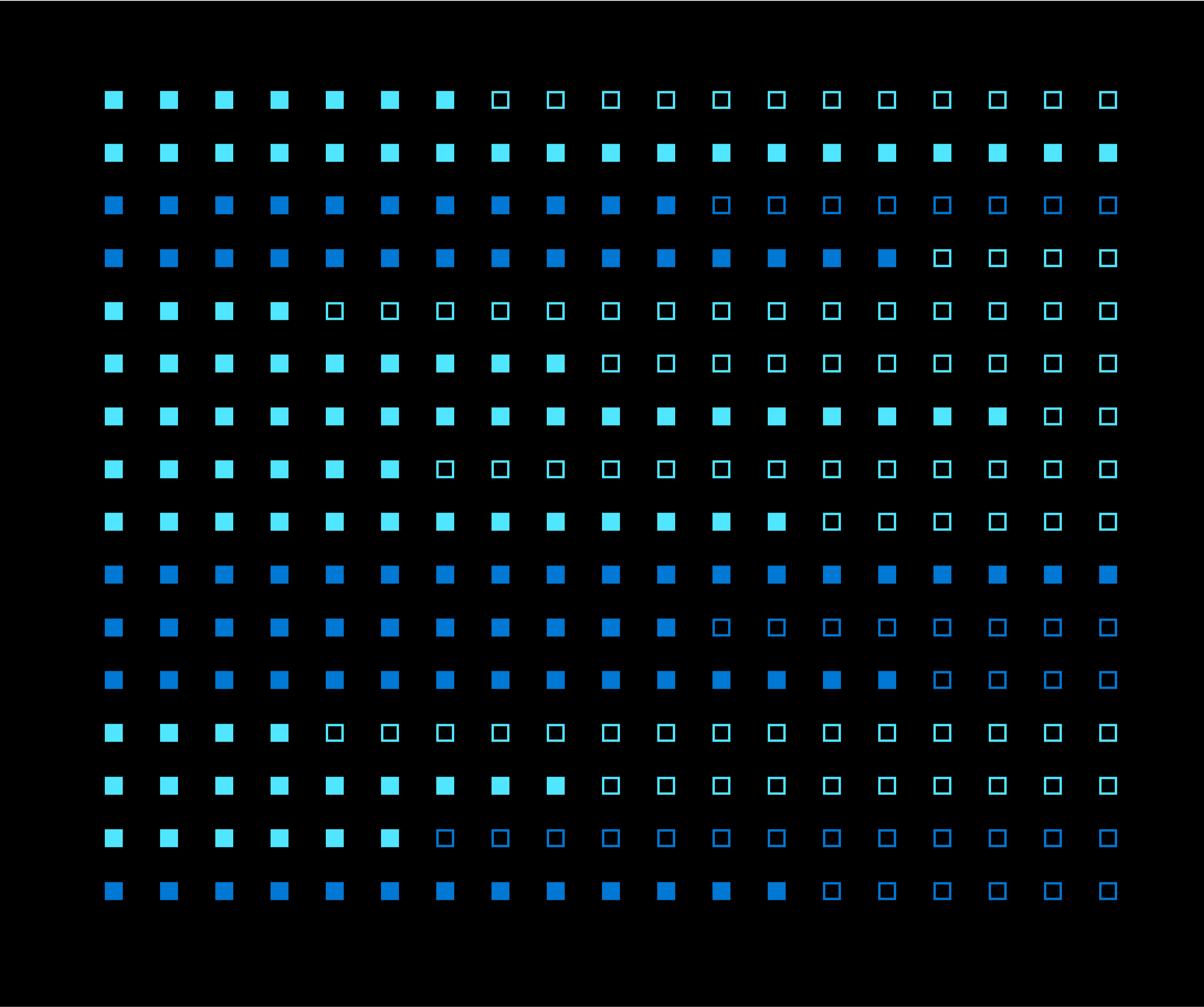
**Azure Database for PostgreSQL Performance Tuning Guide (Microsoft Internal Only)**

**Technical white paper**

**Published:** July 2021

**Applies to:** Azure Database for PostgreSQL

**Copyright**

The information contained in this document represents the current view of Microsoft Corporation on the issues discussed as of the date of publication. This content may include product or service’ currently in preview and as such, we cannot guarantee that all details included herein will be exactly as what is found in the shipping product. Because Microsoft must respond to changing market conditions, it should not be interpreted to be a commitment on the part of Microsoft, and Microsoft cannot guarantee the accuracy of any information presented after the date of publication. The information represents the product or service at the time this document was shared and should be used for planning purposes only.

This white paper is for informational purposes only. MICROSOFT MAKES NO WARRANTIES, EXPRESS, IMPLIED, OR STATUTORY, AS TO THE INFORMATION IN THIS DOCUMENT.

Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in, or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Microsoft Corporation.

Microsoft may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any written license agreement from Microsoft, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property. Information subject to change at any time without prior notice.

Microsoft, Active Directory, Azure, Bing, Excel, Power BI, SharePoint, Silverlight, SQL Server, Visual Studio, Windows, and Windows Server are trademarks of the Microsoft group of companies.

All other trademarks are property of their respective owners.

© 2021 Microsoft Corporation. All rights reserved.

