## **✅ Can You Run VACUUM Without Active Connections?**

### **🔸 Short Answer:**

**Yes**, you can run VACUUM **without downtime** and **even when there are active connections** — **if it's a regular VACUUM**.

## **✅ Types of VACUUM and Downtime Requirements**

| **Command** | **Blocks Queries?** | **Needs Downtime?** | **Notes** |
| --- | --- | --- | --- |
| VACUUM | ❌ No | ❌ No | Safe for production, runs in background |
| VACUUM ANALYZE | ❌ No | ❌ No | Safe, also updates statistics |
| VACUUM FULL | ✅ Yes (Locks Table) | ✅ Yes or Maintenance Window | Rewrites the table, **requires exclusive lock** |

### **🔍 Real Production Guidance:**

* ✅ Use **VACUUM** or **VACUUM ANALYZE** regularly — even while users are connected.
* ⚠️ Avoid **VACUUM FULL** during business hours. It locks the entire table and **blocks reads/writes**.

"VACUUM and VACUUM ANALYZE can be safely run in production environments without downtime, as they do not block user queries. Only VACUUM FULL requires exclusive locks and should be scheduled during maintenance windows."

##################################################################################

### **🔍 What is VACUUM VERBOSE?**

* VACUUM VERBOSE runs the vacuum operation **and prints detailed output** about what it’s doing.
* It’s mostly used for **debugging, monitoring, or learning** purposes.

### **🧪 Example Command:**

sql

CopyEdit

VACUUM VERBOSE my\_table;

You’ll see output like:

pgsql

CopyEdit

INFO: vacuuming "public.my\_table"

INFO: index "my\_table\_pkey" now contains 100000 row versions in 1500 pages

INFO: "my\_table": removed 5000 dead row versions in 100 pages

### **🔧 What It Shows:**

| **Info Output** | **Meaning** |
| --- | --- |
| Removed dead tuples | How many outdated rows were cleaned |
| Pages scanned | How much of the table was touched |
| Indexes cleaned | Info about index maintenance |
| Dead row versions | Helps you detect bloat |

### **🧠 When to Use VACUUM VERBOSE**

| **Situation** | **Why Use It** |
| --- | --- |
| 🧪 Debugging autovacuum issues | To see how many dead tuples are removed |
| 🧹 Manual cleanup | To confirm what is being vacuumed |
| 🎓 Learning/test environments | To understand how vacuum works |

### **⚠️ Note:**

* **VACUUM VERBOSE generates a lot of output**, especially on large tables.
* Avoid using it frequently in production unless you're diagnosing something specific.

### **🧠 Interview Tip:**

"VACUUM VERBOSE is helpful for analyzing vacuum activity in detail, especially when diagnosing performance or bloat issues."

############################################################################

## **✅ What Happens if a Logical Replication Slot Fails?**

### **🔴 If a replication slot fails or is inactive:**

* **WAL files are retained** on the primary server.
* If the slot is inactive for too long and the **replica doesn’t catch up**, the **WAL directory can fill up**.
* This may lead to:  
  + Disk space exhaustion on the primary.
  + **PostgreSQL crash** if disk becomes full.
  + Replication **lag or failure**.

## **🔍 Symptoms:**

* Disk usage increases abnormally on the primary (pg\_wal directory).
* pg\_stat\_replication shows **no activity** for the slot.
* Replica doesn’t receive new changes.

## **✅ How to Check Slot Status:**

sql

CopyEdit

SELECT \* FROM pg\_replication\_slots;

Look at:

* active: whether the slot is in use.
* restart\_lsn: the oldest LSN still needed by the slot (used for WAL retention).

## **🔧 How to Fix or Restore the Slot**

### **🛠️ Step 1: Identify the Problem**

Check if:

* The subscriber (replica) is down
* Network issues exist
* The replication process is broken

### **🛠️ Step 2: If You Can Restart the Subscriber**

* Restart the **subscription** on the replica:

sql

CopyEdit

ALTER SUBSCRIPTION your\_subscription\_name ENABLE;

* Or drop and recreate the subscription:

sql

CopyEdit

DROP SUBSCRIPTION your\_subscription\_name;

-- then recreate it using CREATE SUBSCRIPTION

### **🛠️ Step 3: If the Slot is Broken or Corrupted**

You may need to:

**Drop the old slot**:  
  
 sql  
CopyEdit  
SELECT pg\_drop\_replication\_slot('your\_slot\_name');

1. **Recreate the subscription from scratch**:  
   * This will auto-create a new replication slot.

