# Performance Tuning and Optimization


Instructor: <Trainer Name>

EDB

# Course Agenda

- Introduction
- SQL Tuning
- Performance Tuning
- Database Maintenance

# Module - 1
# Introduction

# Module Objectives

- EDB Portfolio
- Lab Setup - Prepare a Sample Database

# EDB Supported Databases

### Postgres

Open source Postgres

- EDB continues to be committed to advancing features in collaboration with the broader community

### Postgres Extended

EDB proprietary distribution for EDB Postgres Distributed use cases with Transparent Data Encryption

- SQL compatible with Postgres, extended for stringent availability and advanced replication needs
- Transparent Data Encryption
- Formerly known as 2ndQPostgres

### Postgres Advanced Server

EDB proprietary distribution with Transparent Data Encryption

- SQL compatible with Oracle, reduces effort to migrate applications and data to Postgres
- Transparent Data Encryption
- Additional value-add enterprise features

# Lab Setup Guidelines

- All the instructor demos and labs are based on Linux. Here are the machine requirements:
  - CentOS 7 with 2 GB RAM and 40 GB storage space is recommended
  - Latest version of EDB Postgres Advanced Server and Postgres Enterprise Manager(optional) installed
  - superuser access to Database and Operating System
  - EDB credentials for EDB Repos 2.0
  - Internet access for downloading EDB packages

# Prepare a Sample Database

- In the training materials provided by EnterpriseDB there is a script file **edbstore.sql** that can be executed using `edb-psql` to create a sample `edbstore` database. Here are the steps:
  - Download the **edbstore.sql** file and place in a directory which is accessible to the `enterprisedb` user
  - Login as `enterprisedb` OS user
  - Run the `edb-psql` command with the `-f` option to execute the **edbstore.sql** file and install all the sample objects required for this training

```
$ edb-psql -p 5444 -f edbstore.sql -d edb -U enterprisedb
```

    - Note - The above command will prompt for `edbuser` password. The password is `edbuser`

# Prepare a Sample Database (Continued)

- Connect to edbstore database using edbuser
- Verify the existence of the objects in psql by running `\d`

```
edbstore=> \d
                    List of relations
 Schema  |           Name            |   Type   |  Owner
---------+---------------------------+----------+---------
 edbuser | categories                | table    | edbuser
 edbuser | categories_category_seq   | sequence | edbuser
 edbuser | cust_hist                 | table    | edbuser
 edbuser | customers                 | table    | edbuser
 edbuser | customers_customerid_seq  | sequence | edbuser
 edbuser | dept                      | table    | edbuser
 edbuser | emp                       | table    | edbuser
 edbuser | inventory                 | table    | edbuser
 edbuser | job_grd                   | table    | edbuser
 edbuser | jobhist                   | table    | edbuser
 edbuser | locations                 | table    | edbuser
 edbuser | next_empno                | sequence | edbuser
 edbuser | orderlines                | table    | edbuser
 edbuser | orders                    | table    | edbuser
 edbuser | orders_orderid_seq        | sequence | edbuser
 edbuser | products                  | table    | edbuser
 edbuser | products_prod_id_seq      | sequence | edbuser
 edbuser | reorder                   | table    | edbuser
 edbuser | salesemp                  | view     | edbuser
(19 rows)
```

# Module Summary

- EDB Portfolio
- Lab Setup - Prepare a Sample Database

# Module 2
# SQL Tuning

# Module Objectives

- SQL queries planning and execution
- Find and tune slow running SQL
- Types of Indexes and their usage
- Writing an efficient SQL query

# Statement Processing - SQL Engine

- Check Syntax
- Call Traffic Cop
- Identify Query Type
- Command Processor if needed
- Break Query in Tokens

## Parse

## Optimize

- Planner generate Plan
- Uses Database Statistics
- Query Cost Calculation
- Choose best plan

- Execute Query based on query plan

## Execute

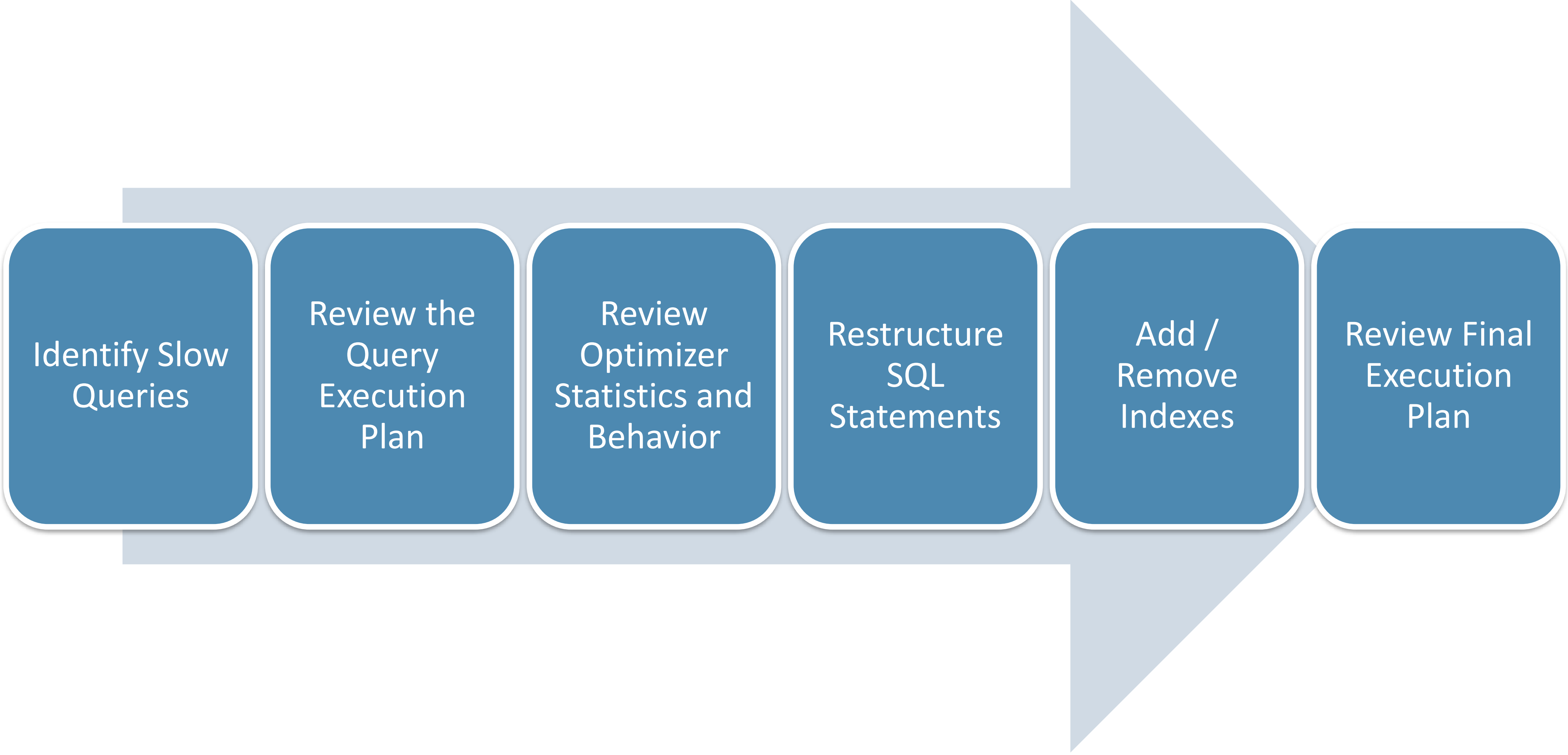
# Common Query Performance Issues

- Full table scans
- Bad SQL
- Sorts using disk
- Join orders
- Old or missing statistics
- I/O issues
- Bad connection management

# SQL Tuning Goals

| | |
|---|---|
| Identify | Identify bad or slow SQL |
| Find | Find the possible performance issue in a query |
| Reduce | Reduce total execution time |
| Reduce | Reduce the resource usage of a query |
| Determine | Determine most efficient execution plan |
| Balance or parallelize | Balance or parallelize the workload |

# SQL Tuning Steps



# Step 1- Identify Slow Queries

# Tracking Slow Queries

`log_min_duration_statement` tracks slow running SQL

Use **PEM's Log Manager** Wizard to configure logging of slow queries

Use **PEM's Postgres Log Analysis Expert** Wizard to analyze log messages

Run a **PEM's SQL Profiler** trace to find, troubleshoot, and optimize slow running SQL

# Instructor Demos

## Tracking Slow Queries

# Step 2 - Review the Query Execution Plan

# Execution Plan

- An execution plan shows the detailed steps necessary to execute a SQL statement
- Planner is responsible for generating the execution plan
- The Optimizer determines the most efficient execution plan
- Optimization is cost-based, cost is estimated resource usage for a plan
- Cost estimates rely on accurate table statistics, gathered with `ANALYZE`
- Costs also rely on `seq_page_cost`, `random_page_cost`, and others
- The `EXPLAIN` command is used to view a query plan
- `EXPLAIN ANALYZE` is used to run the query to get actual runtime stats

# Execution Plan Components

Syntax:

**EXPLAIN** [ ( option [, ...] ) ] statement

**EXPLAIN** [ **ANALYZE** ] [ **VERBOSE** ] statement

where option can be one of:

**ANALYZE** [ boolean ]
**VERBOSE** [ boolean ]
**COSTS** [ boolean ]
**SETTINGS** [ boolean ]
**BUFFERS** [ boolean ]
**WAL** [ boolean ]
**TIMING** [ boolean ]
**SUMMARY** [ boolean ]
**FORMAT** { TEXT | XML | JSON | YAML }

Execution Plan Components:

- **Cardinality** - Row Estimates
- **Access Method** - Sequential or Index
- **Join Method** - Hash, Nested Loop etc.
- **Join Type, Join Order**
- **Sort and Aggregates**

# Explain Example - One Table

- Example

```
postgres=# EXPLAIN SELECT * FROM emp;
                    QUERY PLAN
-------------------------------------------
Seq Scan on emp  (cost=0.00..1.14 rows=14 width=145)
```

- The numbers that are quoted by EXPLAIN are:
  - Estimated start-up cost
  - Estimated total cost
  - Estimated number of rows output by this plan node
  - Estimated average width (in bytes) of rows output by this plan node

# Explain Example - Multiple Tables

- Create the tables
  - `=# CREATE TABLE city (cityid numeric(5) primary key, cityname varchar(30));`
  - `=# CREATE TABLE office(officeid numeric(5) primary key, cityid numeric(5) references city(cityid));`
- Let's see the plan without data and updating statistics:
  - `=# EXPLAIN ANALYZE SELECT city.cityname, office.officeid, office.cityid FROM city,office WHERE office.cityid = city.cityid;`
- Output:
  - `Hash Join  (cost=25.30..71.16 rows=1510 width=102) (actual time=0.002..0.002 rows=0 loops=1)`
  - `   Hash Cond: (office.cityid = city.cityid)`
  - `   ->  Seq Scan on office  (cost=0.00..25.10 rows=1510 width=24) (actual time=0.001..0.001 rows=0 loops=1)`
  - `   ->  Hash  (cost=16.80..16.80 rows=680 width=90) (never executed)`
  - `         ->  Seq Scan on city  (cost=0.00..16.80 rows=680 width=90) (never executed)`
  - `Planning time: 0.456 ms`
  - `Execution time: 0.067 ms`

# Explain Example – Load and Analyze

- Load data:

```
=# INSERT INTO city
values(1,'Edmonton'),(2,'Calgary'),(3,'SherwoodPark'),(4,'STAlbert');

=# INSERT INTO office VALUES(generate_series(1,100),4);

=# INSERT INTO office VALUES(generate_series(101,200),3);

=# INSERT INTO office VALUES(generate_series(201,300),2);

=# INSERT INTO office VALUES(generate_series(301,400),1);
```

- Update the statistics for city and office table:

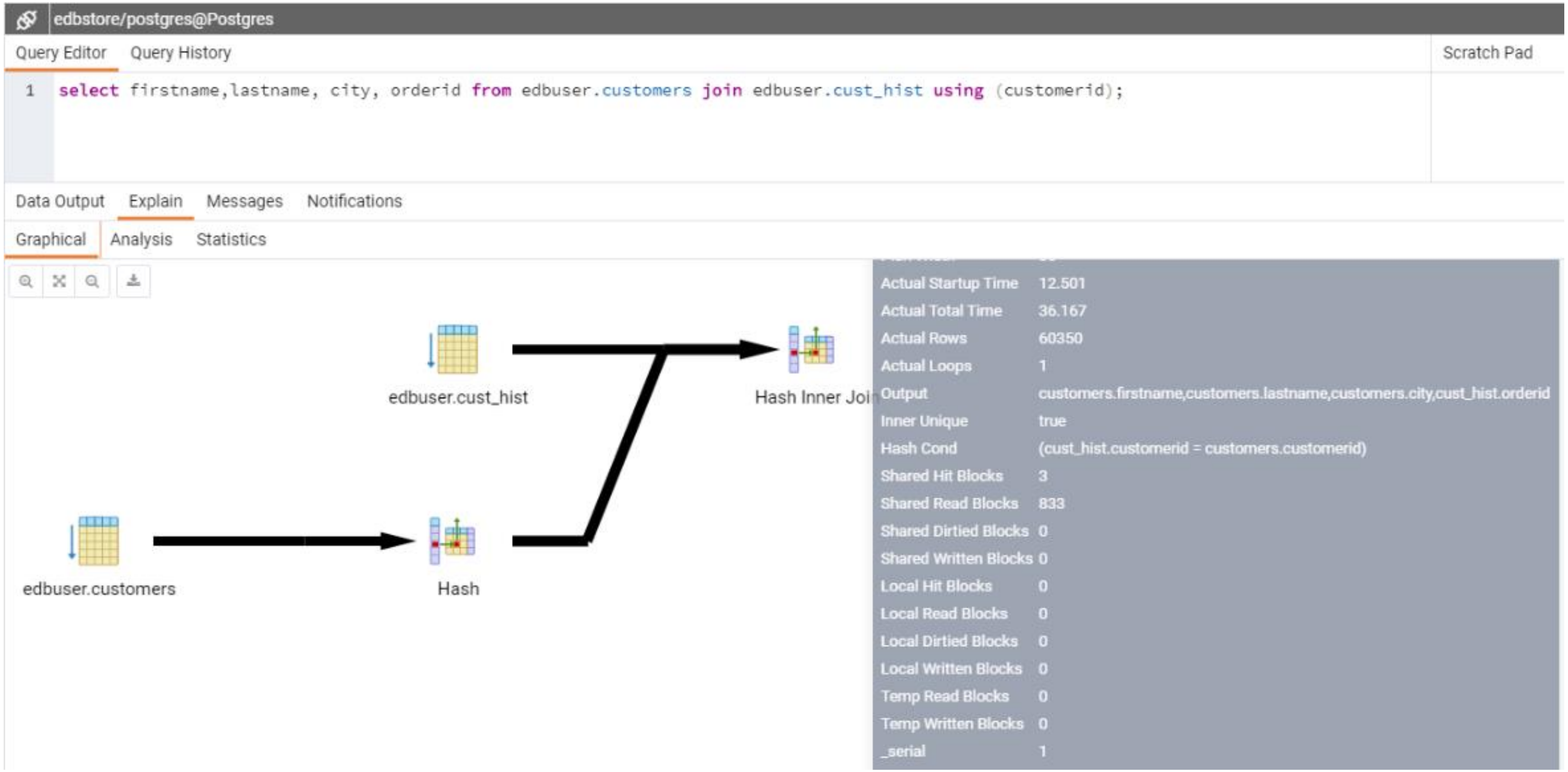
```
=# ANALYZE city;

=# ANALYZE office;
```

# Explain Example – Explain Analyze

- Plan:

```
    =# EXPLAIN ANALYZE SELECT city.cityname, office.officeid,
office.cityid FROM city, office WHERE office.cityid =
city.cityid;

 Hash Join  (cost=1.09..12.59 rows=400 width=20) (actual time=0.057..0.770 rows=400 loops=1)

    Hash Cond: (office.cityid = city.cityid)

    ->  Seq Scan on office  (cost=0.00..6.00 rows=400 width=10) (actual time=0.012..0.128 rows=400
 loops=1)

    ->  Hash  (cost=1.04..1.04 rows=4 width=15) (actual time=0.014..0.014 rows=4 loops=1)

          Buckets: 1024  Batches: 1  Memory Usage: 1kB

          ->  Seq Scan on city  (cost=0.00..1.04 rows=4 width=15) (actual time=0.002..0.005 rows=4
 loops=1)

  Planning time: 0.577 ms

  Execution time: 0.893 ms
```

# PEM - Query Tool's Visual Explain



# Reviewing Explain Plans

Examine the various costs at different levels in the explain plan

Look for sequential scans on large tables

All sequential scans are not inefficient

Check whether a join type is appropriate for the number of rows returned

Check for indexes used in the explain plan

Examine the cost of sorting and aggregations

Review whether views are used efficiently

# Instructor Demos

## Explain Command Examples

# Step 3 - Review Optimizer Statistics and Behavior

## Optimizer Statistics

- The Postgres Optimizer and Planner use table statistics for generating query plans
- Choice of query plans are only as good as table stats
- Table statistics
  - Stored in catalog tables like `pg_class` and `pg_stats`
  - Stores row sampling information

# Updating Planner Statistics

- Table Statistics
  - Absolutely critical to have accurate statistics to ensure optimal query plans
  - Are not updated in real time, so should be manually updated after bulk operations
  - Can be updated using ANALYZE command or OS command vacuumdb with -Z option
  - Stored in `pg_class` and `pg_statistics`
  - You can run the `ANALYZE` command from psql on specific tables and just specific columns
  - **Autovacuum** will run `ANALYZE` as configured
- Syntax for `ANALYZE`

```
ANALYZE [VERBOSE] [SKIP_LOCKED][table_name[(column_name[,...])]]
```

# Controlling Statistics Collection

- Postgres gathers and maintains table and column level statistics
- Statistics collection level can be controlled using:

```
=# ALTER TABLE <table> ALTER COLUMN <column> SET
   STATISTICS <number>;
```

- The `<number>` can be set between 1 and 10000
- A higher `<number>` will signal the server to gather and update more statistics but may have slow autovacuum and analyze operation on stat tables. Higher numbers only useful for tables with large irregular data distributions

# Instructor Demos

## Controlling Statistics Collection

# Step 4 - Restructuring SQL Statements and Optimizer Hints

# Restructure SQL Statements

Rewriting inefficient SQL is often easier than repairing it

Avoid implicit type conversion

Avoid expressions as the optimizer may ignore the indexes on such columns

Use equijoins wherever possible to improve SQL efficiency

Try changing the access path and join orders with hints

Avoid full table scans if using an index is more efficient

The join order can have a significant effect on performance

Use views or materialized views for complex queries

Use parallel query scans to improve the query performance

# Optimizer Hints

- Optimizer hints are directives embedded in comment-like syntax immediately following the `DELETE, INSERT, SELECT` or `UPDATE`
- Optimizer hints can alter execution plans
- Optimizer hints are used to directly influence selection of all or part of the final plan
- Optimizer hints influence optimizer decisions

# Embedding Optimizer Hints

- Optimizer hints may be given in two different formats:

```
{ DELETE | INSERT | SELECT | UPDATE } /*+ { hint [ comment ] } [...]
*/ statement_body


{ DELETE | INSERT | SELECT | UPDATE } --+ { hint [ comment ] } [...]
statement_body
```

# Usage Description

- Planner parameters override optimizer hints
- Hints must be placed immediately after the first statement keyword
- If the hint is misspelled in the SQL command, there will be no indication that any sort of error has occurred
- If an alias is used for a table or view name in the SQL command, then the alias name, not the original object name, must be used in the hint
- Hints apply only to the statement where it is specified
- Use `EXPLAIN` to ensure that hints are correctly used

# Access Method Hints

- The following hints influence how the optimizer accesses relations to create the result set:

| Hint | Description |
|---|---|
| `FULL (table)` | Perform a full sequential scan on `table` |
| `INDEX (table [index] […])` | Use `index` on `table` to access the relation |
| `NO_INDEX (table [index] […])` | Do not use `index` on `table` to access the relation |

- The `FULL` hint is used to force a full sequential scan instead of using the index:

```
=# EXPLAIN SELECT /*+ FULL(accounts) */ * FROM
   accounts WHERE aid = 100;
```

- The `NO_INDEX` hint also forces a sequential scan

# Join Hints

- There are three possible plans that may be used to perform a join between two tables:
  - Nested Loop Join, Merge Sort Join and Hash Join

| Hint | Description |
|---|---|
| `USE_HASH(table […])` | Use a hash join with a hash table created from the join attributes of `table` |
| `NO_USE_HASH(table […])` | Do not use a hash join created from the join attributes of `table` |
| `USE_MERGE(table […])` | Use a merge sort join for `table` |
| `NO_USE_MERGE(table […])` | Do not use a merge sort join for `table` |
| `USE_NL(table […])` | Use a nested loop join sort for `table` |
| `NO_USE_NL(table […])` | Do not use a nested loop join sort for `table` |

Example:

```
=# EXPLAIN SELECT /*+ USE_HASH(a) */ b.bid, a.aid, abalance FROM
   branches b, accounts a WHERE b.bid = a.bid;
```

# APPEND Optimizer Hint

- By default, Advanced Server will add new data into the first available free-space
- Use `APPEND` optimizer hint with `INSERT` or `SELECT` for adding new rows at the end of the table
- This is useful when bulk loading data
- Example:

```
=# INSERT /*+APPEND*/ INTO sales VALUES (10, 10, '01-SEP-2014', 10, 'OR');
=# INSERT INTO sales_history SELECT /*+APPEND*/ FROM sales;
```

# Parallelism Hints

- Parallel scanning provides performance improvement over sequential scan
- The `PARALLEL` optimizer hint is used to force parallel scanning
- The `NO_PARALLEL` optimizer hint prevents usage of a parallel scan
- Syntax:

```
PARALLEL (table [ parallel_degree | DEFAULT ])
NO_PARALLEL (table)
```

- Example:

```
=# EXPLAIN SELECT /*+ PARALLEL(pgbench_accounts) */ * FROM
pgbench_accounts;
```

# Instructor Demos

## Optimizer Hints

# Step 5 - Review Indexes

## General Indexing Guidelines

- Create indexes only when needed
- Remove unused indexes
- Adding an index for one SQL can slow down other queries
- Verify index usage using the `EXPLAIN` command

# Indexes

| Indexes | Description |
|---|---|
| B-tree | equality and range queries on data that can be sorted (Default index type) |
| Hash | only simple equality comparisons |
| GiST | can be used depending on the indexing strategy (the operator class) |
| SP-GiST | like GiST, offer an infrastructure that supports various kinds of searches |
| GIN | can handle values that contain more than one key, for example arrays |
| BRIN (Block Range Index) | accelerates scanning of large tables by maintaining summary data about block ranges. Very small index size compared to B-Tree, the tradeoff being you can't select a specific row so they are not useful for all data sets |

# Multicolumn Indexes

- An index can be defined on more than one column of a table
- Currently, only the B-tree, GiST, and GIN index types support multicolumn indexes
- Example:

```
=> CREATE INDEX test_idx1 ON test (id_1, id_2);
```

  - This index will be used when you write:

```
=> SELECT * FROM test WHERE id_1=1880 AND
id_2= 4500;
```

- Multicolumn indexes should be used sparingly

# Indexes and ORDER BY

- You can adjust the ordering of a B-tree index by including the options `ASC, DESC, NULLS FIRST, and/or NULLS LAST` when creating the index to save sorting time spent by the query
- By default, B-tree indexes store their entries in ascending order with nulls last
- Examples:

```
=# CREATE INDEX test2_info_nulls_low ON test2 (info NULLS FIRST);

=# CREATE INDEX test3_desc_index ON test3 (id DESC NULLS LAST);
```

# Instructor Demos

## Index and Order By

# Unique Indexes

- Indexes can also be used to enforce uniqueness of a column's value or the uniqueness of the combined values of more than one column

```
=> CREATE UNIQUE INDEX name ON table (column [, ...]);
```

  - Currently, only B-tree indexes can be declared unique
- Postgres automatically creates a unique index when a unique constraint or primary key is defined for a table

# Functional Indexes

- An index can be created on a computed value from the table columns.
  - For example:
    ```
    => CREATE INDEX test1_lower_col1_idx ON test1 (lower(col1));
    ```
- Index expressions are relatively expensive to maintain
- Indexes on expressions are useful when retrieval speed is more important than insertion and update speed
- Indexes can also be created on user defined functions
- `ALTER INDEX ALTER COLUMN SET STATISTICS` can be used to set statistic target for index columns that are defined as expression

# Partial Indexes

- A partial index is an index built over a subset of a table.
- The index contains entries only for those table rows that satisfy the predicate.
  - Example:

```
edbstore=> CREATE TABLE citizen (id int, name varchar(50), city varchar(30),
state varchar(20));
CREATE TABLE

edbstore=> CREATE INDEX citizen_state_idx_AR ON citizen(id) WHERE state='AR';
CREATE INDEX
```

# Instructor Demos

## Partial Indexes

# BRIN Indexes

BRIN (Block Range Index) stores metadata on range (min and max) of pages

Block Range is by default set to 1MB and can be modified using pages_per_range parameter

BRIN is small in size, easier to maintain and eliminates disk I/O by storing summary of data distribution on disk

Designed to handle large table columns, which correlate to the physical location within the table

# Example - BRIN Indexes

```
postgres=# CREATE TABLE brin_example AS
 SELECT generate_series(1,100000000) AS id;
postgres=#
postgres=# CREATE INDEX brin_index on brin_example USING brin(id);
postgres=#
postgres=# CREATE INDEX btree_index ON brin_example(id);
postgres=#
postgres=# SELECT relname, pg_size_pretty(pg_relation_size(oid))
postgres-# FROM pg_class
postgres-# WHERE relname LIKE 'brin_%' OR relname='btree_index'
postgres-# ORDER BY relname;
   relname    | pg_size_pretty
--------------+----------------
 brin_example | 3457 MB
 brin_index   | 104 kB
 btree_index  | 2142 MB
(3 rows)
```

# Instructor Demos

## BRIN Indexes

# Index Only Scans

- Indexes speed up data retrieval and sorting by storing indexed column value and pointers in index pages
- If all the columns fetched by a query are indexed, PostgreSQL can perform an index only scan
- Index-only scan is a faster version of the ordinary index scan
- Any non-key column can also be included with indexed column in the index page using `INCLUDE` option of `CREATE INDEX` statement

# Instructor Demos

## Index-only Scan

# Examining Index Usage

- It is difficult to formulate a general procedure for determining which indexes to create.
  - Always run `ANALYZE` first.
  - Use real data for experimentation.
  - When indexes are not used, it can be useful for testing to force their use.
  - The `EXPLAIN ANALYZE` command can be useful here.

