SquadStack

PGConf India

Hemendra Sharma

Platform Engineer @SquadStack

https://www.linkedin.com/in/hemendrasharma04/

SquadStack

PGConf India

Postgres as a Message Broker to handle asynchronous task at scale

Agenda

1.	Our Use case - **Why we use postgres as a message broker?**
2.	Problems we faced while using Amazon MQ (RMQ)
3.	PgQueue introduction and workflow
4.	How task processed in PgQueue
5.	PgQueue Components
6.	Issues we faced in PgQueue
7.	Conclusion
8.	END

1. Our Use Case

Why we use postgres as a message broker?

2. Problems we faced while using Amazon MQ (RMQ)

1. RMQ Workers RAM Overload Leading to Degraded Performance
2. Limited observability
3. Cost inefficiency
4. Limited customization

3. PgQueue Introduction and workflow

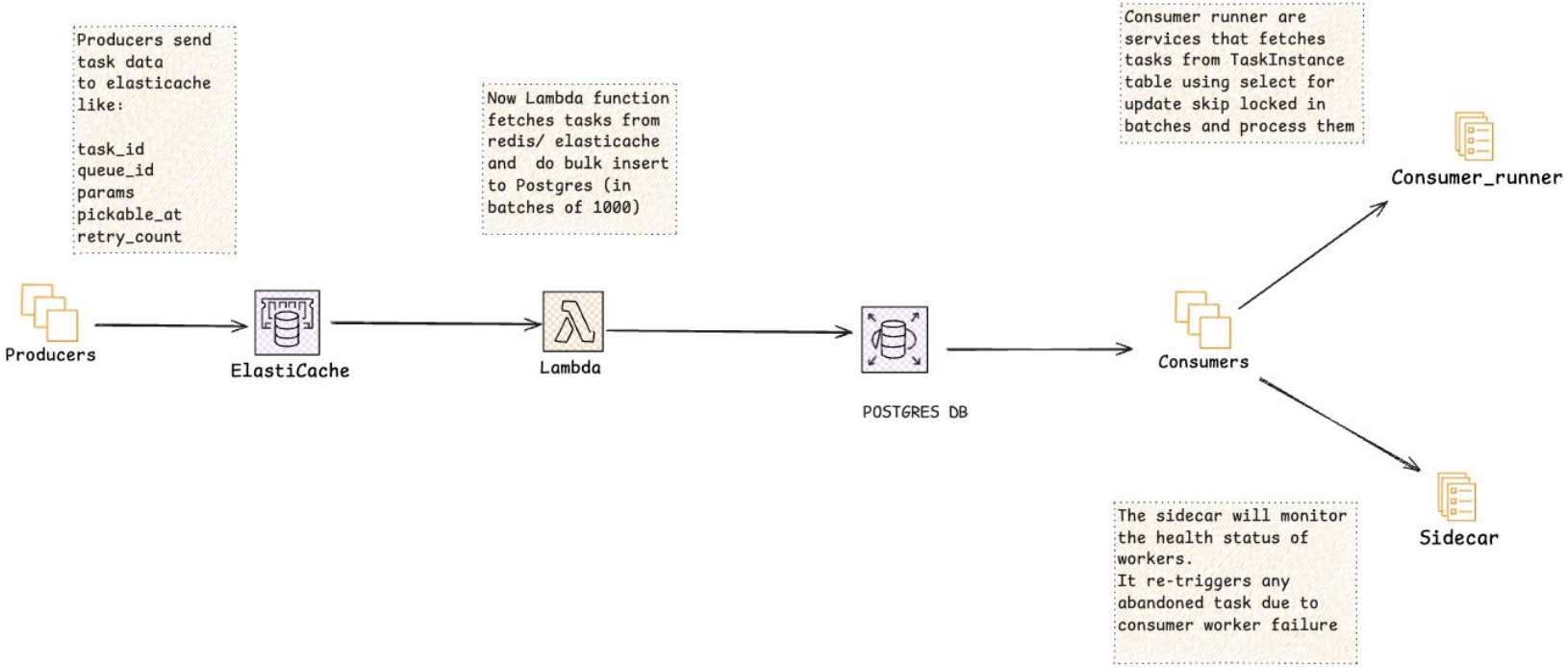
- PgQueue: Custom Task Queue Built with PostgreSQL and Django

- A reliable, scalable, and efficient task queue for asynchronous task processing. Features built-in producers, consumers, and concurrency control.