On the **replica**:  
  
 sql  
CopyEdit  
CREATE SUBSCRIPTION your\_subscription

CONNECTION 'host=primary\_host dbname=your\_db user=rep\_user password=rep\_pass'

PUBLICATION your\_publication

WITH (copy\_data = true);

## **🚨 Best Practice to Prevent This**

* Monitor pg\_replication\_slots and pg\_stat\_replication.
* Set WAL retention properly (wal\_keep\_size, max\_slot\_wal\_keep\_size).
* Consider dropping unused or inactive slots to free space.

## **🧠 Interview Summary Line:**

"If a logical replication slot fails or stays inactive, it causes WAL accumulation and may lead to disk space issues on the primary. To fix this, either restart the subscriber, recreate the subscription, or drop and recreate the slot."

###############################################################################

## **✅ First, What Are Streaming Replication Slots?**

* **Replication slots** in streaming replication ensure that **WAL files are retained** on the primary until the **standby server has received them**.
* They prevent WAL files from being deleted **before** the replica has applied them.

## **🔴 What Happens If a Replication Slot Fails?**

### **🔹 “Fails” = Replica using the slot:**

* **Goes down**
* **Loses connectivity**
* **Is removed or reconfigured**
* **Never reconnects to the slot**

### **🔸 Result:**

1. **WAL files start piling up** on the primary server (in pg\_wal/ directory).
2. Disk space can fill up rapidly.
3. PostgreSQL may **crash** if disk becomes full.
4. The **replication slot becomes inactive**, but the primary still retains WAL files for it.
5. New replicas **can't use the same slot** if it's tied to a broken one.

## **📊 How to Check Slot and Replica Status**

sql

CopyEdit

-- Check all replication slots

SELECT slot\_name, plugin, slot\_type, active, restart\_lsn, confirmed\_flush\_lsn

FROM pg\_replication\_slots;

-- Check replica activity

SELECT \* FROM pg\_stat\_replication;

## **⚠️ If Slot is Inactive Too Long**

### **✅ Fix Option 1: If Replica Is Coming Back Online**

* Just **restart the standby**, it will reconnect using the same slot and catch up.
* Ensure WAL retention (wal\_keep\_size, max\_slot\_wal\_keep\_size) is enough to retain WALs.

### **✅ Fix Option 2: If Replica Is Permanently Gone**

Drop the old slot:  
  
 sql  
CopyEdit  
SELECT pg\_drop\_replication\_slot('your\_slot\_name');

* This **releases WAL file retention**, freeing disk space.
* But be careful — **you lose the replica’s position**, so it may need to be rebuilt from base backup.

## **🛠 How to Recreate a Broken Standby (Replica)**

Take a fresh base backup:  
  
 bash  
CopyEdit  
pg\_basebackup -h primary\_host -D /var/lib/pgsql/data -U replicator -Fp -Xs -P

Recreate the replication slot (optional if using physical slots):  
  
 sql  
CopyEdit  
SELECT \* FROM pg\_create\_physical\_replication\_slot('standby1\_slot');

1. Configure primary\_conninfo and primary\_slot\_name in postgresql.conf or standby.signal.

## **🧠 Interview One-Liner:**

"If a replication slot fails in streaming replication, WAL files accumulate on the primary, risking disk space. You must either bring the replica back online or drop the unused slot to avoid crash due to WAL bloat."

################################################################################

## **🎯 Scenario:**

* You have a **3-node PostgreSQL cluster**:  
  + 1 **Primary**
  + 2 **Standbys**
* **No automatic failover tool** (like Patroni, repmgr, etc.)
* Primary goes **down unexpectedly**
* You need to **manually promote** one of the standbys.

## **✅ How Will You Promote a Standby?**

You’ll promote one of the two standbys manually using:

bash

CopyEdit

pg\_ctl promote -D /var/lib/pgsql/data

or

sql

CopyEdit

SELECT pg\_promote();

## **❓ Which Standby Will You Choose to Promote?**

You should **choose the standby that is MOST UP-TO-DATE**.