Contents

[Introduction 6](#_Toc75967733)

[About PostgreSQL and Azure Database for PostgreSQL 7](#_Toc75967734)

[PostgreSQL Capabilities for Optimal Performance 9](#_Toc75967735)

[Choosing the right number of resources and tier 9](#_Toc75967736)

[Document contents 12](#_Toc75967737)

[Extensions, Parameter and What to Enable 13](#_Toc75967738)

[Extensions 13](#_Toc75967739)

[Parameters 13](#_Toc75967740)

[PostgreSQL Logging, Monitoring and Tuning techniques 15](#_Toc75967741)

[Using the PostgreSQL Slow Query log 19](#_Toc75967742)

[Statistics 21](#_Toc75967743)

[Query Performance Insight 23](#_Toc75967744)

[Performance Recommendations 25](#_Toc75967745)

[Log Analytics 26](#_Toc75967746)

[The Statistics Collector 31](#_Toc75967747)

[Query Store and when to use this vs enabling pg\_stat\_statements 39](#_Toc75967748)

[The Automatic Maintenance Processes 42](#_Toc75967749)

[Autovacuum 42](#_Toc75967750)

[Writes (checkpoints, bgwriter) 45](#_Toc75967751)

[Troubleshooting Locks, Blocks, Deadlock, waits and user activity 47](#_Toc75967752)

[Locks and blocking operations 47](#_Toc75967753)

[Connections 54](#_Toc75967754)

[Important wait events 56](#_Toc75967755)

[EXPLAIN Analyze 58](#_Toc75967756)

[Cost estimation - sequential read 60](#_Toc75967757)

[Nodes and structure of the query plan 61](#_Toc75967758)

[GUC (Grand Unified Configuration) parameters 66](#_Toc75967759)

[Azure DB for PostgreSQL Best Practices for Optimal Performance 69](#_Toc75967760)

[Azure Advisor 69](#_Toc75967761)

[Connection pooling (PgBouncer) 71](#_Toc75967762)

[PgBouncer for Azure Database for PostgreSQL flexible server 73](#_Toc75967763)

[PgBouncer for Azure Database for PostgreSQL single server 74](#_Toc75967764)

[General recommendations for optimize performance - Single server 76](#_Toc75967765)

[Autovacuum 76](#_Toc75967766)

[Bulk inserts 78](#_Toc75967767)

[Query statistics collection 79](#_Toc75967768)

[Application Aspects Optimizations 80](#_Toc75967769)

[Physical Proximity (Colocation) 80](#_Toc75967770)

[Connection Pooling 80](#_Toc75967771)

[Accelerated Networking 81](#_Toc75967772)

[ExpressRoute 82](#_Toc75967773)

[TCP\_NODELAY socket option 84](#_Toc75967774)

[Persistent or Long-Lived Connection 84](#_Toc75967775)

[Tune your application 84](#_Toc75967776)

[Applications that have slow performance because of "chatty" behavior 84](#_Toc75967777)

[Applications that have sub-optimal queries 85](#_Toc75967778)

[Applications that have sub-optimal data access design 86](#_Toc75967779)

[FAQs for Web Apps in Azure 87](#_Toc75967780)

[Possible Issues and Solutions 88](#_Toc75967781)

[Table-level storage parameters 88](#_Toc75967782)

[Bgwriter tuning 91](#_Toc75967783)

[Partitioning 91](#_Toc75967784)

[Lack or too many indexes 92](#_Toc75967785)

[Choosing the right index type 92](#_Toc75967786)

[Database checklist and quick tips 96](#_Toc75967787)

[Daily cleaning 96](#_Toc75967788)

[Recovering disk space 96](#_Toc75967789)

[Update planner statistics 97](#_Toc75967790)

[Update visibility map 97](#_Toc75967791)

[Preventing transaction ID rollback failure 97](#_Toc75967792)

[Multiple transactions and rollbacks 99](#_Toc75967793)

[Automatic clean up background processes 99](#_Toc75967794)

[Periodically rebuilding indexes 100](#_Toc75967795)

**© 2021 Microsoft Corporation. All rights reserved.**



# Introduction

This PostgreSQL performance tuning guide covers and demonstrate all relevant aspects with PostgreSQL, Azure DB Single Server, or Flexible Server deployments.

Many times, customers migrate to non-commercial databases like PostgreSQL or MySQL from commercial databases. However, it is important to prepare and take into consideration that PostgreSQL would behave differently if you migrated from another database engine (regardless to what was the source database engine).

This may not be the case for you. You may just be starting something new and are thinking about using PostgreSQL as your database for your applications or services.

Most of the RDBMSs use different ways to calculate and plan the queries and commands before executing those. The syntax is different, but many rules are the same for example, you should not have too many or missing indexes. Planning the schema properly and tables will highly reflect the performance of your databases, and there are different parameters that can tune the database behavior to adjust it to your needs.

Microsoft Azure has created a fully managed service for you to be able to avoid focusing a lot of time and money in your databases layer and this is the purpose of the Azure DB for PostgreSQL service (will be covered widely in this book).

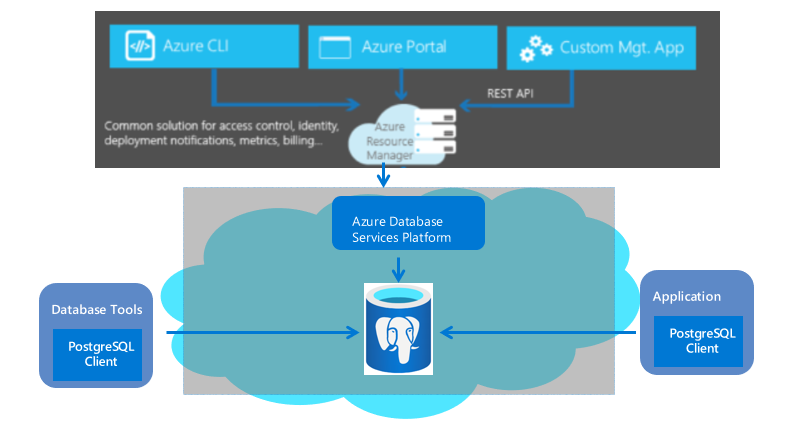
This guide was written to allow more and more users to be able to confidently use Azure Database for PostgreSQL even if they have no or little experience with PostgreSQL databases or using the Azure Database managed service as their data store. We are sure you will find this guide informative and helpful for your running stack.

