Monitoring PostgreSQL Database Server

eG Innovations Product Documentation

www.eginnovations.com

Total Performance Visibility

Table of Contents

Table of Figures

Chapter 1: Introduction

PostgreSQL , often simply Postgres , is an object- relational database management system (ORDBMS) available for many platforms including Linux, FreeBSD, Solaris, Microsoft Windows and Mac OS X. It implements the majority of the SQL:2008 standard, is ACID- compliant, is fully transactional (including all DDL statements), has extensible data types, operators, and indexes, and has a large number of extensions written by third parties.

Owing to its ability to operate on heterogeneous platforms, the PostgreSQL has of late become the preferred backend for many mission- critical service offerings. A second's non- availability of the server, a sudden or steady erosion of free space in one/more of its tablespaces, ineffective caching by the server, and intense locking can cause serious harm to not only the performance of the PostgreSQL server in question, but also the services that rely on it. Continuous monitoring of the database server and prompt detection and resolution of anomalies is hence imperative. For continuously monitoring the PostgreSQL database server, the eG Enterprise provides a specialized monitoring model, which is explained in the upcoming topics.

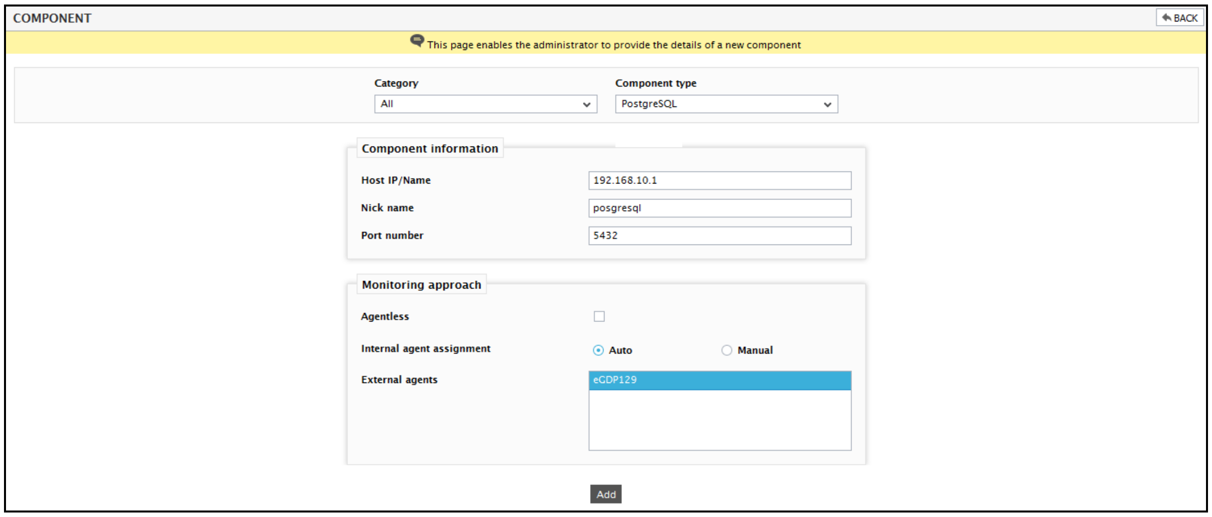
Chapter 2: How does eG Enterprise Monitor PostgreSQL Server?

eG Enterprise employs *agent-based* or *agentless* techniques to monitor the PostgreSQL server. A eG agent periodically checks the status of critical database operations and proactively report problems.

2.1 Managing the PostgreSQL Server

The eG Enterprise cannot automatically discover the PostgreSQL server. This implies that you need to manually add the component for monitoring. Remember that the eG Enterprise automatically manages the components that are added manually. To manage a PostgreSQL Server component, do the following:

1. Log into the eG administrative interface.
2. eG Enterprise cannot automatically discover the PostgreSQL component. You need to manually add the server using the **COMPONENTS** page (see Figure 2.1) that appears when the Infrastructure -> Components -> Add/Modify menu sequence is followed. Remember that components manually added are managed automatically.



This page enables the administrator to provide the details of a new component

Category: All
Component type: PostgreSQL

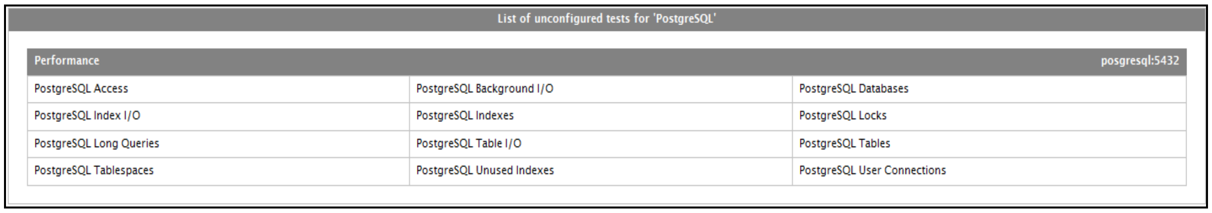
Component information
Host IP/Name: 192.168.10.1
Nick name: posgresql
Port number: 5432

Monitoring approach
Agentless: ☐
Internal agent assignment: [✓ Auto] Manual
External agents: eGDP129

Figure 2.1: Adding the PostgreSQL

3. Specify the **Host IP** and the **Nick name** of the Oracle Cluster in Figure 2.1. The **Port number** will be set as 5432 by default. If the server is listening on a different port, then override this settings.

4. Then, click the **Add** button to register the changes.
5. When you attempt to sign out, a list of unconfigured tests appears (see Figure 2.2).



List of unconfigured tests for 'PostgreSQL'

Performance		posgresql:5432
PostgreSQL Access	PostgreSQL Background I/O	PostgreSQL Databases
PostgreSQL Index I/O	PostgreSQL Indexes	PostgreSQL Locks
PostgreSQL Long Queries	PostgreSQL Table I/O	PostgreSQL Tables
PostgreSQL Tablespaces	PostgreSQL Unused Indexes	PostgreSQL User Connections

Figure 2.2: List of tests to be configured for the PostgreSQL

6. Click on the **PostgreSQL Access** test to configure it. To know how to configure the test, refer to the Section **3.4.3**.
7. Once all the tests are configured, signout of the eG administrative interface.

Chapter 3: Monitoring the PostgreSQL Server

eG Enterprise offers a 100%, web-based PostgreSQL monitoring model (see Figure 3.1) that provides indepth insights into the performance and problems related to the PostgreSQL database server. **This model can be used for monitoring PostgreSQL version 9.0 onwards**.

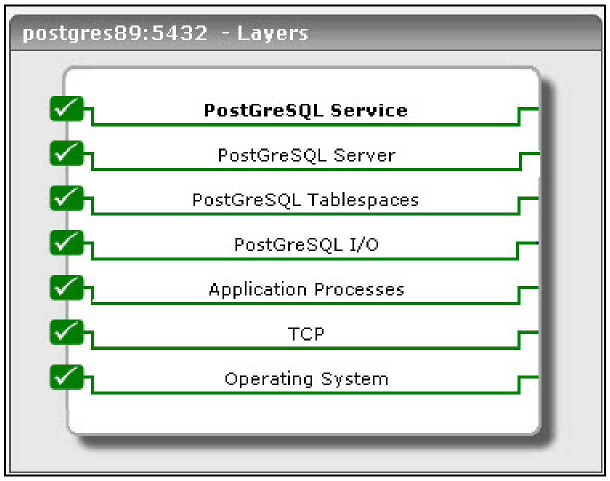


Figure 3.1: Layermodel of the PostgresSQL database server

This model can be configured to employ *agent-based* or *agentless* techniques to periodically check the status of critical database operations and proactively report problems. These metrics enable database administrators to find quick and accurate answers to the following performance queries:

- Is the database server available? If so, how quickly does it repond to client queries?
- Is the buffer cache utilized optimally, or are requests for heap blocks and index blocks being increasingly serviced by direct disk accesses?
- Is any tablespace running low on free space? If so, which one?
- How well does the background writer perform checkpointing? Is too much I/O load being imposed by the writer in the process of checkpointing?
- Are too many rollbacks occurring on any database? If so, which one?
- Are indexes used effectively?
- Are there any useless/unused indexes on the server? Which ones are these?
- Have too many sequential scans occurred on any table?
- Are inserts, updates, and deletes happening too slowly on any table?

- Is any table experiencing extreme or major issues while querying data from the server?
- Does any user have too many idle connections on the server?
- Is any user's connection waiting for a locked resource to be released?
- Are too many locks being currently held on the server? Which lock mode is the maximum?
- Are any queries running for too long a time on the server? If so, which ones are these?

The sections that follow will deal with the top four layers of Figure 3.1 as the other layers have already been dealt with in the *Monitoring Unix and Windows* servers document.

