# Section 7:
# Transactions, Basic Authorization

**Fanavaran Anisa**
**Iran Linux House**
Linux & Open Source Training Center

www.anisa.co.ir

# Outline

- Views
- Integrity Constraints
- Domain vs User Defined Type
- Authorization

# VIEWS

# Views

- In some cases, it is not desirable for all users to see the entire logical model (that is, all the actual relations stored in the database.)
- Consider a person who needs to know an instructors name and department, but not the salary. This person should see a relation described, in SQL, by

  **select** *ID*, *name*, *dept_name*
  **from** *instructor*

- A **view** provides a mechanism to hide certain data from the view of certain users.
- Any relation that is not of the conceptual model but is made visible to a user as a "virtual relation" is called a **view**.

# View Definition

- A view is defined using the **create view** statement which has the form

  **create view** *v* **as** < query expression >

  where <query expression> is any legal SQL expression. The view name is represented by *v.*
- Once a view is defined, the view name can be used to refer to the virtual relation that the view generates.
- View definition is not the same as creating a new relation by evaluating the query expression
  - Rather, a view definition causes the saving of an expression; the expression is substituted into queries using the view.

# View Definition and Use

- A view of instructors without their salary

**create view** ***faculty*** **as**
  **select** *ID*, *name*, *dept_name*
  **from** *instructor*

- Find all instructors in the Biology department

**select** *name*
**from** ***faculty***
**where** *dept_name* = 'Biology'

- Create a view of department salary totals

**create view** ***departments_total_salary(dept_name, total_salary)*** **as**
  **select** *dept_name*, **sum** (*salary*)
  **from** *instructor*
  **group by** *dept_name*;

# Views Defined Using Other Views

- **create view** ***physics_fall_2017*** **as**
  **select** *course*.*course_id*, *sec_id*, *building*, *room_number*
  **from** *course*, *section*
  **where** *course*.*course_id* = *section*.*course_id*
  **and** *course*.*dept_name* = 'Physics'
  **and** *section*.*semester* = 'Fall'
  **and** *section*.*year* = '2017';

- **create view** ***physics_fall_2017****_watson* **as**
  **select** *course_id*, *room_number*
  **from** ***physics_fall_2017***
  **where** *building*= 'Watson';

# View Expansion

- Expand the view :

**create view** ***physics_fall_2017_watson*** **as**
  **select** *course_id*, *room_number*
  **from** ***physics_fall_2017***
  **where** *building*= 'Watson'

- To:

**create view** ***physics_fall_2017_watson*** **as**
  **select** *course_id*, *room_number*
  **from** (**select** *course*.*course_id*, *building*, *room_number*
    **from** *course*, *section*
    **where** *course*.*course_id* = *section*.*course_id*
      **and** *course*.*dept_name* = 'Physics'
      **and** *section*.*semester* = 'Fall'
      **and** *section*.*year* = '2017')
  **where** *building*= 'Watson';

# Update of a View

- Add a new tuple to *faculty* view which we defined earlier

  **insert into** *faculty*

  **values** ('30765', 'Green', 'Music');

- This insertion must be represented by the insertion into the *instructor* relation
  - Must have a value for salary.
- Two approaches
  - Reject the insert
  - Insert the tuple

    ('30765', 'Green', 'Music', null)

  into the *instructor* relation

# Some Updates Cannot be Translated Uniquely

- **create view** *instructor_info* **as**
  **select** *ID*, *name*, *building*
  **from** *instructor*, *department*
  **where** *instructor*.*dept_name* = *department*.*dept_name*;
- **insert into** *instructor_info*

  **values** ('69987', 'White', 'Taylor');
- Issues
  - Which department, if multiple departments in Taylor?
  - What if no department is in Taylor?

# And Some Not at All

- **create view** *history_instructors* **as**
  **select** *
  **from** *instructor*
  **where** *dept_name*= 'History';
- What happens if we insert

  ('25566', 'Brown', 'Biology', 100000)

  into *history_instructors?*

# View Updates in SQL

- Most SQL implementations allow updates only on simple views
  - The **from** clause has only one database relation.
  - The **select** clause contains only attribute names of the relation, and does not have any expressions, aggregates, or **distinct** specification.
  - Any attribute not listed in the **select** clause can be set to null
  - The query does not have a **group** by or **having** clause.

# Materialized Views

- Certain database systems allow view relations to be physically stored.
  - Physical copy created when the view is defined.
  - Such views are called **Materialized view**:
- If relations used in the query are updated, the materialized view result becomes out of date
  - Need to **maintain** the view, by updating the view whenever the underlying relations are updated.

```
CREATE MATERIALIZED VIEW view_name
AS
query
WITH [NO] DATA;
```

**Refreshing data for materialized views**

```
REFRESH MATERIALIZED VIEW view_name;
```

# Materialized Views

When you refresh data for a materialized view, PostgreSQL locks the entire table therefore you cannot query data against it. To avoid this, you can use the CONCURRENTLY option.

**REFRESH MATERIALIZED VIEW CONCURRENTLY view_name;**

With CONCURRENTLY option, PostgreSQL creates a temporary updated version of the materialized view, compares two versions, and performs INSERT and UPDATE only the differences.

to refresh it with CONCURRENTLY option, you need to create a UNIQUE index for the view first.

# Materialized Views

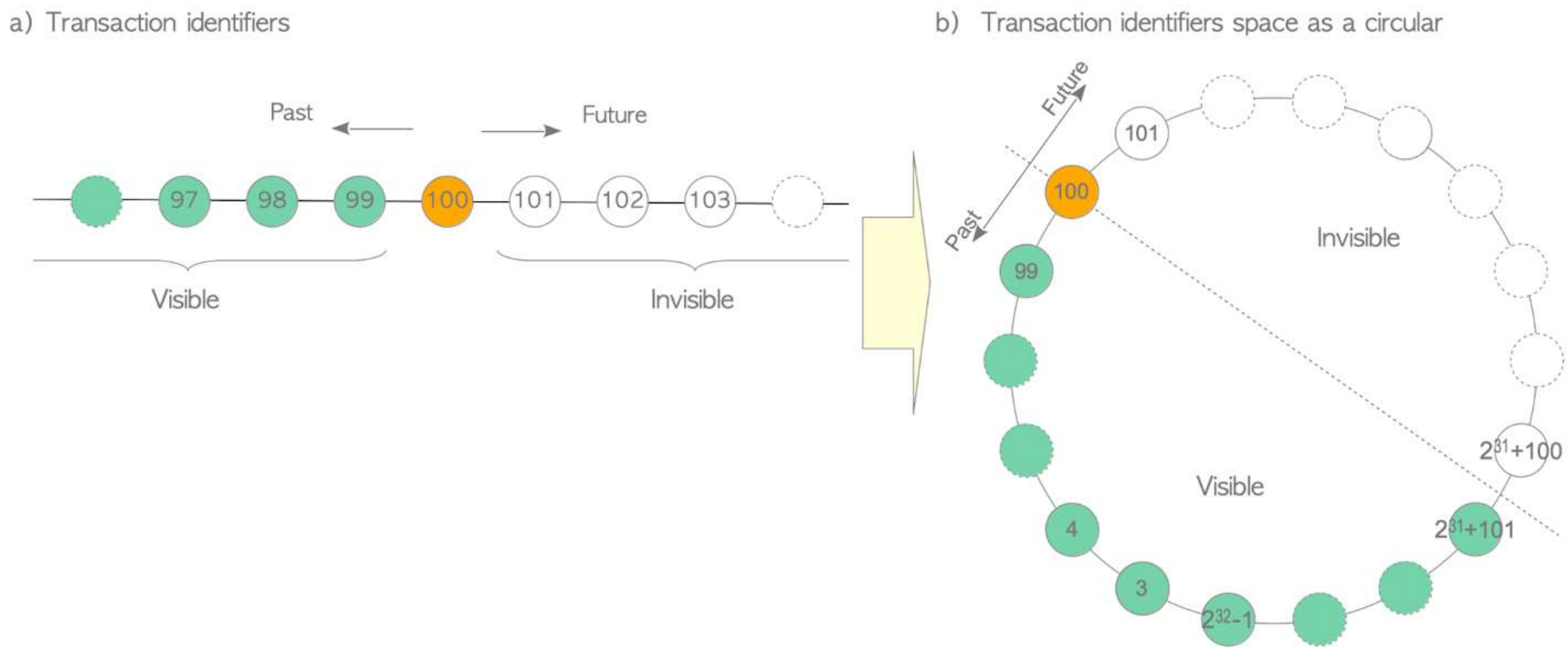
```
CREATE MATERIALIZED VIEW rental_by_category
AS
 SELECT c.name AS category,
    sum(p.amount) AS total_sales
   FROM (((((payment p
     JOIN rental r ON ((p.rental_id = r.rental_id)))
     JOIN inventory i ON ((r.inventory_id = i.inventory_id)))
     JOIN film f ON ((i.film_id = f.film_id)))
     JOIN film_category fc ON ((f.film_id = fc.film_id)))
     JOIN category c ON ((fc.category_id = c.category_id)))
  GROUP BY c.name
  ORDER BY sum(p.amount) DESC
WITH DATA;
```

# TRANSACTIONS

# Transactions

- A **transaction** consists of a sequence of query and/or update statements and is a "unit" of work
- The SQL standard specifies that a transaction begins implicitly when an SQL statement is executed.
- The transaction must end with one of the following statements:
  - **Commit work**. The updates performed by the transaction become permanent in the database.
  - **Rollback work**. All the updates performed by the SQL statements in the transaction are undone.
- Atomic transaction
  - either fully executed or rolled back as if it never occurred
- Isolation from concurrent transactions