# About PostgreSQL and Azure Database for PostgreSQL

From Azure Database for PostgreSQL [starter page](https://azure.microsoft.com/en-us/services/postgresql/) or Azure’s [documentation](https://docs.microsoft.com/en-us/azure/postgresql/overview):

Azure Database for PostgreSQL is a relational database service in the Microsoft cloud based on the [PostgreSQL Community Edition](https://www.postgresql.org/) (available under the GPLv2 license) database engine. Azure Database for PostgreSQL delivers:

* Built-in high availability.
* Data protection using automatic backups and point-in-time-restore for up to 35 days.
* Automated maintenance for underlying hardware, operating system, and database engine to keep the service secure and up to date.
* Predictable performance, using inclusive pay-as-you-g o pricing.
* Elastic scaling within seconds.
* Enterprise grade security and industry-leading compliance to protect sensitive data at-rest and in-motion.
* Monitoring and automation to simplify management and monitoring for large-scale deployments.
* Industry-leading support experience.



These capabilities require almost no administration, and all are provided at no additional cost. They allow you to focus on rapid application development and accelerating your time to market rather than allocating precious time and resources to manage virtual machines and infrastructure. In addition, you can continue to develop your applications with the open-source tools and platforms of your choice to deliver with the speed and efficiency your business demands, all without having to learn new skills.

This document will only cover the following two Azure Database for PostgreSQL deployments options:

1. [Single Server](https://docs.microsoft.com/en-us/azure/postgresql/overview-single-server) – Single Server is a fully managed database service with minimal requirements for customizations of the database. The single server platform is designed to handle most of the database management functions such as patching, backups, high availability, security with minimal user configuration and control. The architecture is optimized to provide 99.99% availability on single availability zone. It supports community version of PostgreSQL of [9.6, 10, and 11](https://docs.microsoft.com/en-us/azure/postgresql/concepts-supported-versions). The service is generally available today in wide variety of Azure regions.

Single servers are best suited for cloud native applications designed to handle automated patching without the need for granular control on the patching schedule and custom PostgreSQL configuration settings.

1. [Flexible Server (Preview)](https://docs.microsoft.com/en-us/azure/postgresql/flexible-server/overview) - Azure Database for PostgreSQL - Flexible Server is a fully managed database service designed to provide more granular control and flexibility over database management functions and configuration settings. In general, the service provides more flexibility and server configuration customizations based on the user requirements. The flexible server architecture allows users to co-locate database engine with the client-tier for lower latency, choose high availability within a single availability zone and across multiple availability zones. Flexible servers also provide better cost optimization controls with ability to stop/start your server and burstable compute tier that is ideal for workloads that do not need full compute capacity continuously. The service currently supports community version of [PostgreSQL 11, 12 and 13](https://docs.microsoft.com/en-us/azure/postgresql/flexible-server/concepts-supported-versions). The service is currently in preview, available today in wide variety of Azure regions.  
   Flexible servers are best suited for:

* Application developments requiring better control and customizations.
* Zone redundant high availability
* Managed maintenance windows

# Document contents

In this document we will be discussing the following topics related to PostgreSQL performance:

* **Extensions and Parameters** – Extensions are SQL objects packaged together to simplify database management and development; Initial Parameters will determine the behavior of the Azure instance hosting the Postgres databases.
* **Logging methods –** what to log and how to log messages as part of a troubleshooting and Performance Tuning exercise.
* **Database statistics –** what are statistics and how we can use them to troubleshoot performance issues.
* **Query Performance Insights** **–** how to use this Azure graphical view of query execution history and details, specifically viewing Long Running Queries and Wait Statistics information.
* **Azure Performance Recommendations –** how to use this Azure feature, recommending index modifications.
* **Log Analytics -** a tool in Azure that allows us to analyze logs from data collected by Azure Monitor Logs and interactively analyze their results to troubleshoot queries.
* **The Statistics Collector** – which is “a subsystem that supports collection and reporting of information about server activity”.
* **Query Store and when to use this vs enabling pg\_stat\_statements** – Query Store is an Azure feature that allows you to trace query performance over time. The pg\_stat\_statements is a PostgreSQL extension that can track execution statistics for all SQL statements executed by the server.
* **The automated and background processes and how to properly configure them –** specifically, the autovacuum and checkpoint processes.
* **Troubleshooting Locks, Blocks, Deadlock, waits and user activity** - how do locks, blocks and deadlocks occur, how to detect and resolve/reduce them.
* **Azure DB for PostgreSQL Best Practices for Optimal Performance** - list of main performance Best Practices.
* **Application Aspects Optimizations** - performance is not only about database and queries. This section lists the non-database application performance Best Practices.
* **Possible Issues and Solutions** - some examples of performance issues and how to resolve them.
* **Database checklist and quick tips** – a summary of all the main “low-hanging-fruits" for improved performance.

# The Automatic Maintenance Processes

This section discusses the automatic and background processes along with the parameters that can be used to improve their performance.

