# ACID

## What is a Transaction?

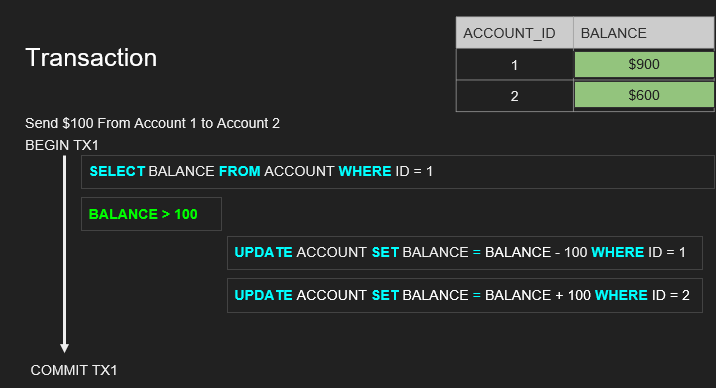
* A collection of queries
* One unit of work
* E.g. Account deposit (SELECT, UPDATE, UPDATE)

**Transaction Lifespan**

* Transaction BEGIN
* Transaction COMMIT
* Transaction ROLLBACK
* Transaction unexpected ending = ROLLBACK (e.g. crash)

**Nature of Transactions**

* Usually Transactions are used to change and modify data
* However, it is perfectly normal to have a read only transaction
* Example, you want to generate a report and you want to get consistent snapshot based at the time of transaction
* We will learn more about this in the Isolation section



## Atomicity

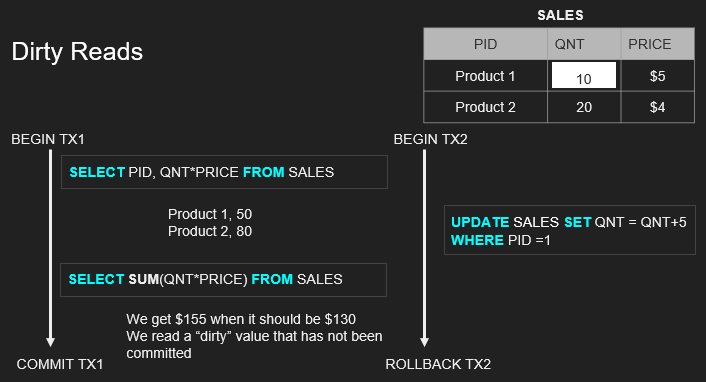
* All queries in a transaction must succeed.
* If one query fails, all prior successful queries in the transaction should rollback.
* If the database went down prior to a commit of a transaction, all the successful queries in the transactions should rollback
* An atomic transaction is a transaction that will rollback all queries if one or more queries failed.

## Isolation

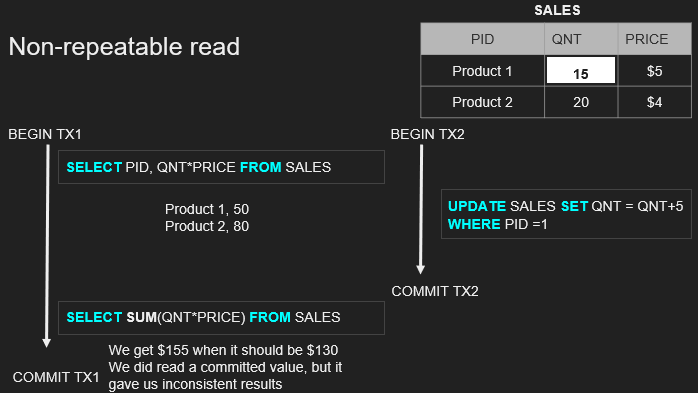
* Can my inflight transaction see changes made by other transactions?

### Read phenomena

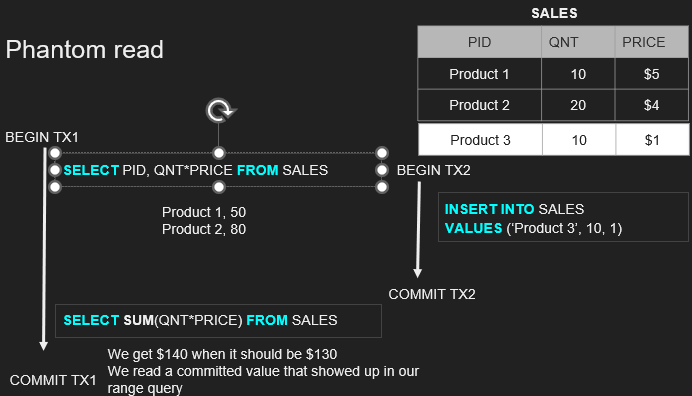
* + Dirty reads – you read something that some other transaction wrote but didn’t commit.