- Supports periodic tasks and ETA-based tasks with retry mechanism .

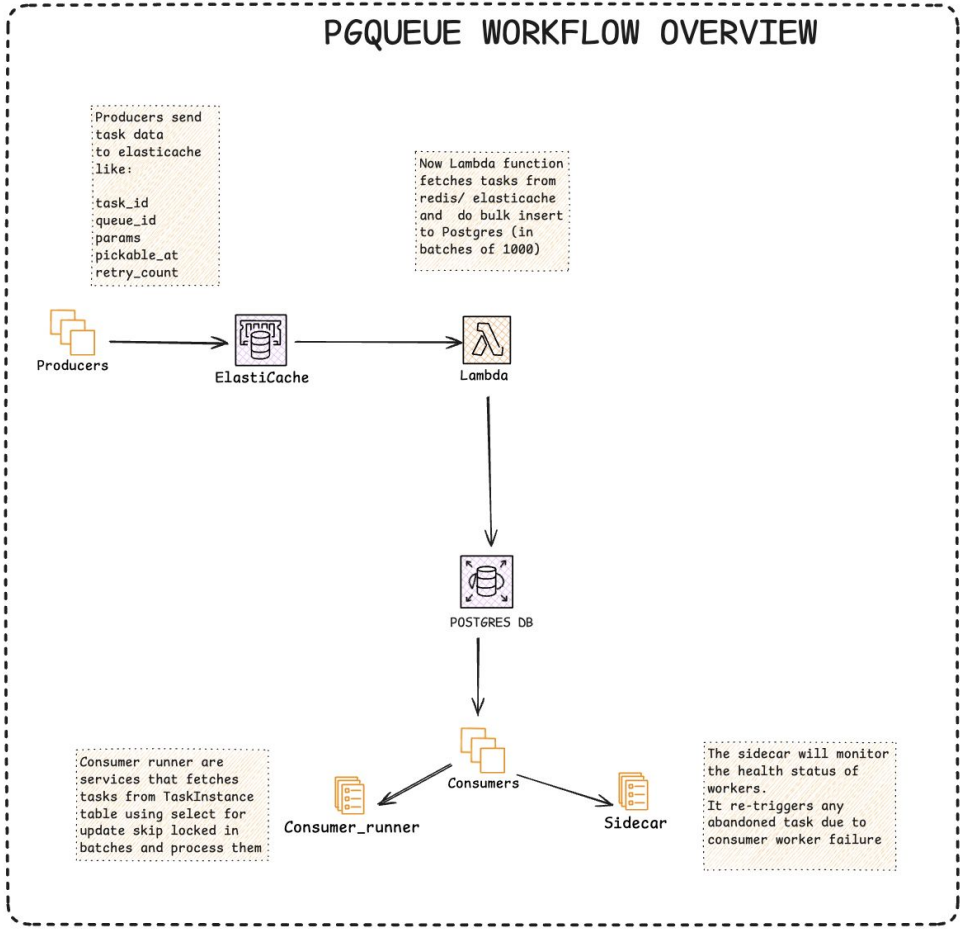
- Includes a sidecar process to monitor and manage PgQueue consumer states.

