PostgreSQL

Advanced Postgres

Surendra Panpaliya

Agenda

Transactions and Concurrency Control

ACID properties

Transaction isolation levels

Handling deadlocks

Agenda

Monitoring and Maintenance

Monitoring tools

Routine maintenance tasks

Logging and troubleshooting

Transactions and Concurrency Control

Ensure data integrity

Consistency

when multiple transactions

occur simultaneously

Transactions

A Sequence of operations

performed as

A single logical unit of work

Transactions

Follow the ACID properties

Atomicity

Consistency

Isolation

Durability

Atomicity

Ensures that all operations

within a transaction are completed successfully

If any operation fails,

the entire transaction is rolled back

Consistency

Ensures that

A transaction brings

the database from

one valid state to another

Isolation

ENSURES THAT

CONCURRENT TRANSACTIONS

DO NOT AFFECT EACH OTHER.

Durability

Ensures that

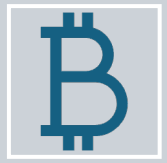
Once a transaction is committed,

It remains so,

even in the event of a system failure.

Transaction Commands

BEGIN



Starts a new transaction.

Transaction Commands

COMMIT

Ends the current transaction and

makes all changes permanent.

Transaction Commands

ROLLBACK

Ends the current transaction and

discards all changes.

Transaction Example

BEGIN;

INSERT INTO patients (name, ssn) VALUES ('Jane Doe', '123-45-6789');

UPDATE patients SET name = 'Jane Smith' WHERE ssn = '123-45-6789';

COMMIT;

Concurrency Control

PostgreSQL uses

Multi-Version Concurrency Control (MVCC)

to handle concurrency

Concurrency Control

MVCC allows multiple transactions

to read and write data

without locking each other out,

ensuring high performance and

Isolation Levels

Supports four isolation levels

Read Uncommitted

Read Committed

Repeatable Read

Serializable

Read Uncommitted

Lowest isolation level.

Transactions can see

changes made by

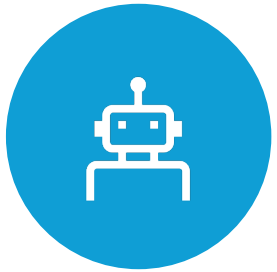
other uncommitted transactions.

PostgreSQL treats this level as Read Committed

Read Committed

DEFAULT LEVEL.

TRANSACTIONS ONLY SEE



CHANGES COMMITTED

BEFORE THE QUERY BEGAN.

Repeatable Read

Ensures that

if a transaction reads a row,

it will see the same data

if it reads the row again

within the same transaction.

Serializable

Highest isolation level.

Transactions are executed

in a way that their effect is

the same as if

they were executed serially

Example: Setting Isolation Level

SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;

BEGIN;

-- Transaction operations

COMMIT;

Multi-Version Concurrency Control

MVCC allows

Multiple versions

of data to exist

simultaneously

Multi-Version Concurrency Control

Each transaction sees

a snapshot of the database

at a specific point in time,

ensuring consistent reads.

Visibility

Each transaction works

with a snapshot,

making changes invisible

to other transactions

until they are committed.

Version Management

Old versions of data	are retained until	no longer needed
by any transaction,	then cleaned up	by the VACUUM process.

Locking Mechanisms

While MVCC reduces

the need for locking,

PostgreSQL still uses locks

to manage concurrent access

Locking Mechanisms

Row Locks:

Acquired when

a row is modified.

Released after

the transaction ends.

Locking Mechanisms

Table Locks:

Acquired for operations

affecting an entire table

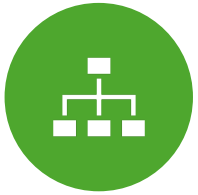
Locking Mechanisms

Advisory Locks:

Explicit locks that

applications can use

to enforce complex



business rules.

Example: Row Lock

```
BEGIN;

SELECT * FROM patients WHERE ssn = '123-45-6789' FOR UPDATE;

-- Perform some operations

COMMIT;
```

Deadlock Detection

Deadlocks occur

when two or more transactions

wait for each other

to release locks.

Deadlock Detection

PostgreSQL detects and

resolves deadlocks

by aborting one of

the transactions.

