Transaction

Standard form

```
BEGIN;-- or BEGIN TRANSACTION
UPDATE accounts SET balance = balance - 100.00
 WHERE name = 'zhangsan';
-- etc etc
COMMIT;--or COMMIT TRANSACTION or ROLLBACK
```

PostgreSQL actually treats every SQL statement as being executed within a transaction. If you do not issue a `BEGIN` command, then each individual statement has an implicit `BEGIN` and (if successful) `COMMIT` wrapped around it. A group of statements surrounded by `BEGIN` and `COMMIT` is sometimes called a *transaction block*.

Notice: Some client libraries issue `BEGIN` and `COMMIT` commands automatically, so that you might get the effect of transaction blocks without asking. Check the documentation for the interface you are using.

we use follow table

```
create table emp(
    id integer primary key ,
    name varchar(30),
    age integer,
    dept_code integer,
    office_loc varchar(100),
    salary integer default 0
);

insert into emp values (1,'张三',22,1,'宝安',6000),
                       (2,'李四',31,1,'宝安',7000),
                       (3,'王五',25,2,'福田',6000),
                       (4,'赵六',24,1,'宝安',5000),
                       (5,'庄七',22,3,'光明',8000),
                       (6,'康八',45,2,'福田',15000),
                       (7,'聂九',34,3,'光明',7500),
                       (8,'刘二麻子',56,4,'光明',17000),
                       (9,'孙小毛',17,1,'宝安',3000),
                       (10,'陈老大',37,1,'宝安',7000);
```

Feature ACID (Atomicity, Consistency, Isolation, Durability)

- **Atomicity**: In a transaction involving two or more discrete pieces of information, either **all** of the pieces are committed or **none** are.
- **Consistency**: A transaction either creates a new and valid state of data, or, if any failure occurs, returns all data to its state before the transaction was started.

- **Isolation**: A transaction in process and not yet committed must remain isolated from any other transaction.

  The SQL standard defines four levels of transaction isolation:

  `dirty read`: A transaction reads data written by a concurrent uncommitted transaction.

  `nonrepeatable read`: A transaction re-reads data it has previously read and finds that data has been modified by another transaction (that committed since the initial read).

  `phantom read`: A transaction re-executes a query returning a set of rows that satisfy a search condition and finds that the set of rows satisfying the condition has changed due to another recently-committed transaction.

  `serialization anomaly`: The result of successfully committing a group of transactions is inconsistent with all possible orderings of running those transactions one at a time.

  You can set above level in Postgres using **SET TRANSACTION**

  https://www.postgresql.org/docs/12/sql-set-transaction.html

  ```
  SET TRANSACTION transaction_mode [, ...]
  SET TRANSACTION SNAPSHOT snapshot_id
  SET SESSION CHARACTERISTICS AS TRANSACTION transaction_mode [, ...]

  where transaction_mode is one of:

      ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ | READ COMMITTED | READ
  UNCOMMITTED }
      READ WRITE | READ ONLY
      [ NOT ] DEFERRABLE
  ```

- **Durability**: Committed data is saved by the system such that, even in the event of a failure and system restart, the data is available in its correct state.

Experiment 1: Atomicity

```
BEGIN;
DELETE from emp where age <17;
select * from emp where age <17;
COMMIT ;
select * from emp where age <17;

BEGIN;
DELETE from emp where name = '张三';
select * from emp where name = '张三';
ROLLBACK;
select * from emp where name = '张三';

BEGIN TRANSACTION ;
UPDATE emp SET salary=salary+1000 where id=5;
SAVEPOINT sp01;
UPDATE emp SET salary=salary-1000 where id=6;
select * from emp;
ROLLBACK TO sp01;
select * from emp;
UPDATE emp SET salary=salary-1000 where id=7;
```

```sql
select * from emp;
COMMIT ;
select * from emp;
```

Experiment 2: Consistency

```sql
select * from emp;

BEGIN TRANSACTION ;
UPDATE emp SET salary=salary+1000 where id=5;
SAVEPOINT sp01;
UPDATE emp SET salary=salary-1000 where id=6;
select * from emp;
ROLLBACK TO sp01;
select * from emp;

-- close this console, and check the table in database
select * from emp;
```

