ALLEN'`

	`UPDATE EMP SET ...`
	`WHERE ENAME = 'JAMES'`

`UPDATE EMP SET ...`
`WHERE ENAME = 'ALLEN'`	

Deadlocks (2)

- In this case, one of the transactions involved in the deadlock must be rolled back.

  This will free the locks held by this transaction, and the other transaction can continue. Oracle does not roll back a transaction automatically, but will end one of the `UPDATE` requests with an error. In this case the application program should call `ROLLBACK`.

- The deadlock test is costly, therefore some systems do it only from time to time (or only after a transaction has waited some time for a lock).

- Application programs should be analyzed for possible deadlocks.

Dirty Read Problem (1)

- Transaction A sets a salary to 1 Mio $, discovers the error, calls `ROLLBACK`. B checks the budget.

Transaction A	Transaction B

`UPDATE EMP`
`SET SAL = 1000000`
`WHERE ENAME = 'JAMES'`	

	`SELECT SAL`
	`FROM EMP`
	`WHERE ENAME = 'JAMES'`
	$\longrightarrow$ 1000000

`ROLLBACK`	**(Normally excluded)**

Dirty Read Problem (2)

- In the above schedule, Transaction B sees data which “never existed”.

  Also if Transaction A later again changes the balance by another update, we would speak of a “Dirty Read” (even if Transaction A finally commits).

- No transaction should be able to observe an intermediate state of another transaction.

- Transactions should see a transaction-consistent state, i.e. the result of a sequence of committed transactions (plus its own changes).

Dirty Read Problem (3)

- It is not difficult to exclude dirty reads, and most DBMS automatically do that.
- The schedule on slide 11-42 cannot occur in modern DBMS. The programmer does not have to worry about dirty reads.
- There are basically two solutions for the dirty read problem that are used in different systems:
  - ◇ Exclusive locks on changed tuples.
  - ◇ Multi-Version concurrency control.

Dirty Read Problem (4)

Solution with Locks:

- The system sets write locks on tuples changed by a transaction and keeps these locks until the commit.

  The locks are set before the change and removed after the commit (when the log entries are successfully written to the disk). Therefore, the uncommitted data is not accessible for other transactions.

- A transaction which wants to read a tuple tries to acquire a read lock for it. This is possible only if there is no write lock.

  The read lock can be taken off immediately after the read is processed (this is sufficient to avoid dirty reads).

Dirty Read Problem (5)

"Multi Version Concurrency Control" (Oracle):

- For read accesses, Oracle will restore versions of the accessed DB blocks which correspond to a state after the last committed transaction.
- I.e. all uncommitted changes are undone for the purpose of this read only.

  When a query accesses table data, even committed changes of other transactions are undone if the commit was after the evaluation of the query started. Oracle guarantees a consistent state for long-running SELECT queries. Oracle does not guarantee a consistent state for successive queries ("non-repeatable read problem").

Dirty Read Problem (6)

Transaction A	Transaction B

`SELECT SAL FROM EMP`
`WHERE ENAME = 'JAMES'`
$\longrightarrow$ 950	

	`UPDATE EMP`
	`SET SAL = SAL * 2`
	`WHERE ENAME = 'JAMES'`
	`SELECT ...` $\longrightarrow$ 1900

`SELECT ...` $\longrightarrow$ 950	
	`COMMIT`
`SELECT ...` $\longrightarrow$ 1900	

Lost Update (1)

- Consider two updates running concurrently.

  Before they are executed, the salary of James is 950.

Transaction A	Transaction B

`UPDATE EMP`
`SET SAL = SAL * 2`
`WHERE ENAME = 'JAMES'`
	`UPDATE EMP`
	`SET SAL = SAL + 50`
	`WHERE ENAME = 'JAMES'`

- In a serial execution of the two transactions, the two possible outcomes are 1950 (A is executed before B), and 2000 (B is executed before A).

  However, each update consists of a read and a write, and it is dangerous to interleave the execution of these operations.

Lost Update (2)

- Without precautions, there are schedules of read and write operations in which only one of the two updates persists (**normally excluded**):

Transaction A	Transaction B
	`read(X, 'SAL ...');`
`read(Y, 'SAL ...');`	
`Y := Y * 2;`	
`write(Y, 'SAL ...');`	
	`X := X + 50;`
	`write(X, 'SAL ...');`

- So the second `write` overwrites the first, and the final salary in the database is only 1000.