Example: Simulating a Deadlock

```
-- Session 1
BEGIN;
UPDATE patients
SET name = 'John Doe’
WHERE ssn = '123-45-6789';
```

Example: Simulating a Deadlock

```
-- Session 2
BEGIN;
UPDATE patients SET name = 'Jane Doe’
WHERE ssn = '987-65-4321';
UPDATE patients SET name = 'Jane Smith’
WHERE ssn = '123-45-6789';  -- This will wait
```

Example: Simulating a Deadlock

```
-- Session 1
UPDATE patients SET name = 'John Smith’
WHERE ssn = '987-65-4321’;
-- This will cause a deadlock
COMMIT;
```

ACID properties

Surendra Panpaliya

ACID properties in healthcare domain

CRUCIAL FOR

ENSURING RELIABLE

CONSISTENT DATABASE

TRANSACTIONS

1. Atomicity

Atomicity ensures that

a series of operations

within a transaction

are treated as a single unit.

1. Atomicity

Either all operations are

completed successfully, or

none are applied.

Example in Healthcare

Scenario: Updating patient records.

Operations:

A healthcare application updates a

patient's address and

phone number simultaneously.

Importance

Ensures that both the address and

phone number are updated together.

If one update fails, both are rolled back

to maintain data integrity.

SQL Example

```
BEGIN;
-- Update patient address
UPDATE patients
SET address = '456 Elm St'
WHERE patient_id = 123;
```

SQL Example

```
-- Update patient phone number
UPDATE patients
SET phone_number = '555-1234'
WHERE patient_id = 123;

COMMIT;
```

2. Consistency

Consistency ensures that

a transaction brings

the database from

one valid state to another,

maintaining database rules,

such as integrity constraints.

Example in Healthcare

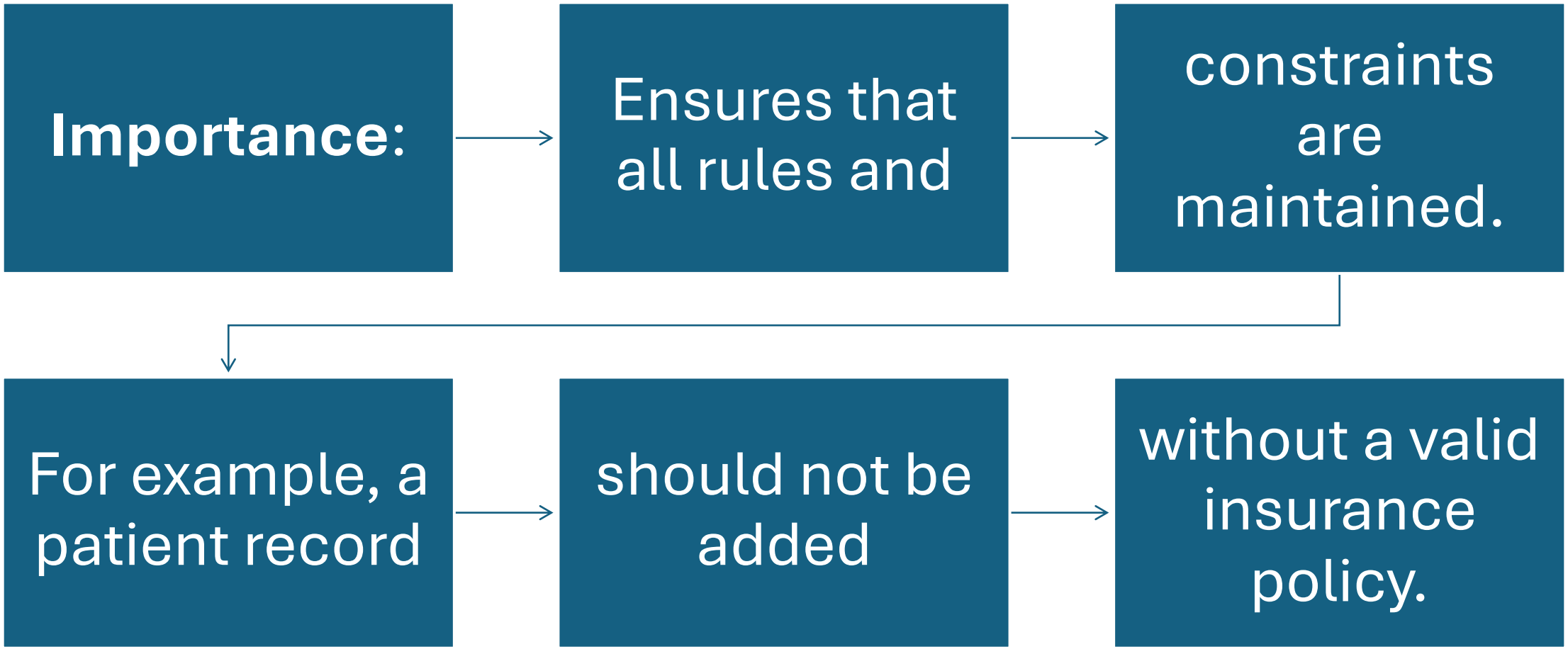
Scenario: Adding a new patient record.