# Step 6 - Review Final Execution Plan

# Last Step - Review the Final Plan

Check again for missing indexes

Check table statistics are showing correct estimates

Check large table sequential scans

Compare the cost of first and final plans

# Module Summary

- SQL queries planning and execution
- Find and tune slow running SQL
- Types of Indexes and their usage
- Writing an efficient SQL query

# Lab Exercise - 1

1. You are working as a DBA. Users are complaining about long running queries and high execution times. Configure your database instance to log slow queries. Any query taking more than 5 seconds must be logged.
2. After logging slow queries you find the following query taking longer than expected:

```
=> SELECT * FROM customers JOIN orders USING(customerid);
```

3. View the explain plan of the above query.
4. View the execution time for the above query in `psql` terminal.

# Lab Exercise - 2

1. Create a table using following queries:
   - `CREATE TABLE lab_test1 (c1 int4, c2 int4);`
   - `INSERT INTO lab_test1(c1, c2) values(generate_series(1, 100000), 1);`
   - `INSERT INTO lab_test1(c1, c2) values(generate_series(100001, 200000), 2);`
   - `INSERT INTO lab_test1(c1, c2) values(generate_series(200001, 300000), 3);`

2. Create three partial indexes on `lab_test1` table as following:

| Index Name | Predicate |
|---|---|
| idx1_c1 | c1 between 1 and 100000 |
| idx2_c1 | c1 between 100001 and 200000 |
| idx3_c3 | c1 between 200001 and 300000 |

# Lab Exercise - 3

1. Detect the index usage for all the user indexes in `edbstore` database.
2. Reindex all the indexes.
3. Manually update the statistics for all the objects in `edbstore` database.

# Module 3
# Performance Tuning

# Module Objectives

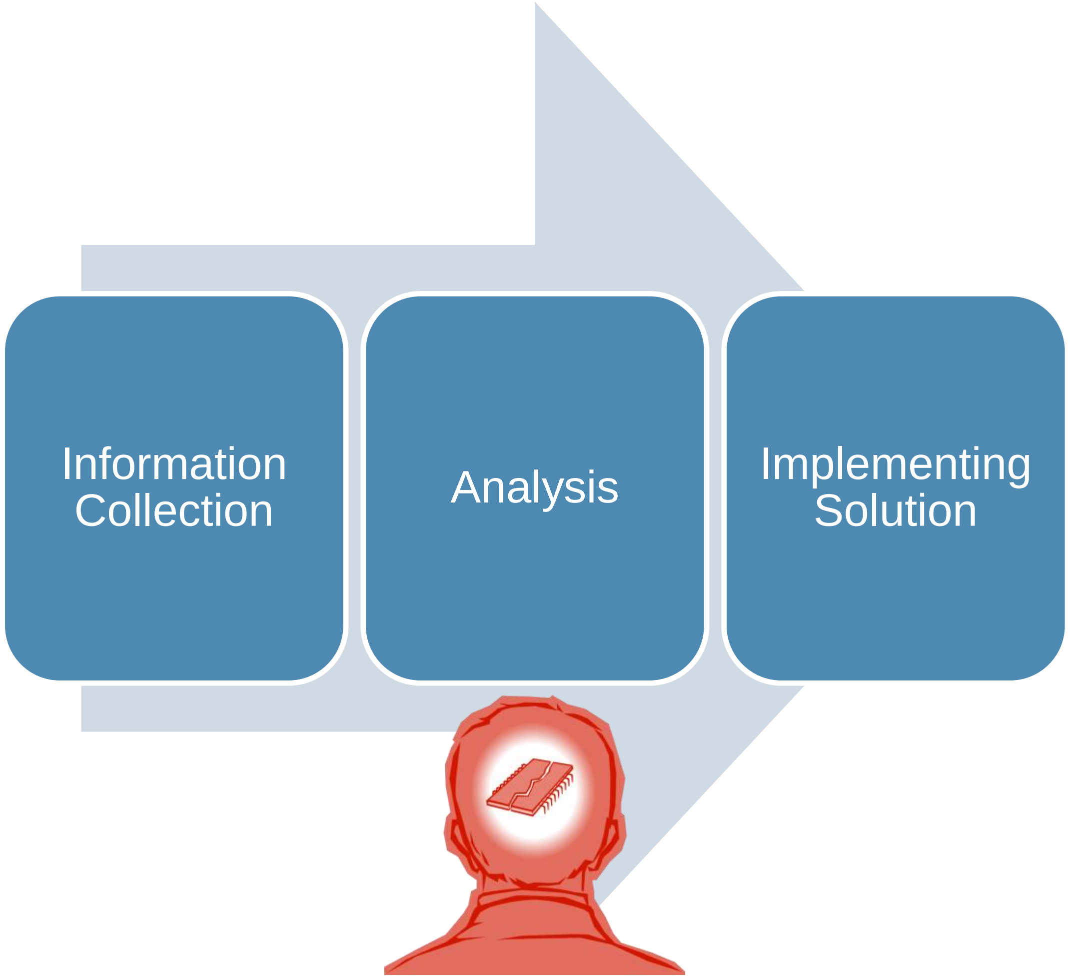
- Parameter Tuning in Postgres
- Hot cache vs cold cache
- Data Loading best practices
- Tuning Postgres Using PEM

# Performance Tuning - Overview

- Performance Tuning is a group of activities used to optimize a database
- Database Tuning is used to correct:
  - Poorly written SQL
  - Poor session management
  - Misconfigured database parameters
  - Operating system I/O issues
  - No Database maintenance

# The Performance Tuning Process

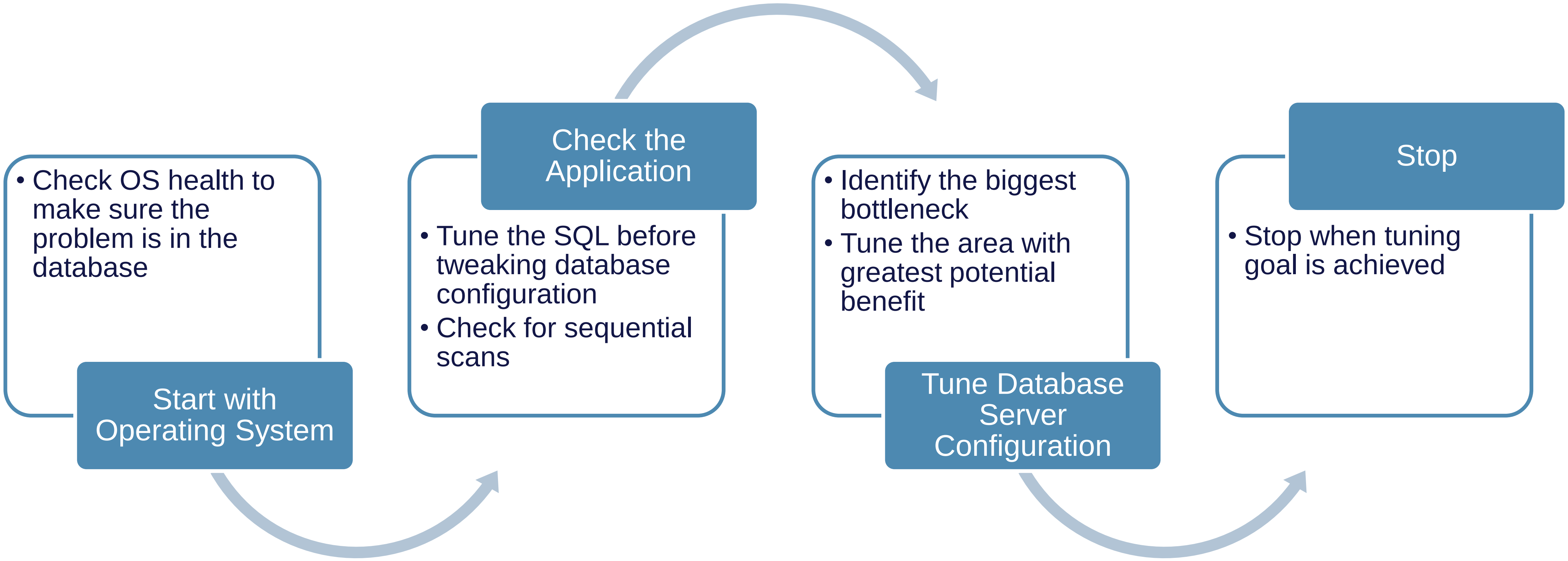
- Identify the information relevant to diagnosing performance problems
- Collect that information on a regular basis
- Expert analysis is needed to understand and correlate all the relevant statistics together
- Multiple solutions for different problems
- Use your own judgment to prioritize and quantify the solutions by impact



# Performance Monitoring Using PEM

- Postgres Enterprise Manager (PEM) simplifies collection of performance data
- Automatic analysis of performance and diagnostics data
- Performance Dashboards - view I/O, memory usage, session activity, and wait statistics
- SQL Profiler - optimize slow SQL
- Index Advisor
- Setup alerts and thresholds

# Tuning Technique



# Operating System Considerations

# Operating System Issues

## Memory

Increased memory demand may lead to 100% memory usage and swapping

Check memory usage

Solution is to reduce memory usage or increase system RAM

## CPU

CPU may be the bottleneck when load or process wait is high

Check CPU usage (%) for the database connections

Solution is to reduce CPU usage (%)

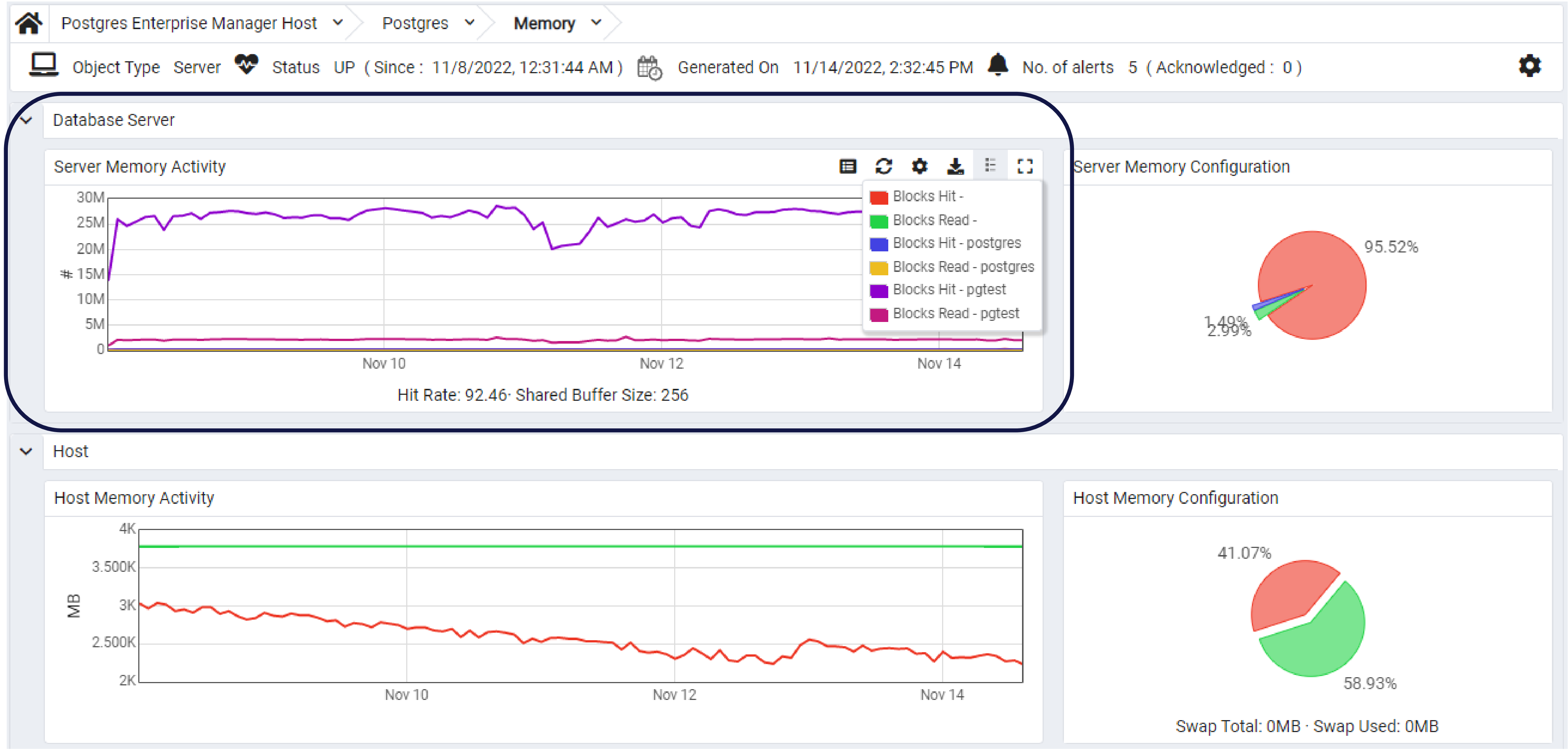
## Disk (I/O)

