



POSTGRESQL VIEW. INDEX. TRANSACTIONS. ACID PROPERTIES.

DBMS. LECTURE WEEK9-10.



POSTGRESQL INDEXES

- PostgreSQL indexes are effective tools to enhance database performance.
- Indexes help the database server find specific rows much faster than it could do without indexes.
- However, indexes add write and storage overheads to the database system.
- Therefore, using them appropriately is very important.

EXPLANATION

- Let's assume we have a table:

```
CREATE TABLE test1 (  
  Id INT,  
  Content VARCHAR );
```

- `SELECT content FROM test1 WHERE id = number;`

SYNTAX

```
CREATE INDEX index_name ON table_name [USING method]
( column_name [ASC | DESC] [NULLS {FIRST | LAST }], ... );
```

- In this syntax:
 - First, specify the index name after the CREATE INDEX clause. The index name should be meaningful and easy to remember.
 - Second, specify the name of the table to which the index belongs.
 - Third, specify the index method such as btree, hash, gist, spgist, gin, and brin. PostgreSQL uses btree by default.
 - Fourth, list one or more columns that are to be stored in the index.
 - The ASC and DESC specify the sort order. ASC is the default.
 - NULLS FIRST or NULLS LAST specifies nulls sort before or after non-nulls. The NULLS FIRST is the default when DESC is specified and NULLS LAST is the default when DESC is not specified.
- To check if a query uses an index or not, you use the EXPLAIN statement.

CREATION EXAMPLE

```
CREATE INDEX test1 id index ON test1 (id);
```



Name of the index is custom.

To drop index you need to use:

```
DROP INDEX index_name
```

EXAMPLE

```
explain  
select * from film
```

To look on the query plan.

Gives us:

Scanning a table sequentially.

The screenshot shows a database interface with a 'QUERY PLAN' section. The first entry is '1 Seq Scan on film (cost=0.00..64.00 rows=1000 width=384)'. The text 'Seq Scan on' is circled in green. Above the screenshot, the text 'Gives us:' is followed by an arrow pointing to the circled text. To the right, a text box contains the explanation 'Scanning a table sequentially.', with an arrow pointing from the circled text to this box.

EXAMPLE CONT.

```
create index idx_title on film(title);
```

Creating a new index on *title* column in *film* table.

```
explain  
select * from film  
where film.title like 'A%';
```

Scanning a table by using index column.

Query plan gives:

The screenshot shows a database interface with a 'QUERY PLAN' section. The first step is 'Index Scan Using idx_title on film (cost=0.28..11.64 rows=40 width=384)'. The second step is 'Index Cond: (((title)::text >= 'A'::text) AND ((title)::text < 'B'::text))'. The third step is 'Filter: ((title)::text ~ 'A%'::text)'. The 'Index Scan' step is circled in red.

```
drop index idx_title;
```

To delete an index.

LIST INDEXES:

```
SELECT tablename,  
indexname,  
indexdef  
FROM pg_indexes  
WHERE schemaname = 'public'  
ORDER BY tablename, indexname;
```

stores name of the table to which the index belongs.

stores name of the index.

stores index definition command in the form of CREATE INDEX statement.

list indexes.

stores the name of the schema that contains tables and indexes.


```
select *
from pg_indexes
where tablename = 'film';
```

Gives a list of indexes in a *film* table:

	schemaname	tablename	indexname	tablespace	indexdef
1	public	film	film_pkey	<null>	CREATE UNIQUE INDEX film_pkey ON public...
2	public	film	film_fulltext_idx	<null>	CREATE INDEX film_fulltext_idx ON public...
3	public	film	idx_fk_language_id	<null>	CREATE INDEX idx_fk_language_id ON publi...
4	public	film	idx_title	<null>	CREATE INDEX idx_title ON public.film US...

- columns 13
- keys 1
- foreign keys 1
- indexes 4
 - film_pkey (film_id) UNIQUE
 - film_fulltext_idx (fulltext)
 - idx_fk_language_id (language_id)
 - idx_title (title)

Place in memory for indexes

INDEXES WITH ORDER BY CLAUSE

- In addition to simply finding strings to return from a query, indexes can also be used to sort strings in a specific order.
- Of all the index types that PostgreSQL supports, only B-trees can sort data - other types of indexes return rows in an undefined, implementation-dependent order.

