ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/25nov415

A Simple Way to Run One Copy of a Job on Many Servers

Xinyi Lu[1]

[1]Panther Creek High School

Publication Date: 2025/11/14

Abstract: In many school projects or small applications, it is common to run operations such as sending a report or cleaning old data regularly. If you have only one server, you can handle this efficiently with a cron job. However, if you have multiple servers, operating the same job simultaneously would be time-consuming. This paper illustrates a simple solution: let all servers "race" to insert one row into a normal database table. The database's unique constraint makes sure only one succeeds. The successful server performs the operation. After it finishes, it removes the row so the process can be performed repeatedly. We do not need special tools or external help; just one small table and a simple trick.

Keywords: *Decentralized Scheduling; Unique Constraint Lock; Lightweight Coordination; Distributed Job Execution; Fault Tolerance; PostgreSQL.*

How to Cite: Xinyi Lu (2025) A Simple Way to Run One Copy of a Job on Many Servers. *International Journal of Innovative Science and Research Technology*, 10(11), 388-392. https://doi.org/10.38124/ijisrt/25nov415

I. INTRODUCTION

When a system scales from a single sever to multiple severs, scheduled jobs can become difficult. If you copy the same cron schedule onto every machine, the job runs multiple times. Sometimes leads to certain consequences like double emails, duplicate cleanups, and wasted CPU.

Large organizations solve this problem with complex schedulers. However, these can be excessive for small teams or student projects. Our goal was to find a solution that is: - Easy to explain. - Fast to implement. - Secure and consistent.

So we use a normal relational database (like PostgreSQL) and a unique constraint as a "soft lock." All nodes try the same insert. The one succeed will run the job.

II. BACKGROUND

➢ *Common Patterns Includes:*

- Single Machine + Cron: Functioning until that single machine fails, at which point scheduled jobs stop running
- Central scheduler: the control point that manages the execution of jobs across other severs, which is critical and necessary. If it fails, the whole system stalls.

We aim to eliminate the need for a central scheduler process. Every sever should be able to attempt to operate the job, but only one sever should actually success to do.

III. CORE IDEA

We make a table named job_lock with a primary key on the job name.

Analogy: Imagine five students trying to grab a single labeled seat in the library when the hour starts. Whoever sits first uses the seat and does the job. When finished, they leave. Next hour, the seat is free again.

How it works: - At the scheduled time, every server tries: INSERT a row (job_name = 'hourly_report'). - The database only lets the first one succeed (because of the unique key). - That server runs the operation. - It deletes the row afterward.

As a result, there will be no need for polling loops, leader election of the central scheduler, since the database already solves the race efficiently.

IV. ADVANTAGES

- Decentralized: No central schedular as the control point
- Reliable and Consistent: If one machine fails or crashed somehow, another machine can continue to run the next operation.
- Minimal setup: Most systems already have a database. We only need to add ONE small table (job_lock). So we do not need Redis, ZooKeeper, or special locking service.
- Handles long jobs: It is not tied to a network session like advisory locks.
- Easy to explain: "Try the insert; if it works, run the job."

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/25nov415

V. IMPLEMENTATION

➢ *The Table*

```
CREATE TABLE job_lock (
   job_name   VARCHAR(255) PRIMARY KEY,
   created_at TIMESTAMP DEFAULT CURRENT_TIME
STAMP
);
```

➢ *Lock Helper (Python + SQLAlchemy)*

```
from contextlib import contextmanager
from datetime import datetime
from sqlalchemy import Column, String, DateTime
from sqlalchemy.orm import declarative_base, sessionmak
er

Base = declarative_base()
SessionLocal = sessionmaker(bind=engine)  # assume engin
e defined elsewhere

class JobLock(Base):
   __tablename__ = "job_lock"
   job_name = Column(String(255), primary_key=True)
   created_at = Column(DateTime, default=datetime.utcnow)

@contextmanager
def session_scope():
   session = SessionLocal()
   try:
      yield session
      session.commit()
   except:
      session.rollback()
      raise
   finally:
      session.close()

@contextmanager
def acquire_job_lock(job_name: str):
   # Try to insert. If row exists, someone else is running the j
ob.
   with session_scope() as session:
      existing = session.query(JobLock).filter_by(job_name=
job_name).first()
      if existing:
         raise RuntimeError("Lock held")
      session.add(JobLock(job_name=job_name))
   try:
      yield
   finally:
      # Clean up so next run can happen
      with session_scope() as session:
         row = session.query(JobLock).filter_by(job_name=j
ob_name).first()
         if row:
            session.delete(row)
(You could also use a single INSERT ... ON CONFLICT
DO NOTHING and check row count. We kept it simple for
clarity.)
```