Operations:

Inserting patient data and

ensuring a valid insurance policy reference.

Example in Healthcare



SQL Example

```
BEGIN;

-- Insert patient data

INSERT INTO patients

(patient_id, name, ssn, insurance_policy_id)

VALUES (124, 'Alice Brown', '987-65-4321', 789);
```

SQL Example

```
-- Insert insurance policy data
INSERT INTO insurance_policies (policy_id, patient_id, coverage)
VALUES (789, 124, 'Full Coverage');
COMMIT;
```

3. Isolation

Isolation ensures that

transactions are executed independently

without interference.

Each transaction sees

a consistent view of the database.

Example in Healthcare

Scenario:

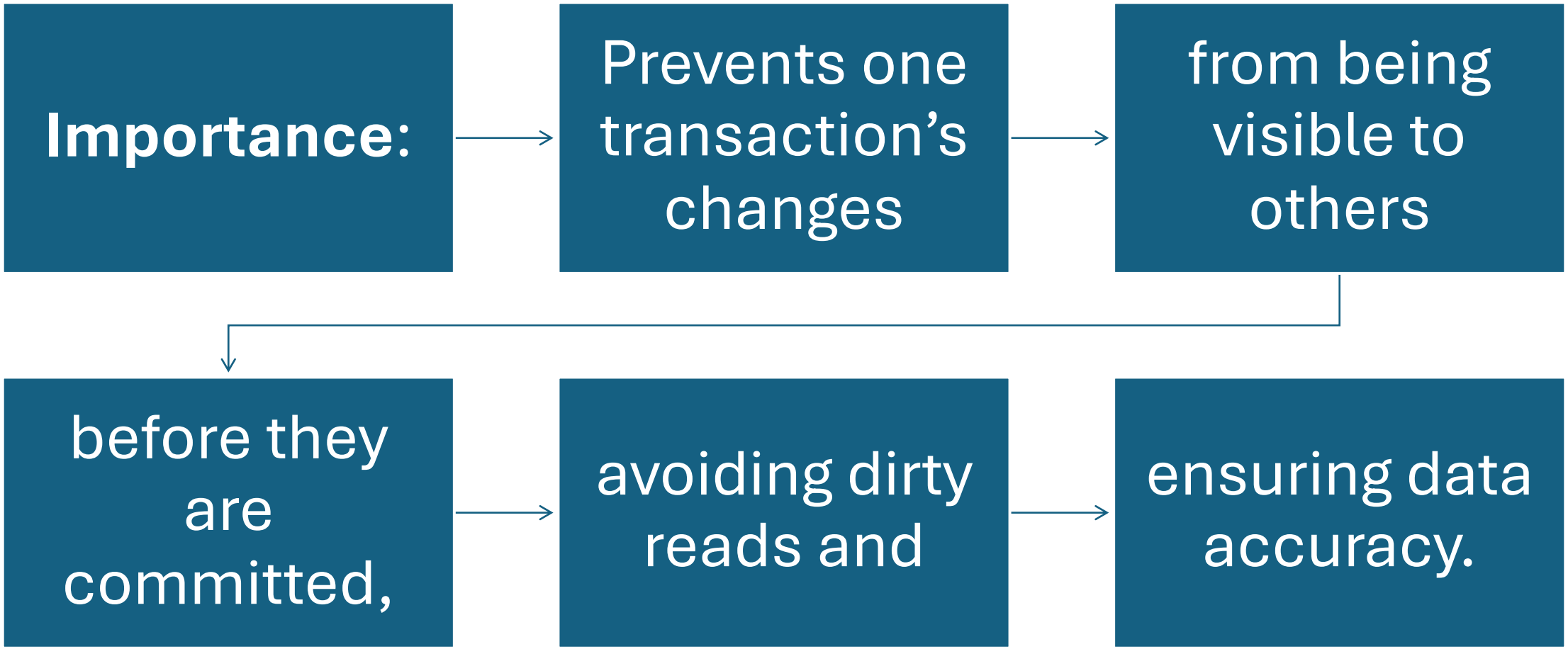
Concurrent access to patient records.