## **🔍 How to Decide Which Standby Is Most Up-To-Date**

Run this query on both standbys:

sql

CopyEdit

SELECT pg\_last\_wal\_receive\_lsn(), pg\_last\_wal\_replay\_lsn();

| **Function** | **Description** |
| --- | --- |
| pg\_last\_wal\_receive\_lsn() | WAL received from primary |
| pg\_last\_wal\_replay\_lsn() | WAL applied/replayed |

Compare the pg\_last\_wal\_replay\_lsn() between the two standbys — **the higher LSN is more current**.

## **✅ Steps to Perform Manual Failover:**

1. **Login to both standbys**

Run:  
  
 sql  
CopyEdit  
SELECT pg\_last\_wal\_replay\_lsn();

1. Identify the **most up-to-date standby**

Promote it:  
  
 bash  
CopyEdit  
pg\_ctl promote -D /path/to/data

1. Update application connection strings or HA proxy to point to the new primary.
2. Reconfigure the remaining standby to follow the **new primary**.

## **🧠 Why Pick the Most Recent Standby?**

* To **avoid data loss**
* The one with the most recent LSN has **replayed the most WAL changes**
* If you promote an out-of-date standby, you risk **data inconsistency**

## **🧠 Interview One-Liner:**

"In a manual failover situation without auto tools, I promote the standby with the highest pg\_last\_wal\_replay\_lsn() to minimize data loss and ensure consistency."

#############################################################################

**if memory is 100GB can you assign work memory to 1 GB ?**

### **✅ Short Answer:**

Yes, you **can** assign work\_mem = 1GB in PostgreSQL \*\*but it is **not recommended globally** — unless you are **very sure** of your workload and memory limits.

### **📌 What is work\_mem?**

* It is the **amount of memory** used **per operation** (like sort, hash join, etc.)
* It is **allocated per query and per operation**, not per user or session.

### **❗ Important Consideration:**

If you set:

conf

CopyEdit

work\_mem = 1GB

And if a query uses 10 sort operations (common in joins or subqueries), it could use:

graphql

CopyEdit

1GB × 10 = 10GB per query

With **100 concurrent queries**, memory usage could shoot to:

CopyEdit

1GB × 10 × 100 = 1000GB (1TB)

Which is **10× your physical memory** — leading to swapping, slow performance, or system crash.

### **✅ Best Practices:**

| **Scenario** | **Recommended Setting** |
| --- | --- |
| Production (Global) | 4MB to 64MB |
| Session-level tuning for big queries | 128MB–1GB (with caution) |
| Specific query session | Use SET work\_mem before the query |

Example:

sql

CopyEdit

SET work\_mem = '512MB';

SELECT \* FROM large\_table ORDER BY col1;

### **🧠 Interview One-Liner:**

"While I can assign work\_mem = 1GB, I usually avoid it globally because it's per operation and can lead to memory overuse. Instead, I tune it per session or query based on workload needs."

##################################################################################

## **🎯 Scenario:**

You're using **streaming replication** in PostgreSQL.  
 Suddenly, there is a **replication lag of 2 hours** on the standby.

## **✅ What Is the Impact?**

| **🔴 Problem** | **⚠️ Impact** |
| --- | --- |
| Replication lag of 2 hours | Standby is **2 hours behind** primary |
| Application using standby for reads | Will serve **stale/outdated data** |
| Switchover or Failover happens | May result in **data loss** or **split brain** |
| WAL disk on primary fills up | Risk of **disk full / crash** if WALs aren't recycled |

## **🧪 How to Identify Replication Lag**

### **🔎 Query on Primary:**

sql

CopyEdit

SELECT pid, application\_name, state, write\_lag, flush\_lag, replay\_lag

FROM pg\_stat\_replication;

* replay\_lag: actual delay in applying WALs
* flush\_lag: delay in flushing WAL to disk
* write\_lag: delay in receiving WAL

If replay\_lag shows ~2 hours, standby is very behind.

### **🔎 On Standby:**

Compare the replay location with current WAL location of the primary.

sql

CopyEdit

-- On standby

SELECT pg\_last\_wal\_replay\_lsn();

-- On primary

SELECT pg\_current\_wal\_lsn();

Use this to measure how far behind the standby is (in bytes), or convert LSN diff to time.

### **Optional (Time-based Lag):**

sql

CopyEdit

-- On standby

SELECT now() - pg\_last\_xact\_replay\_timestamp() AS replication\_lag;

This shows **actual time lag**, e.g., 02:00:00.