# Transaction Space



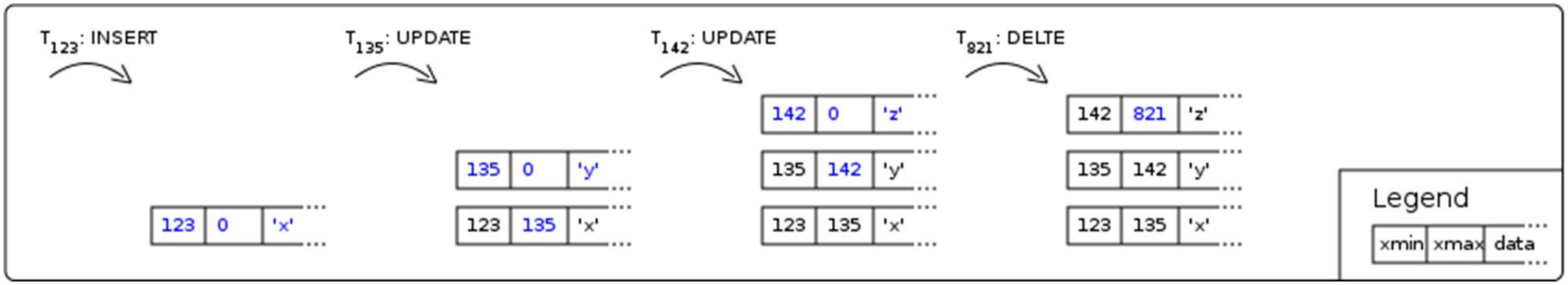
# ACID Properties

Transactions have the following four standard properties, **usually referred to by the acronym ACID**

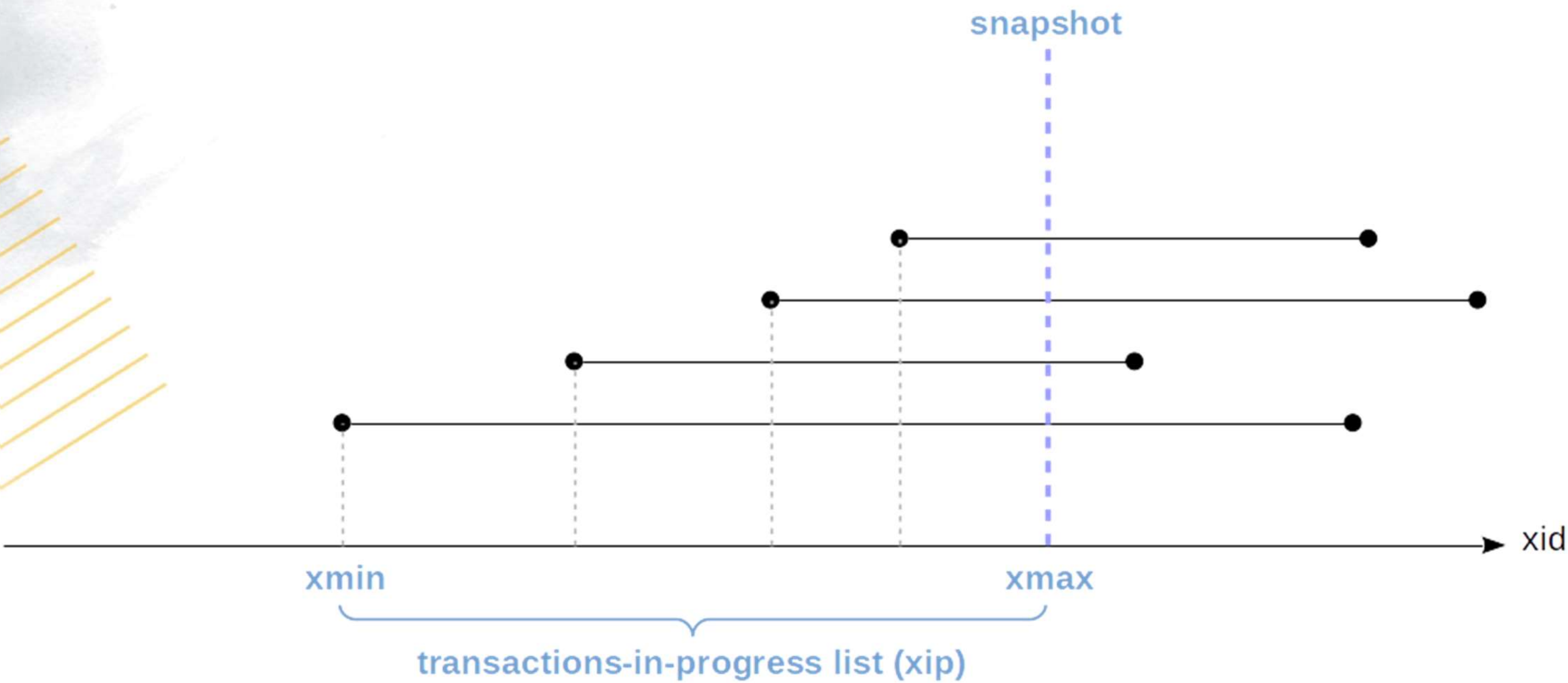
- **Atomicity** − Ensures that all operations within the work unit are completed successfully; otherwise, the transaction is aborted at the point of failure and previous operations are rolled back to their former state.
- **Consistency** − Ensures that the database properly changes states upon a successfully committed transaction.
- **Isolation** − Enables transactions to operate independently of and transparent to each other.
- **Durability** − Ensures that the result or effect of a committed transaction persists in case of a system failure.

# MVCC

- data consistency is maintained by using a multiversion model (**Multiversion Concurrency Control, MVCC**)
- each SQL statement sees a snapshot of data (a database version)
- This prevents statements from viewing inconsistent data produced by **concurrent transactions** performing updates on the same data rows, providing **transaction isolation** for each database session

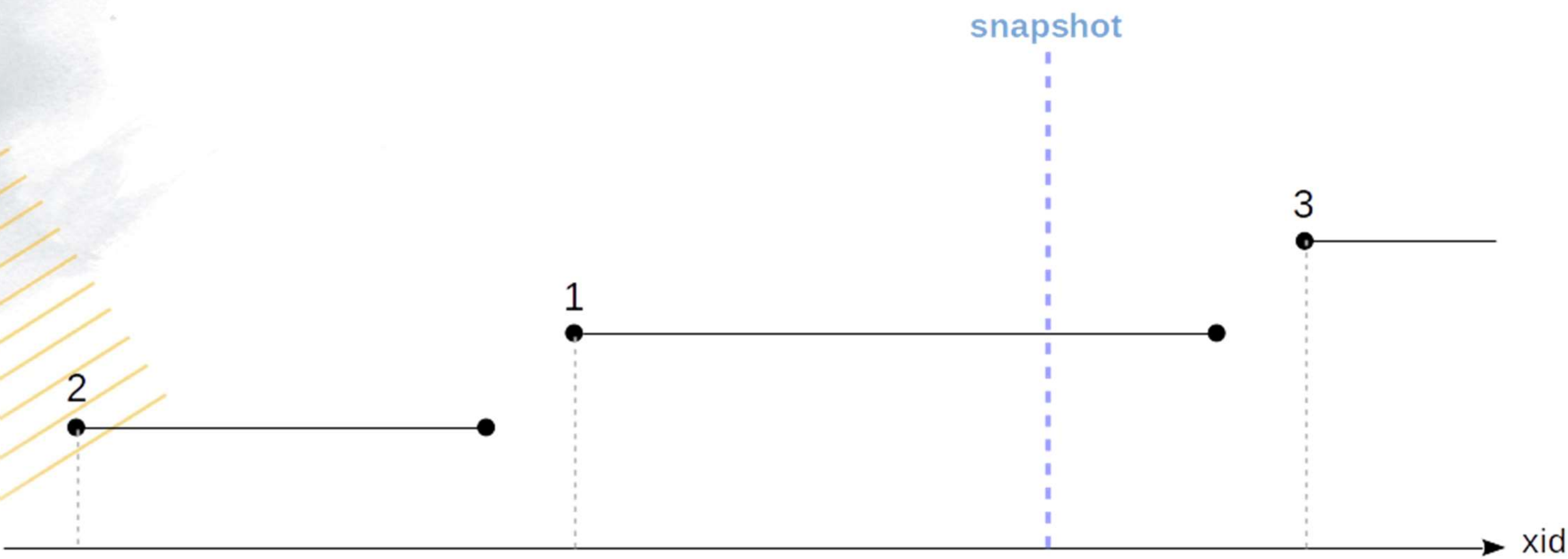


# MVCC relies on Snapshots




This information is available in the shared memory of the server, in the ProcArray structure, which contains the list of all active sessions and their transactions.

# MVCC relies on Snapshots




- Changes of the transaction 2 will be visible since it was completed before the snapshot was created.
- Changes of the transaction 1 will not be visible since it was active at the moment the snapshot was created.
- Changes of the transaction 3 will not be visible since it started after the snapshot was created (regardless of whether it was completed or not).

