Transactions in SQL

Transactions

- If nothing is specified, automatically for each individual statement
- Begin Transaction (or Start Transaction)
- followed by the statements that are part of that transaction
- Commit or Rollback

Example

```
db1=# begin transaction;
BEGIN
db1=# select * from sp;
 name | item | price
------+------+-------
 S2   | P3   |   100
 S1   | P2   |    20
 S1   | P1   |    10
 S3   | P4   |  1000
 S2   | P1   |    11
 S4   | P1   |     9
      | P1   |
 S5   |      |
 S4   | P3   |
(9 rows)
```

Update Table

```
db1=# update sp set price = price * 2;
UPDATE 9
db1=# select * from sp;
 name | item | price
------+------+-------
 S2   | P3   |   200
 S1   | P2   |    40
 S1   | P1   |    20
 S3   | P4   |  2000
 S2   | P1   |    22
 S4   | P1   |    18
      | P1   |
 S5   |      |
 S4   | P3   |
(9 rows)
```

Change your mind - rollback

```
db1=# rollback;
ROLLBACK
db1=# select * from sp;
 name | item | price
------+------+-------
 S2   | P3   |   100
 S1   | P2   |    20
 S1   | P1   |    10
 S3   | P4   |  1000
 S2   | P1   |    11
 S4   | P1   |     9
      | P1   |
 S5   |      |
 S4   | P3   |
(9 rows)
```

Savepoint

- In longer transactions, mark places to rollback to if necessary

```
db1=# begin transaction;
BEGIN
db1=# update sp set price = price * 2;
UPDATE 9
db1=# select * from sp;
 name | item | price
------+------+-------
 S2   | P3   |   200
 S1   | P2   |    40
 S1   | P1   |    20
 S3   | P4   |  2000
 S2   | P1   |    22
 S4   | P1   |    18
      | P1   |
 S5   |      |
 S4   | P3   |
(9 rows)

db1=# savepoint save1;
SAVEPOINT
```

Savepoint

- rollback to a previously saved (named) save point

```
db1=# update sp set price = price * 3;
UPDATE 9
db1=# select * from sp;
 name | item | price
------+------+-------
 S2   | P3   |   600
 S1   | P2   |   120
 S1   | P1   |    60
 S3   | P4   |  6000
 S2   | P1   |    66
 S4   | P1   |    54
      | P1   |
 S5   |      |
 S4   | P3   |
(9 rows)

db1=# rollback to savepoint save1;
ROLLBACK
```

Commit

- Commit will end a transaction keeping the final state

```
db1=# select * from sp;
 name | item | price
------+------+-------
 S2   | P3   |   200
 S1   | P2   |    40
 S1   | P1   |    20
 S3   | P4   |  2000
 S2   | P1   |    22
 S4   | P1   |    18
      | P1   |
 S5   |      |
 S4   | P3   |
(9 rows)

db1=# commit;
COMMIT
```

What to Lock?

- DB entities are nested in hierarchies, e.g.
  - Table
  - Page
  - Row
- Or a B+ Tree hierarchy
  - Root
  - Subtree at any point
  - Leaf node
  - Row/Record

General Rules

- Lock what will be accessed / modified
- Lower granularity allows higher concurrency
- Lower granularity locks may allow "phantom" problem.
  - A concurrent transaction may insert new rows and commit.
  - Next time the same data is read, it would see the newly inserted row

Isolation Levels

- DBMS and SQL in practice offer varying levels of consistency
  - Serializable
  - Repeatable Read
  - Read Committed
  - Read Uncommitted

Serializable

- The highest level of consistency
- 2 Phase locking, ensure no phantoms by either table level or B+tree based locking (to prevent addition to locked pages)

Repeatable Read

- In practice, could be same as serializable with 2 Phase locking, but don't guarantee the phantom problem
- Essentially lock every read item but no need to lock "higher" level objects to prevent phantoms

Read Committed

- Allows others transactions committed data to be visible while the transaction is in flight
    - Its own changes are not visible to others
- Short read (shared) locks that are released immediately.
- Exclusive locks are released at commit/rollback(abort).
- Good practical level for long running read-mostly transactions - e.g. data analysis where you may even want to see data that's recently added

Read Uncommitted

- Essentially equivalent to no read/shared locks
- Can read dirty (uncommitted) data from other transactions
- Exclusive locks are acquired and kept till commit
- Not all systems support this (e.g. Postgres, Teradata).