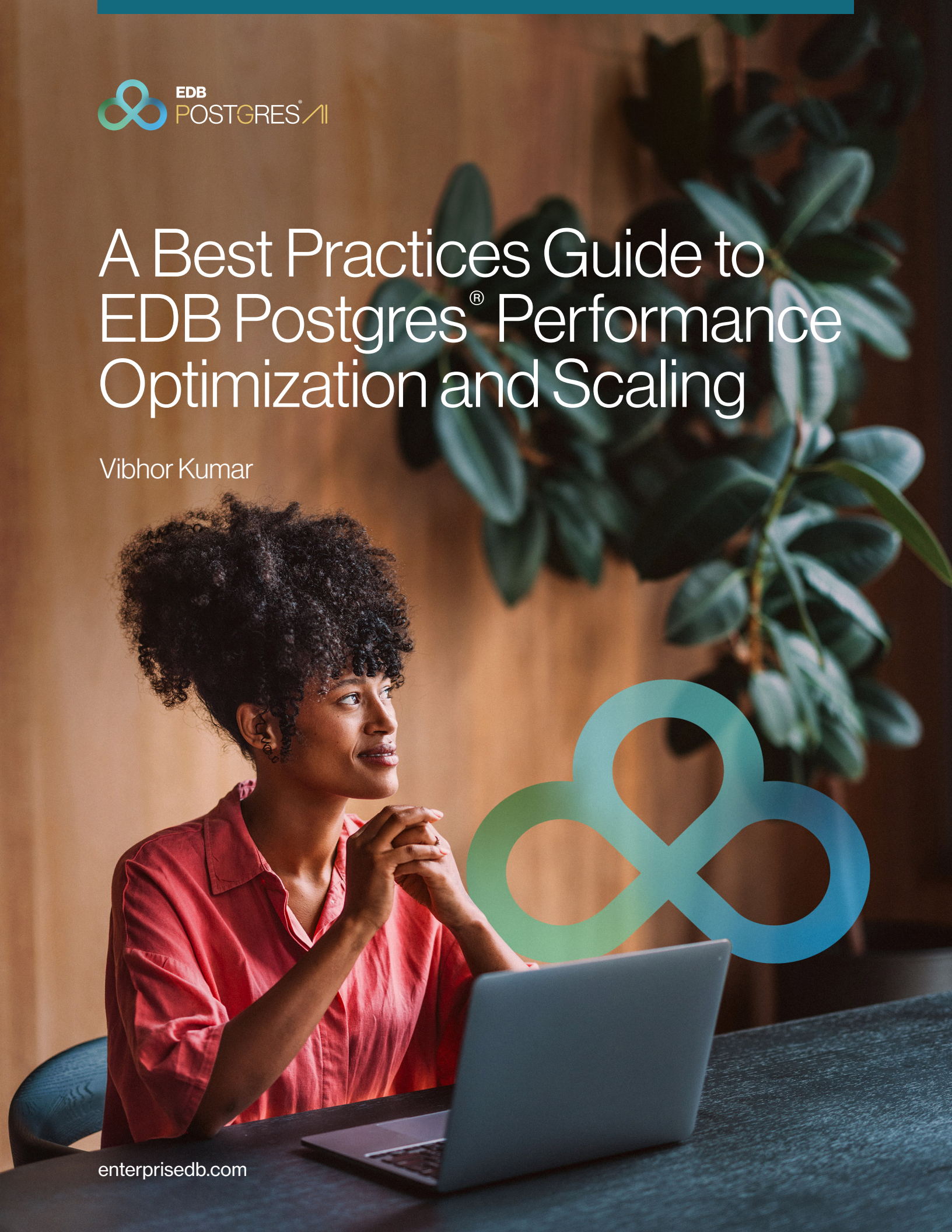
EDB POSTGRES AI

A Best Practices Guide to EDB Postgres® Performance Optimization and Scaling

Vibhor Kumar



enterprisedb.com

Table of contents

The foundation: Infrastructure and sizing 4

Hardware architecture for peak performance 4

CPU: The engine of computation 4

Memory (RAM): The primary performance accelerator 5

Storage subsystems: The I/O backbone 5

Network: The overlooked bottleneck 6

Operating system tuning for Linux 6

Filesystem selection and mount options 7

I/O scheduler and read-ahead 7

Kernel parameter tuning with `sysctl` 7

Mastering memory: Huge pages 8

A framework for scientific capacity planning 8

Configuration and core tuning 9

Mastering `postgresql.conf` 10

Connection management and pooling 10

Memory allocation: A delicate balance 11

WAL tuning 11

Query planner cost parameters 12

Autovacuum: The unsung hero of performance 12

Query and application performance optimization 13

Identifying and analyzing performance bottlenecks 13

Proactive logging 13

Real-time analysis 14

Aggregate analysis with pg_stat_statements 14

Deconstructing execution plans with EXPLAIN 14

Plan overview 16

Key observations in this plan 16

Advanced indexing strategies 17

Data partitioning for large tables 18

Table of Contents

Architecting for scalability and high availability 19

- Vertical vs. horizontal scaling: A strategic comparison 19
- Read scalability with streaming replication 19
- Advanced architectures with logical replication 20
- Active-active and geo-distribution with EDB Postgres Distributed 20

Version-specific features and enterprise tooling 21

- EDB Postgres Advanced Server enhancements 21
  - Enterprise performance and management tooling 21
  - The evolution of performance diagnostics 22
- The performance revolution in EDB Postgres 18 22
  - Deep dive: Asynchronous I/O (AIO) 22
  - Deep dive: B-tree skip scans 23
  - Operational excellence: Preserving planner statistics 23

Conclusion 24

The foundation: Infrastructure and sizing

Achieving optimal performance and scalability with EDB Postgres is not an isolated database administration task; it is a holistic engineering discipline that begins with the foundational layers of hardware and the operating system. The most meticulously tuned database configuration or brilliantly optimized query can be severely handicapped by an improperly sized or configured underlying infrastructure. This section establishes the critical principles for architecting this foundation, moving beyond generic recommendations to provide specific, actionable guidance for designing and sizing systems capable of sustaining demanding, large-scale EDB Postgres workloads. A holistic approach is paramount; the performance of the entire database system is ultimately governed by its most constrained component. A surplus of CPU cores, for instance, offers diminishing returns if the storage subsystem cannot deliver data at a commensurate rate, resulting in processors sitting idle while waiting for I/O. Therefore, the selection and tuning of infrastructure components must be viewed as an exercise in balancing the entire data path, from CPU cache to network interface, to create a synergistic and high-performance whole.

Hardware architecture for peak performance

The selection of physical or virtual hardware is the first and most critical decision in deploying a high-performance EDB Postgres instance. Each component—CPU, memory, storage, and network—plays a distinct and interconnected role in the overall performance profile of the database.

CPU: The engine of computation

The CPU is where data is actively processed. For EDB Postgres, which employs a multi-process architecture in which each client connection is typically handled by its own dedicated server process, the choice of CPU involves a careful balance of core count, clock speed, and cache size.

- **Core count vs. clock speed:** The ideal balance between the number of CPU cores and their individual clock speed is highly dependent on the database workload. While a universal formula is impractical due to this variance, clear guidelines exist for different workload types:
  - **OLTP/high concurrency:** A higher number of CPU cores is beneficial for workloads characterized by high concurrency, such as online transaction processing (OLTP) systems with many simultaneous client connections. The operating system can schedule these numerous EDB Postgres backend processes across many cores, reducing context switching and contention.
  - **OLAP/mixed workload:** Conversely, operations that are inherently single-threaded or less parallelized, such as certain maintenance tasks or complex analytical functions within a single query, benefit more from higher individual core clock speeds. For mixed OLTP/OLAP workloads, a balanced approach is necessary, often favoring a high core count with a reasonably high clock speed.
- **The critical role of L3 cache:** The L3 cache, a larger pool of memory shared across all CPU cores, is a critical performance determinant, particularly for data-intensive workloads. A larger L3 cache can hold more data and instructions, significantly reducing the frequency of slower fetches from main system RAM. This is especially impactful for analytical queries that scan large datasets or for parallel queries in which multiple worker processes need to access the same data segments. A larger L3 cache directly translates to lower memory latency and higher CPU efficiency.
- **NUMA considerations:** Modern multi-socket servers often use a non-uniform memory access (NUMA) architecture, in which each CPU socket has its own local memory bank. Accessing local memory is significantly faster than accessing memory attached to another CPU socket. When deploying EDB Postgres on virtual machines within a NUMA system, it is a critical best practice to employ NUMA pinning. This ensures that a virtual machine's vCPUs and its allocated memory reside on the same physical NUMA node, preventing performance degradation caused by high-latency, cross-socket memory access.

