6.830 Project: Partitioned database with deadlock detection

Benjamin Mattocks, Wenting Zheng

May 16, 2013

Introduction

Our project implements a distributed database with dynamic repartioning. Repartitioning is performed to reflect the database workload such that items accessed together are placed on the same server. Our project also implements distributed deadlock detection and resolution.

Data storage

Our distributed database implements a simple key-value storage system with table support. The interface for accessing data contains the two operations `write(table_name, key, value)` and `read(table_name, key)`.

It also has transactional support for two-phase locking and two-phase commit. Two-phase commits are supported by an RPC protocol. Whenever a transaction is begun, a new thread is created to send and receive RPC requests used in two-phase commit.

Partitioning

This distributed database provides support for repartitioning the data among the servers. The number of partitions can be chosen by the user. Hash partitioning is used because it scales well with the number of inserts. While range partitioning could also be used, it is more of a challenge to manage as the number of inserts increases.

Repartitioning

Deadlock detection

Background

A deadlock occurs when two or more process are requesting locks to items being held by other processes. More formally, it can be modeled as a Wait-For-Graph, in which a resource has a directed edge to another resource which holds a lock to an item the first resource requests. If there is a cycle in this graph, a deadlock is present.

In a distributed system, deadlock detection has been the subject of numerous studies because the distributed nature adds some complexity to the system, so certain deadlock detection algorithms might not work in a distributed environment. Distributed deadlock detection can be done with the use of centralized servers which keep track of the state of a distributed system, or they can be performed in a completely distributed manner. One problem that could occur with deadlock detection is that deadlocks might be falsely detected, which can impact a system. Some specific causes would be message delays across servers or using stale information in the WFG.

Implementation

We decided to implement a fully distributed algorithm, proposed by Chandy, Misra, and Haas. This is an edge-chasing algorithm, which passes probe messages between resources directly. The main idea is that the origin of the first probe knows that a deadlock is present if that probe message is received by the origin a second time. If a process is also waiting for a lock and receives a probe, it sends the probe message along to any other processes which hold locks to resources it is requesting. It retains the origin in its state so the origin will know that it sent that probe if the probe reappears there.

To implement this, we ran a specific thread for each worker on our distributed database which specifically handles these deadlock probes, independently of the main worker thread for a given transaction. This allows a worker to detect a deadlock while a specific lock is being requested. If a deadlock worker thread detects that a probe message it receives was initiated by itself, then it reports a deadlock to the worker thread and aborts the transaction. Specifically, the deadlock detection thread sets a flag in the worker thread, which then triggers an abort message to be sent by our RPC protocol to the same worker thread.

Test scenario

We tested the simple case of two workers in separate threads wanting to obtain a lock held by the other concurrently. Suppose a worker W1 holds a lock on A and is attempting to get a lock on B. If a worker is attempting to lock a resource but cannot obtain the lock, a new probe message is sent to every other resource that holds a lock the worker requests. Suppose W2 holds the lock to B and wants the lock to A. If W1 first sends a probe message to W2, then W2 will send a probe message to W1 because W1 holds a lock to A, which W2 wants. W1 will receive this message and note that this message originated there, so it declares a deadlock and aborts itself.

Deadlock detection analysis

The deadlock detector runs in a separate thread, so this can cause a performance overhead. These threads are always running when a transaction is occurring, and the threads are only utilized when a possible deadlock is happening. The number of probes from an origin is no more than the number of edges in the WFG, but there may be more probes from an origin if the deadlocks are processed in a timely manner. We did not have a specific time requirement for deadlock detection and resolution, so deadlocks might not be detected immediately if the process obtaining a lock is still going. We set a timeout to stop the process from obtaining a lock and to proceed with the abort if there is a deadlock.

Further optimizations that could occur but that we did not implement include localized deadlock detection. That is, if the transactions in a WFG reside in the same server, the probing would not need to occur over RPC because the WFG would be local. This could save some RPC communication overhead in the real world, and because our database repartitions data according to how often values are accessed at the same time, this could present an opportunity to implement localized detection. The probe frequency could also be decreased following some optimizations provided by Chandy et al. These were not implemented in our system.

Analysis

Conclusion