

Scaling PostgreSQL: Navigating Horizontal and Vertical Scalability Pathways



Principal Engineer

Ibrar Ahmed



Postgres

Conference 2024
San Jose / United States
April 19, 2024



Introduction to Scalability in PostgreSQL

- **Horizontal vs. Vertical Scalability:** Enhance PostgreSQL's capability by scaling up (enhancing server capacity) or scaling out (adding more servers).
- **Replication and Partitioning:** Use replication for high availability and partitioning to manage large datasets efficiently.
- **Connection Pooling:** Implement connection pooling with tools like PgBouncer to manage numerous concurrent connections efficiently.
- **Indexing Strategies:** Optimize query performance in large datasets with appropriate indexing (B-tree, GIN, BRIN, etc.).
- **Configuration and Tuning:** Adjust PostgreSQL configuration parameters (e.g., `work_mem`, `shared_buffers`) for optimal performance and scalability.

01

Vertical Scalability

+



+

Vertical Scalability in PostgreSQL

- **Optimize Configuration**

- Adjust critical server settings like `shared_buffers` and `work_mem` to enhance performance, reducing disk I/O and improving query response times.
- Tailoring these parameters to match specific workloads optimizes hardware resource utilization, crucial for vertical scalability.

- **Indexing**

- Employ strategic indexing (e.g., B-tree for general queries, GIN for searches) to facilitate faster data retrieval and lower query execution times.
- Proper index management minimizes system load, directly contributing to more efficient database operations.

- **Query Optimization**

- Analyze and refine SQL queries using EXPLAIN, focusing on efficient joins, optimized subqueries, and minimal data fetching for better performance.
- This process reduces the database's resource consumption and execution time, enhancing its ability to handle more requests.

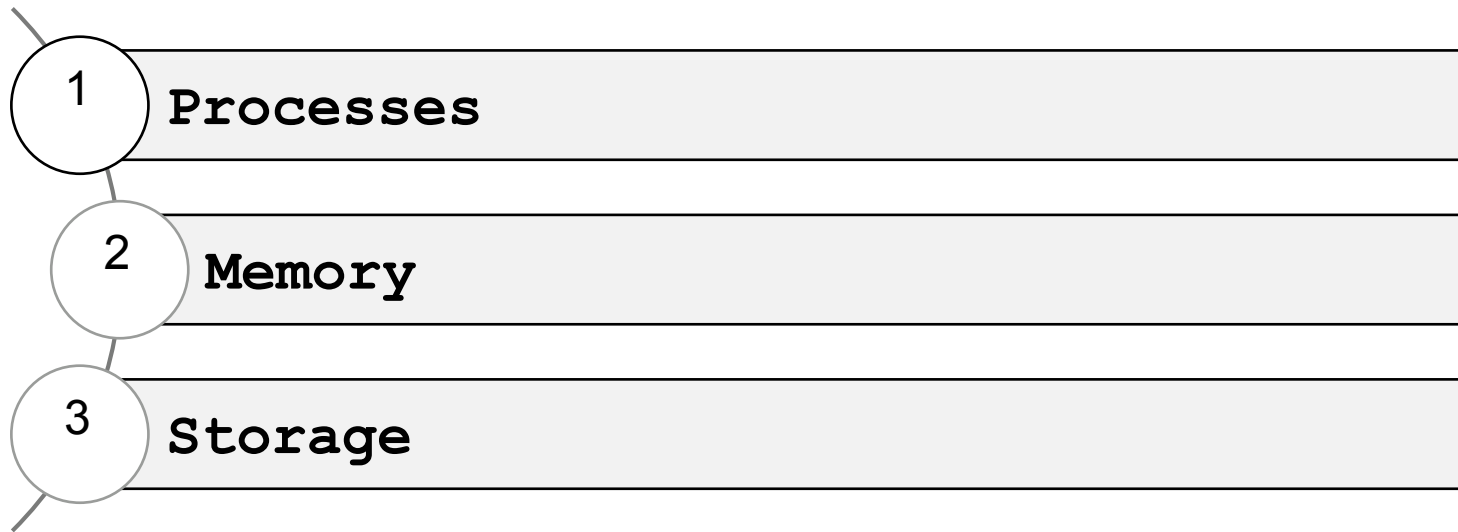
- **Connection Management**

- Implement connection pooling solutions like PgBouncer to manage and reuse database connections, reducing overhead and improving scalability.
- Efficient connection management allows PostgreSQL to handle a higher number of concurrent connections without significant resource strain.

- **Maintenance**

- Regularly perform database maintenance tasks such as vacuuming, analyzing, and reindexing to prevent data bloat and maintain query efficiency.
- These activities ensure the database remains in optimal health, supporting sustained performance and scalability over time.