Memory (RAM): The primary performance accelerator

Random access memory (RAM) is arguably the most effective hardware component for accelerating database performance. Its primary role is to cache data and reduce the need for slow disk I/O operations. When procuring new hardware, it is advisable to install as much RAM as possible initially, as future upgrades can be more costly and require downtime.

- **Sizing for the working set:** The fundamental principle of RAM sizing for EDB Postgres is to allocate enough memory to hold the "working set" or "hot data"—the portion of the database that is most frequently accessed—within the EDB Postgres shared buffer cache. When the working set fits in RAM, read operations are served at memory speed, which is orders of magnitude faster than from even the fastest storage devices. Estimating the working set size can be done by analyzing query patterns, understanding application access logic, or using monitoring tools to observe block access frequencies on an existing system.
- **Beyond the buffer pool:** Sizing RAM requires looking beyond just the `shared_buffers` parameter. Significant memory is also consumed by:
  - **Connection overhead:** Each backend process for a client connection consumes a nontrivial amount of private memory. A server with thousands of connections will require several gigabytes of RAM just for connection management.
  - **Query operations (`work_mem`):** Complex queries that perform sorts or hash joins allocate memory up to the limit defined by `work_mem` for each such operation. A single query can have multiple sort/hash nodes, each consuming `work_mem`.
  - **Operating system file cache:** The operating system (OS) maintains its own file system cache, which is also crucial for performance. Any RAM not allocated to EDB Postgres is available for the OS to use for caching, which benefits operations that read data not currently in `shared_buffers`.

Storage subsystems: The I/O backbone

The storage subsystem is where the database persists its data and is often the most significant performance bottleneck. The choice of storage technology and its configuration must be directly aligned with the application's I/O profile.

- **IOPS vs. latency:** It is crucial to distinguish between IOPs and latency—the time it takes to complete a single I/O operation. Analytical (OLAP) workloads, which often involve large sequential scans, are more sensitive to throughput, or IOPS. For virtually all modern, performance-critical EDB Postgres deployments, NVMe or SSD storage is mandatory due to their extremely low latency compared to traditional spinning disks (HDDs).
- **Storage selection for OLTP vs. OLAP workloads:** The distinction between workload types directly informs the optimal storage technology selection:
  - **For OLTP systems (optimized for latency):** As these workloads are defined by high volumes of small, random read/write operations, they are extremely sensitive to I/O latency. The primary goal is to minimize the time for each transaction. Therefore, high-performance solid-state storage is the standard. NVMe SSDs offer the lowest possible latency and are the ideal choice for the most demanding, performance-critical transactional applications. Standard SSDs are also a viable and mandatory choice over mechanical disks.
  - **For OLAP systems (optimized for throughput):** These workloads are characterized by large, sequential scans of data and are more sensitive to overall throughput than to the latency of any single I/O operation. While modern SSDs also provide excellent sequential throughput, a cost-effective and high-performance solution for data warehousing can involve arrays of traditional HDDs in a RAID 10 configuration. This setup can deliver the high sequential read speeds needed for large analytical queries. However, for mixed-use systems or OLAP environments in which query response time is still a critical factor, SSDs remain the superior choice.

- **Strategic RAID configurations:** Redundant array of independent disks (RAID) configurations should be chosen to match the I/O patterns of different database components.
  - **Write-ahead logging (WAL):** The WAL is a write-intensive, sequential log. For this, RAID 1 or RAID 10 is strongly recommended. These configurations provide redundancy and the low write latency needed for high transaction throughput. Placing the WAL on a physically separate set of drives from the main data directory is a foundational best practice.
  - **Data directory:** The main data directory experiences a mix of random reads and writes. RAID 10 offers the best combination of performance for both reads and writes, along with excellent redundancy, making it the standard choice for production data volumes.
- **Leveraging multiple disks and tablespaces:** To overcome the I/O limits of a single storage volume, the database workload can be spread across multiple physical devices. This can be achieved by placing the WAL, primary data directory, and indexes on separate physical disk arrays. Furthermore, the EDB Postgres tablespaces feature enables a database administrator to define storage locations on different file systems. This enables a tiered storage strategy, in which high-transaction "hot" tables can be placed on the fastest NVMe storage while less frequently accessed archival data can reside on slower, more cost-effective SSDs. Using separate tablespaces and drives for indexes and data can also increase performance, especially on SATA drives, though this is less necessary for SSD and NVMe drives.
- **Virtual machine storage considerations:** When running EDB Postgres in a virtual machine, pre-allocating disks can prevent the host from allocating space during database operations. For environments using copy-on-write (COW) file systems, performance may be improved by disabling `wal_recycle` and `wal_init_zero` in `EDB Postgres.conf`. Disabling these parameters can make creating new WAL files faster than the default recycling behavior.

Network: The overlooked bottleneck

While often considered secondary for a single-node database, the network interface becomes a critical performance component in any scaled-out architecture. Faster or bonded network cards can be beneficial. Network throughput can become the limiting factor for:

- **Streaming replication:** The speed at which WAL records can be sent from the primary to standby replicas directly impacts replication lag.
- **Backup and recovery:** Performing a base backup of a multi-terabyte database can saturate a 1 Gbps network link for hours. Faster, bonded network interfaces (e.g., 10 Gbps or higher) are essential for meeting recovery time objectives (RTOs).
- **Client connectivity:** In systems with a high number of concurrent connections, network bandwidth can limit the overall throughput between the application servers and the database.

Operating system tuning for Linux

The Linux kernel offers a vast array of tunable parameters that can have a profound impact on EDB Postgres performance. Proper OS tuning ensures that the kernel's behavior is aligned with the specific demands of a database workload. Effective tuning is an exercise in shifting control and intelligence. By adjusting kernel parameters, an administrator instructs the operating system to defer to the database's or the hardware's own management capabilities. For example, setting `vm.swappiness` to a low value effectively tells the kernel, "EDB Postgres knows how to manage its memory better than you do." Similarly, using the none I/O scheduler for NVMe storage tells the kernel, "The storage device knows how to manage its I/O queue better than you do." This process of removing layers of abstraction and potential misinterpretation between the database and the hardware is key to unlocking maximum performance.

Filesystem selection and mount options

- **XFS as the de facto standard:** For large-scale database deployments on Linux, the **XFS** file system is the most popular and recommended choice. It is mature, robust, and performs exceptionally well with the large files that EDB Postgres creates. It is the default file system on major enterprise distributions, such as RHEL. It is critical to use the file system with journaling enabled; turning it off risks data corruption and is not a supported configuration. While other file systems, such as Btrfs, are evolving, they are not yet considered production-ready for mission-critical database workloads. Best practices recommend using separate mount points for the data directory (`PGDATA`), the `WAL`, and the EDB Postgres log directory.
- **Mount options for performance:** When mounting the file systems that will host EDB Postgres data, the `noatime` option should always be used. By default, Linux updates a file's metadata every time it is read (`atime`). This operation is useless for EDB Postgres and creates a constant, low-level overhead of CPU cycles and I/O writes. Disabling it with `noatime` provides a small but meaningful performance boost with no downside. An example entry in `/etc/fstab` would be:

  ```
  /dev/mapper/pgdata-vg-data /pgdata xfs defaults,noatime 0 0
  ```

I/O scheduler and read-ahead

- **I/O scheduler configuration:** The I/O scheduler in the Linux kernel manages the order in which block I/O operations are submitted to the storage device. The optimal choice depends on the underlying hardware.
  - **Modern kernel recommendations:** For modern Linux kernels (e.g., RHEL 8 and newer) and modern storage:
    - **NVMe/SSDs:** The `none` (or `noop` for older kernels such as RHEL 7) scheduler is recommended. These devices have no moving parts and benefit from their own sophisticated internal logic for handling parallel I/O requests. An OS-level scheduler adds unnecessary overhead and can hinder the device's native performance.
    - **Spinning disks (HDDs):** The `mq-deadline` (or deadline for RHEL 7) scheduler is preferred. It attempts to order I/O requests to minimize disk head movement, which is the primary source of latency on mechanical drives.
- **Disk read-ahead:** Increasing the disk read-ahead setting can improve I/O throughput by fetching more data from the disk in a single request. A common recommendation is to increase this value from its typical default of 128 KB to 4096 KB on the disks hosting the database.