# MVCC : XMIN/XMAX

– insert

```
 xmin | xmax | val
------+------+-----
 5409 |    0 |   1
```

– delete

```
 xmin | xmax | val
------+------+-----
 5411 | 5412 |   1
```

– update

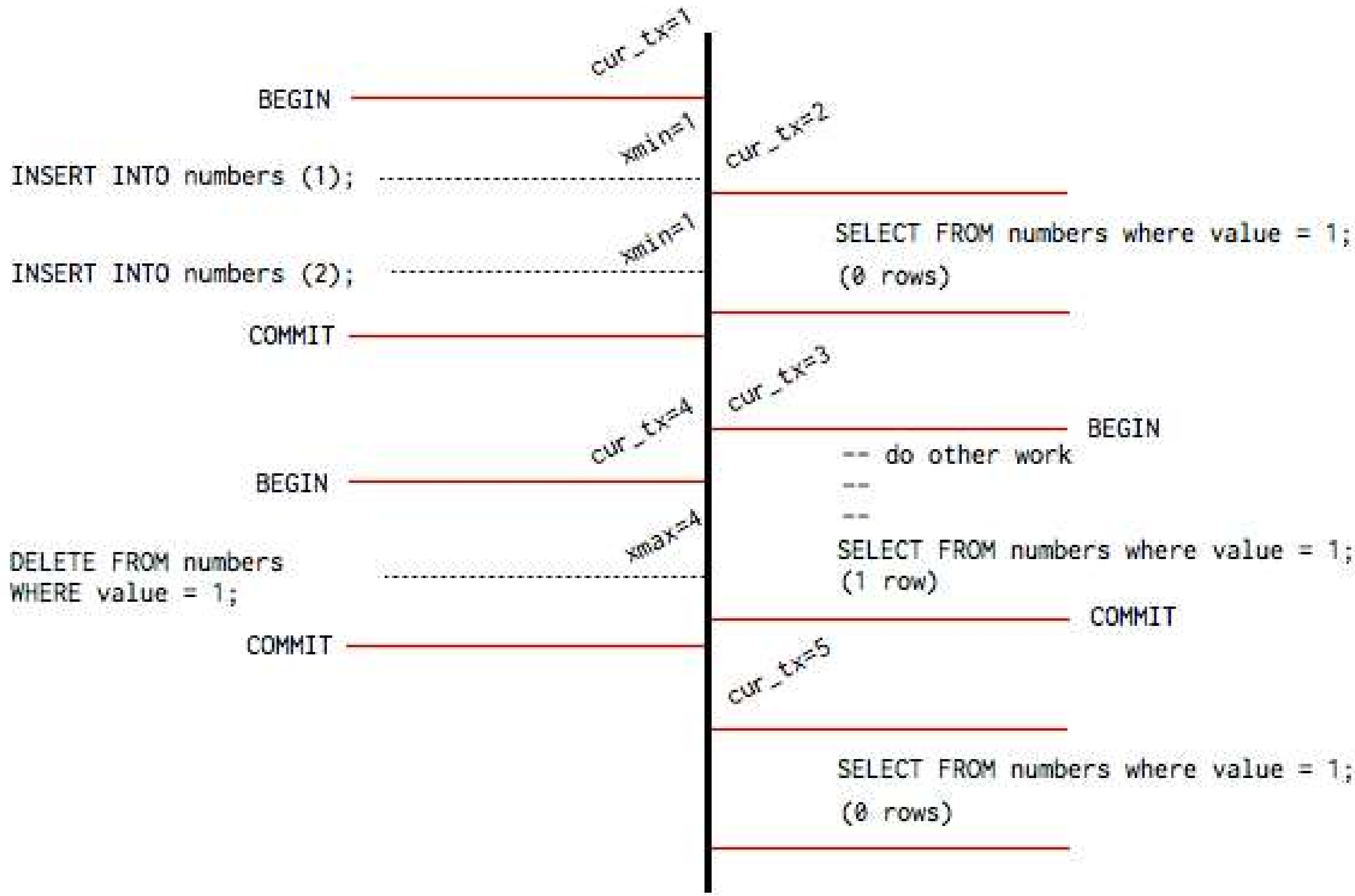
```
 xmin | xmax | val
------+------+-----
 5414 |    0 |   2
```

```
 xmin | xmax | val
------+------+-----
 5413 | 5414 |   1
```

•**xmin** - xid of the **transaction that created** the record
•**xmax** - xid of the **transaction that deleted** the record

• `xmin` and `xmax` indicate the range in which row versions are visible for transactions. This range doesn't imply any direct temporal meaning. The sequence of XIDs reflects only the sequence of transactions' begin events.

# MVCC : XMIN/XMAX



# MVCC in Action

PostgreSQL

```
UPDATE movies
   SET year = 1983
 WHERE name = 'Shaolin and Wu Tang'
```

Original Version

| id | name | year | director |
|---|---|---|---|
| 101 | Executioners from Shaolin | 1977 | [redacted] |
| 102 | Shaolin and Wu Tang | 1985 | Chia-Hui Liu |
| 103 | Five Deadly Venoms | 1978 | Cheh Chang |

**Table Page #1**

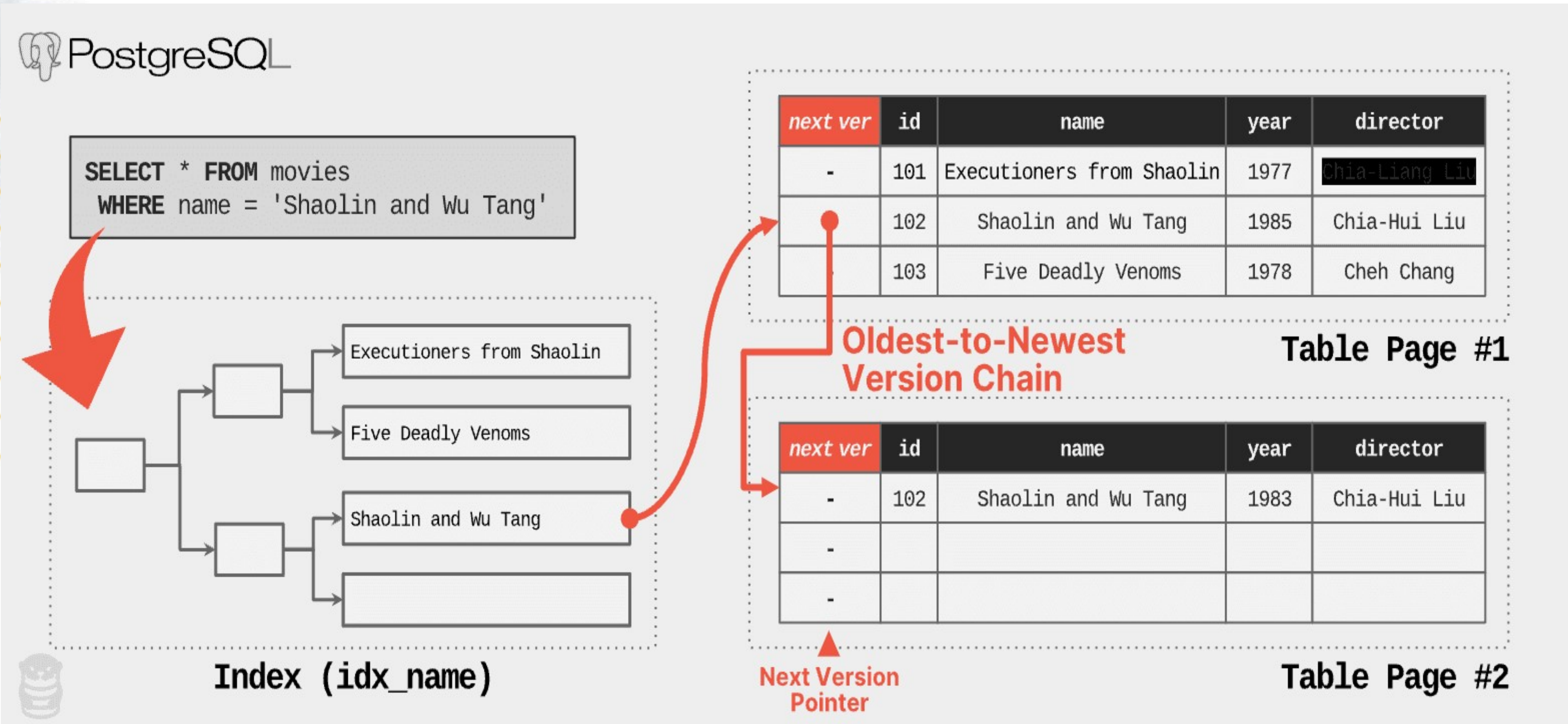
**Copy Original Version to New Version**

New Version

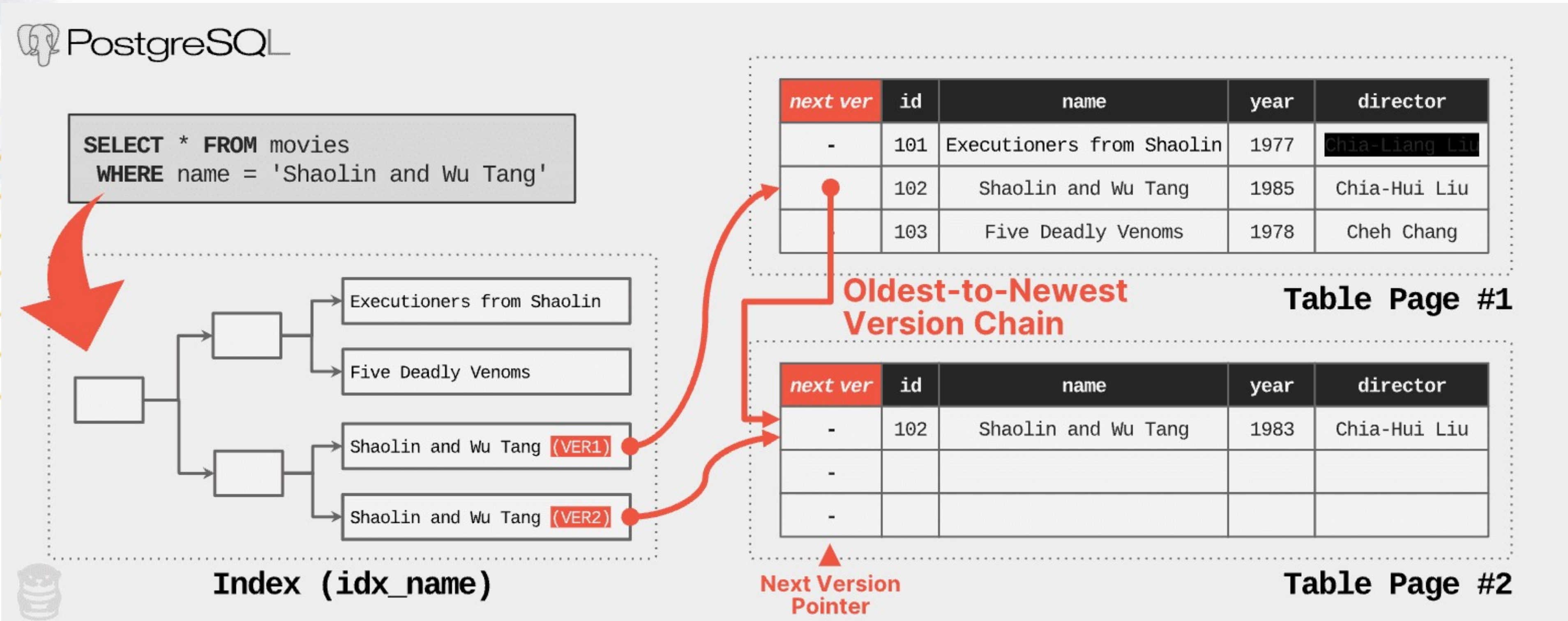
| id | name | year | director |
|---|---|---|---|
| 102 | Shaolin and Wu Tang | 1983 | Chia-Hui Liu |
| | | | |
| | | | |