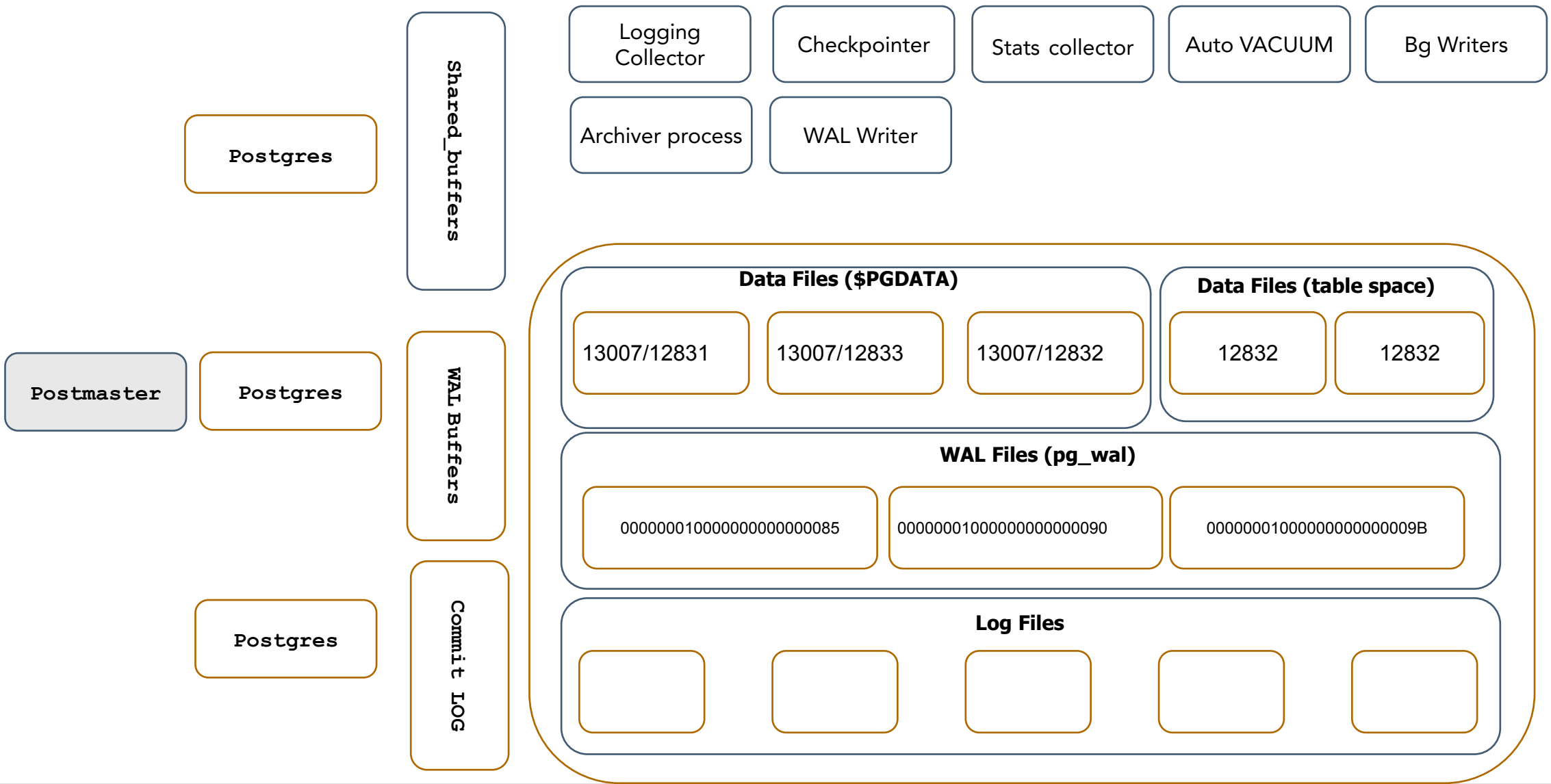
PostgreSQL Architecture



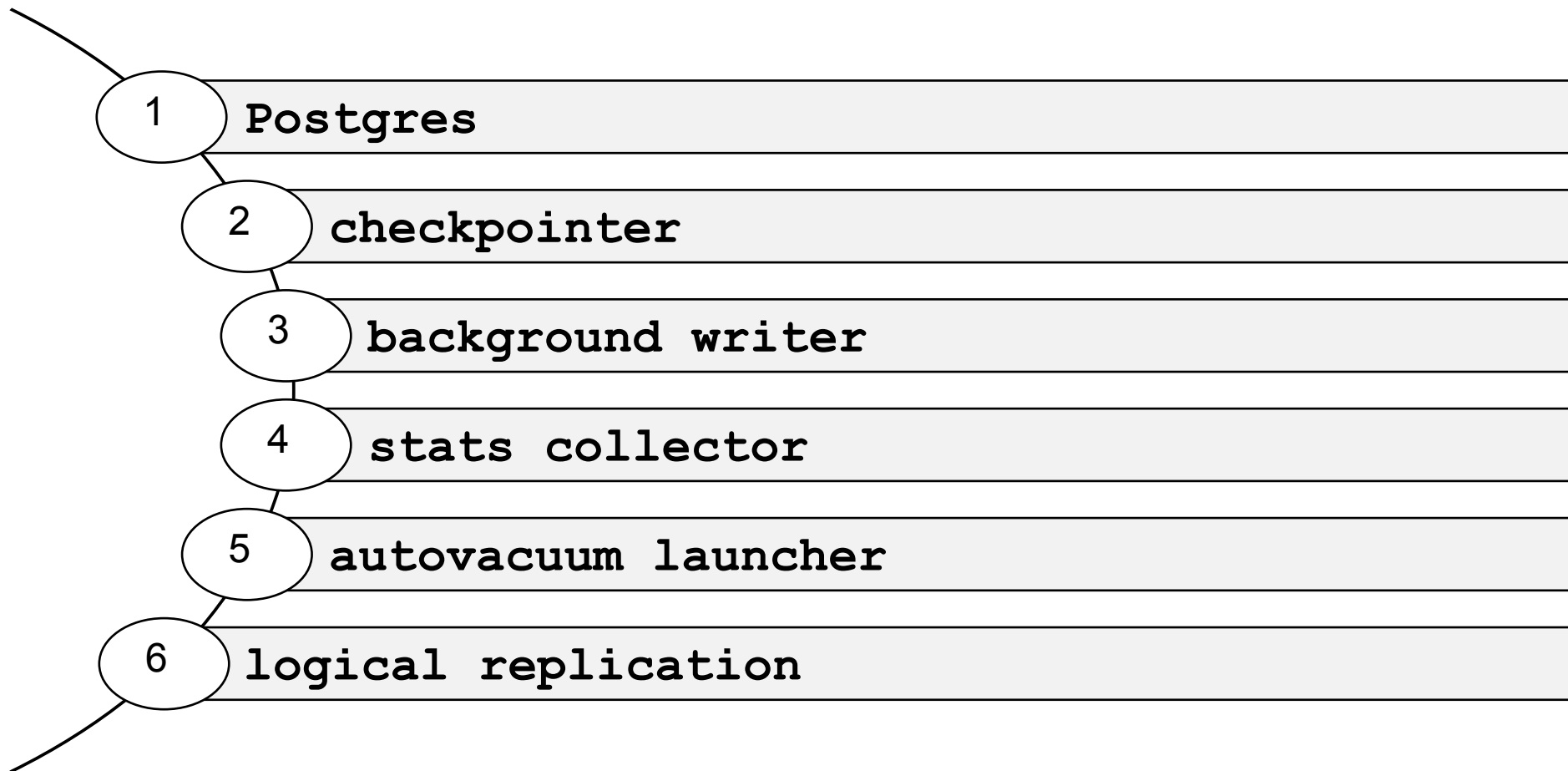
Why Architecture before Installation?

- Many users start installation without knowing the architecture.
- Architecture gives you a better understanding to choose the desired database and configuring it accordingly

PostgreSQL Architecture

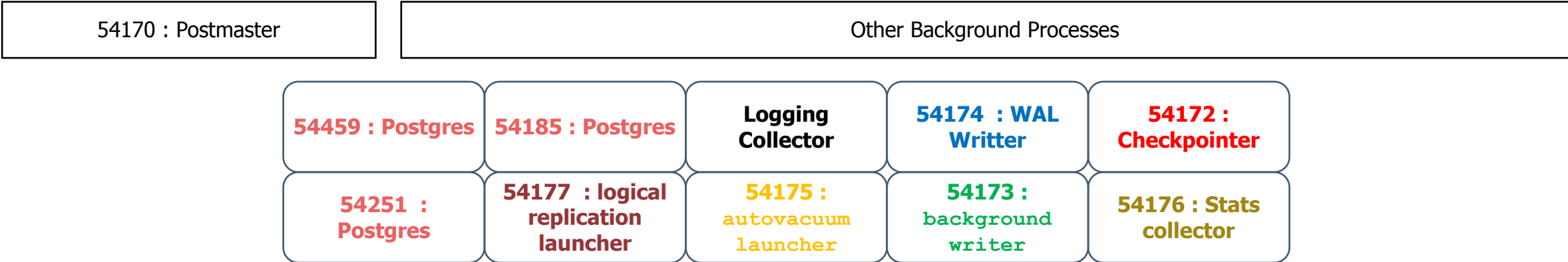


PostgreSQL Processes

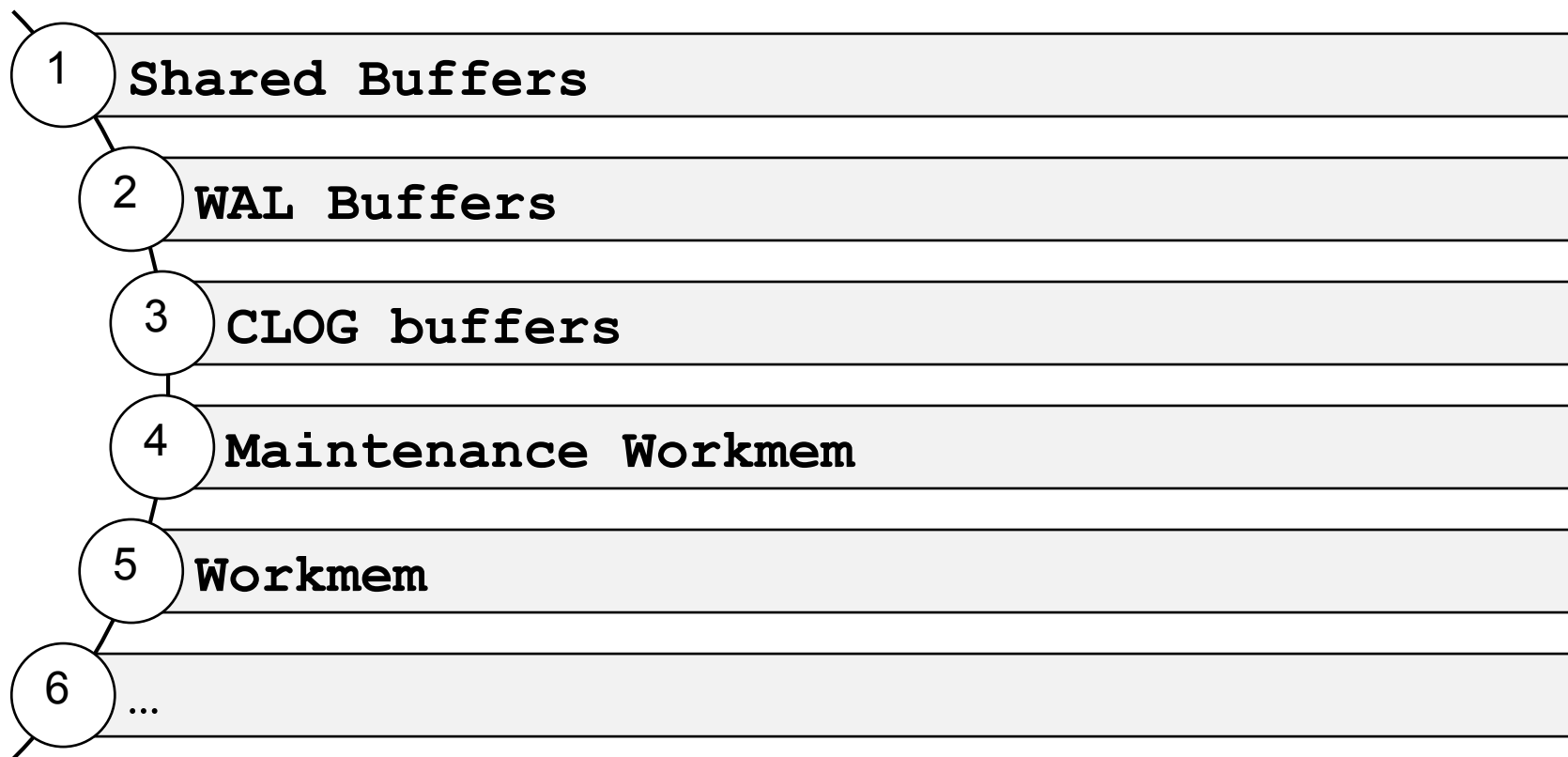


PostgreSQL Processes

postgres	54170	0.0	0.6	292140	18:16	0:00	/usr/local/pgsql.14/bin/postgres
postgres	54184	0.0	0.0	13284	18:16	0:00	psql postgres
postgres	54251	0.0	0.0	13356	18:17	0:00	psql postgres
postgres	54459	0.0	0.0	13284	18:18	0:00	psql postgres
postgres	54185	0.0	0.1	293688	18:16	0:00	postgres vagrant postgres [local] idle
postgres	54252	0.0	0.1	293096	18:17	0:00	postgres vagrant postgres [local] idle
postgres	54460	0.0	0.1	293096	10208	0:00	postgres vagrant postgres [local] idle
postgres	54172	0.0	0.0	292140	18:16	0:00	postgres:checkpointer
postgres	54173	0.0	0.0	292272	18:16	0:00	postgres:background writer
postgres	54174	0.0	0.0	292140	18:16	0:00	postgres:walwriter
postgres	54175	0.0	0.0	292700	18:16	0:00	postgres:autovacuum launcher
postgres	54176	0.0	0.0	23368	18:16	0:00	postgres:stats collector
postgres	54177	0.0	0.0	292692	18:16	0:00	postgres:logical replication launcher