Kernel parameter tuning with `sysctl`

These parameters, typically set in `/etc/sysctl.conf`, control the low-level behavior of the Linux kernel's virtual memory and I/O subsystems.

- **Write caching (`vm.dirty_*`):** The kernel buffers writes in memory (the page cache) before flushing them to disk. The `vm.dirty_background_ratio/bytes` and `vm.dirty_ratio/bytes` parameters control this behavior. Default settings can lead to large amounts of dirty data accumulating in memory, which, when flushed, causes an "I/O storm" that can stall all other disk activity. Tuning these values lower forces the kernel to flush data to disk more frequently but in smaller, more manageable chunks. This transforms bursty, disruptive I/O into a smooth, continuous stream, which is far better for consistent database performance. For modern hardware, it is often better to set these limits in absolute bytes rather than as a ratio of total memory. A suitable starting point is setting `vm.dirty_bytes to 1GB and vm.dirty_background_bytes` to one quarter of that value, allowing the kernel to trickle data to the disk subsystem long before the hard limit is reached.

- **Swapping behavior (`vm.swappiness`):** This parameter, with a value from 0 to 100, controls the kernel's preference for swapping out application memory versus dropping pages from the OS file cache. For a dedicated database server, swapping is extremely detrimental to performance. Therefore, `vm.swappiness` should be set to a very low value, such as 1 or 10. This instructs the kernel to strongly avoid swapping EDB Postgres's memory to disk, preferring instead to reclaim memory from the less critical file cache.
- **Memory overcommit (`vm.overcommit_memory`):** To ensure the stability of the database, it is recommended to set `vm.overcommit_memory` to 2. This setting tells the kernel to deny memory allocation requests that would exceed the total available swap space plus a configurable percentage of physical RAM (set by `vm.overcommit_ratio`). This prevents a situation in which a runaway process allocates excessive memory, triggering the out-of-memory (OOM) killer, which might otherwise terminate the main EDB Postgres postmaster process, causing a database crash.

Mastering memory: Huge pages

The standard memory page size in Linux is 4KB. An EDB Postgres instance with a large `shared_buffers` setting (e.g., 64 GB) will be managing millions of these small pages. This creates significant overhead for the CPU's memory management unit (MMU), which must maintain large page tables to map virtual to physical memory addresses. This overhead can lead to performance degradation from frequent misses in the translation lookaside buffer (TLB), a specialized cache for these mappings.

- **Concept and benefit:** Huge pages allow the kernel to manage memory in much larger chunks, typically 2MB or 1GB. By using huge pages for EDB Postgres's shared memory, the number of pages the MMU has to manage is drastically reduced. This results in smaller page tables and a much higher TLB hit rate, directly accelerating memory access and providing a significant performance boost for large-scale databases.
- **Implementation:** The correct implementation involves two steps:
  1. **Disable transparent huge pages (THP):** THP is a system that attempts to use huge pages automatically, but it is known to cause performance stalls and memory fragmentation issues with databases. It must be disabled system-wide.
  2. **Configure static huge pages:** The administrator must explicitly calculate the number of huge pages required to accommodate EDB Postgres's `shared_buffers` and other shared memory segments, and then pre-allocate them at boot time via kernel parameters. EDB Postgres must then be configured to use these pre-allocated pages. This requires a database restart to take effect.

A framework for scientific capacity planning

Hardware provisioning should not be guesswork; it should be a data-driven process. A structured framework ensures that resources are aligned with workload demands, avoiding both costly over-provisioning and performance-limiting under-provisioning.

- **Phase 1: Workload characterization:** The first step is to thoroughly understand and document the application's workload.
  - **Use case:** Is the primary workload transactional (OLTP), analytical (OLAP), or a hybrid?
  - **Load metrics:** Estimate or measure peak and average concurrent connections, transactions per second (TPS), queries per second (QPS), and the read/write ratio.
  - **Growth projections:** Project the expected growth in data volume and workload over a one- to three-year horizon.
- **Phase 2: Initial hardware sizing:** With a workload profile established, initial sizing can be performed using established heuristics and formulas. For an existing system being migrated, this phase should be informed by baseline performance metrics (CPU usage, %iowait, cache hit ratio) from the current environment.

- **CPU core estimation:** A reasonable starting point can be derived from concurrency or data size. For example, one might use a concurrency-based formula such as `Core Count = (Peak Concurrent Connections / 10) * 2`, or a data-based one such as `Core Count = Data Size in GB / 60`, and take the higher of the two. This must be adjusted for replication overhead (e.g., add one core per streaming standby, or two to four for a logical publisher).
- **RAM estimation:** A formulaic approach can provide a solid baseline: `RAM (GB) = (Hot Data Size (GB) * 0.3) + (Number of Cores * 2GB) + OS Overhead (4GB)`. The factor applied to the hot data size should be increased for read-heavy workloads.
- **Storage estimation:** Storage needs should account for raw data size, an expansion factor for indexes and overhead (typically 2x–3x), the data retention period, and projected growth rates.

- **Phase 3: Validation and iteration:** The initial sizing is a hypothesis that must be tested.
  - **Load testing:** Before going into production, the provisioned hardware should be subjected to realistic load testing using tools such as pgbench to simulate the expected peak workload.
  - **Continuous monitoring:** Once in production, the system must be continuously monitored. Key metrics including CPU utilization, I/O latency, RAM usage, and application response times provide the feedback loop necessary to validate the initial sizing.
  - **Iterative adjustment:** Based on monitoring data, hardware should be iteratively adjusted. If the system is under-provisioned, resources should be scaled up. If it is consistently over-provisioned, resources can be rightsized to optimize costs, always maintaining a buffer for spikes and future growth.

Configuration and core tuning

Once a solid infrastructure foundation is in place, the focus shifts to configuring the EDB Postgres engine itself. The `postgresql.conf` file is the central control panel, containing hundreds of parameters that govern every aspect of the database's behavior, from memory allocation to query planning logic. A default EDB Postgres configuration is designed to run on minimal hardware and is grossly inadequate for any production system at scale. Effective tuning involves transforming this default configuration into an accurate model of the high-performance environment in which the database operates.

This is a critical concept: Parameters such as `effective_cache_size` and `random_page_cost` do not directly allocate resources; they describe the system's reality to the query planner. The more accurately this configuration file describes the fast hardware and abundant memory available, the more intelligent the planner's decisions will be. An inaccurate configuration leads to suboptimal query plans, such as choosing a full table scan when a fast index scan on an SSD would have been orders of magnitude faster, effectively negating the benefits of expensive hardware.

Parameter	Category	Recommended Starting Value / Formula	Rationale and Impact
shared_buffers	Memory	25% of total RAM (up to a max of ~64GB)	The primary data cache. Sizing it correctly is the single most important memory tuning step.
effective_cache_size	Memory	75% of total RAM	Informs the planner about the total available cache (Postgres + OS), influencing the choice between index and sequential scans.
work_mem	Memory	((Total RAM - shared_buffers) / (16 x CPU cores))	Memory-per-query operation. Crucial for sort/hash performance but can lead to OOM if set too high. Must be balanced with concurrency.
maintenance_work_mem	Memory	15% x (Total RAM - shared_buffers) / autovacuum_max_workers (up to 1GB)	Speeds up critical maintenance tasks such as **VACUUM** and **CREATE INDEX** by allowing them to operate in memory.
max_wal_size	WAL	16GB–64GB (or more)	The primary control for checkpoint frequency. A large value smooths I/O by making checkpoints time based, not activity based.
checkpoint_completion_target	WAL	0.9	Spreads checkpoint I/O over the entire interval between checkpoints, preventing I/O spikes.
random_page_cost	Planner	1.1 (for SSD/NVMe), 2.0–3.0 (for SAN), 4.0 (for HDD)	Models the cost of nonsequential I/O. Aligning this with your storage hardware is critical for correct plan choices.
autovacuum_vacuum_scale_factor	Autovacuum	0.05–0.1	Makes autovacuum trigger on a smaller percentage of table changes, ensuring it runs more frequently on active tables.