Operations:

One user updates patient data

while another queries the same data.

Example in Healthcare



SQL Example

```
-- Transaction 1
BEGIN;
UPDATE patients
SET address = '789 Maple St’
WHERE patient_id = 125;
```

SQL Example

```
-- Transaction 2 (executed concurrently)

BEGIN;
SELECT * FROM patients
WHERE patient_id = 125;
```

SQL Example

```
-- This transaction sees the old address
-- until Transaction 1 commits
COMMIT;

-- Transaction 1 commits
COMMIT;
```

4. Durability

Ensures once a

Transaction is committed,

Changes are permanent,

Even in the event

of a system failure

Example in Healthcare

Scenario:

Saving patient diagnosis records.

Operations:

Storing critical diagnosis information.

Importance

Ensures that once the diagnosis

information is recorded,

it is not lost even if

there is a system crash

SQL Example

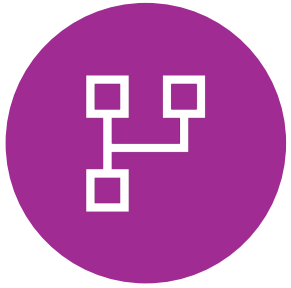
```
BEGIN;


-- Insert diagnosis data
INSERT INTO diagnoses (diagnosis_id, patient_id,
diagnosis_date, diagnosis)
VALUES (2001, 126, '2024-07-01', 'Hypertension');


COMMIT;
```

Transaction isolation levels

Ensure Data Integrity

Consistency

When multiple Transactions



occur concurrently

Transaction isolation levels

PostgreSQL supports

several isolation levels that

control the visibility

of changes made

by one transaction

to other concurrent transactions.

1. Read Uncommitted

Transactions can

read data changes

made by other

Uncommitted transactions.

1. Read Uncommitted

This level is susceptible to

Dirty reads

Non-repeatable reads

Phantom reads.

Phantom reads

Occur in database systems

when a transaction reads a set of rows

that satisfy a certain condition.

Phantom reads

Upon re-executing

the same read operation,

finds additional rows (phantoms)

that did not exist previously.

Use Case in Healthcare

Example:

Generally not recommended

due to the risk of reading uncommitted or dirty data,

which can lead to inconsistent patient records.

Use Case in Healthcare

Example:

Not used in PostgreSQL as

it treats Read Uncommitted as

Read Committed.

2. Read Committed (Default Level)

A transaction sees only

committed changes

made by other transactions.

It does not see

uncommitted changes.

2. Read Committed (Default Level)

This level avoids

dirty reads

but can still encounter

Non-repeatable reads and

Phantom reads.

Use Case in Healthcare

Example:

A doctor updates a patient’s record,

while another transaction reads the patient’s data.

Reader will see only Committed changes.

Example

```
-- Session 1: Update patient record
BEGIN;
UPDATE patients
SET address = '456 Elm St’
WHERE patient_id = 101;
```

Example

```
 -- Session 2: Read patient record
BEGIN;
SELECT * FROM patients WHERE patient_id = 101;  -- Sees the old address until Session 1 commits
COMMIT;


-- Session 1 commits
COMMIT;
```

3. Repeatable Read

Ensures that if a transaction reads a row,

it will see the same data

throughout the transaction,

preventing non-repeatable reads

but still allowing phantom reads.

Use Case in Healthcare

Example:

Ensuring consistent reads of patient data

during a complex query or report generation.

Useful for billing or auditing processes.