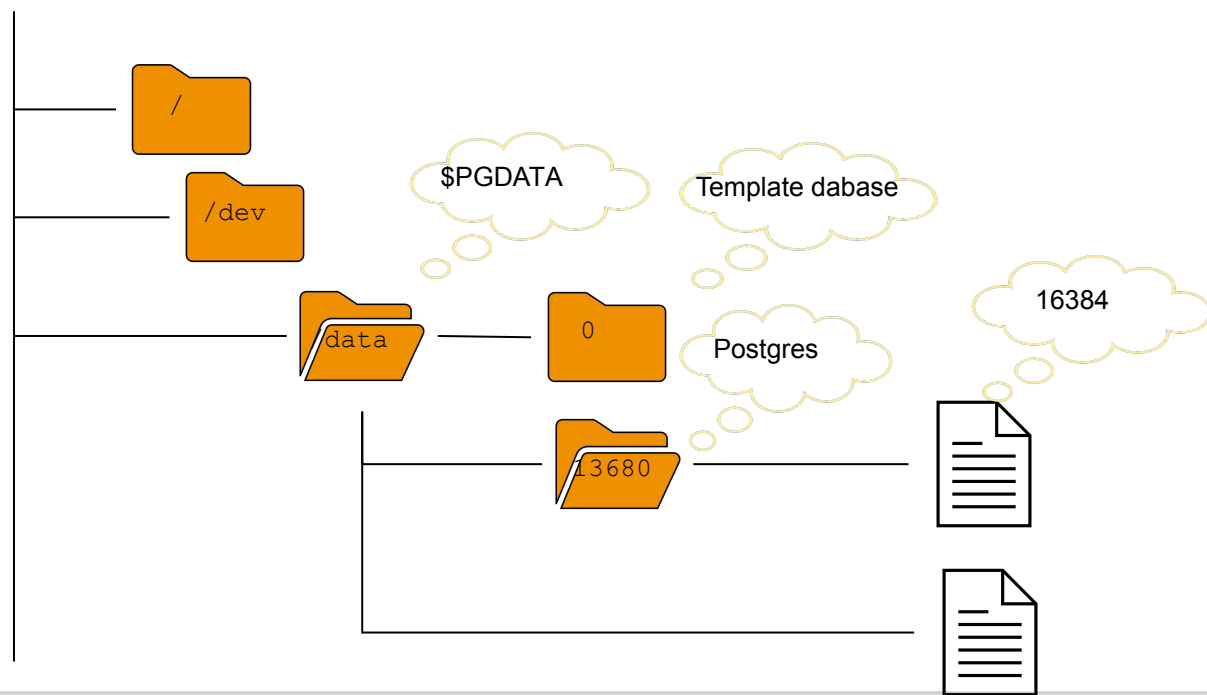
PostgreSQL Memory



PostgreSQL Storage: Table Storage

```
CREATE TABLE admin(id int, name text, dt date);
SELECT relfilenode FROM pg_class WHERE relname LIKE 'admin';
relfilenode
-----
16384

$ ls $PGDATA/base/13680/16384
$PGDATA/base/13680/16384
```



```
postgres=# CREATE DATABASE learn_and_apply;

postgres=# \l
      List of databases
-----+-----+
Name                | Owner  |
-----+-----+
learn_and_apply     | vagrant |
postgres            | vagrant |
template0           | vagrant |
template1           | vagrant |
```

Database Performance



Tuning Parameter

PostgreSQL Tuning
Parameters



PostgreSQL Indexes

Impact of index on Database
Performance



Query Analysis

Analyze your queries for
optimal database performance



Partitioning

Partition your database when
require



Monitoring

Monitor your database to
identify the bottleneck

Tuning Parameters

Memory based configuration
parameters

1

shared_buffers
PostgreSQL
Memory buffer

wal_buffers
Buffer for WAL

effective_cache_size
PostgreSQL
Cache

work_mem
Buffer for sorting

maintenance_work_mem
Buffer for
Maintenance
activity

Other
postgresql.conf
file contains all
the parameters

shared_buffers

- PostgreSQL uses its own buffer along with kernel buffered I/O.
- PostgreSQL does not change the information on disk directly then how?
- Writes the data to shared buffer cache.
- The backend process write that these blocks kernel buffer.


```
postgres=# SHOW shared_buffers;  
shared_buffers  
-----  
128MB  
(1 row)
```



The proper size for the POSTGRESQL shared buffer cache is the largest useful size that does not adversely affect other activity.

—Bruce Momjian

wal_buffer

- Do you have Transaction? Obviously 
- WAL – (Write Ahead LOG) Log your transactions
- Size of WAL files 16MB with 8K Block size (can be changed at compile time)
- PostgreSQL writes WAL into the buffers(*wal_buffer*) and then these buffers are flushed to disk.

effective_cache_size

- **Estimation for Planner:** The `effective_cache_size` setting advises the PostgreSQL query planner on the memory available for caching, helping optimize query execution by influencing the choice between index and sequential scans. It is just a guideline, not the exact allocated memory or cache size.
- **Influences Index Usage:** A higher `effective_cache_size` can make index scans more attractive to the planner by suggesting ample cache memory is available, thus potentially reducing disk I/O for data access.
- **Configurable Based on System Resources:** It should be set considering the server's total RAM and other applications' memory usage, ideally higher than `shared_buffers` but lower than the total system memory.
- **Impact on Performance:** Proper configuration of `effective_cache_size` improves optimizer efficiency and database performance, especially for large dataset operations that benefit from caching.
- **Adjustment Recommendations:** The optimal `effective_cache_size` depends on workload specifics and server setup, requiring performance monitoring and adjustments based on real-world query execution outcomes.

work_mem

- This configuration is used for complex sorting.
- It allows PostgreSQL to do larger in-memory sorts.
- Each value is per session based, that means if you set that value to 10MB and 10 users issue sort queries then 100MB will be allocated.
- In case of merge sort, if x number of tables are involved in the sort then $x * \text{work_mem}$ will be used.
- It will allocate when required.
- Line in `EXPLAIN ANALYZE` "Sort Method: external merge Disk: 70208kB"

work_mem

```
postgres=# SET work_mem = '2MB';
postgres=# EXPLAIN ANALYZE SELECT * FROM foo ORDER BY id;
```

QUERY PLAN

```
-----
---
 Gather Merge  (cost=848382.53..1917901.57 rows=9166666 width=9) (actual time=5646.575..12567.495
 rows=11000000 loops=1)
   -> Sort  (cost=847382.51..858840.84 rows=4583333 width=9) (actual time=5568.049..7110.789 rows=3666667
 loops=3)
 Planning Time: 0.055 ms
 Execution Time: 13724.353 ms
```



```
postgres=# SET work_mem = '1GB';
postgres=# EXPLAIN ANALYZE SELECT * FROM foo ORDER BY id;
```

QUERY PLAN

```
-----
---
 Sort  (cost=1455965.01..1483465.01 rows=11000000 width=9) (actual time=5346.423..6554.609 rows=11000000
 loops=1)
   Sort Key: id
   Sort Method: quicksort  Memory: 916136kB
   -> Seq Scan on foo  (cost=0.00..169460.00 rows=11000000 width=9) (actual time=0.011..1794.912
 rows=11000000 loops=1)
 Planning Time: 0.049 ms
 Execution Time: 7756.950 ms
```



maintenance_work_mem

- maintenance_work_mem is a memory setting used for maintenance tasks.
- The default value is 64MB.
- Setting a large value helps in tasks like
 - VACUUM
 - RESTORE
 - CREATE INDEX
 - ADD FOREIGN KEY
 - ALTER TABLE

maintenance_work_mem

```
CHECKPOINT;  
SET maintenance_work_mem='10MB';  
SHOW maintenance_work_mem;  
maintenance_work_mem  
-----  
10MB  
(1 row)  
postgres=# CREATE INDEX idx_foo ON foo(id);  
Time: 12374.931 ms (00:12.375) ←
```

```
CHECKPOINT;  
SET maintenance_work_mem='1GB';  
SHOW maintenance_work_mem;  
maintenance_work_mem  
-----  
1GB  
(1 row)  
postgres=# CREATE INDEX idx_foo ON foo(id);  
Time: 9550.766 ms (00:09.551) ←
```

synchronous_commit

Controls the commitment of transactions to disk, balancing between data durability and transaction speed by managing the use of the Write-Ahead Logging (WAL) system.