## Autovacuum

Internal data consistency in PostgreSQL is based on Multi-Version Concurrency Control (MVCC) mechanism, which allows database engine to maintain multiple versions of a row providing greater concurrency with only minimal locking between the different processes. PostgreSQL maintains both the previous values and latest values of each modified row in internal structures, including the row version.

For example, when a row is deleted, it is not removed physically, instead, it is marked as dead. Similarly for updates, it marks the existing row as “dead” and inserts a new row. These operations leave behind the dead records, called dead tuples, even after all the transactions that might see those versions finish. Unless cleaned up, these dead tuples will stay, consuming disk space, and for tables with a high update and delete rate, the dead tuples might account for most of the disk space, wasting space and slowing query performance.

To reclaim the space used by dead tuples, the VACUUM command is used. This command scans table and remove dead tuples both from table and indexes. It will not generally return the disk space back to the operating system, but it will make it usable for new rows.

In contrast, VACUUM FULL reclaims the space and returns it to the operating system by writing a completely new version, but it has several disadvantages. First, it acquires an exclusive lock on a table, blocks all operations, including selects. Second, it requires additional disk space for the new copy of a table during the process. Please note that the Autovacuum process will only run VACUUM and never VACUUM ALL.

While VACUUM command can be run manually, it may not run when it is needed, and table may get bloated to the point that VACUUM FULL is needed. Therefore, it is recommended to use Autovacuum, which performs VACUUMs often enough to keep the bloating under control and to avoid needing a manual VACUUM FULL.

Autovacuum is a background process enabled automatically when PostgreSQL instance starts, and it schedules vacuuming dynamically in response to data update activity. While in most cases the default configurations can be sufficient, sometimes it is necessary to tune this process.

**Autovacuum tuning**

The *autovacuum daemon* is enabled by default. In the default PostgreSQL configuration, the *autovacuum* daemon takes care of automatic analyzing of tables when they are first populated with data (for the recent version of Postgres), and as they change throughout regular operation. Click to learn how to [Optimize autovacuum on an Azure Database for PostgreSQL - Single Server](https://docs.microsoft.com/en-us/azure/postgresql/howto-optimize-autovacuum).

**Note:** *autovacuum* should **never** be disabled. In very rare cases and based on specific scenarios, if *autovacuum* is disabled, it is recommended to run ANALYZE periodically, or just after making significant changes in the contents of a table. More accurate statistics will help the planner to choose the best query plan, hence improve the speed of query processing. A common strategy is to run VACUUM and ANALYZE once a day during a low-usage time of day.

There is another background process called [*Stats Collector*](#_The_Statistics_Collector)(discussed previously) that tracks the usage and activity information. The information collected by this process is used by *autovacuum* launcher to identify the list of candidate tables for *autovacuum*. PostgreSQL identifies the tables needing vacuum or analyze automatically, but only when *autovacuum* is enabled. This ensures that database engine heals itself and avoids creating more fragmentation.

The parameters that are responsible for autovacuum in PostgreSQL:

*autovacuum = on*

*track\_counts = on*

*The track\_counts* parameter allows *autovacuum* daemon to access the candidate tables.

Another parameter called *log\_autovacuum\_min\_duration* allows to log the tables on which *autovacuum* spends more time. Setting this parameter to 0 will log every *autovacuumed* table to the log file.

The Autovacuum process automates both the VACUUM and ANALYZE commands on a table, each based on a threshold. This threshold is calculated as the following:

*Autovacuum* **VACUUM** *threshold for a table =*

*autovacuum\_vacuum\_scale\_factor \* number of tuples + autovacuum\_vacuum\_threshold*

*Autovacuum* **ANALYZE** *threshold for a table =*

*autovacuum\_analyze\_scale\_factor \* number of tuples + autovacuum\_analyze\_threshold*

In other words, if the actual number of dead tuples in a table exceeds this effective threshold, because of updates and deletes, the table becomes a candidate for autovacuum VACUUM.

*autovacuum\_vacuum\_scale\_factor* and *autovacuum\_analyze\_scale\_factor* are fraction of the table records that will be added to the equation. For example, a value of 0.1 equals to 10% of the table records.

*autovacuum\_vacuum\_threshold* and *autovacuum\_analyze\_threshold* are the minimum number of obsolete records or DML operations needed to launch autovacuum.

**Example:**

Table with 1000 records and the following autovacuum parameters:

*autovacuum\_vacuum\_scale\_factor = 0.2*

*autovacuum\_vacuum\_threshold = 50*

*autovacuum\_analyze\_scale\_factor = 0.1*

*autovacuum\_analyze\_threshold = 50*

will become a candidate for VACUUM when the total number of obsolete records comes to (0.2 \* 1000) + 50 = 250

and for ANALYZE when the total number of inserts/updates/deletes makes = (0.1 \* 1000) + 50 = 150.

**The reason to tune autovacuum**

In any database one table can have just a few rows, while another can contain millions of records. It is very common situation, where a large table with millions of records may be involved in transactions way more often, the frequency at which a vacuum or an analyze runs automatically could be greater for the table with just a few rows.

For that reason, PostgreSQL allows configuring autovacuum on the table level, which overrides global settings:

|  |
| --- |
| ALTER TABLE Test SET (autovacuum\_vacuum\_scale\_factor = 0, autovacuum\_vacuum\_threshold = 100000); |

For a table with 1 million rows, autovacuum will start once it has more than (0.2 \* 1000000) + 50 = 200050 obsolete records.