Example

```
-- Session 1: Start transaction
BEGIN;
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;

-- Session 1: Read patient record
SELECT * FROM patients WHERE patient_id = 102;
```

Example

```
-- Session 2: Update patient record
BEGIN;
UPDATE patients
SET address = '789 Maple St’
WHERE patient_id = 102;
COMMIT;
```

Example

```
-- Session 1: Read patient record again
SELECT * FROM patients
WHERE patient_id = 102;
 -- Sees the old address until Session 1 commits
COMMIT;
```

4. Serializable

The highest isolation level,

where transactions are executed in a way

that their effect is the same as

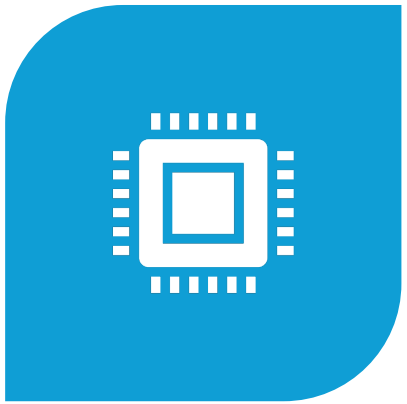
if they were executed serially,

one after another.

4. Serializable

PREVENTS DIRTY READS,

NON-REPEATABLE READS, AND



PHANTOM READS.

Use Case in Healthcare

Example:

Critical operations

where absolute data consistency is required,

such as updating patient medication records or

processing insurance claims.

Example

```
-- Session 1: Start transaction
BEGIN;
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;

-- Session 1: Read patient record
SELECT * FROM patients WHERE patient_id = 103;
```

Example

```
-- Session 2: Attempt to update patient record
BEGIN;
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;

UPDATE patients
SET address = '1010 Pine St’
WHERE patient_id = 103;
```

Example

```
-- This will wait or fail if Session 1 has not yet committed
COMMIT;

-- Session 1: Read patient record again and commit
SELECT * FROM patients WHERE patient_id = 103;
COMMIT;
```

Handling deadlocks

Surendra Panpaliya

Handling deadlocks

INVOLVES PREVENTING, DETECTING

RESOLVING SITUATIONS

WHERE TWO OR MORE TRANSACTIONS

ARE WAITING INDEFINITELY

FOR ONE ANOTHER TO RELEASE LOCKS.

Handling deadlocks

In healthcare systems,

deadlocks can lead to system downtime or

data inconsistencies, which can

have severe consequences.

1. Understanding Deadlocks

A deadlock occurs when two or more transactions

hold locks that the other transactions need,

creating a cycle of dependencies

that prevents any of the transactions from proceeding.

Example in Healthcare

Scenario:

Two transactions are updating patient records and

insurance claims simultaneously,

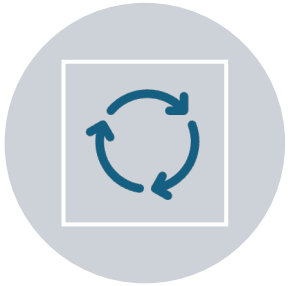
leading to a deadlock.

2. Preventing Deadlocks Strategies

Consistent Lock Ordering:

Ensure that transactions acquire locks

in a consistent order



to prevent cyclical dependencies.

2. Preventing Deadlocks Strategies

MINIMIZE LOCK DURATION:

KEEP TRANSACTIONS SHORT AND

LOCKS HELD FOR THE MINIMAL AMOUNT OF TIME.

2. Preventing Deadlocks Strategies

Use Appropriate Isolation Levels:

Choose isolation levels

that balance consistency

and concurrency.

Scenario

Updating patient records and

insurance claims in a specific order.

Scenario

```
-- Session 1: Update patient record, then update insurance claim
BEGIN;
UPDATE patients SET address = '456 Elm St’
WHERE patient_id = 201;

UPDATE insurance_claims SET status = 'processed’
WHERE claim_id = 301;
COMMIT;
```

Scenario

```
-- Session 2: Update patient record, then update insurance claim
BEGIN;
UPDATE patients SET phone_number = '555-6789’
WHERE patient_id = 202;

UPDATE insurance_claims SET status = 'pending’
WHERE claim_id = 302;
COMMIT;
```

3. Detecting Deadlocks

PostgreSQL automatically detects deadlocks and

aborts one of the transactions to break the cycle.

You can configure logging to monitor deadlocks.

3. Detecting Deadlocks

Configuration:

Enable Deadlock Logging: Modify postgresql.conf.

log_min_error_statement = 'error'

Monitor Logs: Regularly check the PostgreSQL logs for deadlock messages.

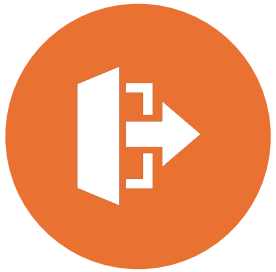
Monitoring and Maintenance

Monitoring tools (pg_stat_activity, pg_stat_statements)

Routine maintenance tasks (VACUUM, ANALYZE)

Logging and troubleshooting

Monitoring and Maintenance



CRUCIAL FOR

ENSURING THE PERFORMANCE,

RELIABILITY, AND STABILITY OF A

POSTGRESQL DATABASE.