**Table Page #2**

# Version Chain



# Version Chain – Long Chain Traversal -> Index



# Version Chain – Long Chain Traversal -> Index

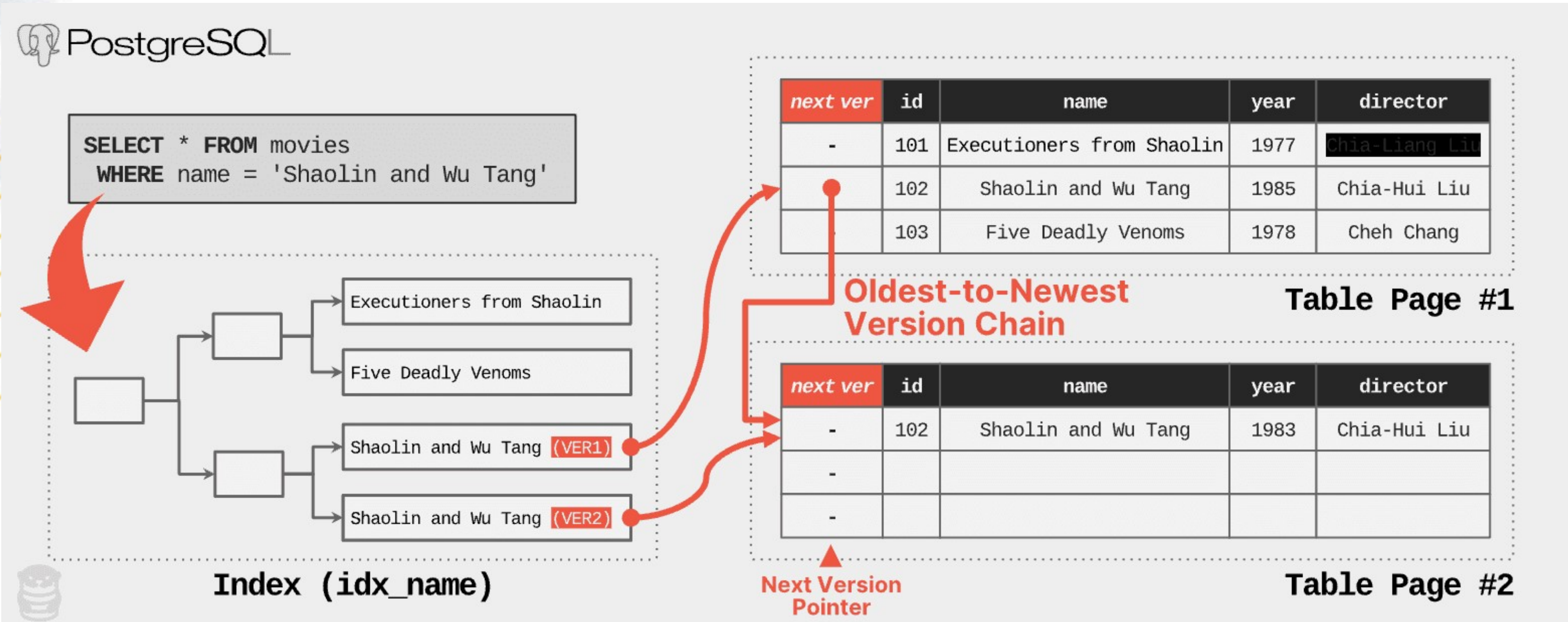


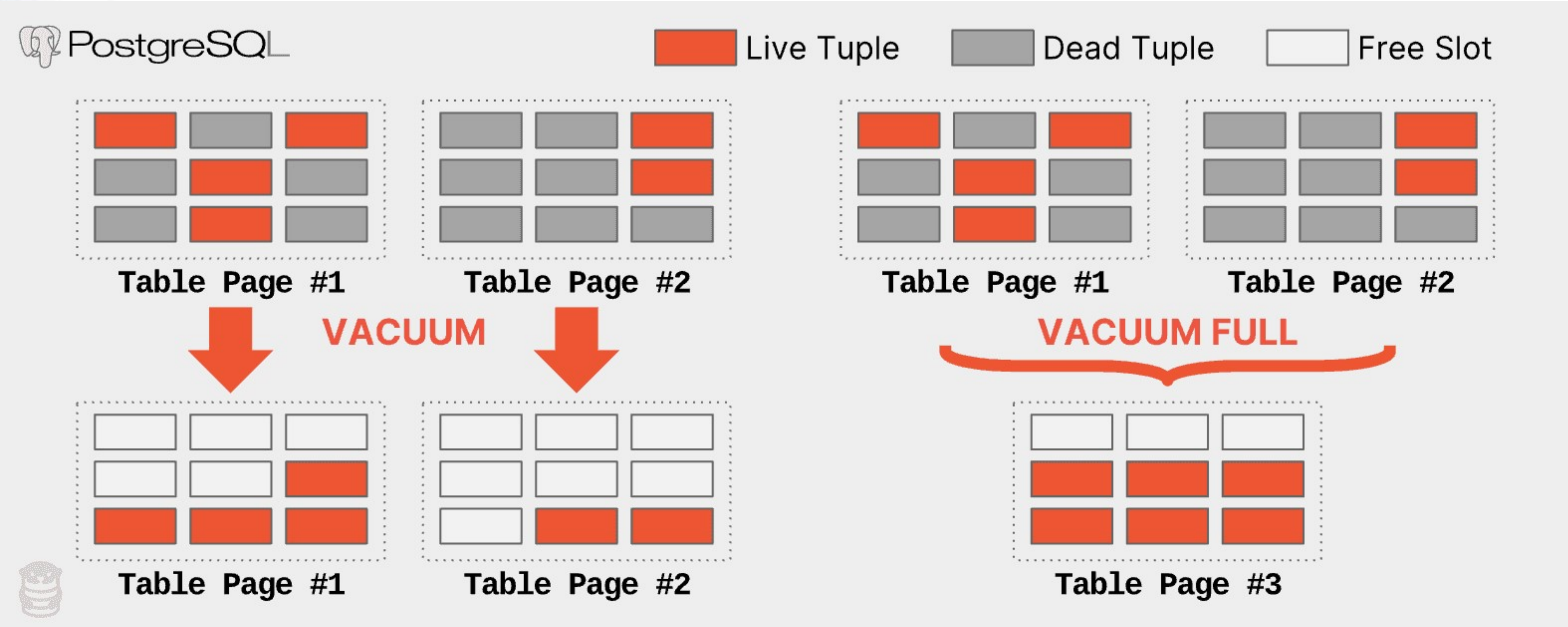

Table Page #1

| next ver | id | name | year | director |
|---|---|---|---|---|
| - | 101 | Executioners from Shaolin | 1977 | [redacted] |
| | 102 | Shaolin and Wu Tang | 1985 | Chia-Hui Liu |
| | 103 | Five Deadly Venoms | 1978 | Cheh Chang |

Table Page #2

| next ver | id | name | year | director |
|---|---|---|---|---|
| - | 102 | Shaolin and Wu Tang | 1983 | Chia-Hui Liu |
| - | | | | |
| - | | | | |

# Live/Dead Tuples – Vacuum Operation



# MVCC Drawbacks - #1 : Version Copying

- With the append-only storage scheme in MVCC, **if a query updates a tuple**, the DBMS **copies all its columns into the new version**.

- This copying occurs no matter if the query updates a single or all of its columns.

- As you can imagine, append-only MVCC results in massive data duplication and increased storage requirements

