

Clustering Technologies Overview



Copyright

© Postgres Professional, 2017–2025

Authors: Egor Rogov, Pavel Luzanov, Ilya Bashtanov, Alexey Beresnev

Photo: Oleg Bartunov (Phu Monastery and Bhrikuti Peak, Nepal)

Use of Course Materials

Non-commercial use of course materials (presentations, demonstrations) is allowed without restrictions. Commercial use is possible only with the written permission of Postgres Professional. It is prohibited to make changes to the course materials.

Feedback

Please send your feedback, comments and suggestions to:

edu@postgrespro.ru

Disclaimer

Postgres Professional assumes no responsibility for any damages and losses, including loss of income, caused by direct or indirect, intentional or accidental use of course materials. Postgres Professional company specifically disclaims any warranties on course materials. Course materials are provided “as is,” and Postgres Professional company has no obligations to provide maintenance, support, updates, enhancements, or modifications.

Performance Expectations

Enablers

Building Blocks

System Examples

High system availability
and reliability

Scalability

Data consistency

Minimizing system downtime

caused by both failures and scheduled maintenance

Some metrics

system uptime ratio	→ 100 %
MTBF – Mean Time Between Failures	→ ∞
RTO – Recovery Time Objective	→ 0
RPO – Recovery Point Objective	→ 0
and others	

System availability is the ratio of time the system remains accessible to its total operational time.

Availability is a broader concept than reliability, as downtime can result not only from failures but also from scheduled maintenance.

Availability is often expressed in percentages:

- "one nine" 90% – 36.5 days of downtime per year,
- "two nines" 99% – 3.65 days of downtime per year,
- "three nines" 99.9% – 8.76 hours of downtime per year,
- "four nines" 99.99% – 52.56 minutes of downtime per year,
- "five nines" 99.999% – 5.26 minutes of downtime per year, and so on.

High availability means getting close to 100%.

For example, managed PostgreSQL database service in Yandex Cloud guarantees 99.99% availability for read nodes and 99.95% for write nodes. (<https://yandex.cloud/en/services/managed-postgresql>).

Some other availability metrics:

- Mean Time Between Failures (MTBF);
- Recovery Time Objective (RTO):
the time it takes for the system to recover;
- Recovery Point Objective (RPO) — the maximum amount of data that can be lost during a failure, and so on.

Scalability

The system's ability to handle higher load with proportionally more resources provided

ideal response time scalability —
response time should remain constant

ideal throughput scalability —
the number of processed requests should increase linearly

Speedup

the ratio of response time on a single node to response time in a cluster with N nodes

A key concept for clusters is *scalability*: it describes how various target metrics react as the number of nodes, data volume, and workload handled by the system increase.

The workload increases both as a result of more queries processed in the same amount of time and because of higher volumes of data overall. Resources in this context usually refer to the number of nodes.

Response time scalability reflects how response time (the time it takes to process a query and respond with a result) changes with increased load and more cluster nodes. This is paramount for OLTP systems. Ideally, response time should remain constant at scale.

Speedup is a derivative metric that demonstrates how response time changes with increased number of nodes, but at a constant load.

Throughput scalability measures how the number of processed requests per second changes as data volume and the number of nodes grow. This is the most important metric for OLAP systems. Ideally, throughput should scale linearly with the number of nodes added.

Perfect scalability (unfortunately unattainable in practice) would let you handle growth simply by adding more nodes to the cluster.

Local transactions

- virtually no guarantees when switching between nodes
- applications must handle consistency concerns

Global (distributed) transactions

- ACID consistency
- transaction atomicity is necessary

The stronger the data consistency guarantees a cluster provides, the more it behaves like a single unified system, making it easier to work with and requiring less special handling in application code. However, this comes at a cost of performance.

In a master-replica cluster with a single write node where transactions execute locally, consistency guarantees can be quite weak. For instance, one node may have already applied the transaction commit and made its results available to clients, while another node has not yet done so. In this scenario, a client may end up reading stale data from one node after having already read the updated value from another node.

Applications must be designed with these situations in mind and be able to handle this configuration.

For a cluster providing global (distributed) transactions, there is consistency in the traditional ACID sense: a global transaction must move the database from one consistent state to another (C), provided the transaction completes fully across all nodes (A) and is isolated from other concurrent transactions (I).

This requires ensuring transaction atomicity: all cluster nodes must take the same action — either commit or roll back the transaction. PostgreSQL offers a relatively straightforward two-phase commit protocol (2PC) for this purpose, though it is not entirely immune to faults.

High Availability

Data Synchronization between Nodes

Failure Types and Detection

Distributed Consensus Protocols

Scalability

Scheduled maintenance without service interruption

- using PostgreSQL stock tools
- temporary node removal from the cluster

Redundant components to eliminate single points of failure

- PostgreSQL servers (and other required systems)
- network hardware
- power supply

Failure detection and recovery management

8

Downtime can result not only from unexpected failures but also from scheduled maintenance that cannot be performed without interrupting service. Each new PostgreSQL release reduces the number of operations that require a server restart. Additionally, the cluster must be able to handle planned shutdown of a node with minimal impact on users.