3.1 The PostGreSQL I/O Layer

Use the tests mapped to this layer to figure out how the server performs caching and how well the buffer cache is utilized. Inadequacies in the cache size are thus revealed.

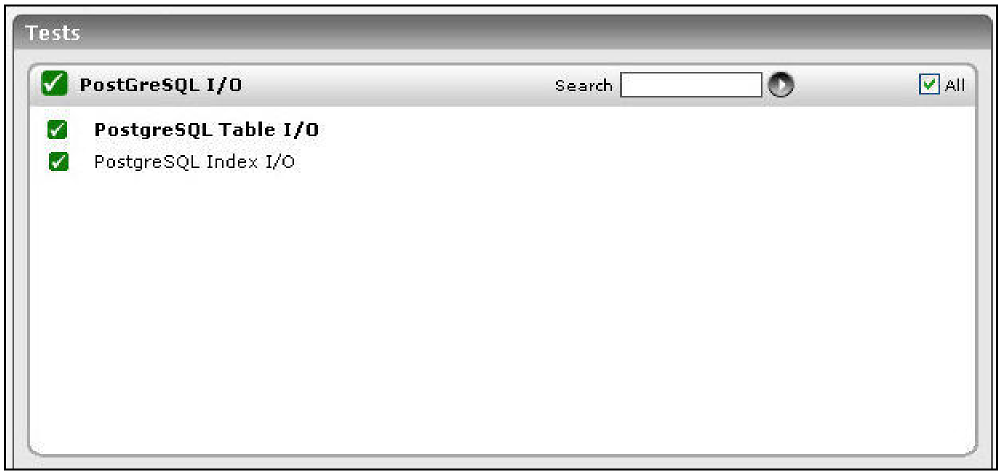


Figure 3.2: The tests mapped to the PostGreSQL I/O

3.1.1 PostgreSQL Table I/O Test

In PostgreSQL, data is stored in tables, and tables are grouped into databases. Each table is stored in its own disk file. The contents of a table are stored in pages. A table can span many pages, depending upon the length of the row data in the table. A page that contains row data is called a heap block. As indexes are also stored in page files, a page that contains index data is called an index block.

Typically, in PostgreSQL, most disk I/O is performed on a page-by-page basis. To minimize disk I/O, PostgreSQL creates an in-memory data structure known as the buffer cache to which the frequently accessed data is stored. The buffer cache is organized as a collection of 8K pages—each page in the buffer cache corresponds to a page in some page file. The buffer cache is shared between all processes servicing a given database.

When you select a row from a table, PostgreSQL will read the heap block that contains the row into the buffer cache. If there is not enough free space in the cache, PostgreSQL will move some other block out of the cache. If a block being removed from the cache has been modified, it will be written back out to disk; otherwise, it will simply be discarded. Index blocks are also buffered in a similar manner.

If the buffer cache is not sized right, it may not be able to hold enough heap or index blocks to serve subsequent requests. If queries do not find the heap/index blocks they need in the buffer cache, they will be forced to access the disk directly to retrieve data. As direct disk accesses are I/O-intensive operations, they may cause serious performance degradations if not nipped in the bud!

Using the **PostgreSQL Table I/O** test, you can continuously monitor the heap blocks read from the tables in configured databases and index blocks read from the indexes that correspond to those tables. In the process, you can understand how the buffer cache serviced these read requests and learn of ineffective cache usage early, so that you can investigate the reasons for the same (whether/not it is owing to an under-sized cache) and initiate appropriate remedial action.

Target of the test : PostgreSQL server

Agent deploying the test: An internal/remote agent

Outputs of the test : One set of results for every table (and corresponding index) in every database that is configured for monitoring in the target PostgreSQL server

Configurable parameters for the test

1. **TEST PERIOD** – How often should the test be executed.
2. **HOST** – The IP address of the server.
3. **PORT** – The port on which the server is listening. The default port is 5432.
4. **USER** – In order to monitor a PostgreSQL server, you need to manually create a special database user account in every PostgreSQL database instance that requires monitoring. When doing so, ensure that this user is vested with the *superuser* privileges. The sample script we recommend for user creation for eG monitoring is:

   CREATE ROLE eguser LOGIN
   ENCRYPTED PASSWORD {'eguser password'}
   SUPERUSER NOINHERIT NOCREATEDB NOCREATEROLE;

The name of this user has to be specified in the **USERNAME** text box.

5. **PASSWORD -** The password associated with the above user name (can be 'NULL'). Here, 'NULL' means that the user does not have any password.
6. **CONFIRM PASSWORD** – Confirm the **PASSWORD** (if any) by retyping it here.
7. **DBNAME -** The name of the database to connect to. The default is "postgres".
8. **INCLUDE DB -** Specify a comma-separated list of databases that you wish to monitor.
9. **EXCLUDE DB -** Specify a comma-separated list of databases that need to be excluded from monitoring. By default, this is set to *rdsadmin*.

Note:

If you are monitoring a PostgreSQL server on the AWS EC2 cloud, then **make sure that you do not remove 'rdsadmin' from the EXCLUDE DB list**.

10. **SSL -** The name of this user has to be specified in the **USERNAME** text box.

Measurements made by the test

Measurement	Description	Measurement Unit	Interpretation
Heap blocks read:	Indicates the rate at which the heap blocks are read from this table.	Reads/Sec	
Heap blocks hit:	Indicates the number of heap block requests to this table that were serviced by the buffer cache during the last measurement period.	Number	Ideally, the value of this measure should be high.
Heap hit ratio:	Indicates the ratio of the heap block read requests to this table to the heap block requests found in the	Percent	Ideally, the value of this measure should be high. A low value is indicative ineffective cache usage, which in turn can increase disk I/O and degrade server performance.

Measurement	Description	Measurement Unit	Interpretation
	buffer cache.		One of the most common reasons for a low cache hit ratio is small cache size. In such a case, you can consider increasing the cache size. There are two ways that you can adjust the size of the cache. You could edit PostgreSQL's configuration file (**$PGDATA/postgresql.conf**) and change the **shared_buffers** variable therein. Alternatively, you can override the **shared_ buffers** configuration variable when you start the postmaster. A sample command for implementing a **shared_buffers** override while starting the postmaster is given below: *pg_start -o "-B 65" -l /tmp/pg.log* If increasing the cache size also does not help, then, you can include a limit clause in your queries to select a sub-set of the queried tables and add them to the cache.
Index block reads:	Indicates the rate at which the index blocks were read from the indexes of this table.	Reads/Sec	
Blocks hit:	Indicates the number of read requests to the indexes of this table that were found in the buffer cache during the last measurement period.	Number	Ideally, the value of this measure should be high.

Measurement	Description	Measurement Unit	Interpretation
Block hit ratio:	Indicates the percentage of index block requests to the indexes of this table that were served by the buffer cache.	Percent	Ideally, the value of this measure should be high. A low value is indicative of ineffective cache usage, which in turn can increase disk I/O and degrade server performance. One of the most common reasons for a low cache hit ratio is small cache size. In such a case, you can consider increasing the cache size. There are two ways that you can adjust the size of the cache. You could edit PostgreSQL's configuration file (**$PGDATA/postgresql.conf**) and change the shared_buffers variable therein. Alternatively, you can override the **shared_ buffers** configuration variable when you start the postmaster. A sample command for implementing a **shared_buffers** override while starting the postmaster is given below: *pg_start -o "-B 65" -l /tmp/pg.log* If increasing the cache size also does not help, then, you can include a limit clause in your queries to select a sub-set of the queried tables and add them to the cache.

3.1.2 PostgreSQL Index I/O Test

Indexes are buffered in the same way as tables are. Therefore, if too many index blocks are not found in the buffer cache, disk I/O increases, causing the overall performance of the PostgreSQL server to suffer. It is hence imperative to monitor how the cache services index block read requests.

The **PostgreSQL Index I/O** test helps monitor each index in a database for read requests. In the process, the test reveals how the buffer cache serviced these read requests and provides early pointers to ineffective cache usage, so that you can investigate the reasons for the same (whether/not it is owing to an under-sized cache) and initiate appropriate remedial action.

Target of the test : PostgreSQL server

Agent deploying the test: An internal/remote agent

Outputs of the test : One set of results for every in index in every database that is configured for monitoring in the target PostgreSQL server

Configurable parameters for the test

1. **TEST PERIOD** – How often should the test be executed
2. **HOST** – The IP address of the server
3. **PORT** – The port on which the server is listening. The default port is 5432.
4. **USER** – In order to monitor a PostgreSQL server, you need to manually create a special database user account in every PostgreSQL database instance that requires monitoring. When doing so, ensure that this user is vested with the *superuser* privileges. The sample script we recommend for user creation for eG monitoring is:

   CREATE ROLE eguser LOGIN
   ENCRYPTED PASSWORD {'eguser password'}
   SUPERUSER NOINHERIT NOCREATEDB NOCREATEROLE;

   The name of this user has to be specified in the **USERNAME** text box.
5. **PASSWORD** - The password associated with the above user name (can be 'NULL'). Here, 'NULL' means that the user does not have any password.
6. **CONFIRM PASSWORD** – Confirm the **PASSWORD** (if any) by retyping it here.
7. **DBNAME** - The name of the database to connect to. The default is "postgres".
8. **INCLUDE DB** - Specify a comma-separated list of databases that you wish to monitor.
9. **EXCLUDE DB** - Specify a comma-separated list of databases that need to be excluded from monitoring. By default, this is set to *rdsadmin*.

