Methods for Horizontal Scalability of Real Time Applications

Namrata Suvarna
J. P. Morgan
namrata.p.suvarna@jpmorgan.com

ABSTRACT

When the demand for your application is soaring, you recognize a need to expand the application's accessibility, power and presence.

The smallest, single-purpose solution is not going to stay that way forever; designing for the traffic and function that an application will initially handle is fine but inability to scale the underlying system will limit the future growth of the platform.

While there are various approaches available in the market to scale our applications based on the input it expects - queue messages, web service requests, etc. - database entries are difficult to process on a multi threaded level while keeping the performance and consistency intact. An easier solution for this challenge will be extensively discussed in the **Timeout** section.

AUDIENCE

This discussion is targeted for those with prior knowledge and some working experience on the topic.

INTRODUCTION

Scalability is a requirement in this day and age where real time applications are expected to handle a growing amount of work while keeping factors like performance, availability and fault tolerance at their optimal levels.

Applications can scale horizontally (scaling out/in) by adding additional resources or vertically (scaling up/down) by adding more power to existing resources.

In this proposal, we shall address some of the horizontal scaling approaches for high performance computing like:

- Restful Web Services
- Messaging Queue: Point to Point, Publisher Subscriber
- Centralized Task Tracker [Heartbeat]
- Database: Database Row Level Locking, **Timeout**, Oracle Advanced Queuing

Restful Web Services:

The key reason for scalability of restful services is statelessness.
The server does not store any state about the client session; the relevant session information is stored on the client side and passed to the server when needed; the ST in REST - State Transfer - implies that you transfer the state around instead of having the server store it.
Multiple clients can hit the same REST endpoints, perform the same actions and receive the same responses; load balancers help in distributing the load across the endpoints.

Messaging Queue:

Point to Point

Competing consumers are multiple consumers created on a queue, by an application, and any of them could potentially receive a message delivered on that queue.
Multiple application instances can read from or publish to the same queue.
Spring's Default Message Listener Container *allows us to configure our communication channel and define the number of concurrent consumers on it which can dynamically increase or decrease as the volume of messages fluctuates.*

Publisher Subscriber

A message sent on a point-to-point queue is only processed by a single consuming application but a message sent on a 'publish-subscribe' topic is processed by every interested subscriber unless a message filtering policy is set.
The subscribers to a message topic often perform different functions simultaneously on the same set of messages.
Apache's Kafka *has the advantages of a queuing model and a publish-subscribe model combined, thus providing scalability on two levels - multiple consumer groups reading from different topics and multiple consumers of a group reading from different partitions of the same topic.*

Centralized Task Tracker [Heartbeat]:

A centralized task tracker manages the sharing of control among different worker threads, tracks the status of distributed data to ensure consistency, mutual exclusion, co-operation and fail-safe

synchronization while avoiding a single point of failure.
The simplest implementation of a task tracker can be the usage of a shared synchronized state variable but more robust APIs are available for complex applications.
Apache's Zookeeper *is a distributed application of server nodes to ensure robust co-ordination in a distributed application of clients. The clients send heartbeats to the server nodes at specific intervals to ensure the sessions are alive. Reads are quite fast as the nodes have their own synchronized database; write requests always go to the leader node that ensures the data is replicated across all the server nodes for consistency.*

Database:

Database Row Level Locking
It helps in scalability of applications where the input lies in a database and multiple threads are required to read and process a different set of records.
It allows you to lock a set of records which is released only when a commit or rollback statement is issued and works only by setting auto commit to false or by starting a transaction.
Oracle's *select for update - when used with options like - NOWAIT, the query executes immediately and fails if the requested row is locked; SKIP LOCKED excludes locked rows from the result set.*

Oracle Advanced Queuing
This is a database-integrated message queuing technique.
It provides the same ways of asynchronous communication as a normal messaging queue i.e. point-to-point and 'publish-subscribe', only difference being that the queues are hosted on a database in the form of tables.
It has all the pluses of a queuing system and database combined and can be implemented using interfaces like oracle.JMS package, PL/SQL or oracle.AQ package.

Timeout

Problem Statement:
When a database is an input source for your application, it is difficult to scale the application easily as you need to lookout for issues like race conditions, deadlocks and data inconsistency.
It is easier to run a single threaded application as you can be sure that there will be only one access point.
But what if your data is in the magnitude of millions and you have no choice but to have multiple parallel consumers hitting your database?
You need to ensure that the data is not in an inconsistent state for any of these consumers and that none of the consumers get locked while trying to access the same dataset.
These issues could lead to a performance worse than that of a single thread run, not giving you the solution but ensuring more problems to solve.