High wait times or request rates are symptoms of an I/O problem

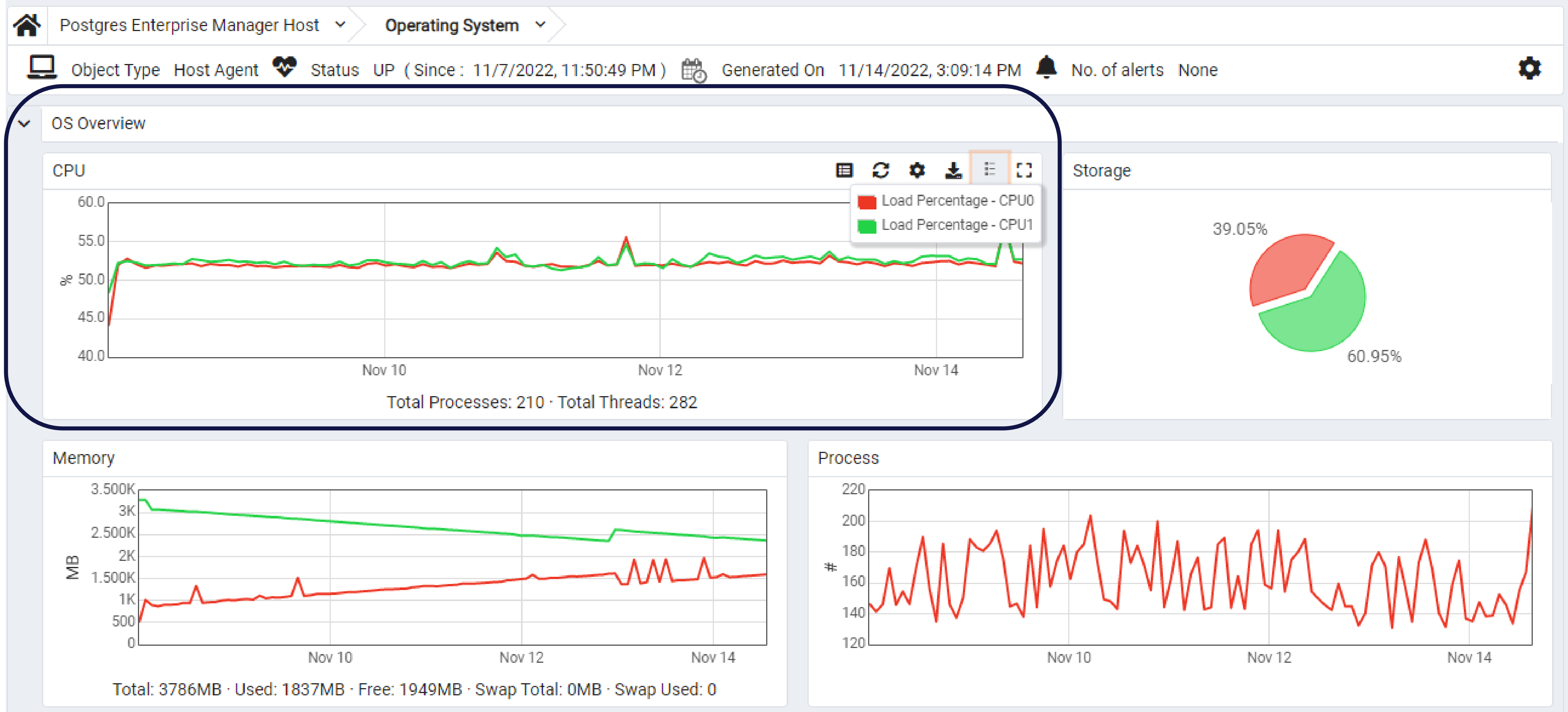
Check for I/O spikes

Solution is to reduce demand or increase capacity

# PEM - Memory Graphs and Alerts



# PEM - CPU Info

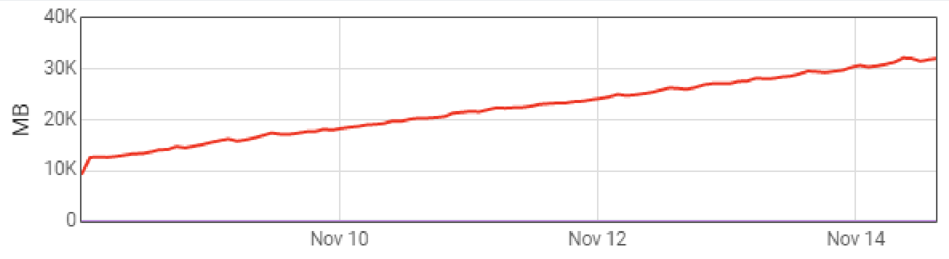


Postgres Enterprise Manager Host
Operating System
Object Type Host Agent
Status UP ( Since : 11/7/2022, 11:50:49 PM )
Generated On 11/14/2022, 3:09:14 PM
No. of alerts None
OS Overview
CPU
Load Percentage - CPU0
Load Percentage - CPU1
60.0
55.0
50.0
45.0
40.0
%
Nov 10
Nov 12
Nov 14
Total Processes: 210 · Total Threads: 282
Storage
39.05%
60.95%
Memory
3.500K
3K
2.500K
2K
1.500K
1K
500
0
MB
Nov 10
Nov 12
Nov 14
Total: 3786MB · Used: 1837MB · Free: 1949MB · Swap Total: 0MB · Swap Used: 0
Process
220
200
180
160
140
120
#
Nov 10
Nov 12
Nov 14

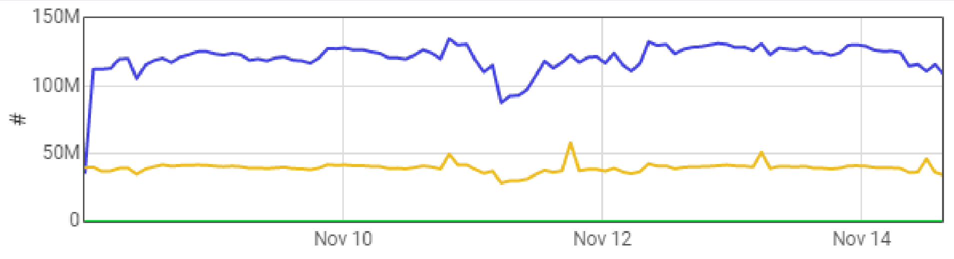
# PEM - Disk Info

Disk

Disk



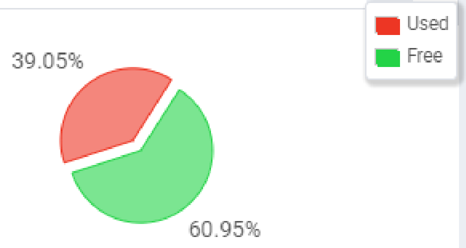

Storage




Host Details

| File System | Size (GB) | Used (GB) | Available (GB) | % Used | Mounted On |
|---|---|---|---|---|---|
| /dev/sda1 | 0.20 | 0.01 | 0.18 | 5.50 | /boot/efi |
| /dev/sda2 | 79.79 | 31.22 | 48.57 | 39.13 | / |

Storage



# Instructor Demos

## Install PEM OS Dashboards

# Hardware Configuration

### Focus on disk speed and RAM over CPU speed

- Like other RDBMS, Postgres is IO intensive

### Separate the transaction log and indexes

- Put the database transaction log (pg_wal) on a dedicated disk resource
- Tablespaces can be used to create indexes on separate drives

### Setup disks to match speed requirements, not just size requirements

- IOPS is just as important as GB when purchasing hardware
- RAID 0 + 1 is optimal for speed and redundancy
- Consider disk speed during failures (e.g. RAID-5 parity rebuild after failed disk)
- Write caches must be persistent battery backed “Write-Back”, or you will get corruption

# OS Configuration

- The filesystem makes a difference
  - A journaling file system is not required for the transaction log
  - Multiple options are available for Linux:
    - EXT2 with sync enabled
    - EXT3 with write-back
    - EXT4 and XFS
    - Remote file systems are not recommended
- Choose the best based on better writes, recoverability, support from multiple vendors and reliability
- Eliminate unnecessary filesystem overhead, such as “noatime”
- Consider a virtual “ram” disk for `stats_temp_directory`

# Server Parameter Tuning

# Server Parameter Tuning

- Default **postgresql.conf** server parameters are configured for wide compatibility
- Server parameters must be tuned to optimal values for better performance
- Parameters can be evaluated using different methods and can be set permanently in the **postgresql.conf** file
- Some parameters require a restart
- Basic information needed for tuning but not limited to:
  - Database size
  - Largest table size
  - Type and frequency of queries
  - Available RAM
  - Number of concurrent connections

# PEM - Postgres Expert

- Postgres Expert analyzes the server configuration and reports potential performance and security issues
- Report also provides suggestions and best practices for addressing potential issues
- Postgres Expert is an advisory utility which provides advice about Performance, Security and Configuration

# PEM - Postgres Expert Report

Postgres Expert Report

Generated On: 2022-11-14 22:15:14 Go to: Postgres

Summary

Servers Tested: 1 Rules Checked: 31 High Alerts: 1 Medium Alerts: 3 Low Alerts: 2

Server: Postgres (127.0.0.1:5433)

Recommendations

Advisor: Configuration Expert

| Rule | Database | Severity |
|---|---|---|
| Check checkpoint_completion_target | - | Medium |
| Check effective_cache_size | - | Medium |
| Check effective_io_concurrency | - | Low |
| Check reducing random_page_cost | - | Low |

Advisor: Schema Expert

| Rule | Database | Severity |
|---|---|---|
| Check data and transaction log on same drive | - | High |

Advisor: Security Expert

| Rule | Database | Severity |
|---|---|---|
| Check SSL for improved connection security | - | Medium |

# Instructor Demos

## Postgres Expert

# Connection Settings

- `max_connections`
  - Sets the maximum number of concurrent connections
  - Each user connection has an associated user backend process on the server
  - User backend processes are terminated when a user logs off
  - Connection pooling can decrease the overhead on postmaster by reusing existing user backend processes

# Memory Parameters - shared_buffers

- Sets the number of shared memory buffers used by the database server
- Each buffer is 8K bytes
- Minimum value must be 16 and at least 2 x `max_connections`
- 6% - 25% of available memory is a good general guideline
- You may find better results keeping the setting relatively low and using the operating system cache more instead

# Memory Parameters - work_mem

- `work_mem`
  - Amount of memory in KB to be used by internal sorts and hash tables before switching to temporary disk files
  - Minimum allowed value is 64 KB
  - It is set in KB
  - Increasing the `work_mem` often helps in faster sorting
  - `work_mem` settings can also be changed on a per session basis

# Memory Parameters - maintenance_work_mem

- `maintenance_work_mem`
  - Maximum memory in KB to be used in maintenance operations such as `VACUUM, CREATE INDEX`, and `ALTER TABLE ADD FOREIGN KEY`
  - Minimum allowed value is 1024 KB
  - It is set in KB
  - Performance for vacuuming and restoring database dumps can be improved by increasing this value

# Memory Parameters - autovacuum_work_mem

- `autovacuum_work_mem`
  - Maximum amount of memory to be used by each autovacuum worker process
  - Default value is `-1`, indicates that `maintenance_work_mem` to be used instead

# Memory Parameters - huge_pages

- `huge_pages`
  - Enables/disables the use of huge/large pages
  - Valid values are `try` (the default), `on`, and `off`
  - May help in increasing performance by using smaller page tables thus less CPU time on memory management
  - This parameter is only supported on Linux and Windows
- `huge_page_size`
  - Controls size of huge pages on Linux systems

# Memory Settings for Planner

- `effective_cache_size`
  - Size of memory available for disk cache that is available to a single query
  - A higher value favors index scans
  - This parameter does not reserve kernel disk cache; it is used only for estimation purposes
  - ½ of RAM is a conservative setting
  - ¾ of RAM is more aggressive
  - Find the optimal value looking at OS stats after increasing or decreasing this parameter

# Temporary Files

- `temp_file_limit`
  - Maximum amount of disk space that a session can use for temporary files
  - A transaction attempting to exceed this limit will be cancelled
  - Default is `-1` (no limit)
  - This setting constrains the total space used at any instant by all temporary files used by a given Postgres session

# WAL Parameters - wal_level

- `wal_level`
  - `wal_level` determines how much information is written to the WAL
  - The default value is `replica`, adds logging required for WAL archiving as well as information required to run read-only queries on a Replica server
  - The value `logical` is used to add information required for logical decoding
  - The value `minimal`, removes all logging except the information required to recover from a crash or immediate shutdown
  - This parameter can only be set at server start

# WAL Parameters - wal_buffers

- `wal_buffers`
  - Number of disk-page buffers allocated in shared memory for WAL data
  - Each buffer is 8K bytes
  - Needs to be only large enough to hold the amount of WAL data created by a typical transaction since the WAL data is flushed out to disk upon every transaction commit
  - Minimum allowed value is 4
  - Default setting is `-1` (auto-tuned)