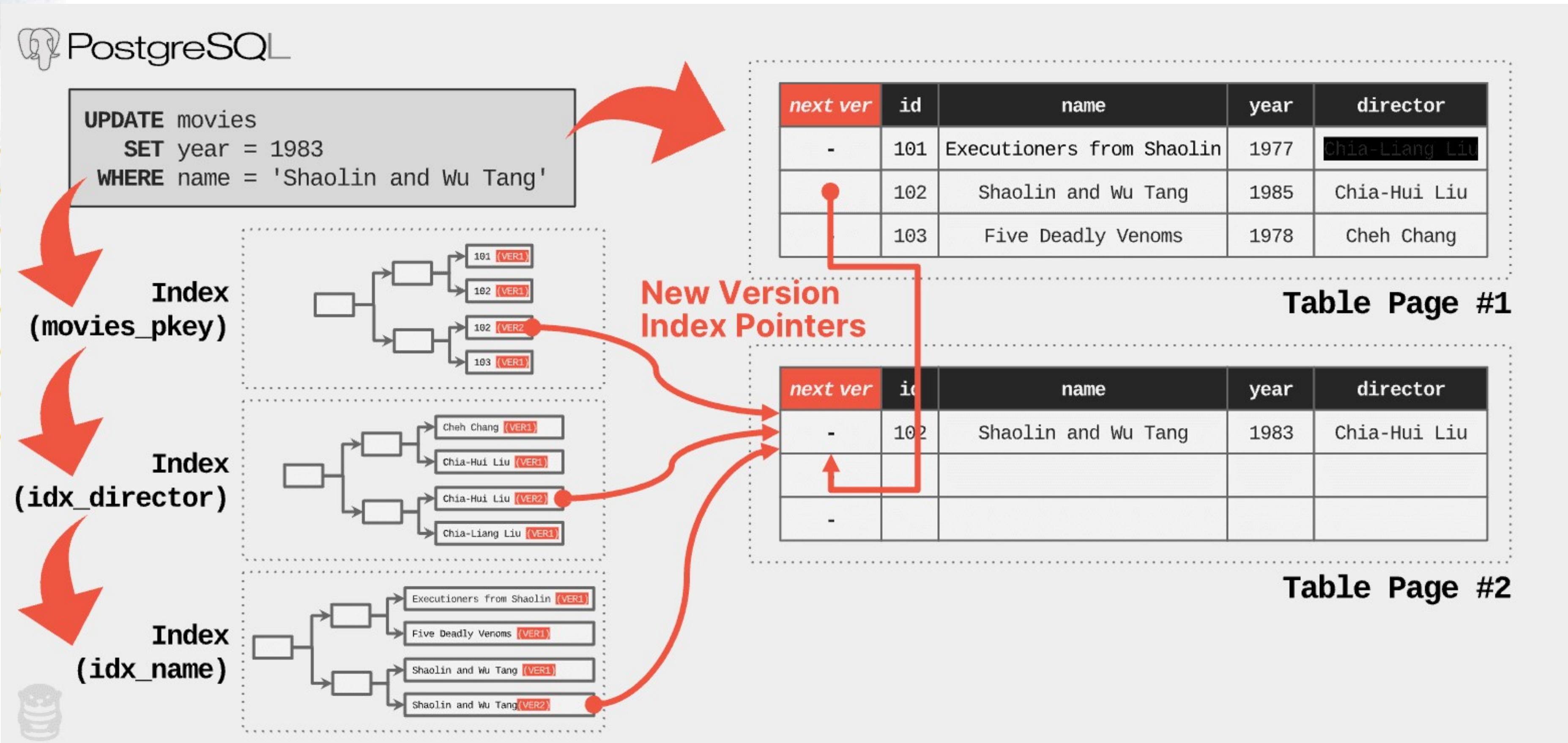
Ref: https://ottertune.com/blog/the-part-of-postgresql-we-hate-the-most

# MVCC Drawbacks - #2 : Table bloat

- Although PostgreSQL's autovacuum will eventually remove these dead tuples, **write-heavy workloads can cause them to accumulate faster than the vacuum can catch up**, resulting in continuous database growth.

- Suppose our movies table has 10 million live and 40 million dead tuples, making 80% of the table obsolete data.
  - Assume also that the table also has many more columns than what we are showing and that the average size of each tuple is 1KB.
  - With this scenario, the live tuples occupy 10GB of storage space while the dead tuples occupy ~40GB of storage;
  - the total size of the table is 50GB

# MVCC Drawbacks - #3 : Secondary Index Maintenance



# MVCC Drawbacks - #4 : Vacuum management

- PostgreSQL's performance relies heavily on the effectiveness of the autovacuum to remove obsolete data and reclaim space

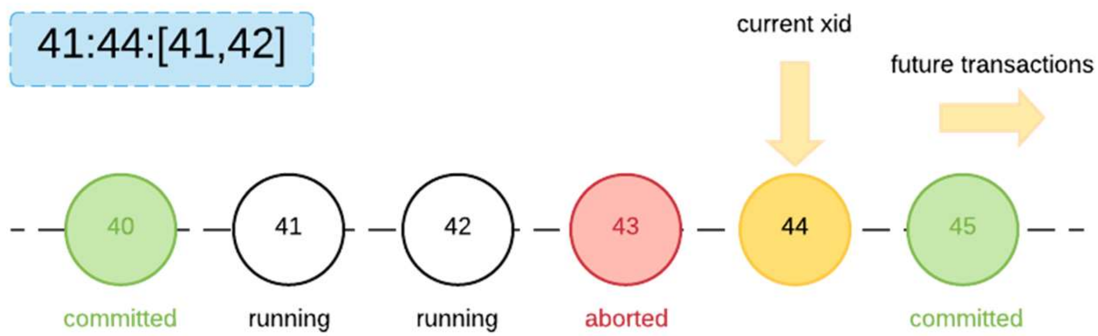
- PostgreSQL's default settings for tuning the autovacuum are not ideal for all tables, particularly for large ones
    - the default setting for the configuration knob that controls what percentage of a table PostgreSQL has to update before the autovacuum kicks in (autovacuum_vacuum_scale_factor) is 20%
    - if a table has 100 million tuples, the DBMS does not trigger the autovacuum until queries update at least 20 million tuples.

- AutoVacuum may get blocked by long-running transactions …

# Vacuum and : XMAX

**The elimination operation must evaluate it against several criteria which must all apply:**

- `xmax` must be different from zero because a value of zero indicates that the row version is not deleted.
- `xmax` must contain an XID which is older than the oldest XID of all currently running transactions. That guarantees that no existing or upcoming transaction will have read or write access to this row version.
- The transaction of `xmax` must be committed. If it is still running or was rollback-ed, this row version is treated as valid (not deleted).
- If there is a situation that the row version is part of multiple transactions, more actions must be taken.



# MVCC VS Locking

- In MVCC locks acquired for querying (**reading**) data do not conflict with locks acquired for **writing** data

- **Reading never blocks Writing and Writing never blocks Reading**

- Table- and row-level **locking facilities are also available** in PostgreSQL for applications which don't generally need full transaction isolation

# Transaction Control

```sql
-- TRANSACTION #1

SELECT balance
FROM accounts
WHERE owner = 'Bob';
-- balance: 500$







UPDATE accounts
SET balance = 600
WHERE owner = 'Bob';
COMMIT;

-- balance: 600$
-- (300$ was lost)
```

```sql
-- TRANSACTION #2






SELECT balance
FROM accounts
WHERE owner = 'Bob';
-- balance: 500$

UPDATE accounts
SET balance = 800
WHERE owner = 'Bob';
COMMIT;
-- balance: 800$






-- balance: should be 900$
-- (500$ + 300$ + 100$)
```

# Transaction Control

**BEGIN [TRANSACTION]** − To start a transaction.

**COMMIT** − To save the changes, alternatively you can use **END TRANSACTION** command.

**ROLLBACK** − To rollback the changes.

```
testdb=# BEGIN;
DELETE FROM COMPANY WHERE AGE = 25;
ROLLBACK;
```

# Transaction Sample

```sql
BEGIN;

UPDATE accounts
SET balance = balance - 1000
WHERE id = 1;

UPDATE accounts
SET balance = balance + 1000
WHERE id = 2;

COMMIT;
```

# Transaction Save Point

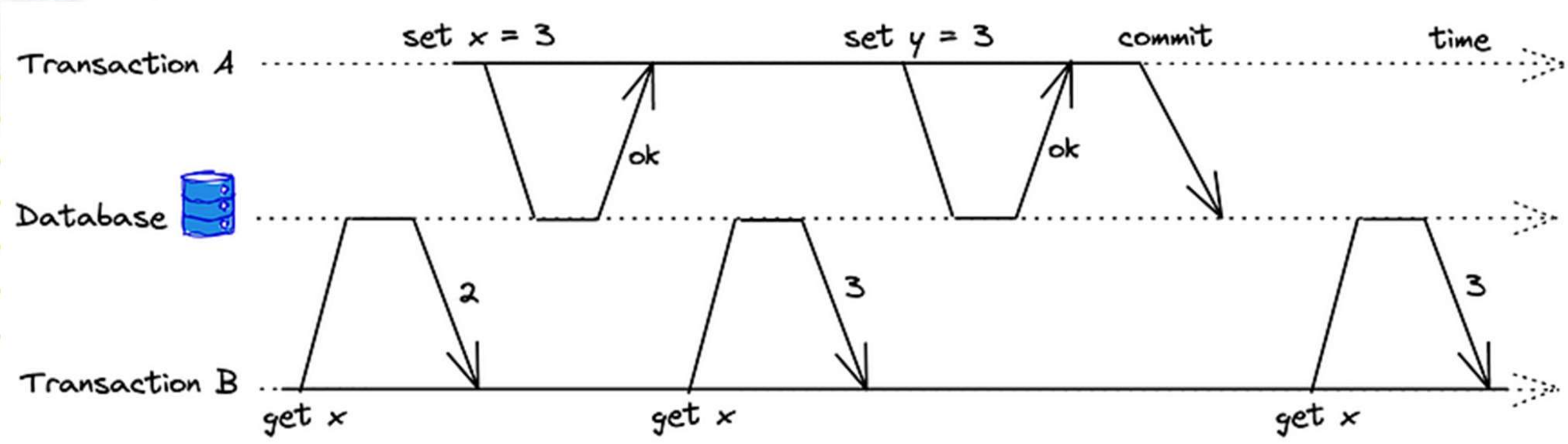
```
BEGIN;
UPDATE accounts
SET balance = balance - 1500
WHERE id = 1;
/* Set a save point that we can return to */
SAVEPOINT save_1;

UPDATE accounts
SET balance = balance + 1500
WHERE id = 3; -- Wrong account number here! We can rollback to the save point though!
/* Gets us back to the state of the transaction at `save_1` */

ROLLBACK TO save_1;
/* Continue the transaction with the correct account number */
UPDATE accounts
SET balance = balance + 1500
WHERE id = 4;
COMMIT;
```