Experiment 3: Isolation

Step 1: Open two console

Step 2: Execute SQL as following order, and check the table emp

order	Console 1	Console 2
1	BEGIN TRANSACTION ;	BEGIN TRANSACTION ;
2	UPDATE emp SET salary=salary+1000 where id=5;	
3	select * from emp where id=5;	select * from emp where id=5;
4		UPDATE emp SET salary=salary-1000 where id=6;
5	COMMIT;	ROLLBACK;
6	select * from emp;	select * from emp;
7		COMMIT;
8	select * from emp;	select * from emp;

Isolation Level

Isolation Level	Dirty Read	Nonrepeatable Read	Phantom Read	Serialization Anomaly
Read uncommitted	Allowed, but not in PG	Possible	Possible	Possible
Read committed	Not possible	Possible	Possible	Possible
Repeatable read	Not possible	Not possible	Allowed, but not in PG	Possible
Serializable	Not possible	Not possible	Not possible	Not possible

In PostgreSQL, you can request any of the four standard transaction isolation levels, but internally only three distinct isolation levels are implemented, i.e., PostgreSQL's Read Uncommitted mode behaves like Read Committed. This is because it is the only sensible way to map the standard isolation levels to PostgreSQL's multi-version concurrency control architecture.

- Read Committed Isolation Level

  Read Committed is the **default** isolation level in PostgreSQL.. When a transaction uses this isolation level, a `SELECT` query (without a `FOR UPDATE/SHARE` clause) sees only **data committed before the query began**; it never sees either uncommitted data or changes committed during query execution by concurrent transactions. In effect, a `SELECT` query sees a snapshot of the database as of the instant the query begins to run. However, `SELECT` does see the effects of previous updates executed within its own transaction, even though they are not yet committed. Also note that two successive `SELECT` commands can see different data, even though they are within a single transaction, if other transactions commit changes after the first `SELECT` starts and before the second `SELECT` starts.

- Repeatable Read Isolation level

  The *Repeatable Read* isolation level only sees **data committed before the transaction began**; it never sees either uncommitted data or changes committed during transaction execution by concurrent transactions.

  This level is different from Read Committed in that a query in a repeatable read transaction sees a snapshot as of the start of the first non-transaction-control statement in the *transaction*, not as of the start of the current statement within the transaction. Thus, successive `SELECT` commands within a *single* transaction see the same data, i.e., they do not see changes made by other transactions that committed after their own transaction started.

  Applications using this level must be prepared to retry transactions due to serialization failures.

  Maybe...

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

  because a repeatable read transaction cannot modify or lock rows changed by other transactions after the repeatable read transaction began.

When an application receives this error message, it should abort the current transaction and retry the whole transaction from the beginning. The second time through, the transaction will see the previously-committed change as part of its initial view of the database, so there is no logical conflict in using the new version of the row as the starting point for the new transaction's update.

Note that only updating transactions might need to be retried; read-only transactions will never have serialization conflicts.

- Serializable Isolation Level

  The *Serializable* isolation level provides the strictest transaction isolation. This level emulates serial transaction execution for all committed transactions; as if **transactions had been executed one after another**, serially, rather than concurrently. However, like the Repeatable Read level, applications using this level must be prepared to retry transactions due to serialization failures. In fact, this isolation level works exactly the same as Repeatable Read except that it monitors for conditions which could make execution of a concurrent set of serializable transactions behave in a manner inconsistent with all possible serial (one at a time) executions of those transactions. This monitoring does not introduce any blocking beyond that present in repeatable read, but there is some overhead to the monitoring, and detection of the conditions which could cause a *serialization anomaly* will trigger a *serialization failure*.

```
ERROR:  could not serialize access due to read/write dependencies among
transactions
```

Experiment 4: Isolation

Step 1: Open two console, set isolation level as default

Step 2: Execute SQL as following order, and check the query result of ①~⑩

order	Console 1	Console 2
1	BEGIN TRANSACTION ;	BEGIN TRANSACTION ;
2	UPDATE emp SET salary=salary+1000 where id=5;	
3	①select salary from emp where id=5;	②select salary from emp where id=5;
4		UPDATE emp SET salary=salary-1000 where id=6;
5	③select salary from emp where id=6;	④select salary from emp where id=6;
6	COMMIT;	ROLLBACK;
7	⑤select salary from emp where id=5;	⑥select salary from emp where id=5;
8	⑦select salary from emp where id=6;	⑧select salary from emp where id=6;
9		COMMIT;
10	⑨select salary from emp where id=6;	⑩select salary from emp where id=5;