Lost Update (3)

- Such "lost updates" are prevented by most DBMS.
- Oracle would acquire a write lock on the tuple for the employee "James" before it reads the value to be updated.
- So if Transaction B is the first to get this lock, then Transaction A has to wait with the update until B commits.

  This also avoids dirty reads. If we only want to avoid lost updates, B could free the lock immediately after the write.

Lost Update (4)

- However, lost updates are automatically prevented only if an `UPDATE` command is used.
- If for a more complex computation, the old value is first read with a `SELECT` command, and then the new value is written back with an `UPDATE` command, lost updates can occur.

    The problem is that a `SELECT` normally does not acquire a lock for the selected tuples (or only a short-living lock, which is removed immediately after the `SELECT`). Keeping locks on selected tuples until the end of the transaction would decrease concurrency/performance too much (and is often not necessary).

Lost Update (5)

Transaction A	Transaction B

`SELECT SAL FROM EMP`
`WHERE ENAME = 'JAMES'`
$\longrightarrow$ 950	

	`SELECT ...` $\longrightarrow$ 950

	`UPDATE EMP`
	`SET SAL = 1900`
	`WHERE ENAME = 'JAMES'`

	`COMMIT`

`UPDATE EMP`
`SET SAL = 1000`
`WHERE ENAME = 'JAMES'`	

`COMMIT`	

Lost Update (6)

- The above schedule with a lost update is possible in Oracle and other DBMS. In order to avoid it, "`FOR UPDATE`" must be added to the query (i.e. the `SELECT` result is potentially input for a later change):

```
SELECT Sal FROM Emp
WHERE EName = 'JAMES'
FOR UPDATE
```

- Only simple queries can use `FOR UPDATE`.

  The system must be able to find out which tuples should be locked. Oracle allows joins, but no aggregations, `DISTINCT`, `UNION`. In general, `FOR UPDATE` can only be used if the query would define an updatable view.

Lost Update (7)

- The FOR UPDATE clause locks the tuples which satisfied the WHERE condition at the time of the query.
- The locks are kept until the end of the transaction.
- One can also specify an attribute:

```
SELECT Sal FROM Emp
WHERE EName = 'JAMES'
FOR UPDATE OF Sal
```

- This is useful for systems which allow locking single attributes.

  In Oracle, which allows joins in the queries, it also defines from which table rows should be locked.

Nonrepeatable Read (1)

- It is possible to query the same data twice in a transaction, and get different results:

Transaction A	Transaction B

`SELECT SAL FROM EMP`
`WHERE ENAME = 'JAMES'`
$\longrightarrow$ 950	

	`UPDATE EMP`
	`SET SAL = SAL + 50`
	`WHERE ENAME = 'JAMES'`

	`COMMIT`

`SELECT SAL FROM EMP`
`WHERE ENAME = 'JAMES'`
$\longrightarrow$ 1000	

Nonrepeatable Read (2)

- This behavior is not possible in a serial execution of transactions, so it violates the isolation principle.

  The same problem is the source of the lost update for updates split into `SELECT` followed by `UPDATE` (see above).
  It is unlikely that exactly the same query would be posed twice in a transaction. But queries could access overlapping sets of tuples.

- The "Nonrepeatable Read" problem can be avoided by keeping read locks on the accessed tuples until the end of the transaction.

  Most systems unlock the tuples immediately after they were read in order to permit more parallelism. However, the "`FOR UPDATE`" clause shown above can be used to ensure that the locks are kept.

Inconsistent Analysis (1)

- E.g. suppose that the sum of all salaries is stored redundantly in another table `BUDGET(AMOUNT)`.

  This table has only one row and one column.

- Then these queries should have the same result:
  - ◇ `SELECT SUM(SAL) FROM EMP`
  - ◇ `SELECT AMOUNT FROM BUDGET`

- However, when both queries are executed one after the other (in order to check the consistency), there is no guarantee that they are evaluated with respect to the same state.

Inconsistent Analysis (2)

- In this case different tuples are accessed, so keeping locks on the tuples from the time of the first access does not help.

  Oracle guarantees that each query is evaluated with respect to only one state. So it would help to put the entire analysis into a single `SELECT` statement.

- In order to exclude this problem, one needs to lock both tables explicitly/manually (see below) before the analysis starts.