- **Default Behavior:** By default, set to on, ensuring transactions wait for WAL records to be written to disk before completion, maximizing data safety.
- **Performance Impact:** Disabling (off) can enhance transaction throughput by reducing disk I/O wait times, beneficial for workloads where speed is prioritized over immediate durability.
- **Durability Trade-off:** With synchronous_commit off, there's a risk of losing the last few seconds of transactions in a crash, trading some data loss risk for improved performance.
- **Use Case Flexibility:** Administrators can adjust this setting to align with specific application requirements, making it a versatile tool for performance tuning and risk management in PostgreSQL.

Tuning Parameters

I/O based configuration
parameters

01

checkpoint_timeout
Time for checkpoint

3

max_wal_size
Max WAL size

5

checkpoint_completion_target
Checkpoint completion
target

checkpoint_timeout

The `checkpoint_timeout` setting in PostgreSQL is a crucial configuration parameter affecting database performance and reliability. It determines the maximum time interval between automatic WAL checkpoints, directly impacting disk I/O operations and overall system performance.

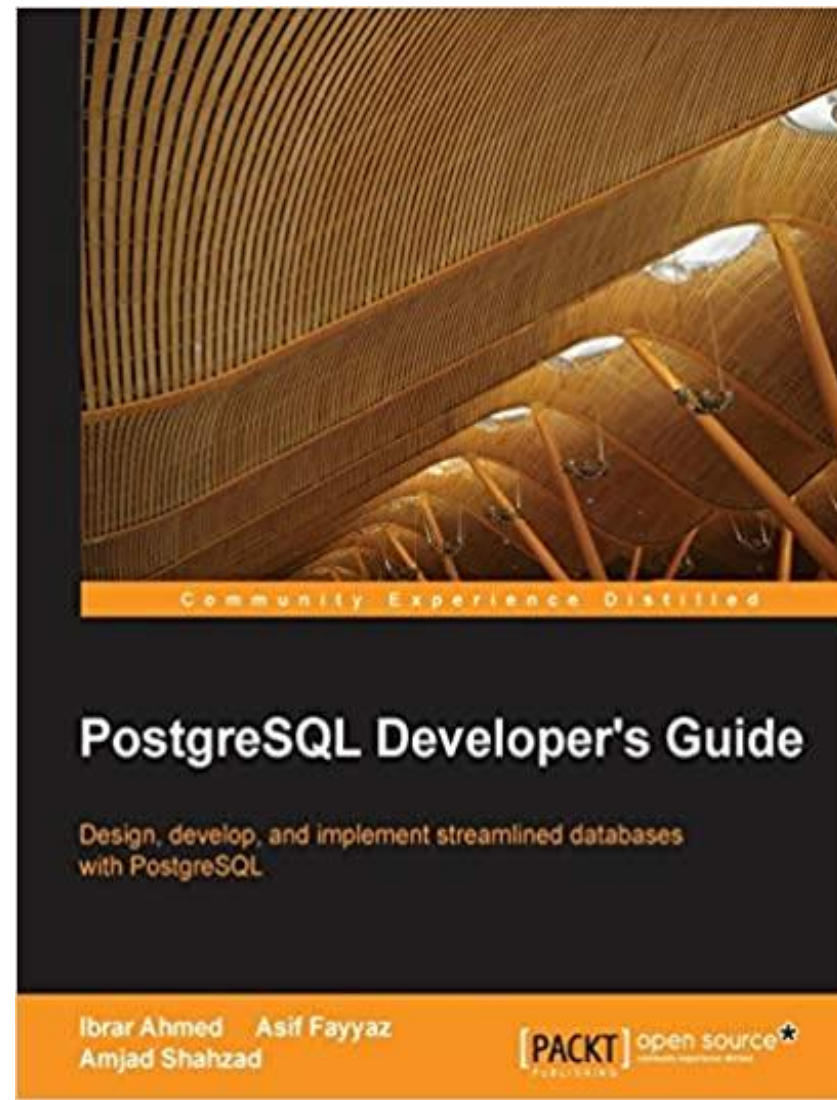
- **Default Setting and Range:** By default, it is set to 5 minutes (5min), with a permissible range from 30 seconds (30s) to 1 hour (1h). This setting can be adjusted to balance between performance and potential recovery time.
- **Performance Implications:** Longer intervals between checkpoints can improve performance by reducing the frequency of disk writes, but they may lead to longer recovery times in the event of a crash.
- **Impact on Recovery Time:** The time needed for recovery after a crash is influenced by the time since the last checkpoint. More recent checkpoints can significantly reduce recovery time.
- **Tuning for Workload:** Adjusting `checkpoint_timeout` requires considering the specific workload, disk performance, and acceptable recovery times, making it a vital parameter for database administrators to tune for optimal performance and reliability.

max_wal_size

The `max_wal_size` setting in PostgreSQL configures the maximum size of WAL (Write-Ahead Logging) files before a checkpoint is forced, influencing database performance and recovery processes. It determines the maximum amount of disk space that WAL files can consume between automatic checkpoints, affecting both performance during normal operations and recovery time after a crash.

- **Default and Adjustable Range:** The default value varies by PostgreSQL version, but it's set to allow a reasonable amount of WAL data accumulation, balancing performance with disk usage. Administrators can adjust this setting based on disk capacity and performance considerations.
- **Impact on Disk Space:** A larger `max_wal_size` allows more transactions to be logged before a checkpoint is triggered, potentially increasing disk space usage but can improve performance by reducing checkpoint frequency.
- **Recovery Implications:** While a larger WAL size can improve operational efficiency, it may lead to longer recovery times in the event of a system crash, as more WAL records might need to be processed to bring the database to a consistent state.
- **Tuning Considerations:** Adjusting `max_wal_size` is a tuning measure that should take into account the specific needs of your database workload, available disk space, and acceptable recovery times, aiming for an optimal balance between performance and reliability.

Heap / Index



about 178
using 178, 179
PQsetdbLogin function
about 178
using 178
prepared statements, executing
PQexecPrepared, using 186, 187
PQprepare, using 185, 186

Q

query, executing
about 184
PQexecParams, using 185
PQexec, using 184
query optimization
about 137
configuration parameters 153
cost parameters 141
EXPLAIN command 138
hints 151
query planning
about 149, 150
window functions 150, 151
query tree 102

R

range partition
about 124
constraint exclusion, enabling 129
index, creating on child tables 127
master table, creating 124, 125
range partition table, creating 125, 126
trigger, creating on master table 127, 128

rank() function

about 114
calling 114
row-level trigger 87
row_number() function
about 113
calling 113
rules
about 85
versus triggers 103, 104

S

schema_name parameter 225
self join 120
semi join 148
sequential scan 138, 142
set of variables, TriggerData
NEW 90
OLD 90
TG_OP 90
TG_TABLE_NAME 90
TG_WHEN 90
shared buffers parameter 155
single-column index
about 69
creating 69, 70
SQL commands, running
about 203
dynamic SQL, 206
host variables, using 205
values, obtaining from SQL 205
values, passing to SQL 205
SQL Communication Area (sqlca)
about 210
using 210-212
SQL file 216
SQL/MED (SQL/Management of External Data) 213
start up cost 138
statement-level trigger 87
status functions
PQresStatus, using 196
PQresultStatus, using 195
using 195

T

table partition
creating 123
TriggerData
about 86
set of variables 90
trigger function
about 85, 86
creating, with PL/pgSQL 90-92
defining 86
triggers
about 86
creating, in PL/Perl 96-98

[246]

Chapter 4

PostgreSQL comes with two main types of triggers: **row-level trigger** and **statement-level trigger**. These are specified with `FOR EACH ROW` (row-level triggers) and `FOR EACH STATEMENT` (statement-level triggers). The two can be differentiated by how many times the trigger is invoked and at what time. This means that if an `UPDATE` statement is executed that affects 10 rows, the row-level trigger will be invoked 10 times, whereas the statement-level trigger defined for a similar operation will be invoked only once per SQL statement.

Triggers can be attached to both tables and views. Triggers can be fired (or tables before or after any `INSERT`, `UPDATE`, or `DELETE` operation; they can be fired once per affected row, or once per SQL statement. Triggers can be executed for the `TRUNCATE` statements as well. When a trigger event occurs, the trigger function is invoked to make the appropriate changes as per the logic you have defined in the trigger function.

The triggers defined with `INSTEAD OF` are used for `INSERT`, `UPDATE`, or `DELETE` on the views. In the case of views, triggers fired before or after `INSERT`, `UPDATE`, or `DELETE` can only be defined at the statement level, whereas triggers that fire `INSTEAD OF` on `INSERT`, `UPDATE`, or `DELETE` will only be defined at the row level.

Triggers are quite helpful where your database is being accessed by multiple applications, and you want to maintain complex data integrity (this will be difficult with available means) and monitor or log changes whenever a table data is being modified.

The next topic is a concise explanation of tricky trigger concepts and behaviors that we discussed previously. They can be helpful in a database design that involves triggers.