   Note:

	If you are monitoring a PostgreSQL server on the AWS EC2 cloud, then **make sure that you do not remove 'rdsadmin' from the EXCLUDE DB list**.
10.	**SSL -** The name of this user has to be specified in the **USERNAME** text box.

Measurements made by the test

Measurement	Description	Measurement Unit	Interpretation
Index block reads:	Indicates the rate at which the index blocks were read from this index during the last measurement period.	Reads/Sec	
Blocks hit:	Indicates the number of read requests to this index that were found in the buffer cache during the last measurement period.	Number	Ideally, the value of this measure should be high.
Hit ratio:	Indicates the percentage of index block requests to this index that were served by the buffer cache.	Percent	Ideally, the value of this measure should be high. A low value is indicative ineffective cache usage, which in turn can increase disk I/O and degrade server performance. One of the most common reasons for a low cache hit ratio is small cache size. In such a case, you can consider increasing the cache size. There are two ways that you can adjust the size of the cache. You could edit PostgreSQL's configuration file (**$PGDATA/postgresql.conf**) and change the shared_buffers variable therein. Alternatively, you can override the **shared_ buffers**

Measurement	Description	Measurement Unit	Interpretation
			configuration variable when you start the postmaster. A sample command for implementing a **shared_buffers** override while starting the postmaster is given below: *pg_start -o "-B 65" -l /tmp/pg.log* If increasing the cache size also does not help, then, you can include a limit clause in your queries to select a sub-set of the queried tables and add them to the cache.

3.2 The PostGreSQL Tablespaces Layer

To know whether any tablespace has been excessively utilized, use the test mapped to this layer.

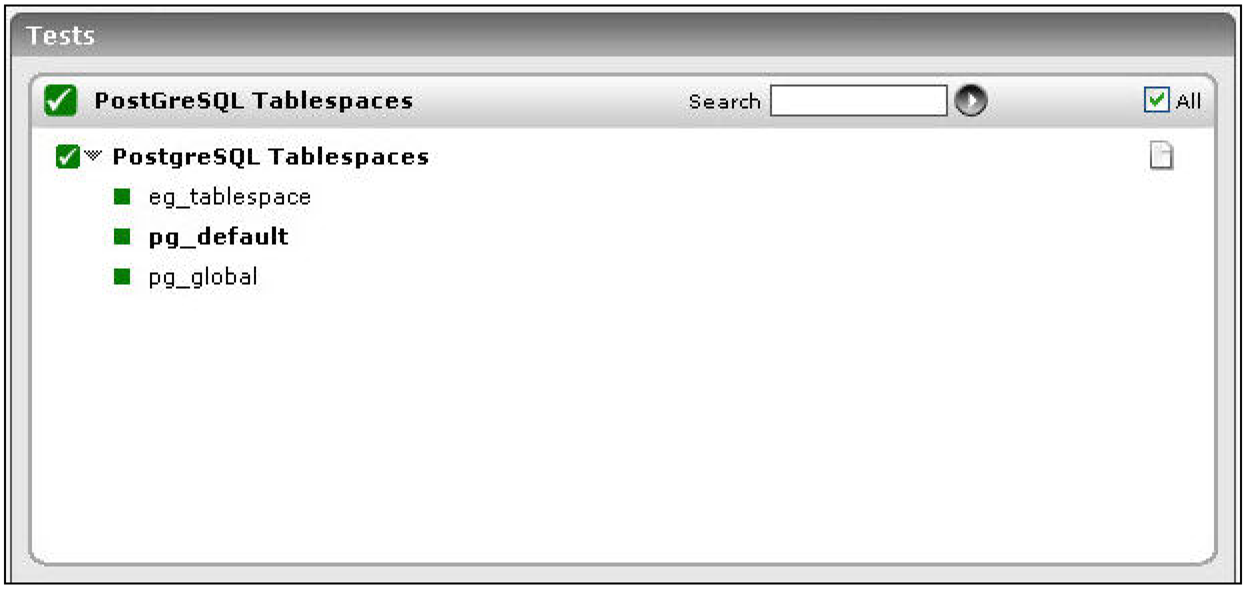


Figure 3.3: The test mapped to the PostGreSQL Tablespaces layer

3.2.1 PostgreSQL Tablespaces Test

Tablespaces in PostgreSQL allow database administrators to define locations in the file system where the files representing database objects can be stored. Once created, a tablespace can be referred to by name when creating database objects.

By using tablespaces, an administrator can control the disk layout of a PostgreSQL installation. This is useful in at least two ways. First, if the partition or volume on which the cluster was initialized runs out of space and cannot be extended, a tablespace can be created on a different partition and used until the system can be reconfigured.

Second, tablespaces allow an administrator to use knowledge of the usage pattern of database objects to optimize performance. For example, an index which is very heavily used can be placed on a very fast, highly available disk, such as an expensive solid state device. At the same time a table storing archived data which is rarely used or not performance critical could be stored on a less expensive, slower disk system.

Tablespaces should be adequately sized. If not, the tablespaces may not be able to accomodate many critical database objects, thereby causing the performance of the database to suffer. Continuous monitoring of tablespace size and usage is hence important. The **PostgreSQL Tablespaces** test does just that. This test auto-discovers tablespaces managed by this PostgreSQL server and reports how well the tablespace has been utilized.

Target of the test : PostgreSQL server

Agent deploying the test: An internal/remote agent

Outputs of the test : One set of results for every tablespace in every database that is configured for monitoring in the target PostgreSQL server

Configurable parameters for the test

1. **TEST PERIOD** – How often should the test be executed.
2. **HOST** – The IP address of the server.
3. **PORT** – The port on which the server is listening. The default port is 5432.
4. **USER** – In order to monitor a PostgreSQL server, you need to manually create a special database user account in every PostgreSQL database instance that requires monitoring. When doing so, ensure that this user is vested with the *superuser* privileges. The sample script we recommend for user creation for eG monitoring is:

CREATE ROLE eguser LOGIN
ENCRYPTED PASSWORD {'eguser password'}
SUPERUSER NOINHERIT NOCREATEDB NOCREATEROLE;

The name of this user has to be specified in the **USERNAME** text box.

5. **PASSWORD** - The password associated with the above user name (can be 'NULL'). Here, 'NULL' means that the user does not have any password.
6. **CONFIRM PASSWORD** – Confirm the **PASSWORD** (if any) by retyping it here.
7. **DBNAME** - The name of the database to connect to. The default is "postgres".
8. **INCLUDE DB** - Specify a comma-separated list of databases that you wish to monitor.
9. **EXCLUDE DB** - Specify a comma-separated list of databases that need to be excluded from monitoring.
10. **SSL** - The name of this user has to be specified in the **USERNAME** text box.

Measurements made by the test

Measurement	Description	Measurement Unit	Interpretation
Tablespace size:	Indicates the amount of space currently used in this tablespace.	MB	A high value of this measure indicates that the table consumes a large chunk of space in the tablespace which may cause serious performance issues ranging from slowdown to shutdowns of this database.

3.3 The PostGreSQL Server Layer

Using the tests mapped to this layer, you can determine:

- Whether/not the background writer minimizes the I/O load on the server;
- How well the server handles the transaction load to it, and whether any processing pain-points can be noticed;
- Whether/ not the indexes are properly used;
- The number and names of unused indexes (if any);
- The count of sequential scans and index scans that occurred per table and the rows that were returned in the process.