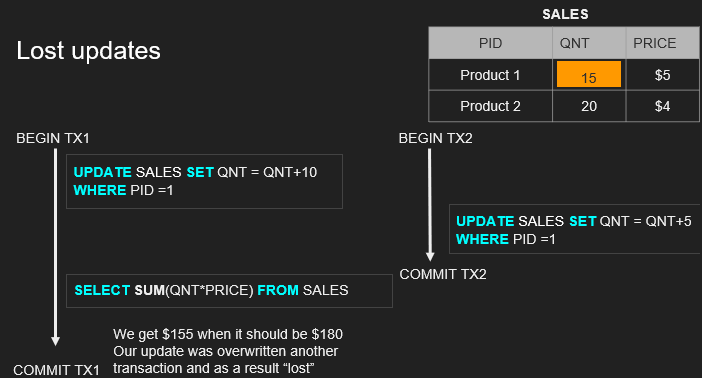
* + Non-repeatable reads – occurs when a transaction reads the same row twice, but gets different data each time.
    - Might occur when read locks are not acquired when performing read operation.



* + Phantom reads – can occur when 2 identical read operations are performed, but 2 different set of results are returned because an update has occurred on the data between the read operations.
    - Eg. Calculating sum after the other transaction inserts the new row.



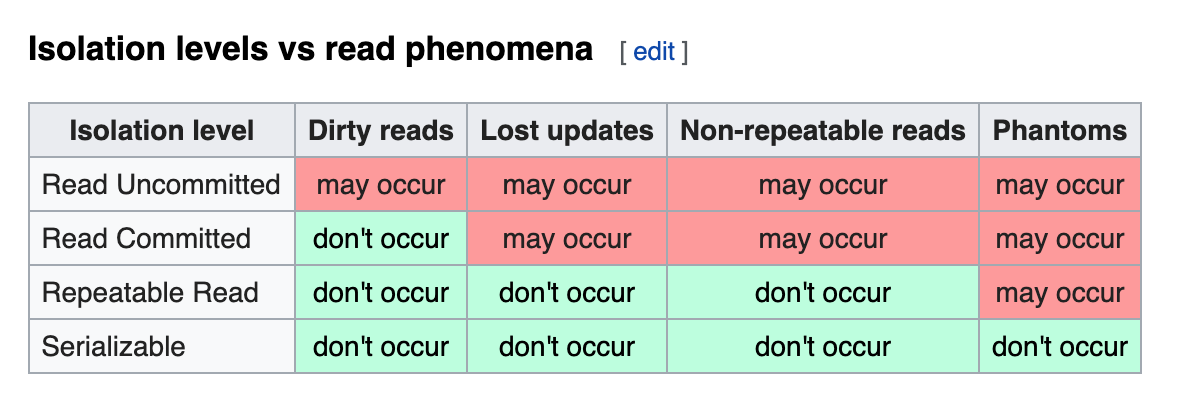
* + Lost updates –



### Isolation Levels

**Isolation - Isolation Levels for inflight transactions**

* **Read uncommitted -** No Isolation, any change from the outside is visible to the transaction, committed or not.
* fast
* **Read committed -** Each query in a transaction only sees committed changes by other transactions
* Default for many
* **Repeatable Read -** The transaction will make sure that when a query reads a row, that row will remain unchanged while its running.
* Doesn’t get rid of phantom
* **Snapshot -** Each query in a transaction only sees changes that have been committed up to the start of the transaction. It's like a snapshot version of the database at that moment.
* **Serializable -** Transactions are run as if they serialized one after the other.
* No concurrency at all, slowest



**Database Implementation of Isolation**

* Each DBMS implements Isolation level differently
* Pessimistic - Row level locks, table locks, page locks to avoid lost updates
* Optimistic - No locks, just track if things changed and fail the transaction if so
* Repeatable read “locks” the rows it reads but it could be expensive if you read a lot of rows, postgres implements RR as snapshot. That is why you don’t get phantom reads with postgres in repeatable read
* Serializable are usually implemented with optimistic concurrency control, you can implement it pessimistically with SELECT FOR UPDATE