Monitoring and Maintenance

Effective monitoring helps in

identifying and resolving issues

before they impact the system,

while regular maintenance

keeps the database running efficiently.

1. pg_stat_activity

This view shows the current activity in the database.

Example:

```
SELECT pid, usename, application_name,
client_addr, state, query
FROM pg_stat_activity;
```

2. pg_stat_database

Provides aggregate statistics for each database in the cluster.

Example:

```
SELECT datname, numbackends, xact_commit, xact_rollback, blks_read, blks_hit
FROM pg_stat_database;
```

3. pg_stat_user_tables

Shows access and usage statistics for tables.

Example:

SELECT relname, seq_scan, seq_tup_read, idx_scan, idx_tup_fetch, n_tup_ins, n_tup_upd, n_tup_del

FROM pg_stat_user_tables;

4. pg_stat_user_indexes

Provides statistics on indexes.

Example:

```
SELECT relname, indexrelname, idx_scan, idx_tup_read, idx_tup_fetch
FROM pg_stat_user_indexes;
```

5. pg_stat_bgwriter

Statistics about the background writer process, which is responsible for writing dirty pages to disk.

Example:

SELECT checkpoints_timed, checkpoints_req, buffers_checkpoint, buffers_clean, maxwritten_clean

FROM pg_stat_bgwriter;

6. pg_stat_statements

Tracks execution statistics of all SQL statements executed by the server.

Installation:

CREATE EXTENSION pg_stat_statements;

Example

```
SELECT query, calls, total_time, rows,
shared_blks_hit, shared_blks_read
FROM pg_stat_statements
ORDER BY total_time DESC
LIMIT 5;
```

Routine maintenance tasks

VACUUM and ANALYZE are essential

for maintaining the performance and

reliability of PostgreSQL databases

Routine maintenance tasks

Ensure that

the database operates efficiently

by reclaiming storage,

updating statistics for

the query planner, and

preventing database bloat.

1. VACUUM

Used to clean up

dead tuples

Rows that have been

updated or deleted

1. VACUUM

Helps in reclaiming storage

preventing table bloat,

which can significantly

degrade performance over time.

Types of VACUUM

Standard VACUUM:

Reclaims storage

occupied by

dead tuples.

Types of VACUUM

VACUUM FULL:

Rewrites the entire table to reclaim space and

shrink the table size.

It locks the table during the operation.

Routine Cleanup of Patient Records

Regular updates and

deletions of patient records

can create dead tuples,

requiring routine cleanup

to maintain performance.

VACUUM patients;

Reclaiming Space After Bulk Deletions

When large amounts of outdated data,

such as old patient records, are deleted,

running VACUUM FULL helps reclaim the space.

VACUUM FULL patients;

Configuration for Autovacuum

Autovacuum is a background process

that automatically vacuums tables

at regular intervals,

reducing the need

for manual intervention.

Configuration for Autovacuum

psql -U postgres

SHOW config_file;

C:\Program Files\PostgreSQL\16\data\postgresql.conf

Enable and configure autovacuum in postgresql.conf

autovacuum = on

autovacuum_max_workers = 3

autovacuum_naptime = 1min

autovacuum_vacuum_threshold = 50

autovacuum_analyze_threshold = 50

autovacuum_vacuum_scale_factor = 0.2

autovacuum_analyze_scale_factor = 0.1

2. ANALYZE

Command updates

the statistics used by

the PostgreSQL query planner

to determine the most efficient

way to execute queries.

Example Use Cases

Improving Query Performance for Patient Lookups

Frequent lookups of patient data
can benefit from updated statistics
to ensure optimal query plans.

ANALYZE patients;

Updating Statistics After Large Data Modifications

After bulk insertions, updates, or deletions
in tables like appointments or lab_results,
running ANALYZE ensures that
the query planner has the most recent statistics.

ANALYZE appointments;

Combining VACUUM and ANALYZE

To perform both VACUUM and ANALYZE
in a single command,
use VACUUM ANALYZE.
This reclaims storage and
updates statistics simultaneously.