Mastering `postgresql.conf`

While there are many parameters, a select few provide the majority of the performance impact. The following table and subsequent discussion focus on these critical settings.

Connection management and pooling

`max_connections`: This parameter sets the hard limit on the number of concurrent client connections the server will accept. Each connection spawns a dedicated backend process, which consumes both private and shared memory. It is a common mistake to set this value excessively high. A better practice is to determine the actual number of connections required by the application and set `max_connections` to a value slightly above that. As a starting point, you can use `GREATEST(4 * CPU cores, 100)`. For smaller systems (e.g., four cores), this may be more than necessary. In such cases, observe the actual concurrency and set `max_connections` to about 20%–30% above the observed concurrency. For applications that have high connection churn or require hundreds or thousands of connections, an external connection pooler such as PgBouncer is an architectural necessity. A pooler maintains a small, persistent set of connections to the database and serves thousands of short-lived client connections from this pool, drastically reducing the overhead on the database server.

Memory allocation: A delicate balance

- **`shared_buffers`:** This is the most important memory-related parameter, defining the size of EDB Postgres's dedicated data cache in shared memory. For a dedicated database server, a common starting point is `25% of the total system RAM`. A more nuanced approach considers system memory variance; for example, on systems with more than 64GB of RAM, a value of `GREATEST(16 GB, RAM / 6)` can be used, with an upper cap of 64GB, as there are diminishing returns beyond this point. On systems with very large amounts of RAM (e.g., >256 GB), the benefit of an extremely large `shared_buffers` diminishes, and it may be more effective to cap it around 64 GB, leaving the remaining memory for the operating system's file cache.
- **`effective_cache_size`:** This parameter does not allocate memory. Instead, it serves as a crucial hint to the query planner, estimating the total amount of memory available for caching data, including both EDB Postgres's shared_buffers and the OS file system cache. A higher value makes the planner more optimistic about the likelihood of finding data in a cache, thus lowering the estimated cost of index scans. For a dedicated server, this should be set to a high value, typically 50% to 75% of total system RAM. A good practice is to set it to the sum of `shared_buffers` and the available Linux buffer cache.
- **`work_mem`:** This parameter specifies the amount of memory that can be used by internal sort operations and hash tables before writing to temporary disk files. It is allocated per operation, and a single complex query can have multiple sort or hash operations running in parallel. Therefore, the total memory consumed can be many times the `work_mem` value. While increasing `work_mem` can dramatically improve the performance of complex analytical queries by allowing sorts and joins to happen entirely in memory, setting it too high on a system with many concurrent queries is a common cause of memory exhaustion. A reasonable starting formula is `((Total RAM - shared_buffers) / (16 * CPU cores))`.
- **`maintenance_work_mem`:** This sets a separate, larger memory budget for maintenance operations such as `VACUUM`, `CREATE INDEX`, `REINDEX`, and `ALTER TABLE ADD FOREIGN KEY`. These operations can benefit significantly from more memory. Setting this to a generous value, such as 1 GB to 4 GB or roughly `15% x (Total RAM - shared_buffers) / autovacuum_max_workers` (up to 1GB), can drastically reduce the time required for these essential tasks.
- **`effective_io_concurrency`:** For systems using solid-state disks (SSDs), this parameter should be set to a high value, such as 200, to inform the planner about the storage's ability to handle concurrent I/O operations.

WAL tuning

The WAL is central to EDB Postgres's durability and recovery. A checkpoint is a point in the WAL sequence in which all data files have been updated to reflect the information in the log; this is an I/O-intensive operation. The goal of WAL tuning is to smooth out these I/O spikes and ensure that checkpoints are driven by time rather than by the volume of WAL generated.

- **Key parameters:**
  - **`max_wal_size`:** This parameter defines a soft limit on the total size of WAL files. When this limit is approached, a checkpoint is triggered. For write-heavy systems, the default value is far too low and will cause frequent, performance-degrading checkpoints. This should be increased significantly to a value such as 16 GB, 32 GB, or even higher, depending on available disk space and write volume. This allows checkpoints to be primarily triggered by `checkpoint_timeout`. If disk space is constrained, a safe starting point is 50–75% of the partition size dedicated to WAL.

- **`checkpoint_timeout`:** This is the maximum time between automatic checkpoints. A value between 10 and 15 minutes is a common and effective starting point for most systems.
- **`checkpoint_completion_target`:** This parameter controls the duration over which a checkpoint's I/O is spread, as a fraction of the checkpoint_timeout. Setting this to the modern default and best practice of 0.9 ensures that the writes are spread out over nearly the entire interval between checkpoints, effectively eliminating I/O spikes and creating a smoother performance profile.

- **`wal_buffers`:** This defines the amount of shared memory used for buffering WAL data before it is written to disk. A larger value can improve performance on systems with many concurrent transactions. The default of -1 allows EDB Postgres to auto-size it based on `shared_buffers` (up to a limit of 16MB), which is often sufficient. A manual setting of 64MB is also a safe, high-performance choice.
- **`wal_compression`:** When enabled, this parameter compresses full-page images written to the WAL during the first modification of a page after a checkpoint. This reduces the I/O volume of the WAL at the cost of some CPU, which is almost always a beneficial trade-off. It should be set to **on**.
- **`wal_log_hints`:** This parameter should be set to on to enable tools such as `pg_rewind`, which is crucial for certain high-availability and recovery scenarios.

Query planner cost parameters

The query planner makes decisions by estimating the "cost" of various execution paths. These cost parameters are the variables in its calculations.

- **`random_page_cost` and `seq_page_cost`:** These parameters model the planner's estimate of the cost of fetching a single database page from disk. `seq_page_cost` defaults to 1.0; `random_page_cost` defaults to 4.0, which reflects the reality of slow-spinning disks and the high expense of a random seek compared to a sequential read. On modern SSD or NVMe storage, this assumption is false; random I/O is nearly as fast as sequential. Therefore, `random_page_cost` should be lowered to a value close to 1.0, such as 1.1. This simple change is one of the most effective ways to encourage the planner to correctly choose index scans over sequential scans on fast storage.
- **`cpu_tuple_cost`:** This parameter specifies the planner's estimate of the cost of processing each row. The default value is likely too low for modern systems and should be increased to a more realistic value, say 0.03.

Autovacuum: The unsung hero of performance

EDB Postgres's multiversion concurrency control (MVCC) architecture is a key feature that allows readers not to block writers and writers not to block readers. It achieves this by not modifying data in place. Instead, an `UPDATE` or `DELETE` operation creates a new version of the row and marks the old version as no longer visible to new transactions. These obsolete row versions, known as "dead tuples," remain in the table's data files.

The autovacuum process is responsible for two critical maintenance tasks:

- **Reclaiming space:** It scans tables to find and remove dead tuples, making the space they occupy available for reuse. If this is not done, tables become bloated with dead rows, leading to wasted disk space and slower sequential scans.
- **Preventing transaction ID wraparound:** It updates metadata in table headers to freeze very old transaction IDs, preventing a catastrophic failure condition in which the 32-bit transaction counter wraps around, causing past data to appear in the future.

The default autovacuum configuration is designed to be nonintrusive and is often insufficient for active production databases. Aggressive tuning is a necessity for maintaining performance at scale.

- **Tuning for responsiveness:** The goal is to make autovacuum run more frequently and more quickly.
  - **Triggering:** Autovacuum is triggered on a table when the number of dead tuples exceeds `(autovacuum_vacuum_scale_factor * table_size) + autovacuum_vacuum_threshold`. The default scale factor of 0.2 (20%) is far too high for large, active tables. Lowering `autovacuum_vacuum_scale_factor` to a value of about 0.05 or 0.1 and adjusting `autovacuum_vacuum_threshold` will ensure that vacuuming happens more proactively.
  - **Parallelism:** Increasing `autovacuum_max_workers` (e.g., to 5) enables the system to vacuum more tables concurrently, which is crucial for databases with numerous active tables.
  - **Throttling:** Autovacuum is designed to throttle itself to minimize its impact on the foreground workload, controlled by `autovacuum_vacuum_cost_delay` and `autovacuum_vacuum_cost_limit`. On modern hardware, these limits can often be made more aggressive (e.g., increasing `autovacuum_vacuum_cost_limit` to 3000 or higher, with some recommendations as high as 5000 or more, depending on `autovacuum_max_workers`) to allow vacuuming to complete more quickly, especially during known periods of low database activity.
  - **Logging:** To help tune autovacuum, it is useful to monitor its activity by setting `log_autovacuum_min_duration` to 0, which logs all actions it performs.
- **Per-table tuning:** A one-size-fits-all global configuration is rarely optimal. For very large or very high-transaction tables, it is a best practice to set autovacuum parameters on a per-table basis using the `ALTER TABLE... SET (...)` syntax. This allows an administrator to configure a very large, static table to be vacuumed infrequently, while a smaller, high-update "hot" table can be configured to be vacuumed aggressively.