## Consistency

### Consistency in Data

* Defined by the user
* Referential integrity (foreign keys)
* Atomicity
* Isolation

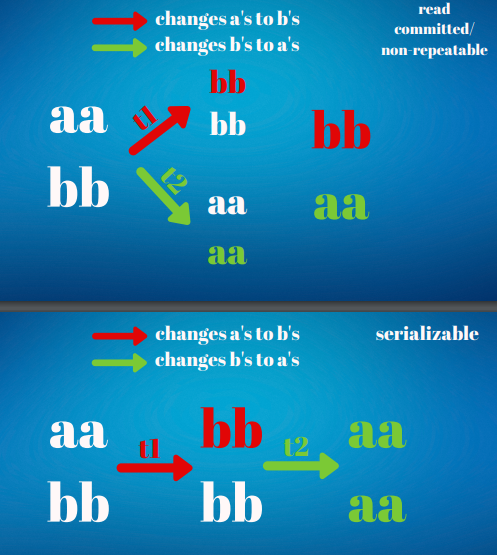
### Consistency in reads

* If a transaction committed a change will a new transaction immediately see the change?
* Affects the system as a whole
* Relational and NoSQL databases suffer from this
* Eventual consistency

## Durability

* Changes made by committed transactions must be persisted in a durable non-volatile storage.
* Durability techniques
  + WAL - Write ahead log
    - Writing a lot of data to disk is expensive (indexes, data files, columns, rows, etc..)
    - That is why DBMSs persist a compressed version of the changes known as WAL (write-ahead-log segments)
  + Asynchronous snapshot (in the background)
  + AOF – append only file, similar to WAL
* Durability - OS Cache
  + A write request in OS usually goes to the OS cache
  + When the writes go the OS cache, an OS crash, machine restart could lead to loss of data
  + Fsync OS command forces writes to always go to disk
  + fsync can be expensive and slows down commits

## Serializable vs Repeatable reads



Notes: -

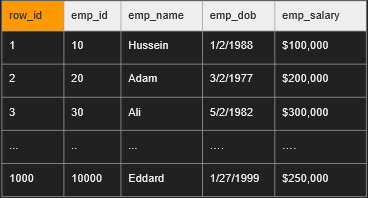
* If statement is not executed in a transaction, database will wrap it in its own transaction and commit immediately. Calling rollback won’t do anything (nothing to rollback)
* Transaction always sees the changes it makes regardless of isolation levels. Isolation level only applies to other concurrent transactions.

# Database Internals

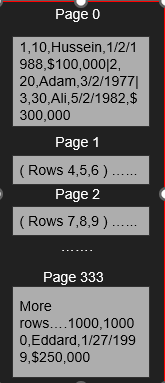
## How tables and indexes are stored on disk And how they are queried

**Storage concepts**

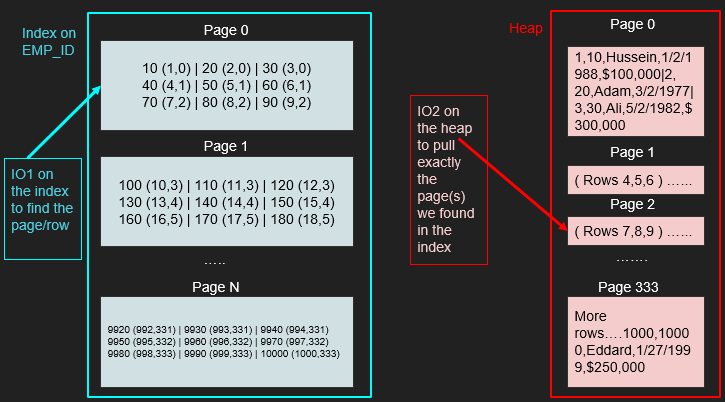
* **Table**
* **Row\_id**
  + Internal and system maintained
  + In certain databases (mysql -innoDB) it is the same as the primary key but other databases like Postgres have a system column row\_id (tuple\_id)