PGQUEUE WORKFLOW OVERVIEW

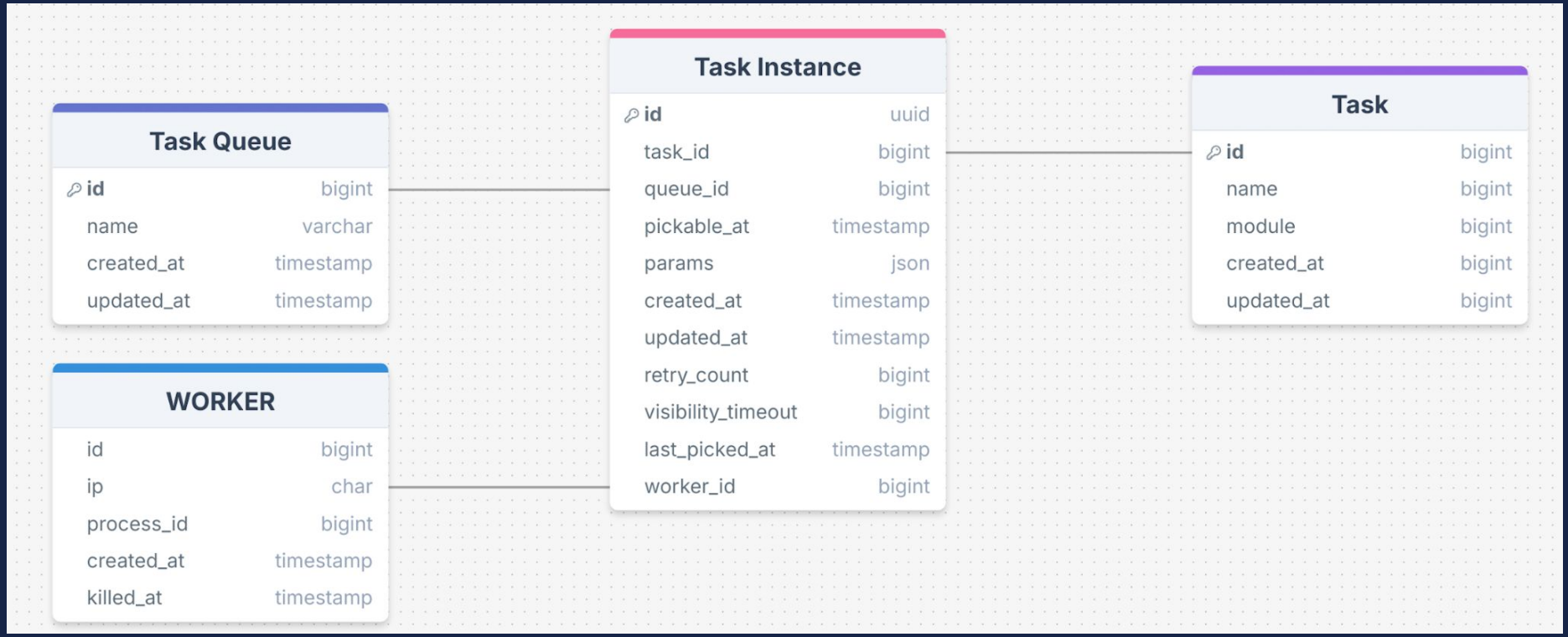


4. How task processed in PgQueue

1. **Task Creation** : Producer (backend server) creates tasks.
2. **Data Transfer** : Task data is sent to Redis.
3. **Lambda Trigger** : Lambda processes and transfers data to PostgreSQL in bulk.
4. **Task Fetching** : Consumers retrieve tasks in batches of 10 using **SELECT FOR UPDATE SKIP LOCKED**.
5. **Task Execution** : Tasks are processed, and their status is updated in the database.



SquadStack


Task Queue
id bigint
name varchar
created_at timestamp
updated_at timestamp
WORKER
id bigint
ip char
process_id bigint
created_at timestamp
killed_at timestamp
Task Instance
id uuid
task_id bigint
queue_id bigint
pickable_at timestamp
params json
created_at timestamp
updated_at timestamp
retry_count bigint
visibility_timeout bigint
last_picked_at timestamp
worker_id bigint
Task
id bigint
name bigint
module bigint
created_at bigint
updated_at bigint

5.PgQueue Components

- PRODUCER
- CONSUMER
- CRON RUNNER
- SIDECAR

PRODUCER

- **Task Creation** : Producers push tasks to Redis with details like module, queue, pickable_at, and parameters.
- **Fail-Safe Mechanism** : If Redis fails, tasks are directly created in the database to ensure no data loss.
- **Retry Logic** : Built-in retry function allows tasks to be retried effectively in case of failures.
- **Reliability** : Ensures robust task handling, minimizes data loss, and gracefully manages failures.

```python
@pgqueue_task(bind=True,queue=PGQUEUE_TEST_QUEUE)
def test_pgqueue_task():
    try:
        fetch_data()
        logger.info("running test_pgqueue_task")
    except Exception as e:
        self.retry(countdown=10)



test_pgqueue_task.apply_async(countdown=10)
# or with .apply_async(eta=timezone.now() + timedelta(seconds=10))


@pgqueue_cron(run_every=crontab(minute="*/5"), queue=PGQUEUE_TEST_QUEUE)
def test_pgqueue_cron():
    logger.info("running test_pgqueue_cron")
```

CONSUMER

Batch Processing : Multiple workers process tasks in batches (default size: 10) from specified queues.