Query and application performance optimization

While infrastructure and server configuration create the potential for high performance, it is the efficiency of the SQL queries themselves that ultimately determines the application's responsiveness and the system's ability to scale. Query optimization is an iterative process of identifying performance bottlenecks, analyzing their root causes through execution plans, and implementing structural or logical improvements.

Identifying and analyzing performance bottlenecks

The first step in optimization is measurement. A robust strategy for identifying slow and resource-intensive queries combines proactive logging, real-time monitoring, and aggregate statistical analysis.

Proactive logging

Logging is the most direct method for capturing problematic queries as they occur in a production environment.

- **`log_min_duration_statement`:** This is the single most effective parameter for performance analysis. When set to a value in milliseconds (e.g., `200ms or 500ms`), EDB Postgres will log any statement that exceeds this execution time, including its duration and parameters, in the server log. This log becomes a direct, prioritized list of the slowest queries in the system that require investigation. For transactional workloads, a query running longer than 250ms might be considered slow, while a starting point of 1 second (1000ms) is reasonable for general use.
- **`auto_explain`:** This powerful extension, when loaded via `shared_preload_libraries`, can be configured to automatically log the `EXPLAIN` plan of any query that exceeds a specified duration. This is invaluable because it captures the exact execution plan that caused the slowness, saving the DBA the often difficult task of reproducing the issue manually.
- **Enhanced logging configuration**: For comprehensive analysis, several other logging parameters should be enabled:
  - **`logging_collector = on`:** Enables the capture of logs.
  - **`log_directory`:** Should be set to a location outside of the main data directory.

- **`log_checkpoints = on`:** Logs checkpoint activity, which is useful for I/O tuning.
- **`log_lock_waits = on`:** Logs waits for locks, which is essential for diagnosing contention issues.
- **`log_statement = 'ddl'`:** Logs all Data Definition Language commands, providing a basic audit trail.
- **`log_temp_files = 0`:** Logs all temporary files created, which can indicate that work_mem is tuned too low.
- **`log_connections = on and log_disconnections = on`:** Log all connection and disconnection events for security and connection pooler analysis.
- **`log_line_prefix`:** A detailed prefix such as `'%m [%p]: u=[%u] db=[%d] app=[%a] c=[%h] s=[%c:%l] tx=[%v:%x]'` provides rich context for each log entry.

Real-time analysis

For diagnosing issues as they are happening, EDB Postgres provides several dynamic views and functions:

- **`pg_stat_activity`:** This view is the primary tool for observing the current state of the database. It displays one row for every backend process, showing crucial information such as the user, client address, the current state (**`active, idle in transaction`**), the **`query_start`** time, and the full text of the currently executing query. Administrators can use this view to write queries that identify:
  - **Long-running queries:** By filtering on `state = 'active'` and a `query_start` time older than a certain threshold.
  - **Idle in transaction sessions:** These are dangerous sessions that hold locks while doing no work. They can be found by filtering for `state = 'idle in transaction'`. Setting `idle_in_transaction_session_timeout` (e.g., to 10 minutes) can automatically terminate these problematic sessions.
  - **Blocking locks:** The `wait_event_type` and `wait_event` columns will indicate whether a process is blocked while waiting for a lock.
- **`pg_locks and pg_blocking_pids()`:** For a deeper analysis of locking contention, the `pg_locks` view shows all locks currently held in the system. The `pg_blocking_pids()` function provides a convenient way to find the process IDs that are blocking a specific blocked process, allowing for a quick resolution of lock chains.

Aggregate analysis with `pg_stat_statements`

While logging captures individual slow queries, it may miss queries that are fast individually but are executed so frequently that they consume the majority of the system's resources. The pg_stat_statements extension is essential for finding these.

- **Functionality:** When enabled via **`shared_preload_libraries, pg_stat_statements`** tracks execution statistics for every normalized query executed on the system. It aggregates data such as calls (execution count), **`total_exec_time`**, rows returned, and block **`I/O (shared_blks_hit`**, **`shared_blks_read)`**.
- **Analysis:** By querying the `pg_stat_statements` view and ordering by `total_exec_time`, an administrator can quickly identify the top queries that are responsible for the most database workload over time. These are often the most important candidates for optimization, even if their average execution time is low.

Deconstructing execution plans with `EXPLAIN`

The `EXPLAIN` command is the DBA's most powerful diagnostic tool. It reveals the execution plan chosen by the EDB Postgres query planner, showing how it intends to access tables and join data to satisfy a query.

- **The power of `ANALYZE`:** Using `EXPLAIN` by itself only shows the planner's estimated costs and row counts. To get a true picture of performance, the `EXPLAIN (ANALYZE)` command must be used. This command actually executes the query and then displays the plan annotated with the actual execution times and row counts for each step. The discrepancy between the planner's estimates and the actual results is the primary clue for diagnosing performance problems.

- **Reading the plan:** An execution plan is a tree of nodes, read from the inside out. Each node represents an operation (e.g., a table scan, a join, a sort). Key information for each node includes:
  - **`Cost`:** Two numbers, `cost=startup..total`. The startup cost is the estimated cost to return the first row, and the total cost is for all rows.
  - **`Rows`:** The estimated number of rows this node will output.
  - **`Width`:** The estimated average width in bytes of the rows output.
  - **`Actual time`:** Provided by `ANALYZE`, this shows the actual startup and total time in milliseconds.
  - **`Actual rows`:** Provided by `ANALYZE`, this shows the actual number of rows output.
- **Identifying anti-patterns:** When analyzing an execution plan, certain patterns are red flags for performance issues:
  - **Sequential scans on large tables:** This indicates that the planner is reading the entire table from disk. It is often a sign of a missing index or a `WHERE` clause that prevents an existing index from being used (e.g., by applying a function to an indexed column).
  - **Large estimation errors:** A significant mismatch between estimated rows and actual rows points to stale statistics (requiring an `ANALYZE` on the table) or a complex query predicate that the planner cannot accurately model.
  - **External sorts:** The plan will show "`Sort Method: external merge on disk`". This means the data to be sorted did not fit into the memory allocated by `work_mem` and had to be spilled to slow temporary disk files. This is a clear indicator that `work_mem` should be increased for this query.
  - **Heap fetches/lossy bitmap scans:** A "`Bitmap Heap Scan`" plan might show a high number of "`Heap Fetches`". This means the index identified many potentially matching blocks, but then many rows within those blocks had to be rechecked against the query conditions. If the number of heap fetches is high, the index may not be very selective for the query.
- **Deeper insights with additional options:**
  - **`BUFFERS`:** The `EXPLAIN (ANALYZE, BUFFERS)` option is invaluable. It adds a line to each node showing the number of 8KB data blocks that were read from disk (read) versus found in EDB Postgres's cache (hit). This provides a direct measure of the query's I/O impact.
  - **`TIMING`:** This option, which is on by default with `ANALYZE`, provides the per-node timing details, making it easy to see exactly which step in a multistage plan is consuming the most time.
  - **`WAL`:** For DML statements (`INSERT`, `UPDATE`, `DELETE`), this option shows the volume of WAL records generated, which is useful for identifying and optimizing unexpectedly "noisy" write operations.