Tricky triggers

In `FOR EACH ROW` triggers, function variables contain table rows as either a `NEW` or `OLD` record variable, for example, in the case of `INSERT`, the table rows will be `NEW`, for `DELETE`, it is `OLD`, and for `UPDATE`, it will be both. The `NEW` variable contains the row after `UPDATE` and `OLD` variable holds the row state before `UPDATE`.

Hence, you can manipulate this data in contrast to `FOR EACH STATEMENT` triggers. This explains one thing clearly, that if you have to manipulate data, use `FOR EACH ROW` triggers.

The next question that strikes the mind is how to choose between row-level `AFTER` and `BEFORE` triggers.

[87]

PostgreSQL Indexes



B-Tree

PostgreSQL default index
Based on B-Tree



Hash

PostgreSQL Index Method
Based on Hasing



Brin

Block Range Index



GIST

Generalized Search Tree



GIN

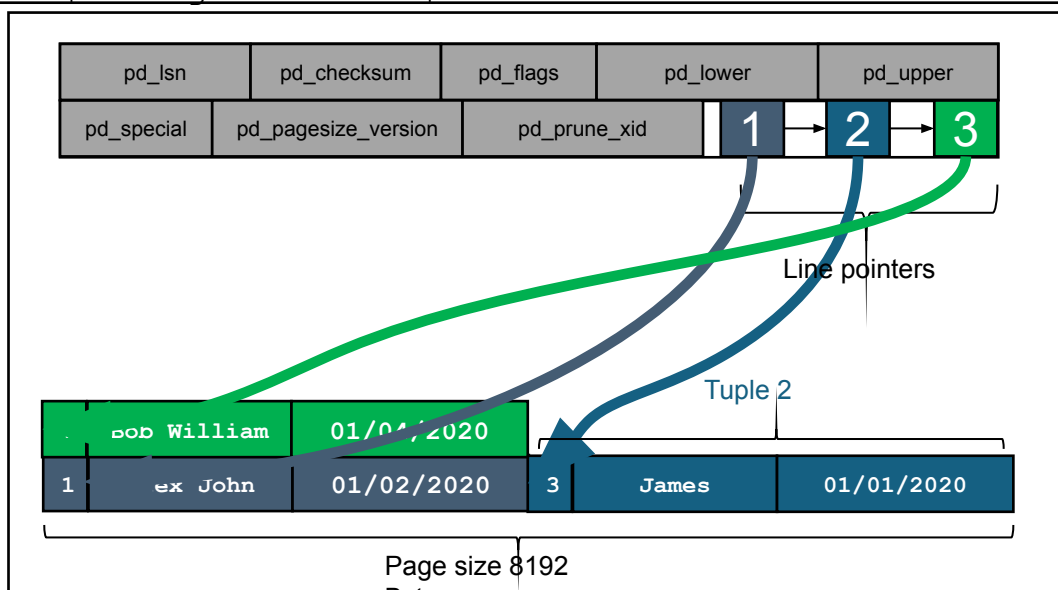
Generalized Inverted Index

Sequential Scan

```
SELECT * FROM admin WHERE dt <
'2021/04/01';
```

id	name	dt
3	James	2020-01-01
1	Alex Johns	2020-01-02
7	Bob William	2020-01-04
8	Charli	2020-01-01
6	David	2020-08-02
9	Benjamin	1990-01-02

Block 0

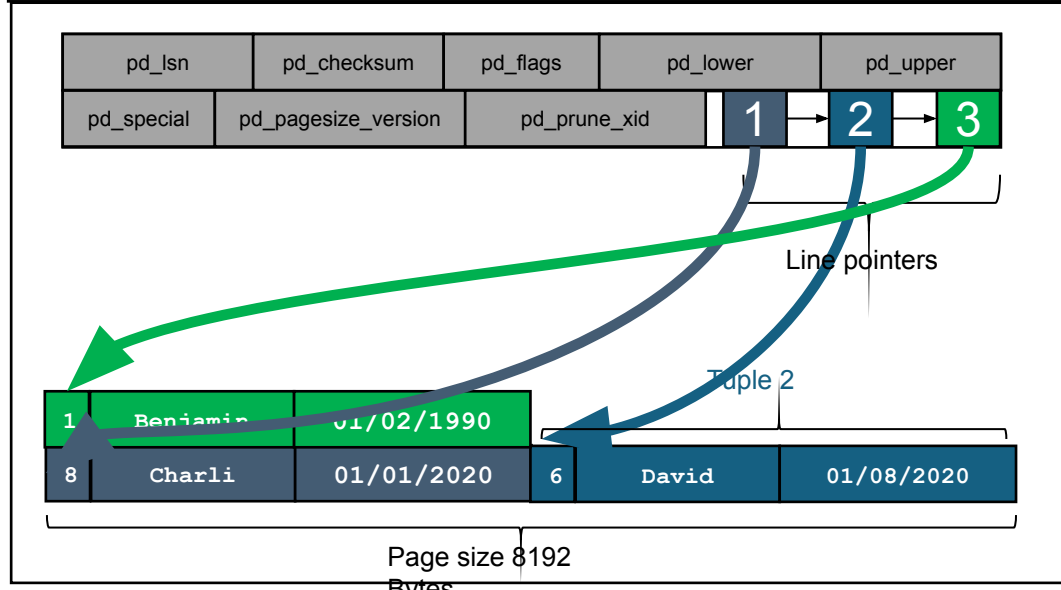


```
SELECT ctid, * FROM admin WHERE id =
8;
```

ctid	id	name
(1,0)	16	Charli

(1 rows)