➢ *Example Job Runner*

- *Script (hourly_job.py)*

```
import time
import logging
from acquire_lock_module import acquire_job_lock  # pret
end import

logger = logging.getLogger(__name__)

def run_distributed_job(job_name, fn, *args, **kwargs):
   try:
      with acquire_job_lock(job_name):
         start = time.perf_counter()
         has_error = False
         try:
            fn(*args, **kwargs)
         except Exception:
            logger.exception("Job failed")
            has_error = True
         duration = time.perf_counter() - start
         logger.info("Job finished in %.2fs error=%s", duratio
n, has_error)
   except RuntimeError:
      logger.info("Skipped; another node is running '%s'", jo
b_name)

# Example usage
if __name__ == "__main__":
   run_distributed_job("hourly_report", lambda: print("Gen
erate report"))
```

- *Cron Line (Put on Every Server)*

```
# Minute 0 every hour
0 * * * * /usr/bin/env python3 /opt/app/hourly_job.py >> /va
r/log/hourly_job.log 2>&1
```

All machines run the script. Only one keeps going past lock setup.

VI. EVALUATION

This section evaluates the decentralized lock acquisition with 10 servers (threads) over 100 rounds. In each round all servers attempt to insert the same key; exactly one succeeds, giving us one winner and its lock acquisition time cost.

➢ *Per-Round Lock Acquisition*

We launched 10 worker threads simultaneously for 100 rounds. Each round records: - Round number - Winning server (server that acquired the lock and able to execute the job) - Lock acquisition time (seconds)

Most acquisition times cluster near 0.10–0.12 seconds, with a few outliers (e.g. 0.50–0.90 s) likely due to transient scheduling or thread wake delays.

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/25nov415

- *Summary Statistics (Computed from Table 1):*

✓ Mean ≈ 0.134 s
✓ Median ≈ 0.111 s
✓ Min = 0.0942 s
✓ Max = 0.8992 s
✓ 90th percentile ≈ 0.147 s

This overhead is tiny compared to typical job runtimes (minutes or even hours), so the approach adds negligible scheduling cost.

Table 1 Per-Round Lock Winner and Acquisition Time (100 Rounds).

Round	Node	Lock Cost (s)
1	Server_6	0.8992
2	Server_3	0.1151
3	Server_6	0.1056
4	Server_5	0.1099
5	Server_9	0.1615
6	Server_2	0.1076
7	Server_4	0.1064
8	Server_3	0.0984
9	Server_2	0.1474
10	Server_3	0.0977
11	Server_5	0.1033
12	Server_3	0.1114
13	Server_8	0.8430
14	Server_0	0.1243
15	Server_5	0.1082
16	Server_4	0.1091
17	Server_0	0.5008
18	Server_9	0.1089
19	Server_5	0.1065
20	Server_9	0.1020
21	Server_1	0.4904
22	Server_2	0.1170
23	Server_9	0.1320
24	Server_6	0.1041
25	Server_9	0.1317
26	Server_5	0.1167
27	Server_0	0.1351
28	Server_3	0.1189
29	Server_8	0.1091
30	Server_9	0.1165
31	Server_8	0.1193
32	Server_9	0.1405
33	Server_0	0.1057
34	Server_5	0.1168
35	Server_9	0.1275
36	Server_1	0.1143
37	Server_5	0.2196
38	Server_1	0.1052
39	Server_2	0.1298
40	Server_5	0.1342
41	Server_2	0.1088
42	Server_0	0.1064
43	Server_6	0.1470
44	Server_8	0.1157
45	Server_5	0.1072
46	Server_1	0.1147
47	Server_2	0.1202
48	Server_4	0.1153
49	Server_8	0.1087