YOU MAY ORDER BY ADDING:

- ASC,
- DESC,
- NULLS FIRST
- and / or NULLS LAST order
when creating an index

- **Examples:**

- `CREATE INDEX test2_info_nulls_low ON test2 (info NULLS FIRST);`
- `CREATE INDEX test3_desc_index ON test3 (id DESC NULLS LAST);`

UNIQUE INDEXES

- Indexes can also enforce the uniqueness of a value in a column or a unique combination of values in multiple columns.
- Currently, only B-tree indexes can be unique.
- When you define a UNIQUE index for a column, the column cannot store multiple rows with the same values.
- If you define a UNIQUE index for two or more columns, the combined values in these columns cannot be duplicated in multiple rows.
- PostgreSQL treats NULL as a distinct value, therefore, you can have multiple NULL values in a column with a UNIQUE index.
- When you define a primary key or a unique constraint for a table, PostgreSQL automatically creates a corresponding UNIQUE index.

■ `CREATE UNIQUE INDEX index_name ON table_name (column [, ...]);`

NOTE: Unique columns do not need to manually create separate indexes — they will simply duplicate the automatically generated indexes.

MULTICOLUMN INDEXES

- You can create an index on more than one column of a table.
- This index is called a **multicolumn index**, a **composite** index, a **combined** index, or a **concatenated** index.
- A multicolumn index can have maximum of 32 columns of a table. The limit can be changed by modifying the `pg_config_manual.h` when building PostgreSQL.
- In addition, only B-tree, GIST, GIN, and BRIN index types support multicolumn indexes.

```
CREATE INDEX index name ON  
table name (a, b, c, ...);
```

MULTICOLUMN INDEXES

- We have a table:

```
CREATE TABLE test2 (  
major INT,  
minor INT,  
name VARCHAR );
```

- You need to:

```
SELECT name FROM test2 WHERE major = value AND  
minor = value;
```

- In this case you may:

```
CREATE INDEX test2_mm_idx ON test2 (major,  
minor);
```

INDEXES ON EXPRESSIONS (FUNCTIONAL-BASED INDEXES)

- An index can be created not only on a column of the underlying table but also on a function or expression with one or more table columns. This allows you to quickly find data in a table based on the results of calculations.
- In this statement:
 - First, specify the name of the index after the CREATE INDEX clause.
 - Then, form an expression that involves table columns of the table_name.
- Once you define an index expression, PostgreSQL will consider using that index when the expression that defines the index appears in the WHERE clause or in the ORDER BY clause of the SQL statement.
- Note that indexes on expressions are quite expensive to maintain because PostgreSQL has to evaluate the expression for each row when it is inserted or updated and use the result for indexing. Therefore, you should use the indexes on expressions when retrieval speed is more critical than insertion and update speed.

```
CREATE INDEX index name ON  
table name (expression);
```

- For example, for case-insensitive comparisons:

```
SELECT * FROM test1 WHERE  
lower(col1) = 'value';
```

- We can use index:

```
CREATE INDEX test1_lower_col1_idx  
ON test1 (lower(col1));
```

- **Example2:**

- `SELECT * FROM people WHERE (first_name || ' ' || last_name) = 'John Smith';`

- **Index for Example 2 will be:**

- `CREATE INDEX people_names ON people ((first_name || ' ' || last_name));`

REINDEX

- In practice, an index can become corrupted and no longer contains valid data due to hardware failures or software bugs. To recover the index, you can use the REINDEX statement:

```
■ REINDEX [ (VERBOSE) ] { INDEX | TABLE | SCHEMA | DATABASE | SYSTEM } name;
```

- REINDEX INDEX index_name; -- to recreate a single index
- REINDEX TABLE table_name; -- to recreate all indexes of a table
- REINDEX SCHEMA schema_name; -- to recreate all indices in a schema
- REINDEX DATABASE database_name; -- to recreate all indices in a specific database
- REINDEX SYSTEM database_name; -- to recreate all indices on system catalogs

REINDEX VS. DROP INDEX & CREATE INDEX

The **REINDEX** statement:

- Locks writes but not reads of the table to which the index belongs.
- Takes an exclusive lock on the index that is being processed, which blocks reads that attempt to use the index.