# WAL Parameters – Checkpoints and max_wal_size

- Checkpoints
  - Writes the current in-memory modified pages (known as dirty pages) to the disk
  - An automatic checkpoint occurs each time the `max_wal_size` is reached
- `max_wal_size` (default: 1GB)
  - Maximum distance between automatic WAL checkpoints
  - Each log file segment is 16 megabytes and the size can changed at the time of initialization using --wal-segsize
  - A checkpoint is forced when the `max_wal_size` is reached
  - `max_wal_size` is soft limit and WAL size may exceed during heavy load, failed archive command, or high `wal_keep_size`
  - Increase in `max_wal_size` also increases mean time to recovery

# WAL Parameters - checkpoint_timeout

- `checkpoint_timeout`
  - Maximum time between automatic WAL checkpoints in seconds before a checkpoint is forced
  - A larger setting results in fewer checkpoints
  - Range is `30 - 3600` seconds
  - The default is `300` seconds

# WAL Parameters - fsync

- `fsync`
  - Ensures that all the WAL buffers are written to the WAL logs at each `COMMIT`
  - When on, `fsync()` or other `wal_sync_method` is forked
  - Turning this off will be a performance boost but there is a risk of data corruption
  - Can be turned off during initial loading of a new database cluster from a backup file
  - `synchronous_commit = off` can provide similar benefits for noncritical transactions without any risk of data corruption

# Instructor Demos

## Parameter Tuning - work_mem

# pg_test_fsync Tool

- `wal_sync_method` is used for forcing WAL updates out to disk
- **pg_test_fsync** can determine the fastest `wal_sync_method` on your specific system
- **pg_test_fsync** reports average file sync operation time
- Diagnostic information for an identified I/O problem

# Instructor Demos

## Using pg_test_fsync

# Parallel Queries – Parallel Plans

- Parallel scans – The following types of parallel table scans are supported:
  - Parallel sequential scan – Allows multiple workers to perform a sequential scan
  - Parallel bitmap heap scan – Allows a single index scan to dispatch parallel workers to process different areas of the heap
  - Parallel index scan or parallel index-only scan – Allows B-tree index pages to be searched by separate parallel workers
- Parallel joins – Allows nested loop, hash join or merge joins to be performed in parallel
- Parallel aggregation – Allows queries with aggregations to be parallelized
- Parallel DDLs - `CREATE TABLE AS SELECT`, `CREATE INDEX`, and `CREATE MATERIALIZED VIEW`

# Parallel Query Scan Parameters

- Postgres supports parallel execution of the read-only queries
- Parallel scans can be enabled and configured using various configuration parameters

| Parameter Name | Default Value |
|---|---|
| enable_parallel_append | on |
| enable_parallel_hash | on |
| force_parallel_mode | off |
| max_parallel_maintenance_workers | 2 |
| max_parallel_workers | 8 |
| max_parallel_workers_per_gather | 2 |
| min_parallel_index_scan_size | 64 |
| min_parallel_table_scan_size | 1024 |
| parallel_setup_cost | 1000 |
| parallel_tuple_cost | 0.1 |

# Example – Parallel Query Scan

- Create table and insert data

```
postgres=# CREATE TABLE test (id int);
CREATE TABLE
postgres=# INSERT INTO test values(generate_series(1,100000000));
INSERT 0 100000000
postgres=# ANALYZE test;
ANALYZE
```

- Disable parallel scan and check execution time:

```
postgres=# set max_parallel_workers_per_gather=0;
SET
postgres=# explain analyze select * from test where id=1;
                                              QUERY PLAN

--------------------------------------------------------------------------------
------------------------
 Seq Scan on test  (cost=0.00..1692478.40 rows=1 width=4) (actual time=1.757..13
121.173 rows=1 loops=1)
   Filter: (id = 1)
   Rows Removed by Filter: 99999999
 Planning time: 0.233 ms
 Execution time: 13126.936 ms
(5 rows)
```

# Example – Parallel Query Scan (continued)

- Enable parallel scan and check execution time:

```
postgres=# set max_parallel_workers_per_gather=2;
SET
postgres=# explain analyze select * from test where id=1;
                                                       QUERY PLAN

--------------------------------------------------------------------------------
----------------------------------------
 Gather  (cost=1000.00..964311.60 rows=1 width=4) (actual time=2.173..5527.018 r
ows=1 loops=1)
   Workers Planned: 2
   Workers Launched: 2
   ->  Parallel Seq Scan on test  (cost=0.00..963311.50 rows=1 width=4) (actual
time=3661.332..5502.736 rows=0 loops=3)
         Filter: (id = 1)
         Rows Removed by Filter: 33333333
 Planning time: 0.102 ms
 Execution time: 5527.053 ms
(8 rows)
```

# Instructor Demos

## Install Parallel Queries

# Loading a Table into Memory

- `pg_prewarm`
  - Can be used to load relation data into either the operating system buffer cache or into the PostgreSQL buffer cache
  - Supports `prefetch` method for operating system buffer cache and `buffer` method for PostgreSQL buffer cache
- pg_prewarm can be used to automatically load shared blocks at the time of server restart
- Add `pg_prewarm` to `shared_preload_libraries` to start autoprewarm master process
- Configuration Parameters:
  - `pg_prewarm.autoprewarm` (default is on)
  - `pg_prewarm.autoprewarm_interval` (default is 300 seconds)

# pg_prewarm Example

- Install PostgreSQL Contrib:
  - `yum install postgresql14-contrib`
- Add pg_prewarm extension to a database and load table data into shared buffers:

```
pgtest=# CREATE EXTENSION pg_prewarm;
CREATE EXTENSION
pgtest=# SELECT pg_prewarm('pgbench_tellers','buffer');
 pg_prewarm
------------
        552
(1 row)

pgtest=#
```

# Instructor Demos

## pg_prewarm

# Best Practices for Inserting Large Amounts of Data

- While inserting data using multiple inserts use `BEGIN` at the start and `COMMIT` at the end
- Use `COPY` to load all the rows in one command, instead of using a series of `INSERT` commands
- If you cannot use `COPY`, it might help to use `PREPARE` to create a prepared `INSERT` statement, and then use `EXECUTE` as many times as required
- If you are loading a freshly created table, the fastest method is to create the table, bulk load the table's data using `COPY`, then create any indexes needed for the table. EDB*Loader of Advanced Server is 2x faster than COPY
- It might be useful to drop foreign key constraints, load the data, and then re-create the constraints

# Best Practices for Inserting Large Amounts of Data (Continued)

- Temporarily increasing the `maintenance_work_mem` and `max_wal_size` configuration variables when loading large amounts of data can lead to improved performance
- Disable WAL Archival and Streaming Replication
- Triggers and Autovacuum can also be disabled
- Certain commands run faster if `wal_level` is minimal:
  - `CREATE TABLE AS SELECT`
  - `CREATE INDEX` (and variants such as `ALTER TABLE ADD PRIMARY KEY`)
  - `ALTER TABLE SET TABLESPACE`
  - `CLUSTER`
  - `COPY FROM`, when the target table has been created or truncated earlier in the same transaction

# Non-Durable Settings

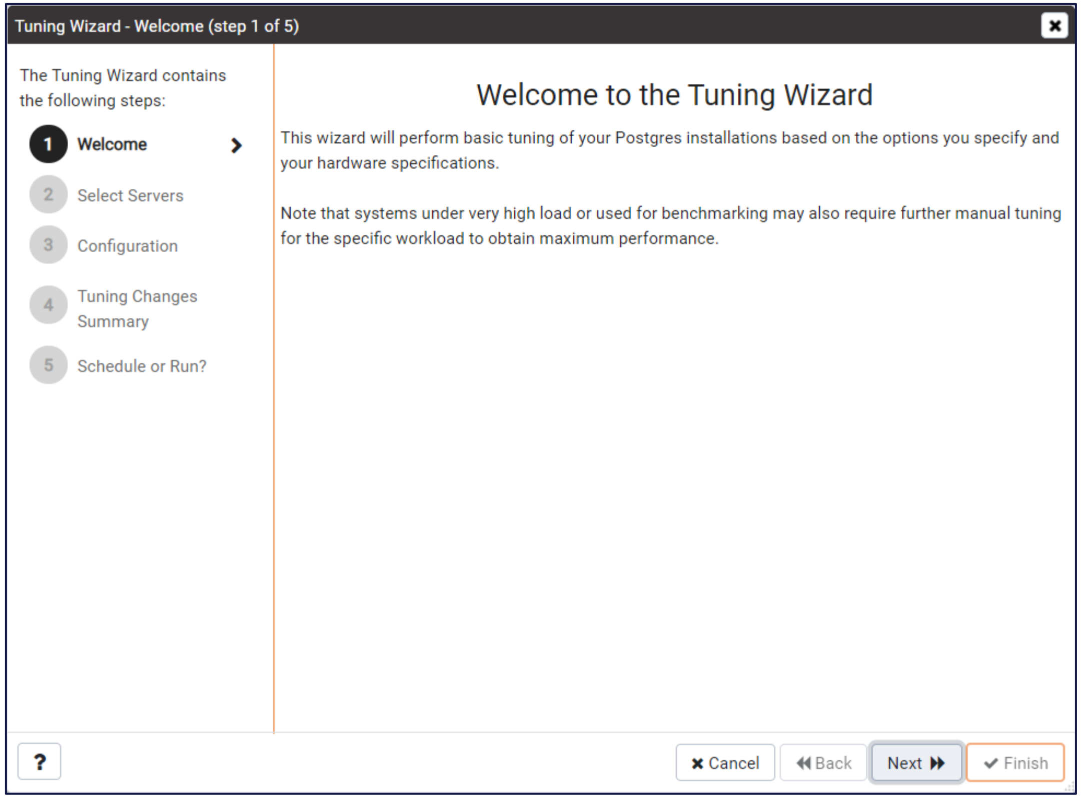
- Durability guarantees the recording of committed transactions but adds significant overhead.
- Postgres can be configured to run without durability.
- Turn off `fsync` when there is no need to flush wal data to disk.
- Turn off `full_page_writes`.
- Increase `max_wal_size` and `checkpoint_timeout`; this reduces the frequency of checkpoints.
- Turn off `synchronous_commit` when there is no need to write the WAL to disk on every commit.

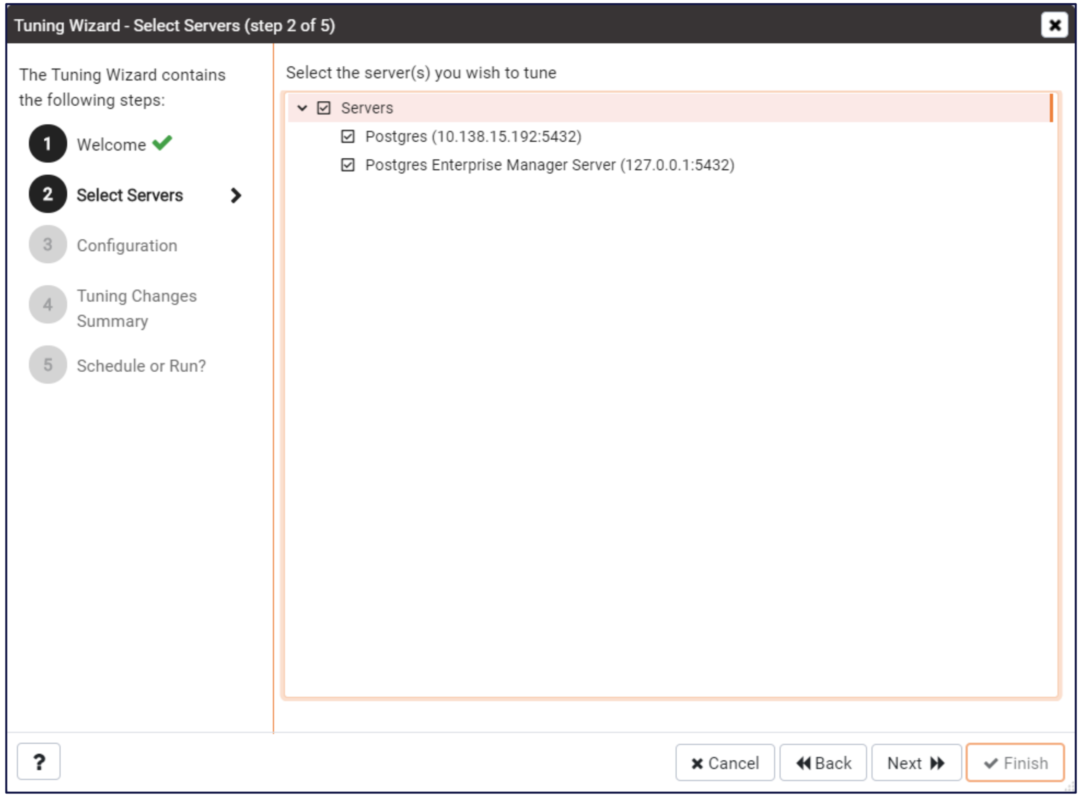
# Tuning with Postgres Enterprise Manager