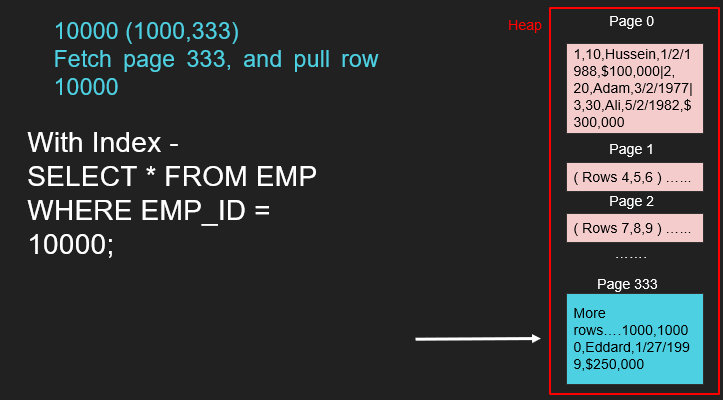


* **Page**
* Depending on the storage model (row vs column store), the rows are stored and read in logical pages.
* The database doesn’t read a single row, it reads a page or more in a single IO and we get a lot of rows in that IO.
* Each page has a size (e.g. 8KB in postgres, 16KB in MySQL)
* Assume each page holds 3 rows in this example, with 1001 rows you will have 1001/3 = 333~ pages

****

* **IO**
  + IO operation (input/output) is a read request to the disk
  + We try to minimize this as much as possible
  + An IO can fetch 1 page or more depending on the disk partitions and other factors
  + An IO cannot read a single row, its a page with many rows in them, you get them for free.
  + You want to minimize the number of IOs as they are expensive.
  + Some IOs in operating systems goes to the operating system cache and not disk
* **Heap data structure** 
  + IO operation (input/output) is a read request to the disk
  + We try to minimize this as much as possible
  + An IO can fetch 1 page or more depending on the disk partitions and other factors
  + An IO cannot read a single row, its a page with many rows in them, you get them for free.
  + You want to minimize the number of IOs as they are expensive.
  + Some IOs in operating systems goes to the operating system cache and not disk
* **Index data structure b-tree**
  + An index is another data structure separate from the heap that has “pointers” to the heap
  + It has part of the data and used to quickly search for something
  + You can index on one column or more.
  + Once you find a value of the index, you go to the heap to fetch more information where everything is there
  + Index tells you EXACTLY which page to fetch in the heap instead of taking the hit to scan every page in the heap
  + The index is also stored as pages and cost IO to pull the entries of the index.
  + The smaller the index, the more it can fit in memory the faster the search
  + Popular data structure for index is b-trees, learn more on that in the b-tree section



* **Example of a query**
* ****

**Note:-**

* Sometimes the heap table can be organized around a single index. This is called a clustered index or an Index Organized Table.
* Primary key is usually a clustered index unless otherwise specified.
* MySQL InnoDB always have a primary key (clustered index) other indexes point to the primary key “value”
* Postgres only have secondary indexes and all indexes point directly to the row\_id which lives in the heap.

## Row-Based vs Column-Based Databases

Queries

* No indexes
* Select first\_name from emp where ssn = 666
* Select \* from emp where id = 1
* Select sum(salary) from emp

## Row-Oriented Database

* Tables are stored as rows in disk
* A single block io read to the table fetches multiple rows with all their columns.
* More IOs are required to find a particular row in a table scan but once you find the row you get all columns for that row.

## Column-Oriented Database

* Tables are stored as columns first in disk
* A single block io read to the table fetches multiple columns with all matching rows
* Less IOs are required to get more values of a given column. But working with multiple columns require more IOs.
* OLAP

## Pros & Cons

|  |  |
| --- | --- |
| **Row-Based**   * Optimal for read/writes * OLTP * Compression isn’t efficient * Aggregation isn’t efficient * Efficient queries w/multi-columns | **Column-Based**   * Writes are slower * OLAP * Compress greatly * Amazing for aggregation * Inefficient queries w/multi-columns |

# Database Indexing

## Create Postgres Table with a million Rows

docker run -e POSTGRES\_PASSWORD=postgres --name pg postgres

pg 🡪 docker container name

postgres 🡪 user name

shatadb 🡪

docker exec -it pg psql -U postgres create table temp (t int);

insert into temp (t) select random()\*100 from generate\_series(0,100000)

Creating Index

Create index employees\_name\_index on employees(name);

Bitmap index scan vs Index scan vs Heap scan vs Table Scan