Safe Task Handling : Ensures reliable task claiming and processing using database transactions with SELECT FOR UPDATE SKIP LOCKED.

Queue Flexibility : By default, processes tasks from all queues but can be configured for specific queues.

So we have two types of services for running consumer_runner

1. Common consumer runner service : In this service we fetch tasks from all the queues.

2. Dedicated Consumer Services for Specific Queues : In this service we fetches task from specific queues by passing the queue names

Fetches from all queues

```
task_ids = list(
    TaskInstance.objects.filter(status=STATUS_PENDING, pickable_at__lte=now_time)
    .exclude(queue__name__in=PGQUEUE_CONSUMER_QUEUE_EXCLUSIONS)
    .filter(queue__isnull=False)
    .select_for_update(skip_locked=True)[:batch_size]
    .values_list("id", flat=True)
)
```

Fetches from specific queues

```
queue_ids = TaskQueue.get_queue_ids_by_name_list(queues)

task_ids = list(
    TaskInstance.objects.filter(
        status=STATUS_PENDING,
        queue_id__in=queue_ids,
        pickable_at__lte=now_time,
    )
    .select_for_update(skip_locked=True, no_key=True)[:batch_size]
    .values_list("id", flat=True)
)
```

CRON RUNNER

Cron Runner: Manages Periodic Tasks in PgQueue

The Cron Runner handles periodic tasks in PgQueue using Celery Beat and a custom scheduler.

Tasks are defined with a run_every field, specifying execution intervals (e.g., every 10 minutes, hourly).

How It Works

Task Scheduling : Celery Beat uses the PeriodicTask model to schedule tasks based on the run_every interval.

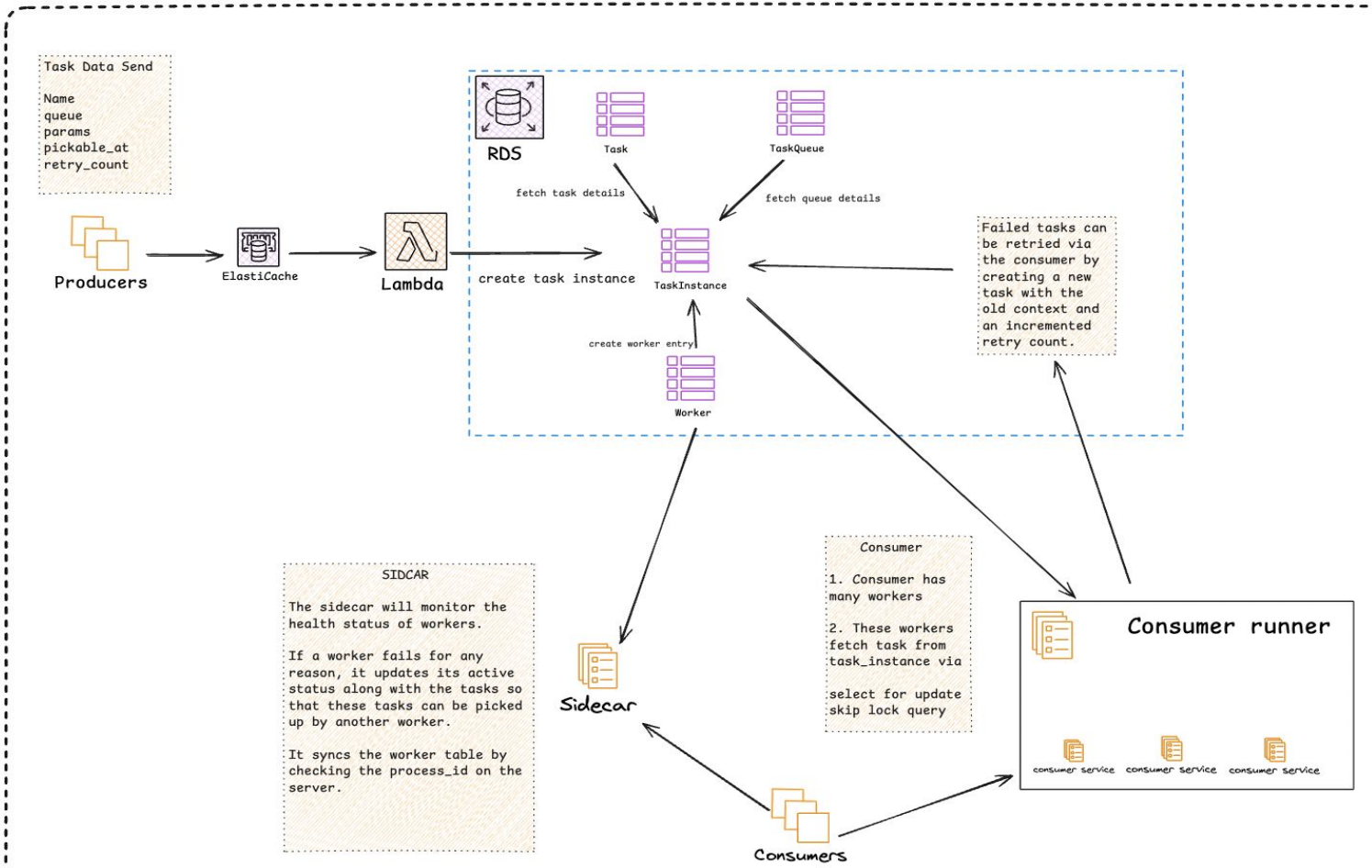
Task Queuing : When a task is due, it is sent to Redis as a pending task.