The **DROP INDEX & CREATE INDEX** statements:

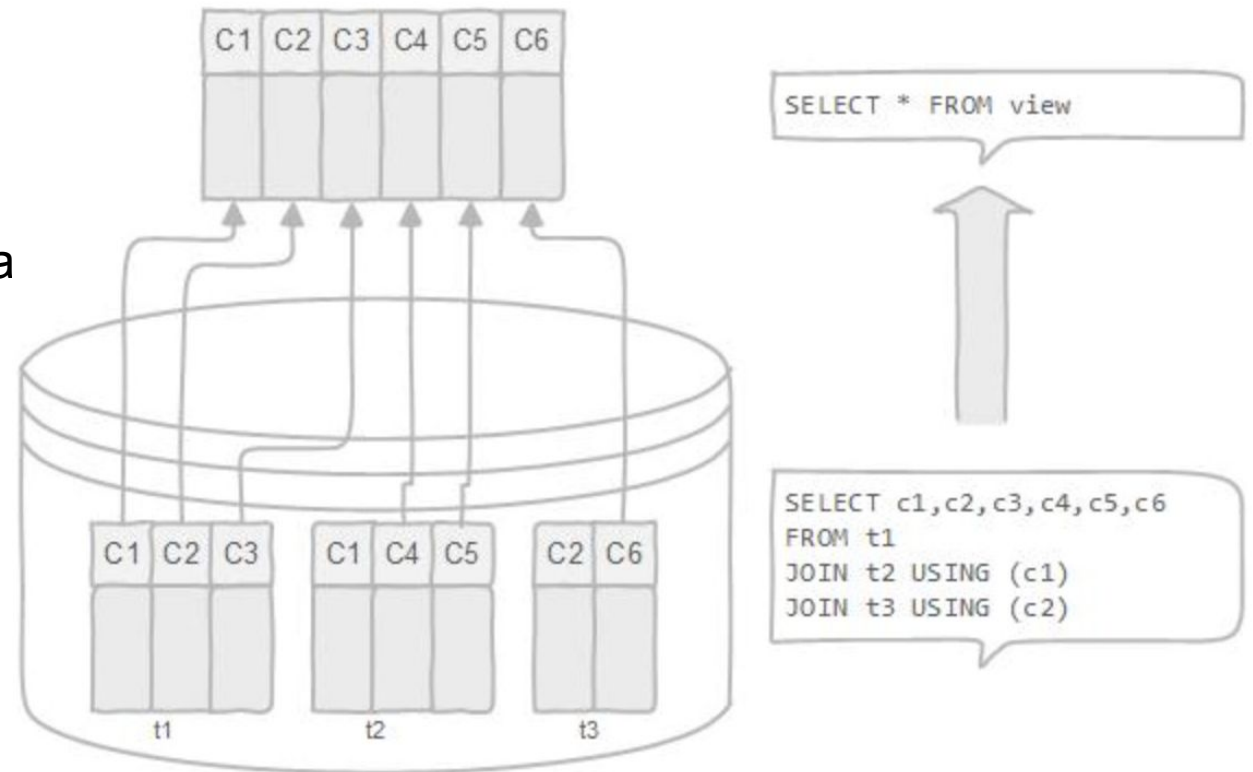
- First, the **DROP INDEX** locks both writes and reads of the table to which the index belongs by acquiring an exclusive lock on the table.
- Then, the subsequent **CREATE INDEX** statement locks out writes but not reads from the index's parent table. However, reads might be expensive during the creation of the index.

POSTGRESQL VIEW

- A view is a database object that is of a named (stored) query.
- When you create a view, you basically create a query and assign a name to the query. Therefore, a view is useful for wrapping a commonly used complex query.
- In PostgreSQL, a view is a pseudo-table.
- This means that a view is not a real table.
- However, we can `SELECT` it as an ordinary table.
- A view can have all or some of the table columns.
- A view can also be a representation of more than one table.
- A view itself does not store data physically except for materialized views.
- Materialized views store physical data and refreshes data periodically.