One of the pillars of fault tolerance is redundancy across all system components: not only database servers but also network equipment, power supply systems, and other infrastructure.

But simply duplicating system components isn't enough.

First, you have to set up data synchronization between nodes.

Then, you need failure detection and handling – proper failover to standby nodes and recovery after resolving the failures.

Vanilla PostgreSQL provides limited, if any, native support for setting these up. Generally speaking, the higher the system's target availability, the more complex its architecture.

Shared nothing

- physical replication
- logical replication
- foreign tables, postgres_fdw

Shared disk

- no synchronization, but needs a fault-tolerant storage
- no stock shared disk support

If you want a cluster of vanilla PostgreSQL servers, your only bet is the `shared nothing` architecture: each server has its own storage and memory, and they talk to each other over the network. Data is synchronized by means of replication, which we explored previously.

Physical replication, unidirectional as it is, lends itself to setups with one leader and multiple replicas.

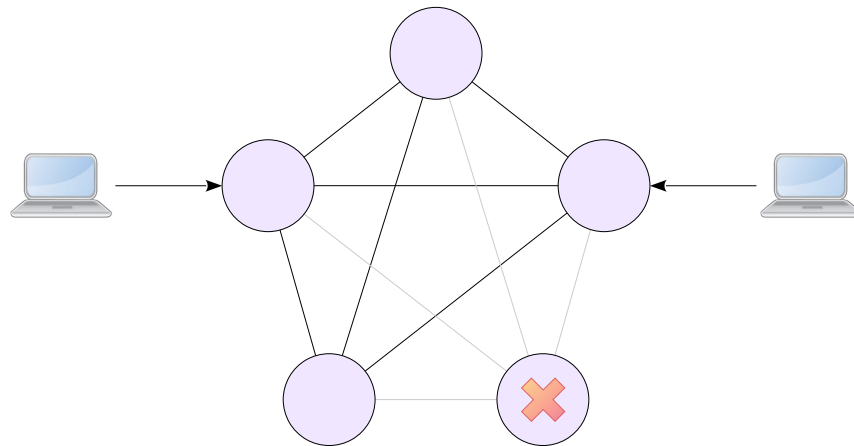
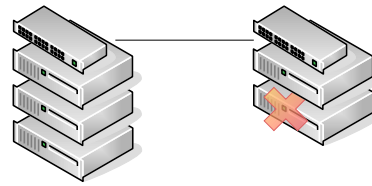
Logical replication, the more flexible of the two, supports bidirectional synchronization setups as well as selective replication, down to specific tables and rows. It enables geographically distributed clusters with multiple servers running independently, but sharing any changes.

Selective replication can also be achieved by other means, such as external tables with the postgres_fdw utility.

The shared disk architecture requires modifications to the PostgreSQL core, but offers some exciting new opportunities. With a shared disk, synchronization is unnecessary (the storage itself might use block-level replication for redundancy, but it appears as a single device to PostgreSQL). Since compute nodes are stateless, they can be started and stopped at any time. This approach is a natural fit for dynamic workloads and is ideal for cloud environments.

Node Failure

failure detection:
periodic signal indicating normal operation
(heartbeat)



10

Before discussing failure detection and management, we need to define what constitutes a failure. From a cluster software perspective, there are typically two types of failures: node failure and network partitioning.

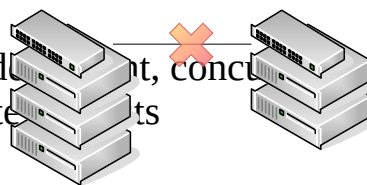
When a node fails, it stops operating as part of the cluster. A failure can result from a database process stop or crash, an operating system crash, server shutdown, or similar issues.

To detect failures, cluster nodes periodically exchange brief heartbeat messages. When a node fails to respond in time, the system registers the failure and initiates the appropriate recovery procedures.

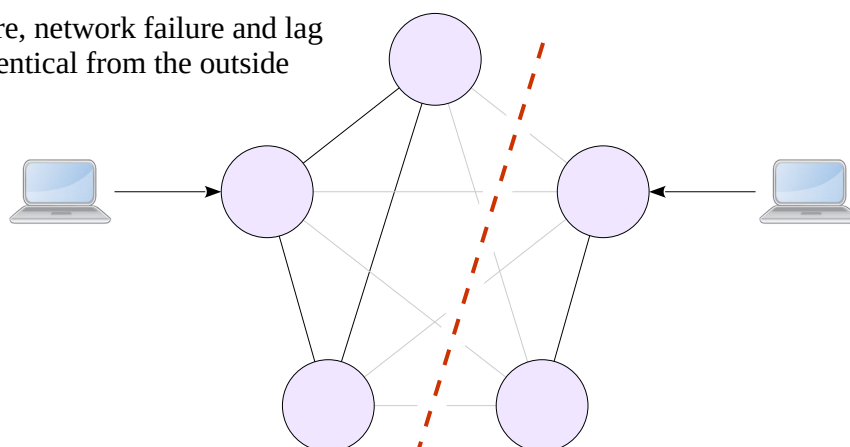
The failure recovery procedure depends on the cluster architecture. In a leader-replica cluster, for example, if the leader fails, one of the replicas must take over its role through a failover.