# PEM Tuning Wizard

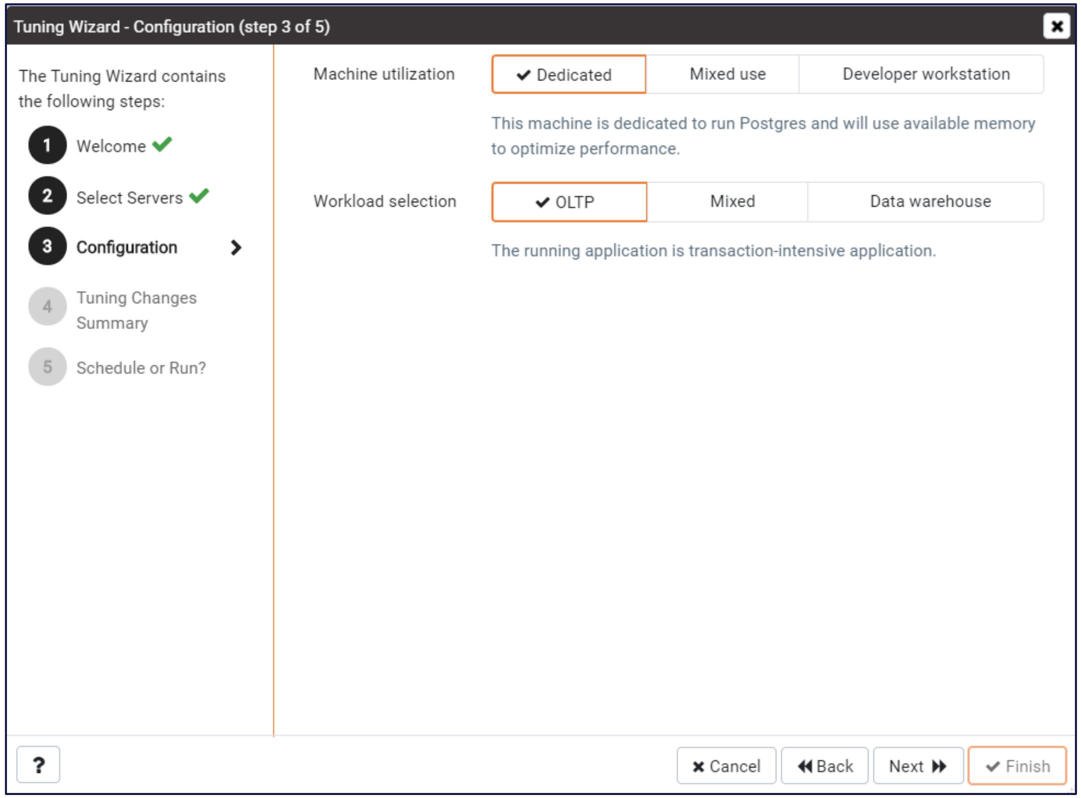
- The PEM Tuning Wizard reviews your PostgreSQL installation and recommends a set of configuration options to tune your server
- You must specify `Service ID` field on the `Advanced` tab of the server's `Properties` dialog
- It can make recommendations for servers that reside on the same server as their bound PEM agent
- If a value of `Yes` is specified in the `Remote monitoring` field while defining the server then it will not be displayed in Tuning Wizard tree control

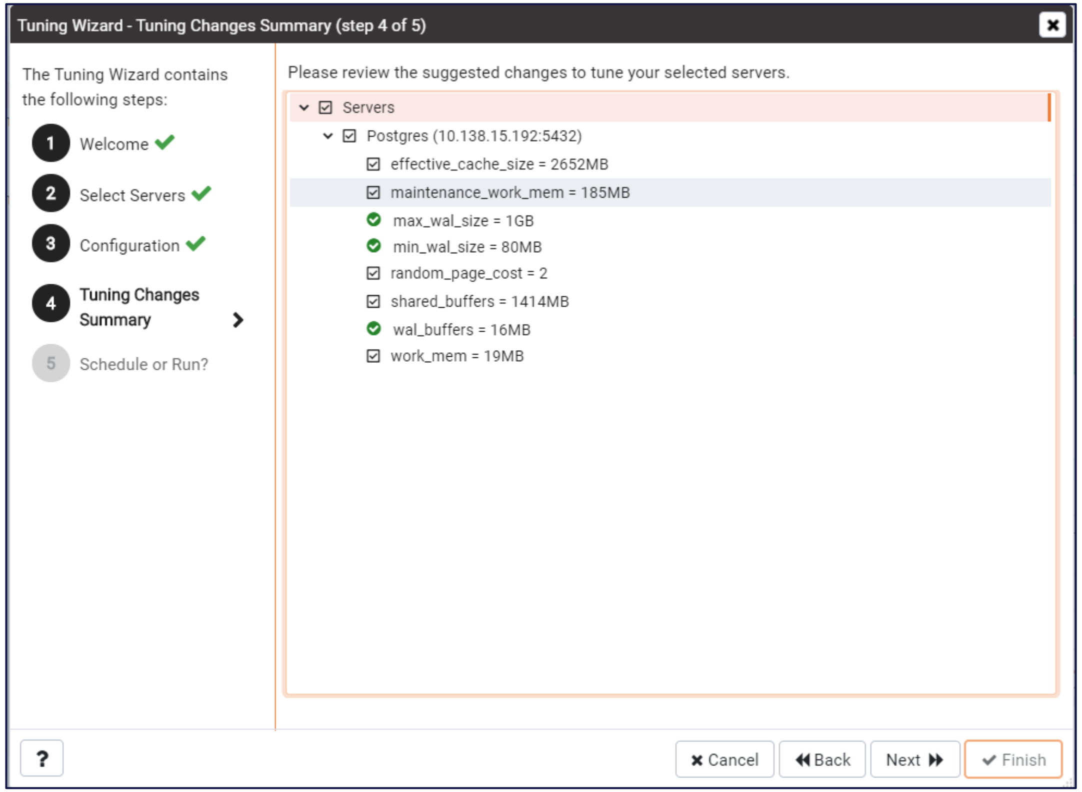
# PEM Tuning Wizard – Welcome and Select Servers



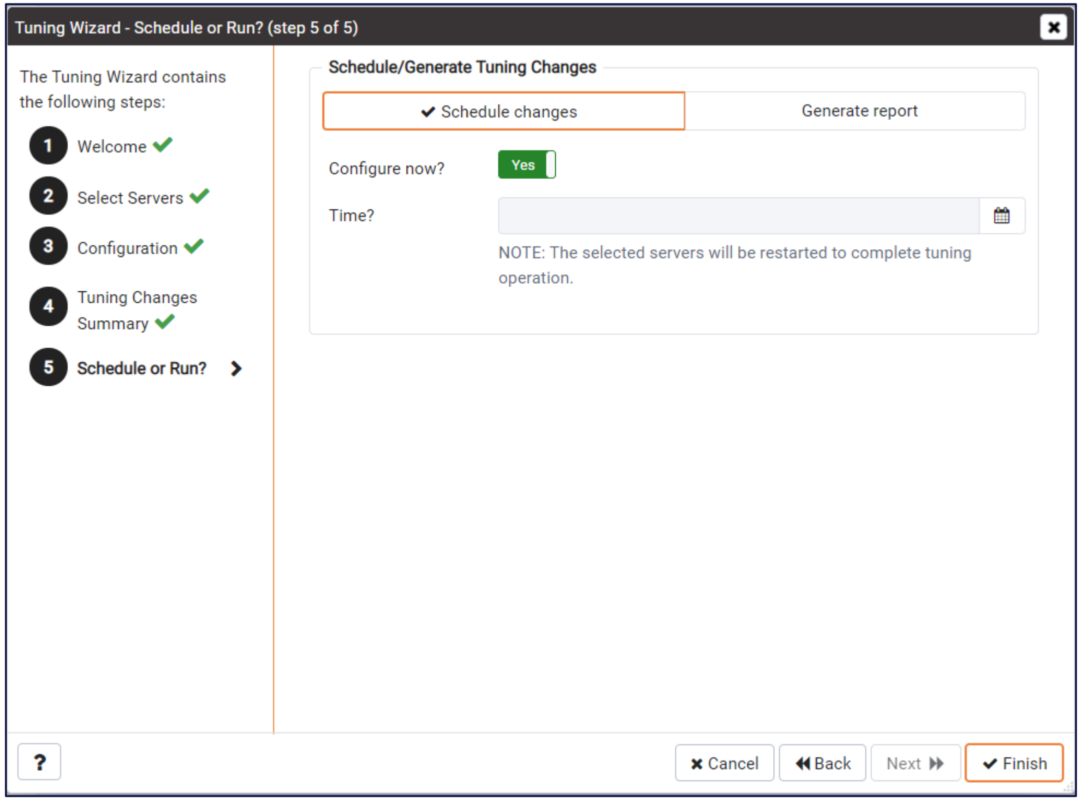



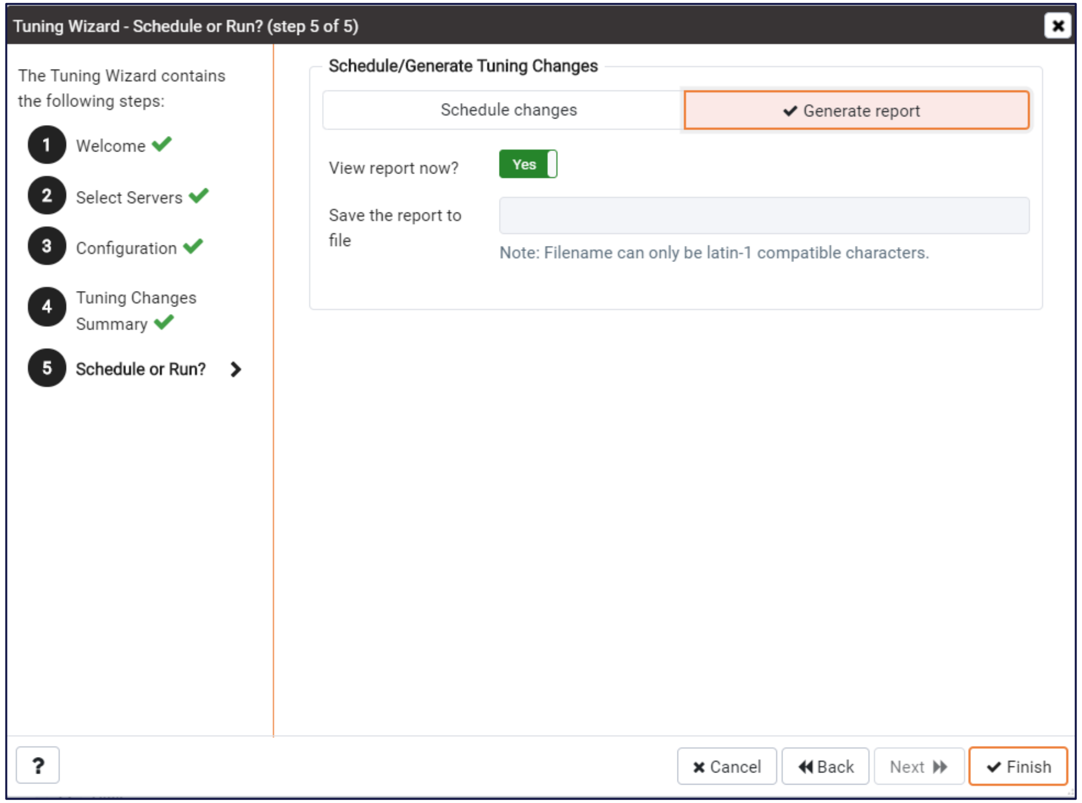
# PEM Tuning Wizard – Configuration and Tuning Changes Summary



# PEM Tuning Wizard – Schedule/Run and Generate Report



# PEM Tuning Wizard Report

Tuning Wizard Report

Generated On: 2022-11-14 22:26:15

Go to: Postgres

Summary

Number of servers selected: 1 Machine utilization: Dedicated Workload profile: OLTP

Server: Postgres (127.0.0.1:5433)

| GUC Parameter | Original Value | Recommended Value |
|---|---|---|
| effective_cache_size | 4096MB | 1420MB |
| maintenance_work_mem | 64MB | 99MB |
| random_page_cost | 4 | 2 |
| shared_buffers | 256MB | 757MB |
| wal_buffers | 8MB | 16MB |
| work_mem | 4MB | 10MB |

# Instructor Demos

## Tuning Wizard

# SQL Profiler

- SQL Profiler is a graphical wizard available in PEM
- PEM SQL Profiler capabilities are very similar to Microsoft SQL Server’s Profiler
- It can be used to:
    - Capture workloads in a SQL trace
    - Monitor and analyze SQL
    - Diagnose slow-running queries
    - View the query execution statistics
    - Schedule a SQL trace to run during heavy workloads
    - Get advice on indexes
- SQL Profiler plug-in must be installed and configured for each Postgres database