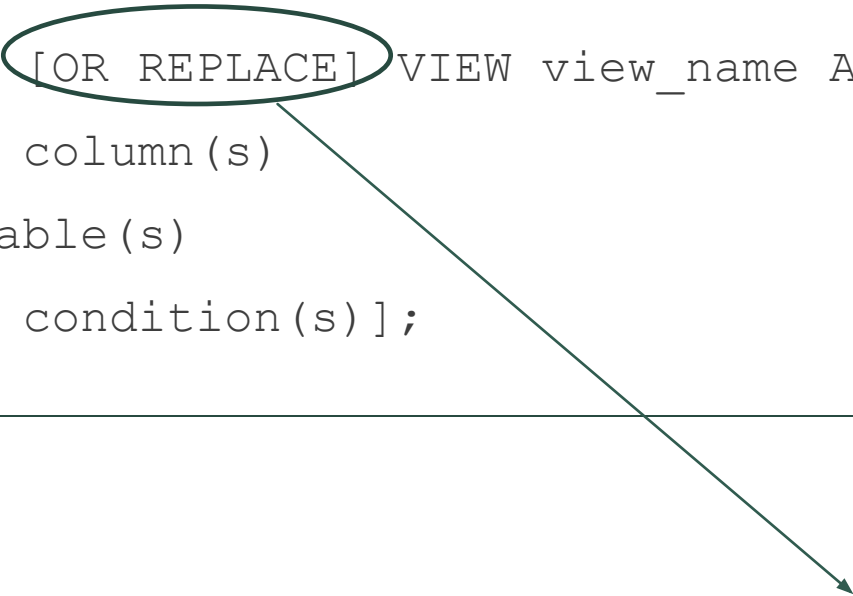
POSTGRESQL VIEW BENEFITS

- A view can be very useful in some cases such as:
 - A view helps **simplify the complexity of a query** because you can query a view, which is based on a complex query, using a simple SELECT statement.
 - Like a table, you can **grant permission** to users through a view that contains specific data that the users are authorized to see.
 - A view **provides a consistent layer** even the columns of the underlying table change.



CREATING VIEWS

```
CREATE [OR REPLACE] VIEW view_name AS  
SELECT column(s)  
FROM table(s)  
[WHERE condition(s)];
```



The `OR REPLACE` parameter will replace the view if it already exists. If omitted and the view already exists, an error will be returned.

MODIFYING AND REMOVING VIEWS

- **CREATE OR REPLACE** view_name **AS** query

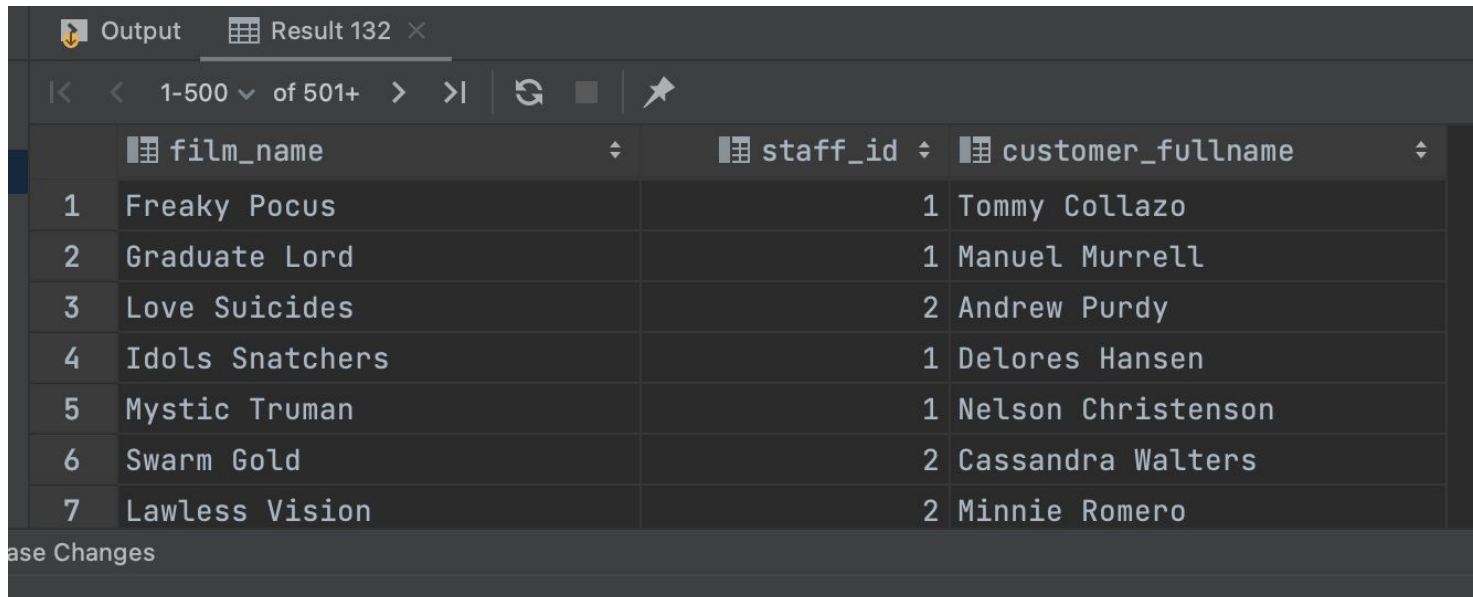
- **ALTER VIEW** view_name **RENAME TO** new_name;