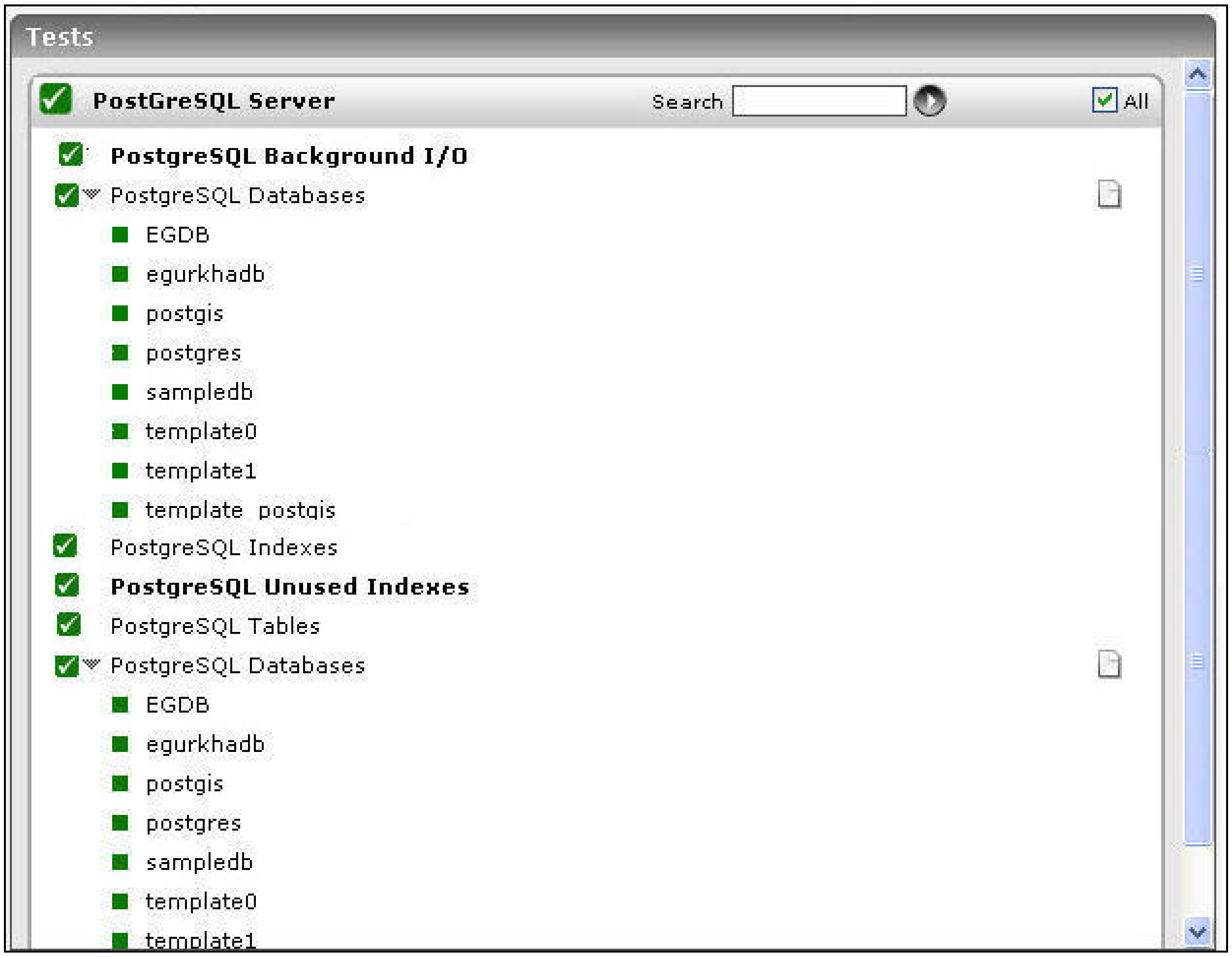


Figure 3.4: The tests mapped to the PostGreSQL Server layer

3.3.1 PostgreSQL Background I/O Test

Checkpoints are points in the sequence of transactions at which it is guaranteed that the heap and index data files have been updated with all information written before the checkpoint. At checkpoint time, all dirty data pages are flushed to disk and a special checkpoint record is written to the log file. In the event of a crash, the crash recovery procedure looks at the latest checkpoint record to determine the point in the log (known as the redo record) from which it should start the REDO operation. Any changes made to data files before that point are guaranteed to be already on disk.

The checkpoint requirement of flushing all dirty data pages to disk can cause a significant I/O load. To minimize this I/O, there is a separate server process called the background writer in PostgreSQL, whose sole function is to issue writes of “dirty” shared buffers. The background writer will continuously trickle out dirty pages to disk, so that only a few pages will need to be forced out when checkpoint time arrives, instead of the storm of dirty-buffer writes that formerly occurred at each checkpoint. However, there is a net overall increase in I/O load, because where a repeatedly-dirtied page might before have been written only once per checkpoint interval, the background writer might write it several times in the same interval.

You hence need to continuously track how often the background writer performs checkpointing and how much I/O load it imposes on the server, so that you can proactively detect potential overload conditions, appropriately fine-tune the checkpointing activity performed by the background writer to minimize the I/O, and thus prevent the performance degradation that may otherwise occur on the server. The **PostgreSQL Background I/O** test helps achieve all of the above. In the process, the test also reports useful statistics related to shared buffers.

Target of the test : PostgreSQL server

Agent deploying the test: An internal/remote agent

Outputs of the test : One set of results for the target PostgreSQL server

Configurable parameters for the test

1. **TEST PERIOD** – How often should the test be executed.
2. **HOST** – The IP address of the server.
3. **PORT** – The port on which the server is listening. The default port is 5432.
4. **USER** – In order to monitor a PostgreSQL server, you need to manually create a special database user account in every PostgreSQL database instance that requires monitoring. When doing so, ensure that this user is vested with the *superuser* privileges. The sample script we recommend for user creation for eG monitoring is:

   CREATE ROLE eguser LOGIN
   ENCRYPTED PASSWORD {'eguser password'}
   SUPERUSER NOINHERIT NOCREATEDB NOCREATEROLE;

   The name of this user has to be specified in the **USERNAME** text box.
5. **PASSWORD** - The password associated with the above user name (can be 'NULL'). Here, 'NULL' means that the user does not have any password.
6. **CONFIRM PASSWORD** – Confirm the **PASSWORD** (if any) by retyping it here.
7. **DBNAME** - The name of the database to connect to. The default is "postgres".
8. **SSL** - The name of this user has to be specified in the **USERNAME** text box.

Measurements made by the test

Measurement	Description	Measurement Unit	Interpretation
Checkpoint requests:	Indicates the number of checkpoint requests received by the server during the last measurement period.	Number	A checkpoint request is generated every checkpoint_ segments log segments, or every checkpoint_ timeout seconds, whichever comes first. While checkpoint_ segments denotes the maximum number of log file segments between automatic WAL checkpoints, the checkpoint_ timeout indicates the maximum time between WAL checkpoints. The default settings are 3 segments and 300 seconds (5 minutes), respectively. Reducing checkpoint_ segments and/or *checkpoint_ timeout* causes checkpoints to occur more often. This allows faster after- crash recovery (since less work will need to be redone). However, one must balance this against the increased cost of flushing dirty data pages more often. If *full_ page_ writes* is set (as is the default), there is another factor to consider. To ensure data page consistency, the first modification of a data page after each checkpoint results in logging the entire page content. In that case, a smaller checkpoint interval increases the volume of output to the WAL log, partially negating the goal of using a smaller interval, and in any case causing

Measurement	Description	Measurement Unit	Interpretation
			more disk I/O. Checkpoints are fairly expensive, first because they require writing out all currently dirty buffers, and second because they result in extra subsequent WAL traffic as discussed above. It is therefore wise to set the checkpointing parameters high enough that checkpoints don't happen too often.
Check point time outs:	Indicates the number of scheduled checkpoints that did not occur even after the checkpoint_ timeout setting was violated during the last measurement period.	Number	Ideally, the value of this measure should be low. A consistent increase in this value is a cause of concern, as it indicates that checkpoints are not occurring in the desired frequency. This in turn will significantly slowdown after- crash recovery, as more work will have to be redone.
Buffers freed:	Indicates the total number of buffers that were released for re-use from the buffer cache during the last measurement period, when the checkpoint_ segments setting was violated; this typically causes the background writer to automatically write dirty buffers to the disk.	Number	A high value is desired for this measure. A low value could indicate a checkpointing bottleneck, owing to which the background writer is unable to write updated index and heap files to the disk at an optimal rate. In such cases, the buffer cache may not have adequate free buffers to service subsequent write requests. This is a cause for concern in write-intensive database environments.
Buffers cleaned:	Indicates the number of buffer that were written to the disk during the last	Number	The background writer typically stalls some other process for a

Measurement	Description	Measurement Unit	Interpretation
	measurement period in anticipation of being allocated in the future.		moment while it writes out dirty data. To keep that from happening as often, the background writer process scans forward looking for blocks that might be allocated in the near future that are dirty and that have a low usage count (alternatively called the Least Recently Used or LRUblocks). When it finds them, it writes some of them out pre- emptively, based on historical allocation rates.
Max written:	Indicates the maximum number of dirty buffers that can be written into the buffer cache during the last measurement period.	Number	If this measure indicates a high value it indicates that adequate buffers are not free in the cache. To optimize the value of this measure, you can increase the value of the *bgwriter_ lru_ maxpages* parameter.
Buffers freed by connections:	Indicates the number of buffers that were released from the cache for re-use during the last measurement period, when users wrote data directly to the disk.	Number	A high value is desired for this measure, as it reduces the need for an I/O- intensive operation such as ‘checkpointing’.
Buffers allocated:	Indicates the total number of calls to allocate a new buffer for a page (whether or not it was already cached) during the last measurement period.	Number	

3.3.2 PostgreSQL Databases Test

For each database on the PostgreSQL server, this test reports the transaction load on the database and reveals how well the database processes the transaction requests to it and how well it utilizes its cache. Overload conditions and processing bottlenecks are thus revealed.