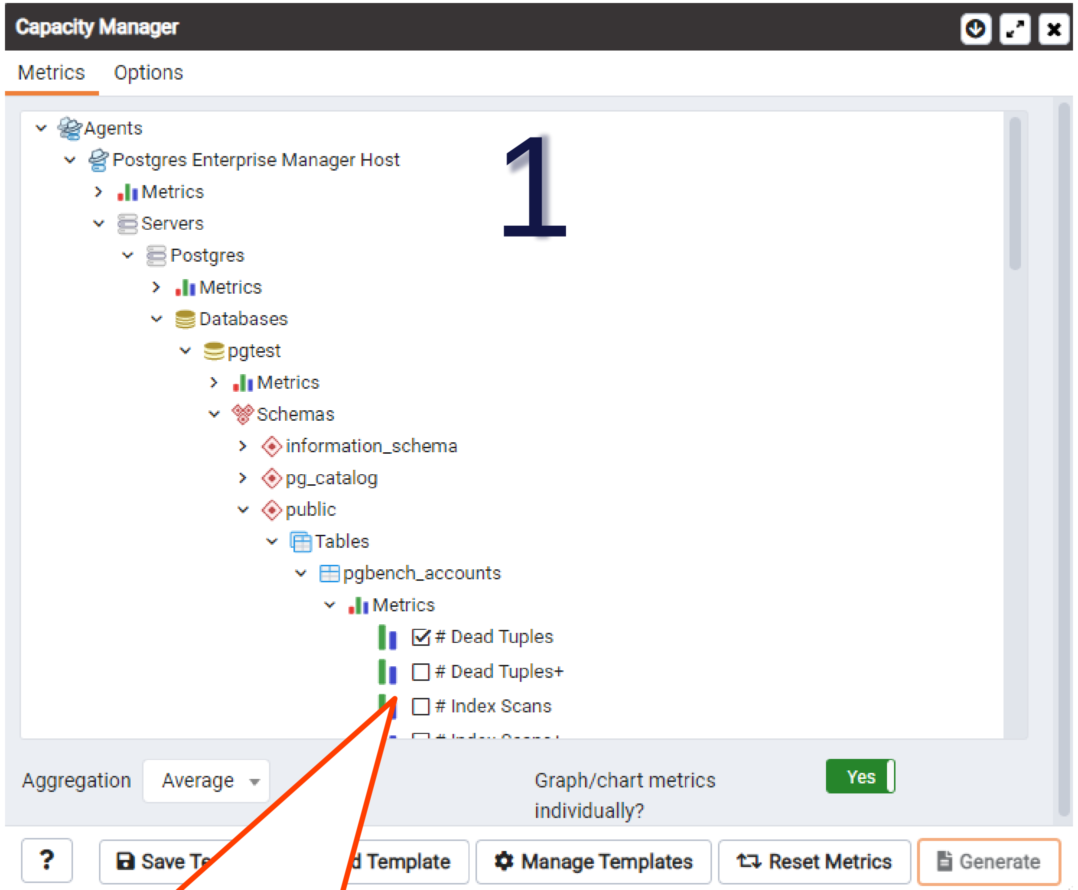
# Capacity Planning using Capacity Manager

- PEM contains built-in capabilities for performing database capacity planning
- Capacity planning helps by providing answers to questions like:
  - How much storage will my database need 6 months from now?
  - How fast is my database growing?
  - What objects are responsible for the growth in my database?
  - Will my server be able to support another database instance?
  - Is the performance of my database getting better, staying the same, or getting worse?

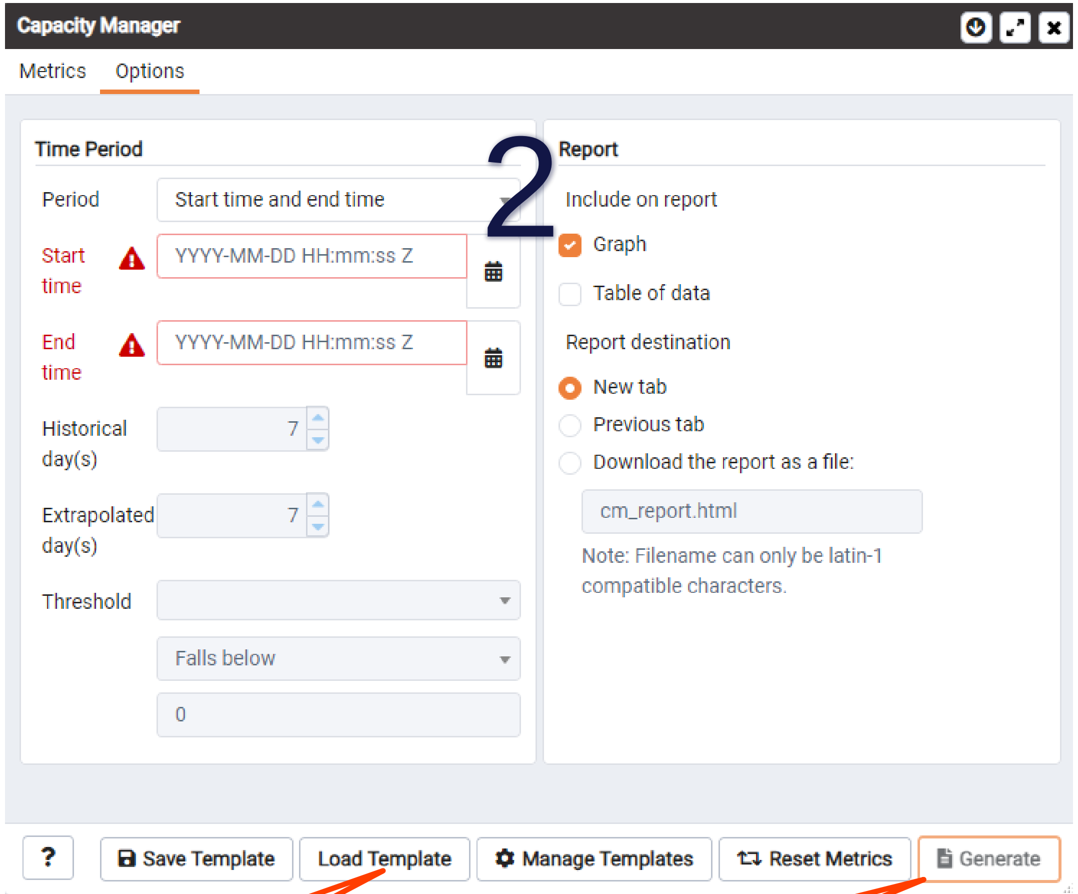
# Capacity Manager

- Analyzes collected statistics to generate a graph or table
- Displays the historical usage statistics of an object
- Can project the anticipated usage statistics for an object
- Analyze metrics for a specific:
  - Host/operating system
  - EDB Postgres Advanced Server or PostgreSQL server
  - Database
  - Database object (table, index, function etc.)
- You can choose a specific metric (or metrics) to include in the report
- You can also specify a start and end date for the Capacity Manager report

# Example - Capacity Manager



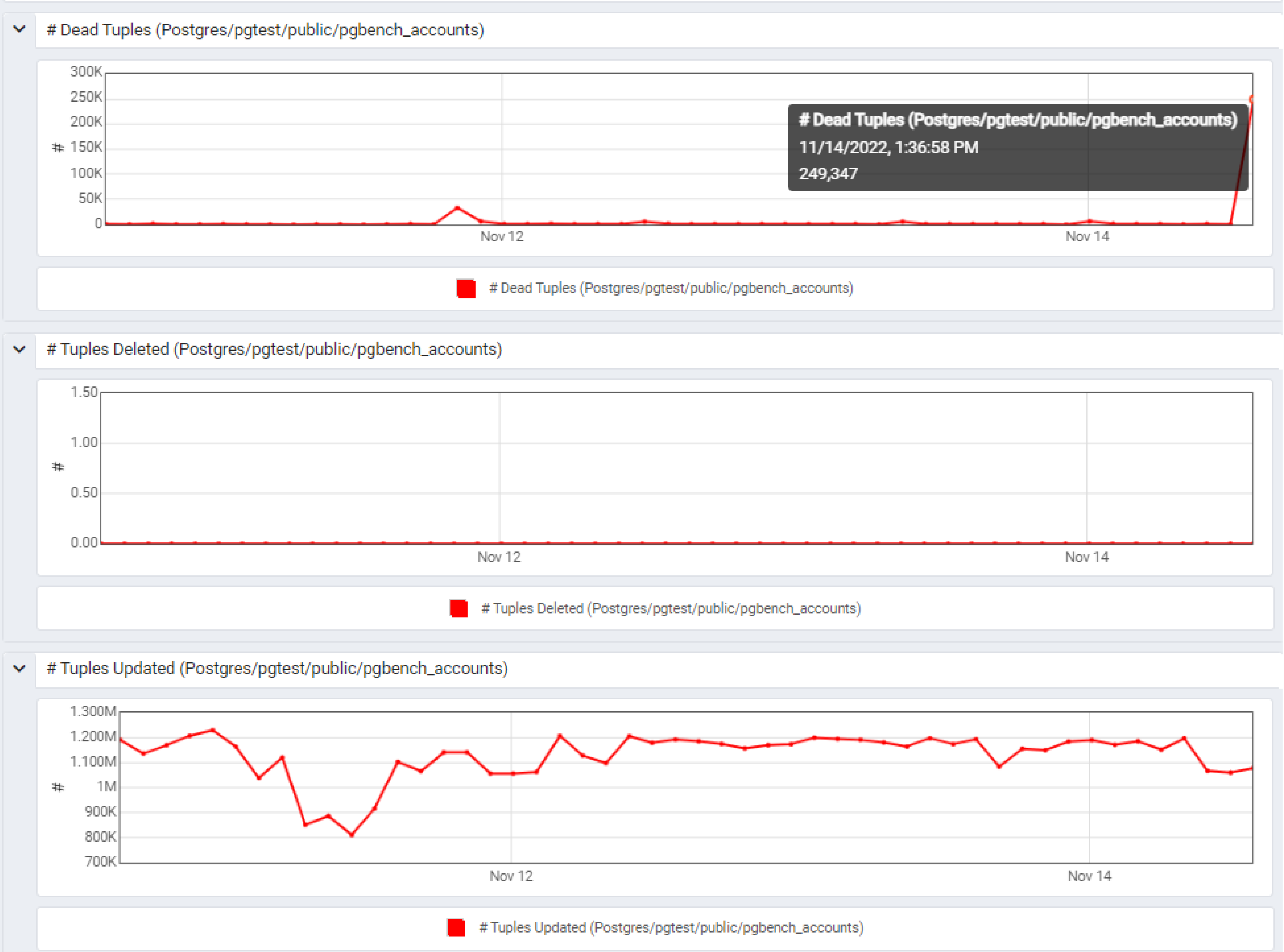

Select the metrics: Dead, Updated and Deleted Tuples




Buttons for Saving and Loading Templates

Click Generate for Report

# Sample Capacity Manager Report



# Instructor Demos

## Capacity Planner

# DRITA

# DRITA

- Dynamic Runtime Instrumentation Tools Architecture (DRITA):
  - Allows DBAs to query catalogs
  - Determines the wait events affecting performance
  - DRITA records wait event information while consuming minimal resources
  - Compares snapshots to evaluate the performance of a system
  - Snapshots identified by a unique ID number
  - Functions available for creating, viewing and comparing snapshots
  - DRITA consumes minimal system resources

# DRITA Snapshots

- A snapshot is a saved set of system performance data at a given point in time
- A unique ID number identifies each snapshot
- Snapshot ID numbers are used with DRITA reporting functions to return system performance statistics

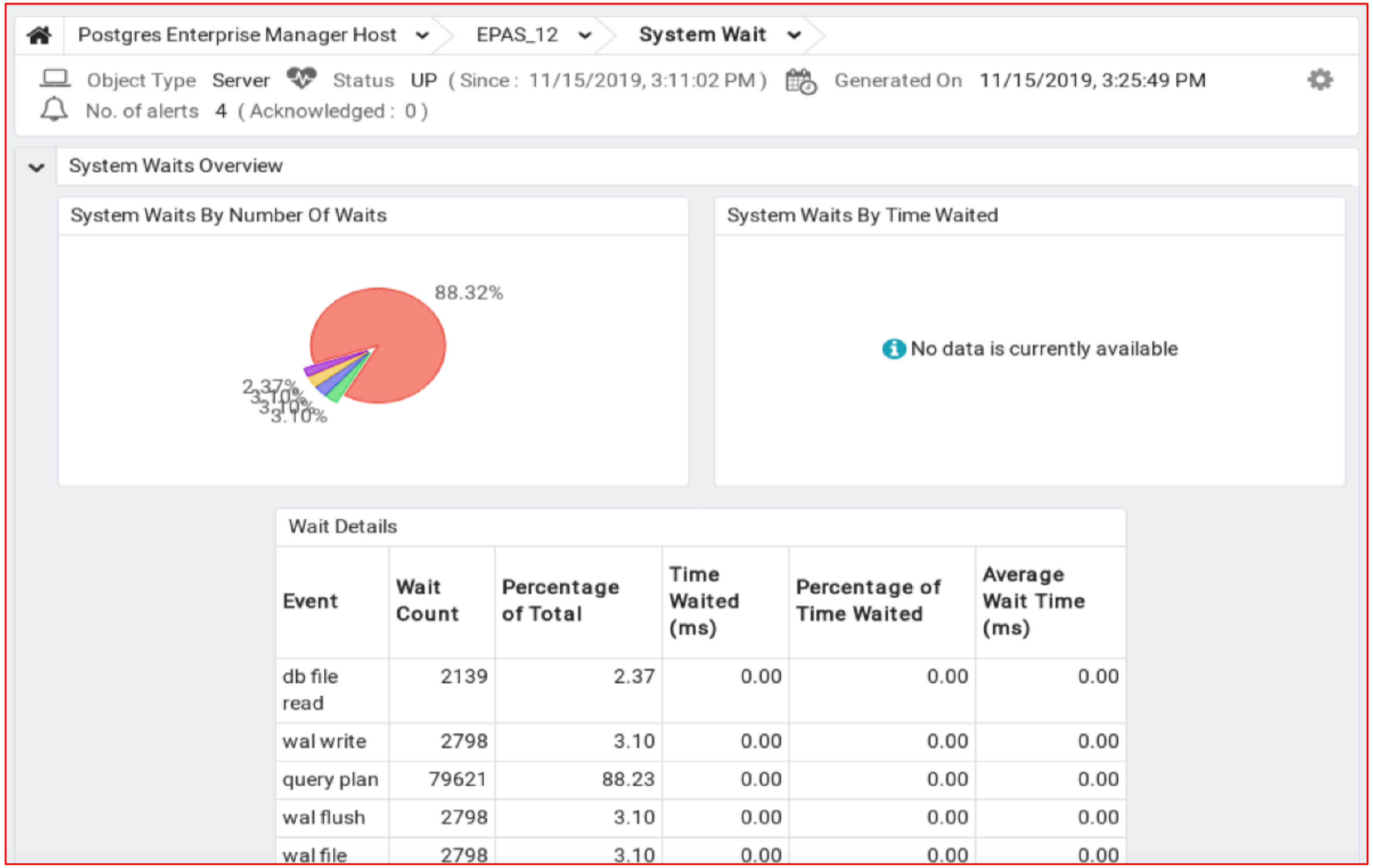
# Using DRITA Reports for Performance Tuning

