PostgreSQL connection pooling

Last updated by | Lisa Liu | Nov 6, 2020 at 10:34 AM PST

PostgreSQL connection pooling

Wednesday, April 3, 2019
11:30 AM

Connection pooling is a must setup at the application side for chatty applications as it tend to open too many connections with the database at the same second which adds an extra layer of latency that can be reduced by connection pooling

Important blogs on this:

1. https://techcommunity.microsoft.com/t5/Azure-Database-for-PostgreSQL/Not-all-Postgres-connection-pooling-is-equal/ba-p/825717
2. https://techcommunity.microsoft.com/t5/Azure-Database-for-PostgreSQL/Not-all-Postgres-connection-pooling-is-equal/ba-p/825717
3. https://techcommunity.microsoft.com/t5/Azure-Database-for-PostgreSQL/Steps-to-install-and-setup-PgBouncer-connection-pooling-proxy/ba-p/730555
4. https://techcommunity.microsoft.com/t5/Azure-Database-for-PostgreSQL/Set-up-Pgpool-II-Query-Caching-with-Azure-Database-for/ba-p/788005

In PostgreSQL, establishing a connection is an expensive operation. This is attributed to the fact that each new connection to the PostgreSQL requires forking of the OS process and a new memory allocation for the connection. As a result, transactional applications frequently opening and closing the connections at the end of transactions can experience higher connection latency, resulting in lower database throughput (transactions per second) and overall higher application latency. It is therefore recommended to leverage connection pooling when designing applications using Azure Database for PostgreSQL. This significantly reduces connection latency by reusing existing connections and enables higher database throughput (transactions per second) on the server. With connection pooling, a fixed set of connections are established at the startup time and maintained. This also helps reduce the memory fragmentation on the server that is caused by the dynamic new connections established on the database server.
From <https://azure.microsoft.com/en-us/blog/performance-best-practices-for-using-azure-database-for-postgresql-connection-pooling/>

Connecting to a database server typically consists of several time-consuming steps. A physical channel such as a socket or a named pipe must be established, the initial handshake with the server must occur, the connection string information must be parsed, the connection must be authenticated by the server, checks must be run for enlisting in the current transaction, and so on.

In practice, most applications use only one or a few different configurations for connections. This means that during application execution, many identical connections will be repeatedly opened and closed.

Connection pooling reduces the number of times that new connections must be opened. The pooler maintains ownership of the physical connection. It manages connections by keeping alive a set of active connections for each given connection configuration. Whenever a user calls Open on a connection, the pooler looks for an available connection in the pool. If a pooled connection is available, it returns it to the caller instead of opening a new connection. When the application calls Close on the connection, the pooler returns it to the pooled set of active connections instead of closing it. Once the connection is returned to the pool, it is ready to be reused on the next Open call.

How to know if your customer needs connection pooling

```
//number of connections created per second  (if the customer is exceeding 15-20 new connection per second this will mean that connection pooling is needed here)
MonLogin
| where (logical_server_name =~ "{ServerName}" )
| where AppTypeName == "Gateway.PG"
| where event == "process_login_finish"
//| where is_success
| summarize count() by bin(originalEventTimestamp, 3s)
//| where count_ > 25

//latency establishing a connection with the server - latencies anywhere between 200ms to 800ms are considered okay! More than that is a good indication for issues can be resolved by connection pooling

MonLogin
| where TIMESTAMP >= ago(2d)
| where (logical_server_name =~ "{ServerName}" )
| where event == "process_login_finish" or event == "connection_accept"
| where AppTypeName == "Gateway.PG" or AppTypeName == "Host.PG"
| where is_success == true
| where connection_id != "00000000-0000-0000-0000-000000000000"
| summarize start=min(originalEventTimestamp), end=max(originalEventTimestamp) by connection_id, logical_server_name
| extend duration = bin(end - start, 1tick)
| summarize latency_seconds=tolong(avg(duration))/10000000.0 by TIMESTAMP=bin(start, 1m), logical_server_name
```

Created with Microsoft OneNote 2016.

How good have you found this content?