Network Partition

split-brain: the cluster fragments into independent, concurrent operating segments that each accept write requests



node failure, network failure and lag all look identical from the outside



11

A network partition occurs when network equipment fails. When this happens, all nodes may remain operational, but one group of nodes loses connectivity with the other.

The worst case is when both groups maintain connectivity with clients. This can lead to a split-brain: the cluster splits into two segments, each operating independently and continuing to accept write requests. This leads to data divergence, and when recovery takes place, one group's data will inevitably be lost.

Note that a network can fail completely, or it may experience temporary latency or packet loss. (Additionally, in some operating systems, factors such as CPU load can cause network delays).

From the cluster's perspective, all these scenarios are indistinguishable from a node failure: if a node doesn't send a heartbeat in time, the cluster must consider it downed. A node may or may not still be functioning, but there's no way to tell without any connection to it.

Many systems will operate unreliably when network connectivity is unstable. To reduce the likelihood of false positives, you can increase the threshold value, but this will also increase the time it takes to detect an actual failure, which in turn reduces availability.

For example, a popular distributed key-value store etcd suggests setting the threshold to ten times the regular node response time.

<https://github.com/etcd-io/etcd>

The Consensus Problem

Cluster nodes must be able to reach consensus

- each node sees the same current state of the whole cluster
- atomic commits of global transactions

Consensus through messages

- no shared memory (shared nothing)
- dedicated distributed protocols that account for potential failures at every stage of voting

Designing, proving correctness, and testing the implementation are all challenging tasks!

A cluster made up of independent nodes must operate as a single system, in a sense.

To handle failures, nodes must share a common understanding of each other's states. To commit a global transaction, all cluster nodes must agree that it does not break anything, and then execute the commit atomically.

Since the nodes have no shared storage (we're looking at shared-nothing systems here), the only way to reach agreement is through a *distributed consensus protocol*. The protocol specifies what messages nodes need to exchange – and in what order – to reach consensus.

Distributed algorithms are *extremely complex* because they must account for numerous edge cases. For example, they must function correctly even when a failure occurs at any stage of the voting process. Proving the correctness of such algorithms and testing their implementations is equally challenging. You shouldn't attempt to create your own or trust unproven ones.

2PC and 3PC Protocols

Two-phase commit

prepare: the coordinator holds a vote
commit: if every node votes yes, the coordinator signals to commit
supported by PostgreSQL (PREPARE TRANSACTION + COMMIT)

Three-phase commit

prepare: the coordinator holds a vote
precommit: if every node votes yes, the coordinator notifies all participants of the intent to commit
commit: if all nodes acknowledge, the coordinator signals to commit

Properties

there are situations (such as network partition) where a transaction cannot be resolved until the failure is fixed

13

PostgreSQL comes with a straightforward two-phase commit (2PC) protocol that handles global commits. The node executing the transaction acts as the coordinator.

During the prepare phase, the coordinator informs the other nodes about the transaction commit and waits for them to confirm that they're ready to commit (if at least one node isn't ready, the transaction is aborted). During the commit phase, the coordinator orders all nodes to commit the transaction. Once the coordinator receives commit confirmation from all nodes, it commits the transaction as well.

This protocol requires minimal message exchange but isn't fault-tolerant. If the coordinator fails after the prepare phase, the remaining nodes have no way of knowing whether to commit or abort the transaction. They'll have to wait until the failure is resolved.

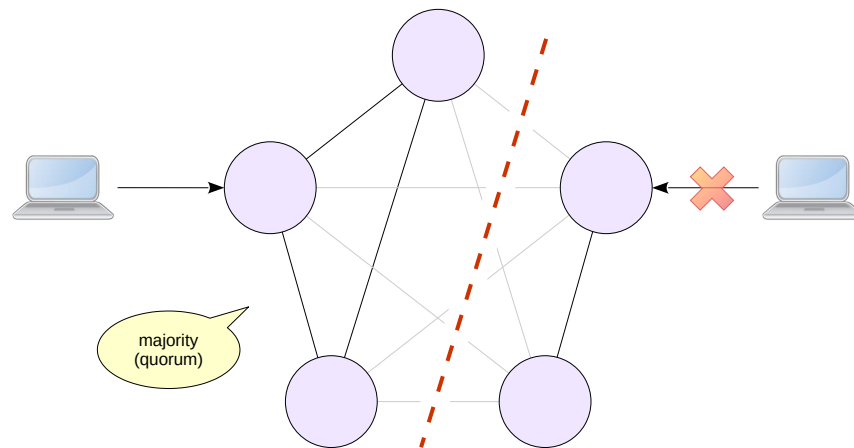
A more sophisticated protocol called three-phase commit (3PC) addresses this limitation. It introduces an additional precommit phase where the coordinator informs all nodes of its *decision* to commit the transaction. After that, if the coordinator fails, the remaining nodes can proceed to commit the transaction on their own (or abort it independently if the failure occurs before the precommit phase).

However, both algorithms break down in network partition scenarios.