- **DROP VIEW** [**IF EXISTS**] view_name;

EXAMPLE

```
select f.title film_name, r.staff_id, concat(c.first_name, ' ', c.last_name) customer_fullname
from film f join inventory i on f.film_id = i.film_id
join rental r on i.inventory_id = r.inventory_id
join customer c on r.customer_id = c.customer_id;
```

SELECT statement gives info about customers and films they took in rent:



Output Result 132

1-500 of 501+

	film_name	staff_id	customer_fullname
1	Freaky Pocus	1	Tommy Collazo
2	Graduate Lord	1	Manuel Murrell
3	Love Suicides	2	Andrew Purdy
4	Idols Snatchers	1	Delores Hansen
5	Mystic Truman	1	Nelson Christenson
6	Swarm Gold	2	Cassandra Walters
7	Lawless Vision	2	Minnie Romero

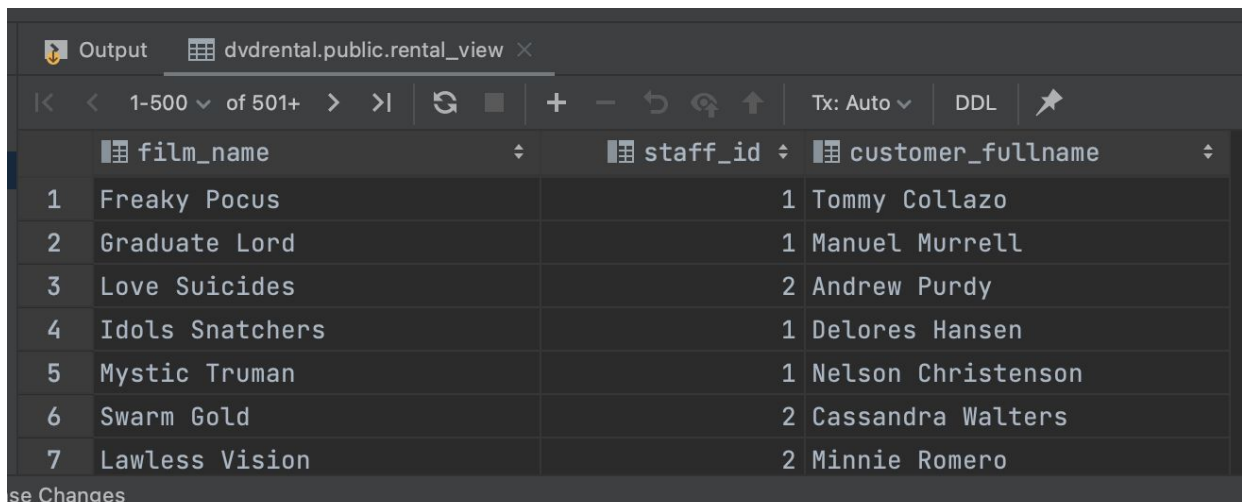
Base Changes

EXAMPLE CONT.