# Concurrency Issues – Dirty Read

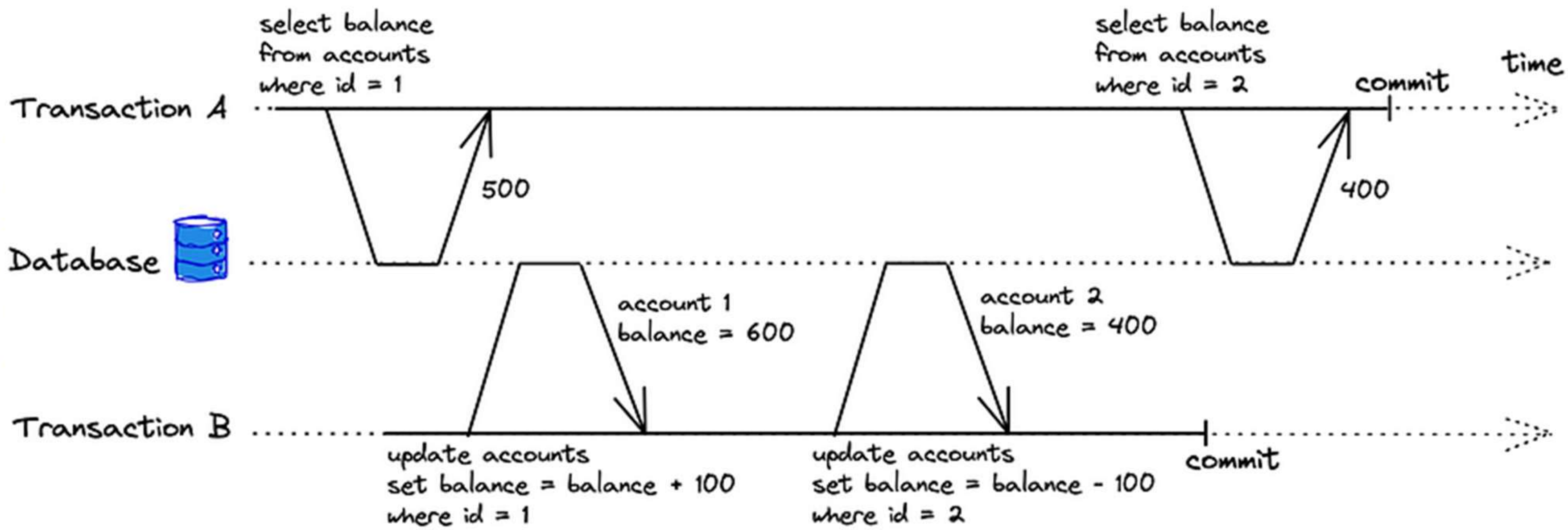
*Dirty read* means a transaction can see data that hasn't been committed by other transactions.




- transaction B can see the new value of x (3) even though transaction A hasn't been committed
- Furthermore, it also violates the **atomicity** property. If transaction A fails, the intermediate data will not be discarded and will probably be saved to the database by transaction B

# Concurrency Issues – Non-repeatable read

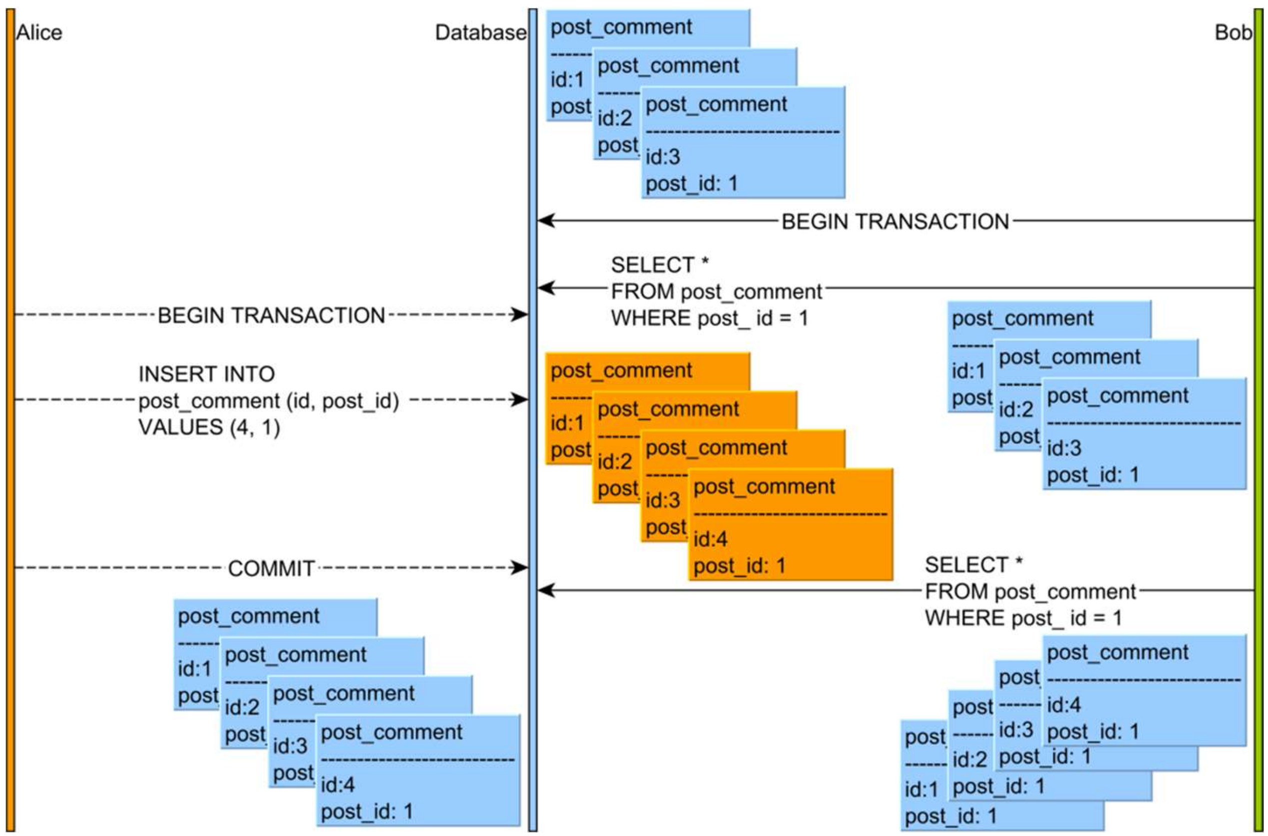
*Non-repeatable read* is the problem that a transaction queries data at different points of time but it gets **different results** because the data has been **modified** by other **committed** transactions.




Let's imagine a user has a total of $1000 and divides them equally into 2 accounts. One day he transfers $100 from account 2 to account 1 (transaction B). In the end, account 1 should have $600 and account 2 should have $400. At the same time, an admin of the system queries the balances of two accounts (transaction A).

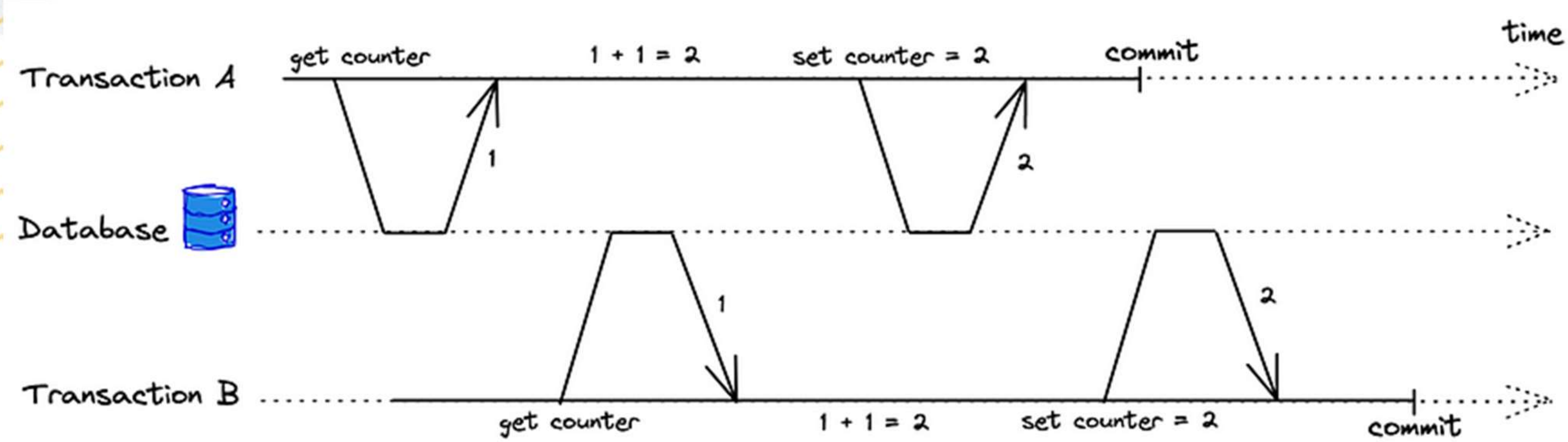
# Concurrency Issues – Phantom Read

Phantom read is the problem that a transaction queries data at different points of time but it gets **different results** because the data has been **inserted** or **deleted** by other **committed** transactions.



# Concurrency Issues – Lost Update

*Lost update* happens when multiple concurrent transactions **read** the same value from database, **modify** and **write back** their modified value



# Concurrency Issues – Write skew

If multiple concurrent transactions query data from database, make a decision based on it, write **different parts** of data back, and cause the data to become inconsistent, it is called *Write skew* problem.