Target of the test : PostgreSQL server

Agent deploying the test: An internal/remote agent

Outputs of the test : One set of results for every database on the target PostgreSQL server

Configurable parameters for the test

1. **TEST PERIOD** – How often should the test be executed.
2. **HOST** – The IP address of the server.
3. **PORT** – The port on which the server is listening. The default port is 5432.
4. **USER** – In order to monitor a PostgreSQL server, you need to manually create a special database user account in every PostgreSQL database instance that requires monitoring. When doing so, ensure that this user is vested with the *superuser* privileges. The sample script we recommend for user creation for eG monitoring is:

   CREATE ROLE eguser LOGIN
   ENCRYPTED PASSWORD {'eguser password'}
   SUPERUSER NOINHERIT NOCREATEDB NOCREATEROLE;

   The name of this user has to be specified in the **USERNAME** text box.
5. **PASSWORD** - The password associated with the above user name (can be 'NULL'). Here, 'NULL' means that the user does not have any password.
6. **CONFIRM PASSWORD** – Confirm the **PASSWORD** (if any) by retyping it here.
7. **DBNAME** - The name of the database to connect to. The default is "postgres".
8. **EXCLUDE DB** - Specify a comma-separated list of databases that need to be excluded from monitoring. By default, this is set to *rdsadmin*.

   Note:

   If you are monitoring a PostgreSQL server on the AWS EC2 cloud, then **make sure that you do not remove 'rdsadmin' from the EXCLUDE DB list**.

9. **SSL -** The name of this user has to be specified in the **USERNAME** text box.
10. To make diagnosis more efficient and accurate, the eG Enterprise suite embeds an optional detailed diagnostic capability. With this capability, the eG agents can be configured to run detailed, more elaborate tests as and when specific problems are detected. To enable the detailed diagnosis capability of this test for a particular server, choose the **On** option. To disable the capability, click on the **Off** option.

    The option to selectively enable/disable the detailed diagnosis capability will be available only if the following conditions are fulfilled:

    - The eG manager license should allow the detailed diagnosis capability
    - Both the normal and abnormal frequencies configured for the detailed diagnosis measures should not be 0.

Measurements made by the test

Measurement	Description	Measurement Unit	Interpretation
Database size:	Indicates the current size of this database.	KB	
Cache hit ratio:	Indicates the percentage of requests to this database that were serviced by the cache, without having to read from disk.	Percent	Because reading from the cache is less expensive than reading from disk, you want the ratio to be high. The higher this value is, the better. Generally, you can increase the cache hit ratio by increasing the amount of memory available to the database server. The detailed diagnosis of this measure provides you with the complete details of the database such as the number of server processes running on it, the number of transactions committed and rolled back, and the number of rows inserted, updated, and deleted.

Measurement	Description	Measurement Unit	Interpretation
Commit ratio:	Indicates the rate at which live rows are fetched while this index is scanned.	Percent	
Server process:	Indicates the number of processes that are currently running on this database.	Number	
Inserts:	Indicates the rate at which the records are inserted into this database.	Inserts/Sec	
Deletes:	Indicates the rate at which the records are deleted from this database.	Deletes/Sec	
Updates:	Indicates the rate at which records are updated into this database.	Updates/Sec	
Commits:	Indicates the transaction throughput.	Commits/Sec	A decrease in this measure during the monitoring period may indicate that the applications are not doing frequent commits. This may lead to problems with logging and data concurrency. The cause has to be probed in the application.
Rollbacks:	Indicates the rate at which rollbacks occurred on this database.	Rollbacks/Sec	A high rollback rate is an indicator of bad performance, since work performed up to the rollback point is wasted. The cause of the rollbacks has to be probed in the

Measurement	Description	Measurement Unit	Interpretation
			application.
Rows fetched:	Indicates the rate at which the rows that were read from this database based on a user query are stored in the buffer.	Fetches/Sec	
Rows returned:	Indicates the rate at which the rows are fetched from the buffer and sent to the client application.	Returns/Sec	If the size of the rows that are fetched from the buffer is too large, then the rows are fragmented and transferred to the client which is time consuming. This may in turn affect the performance of the database to some extent.
Blocks read:	Indicates the rate at which the blocks are read from this database.	Fetches/Sec	
Block hits:	Indicates the rate at which the blocks are fetched after a read is performed in this database.	Hits/Sec	

3.3.3 PostgreSQL Indexes Test

An index is a data structure that a database uses to reduce the amount of time it takes to perform certain operations. An index can also be used to ensure that duplicate values don't appear where they are not needed.

This test monitors the indexes on the PostgreSQL server and helps administrators quickly and accurately assess the effectiveness of these indexes.

Target of the test : PostgreSQL server

Agent deploying the test: An internal/remote agent

Outputs of the test : One set of results for every index for every table in each database that is configured for monitoring on the target PostgreSQL server

Configurable parameters for the test

1. **TEST PERIOD** – How often should the test be executed.
2. **HOST** – The IP address of the server.
3. **PORT** – The port on which the server is listening. The default port is 5432.
4. **USER** – In order to monitor a PostgreSQL server, you need to manually create a special database user account in every PostgreSQL database instance that requires monitoring. When doing so, ensure that this user is vested with the *superuser* privileges. The sample script we recommend for user creation for eG monitoring is:

   CREATE ROLE eguser LOGIN
   ENCRYPTED PASSWORD {'eguser password'}
   SUPERUSER NOINHERIT NOCREATEDB NOCREATEROLE;

   The name of this user has to be specified in the **USERNAME** text box.
5. **PASSWORD** - The password associated with the above user name (can be 'NULL'). Here, 'NULL' means that the user does not have any password.
6. **CONFIRM PASSWORD** – Confirm the **PASSWORD** (if any) by retyping it here.
7. **DBNAME** - The name of the database to connect to. The default is "postgres".
8. **INCLUDE DB** - Specify a comma-separated list of databases that you wish to monitor.
9. **EXCLUDE DB** - Specify a comma-separated list of databases that need to be excluded from monitoring. By default, this is set to *rdsadmin*.

   Note:

   If you are monitoring a PostgreSQL server on the AWS EC2 cloud, then **make sure that you do not remove 'rdsadmin' from the EXCLUDE DB list**.
10. **SSL** - The name of this user has to be specified in the **USERNAME** text box.

Measurements made by the test

Measurement	Description	Measurement Unit	Interpretation
Index scans:	Indicates the rate at which the index scans are initiated on this index in this database .	Scans/Sec	
Rows read:	Indicates the rate at which the index entries (rows) are read during the index scans on this index.	Reads/Sec	
Rows fetched:	Indicates the rate at which the rows are fetched from this index upon execution of a query.	Fetches/Sec	If the value of this measure is greater than the value of the Rows read measure, it indicates a possibility of index fragmentation or that the executed query is inefficient.

3.3.4 PostgreSQL Unused Indexes Test

While at one end indexes greatly enhance database performance, at the other they also add significant overhead to table change operations. Useless/unused indices can therefore be unnecessary resource hogs. Such indexes are typically not used by any regular query and may not enforce a constraint. However, these unneeded indexes cost you in several ways: they slow updates, inserts and deletes; they may keep HOT from updating the row in-place, requiring more VACUUMs; they take time to VACUUM; they add to query planning time; they take time to backup and restore. Administrators hence need to identify such indexes and eliminate them. The **PostgreSQL Unused Indexes** test helps administrators achieve the same. This test reports the number and names of unused/useless indexes, and thus prompts administrators to remove them so as to save the server from unnecessary performance degradations.

Target of the test : PostgreSQL server

Agent deploying the test: An internal/remote agent

Outputs of the test : One set of results for the target PostgreSQL server

Configurable parameters for the test

1. **TEST PERIOD** – How often should the test be executed.
2. **HOST** – The IP address of the server.
3. **PORT** – The port on which the server is listening. The default port is 5432.
4. **USER** – In order to monitor a PostgreSQL server, you need to manually create a special database user account in every PostgreSQL database instance that requires monitoring. When doing so, ensure that this user is vested with the *superuser* privileges. The sample script we recommend for user creation for eG monitoring is:

   CREATE ROLE eguser LOGIN
   ENCRYPTED PASSWORD {'eguser password'}
   SUPERUSER NOINHERIT NOCREATEDB NOCREATEROLE;

   The name of this user has to be specified in the **USERNAME** text box.
5. **PASSWORD** - The password associated with the above user name (can be 'NULL'). Here, 'NULL' means that the user does not have any password.
6. **CONFIRM PASSWORD** – Confirm the **PASSWORD** (if any) by retyping it here.
7. **DBNAME** - The name of the database to connect to. The default is "postgres".
8. **INCLUDE DB** - Specify a comma-separated list of databases that you wish to monitor.
9. **EXCLUDE DB** - Specify a comma-separated list of databases that need to be excluded from monitoring. By default, this is set to *rdsadmin*.