Consensus protocols

Paxos, Raft, ZAB, E3PC, and others

if up to $N/2-1$ nodes fail, the connected majority continues operating
the minority remains blocked until the failure is resolved



14

Consensus (majority voting) algorithms enable nodes that maintain connectivity with each other and form a quorum to continue operating during network partitions. With equal voting weight across all nodes, $N/2+1$ out of N nodes can remain operational.

Nodes in the minority must wait for the failure to be resolved before they can commit or roll back their open transactions. These nodes can either stop serving clients entirely or at least refuse write requests to prevent split-brain.

Several mathematically proven algorithms exist with battle-tested implementations.

- Paxos was published in 1998, though it actually originated in the late 1980s. There's an entire family of these algorithms designed for different scenarios.
- Raft was designed as an algorithm equivalent to Paxos but easier to comprehend. In Raft, one node is elected as the leader with write privileges, while the remaining nodes are read-only. It became the basis for distributed key-value storage systems like etcd and Consul.
- ZAB is the algorithm used by ZooKeeper.
- Enhanced 3PC (E3PC) improves upon the three-phase commit algorithm by allowing transaction commits with a majority vote rather than requiring unanimous agreement.

Other approaches exist for solving the consensus problem, such as Virtual Synchrony, which is used by the Corosync system.

Read throughput

load distribution across cluster nodes

Write throughput

cannot be achieved with full data replication to all nodes

sharding (distributing table rows across different nodes) depends heavily on the nature of the data being stored

Response time

parallel computation across multiple nodes

15

Read throughput is the easiest to scale: just distribute read transactions more or less evenly across the cluster nodes.

Scaling write throughput is typically more challenging: in leader-leader configurations, nodes must exchange messages and data with each other, which means the overall system load actually increases as more nodes are added.

However, this becomes feasible when data is distributed across nodes rather than being fully replicated to each one.

Sharding – distribution of data at table row level – is one approach that is used often. You can think of it as partitioning where the partitions are located on different cluster nodes.

Note that specific sharding schemes are heavily dependent on the data logic; there's no one-size-fits-all solution here.

Sharding also enables response time scalability. To achieve this, the cluster needs to be able to parallelize queries by executing them simultaneously across multiple nodes that hold the required data. This is similar to parallel query execution in standard PostgreSQL, except the parallel processes run across different nodes instead of within a single server. Of course, this increases the overhead of data transfer between processes, such as when passing partial aggregation results.

PostgreSQL-specific tools

- libpq library
- PgBouncer – connection pool manager
- Pgpool-II – load balancer and connection pool manager
- Odyssey – multithreaded connection pool manager
- ...

Other tools

- virtual IP address
- HAProxy – TCP/HTTP load balancer
- round-robin DNS
- ...

16

There are plenty of ready-made solutions available for load distribution; typically, these tools are not part of the database cluster itself.

- The libpq connection string supports multiple hosts. The order of connections is defined by the *load_balance_hosts* parameter, and *target_session_attrs* lets you set session attributes, such as read-only connections.
- PgBouncer is a lightweight connection pooling manager for PostgreSQL.
- Pgpool-II is a connection pooling manager and load balancer that can route write traffic to the leader and read traffic to replicas.
- Odyssey is a multithreaded connection pooling manager with full SSL/TLS support and LDAP and PAM authentication capabilities.

Some general-purpose tools not specific to PostgreSQL:

- *Virtual IP address* – an IP controlled by the cluster that points the clients to specific nodes.
- *HAProxy* is a load balancer and proxy server that operates at the TCP and HTTP layers. Node lists can be kept up to date with tools like *confd*, or you can use HAProxy's own health check tools to keep track of healthy nodes and kick faulty ones from the balancing pool.
- Some systems (like Consul) provide *round-robin DNS*: node selection and load balancing can happen at the name resolution stage (though caching negates the benefits of this approach).

There are other tools available as well, and many of them work well together.

Required Third-Party Components

System Examples

Agent

manages a single PostgreSQL instance

Control node

cluster-wide management logic

Consensus

external system managed by a consensus protocol

shared state support for control nodes

Entry point

a mechanism to route clients to the appropriate cluster node

If what you really want from your cluster above else is high availability and reliability, the obvious route is setting up several instances of baseline PostgreSQL and linking them together with some other software. Data synchronization, then, will be achieved by physical replication, meaning that the cluster will have one leader and several replicas.

Since PostgreSQL cannot configure, restart, or stop itself, a dedicated program is needed to handle these operations. This role is performed by an **agent**.

The cluster management logic is handled by **control nodes**. At this level, decisions are made about which nodes should be considered failed, which replica should be promoted to the new leader when the old one fails, and so on.

To maintain a shared view of the cluster state, the control nodes rely on a system that implements one of the distributed **consensus** protocols discussed earlier.

Finally, clients need a single **entry point** that routes them to the appropriate cluster node or nodes.

Different clustering systems may use different names for these components, and some may be combined into a single entity. If certain components are missing, the cluster must be supplemented with third-party solutions.

Isolating a node in an unknown state from clients

a network partition can lead to split-brain

must not only isolate the node but also prevent failover to a replica during this time

Mechanisms



power cutoff

network port cutoff

client connection termination and redirection

Since PostgreSQL servers in this approach are unaware they're operating within a cluster, it's essential to be able to fence off a failed node (which may still be running but has lost connectivity to the cluster) from the outside world. This procedure is called fencing.