Block 1

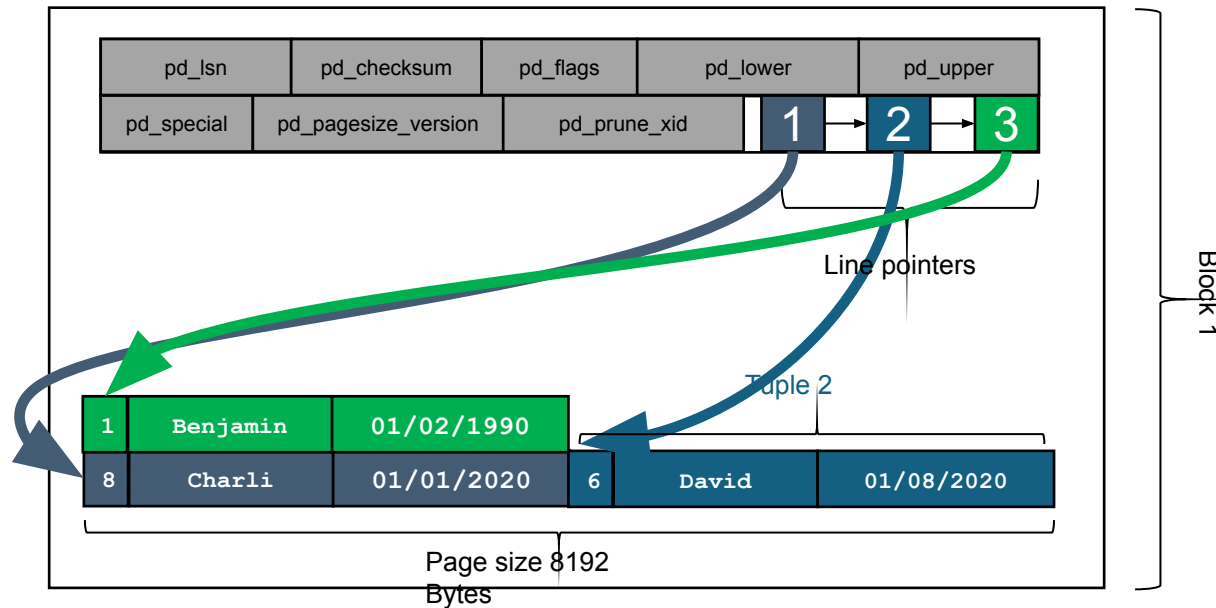
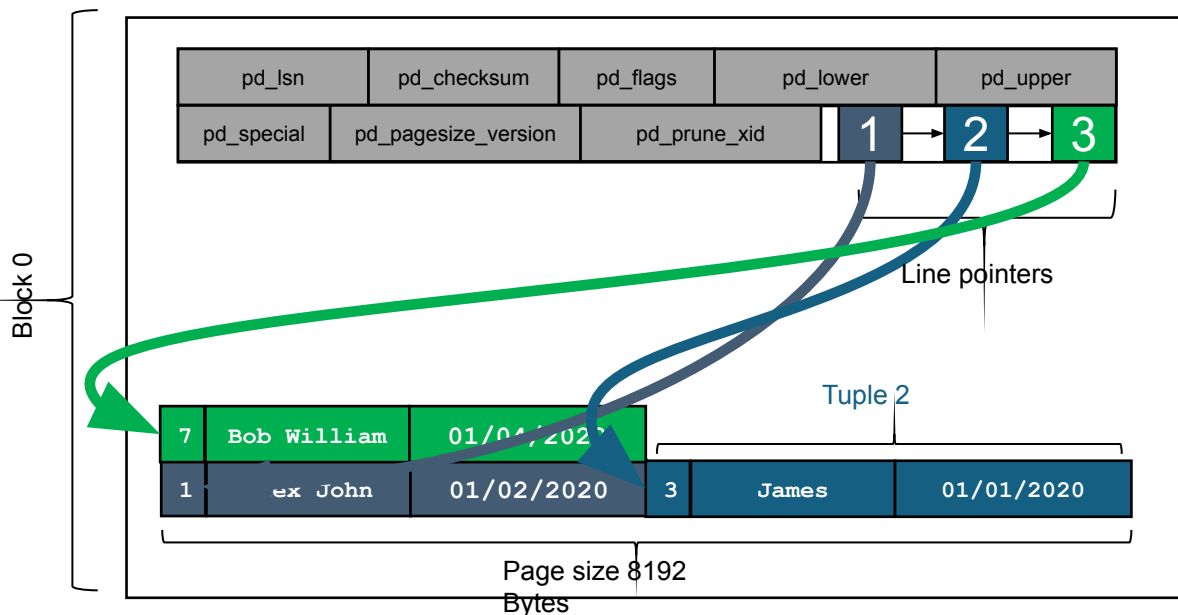
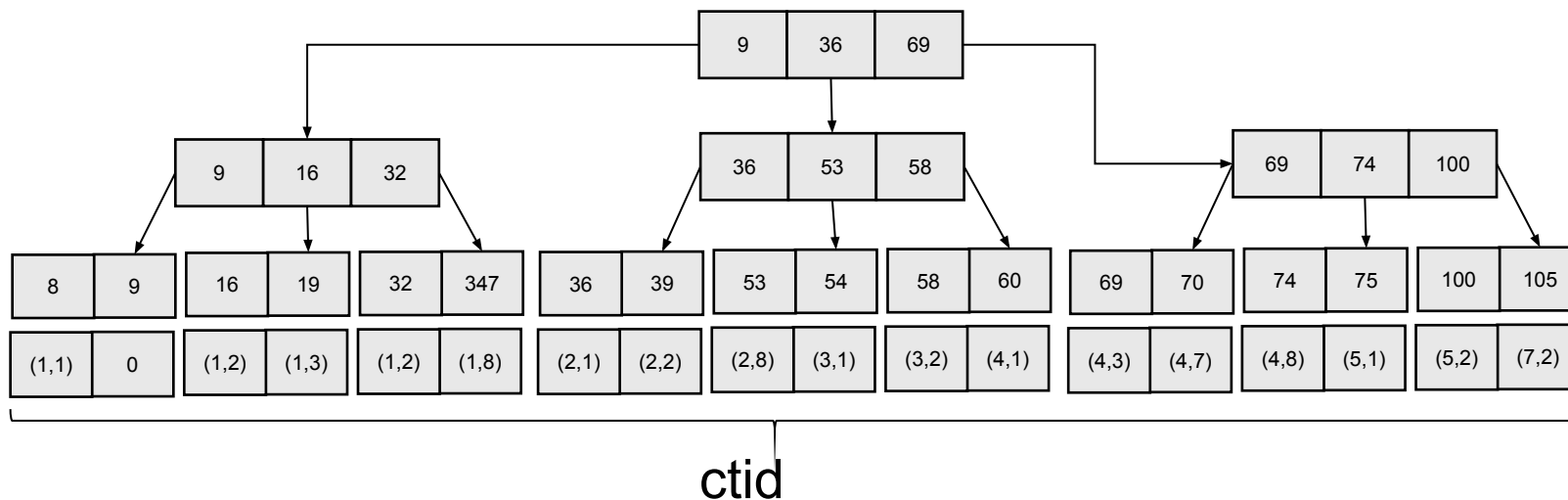


B-Tree Index

```
SELECT id, name FROM
```

```
admin WHERE id = 8;
```

```
id | name
---+-----
 8 | Charli
(1 rows)
```



Hash Index

```
SELECT id, name FROM admin WHERE name
LIKE 'Alex Johns';
```

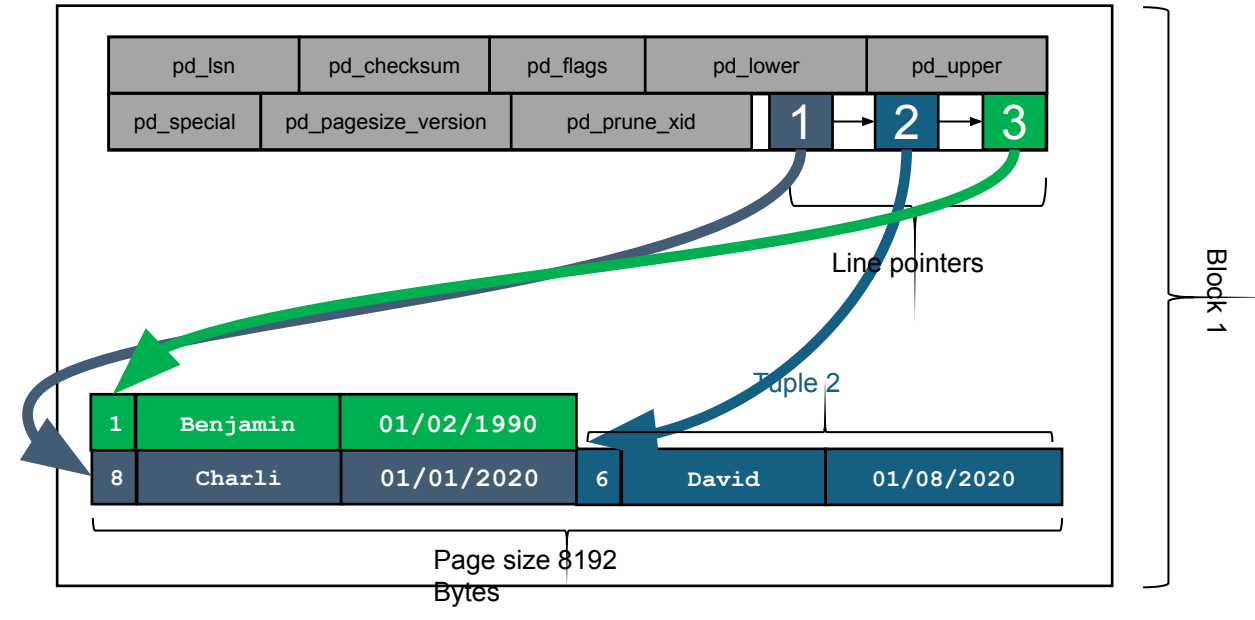
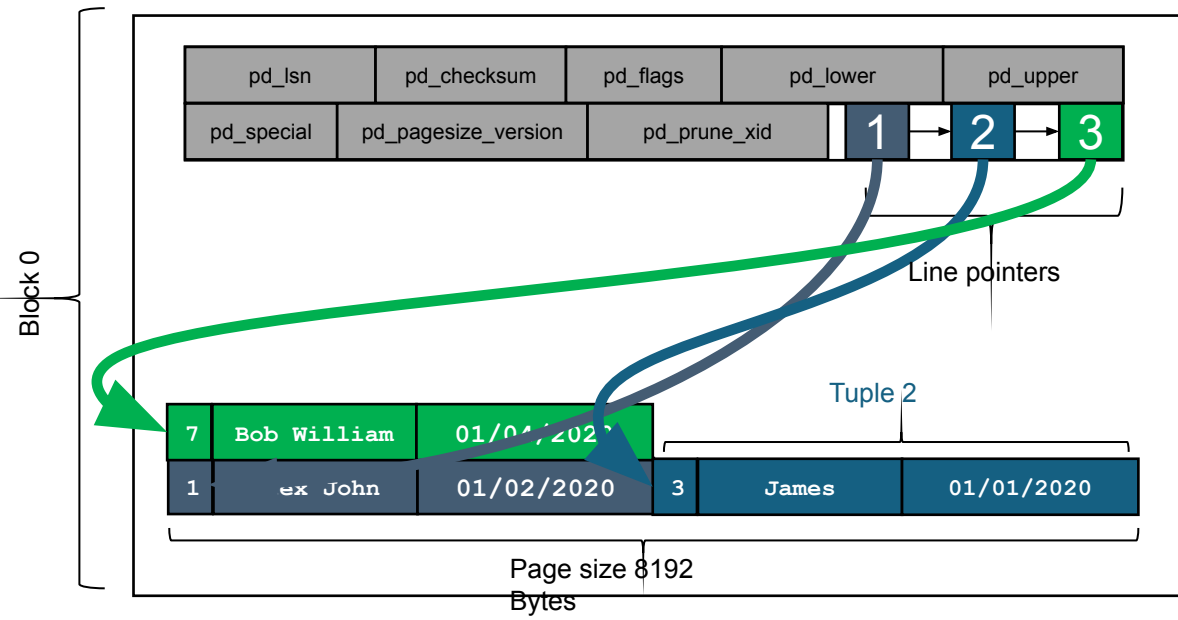
```
id | name
---+-----
 16 | Alex Johns
(1 rows)
```

- Alex Johns
- James
- Bob William
- Charli
- David
- Benjamin

$F_h(x)$

0000	0,0
0010	0,1
0011	0,2
0100	1,0
0101	1,1
0111	1,2

CTID



02

Horizontal Scalability



Horizontal Scalability in PostgreSQL

Horizontal scalability in PostgreSQL involves expanding database capacity by adding more servers or instances, allowing the system to distribute load and data across multiple machines.