Buffer Handling : The Cron Runner picks up the task from Redis and sends it to a buffer in Redis.

Task Execution : Lambda fetches the data from the buffer and creates the task instance for execution.

SIDECAR

- **Monitoring and Management**: The sidecar process monitor consumer processes running on a host, ensuring they function correctly.
- **Handling Inactive Tasks**: If a consumer becomes inactive, the sidecar marks it as inactive and resets its tasks for reprocessing to prevent stuck jobs.
- **Process Detection**: he sidecar queries the database for active workers and compares their stored process IDs with running PIDs, marking any mismatched workers as inactive


Task Data Send
Name
queue
params
pickable_at
retry_count
RDS
Task
TaskQueue
fetch task details
fetch queue details
Producers
ElastiCache
Lambda
create task instance
TaskInstance
Failed tasks can be retried via the consumer by creating a new task with the old context and an incremented retry count.
create worker entry
Worker
SIDCAR
The sidecar will monitor the health status of workers.
If a worker fails for any reason, it updates its active status along with the tasks so that these tasks can be picked up by another worker.
It syncs the worker table by checking the process_id on the server.
Sidecar
Consumer
1. Consumer has many workers
2. These workers fetch task from task_instance via
select for update skip lock query
Consumer runner
consumer service
consumer service
consumer service
Consumers

6. Issues we faced in PgQueue

1. High API response time: API response times increased significantly due to slow database read and write queries.

2. Task starvation : Due to getting too much tasks in a single queue , most of the consumers were fetching tasks from that queue leaving other tasks in different queues starve

3 . Lock Manager wait event : The database experienced high load due to contention in the lock manager

High API Response Time

- API response times increased significantly due to slow database read and write queries.

- This was caused by heavy load on the database from both task insertion and task processing.

Analysis

1. Inefficient Write Operations : Initially, tasks were written to the database one by one , resulting in high contention and frequent locking.

2. **Lock Contention (Lock:transactionid)** : Row-level lock conflicts occurred when multiple transactions tried to access or modify the same rows. This was because of select for update command

3. **High Concurrency** : Many workers fetching and processing tasks simultaneously led to contention.

How Did We Solve This?