## **🛠️ How to Fix the Replication Lag**

### **🔹 Step 1: Identify the Bottleneck**

* Network issue between primary and standby?
* Standby is under heavy CPU/disk load?
* Disk full on standby?
* Standby is stuck or paused?

### **🔹 Step 2: Check Standby Logs**

Look for errors in:

bash

CopyEdit

/var/lib/pgsql/data/log/

Check for:

* Connection timeout
* WAL archive issues
* Disk write failures

### **🔹 Step 3: Actions You Might Take**

| **Problem** | **Solution** |
| --- | --- |
| Network issues | Restart replication or fix network |
| High I/O on standby | Reduce load or increase disk throughput |
| WALs not applying | Restart standby or manually apply WALs |
| Very large transactions | Identify long-running queries and tune them |
| Disk full | Free up space or add storage |

### **⚠️ If Lag is Too Large and Standby Can’t Catch Up**

You may need to:

1. Drop the standby
2. Take fresh base backup
3. Re-initialize replication

## **🧠 Interview One-Liner:**

"A 2-hour replication lag means the standby is behind and could serve stale data. I’d check pg\_stat\_replication, logs, and disk/network status to find the root cause and resolve it before it causes data loss or disk overflow."

## **✅ What is Point-in-Time Recovery (PITR)?**

PITR allows you to restore your PostgreSQL database to **any specific time or transaction** — like just **before a mistake or failure**.

It’s based on:

* **Base backup** (taken via pg\_basebackup or file-level)
* **WAL (Write-Ahead Log)** archiving

## **🧰 Requirements:**

1. **Base backup**
2. **WAL archive files** between backup time and recovery time
3. **archive\_mode = on** and archive\_command must be enabled during runtime
4. wal\_level = replica (or logical if logical decoding is used)

## **🧪 Step-by-Step: Perform PITR**

### **✅ Step 1: Configure PostgreSQL for Archiving (before any recovery)**

Ensure your postgresql.conf has:

conf

CopyEdit

archive\_mode = on

archive\_command = 'cp %p /path\_to\_wal\_archive/%f'

wal\_level = replica

Take a base backup (use pg\_basebackup or manual method):

bash

CopyEdit

pg\_basebackup -D /backups/my\_backup -Ft -z -P -U postgres

### **✅ Step 2: Simulate Failure or Need for PITR**

For example: accidental delete or data corruption

### **✅ Step 3: Restore Base Backup**

bash

CopyEdit

rm -rf /var/lib/pgsql/data/\*

tar -xvf /backups/my\_backup/base.tar -C /var/lib/pgsql/data/

### **✅ Step 4: Create recovery.signal and Configure postgresql.auto.conf**

Edit postgresql.auto.conf or postgresql.conf:

conf

CopyEdit

restore\_command = 'cp /path\_to\_wal\_archive/%f %p'

Create recovery.signal file to trigger recovery:

bash

CopyEdit

touch /var/lib/pgsql/data/recovery.signal

And set recovery target (in postgresql.conf or postgresql.auto.conf):

conf

CopyEdit

recovery\_target\_time = '2025-05-13 10:30:00' # Example time

Or use:

conf

CopyEdit

recovery\_target\_xid = '12345678'

recovery\_target\_name = 'my\_restore\_point'

You can also specify:

conf

CopyEdit

recovery\_target\_action = 'pause' # pause, promote, or shutdown

### **✅ Step 5: Start PostgreSQL**

bash

CopyEdit

pg\_ctl start -D /var/lib/pgsql/data

PostgreSQL will restore to the exact point you specified using WALs.

You can check logs:

pgsql

CopyEdit

LOG: recovery stopping after reaching consistency at 0/3000020

### **✅ Step 6: Promote the Server**

If you used recovery\_target\_action = 'pause', manually promote:

bash

CopyEdit

pg\_ctl promote -D /var/lib/pgsql/data

This turns the recovered system into a standalone, writable primary.

## **🧠 Interview One-Liner:**

"To perform PITR, I restore from a base backup, apply WAL files using restore\_command, and set a recovery\_target\_time. PostgreSQL replays WALs to that point, and then I promote the server."