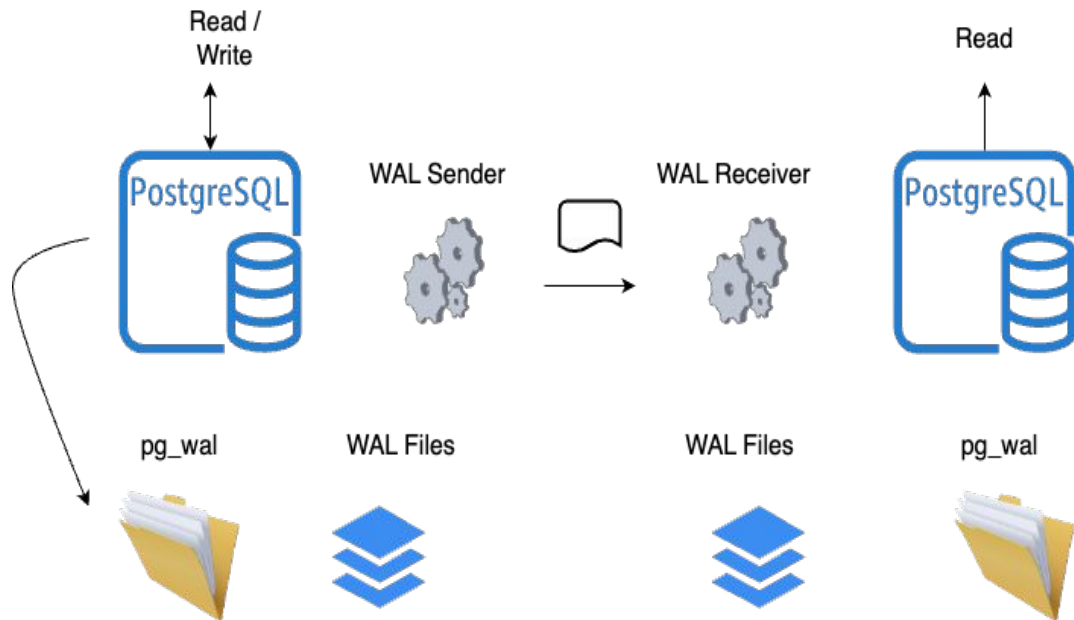
- **Replication:** PostgreSQL supports several replication methods, including streaming replication and logical replication, to synchronize data across multiple servers, ensuring data consistency and high availability.
- **Partitioning:** Data partitioning divides large tables into smaller, more manageable pieces across different servers, improving query performance and data management efficiency.
- **Sharding:** While not natively supported in PostgreSQL core, sharding can be implemented through extensions or application-level sharding. This involves distributing data across multiple databases to spread the load.
- **Load Balancing:** Using load balancers or connection poolers like pgCat, PgBouncer or Pgpool-II can distribute incoming connections and queries across multiple PostgreSQL servers, optimizing resource usage and response times.
- **High Availability Clusters:** Setting up PostgreSQL in a high availability (HA) cluster configuration ensures that the database service remains available even in the event of server failure, through failover mechanisms.
- **Scalability Tools and Extensions:** Utilizing tools and extensions like Spock, Citus, Postgres-XL and Patroni.

Replication

- PostgreSQL supports several replication methods, including logical, streaming, and snapshot, each catering to different requirements and use cases.
- Streaming Replication: A popular method for real-time replication, streaming replication involves a primary server sending data changes to one or more standby servers. This method is helpful for high availability and load balancing.
- Logical Replication: Allows selective data replication at the table level, allowing the flexibility to replicate only specific tables or rows. It's beneficial for upgrading systems with minimal downtime and integrating data across different PostgreSQL versions.
- Synchronous vs. Asynchronous Replication: In synchronous replication, transactions must be confirmed by both the primary and standby servers before being committed, ensuring data consistency but potentially affecting performance. Asynchronous replication, while faster, does not guarantee immediate consistency across servers.

Physical Replication

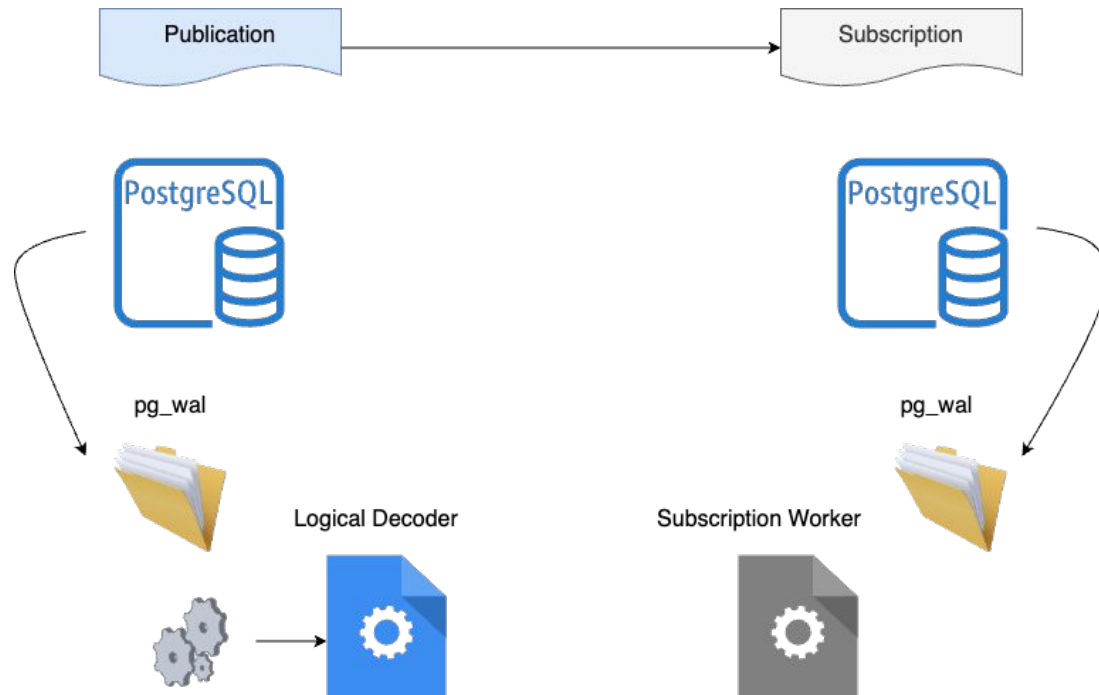
Physical replication in PostgreSQL is a method for copying and synchronizing data from a primary server to standby servers in real-time.



- Real-time transfer of WAL records from a primary to standby servers to ensure data consistency and up-to-date replicas.
- Standby servers can run in hot standby mode, allowing them to handle read-only queries while replicating changes.
- Configurable as either synchronous, for strict data integrity, or asynchronous, for improved write performance.
- Facilitates automatic failover by promoting a standby to primary in case of primary server failure.

Logical Replication

Logical replication is method of copying data objects and changes based on replication identity.



- Logical replication is method of copying data objects and changes based on replication identity.
- Provides fine grained control over data replication and security.
- Publisher / Subscriber model - one or more subscriber subscribe to one or more publisher.
- Copy data in format that can be interpreted by other systems using logical decoding plugins.
- Publication is set of changes generated from a table or group of tables.
- Subscription is the downstream end of logical replication.

Replication

Partitioning in PostgreSQL is a powerful feature for managing large tables by splitting them into smaller, more manageable pieces, called partitions.

- **Types of Partitioning:** PostgreSQL supports two main types of partitioning:
 - Range partitioning: where data is divided based on a range of values.
 - List partitioning: where data is divided based on a list of specific values.
 - Hash Partitioning
- **Automatic Partition Management:** Starting from PostgreSQL 10, it introduced declarative partitioning, allowing the database to automatically manage partitions based on the partition key, simplifying the creation and maintenance of partitioned tables.
- **Improved Performance:** Partitioning can significantly improve query performance, especially for operations that can be limited to a few partitions, reducing the amount of data that needs to be scanned.
- **Indexing and Constraints:** Each partition can be indexed and constrained independently, allowing for more efficient indexing strategies and constraint enforcement, which further enhances query performance and data integrity.
- **Partition Pruning:** PostgreSQL's optimizer can perform partition pruning, automatically excluding irrelevant partitions from query execution plans based on the query conditions, which can drastically reduce the amount of data processed.

Balancing Vertical and Horizontal Scalability

- Balancing vertical scalability allows for adding resources to a single server, optimizing performance for intensive tasks and ensuring consistent response times.
- Horizontal scalability suits scenarios with high data volume and reads, enabling distributed processing of queries across multiple servers for improved throughput.
- Strategies include vertical scaling for OLAP workloads, horizontal scaling for read-heavy applications, and hybrid approaches for a flexible and efficient PostgreSQL deployment.