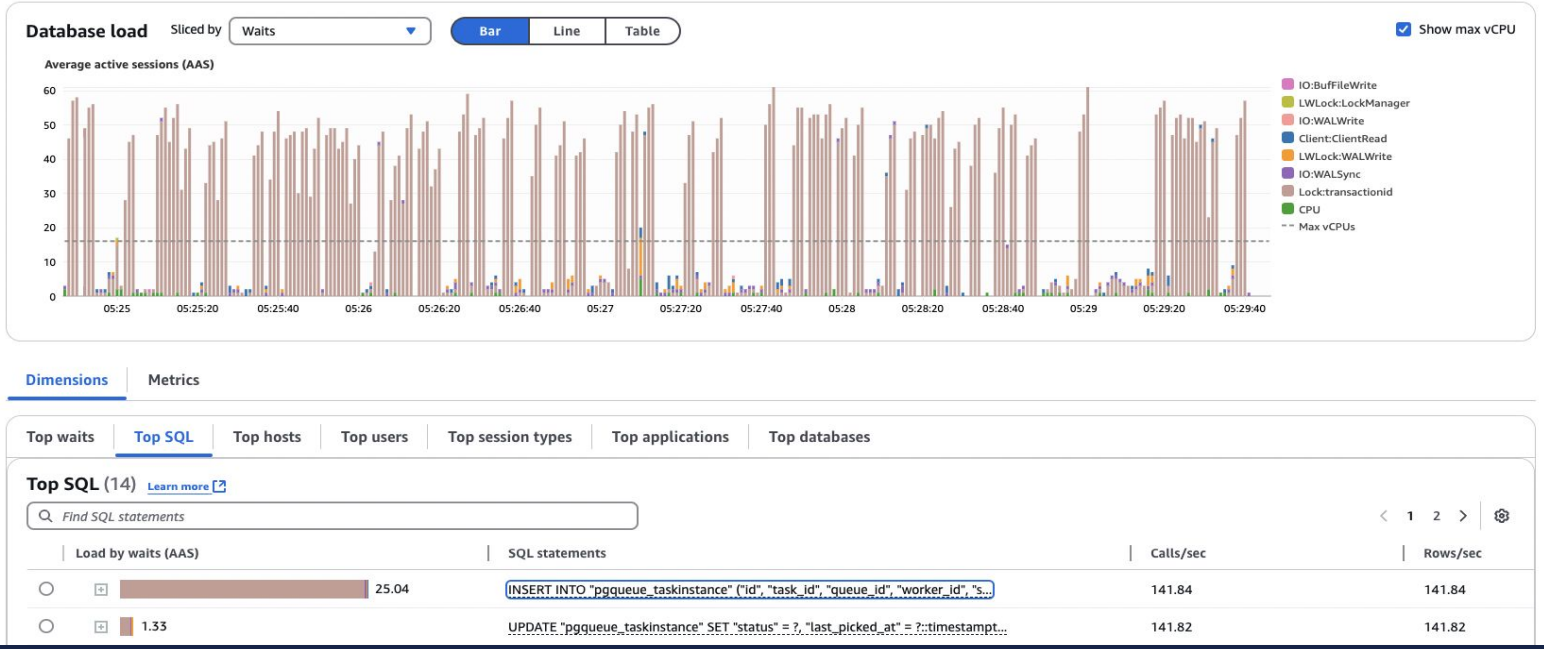
Redis as a Buffer

We introduced Redis as an intermediate buffer for task writes, improving performance and reducing contention.

Bulk Writes: All write queries are now batched, minimizing individual transactions.

Reduced Lock Contention: Bulk inserts have significantly decreased row-level lock conflicts.

Efficient Task Fetching: Workers using SELECT FOR UPDATE SKIP LOCKED now experience less contention.


Database load
Sliced by
Waits
Bar
Line
Table
Show max vCPU
Average active sessions (AAS)
60
50
40
30
20
10
0
05:25
05:25:20
05:25:40
05:26
05:26:20
05:26:40
05:27
05:27:20
05:27:40
05:28
05:28:20
05:28:40
05:29
05:29:20
05:29:40
IO:BufFileWrite
LWLock:LockManager
IO:WALWrite
Client:ClientRead
LWLock:WALWrite
IO:WALSync
Lock:transactionid
CPU
Max vCPUs
Dimensions
Metrics
Top waits
Top SQL
Top hosts
Top users
Top session types
Top applications
Top databases
Top SQL (14)
Learn more
Find SQL statements
1
2
Load by waits (AAS)
SQL statements
Calls/sec
Rows/sec
25.04
INSERT INTO "pgqueue_taskinstance" ("id", "task_id", "queue_id", "worker_id", "s...
141.84
141.84
1.33
UPDATE "pgqueue_taskinstance" SET "status" = ?, "last_picked_at" = ?::timestampt...
141.82
141.82

SquadStack

Task Starvation

Task starvation occurs when tasks in low-volume queues are delayed because consumers are occupied processing tasks from high-volume queues.