Example and explanation:

```
EXPLAIN (ANALYZE, BUFFERS, VERBOSE, WAL)

SELECT account_id, SUM(amount)

FROM transactions

   WHERE txn_type = 'CREDIT'

   GROUP BY account_id;

                                QUERY PLAN

------------------------------------------------------------------------------

HashAggregate

   (cost=131.42..134.67 rows=100 width=41)
```

```
   (actual time=0.418..0.434 rows=100 loops=1)
   Output: account_id, sum(amount)
   Group Key: transactions.account_id
   Batches: 1  Memory Usage: 80kB
   Buffers: shared hit=87
   ->  Seq Scan on public.transactions
         (cost=0.00..128.15 rows=655 width=14)
         (actual time=0.073..0.255 rows=655 loops=1)
         Output: account_id, id, amount, currency, txn_type, channel, merchant_category,
                 counterparty_acct, counterparty_bank, counterparty_city,
                 account_city, device_id, ip_address, round_amount_flag, txn_hour, is_
round_amount
         Filter: (transactions.txn_type = 'CREDIT'::text)
         Rows Removed by Filter: 611
         Buffers: shared hit=87
Planning Time: 0.122 ms
Execution Time: 0.482 ms
(13 rows)
```

Options used here:

- **ANALYZE:** Executes the query to provide real execution stats
- **BUFFERS:** Shows cache vs. disk usage
- **VERBOSE:** Includes column-level details
- **WAL:** Shows WAL writes (useful in write-heavy queries; here it's SELECT-only, so none generated)

Plan overview

- **GroupAggregate:** Performs the `GROUP BY account_id` and aggregates `SUM`(amount)
- **Sort:** Orders rows by account_id before grouping
- **Seq Scan:** Reads all rows from transactions

Key observations in this plan

1. Sequential scan

- `Seq Scan` on `public.transactions`
- Filter: (`txn_type = 'CREDIT'`) applied row by row
- Rows removed by filter: 611 `(out of ~655 scanned)`
- `Buffers`: Shared hit=87 → all data served from cache, no disk reads
- Good for small tables; on large datasets, inefficient → recommend an index on (txn_type, account_id)

2. Sort node

- Method: `quicksort`; memory: 80kB → in-memory, no spill to disk
- No immediate issue
- On larger data, tuning `work_mem` or indexing `account_id` would avoid costly sorts

3. Group aggregate

- Groups rows by `account_id`, computes `SUM(amount)`
- Execution time very fast (0.73 ms)
- Works fine here, but on high-volume workloads, consider partitioning or parallel aggregates

4. Performance notes

- Planning time: 0.122 ms
- Execution time: 0.482 ms
- Buffers: All cache hits
- Extremely efficient for this small dataset, but will not scale well without indexes

Takeaway

The plan is fine for a small test set. But in production with millions of rows:

- Add an index on `(txn_type, account_id)` to avoid `Seq Scan + Sort`.
- Monitor `work_mem` for larger sorts.
- For write-heavy workloads, keep an eye on WAL output (not applicable here).

Advanced indexing strategies

Indexes are the primary mechanism for improving query performance. They allow the database to find specific rows without scanning the entire table. A well-designed indexing strategy is critical for any large-scale application.

- **Choosing the right index type:** EDB Postgres offers several index types, each suited for different data types and query patterns.
  - **B-tree:** The default and most common index type. It is highly efficient for equality (=) and range (`<`, `>`, `BETWEEN`) queries on standard data types such as integers, text, and timestamps. It also supports sorted output, which can eliminate a separate sort step for `ORDER BY` clauses.
  - **GIN (Generalized Inverted Index):** Designed for indexing composite values whose items can appear multiple times within a single column. Its primary use cases are for accelerating full-text search queries and for indexing keys or elements within `JSONB` documents and arrays.
  - **GiST (Generalized Search Tree):** A flexible framework for building indexes over complex data types. It is used for indexing geometric data (for spatial queries) and for implementing exclusion constraints.
  - **BRIN (Block Range Index):** A specialized index for very large tables in which the data has a strong physical correlation with the indexed values (e.g., a timestamp column in a time-series table). BRIN indexes are extremely small because they only store the minimum and maximum value for a large range of table blocks. They are less precise than B-trees but can provide significant performance gains for analytical queries on large, ordered datasets.
- **Advanced indexing techniques:** Beyond choosing the right type, several advanced techniques can be used to create highly efficient indexes.

- **Partial indexes:** A partial index is built on a subset of a table's rows, defined by a `WHERE` clause in the `CREATE INDEX` statement. For example, `CREATE INDEX ON orders (order_id) WHERE status = 'shipped'`. This is incredibly powerful for reducing index size and maintenance overhead. The index will be much smaller and faster to update because it only includes rows that meet the condition.
- **Covering indexes (`INCLUDE clause`):** Normally, after finding an entry in an index, EDB Postgres must perform a heap fetch to retrieve the rest of the row's data from the main table. An index-only scan avoids this by satisfying the query entirely from the index. This can be achieved by creating a covering index using the `INCLUDE` clause to add columns that are needed by the `SELECT` list but are not part of the search key.
- **Expression indexes:** If queries frequently filter on the result of a function or expression, an index can be created on that expression. A classic example is case-insensitive search: `CREATE INDEX ON users (lower(email))`. This allows a query with `WHERE lower(email) = '...'` to use the index directly, avoiding a sequential scan.

Data partitioning for large tables

As tables grow into the hundreds of gigabytes or terabytes, managing them becomes challenging. Operations such as indexing, vacuuming, and even simple queries can become slow. Native declarative partitioning, available in modern EDB Postgres versions, is the solution for managing very large tables.

- **Partitioning methods:** A table can be declared as `PARTITION BY` via one of three methods:
  - **`RANGE`:** The partition key (often a timestamp or an ID) is divided into non-overlapping ranges, with each range assigned to a partition. This is the most common method for time-series data.
  - **`LIST`:** Each partition is explicitly defined to hold rows with specific values in the partition key (e.g., a `country_code` column).
  - **`HASH`:** Rows are distributed among a specified number of partitions based on the hash of the partition key. This is useful for evenly distributing data when there is no natural range or list to partition by.

  In addition:

    - **Autopartition Extension, pg_partman:** Available for PostgreSQL, it helps DBAs automate the creation of partitions as data arrives, reducing manual intervention.
    - **Interval partitioning (EDB Postgres Advanced):** An EDB-specific feature that allows automatic creation of new partitions at defined intervals (e.g., monthly or daily) based on the partition key, making it especially useful for time-series workloads.
- **Performance benefits:**
  - **Partition pruning:** This is the primary performance benefit. When a query's `WHERE` clause contains a condition on the partition key, the query planner can intelligently exclude (or "prune") any partitions that cannot possibly contain matching data. For a time-series table partitioned by month, a query for a single day's data will only scan one partition, instead of the entire multi-terabyte table. Additionally, creating appropriate indexes on each partition can further improve the efficiency of partition pruning.
  - **Improved maintenance:** Maintenance operations can be performed on a per-partition basis. An old, historical partition can be backed up and then detached from the partitioned table, which is an instantaneous metadata-only operation, providing an extremely efficient way to archive data. Similarly, `VACUUM` or `REINDEX` can target a single active partition instead of the entire table set.

Architecting for scalability and high availability

When a single server can no longer meet the performance or availability demands of an application, it becomes necessary to scale the database architecture. The choice of scaling strategy is one of the most consequential decisions an architect will make, with long-term implications for performance, cost, operational complexity, and resilience. This is fundamentally a business decision disguised as a technical one. The "best" architecture is the one that aligns with the business's tolerance for downtime (recovery time objective, or RTO), data loss (recovery point objective, or RPO), and operational overhead. An architect's role is to translate business requirements such as "the application must always be available" into a concrete technical solution, such as a specific replication topology with a defined consistency guarantee.

Vertical vs. horizontal scaling: A strategic comparison

There are two fundamental approaches to scaling a database system.

- **Vertical scaling (scaling up):** This involves increasing the resources of a single server—adding more CPU cores, more RAM, or faster storage. It is the simplest approach to implement, because it requires no changes to the application architecture. However, it has inherent limitations: There is a physical ceiling to how powerful a single machine can be, costs can escalate nonlinearly, and it represents a single point of failure. An outage of the single server results in a complete application outage.
- **Horizontal scaling (scaling out):** This involves distributing the workload across multiple servers. This approach offers nearly limitless scalability and can provide high availability by eliminating single points of failure. However, it introduces significant architectural complexity related to data distribution, consistency, and connection management.