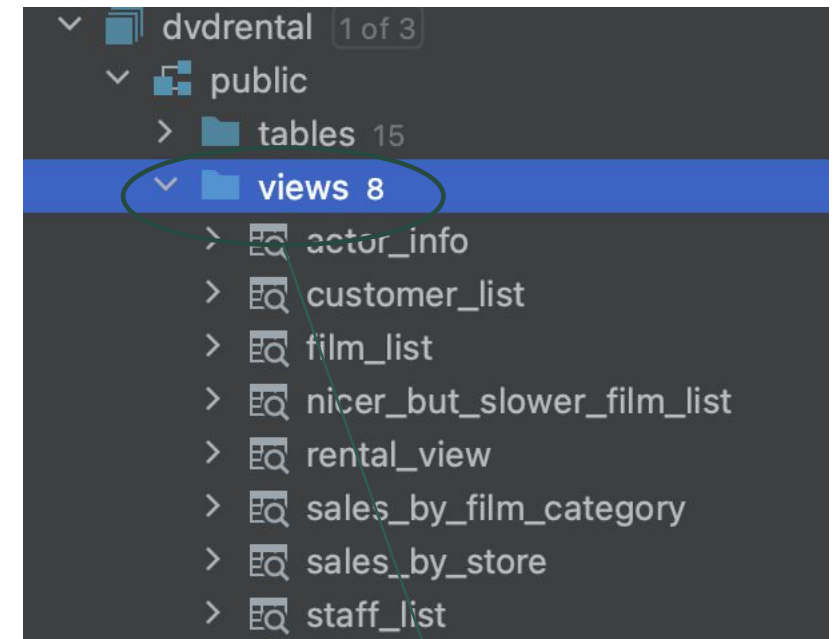
```
✓ create view rental_view as
select f.title film_name, r.staff_id, concat(c.first_name, ' ', c.last_name) customer_fullname
from film f join inventory i on f.film_id = i.film_id
join rental r on i.inventory_id = r.inventory_id
join customer c on r.customer_id = c.customer_id;
```

```
✓ select * from rental_view;
```

By creating a view the SELECT statement becomes shorter, but gives the same result:



	film_name	staff_id	customer_fullname
1	Freaky Pocus	1	Tommy Collazo
2	Graduate Lord	1	Manuel Murrell
3	Love Suicides	2	Andrew Purdy
4	Idols Snatchers	1	Delores Hansen
5	Mystic Truman	1	Nelson Christenson
6	Swarm Gold	2	Cassandra Walters
7	Lawless Vision	2	Minnie Romero



All views of the selected database.