Objective:
The following solution is an easy implementation where multiple threads try to access the same dataset while avoiding locking, consistency issues thus making it possible to scale a database driven application with minimal effort. This solution can support a multi-threaded, multi-instance database driven application.

Solution:
Consider a scenario where we want to process a set of three records simultaneously. We configure three threads to access the same data set where each thread is responsible for processing one record.

A simple select clause only based on the status results in the same records getting picked up repeatedly. Since it is a read operation, no row is locked and hence the same row is returned to every thread till its status is changed by one of them, thereby defeating our purpose.

select * from table where status = 'READY' and rownum <= 1

Without Batch ID, Status, Timeout Handling

ID	BATCH_ID	PROCESSING_TIME	STATUS
1	null	23-APR-18 01.54.27.078392000	SUCCESS
2	null	23-APR-18 01.54.27.089403110	READY
3	null	23-APR-18 01.54.28.090514220	READY

To address the above issue, we flag the records to be processed by a thread using an ID and a status to make it clear to other threads that these records have been reserved without running into locking issues. These records are then picked up using a combination of that ID and status.

update table set batch_id = '<thread name>', status = 'IN_PROGRESS' where status = 'READY' and rownum <= 1

select * from table where batch_id = '<thread name>' and status = 'IN_PROGRESS'

With Batch ID, Status Handling But Without Timeout Handling

ID	BATCH_ID	PROCESSING_TIME	STATUS
1	Thread1	23-APR-18 01.54.27.078392000	SUCCESS
2	Thread2	23-APR-18 01.54.27.089403110	SUCCESS
3	Thread3	23-APR-18 01.54.28.090514220	IN_PROGRESS

However, say the application crashes and some records are left in the IN_PROGRESS state. When the application restarts, it will never pick up these records again as they do not satisfy the select clause anymore.

Finally, to address the above issue and attain eventual consistency, for a thread to process, we pick up those records which are not only READY but also those stuck in IN_PROGRESS for more than the set timeout interval.

```
update table set batch_id = '<thread name>', status =
'IN_PROGRESS' where (status = 'READY' or (status =
'IN_PROGRESS' and processing_time <= <current time> -
<timeout interval>)) and rownum <= 1

select * from table where batch_id = '<thread name>' and
status = 'IN_PROGRESS'
```

With Batch ID, Status, Timeout Handling

ID	BATCH_ID	PROCESSING_TIME	STATUS
1	Thread1	23-APR-18 01.54.27.078392000	SUCCESS
2	Thread2	23-APR-18 01.54.27.089403110	SUCCESS
3	Thread3	23-APR-18 01.54.28.090514220	SUCCESS

If the timeout interval is not appropriate, below scenarios could lead to unnecessary timeouts:

- Records stuck in an IN_PROGRESS state as they genuinely require that much time to get processed
- Backlog of a thread is too high and sequential processing leads to some records not getting processed for quite some time after they have been flagged to IN_PROGRESS

CONCLUSION

While the technologies mentioned before are commonly used and have their established standards of success, horizontal scalability on the database has always been a challenge.
The timeout technique not only ensures that multiple threads tackle different database records simultaneously - providing performance and scalability but also ensures that jammed events reach their final state - providing eventual consistency.

PARTICIPATION STATEMENT

If this proposal is accepted, the speaker will be present at the conference.

BIO

The speaker has around five years of experience, working in technology for the investment banking domain. She has worked on projects requiring multi-threading, caching, database persistence, web services and cloud based development. Her skillset mainly includes technologies like Java 8, Gemfire, Ehcache, Oracle, Sybase, REST APIs, Javascript, HTML 5 and Cloud Foundry among others.

REFERENCES

https://docs.oracle.com/cd/B10500_01/appdev.920/a96587/qintro.htm
https://docs.oracle.com/database/121/SQLRF/statements_10002.htm#SQLRF01702
https://zookeeper.apache.org/doc/current/zookeeperOver.html
https://kafka.apache.org/documentation/
https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/jms/listener/DefaultMessageListenerContainer.html
https://www.tutorialspoint.com/restful/index.htm