Scenario : If a queue receives a large number of tasks (e.g., 1–2 lakhs), consumers may focus on processing those tasks, leaving other queues with fewer tasks unattended.
This results in starvation of tasks in lower-priority or lower-volume queues.

How Did We Solve This?

- To address this issue, we introduced a feature to exclude specific high-volume queues from common consumer runner and run them separately.

- This isolates high-volume queues and doesn't interfere with other queues

```
PGQUEUE_CONSUMER_QUEUE_EXCLUSIONS = ["high_write_queue1", "high_write_queue2"]

queue_ids = TaskQueue.get_queue_ids_by_name_list(PGQUEUE_WORKER_QUEUE_EXCLUSIONS)

task_ids = list(
    TaskInstance.objects.filter(status=STATUS_PENDING, pickable_at__lte=now_time)
    .exclude(queue_id__in=queue_ids)
    .filter(queue__isnull=False)
    .select_for_update(skip_locked=True, no_key=True)[:batch_size]
    .values_list("id", flat=True)
)
```

Lock Manager wait event

What is the Lock Manager?

Manages concurrent access to database resources (tables, rows, indexes).
Ensures data consistency by enforcing locking rules.

Two Locking Mechanisms

Fast Path Mechanism : Lightweight locks managed locally by each backend (up to 16 unique locks per backend defined by FP_LOCK_SLOTS_PER_BACKEND) .

Centralized Lock Manager : Manages locks across all backend processes using shared memory and lightweight locks (LWLocks).

Analysis : LWLock:lock_manager

Exceeding Limits : In PgQueue, when fetching tasks in a transaction, exceeding 16 unique relation locks (tables or indexes) forces the use of the centralized lock manager.

Performance Impact : High contention in the centralized lock manager triggers the LWLock:lock_manager wait event, increasing CPU load and degrading database performance.

BEFORE OPTIMIZATION

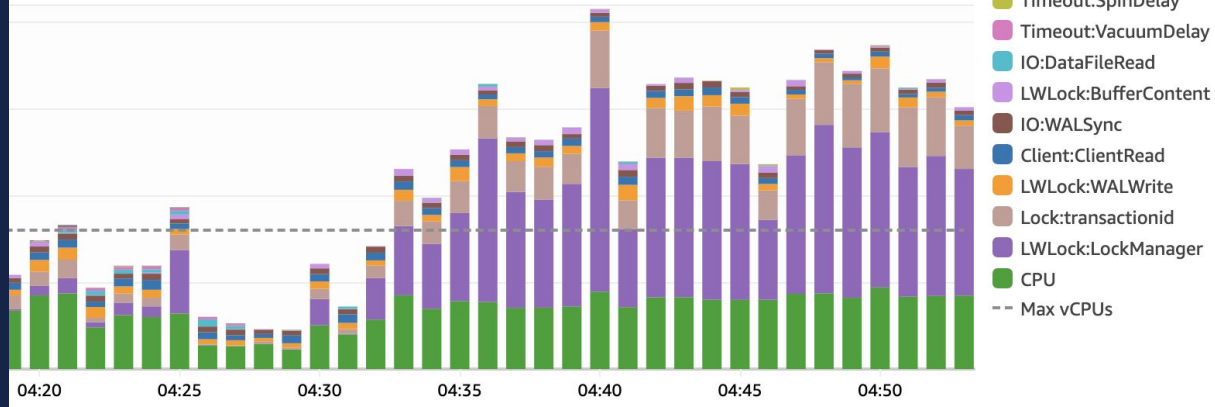
SquadStack

5m 30m 1h 3h 12h Custom UTC timezone Auto refresh

Modify retention tier

Line Table

Show max vCPU



```
SELECT
    pid,
    fastpath,
    COUNT(DISTINCT relation) AS distinct_relation_count
FROM
    pg_locks
WHERE
    locktype = 'relation'
    AND pid != pg_backend_pid()
GROUP BY
    pid, fastpath
ORDER BY
    pid, fastpath DESC;, [])
}
```

```
  pid  | fastpath | distinct_relation_count
-------+----------+-------------------------
 23285 | t        |                      16
 23285 | f        |                       6
(2 rows)
```