The following query helps to identify the number of dead and live tuples and the number of inserts/deletes/updates on a table:

|  |
| --- |
| SELECT  relname,  n\_dead\_tup,  n\_live\_tup,  (n\_dead\_tup / n\_live\_tup) as DeadTuplesRatio,  last\_vacuum,  last\_autovacuum FROM pg\_stat\_user\_tables WHERE relname = 'TABLE\_NAME' |

Tracking this data for a specific interval should help to understand the frequency of DML operations on each table and how to tune autovacuum settings for individual tables.

The ***autovacuum\_max\_workers Server parameter defines*** how many autovacuum processes can be running in parallel within the PostgreSQL instance. The Autovacuum daemon background process starts a worker process for a table that needs a vacuum or an analyze. Which means that if there are 5 databases hosted in the instance with autovacuum\_max\_workers set to 4, then, the 5th database will wait until one of the existing worker processes complete, and only after it waits for the ***autovacuum\_naptime****/*total number of databases*.*

**Autovacuum from the I/O perspective**

Autovacuum can become very I/O intensive when running on large tables with lots of dead tuples during a high-volume workload. Each autovacuum worker reads 8KB pages from the storage or OS cache (which is the default *block\_size* in Postgres) and modifies the pages containing dead tuples. This results in potentially huge Read and Write operations.

**Here are some parameters that can help reduce the impact on the I/O:**

|  |  |  |
| --- | --- | --- |
| [**Parameter**](https://docs.microsoft.com/en-us/azure/postgresql/howto-optimize-autovacuum) | **Description** | **Default value** |
| *autovacuum\_vacuum\_cost\_limit* | Total cost limit autovacuum could reach (combined by all autovacuum jobs). When default value -1 is specified, the regular *vacuum\_cost\_limit* (the accumulated cost that will cause the vacuuming process to sleep. The default value is 200) value will be used. | -1 |
| *autovacuum\_vacuum\_cost\_delay* | The cost delay value used in automatic vacuum operations. If -1 is specified, it will use the value of the regular vacuum\_cost\_delay. The default value is 20 milliseconds. | 20 ms |
| *autovacuum\_naptime* | The minimum delay between autovacuum runs on any given database. In each round, the daemon examines the database and issues VACUUM and ANALYZE commands as needed for tables in that database. The delay is measured in seconds. | 15 s |
| *autovacuum\_max\_workers* | The maximum number of autovacuum processes, other than the autovacuum launcher, that can run at any one time. | 3 |
| *vacuum\_cost\_page\_hit* | Cost of reading a page that is already in shared buffers and does not need a disk read. | 1 |
| *vacuum\_cost\_page\_miss* | Cost of fetching a page that is not in shared buffers. | 10 |
| *vacuum\_cost\_page\_dirty* | Cost of writing to each page when dead tuples are found in it. | 20 |

**Note:** These settings can be overridden for individual tables by changing table storage parameters.

Keeping in mind that the work is equally divided by all the Autovacuum processes running in the instance, up to *autovacuum\_max\_workers.* Increasing the value of *autovacuum\_max\_workers* may cause a delay with the autovacuum execution for the currently running autovacuum workers. Also, increasing the *autovacuum\_vacuum\_cost\_limit* value may increase the amount of I/O and result in performance degradation.

**Note:** Global configuration can be bypassed by setting storage parameters of individual tables.

In OLTP databases with a high workload, the general rule is always recommended to implement a manual VACUUM strategy on frequently changing tables during a time slot of low activity.

# General recommendations for optimize performance - Single server

## Autovacuum

As PostgreSQL uses multi-version concurrency control (MVCC) which allow greater database concurrency, the downside of it is it need appropriate maintenance done by VACUUM and ANALYZE commands.

Azure database for PostgreSQL offers automated maintenance feature called Autovacuum. Autovacuum is a maintenance key process in PostgreSQL that has a direct impact on how efficient the database will operate.

However, Autovacuum should be configured wisely because this operation has a high cost on the overall database resources in the aspects of compute and memory resources, and the locks on database pages.

To achieve best and smooth database operation, autovacuuming parameters adjustment need to be done related to your database characteristics. This Autovacuum maintenance guideline is based on two abstract parameters: need and frequency.

“Need” parameter can be determined by amount of DML operations that evaluates soft-marked rows for deletion. Soft marking identifies dead tuples, that will be purged later to keep the database healthy. The more dead tuples exist, the more the database will experience heavy update or delete operations. The fewer dead tuples exist, the more the database is idle and efficient.

The following sample query monitor the ratio between dead and live tuples that identify the “need” per table.

|  |
| --- |
| SELECT relname, n\_dead\_tup, n\_live\_tup, (n\_dead\_tup/ n\_live\_tup::float) AS DeadTuplesRatio, last\_vacuum, last\_autovacuum  FROM pg\_catalog.pg\_stat\_all\_tables  WHERE n\_live\_tup <> 0  ORDER BY n\_dead\_tup DESC; |

“Frequency” parameter can be determined when the database load is become heavy. For example, database can work intensively on working hours during the day, as a result evaluates more dead tuples than in the night, which makes the need of vacuum process to run more frequently. The variant frequency makes running vacuum jobs manually is inconvenient for maintenance.

Autovacuum can be configured and benefits from tuning based on need and frequency.