The following table provides a strategic comparison to guide the decision-making process:

Strategy	Primary Use Case	Read Scalability	Write Scalability	High Availability (RPO/RTO)	Complexity
Vertical Scaling	Initial growth, simplicity	None	Limited by single node	None (single point of failure)	Low
Streaming Replication	Read offloading, simple high availability	High (add more replicas)	None (single primary)	High (RPO=0 possible, RTO=seconds)	Medium
Logical Replication	Selective replication, zero-downtime upgrades	Medium (replicas are writable)	Limited (single primary model)	Manual/ complex failover	High
EDB Postgres Distributed (PGD)	Geo-distribution, active-active writes, max high availability	Very high	Very high (lead/shadow primary)	Very high (RPO=0, RTO < 5s)	Very high

Read scalability with streaming replication

The most common and straightforward method for horizontal scaling in EDB Postgres is streaming replication. It is primarily used to scale read-intensive workloads and to provide a foundation for high availability.

- **Architecture:** In this model, there is a single primary (read/write) server and one or more standby (read-only) servers. The primary server continuously streams its WAL records over the network to the standbys. The standbys receive these records and apply them, keeping them up to date as near-exact copies of the primary. Applications can then be configured to direct all write traffic to the primary and offload read queries (e.g., from reporting dashboards or analytical tools) to the standby servers, thus distributing the read load.

- **Synchronous vs. asynchronous replication:** The durability and performance trade-off in streaming replication is controlled by its synchronicity.
  - **Asynchronous (default):** When a transaction is committed on the primary, it acknowledges success to the client without waiting for the standby(s) to receive the change. This offers the highest write performance on the primary, as it is not delayed by network latency. However, it introduces a small window for potential data loss. If the primary server fails catastrophically before a recent transaction has been streamed to a standby, that transaction will be lost (RPO > 0).
  - **Synchronous:** The primary server can be configured to wait for confirmation that a transaction has been successfully written and flushed to disk on at least one standby server before it acknowledges the commit to the client. This configuration guarantees zero data loss in the event of a primary failure (RPO = 0). The cost of this guarantee is increased write latency on the primary, as every `COMMIT` must wait for a network round trip to the standby.

Advanced architectures with logical replication

Logical replication operates at a higher level of abstraction than streaming replication. Instead of replicating physical block changes from the WAL, it decodes the WAL into a stream of logical changes—row-level `INSERT`, `UPDATE`, and `DELETE` events—and sends these to a subscriber. This approach offers significantly more flexibility.

- **Key use cases:**
  - **Selective replication:** An administrator can choose to replicate only a specific subset of tables, or even filter rows within those tables. This is useful for creating specialized read replicas or for sharing specific datasets with other systems.
  - **Zero-downtime major version upgrades:** Because logical replication works at the data level, it is possible to replicate from an EDB Postgres 15 primary to an EDB Postgres 16 subscriber. This allows for a seamless major version upgrade: Set up replication, wait for the new version to catch up, and then switch application traffic over to the new server with minimal downtime.
  - **Data integration and consolidation:** Data from multiple different source databases can be logically replicated and consolidated into a single central data warehouse or analytical database.
- **Operational considerations:** The flexibility of logical replication comes with increased operational complexity.
  - **DDL management:** Schema changes (e.g., `ALTER TABLE`) are not automatically replicated. They must be applied manually and carefully on both the publisher and subscriber to keep them in sync.
  - **Sequences and large objects:** Sequences are not replicated, which can lead to primary key collisions if not managed carefully. Large objects are also not replicated.
  - **Initial data copy:** The initial data synchronization must be handled as a separate step before replication can begin.

Active-active and geo-distribution with EDB Postgres Distributed

For applications requiring the absolute highest levels of availability and performance across geographically distributed locations, a multi-active replication solution is required. EDB Postgres Distributed (PGD) is an advanced product that provides this capability on top of EDB Postgres.

- **Architecture:** PGD creates a distributed cluster of EDB Postgres nodes in which every node is writable (active-active). It uses a mesh topology: Changes made on any node are logically replicated to all other nodes in the cluster. This architecture is designed to achieve up to five-nines (99.999%) availability and to support globally distributed applications.

- **Key features:**
  - **Active-active writes:** Applications can connect to and perform writes on any node in the cluster. This is ideal for geo-distributed applications, in which users can be directed to the geographically closest node to minimize latency for both reads and writes.
  - **Advanced conflict management:** In a multi-active system, it is possible for two transactions on different nodes to modify the same row concurrently, creating a conflict. PGD provides sophisticated, automatic conflict detection and configurable resolution mechanisms (e.g., "last update wins") to handle these situations and maintain data consistency.
  - **Commit scopes and durability:** PGD offers granular control over the durability and consistency guarantees for each transaction. An administrator can configure different "commit scopes" to balance performance and consistency. Options range from fully asynchronous commits for maximum speed to globally synchronous commits that ensure a transaction is durable on all nodes in the cluster before returning success, providing the strongest consistency guarantees at the cost of higher latency. This allows an architect to precisely match the database's behavior to the application's business requirements.

Version-specific features and enterprise tooling

The EDB Postgres ecosystem is characterized by rapid innovation, with a new major version of the community database released annually and enterprise vendors such as EDB building powerful proprietary features on top. Understanding the key performance and scalability features introduced in recent and upcoming versions is crucial for making informed decisions about upgrades and technology adoption.

Key Performance/ Scalability Feature	Description and Impact
Resource Manager, SQL Profiler	Provides Oracle-like capabilities for workload isolation and deep query analysis
Global indexes, PWR and edb_wait_states	Introduces global indexes for partitioned tables (critical for Oracle migrations) and modernizes performance diagnostics to an AWR-like model
Asynchronous I/O (AIO) (new in PG 18)	Foundational change to I/O subsystem, hiding latency and dramatically improving read throughput on modern/cloud storage
B-tree skip scans (new in PG 18)	Makes multicolumn indexes significantly more versatile, allowing them to be used even when the leading column is not in the `WHERE` clause
Planner statistics preservation (new in PG 18)	Eliminates the dangerous post-upgrade performance dip by carrying over statistics, making major version upgrades safer and less disruptive

EDB Postgres Advanced Server enhancements

EDB Postgres Advanced Server (EPAS) extends community EDB Postgres with features designed for enterprise-grade security, performance, and manageability, including significant compatibility enhancements for organizations migrating from Oracle.

Enterprise performance and management tooling

EPAS includes a suite of powerful tools for performance analysis and tuning, which are often integrated into the Postgres Enterprise Manager (PEM) graphical console.

- **EDB Index Advisor:** This utility analyzes a given SQL workload and provides recommendations for creating new B-tree indexes to improve query performance. It works by creating hypothetical indexes and using the

query planner to evaluate their potential benefit without the overhead of actually building them. Index Advisor can be used from the command line with a file of SQL queries or interactively through PEM, where it can analyze traces captured by the SQL Profiler. This capability is part of EDB Query Advisor.

- **EDB SQL Profiler:** A tool designed to locate and help optimize inefficient SQL code, which is a primary cause of database performance issues. It allows DBAs to create "traces" to capture SQL workloads, either on-demand or on a schedule. The captured traces can then be analyzed, filtered, and sorted to identify the most resource-intensive queries. It provides both graphical and text-based `EXPLAIN` plans and integrates with the Index Advisor to provide a complete query tuning workflow.
- **EDB Resource Manager:** A feature analogous to Oracle's Resource Manager, it provides the ability to control operating system resource consumption by database processes. DBAs can create "resource groups" and define limits on CPU usage (`cpu_rate_limit`) and I/O activity (`dirty_rate_limit`). By assigning roles or specific database sessions to these groups, administrators can prevent a single rogue query or batch job from consuming all system resources and negatively impacting other, more critical processes.

The evolution of performance diagnostics

EPAS has undergone a significant evolution in its tools for deep performance diagnostics, moving toward a more modern, wait event–based analysis model.