- Review the top five events in a given report
- Look for any event that takes a disproportionately large percentage of resources
- Waits should be evaluated in the context of CPU usage and total time
- When evaluating events, watch for:
  - Checkpoint waits
  - WAL-related waits
  - SQL parse waits
  - db file random reads
  - db file random writes
  - btree random lock acquires

# Instructor Demos

## DRITA Snapshots

# PEM – System Wait Analysis Dashboard




| Event | Wait Count | Percentage of Total | Time Waited (ms) | Percentage of Time Waited | Average Wait Time (ms) |
|---|---|---|---|---|---|
| db file read | 2139 | 2.37 | 0.00 | 0.00 | 0.00 |
| wal write | 2798 | 3.10 | 0.00 | 0.00 | 0.00 |
| query plan | 79621 | 88.23 | 0.00 | 0.00 | 0.00 |
| wal flush | 2798 | 3.10 | 0.00 | 0.00 | 0.00 |
| wal file | 2798 | 3.10 | 0.00 | 0.00 | 0.00 |

# Instructor Demos

## PEM Wait Analysis Dashboard

# Module Summary

- Performance Tuning -  Overview
- Performance Monitoring using PEM
- Operating System Considerations
- Server Parameter Tuning
- Loading a Table into Memory
- Best Practices for Inserting Large Amount of Data
- Non – Durable Settings
- Tuning PostgreSQL Using PEM

# Lab Exercise - 1

1. Users are complaining about slower than normal performance on `edbstore` database
    - Tune **postgresql.conf** parameters for optimal performance based on the `edbstore` database size, largest table, and other necessary information collected from `edbstore` database
    - Change maximum concurrent connections on the cluster to 50

# Lab Exercise - 2

1. Write a statement to load `customers` table from `edbstore` database to the PostgreSQL buffer cache

# Lab Exercise - 3

1. `pg_test_fsync` can determine the fastest `wal_sync_method` on a specific system. Run this tool on your local machine and determine the best `wal_sync_method` settings for your system.

# Module - 4
# Routine Maintenance Tasks

# Module Objectives

- Updating Optimizer Statistics
- Handling Data Fragmentation using Routine Vacuuming
- Preventing Transaction ID Wraparound Failures
- Automatic Maintenance using Autovacuum
- Re-indexing in Postgres

## Database Maintenance

- Data files become fragmented as data is modified and deleted
- Database maintenance helps reconstruct the data files
- If done on time nobody notices but when not done everyone knows
- Must be done before you need it
- Improves performance of the database
- Saves database from transaction ID wraparound failu

# Maintenance Tools

- Maintenance thresholds can be configured using the PEM Client
- Postgres maintenance thresholds can be configured in **postgresql.conf**
- Manual scripts can be written watch stat tables like `pg_stat_user_tables`
- Maintenance commands:
  - `ANALYZE`
  - `VACUUM`
  - `CLUSTER`
- Maintenance command `vacuumdb` can be run from OS prompt
- Autovacuum can help in automatic database maintenance

## Optimizer Statistics

- Optimizer statistics play a vital role in query planning
- Not updated in real time
- Collects information for relations including size, row counts, average row size and row sampling
- Stored permanently in catalog tables
- The maintenance command `ANALYZE` updates the statistics
- Thresholds can be set using PEM Client to alert you when statistics are not collected on time

# Example - Updating Statistics

```
postgres=# CREATE TABLE testanalyze(id integer, name varchar);
CREATE TABLE
postgres=# INSERT INTO testanalyze VALUES(generate_series(1,10000), 'Sample');
INSERT 0 10000
postgres=# SELECT relname, reltuples FROM pg_class WHERE relname='testanalyze';
   relname   | reltuples
-------------+-----------
 testanalyze |         0
(1 row)

postgres=# ANALYZE testanalyze;
ANALYZE
postgres=# SELECT relname, reltuples FROM pg_class WHERE relname='testanalyze';
   relname   | reltuples
-------------+-----------
 testanalyze |     10000
(1 row)
```

## Data Fragmentation and Bloat

- Data is stored in data file pages
- An update or delete of a row does not immediately remove the row from the disk page
- Eventually this row space becomes obsolete and causes fragmentation and bloating
- Set PEM Alert for notifications

# Routine Vacuuming

- Obsoleted rows can be removed or reused using vacuuming
- Helps in shrinking data file size when required
- Vacuuming can be automated using autovacuum
- The `VACUUM` command locks tables in access exclusive mode
- Long running transactions may block vacuuming, thus it should be done during low usage times

## Vacuuming Commands

- When executed, the `VACUUM` command:
  - Can recover or reuse disk space occupied by obsolete rows
  - Updates data statistics
  - Updates the visibility map, which speeds up index-only scans
  - Protects against loss of very old data due to transaction ID wraparound
- The `VACUUM` command can be run in two modes:
  - `VACUUM`
  - `VACUUM FULL`

# Vacuum and Vacuum Full

- `VACUUM`
  - Removes dead rows and marks the space available for future reuse
  - Does not return the space to the operating system
  - Space is reclaimed if obsolete rows are at the end of a table
- `VACUUM FULL`
  - More aggressive algorithm compared to `VACUUM`
  - Compacts tables by writing a complete new version of the table file with no dead space
  - Takes more time
  - Requires extra disk space for the new copy of the table, until the operation completes

# VACUUM Syntax

- `VACUUM [ ( option [, ...] ) ] [ table_and_columns [, ...] ]`

  where option can be one of:

  - `FULL [ boolean ]`
  - `FREEZE [ boolean ]`
  - `VERBOSE [ boolean ]`
  - `ANALYZE [ boolean ]`
  - `DISABLE_PAGE_SKIPPING [ boolean ]`
  - `SKIP_LOCKED [ boolean ]`
  - `INDEX_CLEANUP [ boolean ]`
  - `TRUNCATE [ boolean ]`
  - `PARALLEL integer`

# Example - Vacuuming

```
edb=# CREATE TABLE testvac (id numeric, name varchar2);
CREATE TABLE
edb=# INSERT INTO testvac VALUES(generate_series(1,10000),'Sample');
INSERT 0 10000
edb=# SELECT pg_size_pretty(pg_relation_size('testvac'));
 pg_size_pretty
----------------
 440 kB
(1 row)

edb=# UPDATE testvac SET name='Sample';
UPDATE 10000
edb=# SELECT pg_size_pretty(pg_relation_size('testvac'));
 pg_size_pretty
----------------
 872 kB
(1 row)

edb=#
```

# Example – Vacuuming (continued)

```
edb=# VACUUM testvac;
VACUUM
edb=# UPDATE testvac SET name='Sample';
UPDATE 10000
edb=# SELECT pg_size_pretty(pg_relation_size('testvac'));
 pg_size_pretty
----------------
 872 kB
(1 row)

edb=# VACUUM FULL testvac;
VACUUM
edb=# SELECT pg_size_pretty(pg_relation_size('testvac'));
 pg_size_pretty
----------------
 440 kB
(1 row)

edb=#
```

# Preventing Transaction ID Wraparound Failures

- MVCC depends on transaction ID numbers
- Transaction IDs have limited size (32 bits at this writing)
- A cluster that runs for a long time (more than 4 billion transactions) would suffer transaction ID wraparound
- This causes a catastrophic data loss
- To avoid this problem, every table in the database must be vacuumed at least once for every two billion transactions

# Vacuum Freeze

- `VACUUM FREEZE` will mark rows as frozen
- Postgres reserves a special XID, `FrozenTransactionId`
- `FrozenTransactionId` is always considered older than every normal XID
- `VACUUM FREEZE` replaces transaction IDs with `FrozenTransactionId`, thus rows will appear to be "in the past"
- `vacuum_freeze_min_age` controls when a row will be frozen
- `VACUUM` normally skips pages without dead row versions, but some rows may need `FREEZE`
- `vacuum_freeze_table_age` controls when a whole table must be scanned

## The Visibility Map

- Each heap relation has a Visibility Map which keeps track of which pages contain only tuples
- Stored at `<relfilenode>_vm`
- Helps vacuum to determine whether pages contain dead rows
- Can also be used by index-only scans to answer queries
- `VACUUM` command updates the visibility map
- The visibility map is vastly smaller, so can be cached easily

# vacuumdb Utility

- The `VACUUM` command has a command-line executable wrapper called `vacuumdb`
- `vacuumdb` can `VACUUM` all databases using a single command
- Syntax:
  - `vacuumdb [OPTION]... [DBNAME]`
- Available options can be listed using:
  - `vacuumdb --help`

# Autovacuuming

- Highly recommended feature of Postgres
- It automates the execution of `VACUUM`, `FREEZE` and `ANALYZE` commands
- Autovacuum consists of a launcher and many worker processes
- A maximum of `autovacuum_max_workers` worker processes are allowed
- Launcher will start one worker within each database every `autovacuum_naptime` seconds
- Workers check for inserts, updates and deletes and execute `VACUUM` and/or `ANALYZE` as needed
- `track_counts` must be set to `TRUE` as autovacuum depends on statistics
- Temporary tables cannot be accessed by autovacuum

# Autovacuuming Parameters

## Autovacuum Launcher Process

- **autovacuum**

## Autovacuum Worker Processes

- **autovacuum_max_workers**
- **autovacuum_naptime**

## Vacuuming Thresholds

- **autovacuum_vacuum_scale_factor**
- **autovacuum_vacuum_threshold**
- **autovacuum_analyze_scale_factor**
- **autovacuum_analyze_threshold**
- **autovacuum_vacuum_insert_scale_threshold**
- **autovacuum_vacuum_insert_threshold**
- **autovacuum_freeze_max_age**

# Per-Table Thresholds

- Autovacuum workers are resource intensive
- Table-by-table `autovacuum` parameters can be configured for large tables
- Configure the following parameters using `ALTER TABLE` or `CREATE TABLE`:
  - `autovacuum_enabled`
  - `autovacuum_vacuum_threshold`
  - `autovacuum_vacuum_scale_factor`
  - `autovacuum_analyze_threshold`
  - `autovacuum_analyze_scale_factor`
  - `autovacuum_vacuum_insert_scale_threshold`
  - `autovacuum_vacuum_insert_threshold`
  - `autovacuum_freeze_max_age`

EDB

## Routine Reindexing

- Indexes are used for faster data access
- `UPDATE` and `DELETE` on a table modify underlying index entries
- Indexes are stored on data pages and become fragmented over time
- `REINDEX` rebuilds an index using the data stored in the index's table
- Time required depends on:
  - Number of indexes
  - Size of indexes
  - Load on server when running command

# When to Reindex

- There are several reasons to use `REINDEX`:
  - An index has become "bloated", meaning it contains many empty or nearly-empty pages
  - You have altered a storage parameter (such as fillfactor) for an index
  - An index built with the `CONCURRENTLY` option failed, leaving an "invalid" index
- Syntax:

```
=> REINDEX [ ( VERBOSE ) ] { INDEX | TABLE | SCHEMA | DATABASE |
SYSTEM } [ CONCURRENTLY ] name
```

# Module Summary

- Updating Optimizer Statistics
- Handling Data Fragmentation using Routine Vacuuming
- Preventing Transaction ID Wraparound Failures
- Automatic Maintenance using Autovacuum
- Re-indexing in Postgres

# Lab Exercise - 1

1. While monitoring table statistics on the `edbstore` database, you found that some tables are not automatically maintained by autovacuum. You decided to perform manual maintenance on these tables. Write a SQL script to perform the following maintenance:
    - Reclaim obsolete row space from the `customers` table.
    - Update statistics for `emp` and `dept` tables.
    - Mark all the obsolete rows in the `orders` table for reuse.

2. Execute the newly created maintenance script on `edbstore` database.

# Lab Exercise - 2

1. The composite index named `ix_orderlines_orderid` on (`orderid, orderlineid`) columns of the `orderlines` table is performing very slowly. Write a statement to reindex this index for better performance.

# Conclusion

# Course Summary

- Introduction
- SQL Tuning
- Performance Tuning
- Database Maintenance

# Thank You

Feedback and Questions: training@enterprisedb.com