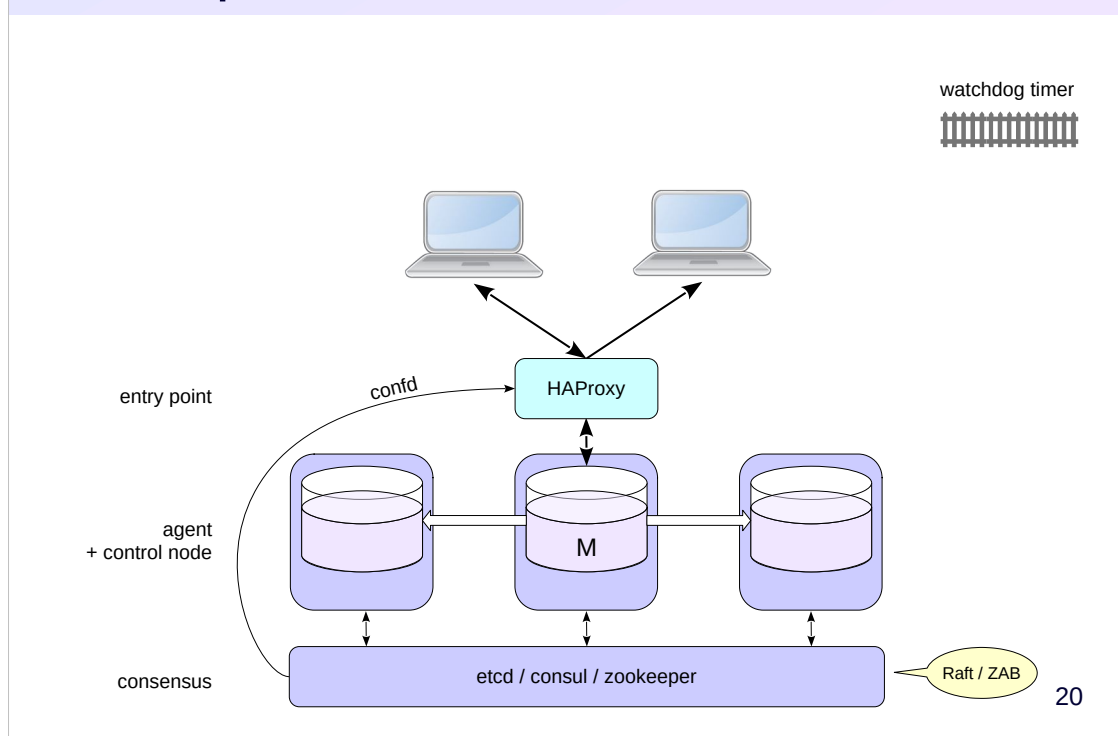
Without proper fencing, a leader that ends up in the minority partition during a network split could continue operating, leading to split-brain. You cannot rely on the control nodes to stop the PostgreSQL server on their own when they detect a failure: they might lose connection with the agent, the agent itself could fail, and so on.

The procedure cannot depend on the node being reachable over the network. Cutoff must be set up differently: For example:

- programmatically power off the server (using IPMI or PDU interfaces);
- programmatically disable the network port on the switch;
- if connections go through a specialized component, clients can be forcibly disconnected from the failed node at that component's level.

While fencing reduces the scope of the problem, it doesn't completely solve it on its own. You must also ensure that failover to a replica cannot occur until the failed leader has been fenced. After all, making the fencing decision takes time, during which the node may continue operating, meaning some committed data could be lost during recovery (zero RPO cannot be achieved).

Example: Patroni



Patroni (<https://github.com/patroni/patroni>) is an open-source clustering solution designed for use in both cloud and dedicated server environments.

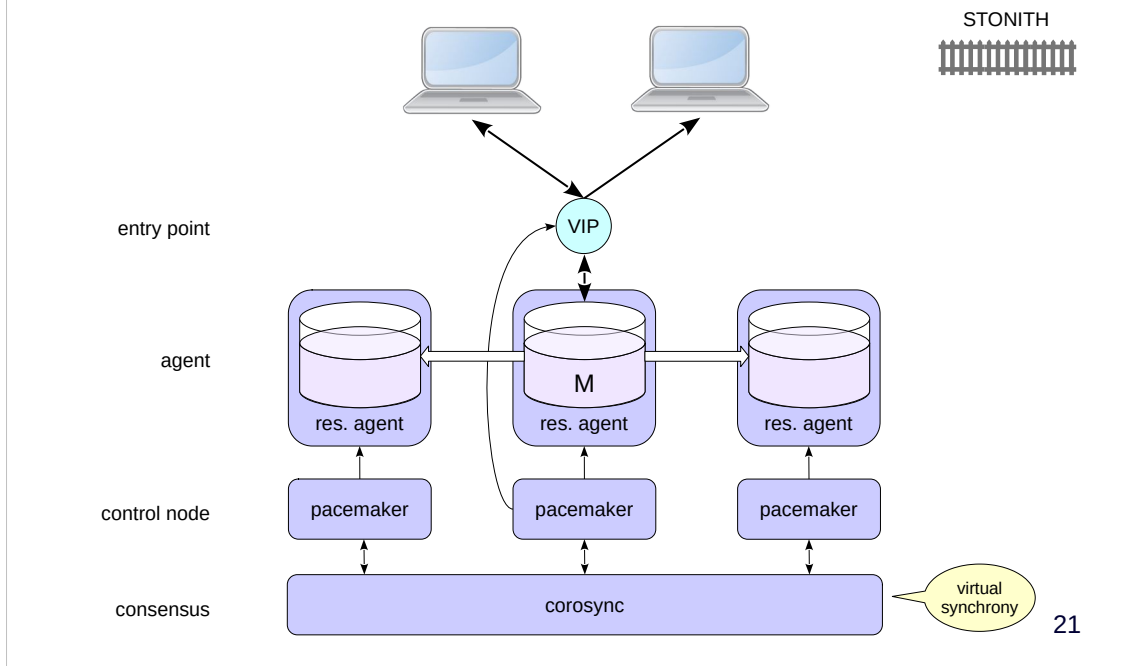
Here, the agent and control functions are combined in a single component. Consensus is achieved using a key-value store provided by etcd (Raft), Consul (Raft), or ZooKeeper (ZAB).