   Note:

   If you are monitoring a PostgreSQL server on the AWS EC2 cloud, then **make sure that you do not remove 'rdsadmin' from the EXCLUDE DB list**.
10. **SSL** - The name of this user has to be specified in the **USERNAME** text box.
11. To make diagnosis more efficient and accurate, the eG Enterprise suite embeds an optional detailed diagnostic capability. With this capability, the eG agents can be configured to run detailed, more elaborate tests as and when specific problems are detected. To enable the detailed diagnosis capability of this test for a particular server, choose the **On** option. To disable the capability, click on the **Off** option.

    The option to selectively enable/disable the detailed diagnosis capability will be available only if the following conditions are fulfilled:

- The eG manager license should allow the detailed diagnosis capability
- Both the normal and abnormal frequencies configured for the detailed diagnosis measures should not be 0.

Measurements made by the test

Measurement	Description	Measurement Unit	Interpretation
Number of indexes:	Indicates the number of indexes that are currently unused/useless on the server.	Number	A high value of this measure is a cause for concern. Use the detailed diagnosis of this measure to identify the unused indexes and take measures to get rid of them.

3.3.5 PostgreSQL Tables Test

The real test of the performance of a database server lies in how quickly the database responds to queries. Whenever users complaint of slow execution of their queries, administrators need to know the reason for the delay - is it because the queries themselves are badly designed? or is it due to how the database server performs table scans and returns the requested result set to the queries? The **PostgreSQL Tables** test helps with this root-cause analysis.

This test auto-discovers the tables in the configured databases and reports the number of times every table was scanned, the type of scanning (sequential or index) that was performed, and the rate at which the server reads data (via index and sequential scans) from each table. On the basis of this data, the test also indicates if any table is experiencing any query processing bottlenecks, and if so, how severe is the problem. In addition, the test also reveals how quickly critical database operations such as inserts, deletes, and updates, are performed on every table. Using this information, administrator can figure out whether/not the number and nature of scans performed on the tables are causing queries to the corresponding database to slowdown.

Target of the test : PostgreSQL server

Agent deploying the test: An internal/remote agent

Outputs of the test : One set of results for each table on every database configured for monitoring on the target PostgreSQL server

Configurable parameters for the test

1. **TEST PERIOD** – How often should the test be executed.
2. **HOST** – The IP address of the server.
3. **PORT** – The port on which the server is listening. The default port is 5432.
4. **USER** – In order to monitor a PostgreSQL server, you need to manually create a special database user account in every PostgreSQL database instance that requires monitoring. When doing so, ensure that this user is vested with the *superuser* privileges. The sample script we recommend for user creation for eG monitoring is:

   CREATE ROLE eguser LOGIN
   ENCRYPTED PASSWORD {'eguser password'}
   SUPERUSER NOINHERIT NOCREATEDB NOCREATEROLE;

   The name of this user has to be specified in the **USERNAME** text box.
5. **PASSWORD** - The password associated with the above user name (can be 'NULL'). Here, 'NULL' means that the user does not have any password.
6. **CONFIRM PASSWORD** – Confirm the **PASSWORD** (if any) by retyping it here.
7. **DBNAME** - The name of the database to connect to. The default is "postgres".
8. **INCLUDE DB** - Specify a comma-separated list of databases that you wish to monitor.
9. **EXCLUDE DB** - Specify a comma-separated list of databases that need to be excluded from monitoring. By default, this is set to *rdsadmin*.

   Note:

   If you are monitoring a PostgreSQL server on the AWS EC2 cloud, then **make sure that you do not remove 'rdsadmin' from the EXCLUDE DB list**.
10. **SSL** - The name of this user has to be specified in the **USERNAME** text box.

Measurements made by the test

Measurement	Descrption	Measurement Unit	Interpretation
Sequence scans count:	Indicates the number of sequential scans initiated on this table	Number	Sequential or Full table scan is a scan made on the database where each

Measurement	Descrption	Measurement Unit	Interpretation
	during the last measurement period.		row of the table under scan is read in a sequential (serial) order and the columns encountered are checked for the validity of a condition. Full table scans are usually the slowest method of scanning a table due to the heavy amount of I/O reads and writes required from the disk which consists of multiple seeks as well as costly disk to memory transfers. Typically therefore, a low value is desired for this measure. However, if a query returns more than approximately 5-10% of all rows in the table, then PostgreSQL prefers the sequential scan over the index scan. This is because an index scan requires several I/O operations for each row (look up the row in the index, then retrieve the row from the heap). Whereas a sequential scan only requires a single I/O for each row - or even less because a block (page) on the disk contains more than one row, so more than one row can be fetched with a single I/O operation.
Sequence reads row count:	Indicates the number of rows that are processed through sequential scan from this table during the last measurement period.	Number	
Average reads per scan:	Indicates the rate at which rows from this	Fetches/Sec	A high value is desired for this measure. If the value is low or falls

Measurement	Descrption	Measurement Unit	Interpretation
	table were processed through a sequential scan.		consistently, it indicates bottlenecks while performing sequential scans on the table.
Index scans:	Indicates the number of index scans initiated over all the indexes belonging to this table during the last measurement period.	Number	An index scan occurs when the database manager accesses an index for any of the following reasons: • To narrow the set of qualifying rows (by scanning the rows in a certain range of the index) before accessing the base table. • To order the output. • To retrieve the requested column data directly. If all of the requested data is in the index, the indexed table does not need to be accessed. This is known as an index-only access. Typically, a high value of this measure is desired, as index scans are I/O-friendly operations. However, if a query returns more than approximately 5-10% of all rows in the table, then PostgreSQL prefers the sequential scan over the index scan. This is because an index scan requires several I/O operations for each row (look up the row in the index, then retrieve the row from the heap). Whereas a sequential scan only requires a single I/O for each row - or even less because a block (page) on the disk contains more than one row, so more than one row can be

Measurement	Descrption	Measurement Unit	Interpretation
			fetched with a single I/O operation.
Average fetch per index:	Indicates the rate at which the rows are processed through an index scan on this table.	Fetches/Sec	A high value is desired for this measure. If the value is low or falls consistently, it indicates bottlenecks while performing index scans on the table.
Table scans:	Indicates the number of times this table was scanned during the last measurement period.	Number	A high value indicates that there are no proper indexes for this table. This may cause delays in query execution.
Inserts:	Indicates the rate at which the rows are inserted into this table.	Inserts/Sec	
Deletes:	Indicates the rate at which the rows are deleted from this table.	Deletes/Sec	
Updates:	Indicates the rate at which the rows are updated in this table.	Updates/Sec	
Priority:	Indicates the type of problem that is currently experienced by this table while processing a query.		The difference between the Sequence scan count and the Index scan count measures determines the Priority of the problem experienced by a table. The various Priorities this measure reports and their numeric equivalents as shown in the table:

Numeric Value	State
1	Minor Problem
2	Major

<table>
<tr><th>Measurement</th><th>Descrption</th><th>Measurement Unit</th><th>Interpretation</th></tr>
<tr><td></td><td></td><td></td><td><table><tr><th>Numeric Value</th><th>State</th></tr><tr><td></td><td>Problem</td></tr><tr><td>3</td><td>Extreme Problem</td></tr></table><p><b>Note:</b></p><p>By default, this measure reports the above- mentioned <b>States</b> while indicating the type of problem that is experienced while querying this database. However, the graph of this measure will be represented using the corresponding numeric equivalents of the states as mentioned in the table above.</p><p>If the severity of this measure is high, it indicates that the query used may be inefficient or there may be a problem with the indexing of the column or there may be a possibility of fragmentation of the table or index of this database.</p></td></tr>
</table>

3.4 The PostGreSQL Service Layer

Besides revealing the availability and responsiveness of the database server, the tests mapped to this layer also sheds light on the idle and waiting user connections on the server, the level of locking activity on the server, and the number and details of queries to the server that have been running for an unreasonably long time.