Phantom Problem (1)

- Suppose that there is a budget of $5000 for a salary raise which should be distributed evenly.

Transaction A	Transaction B

`SELECT COUNT(*)`
`FROM EMP`
$\longrightarrow$ 50	

	`INSERT INTO EMP`
	`VALUES (...)`

	`COMMIT`

`UPDATE EMP`
`SET SAL = SAL + 100`	

Phantom Problem (2)

- The new employee also received the salary raise, and \$ 5100 were spent (\$ 100 too much).
- Note that it would not help to use the query
```
SELECT * FROM EMP
FOR UPDATE OF SAL
```
  and to count the number of result rows.
- Now all existing rows of the table are locked, but the insertion of new rows is still not prevented.

  By using only locks on tuples, an `INSERT` will never be prevented (non-existing tuples cannot be locked). If the query contains a condition (e.g. `DEPTNO = 20`), also updates (setting a `DEPTNO` to 20) can cause a phantom problem.

Phantom Problem (3)

- In this case the entire table must be locked:

  `LOCK TABLE EMP IN EXCLUSIVE MODE`

- The `LOCK TABLE` command is not part of SQL-92.

  It works in Oracle and DB2. In MySQL, the syntax is different: `LOCK TABLES` T_1 `WRITE,` T_2 `READ` (this command releases all previous locks, in this way deadlocks are avoided). `LOCK TABLE` does not work in SQL Server and Access.

- Instead SQL-92 allows specifying an isolation level for a transaction.

  This is also possible in Oracle, but Oracle's isolation level `SERIALIZABLE` gives nearly no parallelism and still does not guarantee serializability.

Isolation Levels (1)

- SQL-92 (and Oracle) have a command

  `SET TRANSACTION ISOLATION LEVEL ⟨Level⟩`

- The SQL-Standard has four isolation levels:
  - ◇ `READ UNCOMMITTED`: The transaction can read the current DB state without waiting for locks.

    E.g. in order to compute the statistics for the optimizer, only an estimate is needed, and it doesn't matter whether some "dirty data" is contained in the statistics.

  - ◇ `READ COMMITTED`: The standard case.

    Read locks are only hold for the duration of the read.

Isolation Levels (2)

- Isolation levels, continued:
  - ◇ `REPEATABLE READ`: Here read locks are only removed after the transaction committed.

    This does not protect against the phantom problem (and against an inconsistent analysis spanning multiple tables).
  - ◇ `SERIALIZABLE`: The theoretical ideal of complete isolation. It excludes the phantom problem.
- Oracle supports only "`READ COMMITTED`" (which is the default) and "`SERIALIZABLE`".

"Serializable" in Oracle8

- Suppose that there are two tables `R(A)` and `S(A)`, both with only one row containing the value `'old'`.
- Then the following schedule is possible in Oracle8, even in the isolation level `SERIALIZABLE`:

Transaction A	Transaction B
`SELECT A FROM R`	
$\longrightarrow$ `old`	
`UPDATE S SET A='new'`	
	`SELECT A FROM S`
	$\longrightarrow$ `old`
	`UPDATE R SET A='new'`
	`COMMIT`
`COMMIT`	

Overview

Transactions (1)

- A transaction can be formally seen as a sequence consisting of commands of the following types:
  - ◇ `read`(x): Make a copy of object x from the DB available to the transaction.

    Objects could e.g., be table rows, or disk blocks.
  - ◇ `write`(x): Replace the current version of object x in the DB by a new version.

    Of course, `write` has also a parameter for the value to be written. For concurrency control, it is only important that x is written.
  - ◇ `rollback`: Undo all changes by this transaction.
  - ◇ `commit`: Make all changes permanent.

Transactions (2)

- Every transaction must end in `commit` or `rollback`, and these commands are only possible as last command in the sequence.
- The transaction manager inside the DBMS does not know how the new version of x computed by the transaction.
- It must assume that the new value of x potentially depends on all objects that were previously read by the transaction.

Transactions (3)

- Note that this formal model assumes that each transaction specifies explicitly which objects (tuples) it wants to read or write.
- This is a simplification of reality, because in SQL one specifies only a condition for the objects to read or update.
- For instance, the phantom problem cannot be studied in this setting.

  Of course, there are more complex formal models that have also operations for reading or writing a set of objects that specifies a condition.