Here are some autovacuum configuration parameters that you can update, along with some guidance:

|  |  |  |
| --- | --- | --- |
| **Parameter** | **Description** | **Default** |
| autovacuum\_vacuum\_threshold | Specifies the minimum number of updated or deleted tuples needed to trigger a vacuum operation in any one table.  Set this parameter only in the postgresql.conf file or on the server command line. To override the setting for individual tables, change the table storage parameters. | 50 |
| autovacuum\_vacuum\_scale\_factor | Specifies a fraction of the table size to add to autovacuum\_vacuum\_threshold when deciding whether to trigger a vacuum operation.  Set this parameter only in the postgresql.conf file or on the server command line. To override the setting for individual tables, change the table storage parameters. | 0.2  (20% of table) |
| autovacuum\_vacuum\_cost\_limit | Specifies the cost limit value used in automatic vacuum operations. If -1 is specified, the regular vacuum\_cost\_limit value is used. If there's more than one worker, the value is distributed proportionally among the running autovacuum workers. The sum of the limits for each worker doesn't exceed the value of this variable.  Set this parameter only in the postgresql.conf file or on the server command line.  To override the setting for individual tables, change the table storage parameters. | -1 |
| autovacuum\_vacuum\_cost\_delay | Specifies the cost delay value used in automatic vacuum operations. If -1 is specified, the regular vacuum\_cost\_delay value is used. The default value is 20 milliseconds.  Set this parameter only in the postgresql.conf file or on the server command line.  To override the setting for individual tables, change the table storage parameters. | 20 ms |
| autovacuum\_naptime | Specifies the minimum delay between autovacuum runs on any given database. In each round, the daemon examines the database and issues VACUUM and ANALYZE commands as needed for tables in that database. The delay is measured in seconds.  Set this parameter only in the postgresql.conf file or on the server command line. | 15 sec |
| autovacuum\_max\_workers | Specifies the maximum number of autovacuum processes, other than the autovacuum launcher, that can run at any one time.  Set this parameter only at server start. | 3 |

Based on the above, Autovacuum is triggered to work when the number of dead tuples exceeds:  
*autovacuum\_vacuum\_threshold + autovacuum\_vacuum\_scale\_factor \* reltuples*

Here, reltuples is a constant.

Some rules of thumb that you need bear in mind when using autovacuum:

* Cleanup from autovacuum must keep up with the database load. Otherwise, you might run out of storage and experience a general slowdown in queries as well. Over time, the rate at which a vacuum operation cleans up dead tuples should equal the rate at which dead tuples are created, as a result, the storage usage will be stabilized.
* Databases with many updates and deletes have more dead tuples and need more space. Generally, databases with many updates and deletes benefit from low values of autovacuum\_vacuum\_scale\_factor and autovacuum\_vacuum\_threshold.
* The low values prevent prolonged accumulation of dead tuples. You can use higher values for both parameters with smaller databases because the need for vacuuming is less urgent.
* Frequent vacuuming comes at the cost of compute and memory.

At the bottom line, autovacuum should be used with a balance between the need and frequency based on the database characteristics and workload behavior.

For more information: [Optimize autovacuum on an Azure Database for PostgreSQL - Single Server](https://docs.microsoft.com/en-us/azure/postgresql/howto-optimize-autovacuum)

## Bulk inserts

For environments involving workload operations that insert large datasets in bulk, consider using unlogged tables.

“Unlogged tables” is a PostgreSQL feature that can be used effectively to optimize bulk inserts while it uses Write-Ahead Logging (WAL). It provides atomicity and durability by default, which are you need to support reliable relational database.

Inserting into an unlogged table means that PostgreSQL does inserts without writing into the transaction log, which is more efficient because the log writing itself is an I/O intensive operation. As a result, these tables are considerably faster than ordinary tables.

You can create new unlogged table or alter an existing logged table very easily:

|  |
| --- |
| CREATE UNLOGGED TABLE <tableName>.  ALTER TABLE <tableName> SET UNLOGGED.  ALTER TABLE <tableName> SET LOGGED. |

Some customer workloads have experienced approximately a 15 percent to 20 percent performance improvement when unlogged tables were used.

Be aware:

* Unlogged tables are not crash-safe.
* An unlogged table is automatically truncated after a crash or subject to an unclean shutdown.
* The contents of an unlogged table also are not replicated to standby servers.
* Any indexes created on an unlogged table are automatically unlogged as well.

After the bulk insert operation completes, make sure changing back the table to “logged” so that the insert is durable.

For more information: [Optimize bulk inserts and use transient data on an Azure Database for PostgreSQL - Single Server](https://docs.microsoft.com/en-us/azure/postgresql/howto-optimize-bulk-inserts)

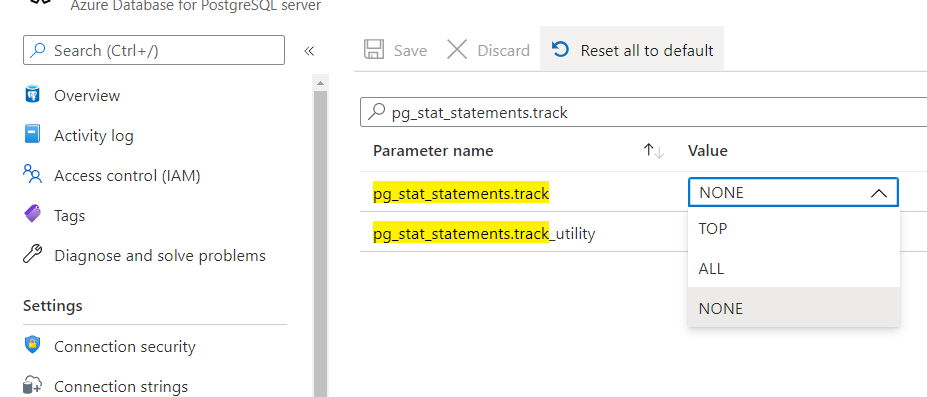
## Query statistics collection

*Pg\_stat\_statements* is a PostgreSQL extension that's enabled by default in Azure Database for PostgreSQL. The extension provides a means to track execution statistics for all SQL statements executed by a server. This module hooks into every query execution and comes with a non-trivial performance cost. For best performance it is highly recommended to disable *Pg\_stat\_statements* extension, because enabling *pg\_stat\_statements* force query text writes to files on disk and, as a result, reduce storage throughput for the database itself.

