Minimize coordination

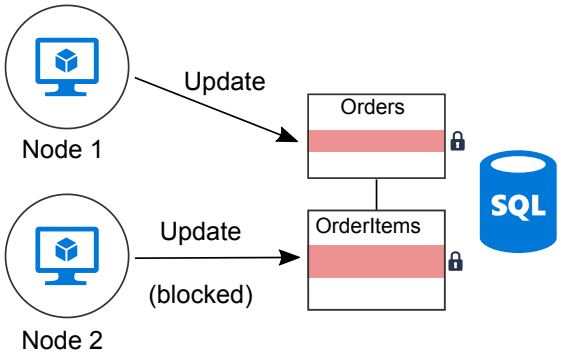
08/30/2018 • 4 minutes to read • Contributors

In this article

Minimize coordination between application services to achieve scalability

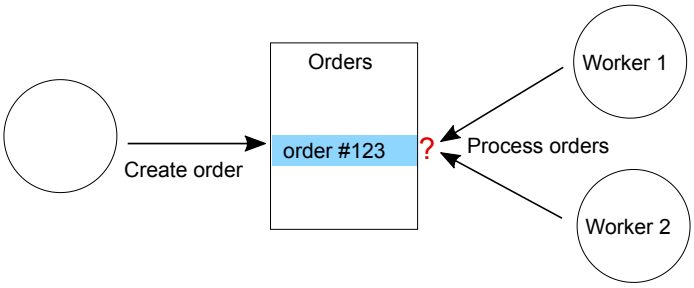
Most cloud applications consist of multiple application services — web front ends, databases, business processes, reporting and analysis, and so on. To achieve scalability and reliability, each of those services should run on multiple instances.

What happens when two instances try to perform concurrent operations that affect some shared state? In some cases, there must be coordination across nodes, for example to preserve ACID guarantees. In this diagram, `Node2` is waiting for `Node1` to release a database lock:



Coordination limits the benefits of horizontal scale and creates bottlenecks. In this example, as you scale out the application and add more instances, you'll see increased lock contention. In the worst case, the front-end instances will spend most of their time waiting on locks.

"Exactly once" semantics are another frequent source of coordination. For example, an order must be processed exactly once. Two workers are listening for new orders. `Worker1` picks up an order for processing. The application must ensure that `Worker2` doesn't duplicate the work, but also if `Worker1` crashes, the order isn't dropped.



You can use a pattern such as Scheduler Agent Supervisor to coordinate between the workers, but in this case a better approach might be to partition the work. Each worker is assigned a certain range of orders (say, by billing region). If a worker crashes, a new instance picks up where the previous instance left off, but multiple instances aren't contending.

Recommendations

Embrace eventual consistency. When data is distributed, it takes coordination to enforce strong consistency guarantees. For example, suppose an operation updates two databases. Instead of putting it into a single transaction scope, it's better if the system can accommodate eventual consistency, perhaps by using the Compensating Transaction pattern to logically roll back after a failure.

Use domain events to synchronize state. A domain event is an event that records when something happens that has significance within the domain. Interested services can listen for the event, rather than using a global transaction to coordinate across multiple services. If this approach is used, the system must tolerate eventual consistency (see previous item).

Consider patterns such as CQRS and event sourcing. These two patterns can help to reduce contention between read workloads and write workloads.

- The CQRS pattern separates read operations from write operations. In some implementations, the read data is physically separated from the write data.
- In the Event Sourcing pattern, state changes are recorded as a series of events to an append-only data store. Appending an event to the stream is an atomic operation, requiring minimal locking.

These two patterns complement each other. If the write-only store in CQRS uses event sourcing, the read-only store can listen for the same events to create a readable snapshot of the current state, optimized for queries. Before adopting CQRS or event sourcing, however, be aware of the challenges of this approach.

Partition data. Avoid putting all of your data into one data schema that is shared across many application services. A microservices architecture enforces this principle by making each service responsible for its own data store. Within a single database, partitioning the data into shards can improve concurrency, because a service writing to one shard does not affect a service writing to a different shard.

Design idempotent operations. When possible, design operations to be idempotent. That way, they can be handled using at-least-once semantics. For example, you can put work items on a queue. If a worker crashes in the middle of an operation, another worker simply picks up the work item.

Use asynchronous parallel processing. If an operation requires multiple steps that are performed asynchronously (such as remote service calls), you might be able to call them in parallel, and then aggregate the results. This approach assumes that each step does not depend on the results of the previous step.

Use optimistic concurrency when possible. Pessimistic concurrency control uses database locks to prevent conflicts. This can cause poor performance and reduce availability. With optimistic concurrency control, each transaction modifies a copy or snapshot of the data. When the transaction is committed, the database engine validates the transaction and rejects any transactions that would affect database consistency.

Azure SQL Database and SQL Server support optimistic concurrency through snapshot isolation. Some Azure storage services support optimistic concurrency through the use of Etags, including Azure Cosmos DB and Azure Storage.

Consider MapReduce or other parallel, distributed algorithms. Depending on the data and type of work to be performed, you may be able to split the work into independent tasks that can be performed by multiple nodes working in parallel. See Big compute architecture style.

Use leader election for coordination. In cases where you need to coordinate operations, make sure the coordinator does not become a single point of failure in the application. Using the Leader Election pattern, one instance is the leader at any time, and acts as the coordinator. If the leader fails, a new instance is elected to be the leader.