A separate solution is needed to provide the entry point. The typical recommendation is to use HAProxy together with confd, which periodically updates the HAProxy configuration based on information from the consensus layer. The standard configuration routes all clients to the node serving as the leader.

To prevent HAProxy from becoming a single point of failure, you need multiple HAProxy servers and a virtual IP address for client connections. However, this means that configurations across different HAProxy servers must be updated synchronously, bringing us back to the clustering problem – just at a different level.

Node fencing can be implemented using the Linux kernel's watchdog timer mechanism. If the node, while serving as the leader, cannot contact the consensus layer for a certain period, the timer triggers an operating system reboot.

Example: The ClusterLabs Stack



ClusterLabs (<https://clusterlabs.org/>) provides an open source component stack for building clusters from virtually any systems.

The agent (referred to as a resource agent in this framework) treats each specific system as a standardized resource by implementing operations like stop, start, replica promotion, and so on. This component needs to be implemented separately for each system.

Pacemaker, the controller component, interacts with resources through their agents. Corosync (Virtual Synchrony) handles cluster-wide state management and failure detection.

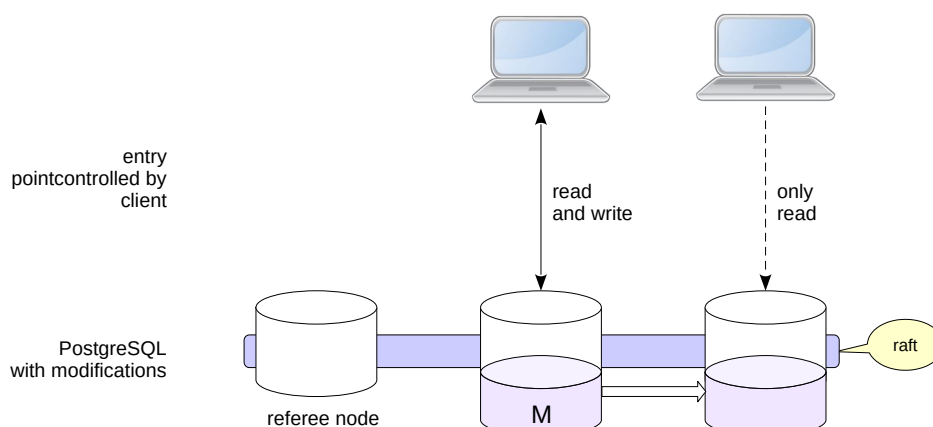
This architecture focuses heavily on node fencing. This approach uses the STONITH technology (Shoot The Other Node In The Head), which reliably isolates a failed node from both the cluster and clients by remotely cutting power or disabling its switch port.

The client entry point is typically a virtual IP address that the controller node assigns to whichever node is currently serving as the leader.

If needed, you can also distribute read workload across replicas using tools like HAProxy, PgBouncer, and similar solutions.

For PostgreSQL specifically, there's an open-source solution PAF (PostgreSQL Automatic Failover): <http://clusterlabs.github.io/PAF/>

Example: BiHA



22

A large number of external components clearly complicates cluster deployment and maintenance. To simplify these tasks, all necessary components can be built directly into PostgreSQL itself. This complex challenge has been addressed in the BiHA (Built-in High Availability) cluster, which is an integral part of Postgres Pro Enterprise.

From the outside, this cluster appears as regular nodes connected by physical replication, but in reality, each Postgres Pro Enterprise node implements failure detection, split-brain prevention, a consensus layer (using the Raft protocol), management logic, and an agent.

Consensus requires at least three nodes, though it's not necessary to store data on all of them. You can deploy a cluster where the third node serves as a referee: it participates in the quorum but doesn't take part in replication. A less powerful server can be used for this type of node.

No additional software is needed for the entry point: the cluster relies on the client's ability to determine on its own whether it's connected to the leader (which handles both read and write operations) or to a replica. Such capabilities are available in libpq, among other tools.