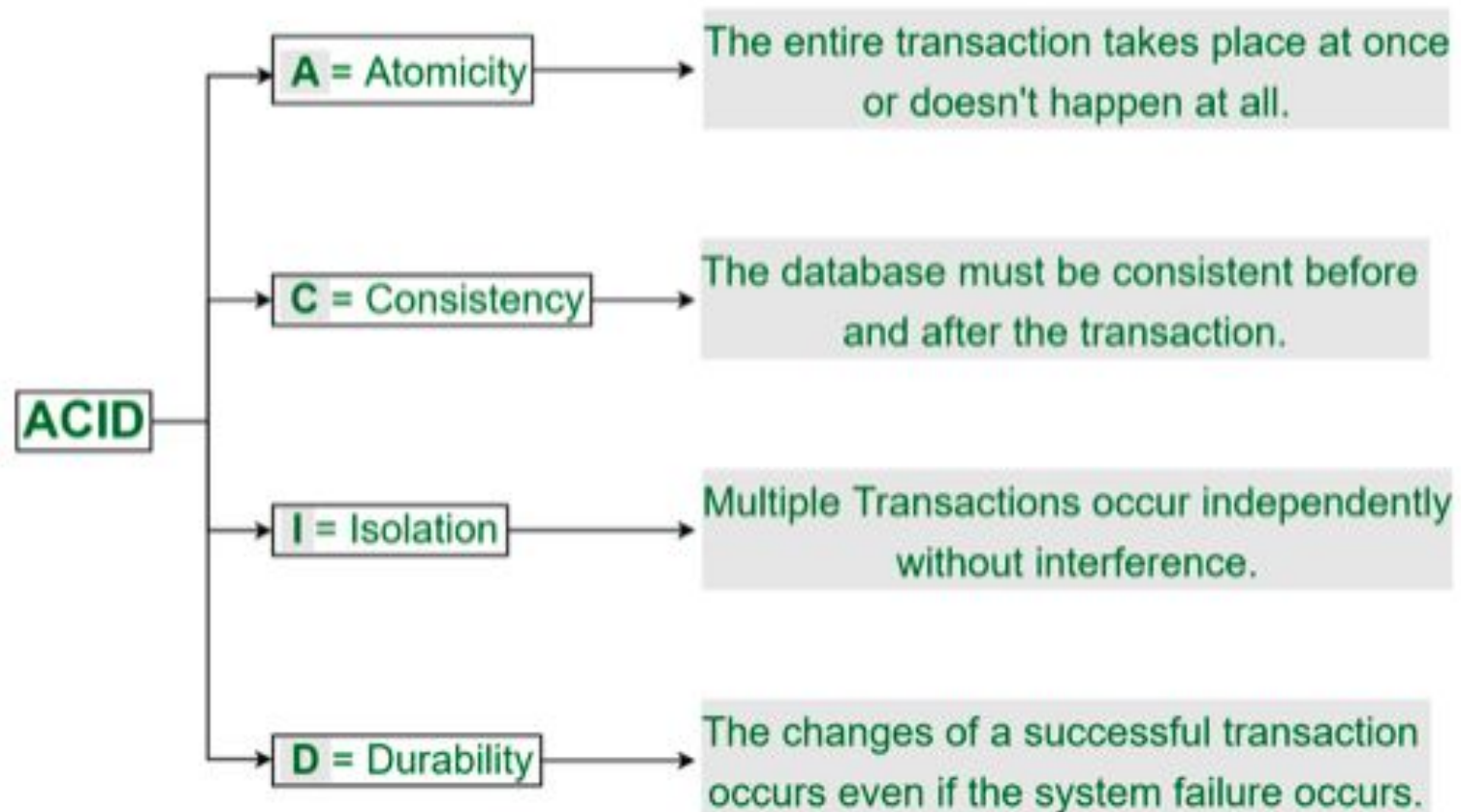
POSTGRESQL UPDATABLE VIEWS

- A PostgreSQL view **is updatable** when it meets the following **conditions**:
 - The defining query of the view **must have** exactly one entry in the **FROM** clause, which can be a table or another updatable view.
 - The defining query **must not contain** one of the following clauses at the top level: GROUP BY, HAVING, LIMIT, OFFSET, DISTINCT, WITH, UNION, INTERSECT, and EXCEPT.
 - The **selection list must not contain** any window function, any set-returning function, or any aggregate function.
 - An updatable view **may contain** both updatable and non-updatable columns. If you try to insert or update a non-updatable column, PostgreSQL will raise an error.
- When you execute an update operation such as INSERT, UPDATE, or DELETE, PostgreSQL will convert this statement into the corresponding statement of the underlying table.
- When you perform update operations, you must have corresponding privileges on the view, but you don't need to have privileges on the underlying table. However, view owners must have the relevant privilege of the underlying table.

DBMS TRANSACTIONS

- Transaction is a fundamental concept in all DBMSs.
- A **transaction** is a single logical unit of work which accesses and possibly modifies the contents of a database.
- The essence of a **transaction** is that *it combines a sequence of actions into one operation*.
- Transactions access data using read and write operations.
- In order to maintain consistency in a database, before and after the transaction, certain properties are followed. These are called **ACID** properties.

ACID PROPERTIES.



WHY USE TRANSACTIONS

- The main selling point for transactions is that they are easy to handle.
- Many database administrators use transactions to take advantage of a database's various features.
- Transactions can also simplify many tasks by automating part or most of the work.
- Transactions also add a layer of protection that can prevent simple mistakes from causing catastrophic failures.

ADVANTAGES OF USING TRANSACTIONS

■ Chaining Events Together

- We can chain some events together using multiple transactions in a database.
- For instance, if we want to design a transaction for customers filling out a form to get money, we can include several other events—such as sending their account balance, sending a request to the payment database, and then paying the customer.
- The only thing a local administrator will have to keep track of is the initial request and the response since most of the other stuff is handled by the transactions in the background.

ADVANTAGES OF USING TRANSACTIONS

■ **Flexibility**

- Flexibility is another primary advantage of database transactions.
- Using transactions allows us to change values in the database without accessing sensitive information—a perfect use case for corporate employee databases.
- In these databases, the user will only be able to access or change their information without knowing any of the sensitive details such as database passwords or server addresses.

ADVANTAGES OF USING TRANSACTIONS

■ **Avoiding Data Loss**

- Data loss is extremely common in the real world, with millions of people losing their data *every day* due to some technical difficulty or a glitch.
- We mentioned above that transactions are consistent, so using transactional databases will help maintain the data without any data losses due to technical errors.
- Transactional databases will also reduce the risk of losing any intermediate data if there is a power cut or an unexpected system shutdown.