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/25nov415

Round	Node	Lock Cost (s)
50	Server_1	0.1072
51	Server_6	0.1011
52	Server_4	0.1037
53	Server_3	0.0942
54	Server_5	0.1043
55	Server_0	0.0975
56	Server_6	0.0957
57	Server_5	0.1082
58	Server_6	0.1014
59	Server_3	0.1320
60	Server_4	0.1146
61	Server_2	0.1104
62	Server_5	0.1149
63	Server_6	0.1221
64	Server_2	0.1247
65	Server_6	0.1037
66	Server_5	0.1108
67	Server_1	0.1189
68	Server_6	0.1091
69	Server_3	0.1087
70	Server_9	0.1209
71	Server_2	0.1228
72	Server_8	0.1092
73	Server_2	0.1083
74	Server_5	0.1027
75	Server_8	0.1418
76	Server_0	0.1152
77	Server_9	0.1501
78	Server_4	0.1298
79	Server_5	0.1031
80	Server_0	0.1163
81	Server_9	0.1324
82	Server_6	0.1190
83	Server_1	0.1120
84	Server_2	0.1223
85	Server_5	0.1048
86	Server_9	0.1027
87	Server_5	0.1088
88	Server_6	0.1196
89	Server_3	0.1102
90	Server_9	0.1405
91	Server_6	0.1168
92	Server_3	0.1025
93	Server_4	0.1166
94	Server_0	0.1055
95	Server_3	0.0951
96	Server_6	0.1026
97	Server_3	0.1145
98	Server_1	0.1086
99	Server_9	0.1114
100	Server_5	0.1125

- *Distribution of Winners*

Table 2 aggregates wins per server. If all 10 servers participate uniformly, the expected share per server is 10%. We observe natural variation: Server_5 (18%) and Server_6 / Server_9 (14%) are above average, while Server_7 shows 0 wins—suggesting its thread never successfully entered the race or was not active. More rounds (e.g. 1,000) or ensuring thread readiness typically reduces these deviations, trending toward balanced distribution.

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/25nov415

Table 2 Lock Acquisition Counts and Percentages.

Server	Wins	Percentage
Server_0	9	9.0%
Server_1	8	8.0%
Server_2	11	11.0%
Server_3	12	12.0%
Server_4	7	7.0%
Server_5	18	18.0%
Server_6	14	14.0%
Server_7	0	0.0%
Server_8	7	7.0%
Server_9	14	14.0%
(Total)	100	100.0%

- Load balance assessment: Except for one inactive (or unlucky) server, wins are spread across the rest without systematic bias. The decentralized insert race does not preferentially favor a single node;

➢ *Script of Above Simulation*

```
def custom_function():
    """A custom function that simulates some work."""
    log.info(f"Thread {threading.current_thread().name} is ex
ecuting custom_function.")
    time.sleep(2)
    log.info(f"Thread {threading.current_thread().name} cust
om_function done.")

def run_job(round: int, job_name: str, func, *args, **kwarg
s):
    try:
        start = time.perf_counter()
        with acquire_job_lock(job_name):
            time_cost = time.perf_counter() - start
            with open("job_run.csv", "a") as f:
                f.write(f"{round},{threading.current_thread().nam
e},{time_cost}\n")
            func(*args, **kwargs)
    except Exception as e:
        log.info("job lock not acquired")

def run_custom_function(round: int):
    run_job(round + 1, "test_job", custom_function)

with open("job_run.csv", "w") as f:
    f.write("round, node id, acquire lock cost(ms)\n")

with ThreadPoolExecutor(max_workers=10, thread_name_
prefix="Server") as executor:
    for round in range(100):
        for i in range(10):
            executor.submit(run_custom_function, round)
        time.sleep(5)
```

VII. FUTURE IMPROVEMENTS

- Problem: What if the node crashes while running an operation and never deletes the lock row?
- Fix ideas: Add a timeout mechanism, if the time that a job run is 200% greater than its average time (e.g., applied to rows later than X minutes as stale). - Track metrics (The average time that each job run). - Write a script that deletes stale locks.

VIII. CONCLUSION

We showed a small pattern to operate one copy of a job across many servers. The tactic is a single table + unique constraint. It works because databases are good at handling concurrency. This approach is: - Simple & Efficient - Cheap - Good enough for many student or small team projects.

You don't always need an enormous scheduler system. Sometimes one simple and short insert is enough.

REFERENCES

➢ *PostgreSQL Basics*

[1]. UNIQUE Constraint https://www.postgresql.org/docs/current/ddl-constraints.html#DDL-CONSTRAINTS-UNIQUE-INDEX
[2]. INSERT … ON CONFLICT https://www.postgresql.org/docs/current/sql-insert.html#SQL-ON-CONFLICT
[3]. MVCC Intro https://www.postgresql.org/docs/current/mvcc-intro.html

➢ *Single-Machine Scheduling*

[4]. Cron man page https://man7.org/linux/man-pages/man8/cron.8.html
[5]. Systemd Timers https://www.freedesktop.org/software/systemd/man/systemd.timer.html

➢ *Larger Systems (for Comparison)*

[6]. Celery https://docs.celeryq.dev/en/stable/getting-started/introduction.html
[7]. Airflow https://airflow.apache.org/docs/apache-airflow/stable/core-concepts/overview.html
[8]. Sidekiq https://sidekiq.org/