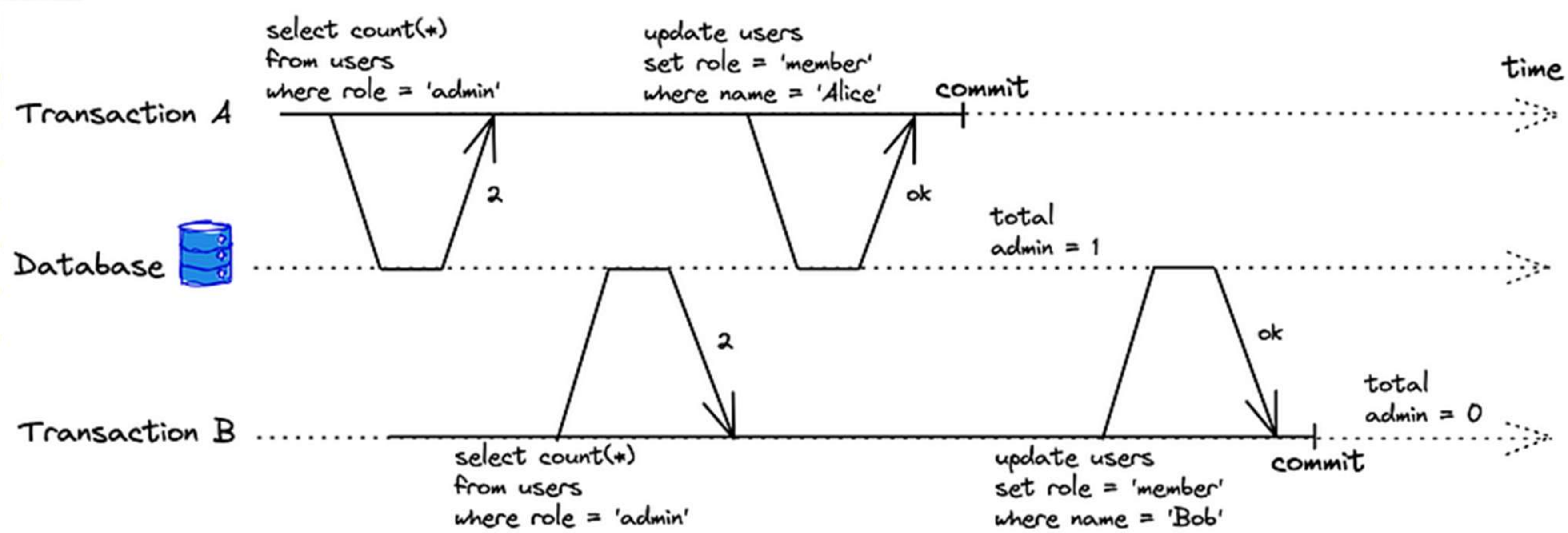
| | name | role |
|---|---|---|
| 1 | Alice | admin |
| 2 | Bob | admin |
| 3 | Jack | member |

**Data before write skew happens**

The system ensures that an organization always has at least one admin to function properly

# Concurrency Issues – Write skew

Alice and Bob just started learning this new system and they changed their role to member to see what can a member do. Unfortunately, they do at the same time and the process happens as this diagram:




*Write skew* is a generalization of *Lost update*. In this case, transactions write distinct data, they don't overwrite each other but the inconsistency still occurs.

# Isolation levels

**READ UNCOMMITTED:**

Allows transactions to read uncommitted changes.
Not natively supported in PostgreSQL.

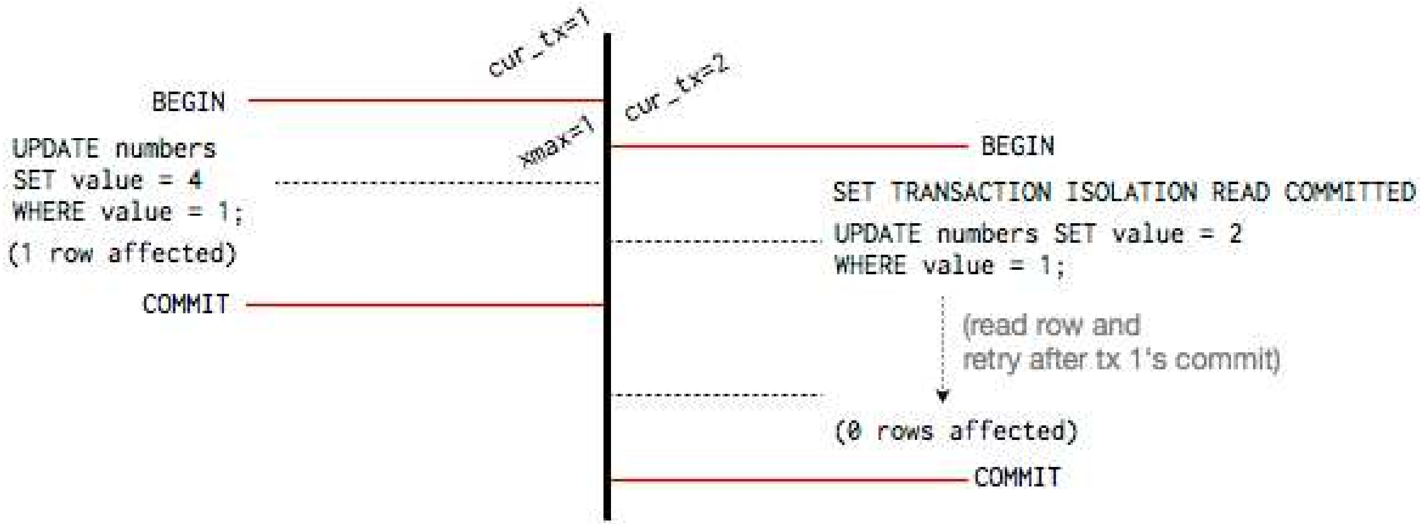
**READ COMMITTED:**

Ensures a transaction sees only committed changes.
Default isolation level in PostgreSQL.
Avoids dirty reads but may allow non-repeatable reads and phantom reads.

# Isolation levels

| | READ UNCOMMITTED | READ COMMITTED | REPEATABLE READ | SERIALIZABLE |
|---|---|---|---|---|
| DIRTY READ | NO | NO | NO | NO |
| NON-REPEATABLE READ | YES | YES | NO | NO |
| PHANTOM READ | YES | YES | NO | NO |
| LOST UPDATE | YES | YES | YES | NO |
| WRITE SKEW | YES | YES | YES | NO |

# Isolation levels



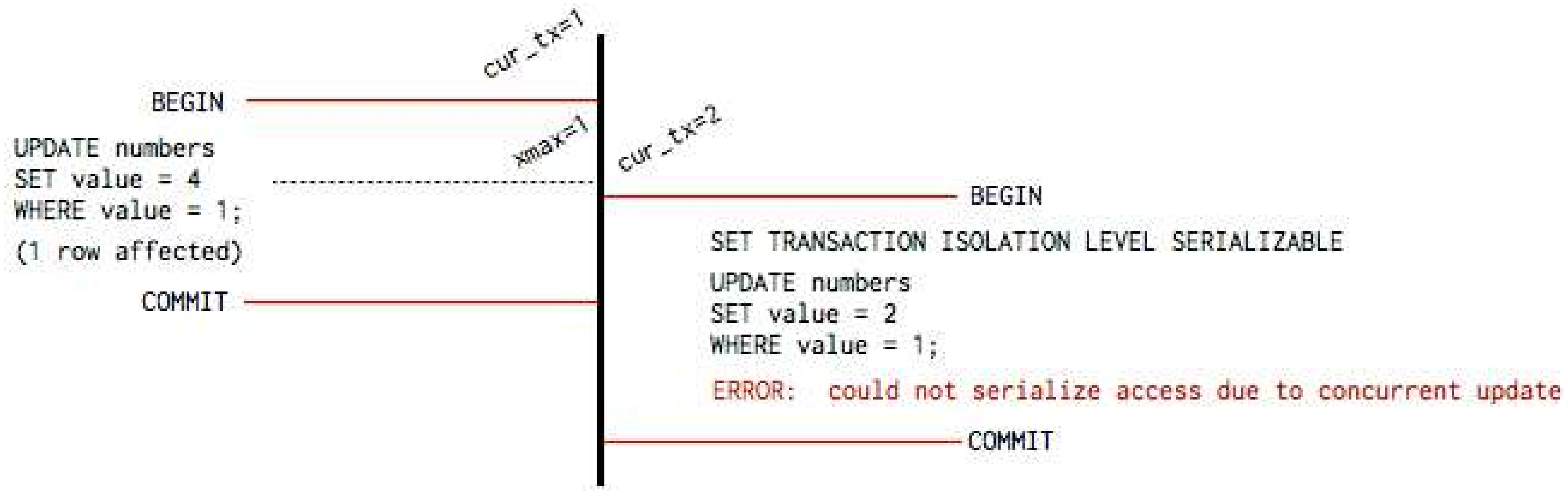

The default, READ COMMITTED, reads the row after the initial transaction has completed and then executes the statement. It basically starts over if the row changed while it was waiting.

# Isolation levels

**REPEATABLE READ:**
Guarantees that within a transaction, the same query produces the same result.
Prevents dirty reads and non-repeatable **(Phantom Read)** reads but may allow phantom reads.

**SERIALIZABLE:**
Provides the highest isolation level.
Guarantees serializability, preventing dirty reads, non-repeatable reads, and phantom reads.
Can be more resource-intensive due to locking.

# Isolation levels




If you need finer control over this behavior, you can set the transaction isolation level to **SERIALIZABLE**.

With this strategy the above scenario fails because it says “**If the row I’m modifying has been modified by another transaction, don’t even try,”** and Postgres responds with the error message ERROR: could not serialize access due to concurrent update.

It’s up to your app to handle that error and try again, or to give up if that’s what makes sense.

# Setting the Isolation level

```
BEGIN ISOLATION LEVEL
<isolation_level>;
statements
COMMIT;
```

**Isolation levels:**

- **READ UNCOMMITTED** (will result in READ COMMITTED since this level isn't implemented in PostgreSQL)
- **READ COMMITTED**
- **REPEATABLE READ**
- **SERIALIZABLE**

# Transaction Control : Which Isolation Level?

```sql
-- TRANSACTION #1

SELECT balance
FROM accounts
WHERE owner = 'Bob';
-- balance: 500$

UPDATE accounts
SET balance = 600
WHERE owner = 'Bob';
COMMIT;

-- balance: 600$
-- (300$ was lost)
```

```sql
-- TRANSACTION #2

SELECT balance
FROM accounts
WHERE owner = 'Bob';
-- balance: 500$

UPDATE accounts
SET balance = 800
WHERE owner = 'Bob';
COMMIT;
-- balance: 800$

-- balance: should be 900$
-- (500$ + 300$ + 100$)
```

# Transaction Control : Which Isolation Level?

When you run the following code in PostgreSQL one of interleaving transactions will crash and it will need to be manually retried from your application:

```
BEGIN ISOLATION LEVEL REPEATABLE READ;
SELECT balance FROM accounts WHERE owner = 'Bob';
UPDATE accounts SET balance = ... WHERE owner = 'Bob';

-- it will crash here if Bob's account has been modified
-- since the beginning of this transaction
COMMIT;
```

If you are curious this is the error that will be thrown:
`ERROR: could not serialize access due to concurrent update SQL state: 40001`

# INTEGRITY CONSTRAINTS

## ADVANCED TOPICS

# Integrity Constraints

- Integrity constraints guard against accidental damage to the database, by ensuring that authorized changes to the database do not result in a loss of data consistency.
  - A checking account must have a balance greater than $10,000.00
  - A salary of a bank employee must be at least $4.00 an hour
  - A customer must have a (non-null) phone number

# Constraints on a Single Relation

- **not null**
- **primary key**
- **unique**
- **check** (P), where P is a predicate

# Not Null Constraints

- **not null**
  - Declare *name* and *budget* to be **not null**

    *name* **varchar**(20) **not null**
    *budget* **numeric**(12,2) **not null**

# Unique Constraints

- **unique** ( $A_1$, $A_2$, …, $A_m$)
  - The unique specification states that the attributes $A_1$, $A_2$, …, $A_m$ form a candidate key.
  - Candidate keys are permitted to be null (in contrast to primary keys).