To do so:

* In the Azure portal:

Change the setting to pg\_stat\_statements.track = NONE , as shown:



* In the Azure CLI:

|  |
| --- |
| az postgres server configuration set to  --name pg\_stat\_statements.track  --resource-group myresourcegroup  --server mydemoserver --value NONE |

Some customer workloads have seen up to a 50 percent performance improvement when *pg\_stat\_statements* is disabled. The tradeoff you make when you disable pg\_stat\_statements is the inability to troubleshoot performance issues. As an alternative to *Pg\_stat\_statements*, it is recommended to use [Query Store](https://docs.microsoft.com/en-us/azure/postgresql/concepts-query-store) feature in Azure Database for PostgreSQL.Query store provides a more effective method to track to track query statistics.

For more information: [Optimize query statistics collection on an Azure Database for PostgreSQL - Single Server](https://docs.microsoft.com/en-us/azure/postgresql/howto-optimize-query-stats-collection)

# Database checklist and quick tips

PostgreSQL, like any database software, requires that certain tasks be performed regularly to achieve optimum performance. One obvious maintenance task is the creation of backup copies of the data on a regular schedule. Without a recent backup, you have no chance of recovery after a catastrophe (disk failure, fire, mistakenly dropping a critical table, etc.).

The other main category of maintenance task is periodic "vacuuming" of the database. Closely related to this is updating the statistics that will be used by the query planner. Another task that might need periodic attention is log file management.

## Daily cleaning

PostgreSQL database requires periodic cleaning and maintenance. For many installations, it is sufficient to have the automatic cleanup daemon perform cleanup. However, it is recommended that VACUUM command will be periodically processing each table for several reasons:

1. Restore or reuse the disk space occupied by updated or deleted rows.
2. Update the data statistics used by the PostgreSQL query planner (ANALYZE).
3. Update the visibility map, it can speed up the scan using only the index.
4. Protect old data from being lost due to transaction ID rewinding or multi-transaction ID rewinding.

This topic is covered in [Autovacuum](#_Autovacuum).

## Recovering disk space

In PostgreSQL, an UPDATE or DELETE of a row does not immediately remove the old version of the row. This approach is necessary to gain the benefits of multi-version concurrency control (MVCC): the row version must not be deleted while it is still potentially visible to other transactions. But eventually, an outdated or deleted row version is no longer of interest to any transaction. The space it occupies must then be reclaimed for reuse by new rows, to avoid unbounded growth of disk space requirements. This is done by running VACUUM.

There are two types of VACUUM:

* Standard VACUUM
* VACUUM FULL

VACUUM FULL can reclaim more disk space but runs slower. VACUUM FULL requires an exclusive lock on the table it works on, so it cannot be paralleled with other uses of this table. Standard VACUUM can run in parallel and CRUD operations (SELECT, INSERT, UPDATE, and DELETE) will continue to work normally. However, during the clean-up table definition commands cannot be issued.

**Note:** VACUUM might generate a lot of I/O traffic, which will cause poor performance of other active sessions.

The standard form of VACUUM removes the dead row version in tables and indexes and marks the space as reusable in the future. However, it will not return the space to the operating system unless in exceptional circumstances one or more pages at the end of the table become completely free, and an exclusive table lock can be easily obtained.  
In contrast, VACUUM FULL actively shrinks tables by writing a completely new version of the table file with no dead space. This minimizes the size of the table but can be significantly longer. It also requires extra disk space for the new copy of the table, until the operation completes.

**Note:** The general goal of routine cleanup is to do more standard VACUUM to avoid the need for VACUUM FULL.

**Note:** When a table contains many dead row versions due to many updates or delete activities, pure VACUUM may not be satisfactory. If you have such a table and you need to reclaim the excess disk space it takes up, you will need to use VACUUM FULL, or CLUSTER, or one of the table rewrite variants of ALTER TABLE.

**Note:** If you have a table, its entire content will be deleted periodically, consider using TRUNCATE instead of DELETE and then VACUUM.

## Update visibility map

Vacuum also maintains a visibility map for each table, which is used to track which pages only contain tuples that are visible to all active transactions.

This has two purposes. First, the cleanup itself can skip such pages in the next run because nothing in it needs to be cleaned up. Second, this allows PostgreSQL to answer some queries that only use indexes without having to refer to the underlying table. Since PostgreSQL indexes do not contain tuple visibility information, a normal index scan fetches the heap tuple for each matching index entry, to check whether it should be seen by the current transaction. An index-only scan, on the other hand, checks the visibility map first. If it is known that all tuples on the page are visible, the heap fetch can be skipped. This is most useful on large data sets where the visibility map can prevent disk accesses. The visibility map is vastly smaller than the heap, so it can easily be cached even when the heap is very large.