Best Practices for PostgreSQL Scalability

- Implement data partitioning to distribute data across multiple disks or tables, improving query performance and allowing for easier management of large datasets.
- Utilize clustering to create a group of interconnected servers that work together, enhancing both performance and fault tolerance in PostgreSQL environments.
- Employ replication to create redundant copies of the database, ensuring high availability and disaster recovery while also offloading read-heavy workloads for better scalability.

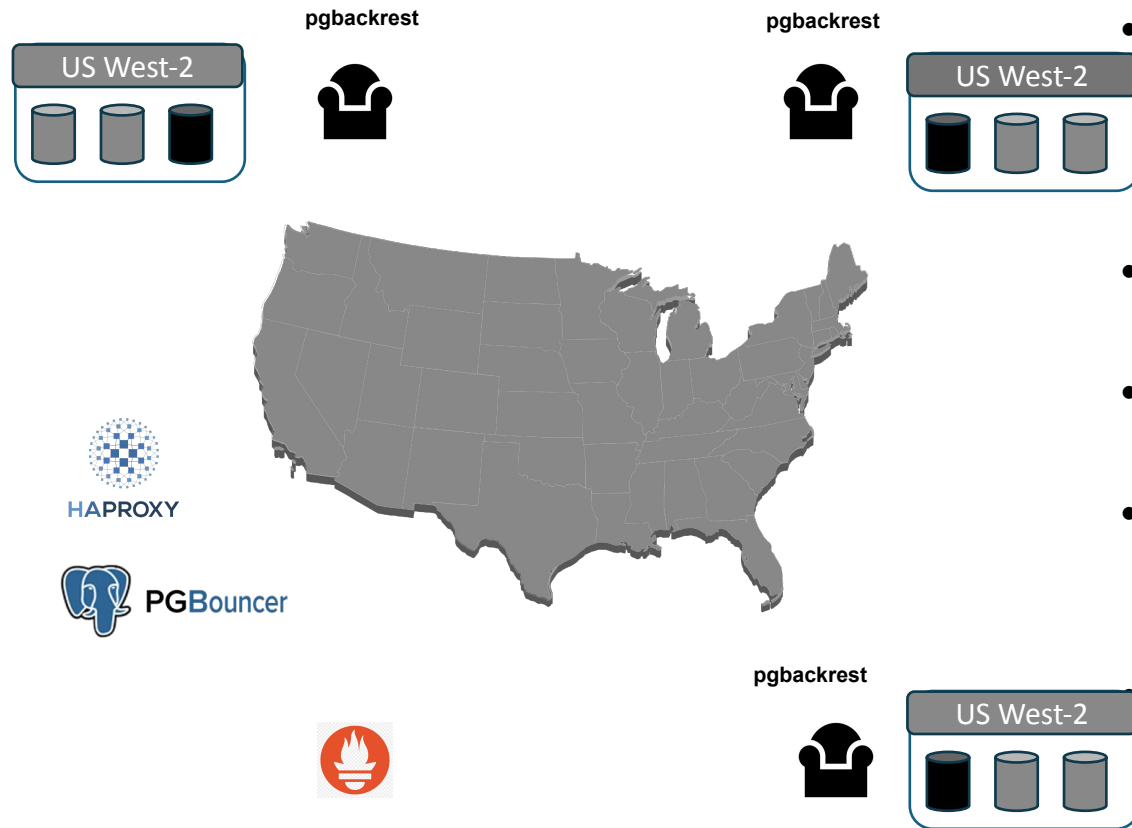
Future Trends in PostgreSQL Scalability

- Advancements in cloud-native databases like Amazon Aurora and Google Cloud Spanner offer PostgreSQL users scalable and highly available solutions.
- Containerization technologies such as Docker and Kubernetes streamline deployment, scaling, and management of PostgreSQL instances in modern IT environments.
- PostgreSQL is evolving by embracing distributed computing methods like sharding and replication to meet the scalability demands of data-intensive applications.

High Availability

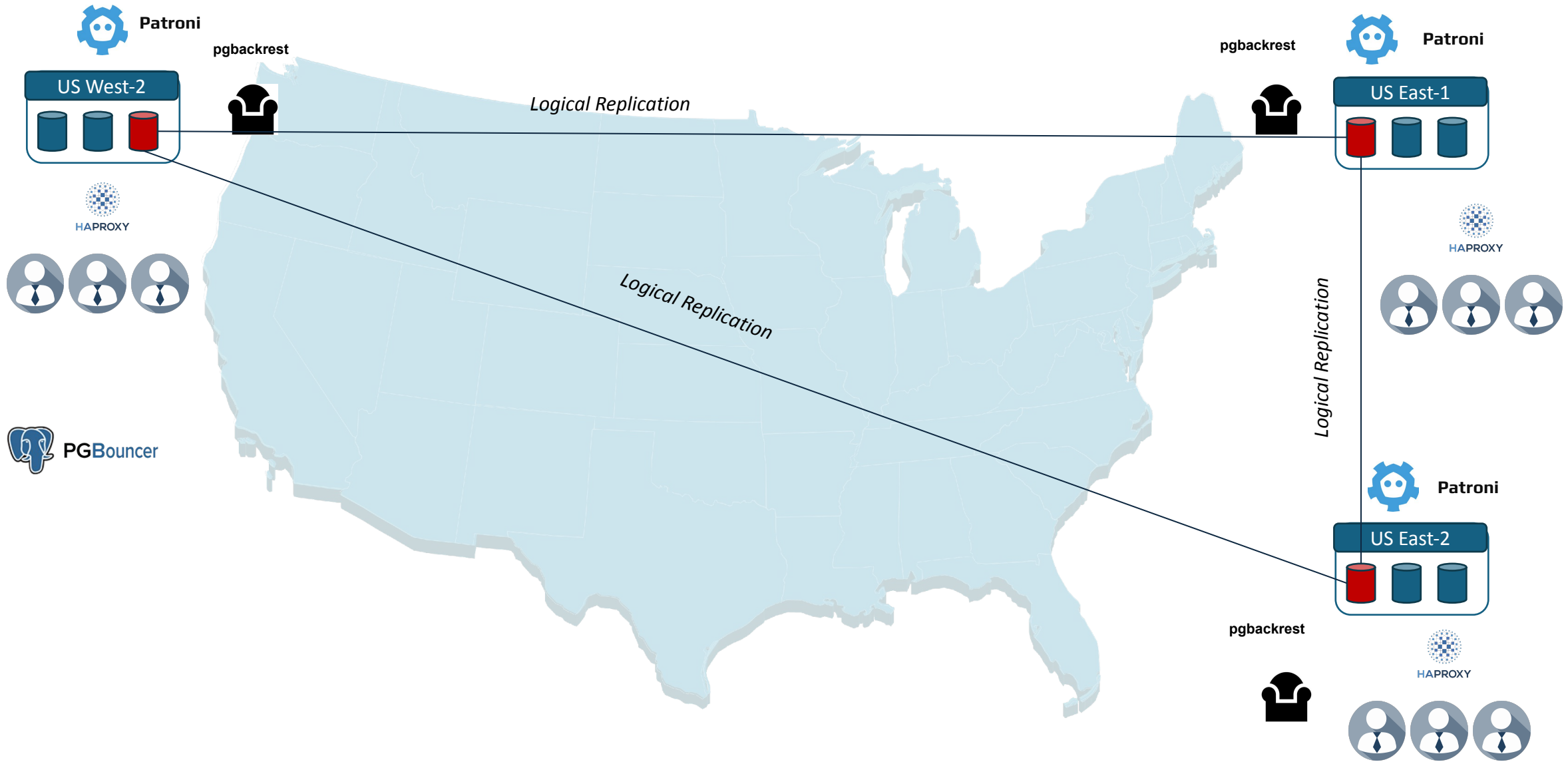
- High availability in PostgreSQL ensures the database is operational and accessible without significant downtime.
- Minimize downtime by implementing redundant systems and automated failover processes for uninterrupted service.
- Ensure data integrity through synchronous replication and high availability features to prevent data loss or corruption.
- Provide seamless failover mechanisms for continuous service by quickly transitioning to standby servers during failures.

High Availability in PostgreSQL



- High availability in PostgreSQL ensures the database is operational and accessible without significant downtime.
- **Replication:** Streaming replication is used to create standby servers that are continuously updated with data from the primary server.
- **Failover:** Automatic failover process to switch to a standby server in case the primary server fails.
- **Load Balancing:** Distribution of queries across multiple servers to improve performance and distribute the workload.
- **Connection Pooling:** Management of database connections to optimize resource usage and improve performance.
- **Monitoring and Management:** Continuous monitoring of database servers to detect and respond to issues promptly, often using tools like pgBouncer and pgpool.
- **Backup and Recovery:** Regular backups and robust recovery plans to protect against data loss and ensure quick service restoration.
- **Clustering:** Grouping multiple servers to work as a single system, providing redundancy and improving availability.

High Availability in PostgreSQL



Questions

Code is like clay; in the hands of a skilled craftsman, it can be molded into something that stands the test of time. Remember, the art is not in writing code, but in crafting solutions that endure. Let's build not just for today, but for the future.



Ibrar Ahmed