Schedules (1)

- Let a set of transactions $T_1, \ldots, T_n$ be given, and for each transaction T_i, a sequence $c_{i,1} \ldots c_{i,m_i}$ of commands.
- Let $\mathcal{S}$ be the set of triples $s = (T_i, j, c_{i,j})$ with $1 \leq i \leq n$ and $1 \leq j \leq m_i$ (steps to be executed).
- The transactions define a partial order on $\mathcal{S}$: $s \prec s'$ if and only if s and s' belong to the same transaction and s comes before s' in that transaction, i.e. $s = (T_i, j, c_{i,j})$ and $s' = (T_i, k, c_{i,k})$ and $j < k$.

Schedules (2)

- A schedule (or history) of these transactions is a linear order $<$ on $\mathcal{S}$ that is compatible with $\prec$.
- I.e. a schedule defines a sequence $s_1 \ldots s_l$ of steps that contains each element of $\mathcal{S}$ exactly once, and that respects the order of steps within a transaction (if $s_i \prec s_j$, then $i < j$).
- I.e. a schedule is an interleaving of the single steps of the transactions.

Schedules (3)

- For instance, suppose that T_1 and T_2 both want to update an object `A`, so they both have the same sequence of operations `read(A)`, `write(A)`, `commit`.
- Then a schedule (that leads to a lost update) is

T_1: `read(A)`,
T_2: `read(A)`,
T_2: `write(A)`,
T_2: `commit`,
T_1: `write(A)`,
T_2: `commit`.

T_1	T_2
`read(A)`	
	`read(A)`
	`write(A)`
	`commit`
`write(A)`	
`commit`	

Schedules (4)

- Serial schedules are schedules that execute each transaction in one piece, i.e. for all steps $s < s' < s''$: if s and s'' belong to transaction T_i, then s' must also belong to transaction T_i.

T_1	T_2
read(A)	
write(A)	
commit	
	read(A)
	write(A)
	commit

T_1	T_2
	read(A)
	write(A)
	commit
read(A)	
write(A)	
commit	

Serializable Schedules (1)

- Informally, a schedule is serializable if and only if it is equivalent to a serial schedule.
- Equivalent means that
  - ◇ The read-operations in all transactions yield the same value of an object in the two schedules.
  - ◇ The two schedules do not differ in the final values of all objects in the database.
- The DBMS must ensure serializability in this sense, but since it does not know how the applications compute values, it might further restrict schedules.

Serializable Schedules (2)

- The following pairs of operations are called conflict operations
  - ◇ T_i: `write`(x) and T_j: `write`(x),
  - ◇ T_i: `write`(x) and T_j: `read`(x),
  - ◇ T_i: `abort` with T_j: `read`(x) and T_j: `write`(x) if T_i contains `write`(x).

    I.e. `abort` simply writes all objects that were modified in the transaction: It must set them back to their original value.
- Conflict operations are operations that cannot be exchanged: Between such operations, it is important, which one is executed first, and which later.

Serializable Schedules (3)

- Let $s_1 \ldots s_k$ be a schedule. A valid elementary modification of the schedule is to exchange two steps that immediately follow on each other, i.e. to transform $s_1 \ldots s_i\; s_{i+1} \ldots s_k$ into $s_1 \ldots s_{i+1}\; s_i \ldots s_k$, if s_i and s_{i+1} are not in conflict with each other.
- A schedule is called conflict-serializable if and only if it can be transformed into a serial schedule by a sequence of valid elementary modifications.

  Conflict-serializability implies the equivalence to a serial schedule. The converse is not true, e.g. two write operations might by chance write the same value.

Serializable Schedules (4)

Example/Exercise:

- Show that this schedule is conflict-serializable:

T_1	T_2
	`read(A)`
`read(A)`	
`read(B)`	
	`write(A)`
`write(B)`	

- Which transaction must be executed first in an equivalent serial schedule?

Serializable Schedules (5)

- An easy test for serializability of a schedule is to construct its conflict graph:
  - ◇ The nodes of the graph are the transactions.
  - ◇ There is an edge from transaction T_i to transaction T_j $(i \neq j)$ if and only if there are steps s of T_i and s' of T_j in the schedule with $s < s'$ and such that s and s' are in conflict with each other.
- A schedule is conflict-serializable if and only if its conflict graph contains no cycles.