####################################################################################

**how to create index in production enviornment**

Creating an index in a **production environment** needs careful planning to ensure minimal disruption to ongoing operations, especially for large databases. Here’s how to approach it:

### **✅ Steps to Create an Index in Production Environment:**

1. **Analyze the Query and Table:**
   * **Identify the most queried columns** in your application. Create indexes on columns that are frequently involved in:  
     + **WHERE conditions** (filters)
     + **JOIN conditions**
     + **ORDER BY clauses**
2. **Use a CONCURRENTLY Option:**
   * In a production environment, creating an index with the CONCURRENTLY option ensures **no locks** on the table, allowing for continued read and write operations.

sql  
CopyEdit  
CREATE INDEX CONCURRENTLY idx\_column\_name ON table\_name (column\_name);

* + **Benefit**: This avoids the **exclusive lock** that the standard CREATE INDEX would impose, so your application can continue to access the table while the index is being built.
  + **Caveat**: The CONCURRENTLY option takes longer than a regular index creation.

1. **Check for Existing Indexes:**
   * Before creating a new index, **check if an index already exists** on the column to avoid duplication:

sql  
CopyEdit  
SELECT \* FROM pg\_indexes WHERE tablename = 'your\_table\_name';

1. **Avoid Heavy Traffic Windows:**
   * If possible, **schedule index creation** during periods of low traffic to minimize the load on the server.
2. **Monitor Resource Usage:**
   * Keep an eye on:  
     + **CPU** usage
     + **Disk I/O** and **storage** during index creation
   * If the table is large, ensure you have enough disk space for the index itself.
3. **Verify the Index After Creation:**

After the index is created, verify that it's being used by your queries:  
  
 sql  
CopyEdit  
EXPLAIN ANALYZE SELECT \* FROM your\_table WHERE column\_name = 'value';

### **✅ Example:**

sql

CopyEdit

-- Create index concurrently on column\_name of the table customers

CREATE INDEX CONCURRENTLY idx\_customer\_email ON customers (email);

This will:

* Build an index on the email column of the customers table.
* Allow the table to remain fully operational (no heavy locking).

### **✅ If Using pg\_repack:**

For large tables with existing fragmentation or bloating, you might use **pg\_repack** to create indexes without locking the table:

bash

CopyEdit

pg\_repack -t your\_table\_name --create-index

This creates the index concurrently and optimizes table storage as well.

### **⚠️ Things to Watch Out For:**

* **Performance Impact**: Index creation, especially on large tables, can still cause some performance degradation due to increased I/O. Always monitor the database’s **resource usage**.
* **Transaction Blocking**: While CONCURRENTLY avoids locking reads, it might still have **performance implications** on writes.

### **🧠 Interview One-Liner:**

"To create an index in a production environment, I use CREATE INDEX CONCURRENTLY to ensure minimal locking and avoid performance disruptions while the index is being built."

##################################################################################

## **🔍 Difference Between INDEX and REINDEX in PostgreSQL**

| **Concept** | **INDEX** | **REINDEX** |
| --- | --- | --- |
| 🔧 **Purpose** | Creates a new index on a table or column | Rebuilds an existing index |
| 🆕 **When Used** | At the time of table design or to speed up queries | When index becomes bloated or corrupted |
| 🔁 **Effect** | Adds a new structure to speed up queries | Recreates the index from scratch, removing bloat |
| 💾 **Storage Impact** | Consumes space for the new index | Replaces the existing index |
| 📈 **Performance** | Helps queries run faster | Restores performance if existing index is inefficient |
| 🧹 **Used in Maintenance?** | No | Yes, part of routine DB maintenance |
| ⚠️ **Locking** | Minimal locking when creating on new column | Locks the table or index during rebuild (blocking) |

## **✅ Example of Creating an Index**

sql

CopyEdit

CREATE INDEX idx\_customer\_name ON customers (name);

Creates a new index to speed up lookups on name.

## **🔁 Example of Reindexing**

sql

CopyEdit

REINDEX INDEX idx\_customer\_name;

-- Or

REINDEX TABLE customers;

Rebuilds the index to:

* Remove **bloat**
* Fix **corruption**
* Improve performance

## **💡 When to Use REINDEX**

* If pg\_stat\_user\_indexes.idx\_scan is high but performance is still poor
* If pgstatindex shows high **bloat**
* After an upgrade or corruption
* When VACUUM or ANALYZE are not enough

## **🧠 Interview One-Liner:**

"INDEX creates a new index to improve query performance, while REINDEX rebuilds an existing index to remove bloat or corruption and restore performance."

###################################################################################