<https://postgrespro.com/docs/enterprise/16/biha-solution>

Global Transaction Manager

- requires modifications to the PostgreSQL core
- can be built in or implemented as a separate component

23

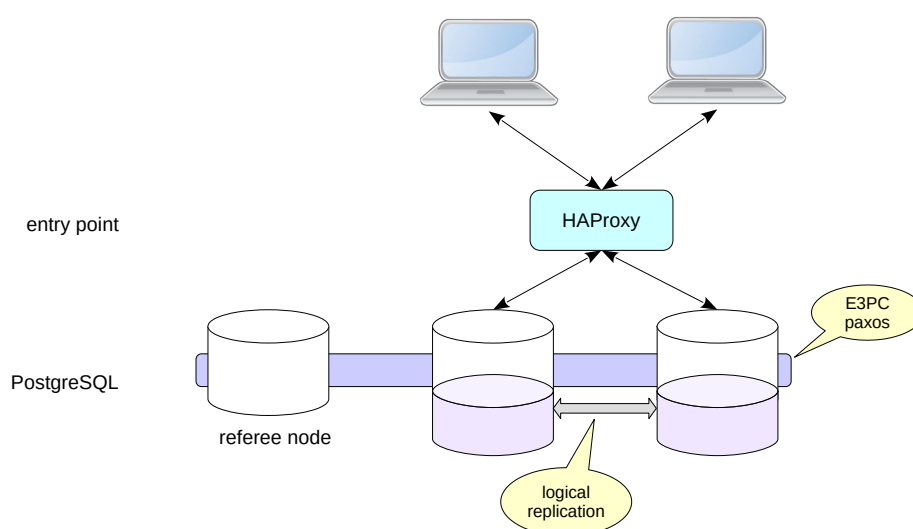
In the previous examples, we added components to PostgreSQL that enable fault tolerance: agents, control nodes (including failure detection systems), and a consensus layer for controlling the state of the cluster. These components could exist either outside the PostgreSQL nodes or be integrated directly into PostgreSQL itself.

To create a cluster that provides clients with a unified, consistent view of data, you need a Global Transaction Manager that implements a distributed consensus protocol to ensure atomic commits for transactions spanning multiple nodes.

The ability to work with global transactions cannot be achieved through external means alone, it must be built into PostgreSQL by modifying the core. Even when global transaction management is implemented as a separate cluster component, PostgreSQL must still be aware that it needs to communicate with this component.

Typically, fault tolerance is required in addition to consistency. The cluster must then include all the components we discussed earlier. If the Global Transaction Manager is implemented as a separate component, you'll also need to ensure its fault tolerance, which adds another layer of complexity to the system.

Example: multimaster



24

The Postgres Pro Enterprise multimaster extension is an example of such a system. It's a Global Transaction Manager integrated into the database system that extends the existing two-phase commit protocol (2PC) to E3PC. To accomplish this, arbiter processes are launched on the nodes to handle additional communication. The system uses the Paxos consensus algorithm.

<https://postgrespro.com/docs/enterprise/16/multimaster>

Any node in the multimaster cluster (except the referee node, if one is configured) can accept both read and write queries. Read-only transactions are executed locally. Write transactions are distributed; the node that commits the transaction acts as the coordinator.

Data replication between cluster nodes relies on logical replication, while DDL changes are replicated using a proprietary algorithm.

To distribute load effectively, it makes sense to balance connections across cluster nodes. For example, you can assign different connection strings to different clients, or use HAProxy for load balancing.

An application running on a single server can be deployed on a multimaster cluster without any additional concerns about data consistency. The supported isolation levels are read committed and repeatable read. However, due to current implementation limitations, applications need to account for certain nuances.

Beyond implementing distributed transactions, the cluster also provides fault tolerance.

Large number of nodes

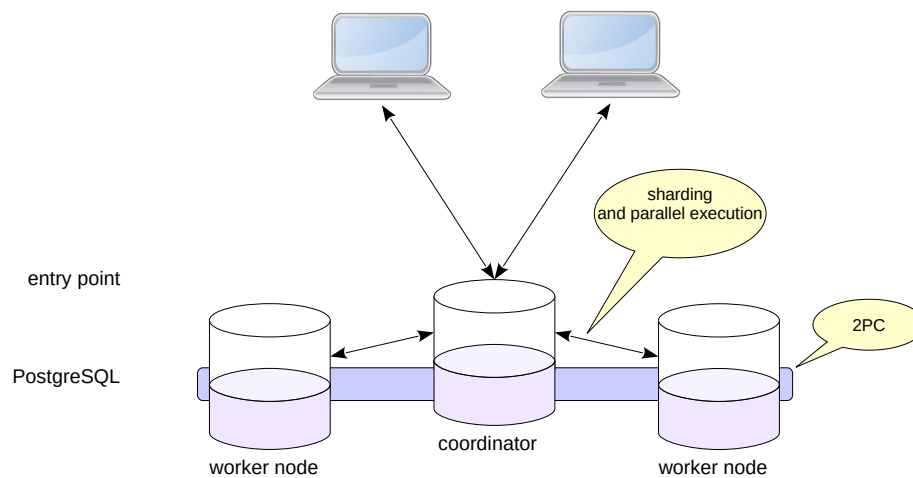
- shared storage or replication
- sharding
- parallel processing

When scalability becomes the primary concern, you can set up a relatively straightforward cluster built from standard PostgreSQL instances without the typical clustering infrastructure. This type of cluster enables load distribution across different nodes, though it won't provide fault tolerance or consistency guarantees.