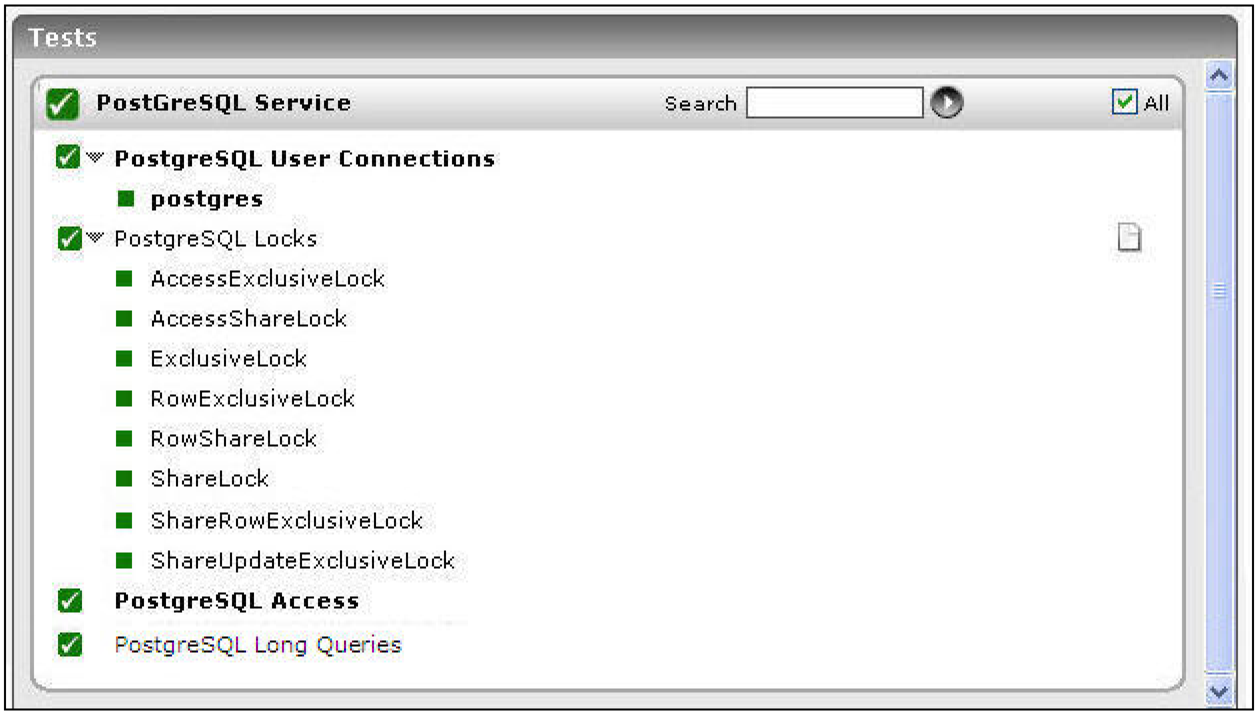


Figure 3.5: The tests mapped to the PostgreSQL Service layer

3.4.1 PostgreSQL User Connections Test

This test monitors the users who are currently connected to the server and reports the number and state of each user connection. Using the metrics reported by this test, administrators can promptly isolate idle and waiting connections, which are a drain on a server's resources.

Target of the test : PostgreSQL server

Agent deploying the test: An internal/remote agent

Outputs of the test : One set of results for each user currently connected to the target PostgreSQL server

Configurable parameters for the test

1. **TEST PERIOD** – How often should the test be executed.
2. **HOST** – The IP address of the server.
3. **PORT** – The port on which the server is listening. The default port is 5432.
4. **USER** – In order to monitor a PostgreSQL server, you need to manually create a special database user account in every PostgreSQL database instance that requires monitoring. When doing so, ensure that this user is vested with the *superuser* privileges. The sample script

we recommend for user creation for eG monitoring is:

CREATE ROLE eguser LOGIN
ENCRYPTED PASSWORD {'eguser password'}
SUPERUSER NOINHERIT NOCREATEDB NOCREATEROLE;

The name of this user has to be specified in the **USERNAME** text box.

5. **PASSWORD -** The password associated with the above user name (can be 'NULL'). Here, 'NULL' means that the user does not have any password.
6. **CONFIRM PASSWORD –** Confirm the **PASSWORD** (if any) by retyping it here.
7. **DBNAME -** The name of the database to connect to. The default is "postgres".
8. **SSL -** The name of this user has to be specified in the **USERNAME** text box.
9. To make diagnosis more efficient and accurate, the eG Enterprise suite embeds an optional detailed diagnostic capability. With this capability, the eG agents can be configured to run detailed, more elaborate tests as and when specific problems are detected. To enable the detailed diagnosis capability of this test for a particular server, choose the **On** option. To disable the capability, click on the **Off** option.

   The option to selectively enable/disable the detailed diagnosis capability will be available only if the following conditions are fulfilled:

   - The eG manager license should allow the detailed diagnosis capability
   - Both the normal and abnormal frequencies configured for the detailed diagnosis measures should not be 0.

Measurements made by the test

Measurement	Description	Measurement Unit	Interpretation
Total connections:	Indicates the total number of connections that are currently established by this user on the server.	Number	
Idle connections:	Indicates the number of connections of this user that are currently idle on the server.	Number	Ideally, the value of this measure should be low. A high value is indicative of a large number of idle connections, which in turn causes

Measurement	Description	Measurement Unit	Interpretation
			unnecessary consumption of critical server resources. Idle connections also unnecessarily lock new connections from the connection pool, thereby denying other users access to the server for performing important tasks. Use the detailed diagnosis of this measure to view the details of the idle connections.
Active connections:	Indicates the number of connections of this user that are currently active.	Number	Use the detailed diagnosis of this measure to view the details of the active connections.
Waiting connections:	Indicates the number of connections of this user that are currently waiting for a resource/database object/ lock to be released.	Number	The value of this measure should be kept at a minimum, as waiting connections also cause a resource drain. Use the detailed diagnosis of this measure to view the details of the waiting connections.

3.4.2 PostgreSQL Locks Test

PostgreSQL provides various lock modes to control concurrent access to data in tables. These modes can be used for application-controlled locking in situations where MVCC does not give the desired behavior. Also, most PostgreSQL commands automatically acquire locks of appropriate modes to ensure that referenced tables are not dropped or modified in incompatible ways while the command executes. The common lock modes are as follows:

- **ACCESS SHARE**

  Conflicts with the ACCESS EXCLUSIVE lock mode only.

  The SELECT command acquires a lock of this mode on referenced tables. In general, any query that only reads a table and does not modify it will acquire this lock mode.

- **ROW SHARE**

  Conflicts with the EXCLUSIVE and ACCESS EXCLUSIVE lock modes.

  The SELECT FOR UPDATE and SELECT FOR SHARE commands acquire a lock of this mode on the target table(s) (in addition to ACCESS SHARE locks on any other tables that are referenced but not selected FOR UPDATE/FOR SHARE).

- **ROW EXCLUSIVE**

  Conflicts with the SHARE , SHARE ROW EXCLUSIVE , EXCLUSIVE , and ACCESS EXCLUSIVE lock modes.

  The commands UPDATE, DELETE, and INSERT acquire this lock mode on the target table (in addition to ACCESS SHARE locks on any other referenced tables). In general, this lock mode will be acquired by any command that modifies the data in a table.

- **SHARE UPDATE EXCLUSIVE**

  Conflicts with the SHARE UPDATE EXCLUSIVE , SHARE , SHARE ROW EXCLUSIVE , EXCLUSIVE , and ACCESS EXCLUSIVE lock modes. This mode protects a table against concurrent schema changes and VACUUM runs.

  Acquired by VACUUM (without FULL), ANALYZE, and CREATE INDEX CONCURRENTLY.

- **SHARE**

  Conflicts with the ROW EXCLUSIVE , SHARE UPDATE EXCLUSIVE , SHARE ROW EXCLUSIVE , EXCLUSIVE , and ACCESS EXCLUSIVE lock modes. This mode protects a table against concurrent data changes.

  Acquired by CREATE INDEX (without CONCURRENTLY).

- **SHARE ROW EXCLUSIVE**

  Conflicts with the ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE, and ACCESS EXCLUSIVE lock modes.

  This lock mode is not automatically acquired by any PostgreSQL command.

- **EXCLUSIVE**

  Conflicts with the ROW SHARE , ROW EXCLUSIVE , SHARE UPDATE EXCLUSIVE , SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE, and ACCESS EXCLUSIVE lock modes.

This mode allows only concurrent ACCESS SHARE locks, i.e., only reads from the table can proceed in parallel with a transaction holding this lock mode.

This lock mode is not automatically acquired on user tables by any PostgreSQL command. However it is acquired on certain system catalogs in some operations.

- **ACCESS EXCLUSIVE**

  Conflicts with locks of all modes (ACCESS SHARE, ROW SHARE, ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE, and ACCESS EXCLUSIVE). This mode guarantees that the holder is the only transaction accessing the table in any way.

  Acquired by the ALTER TABLE, DROP TABLE, TRUNCATE, REINDEX, CLUSTER, and VACUUM FULL commands. This is also the default lock mode for LOCK TABLE statements that do not specify a mode explicitly.