# The check clause

- The **check** (P) clause specifies a predicate P that must be satisfied by every tuple in a relation.
- Example: ensure that semester is one of fall, winter, spring or summer

**create table** *section*
  (*course_id* **varchar** (8),
  *sec_id* **varchar** (8),
  *semester* **varchar** (6),
  *year* **numeric** (4,0),
  *building* **varchar** (15),
  *room_number* **varchar** (7),
  *time slot id* **varchar** (4),
  **primary key** (*course_id*, *sec_id*, *semester*, *year*),
  **check** (*semester* **in** ('Fall', 'Winter', 'Spring', 'Summer')))

# Referential Integrity

- Ensures that a value that appears in one relation for a given set of attributes also appears for a certain set of attributes in another relation.
  - Example: If “Biology” is a department name appearing in one of the tuples in the *instructor* relation, then there exists a tuple in the *department* relation for “Biology”.
- Let A be a set of attributes. Let R and S be two relations that contain attributes A and where A is the primary key of S. A is said to be a **foreign key** of R if for any values of A appearing in R these values also appear in S.

# Referential Integrity (Cont.)

- Foreign *keys can be* specified as part of the SQL **create table** statement

  **foreign key** (*dept_name*) **references** *department*

- By default, a foreign key references the primary-key attributes of the referenced table.
- SQL allows a list of attributes of the referenced relation to be specified explicitly.

  **foreign key** (*dept_name*) **references** *department* (*dept_name*)

# Cascading Actions in Referential Integrity

- When a referential-integrity constraint is violated, the normal procedure is to reject the action that caused the violation.
- An alternative, in case of delete or update is to cascade

**create table** *course* (
(…
*dept_name* **varchar**(20),
**foreign key** (*dept_name*) **references** *department*
**on delete cascade**
**on update cascade**,
. . .)

- Instead of cascade we can use :
  - **set null**,
  - **set default**

# Integrity Constraint Violation During Transactions

- Consider:

  **create table** *person* (
  *ID* **char**(10),
  *name* **char**(40),
  *mother* **char**(10),
  *father* **char**(10),
  **primary key** *ID,*
  **foreign key** *father* **references** *person,*
  **foreign key** *mother* **references** *person*)

- How to insert a tuple without causing constraint violation?
  - Insert father and mother of a person before inserting person
  - OR, set father and mother to null initially, update after inserting all persons (not possible if father and mother attributes declared to be **not null**)
  - OR defer constraint checking

# Complex Check Conditions

- The predicate in the check clause can be an arbitrary predicate that can include a subquery.

  **check** (*time_slot_id* **in** (**select** *time_slot_id* **from** *time_slot*))

  The check condition states that the time_slot_id in each tuple in the *section* relation is actually the identifier of a time slot in the *time_slot* relation.

  - The condition has to be checked not only when a tuple is inserted or modified in *section* , but also when the relation *time_slot* changes

# Assertions

- An **assertion** is a predicate expressing a condition that we wish the database always to satisfy.
- The following constraints, can be expressed using assertions:
- For each tuple in the *student* relation, the value of the attribute *tot_cred* must equal the sum of credits of courses that the student has completed successfully.
- An instructor cannot teach in two different classrooms in a semester in the same time slot
- An assertion in SQL takes the form:

  **create assertion** <assertion-name> **check** (<predicate>);

# Assertions

- **We do not Have Subqueries in Check Constraints Postgres!**
- **We do not Have Assertion in Postgres!**

**List of SQL-Standard Features that not implemented in Postgres:**

https://www.postgresql.org/docs/current/unsupported-features-sql-standard.html

| | | | |
|---|---|---|---|
| F291 | | UNIQUE predicate | |
| F301 | | CORRESPONDING in query expressions | |
| F403 | | Partitioned join tables | |
| F451 | | Character set definition | |
| F461 | | Named character sets | |
| F492 | | Optional table constraint enforcement | |
| F521 | | Assertions | |
| F671 | | Subqueries in CHECK constraints | intentionally omitted |

# User-Defined Types

- **create type** construct in SQL creates user-defined type

  **create type** *Dollars* **as numeric (12,2) final**

- Example:

  **create table** *department*
  (*dept_name* **varchar** (20),
  *building* **varchar** (15),
  *budget Dollars*);

# Domains

- **create domain** construct in SQL-92 creates user-defined domain types

  **create domain** *person_name* **char**(20) **not null**

- Types and domains are similar. Domains can have constraints, such as **not null**, specified on them.
- Example:

  **create domain** *degree_level* **varchar**(10)
  **constraint** *degree_level_test*
  **check** (**value in** ('Bachelors', 'Masters', 'Doctorate'));

# AUTHORIZATION

# Authorization

- We may assign a user several forms of authorizations on parts of the database.
  - **Read** - allows reading, but not modification of data.
  - **Insert** - allows insertion of new data, but not modification of existing data.
  - **Update** - allows modification, but not deletion of data.
  - **Delete** - allows deletion of data.
- Each of these types of authorizations is called a **privilege**. We may authorize the user all, none, or a combination of these types of privileges on specified parts of a database, such as a relation or a view.

# Authorization (Cont.)

- Forms of authorization to modify the database schema
  - **Index** - allows creation and deletion of indices.
  - **Resources** - allows creation of new relations.
  - **Alteration** - allows addition or deletion of attributes in a relation.
  - **Drop** - allows deletion of relations.

# Authorization Specification in SQL

- The **grant** statement is used to confer authorization

  **grant** <privilege list> **on** <relation or view > **to** <user list>
- <user list> is:
  - a user-id
  - **public**, which allows all valid users the privilege granted
  - A role (more on this later)
- Example:
  - **grant select on** *department* **to** Amit, Satoshi
- Granting a privilege on a view does not imply granting any privileges on the underlying relations.
- The grantor of the privilege must already hold the privilege on the specified item (or be the database administrator).

# Privileges in SQL

- **select**: allows read access to relation, or the ability to query using the view
  - Example: grant users $U_1$, $U_2$, and $U_3$ **select** authorization on the *instructor* relation:

    **grant select on** *instructor* **to** $U_1, U_2, U_3$
- **insert**: the ability to insert tuples
- **update**: the ability to update using the SQL update statement
- **delete**: the ability to delete tuples.
- **all privileges**: used as a short form for all the allowable privileges

# Revoking Authorization in SQL

- The **revoke** statement is used to revoke authorization.

  **revoke** <privilege list> **on** <relation or view> **from** <user list>

- Example:

  **revoke select on** *student* **from** $U_1$, $U_2$, $U_3$

- <privilege-list> may be **all** to revoke all privileges the revokee may hold.
- If <revokee-list> includes **public,** all users lose the privilege except those granted it explicitly.
- If the same privilege was granted twice to the same user by different grantees, the user may retain the privilege after the revocation.
- All privileges that depend on the privilege being revoked are also revoked.

# Roles

- A **role** is a way to distinguish among various users as far as what these users can access/update in the database.
- To create a role we use:

  **create a role** <name>
- Example:
  - **create role** instructor
- Once a role is created we can assign "users" to the role using:
  - **grant** <role> **to** <users>

# Roles Example

- **create role** instructor;
- **grant** *instructor* **to** Amit**;**
- Privileges can be granted to roles:
  - **grant select on** *takes* **to** *instructor*;
- Roles can be granted to users, as well as to other roles
  - **create role** *teaching_assistant*
  - **grant** *teaching_assistant* **to** *instructor*;
    - *Instructor* inherits all privileges of *teaching_assistant*
- Chain of roles
  - **create role** *dean*;
  - **grant** *instructor* **to** *dean*;
  - **grant** *dean* **to** Satoshi;

# Authorization on Views

- **create view** *geo_instructor* **as**
  (**select** *
  **from** *instructor*
  **where** *dept_name* = 'Geology');
- **grant select on** *geo_instructor* **to** *geo_staff*
- Suppose that a *geo_staff* member issues
  - **select** *
    **from** *geo_instructor*;
- What if
  - *geo_staff* does not have permissions on *instructor?*
  - Creator of view did not have some permissions on *instructor?*

# Other Authorization Features

- **references** privilege to create foreign key
  - **grant reference** (*dept_name*) **on** *department* **to** Mariano;
  - Why is this required?
- transfer of privileges
  - **grant select on** *department* **to** Amit **with grant option**;
  - **revoke select on** *department* **from** Amit, Satoshi **cascade**;
  - **revoke select on** *department* **from** Amit, Satoshi **restrict**;
  - And more!

# Create User/Role Sample

-- 1. Creating a User

- CREATE USER john WITH PASSWORD 'john_password';

-- 2. Defining a Role

- CREATE ROLE sales_team;

-- 3. Assigning User to Role

- ALTER USER john SET ROLE sales_team;

-- 4. Granting Permission to Role

- GRANT SELECT ON TABLE sales_data TO sales_team;

# End of Chapter 4