Concurrency

Concurrency theoretically

Optimistic vs. pessimistic

There are two approaches to resolve concurrency conflicts: optimistic and pessimistic. Pessimistic concurrency assumes that the conflicts are going to happen, and often. We thus lock the database, making sure no one will interfere and create a conflict. This of course means that during this lock period, no one will be able to access the database until it is unlocked, which limits the user greatly.

Optimistic on the other hand assumes that conflicts will not happen. The user is let to do what he desires and if he unknowingly creates conflicts, they are resolved afterwards. It does not limit the users in any way, which means it is the best solution for systems with relatively low possibility of conflicts. Also, compared to the pessimistic approach it is often easier to implement.

Each approach has its shortcomings.

Pessimistic can run into 'deadlocks' - a state when each resource waits on the other to make some progress, waiting indefinitely. This can lead to the system not responding and being stuck at the same point. Optimistic on the other hand can resolve into 'livelocks' - states similiar to deadlocks, but where each resource is constantly changing its state in relation to others, but without any progress. A good example of a real-life livelock would be when someone comes across a person in a narrow corridor. They both politely try to move aside for the other person but end up going back and forth the same way, without any progress.

maybe talk about how they can be resolved???

Transactions

Transactions are a single unit of various tasks/queries that gets executed. Each task gets executed individually and based on the outcomes, the transaction results in a success (every task was successful) or a failure (at least one task failed). A database transaction should always be ACID - atomic, consistent, isolated & durable. They are often the basis for pessimistic concurrency control, although they can be part of optimistic concurrency as well.

At several places in the code, we tend to use transactions instead of plain Dapper queries. The reason is simple - performance. This means that transactions are used not only to batch SQL queries together which either succeed or are rollbacked, but they are also executed faster, whether the query is an insert or an update. The reason why using transactions is more time-efficient is because the transaction is always implicitly created, for say a write operation. This means that if one specifies when the transaction starts and ends, SQL does not have to figure it out on its own.

(pic for insert)

(pic for update)

It must be noted that naturally, each transaction has an overhead, meaning the less transactions, the better. Having a single transaction instead of ten of them saves a significant amount of time. But there is a downside, if one query fails, the entire transaction is rollbacked, meaning we cannot wrap everything into a single transaction. Another important part play the forementioned isolation levels. When implementing transactions, we had to make sure that they are not wrongly blocking other queries to the database. When it comes to the performance of individual isolation levels there is not much of a difference. (pic)

<https://blog.staticvoid.co.nz/2012/making_dapper_faster_with_transactions/>

<https://www.geeksforgeeks.org/sql-transactions/>

<https://en.wikipedia.org/wiki/Deadlock>

https://agirlamonggeeks.com/2017/02/23/optimistic-concurrency-vs-pessimistic-concurrency-short-comparison/

Our concurrency