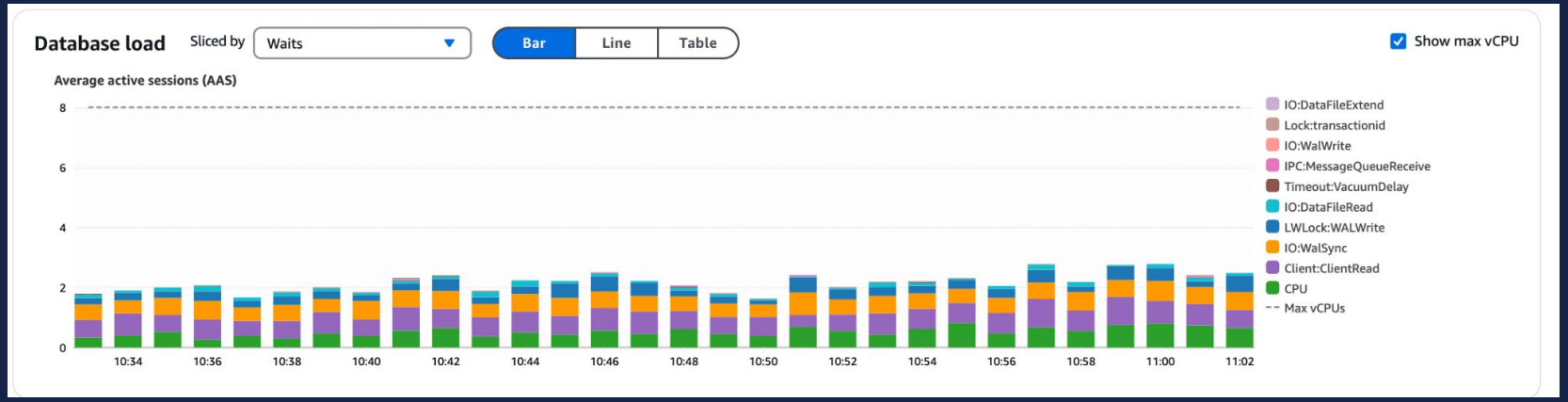
row_num	relation	locktype	pid	mode	fastpath
1	328852	relation	23285	AccessShareLock	t
2	328834	relation	23285	AccessShareLock	t
3	328832	relation	23285	AccessShareLock	t
4	328826	relation	23285	AccessShareLock	t
5	pg_namespace_oid_index	relation	23285	AccessShareLock	t
6	pg_namespace_nspname_index	relation	23285	AccessShareLock	t
7	pg_class_tblspc_relfilenode_index	relation	23285	AccessShareLock	t
8	pg_class_relname_nsp_index	relation	23285	AccessShareLock	t
9	pg_class_oid_index	relation	23285	AccessShareLock	t
10	pg_namespace	relation	23285	AccessShareLock	t
11	pg_class	relation	23285	AccessShareLock	t
12	328870	relation	23285	RowShareLock	t
13	328869	relation	23285	RowShareLock	t
14	328868	relation	23285	RowShareLock	t
15	328850	relation	23285	RowShareLock	t
16	328845	relation	23285	RowShareLock	t
17		virtualxid	23285	ExclusiveLock	t
18	328832	relation	23285	RowShareLock	f
19	328823	relation	23285	AccessShareLock	f
20	328823	relation	23285	RowShareLock	f
21		transactionid	23285	ExclusiveLock	f
22	328817	relation	23285	AccessShareLock	f
23	328817	relation	23285	RowShareLock	f
24	328852	relation	23285	RowShareLock	f
25	328826	relation	23285	RowShareLock	f
26	328834	relation	23285	RowShareLock	f

(26 rows)

How Did We Solve This?

1. Reduced Unused and unnecessary Indexes and Queries Inside Transactions
2. Avoided Long Transactions
3. Properly tune DB parameters
4. Purge old tasks and run vacuum and reindex regularly

AFTER OPTIMIZATION



DB Config

M7g.2xlarge : 8 vCPU , 32GB RAM , 12000 IOPS

SquadStack

7. Conclusion

1. PgQueue Scales well without excessive RAM consumption.
2. Provides better observability via database queries.
3. Ensures reliable task execution with failure handling.
4. Cost effective (we reduced infra cost by ~60%)
5. Custom metrics which can be used for autoscaling of consumers
6. It has access to previous task data, which can be used for debugging if needed

Thank You

SquadStack