## Automatic clean up background processes

PostgreSQL has an optional but highly recommended feature called autovacuum, whose purpose is to automate the execution of VACUUM and ANALYZE commands. When it is enabled, autovacuum checks for tables that have had many inserted, updated or deleted tuples. These checks use the statistics collection facility; therefore, autovacuum cannot be used unless [*track\_counts* parameter](https://postgrespro.com/docs/postgrespro/12/runtime-config-statistics#GUC-TRACK-COUNTS) is set to true. In the default configuration, autovacuuming is enabled and the related configuration parameters are appropriately set.

The “autovacuum daemon” consists of multiple processes. There is a persistent daemon process, called the autovacuum launcher, which oversees starting autovacuum worker processes for all databases. The launcher will distribute the work across time, attempting to start one worker within each database every [*autovacuum\_naptime*](https://postgrespro.com/docs/postgrespro/12/runtime-config-autovacuum#GUC-AUTOVACUUM-NAPTIME) seconds. (Therefore, if the installation has N databases, a new worker will be launched every autovacuum\_naptime/N seconds.)

A maximum of [autovacuum\_max\_workers](https://postgrespro.com/docs/postgrespro/12/runtime-config-autovacuum#GUC-AUTOVACUUM-MAX-WORKERS) worker processes are allowed to run at the same time. If there are more than autovacuum\_max\_workers databases to be processed, the next database will be processed as soon as the first worker finishes. Each worker process will check each table within its database and execute VACUUM and / or ANALYZE as needed. The [*log\_autovacuum\_min\_duration*](https://postgrespro.com/docs/postgrespro/12/runtime-config-autovacuum#GUC-LOG-AUTOVACUUM-MIN-DURATION) parameter can be set to monitor autovacuum workers' activity.

If several large tables all become eligible for vacuuming in a short amount of time, all autovacuum workers might become occupied with vacuuming those tables for a long period. This would result in other tables and databases not being vacuumed until a worker becomes available. There is no limit on how many workers might be in a single database, but workers do try to avoid repeating work that has already been done by other workers.

**Note:** The number of running workers does not count towards [*max\_connections*](https://postgrespro.com/docs/postgrespro/12/runtime-config-connection#GUC-MAX-CONNECTIONS) or[*superuser\_reserved\_connections*](https://postgrespro.com/docs/postgrespro/12/runtime-config-connection#GUC-SUPERUSER-RESERVED-CONNECTIONS) limits.

Tables whose relfrozenxid value is more than [autovacuum\_freeze\_max\_age](https://postgrespro.com/docs/postgrespro/12/runtime-config-autovacuum#GUC-AUTOVACUUM-FREEZE-MAX-AGE) transactions old are always vacuumed (this also applies to those tables whose freeze max age has been modified via storage parameters; see below). Otherwise, if the number of tuples obsoleted since the last VACUUM exceeds the “vacuum threshold”, the table is vacuumed. The

vacuum threshold is defined as:

*vacuum threshold = vacuum base threshold + vacuum scale factor \* number of tuples*

where the vacuum base threshold is [*autovacuum\_vacuum\_threshold*](https://postgrespro.com/docs/postgrespro/12/runtime-config-autovacuum#GUC-AUTOVACUUM-VACUUM-THRESHOLD), the vacuum scale factor is [*autovacuum\_vacuum\_scale\_factor*](https://postgrespro.com/docs/postgrespro/12/runtime-config-autovacuum#GUC-AUTOVACUUM-VACUUM-SCALE-FACTOR), and the number of tuples is *pg\_class.reltuples*. The number of obsolete tuples is obtained from the *statistics collector*; it is a semi-accurate count updated by each UPDATE and DELETE operation. (It is only semi-accurate because some information might be lost under heavy load.) If the *relfrozenxid* value of the table is more than *vacuum\_freeze\_table\_age* transactions old, an aggressive vacuum is performed to freeze old tuples and advance *relfrozenxid*; otherwise, only pages that have been modified since the last vacuum are scanned.

For analyze, a similar condition is used: the threshold, defined as:

*analyze threshold = analyze base threshold + analyze scale factor \* number of tuples*

is compared to the total number of tuples inserted, updated, or deleted since the last ANALYZE.

Temporary tables cannot be accessed by autovacuum. Therefore, appropriate vacuum and analyze operations should be performed via session SQL commands. Autovacuum workers generally do not block other commands. If a process attempts to acquire a lock that conflicts with the SHARE UPDATE EXCLUSIVE lock held by autovacuum, lock acquisition will interrupt the autovacuum. However, if the autovacuum is running to prevent transaction ID wraparound (i.e., the autovacuum query name in the *pg\_stat\_activity* view ends with (to prevent wraparound)), the autovacuum is not automatically interrupted.

## Periodically rebuilding indexes

In some cases, it is worth periodically using the REINDEX command or a series of independent reconstruction steps to rebuild the index. B-tree index pages that have become completely empty are reclaimed and reused. However, there is still a possibility of inefficient space utilization: if all keys except a few index keys on a page are deleted, the page is still allocated. Therefore, in this usage mode in which most but not all keys in each range are eventually deleted, the space usage is very poor. For such usage patterns, periodic reindexing is recommended.

The possible expansion of non-B-tree indexes has not been well quantified. It is a good idea to regularly monitor the physical size of the index when using non-B-tree indexes. For a B-tree index, a newly created index is accessed slightly faster than an index that has been updated multiple times, because in the newly created index, logically adjacent pages are usually also physically adjacent. Just to improve access speed is also worth re-indexing regularly.

REINDEX can be used safely and easily in all situations. But since the command requires an exclusive table lock, a better approach is to use a sequence of creation and replacement steps to perform index reconstruction.  
Index types that support CREATE INDEX with the CONCURRENTLY option can be rebuilt in this way.