- **DRITA (Dynamic Runtime Instrumentation Tools Architecture):** For many versions, DRITA was the primary tool in EPAS for performance diagnostics. It allowed DBAs to take "snapshots" of system performance data at different points in time and then generate reports comparing them, similar in concept to Oracle Statspack or AWR reports. The reports focused on wait events to help diagnose system bottlenecks. To enable data collection for DRITA, the `timed_statistics` parameter must be set to on.
- **Deprecation in EPAS 17:** With the release of EPAS 17, DRITA has been officially deprecated and is scheduled for removal in EPAS 18. This strategic decision was made because of the introduction of a superior, more capable set of tools.
- **The new standard:** PWR and `edb_wait_states` are the modern replacement for DRITA. They are the combination of the edb_wait_states extension and the Postgres Workload Report (PWR) tool.
  - The `edb_wait_states` extension is a background worker that samples session activity at regular intervals, capturing detailed information on wait events, running queries, and CPU usage.
  - The `PWR` tool is a Python-based utility that processes the data collected by `edb_wait_states` to generate comprehensive performance reports that are explicitly designed to mimic Oracle's AWR reports. This combination provides a much more powerful and familiar diagnostic experience for DBAs, offering deeper insights into performance bottlenecks and making it the new standard for performance analysis in EPAS.

The performance revolution in EDB Postgres 18

EDB Postgres 18, released in September 2025, introduces several foundational features that represent a significant leap forward in performance, scalability, and operational manageability.

Deep dive: Asynchronous I/O (AIO)

The most impactful feature in EDB Postgres 18 is the introduction of a native asynchronous I/O subsystem.

- **The problem with synchronous I/O:** In previous versions, EDB Postgres relied on synchronous I/O. When a backend process needed to read a data block that was not in the cache, it would issue a read request to the operating system and then block, waiting idly until the I/O operation completed. In modern environments, especially in the cloud with network-attached storage, I/O latency can be significant. This synchronous model resulted in wasted CPU cycles and limited throughput, as the process could not perform other work while waiting.

- **The AIO solution:** The new AIO subsystem allows a backend process to issue multiple I/O requests concurrently without waiting for each one to complete. It can continue with other processing tasks while the I/O happens in the background, effectively hiding I/O latency. Benchmarking has shown this can lead to performance gains of up to 3x for read-heavy operations such as sequential scans, bitmap heap scans, and `VACUUM`.
- **Implementations (`io_method`):** The AIO subsystem can be configured via the `io_method` parameter with two primary modes:
  - **`worker` (Default):** This is a cross-platform implementation that uses a pool of dedicated background I/O worker processes to execute the I/O requests asynchronously. The number of workers is configurable via `io_workers`.
  - **`io_uring` (Linux only):** For systems running a modern Linux kernel (5.1+), this mode leverages the high-performance io_uring kernel interface. It offers lower overhead and typically better performance than the worker model by using a shared ring buffer between the application and the kernel to submit I/O requests with minimal system calls.

Deep dive: B-tree skip scans

EDB Postgres 18 addresses a long-standing limitation of multicolumn B-tree indexes, making them significantly more powerful and versatile.

- **The "left-most prefix" problem:** Historically, a multicolumn B-tree index, for example on (`customer_id, order_date, status`), could only be used efficiently by the query planner if the `WHERE` clause included a constraint on the leading column (`customer_id`). A query such as `WHERE order_date > '...' AND status = 'pending'` could not use the index effectively because it lacked a constraint on the first column, often forcing the planner to resort to a slow sequential scan.
- **The skip-scan solution:** The new "`skip scan`" optimization allows the planner to intelligently use such an index even without a predicate on the leading column(s). It works by first identifying the distinct values of the leading column(s) and then performing a series of small, targeted index probes for each of those distinct values. For a query on `order_date` and `status`, it would effectively "skip" from one `customer_id` to the next within the index, performing a small search within each customer's block of index entries. This is particularly effective when the cardinality of the skipped leading column is low (i.e., there are relatively few distinct values). This feature can eliminate the need to create many single-purpose indexes, simplifying schema management and reducing storage overhead.

Operational excellence: Preserving planner statistics

Major version upgrades have historically been a source of significant operational risk and anxiety for DBAs of large EDB Postgres databases.

- **The post-upgrade performance dip:** The `pg_upgrade` utility, while efficient at migrating data, did not previously carry over the vital optimizer statistics from the old database cluster. These statistics, which are collected by the `ANALYZE` command, are critical for the query planner to make intelligent decisions. Without them, the planner reverts to default estimates, often resulting in terrible query plans and a severe degradation of application performance immediately following an upgrade. The only remedy was to run a time-consuming `ANALYZE` on the entire database, during which the system would perform poorly.
- **The EDB Postgres 18 solution:** `pg_upgrade` in version 18 now has the capability to transfer the planner statistics from the old cluster to the new one. This is a monumental improvement for operational stability. It means that an upgraded cluster can achieve its expected high-performance state almost immediately after the upgrade is complete, eliminating the post-upgrade performance dip. This dramatically reduces the risk and stress associated with major version upgrades, making it easier for organizations to stay current with the latest EDB Postgres features and security patches.

Conclusion

Achieving and maintaining high performance for EDB Postgres at scale is a multifaceted discipline that demands a holistic, full-stack approach. This report has detailed a comprehensive framework that addresses every layer of the technology stack, from the foundational hardware and operating system to the intricacies of database configuration, query optimization, and scalable architectural design.

The core principles for success can be synthesized into several key takeaways:

1. **The foundation is paramount:** Performance begins with a well-balanced and properly sized infrastructure. CPU, RAM, and storage must be provisioned as a synergistic system in which no single component becomes a bottleneck. OS tuning is not a matter of minor tweaks but a critical process of aligning the kernel's behavior with the specific demands of a database workload, primarily by removing layers of abstraction and allowing the database and hardware to operate more directly.
2. **Configuration is communication:** The `postgresql.conf` file should be viewed not as a set of commands but as a descriptive model of the operating environment that is communicated to the query planner. Accuracy in parameters such as `effective_cache_size` and `random_page_cost` is essential for enabling the planner to make intelligent, cost-based decisions that leverage the full power of the underlying hardware.
3. **Maintenance is not optional:** The importance of aggressive and proactive autovacuum tuning cannot be overstated. In a transactional system of any significant scale, default settings are inadequate and will inevitably lead to performance degradation from table and index bloat. Autovacuum is the unsung hero that ensures the long-term health and stability of the database.
4. **Optimization is an iterative cycle:** Query performance is not a one-time task but a continuous cycle of measurement, analysis, and refinement. A robust strategy combines proactive logging (`log_min_duration_statement`), aggregate analysis (`pg_stat_statements`), and real-time monitoring (`pg_stat_activity`) to identify bottlenecks. Mastery of `EXPLAIN (ANALYZE, BUFFERS)` is the essential skill for deconstructing these bottlenecks and identifying their root causes.
5. **Architecture must align with business needs:** The choice of a scaling architecture—from simple vertical scaling to read replicas with streaming replication to the advanced lead/shadow primary capabilities of EDB Postgres Distributed—is ultimately a business decision. The correct architecture is the one that best meets the organization's specific requirements for availability (RTO), durability (RPO), latency, and operational capacity.

Finally, the EDB Postgres ecosystem continues to evolve at a rapid pace. The enterprise-grade tooling in EDB Postgres Advanced Server, such as the new PWR and `edb_wait_states` for deep performance diagnostics and Global Indexes in EPAS 17, provides powerful capabilities for mission-critical deployments. Simultaneously, the community's work in EDB Postgres 18, particularly the introduction of asynchronous I/O, B-tree skip scans, and the preservation of planner statistics, represents a fundamental leap forward in raw performance and operational stability. Staying abreast of these developments and strategically planning for upgrades is key to leveraging the full potential of EDB Postgres and ensuring that the data platform can meet the performance and scalability challenges of the future.

About the author

Vibhor Kumar
VP, CX Technical Advisor, EDB

Connect on LinkedIn

For two decades, Vibhor has partnered with Fortune 500 and Global 2000 companies, spearheading their digital transformation, cloud computing, AI/ML, and cybersecurity initiatives. As CX Technical Advisor for EDB, he drives technical innovation and strategic alignment for EDB's largest customers. He completed the CTO Executive Program at the Wharton School (University of Pennsylvania) in June 2025.

EDB POSTGRES AI

EDB Postgres AI is the first open, enterprise-grade sovereign data and AI platform, with a secure, compliant, and fully scalable environment, on premises and across clouds. Supported by a global partner network, EDB Postgres AI unifies transactional, analytical, and AI workloads, enabling organizations to operationalize their data and LLMs where, when, and how they need them. For more information, visit enterprisedb.com

© EnterpriseDB Corporation 2026. All rights reserved.