VACUUM ANALYZE patients;

Logging and troubleshooting

Surendra Panpaliya

Logging

Effective logging helps in identifying issues,

tracking changes, and ensuring compliance

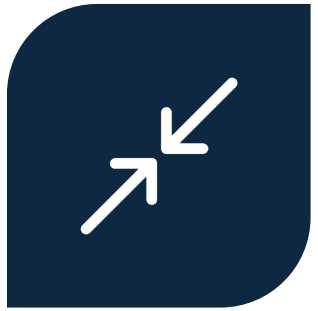
with regulatory requirements.

Troubleshooting

TROUBLESHOOTING TECHNIQUES ARE

ESSENTIAL FOR DIAGNOSING AND

RESOLVING PROBLEMS PROMPTLY



TO MINIMIZE DISRUPTIONS.

Logging in PostgreSQL

PostgreSQL provides

various logging options

to capture detailed information

about database activities,

errors, and performance.

Configuration for Logging

Logging settings are configured in the postgresql.conf file.

psql -U postgres

SHOW config_file;

C:\Program Files\PostgreSQL\16\data\postgresql.conf

Basic Logging Settings

```
logging_collector = on                  # Enable the collection of logs

log_directory = 'pg_log'                # Directory where log files are stored

log_filename = 'postgresql-%Y-%m-%d.log'   # Log file name pattern

log_rotation_age = 1d                   # Rotate log files daily

log_rotation_size = 100MB               # Rotate log files after 100MB

log_statement = 'all'                   # Log all SQL statements
```

Basic Logging Settings

```
log_duration = on              # Log the duration of each completed statement

log_line_prefix = '%m [%p] %q%u@%d '

# Log line prefix with timestamp, PID, user, and database

log_error_verbosity = default        # Default verbosity for error messages

log_min_duration_statement = 500

# Log statements that take longer than 500ms
```

Additional Logging Options

Auditing User Activities:

log_connections = on # Log each successful connection

log_disconnections = on # Log each disconnection

log_lock_waits = on

Log queries that wait for more than the specified amount of time

Troubleshooting in PostgreSQL

Effective troubleshooting involves

identifying the root cause of issues and

resolving them to ensure

minimal disruption

to healthcare operations.

Common Troubleshooting Scenarios

Identifying and Resolving Deadlocks:

Deadlocks can occur

when two or more transactions

hold locks that the other transactions need.

Detecting Deadlocks

Enable deadlock logging in postgresql.conf:

log_lock_waits = on

deadlock_timeout = 1s

Resolving Deadlocks

ANALYZE THE QUERIES

CAUSING THE DEADLOCK AND

OPTIMIZE THEM TO AVOID CONFLICTS.

ENSURE TRANSACTIONS

ACQUIRE LOCKS IN A CONSISTENT ORDER.

2. Addressing Slow Queries

Slow Queries

Can significantly impact

Performance of

Healthcare applications.

Example Query to Identify Slow Queries

```
SELECT query, state, wait_event_type, wait_event,

now() - query_start AS duration

FROM pg_stat_activity

WHERE state <> 'idle' AND

now() - query_start > interval '1 minute'

ORDER BY duration DESC;
```

Optimizing Slow Queries

1

Use indexes to speed up query execution.

2

Rewrite inefficient joins or subqueries.

3

Analyze and update statistics using ANALYZE.

3. Handling Connection Issues

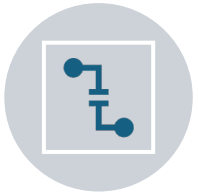
Connection issues can prevent users

from accessing critical healthcare data.

Example Log Analysis for Connection Issues:

grep 'connection' /var/lib/pgsql/pg_log/postgresql-*.log

Resolving Connection Issues

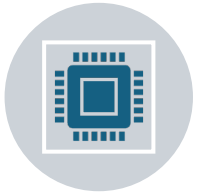


Check for network problems or firewall issues.

Increase the maximum number of connections

if the limit is reached.

Ensure the database server



has sufficient resources (CPU, memory).

4. Monitoring Database Performance

Regular monitoring helps

identify and address performance issues

before they affect users.

pg_stat_statements to Monitor Performance

```
SELECT query, calls, total_time, mean_time, rows,
shared_blks_hit, shared_blks_read
FROM pg_stat_statements
ORDER BY total_time DESC
LIMIT 10;
```

Surendra Panpaliya

Founder and CEO

GKTCS Innovations

https://www.gktcs.com