Data can be fully replicated across nodes (enabling read scalability) or partially replicated (for example, through sharding—which also enables write scalability). Systems designed for cloud infrastructure with dynamic resource scaling can use a shared storage approach where nodes are separated into compute and storage tiers.

Distributing data across a large number of nodes also enables parallel query execution.

Example: Citus



26

Citus (<https://github.com/citusdata/citus>) is a PostgreSQL extension that distributes data and queries across cluster nodes.

Citus is an example of a system where data isn't fully replicated but instead distributed across worker nodes through sharding. One node is designated as the coordinator and contains metadata about data location.

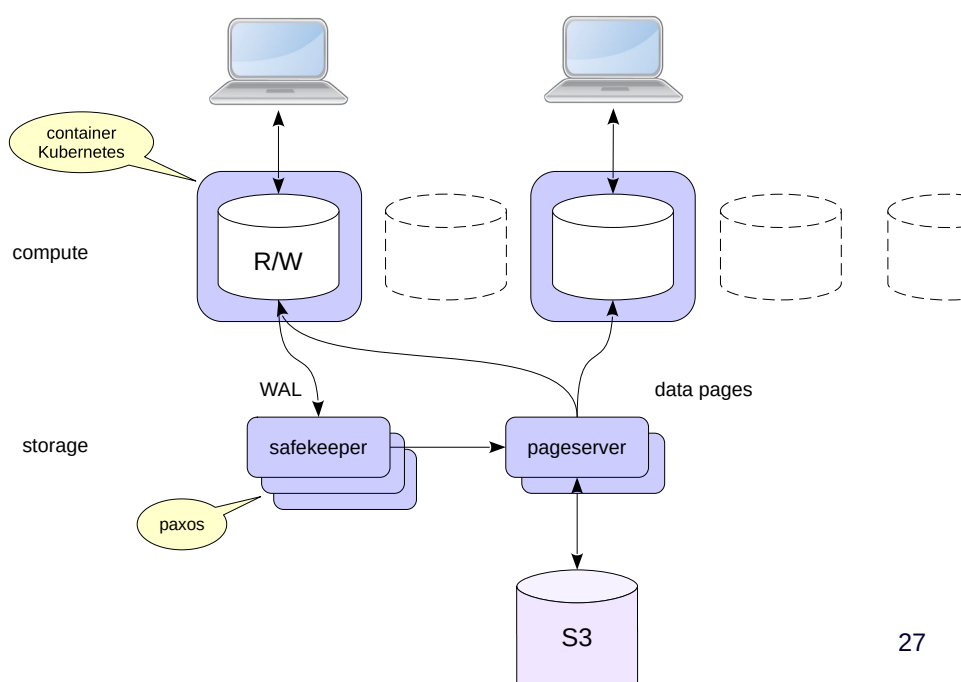
The extension includes a specialized query optimizer that generates parallel execution plans and an executor that runs queries by distributing them across the appropriate nodes. This approach enables a Citus cluster to scale both in terms of response time and write throughput.

Citus ensures atomic changes across all cluster nodes using global transactions with the two-phase commit (2PC) protocol.

Data rebalancing across worker nodes (when adding new nodes or removing old ones) relies on logical replication.

In this configuration, the cluster doesn't provide high availability: if any node fails, some data will be lost. To address this, both the coordinator and worker nodes can be replaced with individual mini-clusters of three nodes each (one primary and two replicas) to ensure fault tolerance.

Example: Neon



27

Neon is an open-source solution for cloud-based serverless computing. The concept involves splitting the database management system into separate stateless nodes while leveraging large, inexpensive, yet slower data storage. Compute nodes (essentially PostgreSQL instances stripped of their logging and disk operations) are launched within Kubernetes-managed containers. Since these nodes are stateless, containers can be spun up and shut down at any time based on client activity, eliminating costs for unused resources.

To persist data, the write instance (only one is allowed) sends WAL records to the safekeeper component. The records from committed transactions must be stored reliably; to achieve this, safekeeper components form a separate fault-tolerant cluster that operates using the Paxos protocol.

WAL records are asynchronously forwarded from the safekeeper to the pageserver component, which applies them to the data pages. The pageserver retrieves pages on demand from clients, flushing them to bottomless S3 object storage and reading them back as needed. In this way, the pageserver acts as a cache for actively used data, functioning as the equivalent of a buffer cache.

Neon scales for read operations and automatically adjusts to workload demands, though it inevitably performs slower than traditional systems.

<https://github.com/neondatabase/neon>

Building a cluster requires tools that aren't included in standard PostgreSQL

There are numerous approaches to building a cluster

- different objectives: high availability, scalability, and consistency
- solutions involving external components or kernel modifications
- solutions with global transaction support

The higher the requirements,
the more complex the system becomes and the higher the cost