The locking activity of a database server must be monitored carefully because an application holding a specific lock for a long time could cause a number of other transactions relying on the same lock to fail. The **PostgreSQL Locks** test does just that. For every lock mode that is currently active on the database server, this test reports the total number of locks that are in that mode.

Target of the test : PostgreSQL server

Agent deploying the test: An internal/remote agent

Outputs of the test : One set of results for each lock mode currently held on the target PostgreSQL server

Configurable parameters for the test

1. **TEST PERIOD** – How often should the test be executed.
2. **HOST** – The IP address of the server.
3. **PORT** – The port on which the server is listening. The default port is 5432.
4. **USER** – In order to monitor a PostgreSQL server, you need to manually create a special database user account in every PostgreSQL database instance that requires monitoring. When doing so, ensure that this user is vested with the *superuser* privileges. The sample script we recommend for user creation for eG monitoring is:

   CREATE ROLE eguser LOGIN
   ENCRYPTED PASSWORD {'eguser password'}
   SUPERUSER NOINHERIT NOCREATEDB NOCREATEROLE;

The name of this user has to be specified in the **USERNAME** text box.

5. **PASSWORD -** The password associated with the above user name (can be 'NULL'). Here, 'NULL' means that the user does not have any password.
6. **CONFIRM PASSWORD** – Confirm the **PASSWORD** (if any) by retyping it here.
7. **DBNAME -** The name of the database to connect to. The default is "postgres".
8. **SSL -** The name of this user has to be specified in the **USERNAME** text box.
9. To make diagnosis more efficient and accurate, the eG Enterprise suite embeds an optional detailed diagnostic capability. With this capability, the eG agents can be configured to run detailed, more elaborate tests as and when specific problems are detected. To enable the detailed diagnosis capability of this test for a particular server, choose the **On** option. To disable the capability, click on the **Off** option.

   The option to selectively enable/disable the detailed diagnosis capability will be available only if the following conditions are fulfilled:

   - The eG manager license should allow the detailed diagnosis capability
   - Both the normal and abnormal frequencies configured for the detailed diagnosis measures should not be 0.

Measurements made by the test

Measurement	Description	Measurement Unit	Interpretation
Number of locks:	Indicates the total number of locks that are currently held on the database server.	Number	A high value may indicate one of the following: • Too many transactions happening • Locked resources not being released properly • Locks are being held unnecessarily. With the help of the detailed diagnosis of this measure, you can determine the query that is waiting

Measurement	Description	Measurement Unit	Interpretation
			for the lock, the user who executed that query, the query that is blocking, and the user who is executing the blocking query. Once the blocked and blocking queries are isolated, you can then proceed to do what's required to release unnecessary locks.

3.4.3 PostgreSQL Access Test

This test emulates a client executing a configured query on the database server, and in the process reports whether the server is available, and if so, how quickly it responds to the client queries. The unavailability of a network connection to the server and bottlenecks to responsiveness can thus be promptly isolated.

Target of the test : PostgreSQL server

Agent deploying the test: An external agent; if you are running this test using the external agent on the eG manager box, then make sure that this external agent is able to communicate with the port on which the target PostgreSQL server is listening. Alternatively, you can deploy the external agent that will be running this test on a host that can access the port on which the target PostgreSQL server is listening.

Outputs of the test : One set of results for the target PostgreSQL server.

Configurable parameters for the test

1. **TEST PERIOD** – How often should the test be executed.
2. **HOST** – The IP address of the server.
3. **PORT** – The port on which the server is listening. The default port is 5432.
4. **USER** – In order to monitor a PostgreSQL server, you need to manually create a special database user account in every PostgreSQL database instance that requires monitoring. When doing so, ensure that this user is vested with the *superuser* privileges. The sample script we recommend for user creation for eG monitoring is:

   CREATE ROLE eguser LOGIN

ENCRYPTED PASSWORD {'eguser password'}
SUPERUSER NOINHERIT NOCREATEDB NOCREATEROLE;

The name of this user has to be specified in the **USERNAME** text box.

5. **PASSWORD** - The password associated with the above user name (can be 'NULL'). Here, 'NULL' means that the user does not have any password.
6. **CONFIRM PASSWORD** – Confirm the **PASSWORD** (if any) by retyping it here.
7. **DBNAME** - The name of the database to connect to. The default is "postgres".
8. **INCLUDE DB** - Specify a comma-separated list of databases that you wish to monitor.
9. **QUERY** - Specify the select query to execute. The default is "select * from pg_tables". Every *DATABASE* being monitored, should have a corresponding **QUERY** specification
10. **SSL** - The name of this user has to be specified in the **USERNAME** text box.

Measurements made by the test

Measurement	Description	Measurement Unit	Interpretation
Availability:	Indicates whether the database server is currently available or not.	Percent	The availability is 100% when the server is responding to a request and 0% when it is not. Availability problems may be caused by a misconfiguration/malfunctioning of the database server, or because the server has not been started.
Response time:	Indicates the time taken by this database to respond to a user query during the last measurement period.	Secs	A sudden increase in response time is indicative of a performance bottleneck at the database server.

3.4.4 PostgreSQL Long Queries Test

This test reports the number of queries that are executing for each database. Using this test, you can identify the query that takes too long to execute and thus the resource-intensive queries to a database can be isolated quickly.

Target of the test : PostgreSQL server

Agent deploying the test: An internal/remote agent

Outputs of the test : One set of results for the target PostgreSQL server

Configurable parameters for the test

1. **TEST PERIOD –** How often should the test be executed.
2. **HOST –** The IP address of the server.
3. **PORT –** The port on which the server is listening. The default port is 5432.
4. **USER** – In order to monitor a PostgreSQL server, you need to manually create a special database user account in every PostgreSQL database instance that requires monitoring. When doing so, ensure that this user is vested with the *superuser* privileges. The sample script we recommend for user creation for eG monitoring is:

   CREATE ROLE eguser LOGIN
   ENCRYPTED PASSWORD {'eguser password'}
   SUPERUSER NOINHERIT NOCREATEDB NOCREATEROLE;

   The name of this user has to be specified in the **USERNAME** text box.
5. **PASSWORD -** The password associated with the above user name (can be 'NULL'). Here, 'NULL' means that the user does not have any password.
6. **CONFIRM PASSWORD –** Confirm the **PASSWORD** (if any) by retyping it here.
7. **DBNAME** - The name of the database to connect to. The default is "postgres".
8. **EXCLUDE DB** - Specify a comma-separated list of databases that need to be excluded from monitoring. By default, this is set to *rdsadmin*.

   Note:

   If you are monitoring a PostgreSQL server on the AWS EC2 cloud, then **make sure that you do not remove 'rdsadmin' from the EXCLUDE DB list**.
9. **ELAPSED TIME** - Specify the duration (in seconds) for which a query should have executed for it to be regarded as a long running query. The default value is 10.
10. **SSL -** The name of this user has to be specified in the **USERNAME** text box.
11. To make diagnosis more efficient and accurate, the eG Enterprise suite embeds an optional detailed diagnostic capability. With this capability, the eG agents can be configured to run detailed, more elaborate tests as and when specific problems are detected. To enable the detailed diagnosis capability

of this test for a particular server, choose the **On** option. To disable the capability, click on the **Off** option.

The option to selectively enable/disable the detailed diagnosis capability will be available only if the following conditions are fulfilled:

- The eG manager license should allow the detailed diagnosis capability
- Both the normal and abnormal frequencies configured for the detailed diagnosis measures should not be 0.

Measurements made by the test

Measurement	Description	Measurement Unit	Interpretation
Number of queries:	Indicates the number of queries currently executing on the database server that have been running for more time than the configured **ELAPSED TIME**.	Number	The detailed diagnosis for this measure indicates the exact queries and which user is executing the queries. This information can be very useful in identifying queries that may be candidates for optimization.

About eG Innovations

eG Innovations provides intelligent performance management solutions that automate and dramatically accelerate the discovery, diagnosis, and resolution of IT performance issues in on-premises, cloud and hybrid environments. Where traditional monitoring tools often fail to provide insight into the performance drivers of business services and user experience, eG Innovations provides total performance visibility across every layer and every tier of the IT infrastructure that supports the business service chain. From desktops to applications, from servers to network and storage, from virtualization to cloud, eG Innovations helps companies proactively discover, instantly diagnose, and rapidly resolve even the most challenging performance and user experience issues.

eG Innovations is dedicated to helping businesses across the globe transform IT service delivery into a competitive advantage and a center for productivity, growth and profit. Many of the world's largest businesses use eG Enterprise to enhance IT service performance, increase operational efficiency, ensure IT effectiveness and deliver on the ROI promise of transformational IT investments across physical, virtual and cloud environments.

To learn more visit www.eginnovations.com.

Contact Us

For support queries, email support@eginnovations.com.

To contact eG Innovations sales team, email sales@eginnovations.com.