ADVANTAGES OF USING TRANSACTIONS

■ Database Management

- Transactional databases make the jobs of many database administrators quite simple.
- Most transactional databases do not provide any way to change the data within a transaction to an end-user, so the user won't be able to change anything in the transaction that can allow them to take advantage of their state.

- In PostgreSQL, a transaction is defined by a set of SQL commands surrounded by **BEGIN** and **COMMIT**.

```
BEGIN;  
UPDATE accounts  
SET balance = balance - 100.00  
WHERE name = 'Alice';  
-- ...  
COMMIT;
```

PostgreSQL actually processes each SQL statement as a transaction

Transaction block

TRANSACTION CONTROL

- There are following commands used to control transactions:
- **BEGIN:** to start a transaction.
- **COMMIT:** to save the changes.
- **ROLLBACK:** to rollback the changes.

Transactional control commands are only used with the DML commands INSERT, UPDATE and DELETE only.

They can not be used while creating tables or dropping them because these operations are automatically committed in the database.

BEGIN COMMAND

- **PostgreSQL BEGIN** command is used to initiate a transaction.
- A **transaction** is nothing but a unit of work done in the database, the work can be anything from creating tables to deleting them.
- **BEGIN** command should be the first word of a transaction.
- **Syntax :**

```
BEGIN;  
// statements  
  
(or)  
  
BEGIN TRANSACTION;  
// statements
```

By default, PostgreSQL transactions are auto-commit, but to end the transaction block we need to give either **COMMIT** or **ROLLBACK** commands.

Statements inside the transaction block execute faster than normally given because the CPU uses special disk computation for defining transactions.

COMMIT COMMAND

- The COMMIT command is the transactional command used to save changes invoked by a transaction to the database.
- The COMMIT command saves all transactions to the database since the last COMMIT or ROLLBACK command.
- The syntax for COMMIT command is as follows:

```
COMMIT;
```

ROLLBACK COMMAND

- **PostgreSQL ROLLBACK** command is used to undo the changes done in transactions.
- As we know transactions in database languages are used for purpose of large computations, for example in banks.
- For suppose, the employee of the bank incremented the balance record of the wrong person mistakenly then he can simply rollback and can go to the previous state.
- Syntax:

```
ROLLBACK TRANSACTION  
(or)  
ROLLBACK;
```

SAVEPOINTS

- **Savepoints** allow you to selectively undo some parts of a transaction and commit all others.
- After defining a **SAVEPOINT**, you can return to it if necessary with the **ROLLBACK TO** command.
- All changes in the database that occurred after the savepoint and before the rollback are canceled, but the changes made earlier are saved.
- You can return to a savepoint several times.

Remember: when you delete or roll back to a savepoint, all savepoints defined after it are automatically destroyed.

EXAMPLE

- Consider a bank database that contains information about customer accounts, as well as total amounts by bank branch.

```
CREATE TABLE accounts (  
  id serial PRIMARY KEY,  
  name VARCHAR(100) NOT NULL,  
  balance DEC(15,2) NOT NULL);
```

EXAMPLE CONT.

- Let's say we want to transfer \$100 from Alice's account to Bob's. The corresponding SQL commands can be written as follows:

```
UPDATE accounts  
SET balance = balance - 100.00  
WHERE name = 'Alice';
```

```
UPDATE accounts  
SET balance = balance + 100.00  
WHERE name = 'Bob';
```


SAVEPOINT EXAMPLE

- Returning to the bank database, suppose we take \$ 100 from Alice's account, add it to Bob's account, and suddenly it turns out that the money needed to be transferred to Wally. In this case, we can apply savepoints:

```
BEGIN;  
UPDATE accounts  
SET balance = balance - 100.00  
WHERE name = 'Alice';  
SAVEPOINT my_savepoint;  
UPDATE accounts  
SET balance = balance + 100.00  
WHERE name = 'Bob';  
-- Wrong step. Needed to be cancelled for giving  
money to Wally  
ROLLBACK TO my_savepoint;  
UPDATE accounts  
SET balance = balance + 100.00  
WHERE name = 'Wally';  
COMMIT;
```