NETWORK LATENCY AND CHATTY SQL SERVER APPS

Taking a SQL Application that normally runs on a LAN and moving it to a high network latency environment such as the cloud or a WAN can be problematic. Queries that previously took less than a millisecond to execute could now be taking 20 or 40 milliseconds due to the network round trip. This order of magnitude deceleration can turn an application from being slick and responsive to being unusably slow. Surprisingly, machine learning could hold the solution.

Unless an application has been optimised for high network latency environments it is likely to execute multiple queries for any given operation. A click of a button may trigger off a sequence of queries that must all complete one after the other before the screen refreshes. If each of these queries has to suffer a large network round trip the combined latency can quickly add up to make an app frustratingly unresponsive.

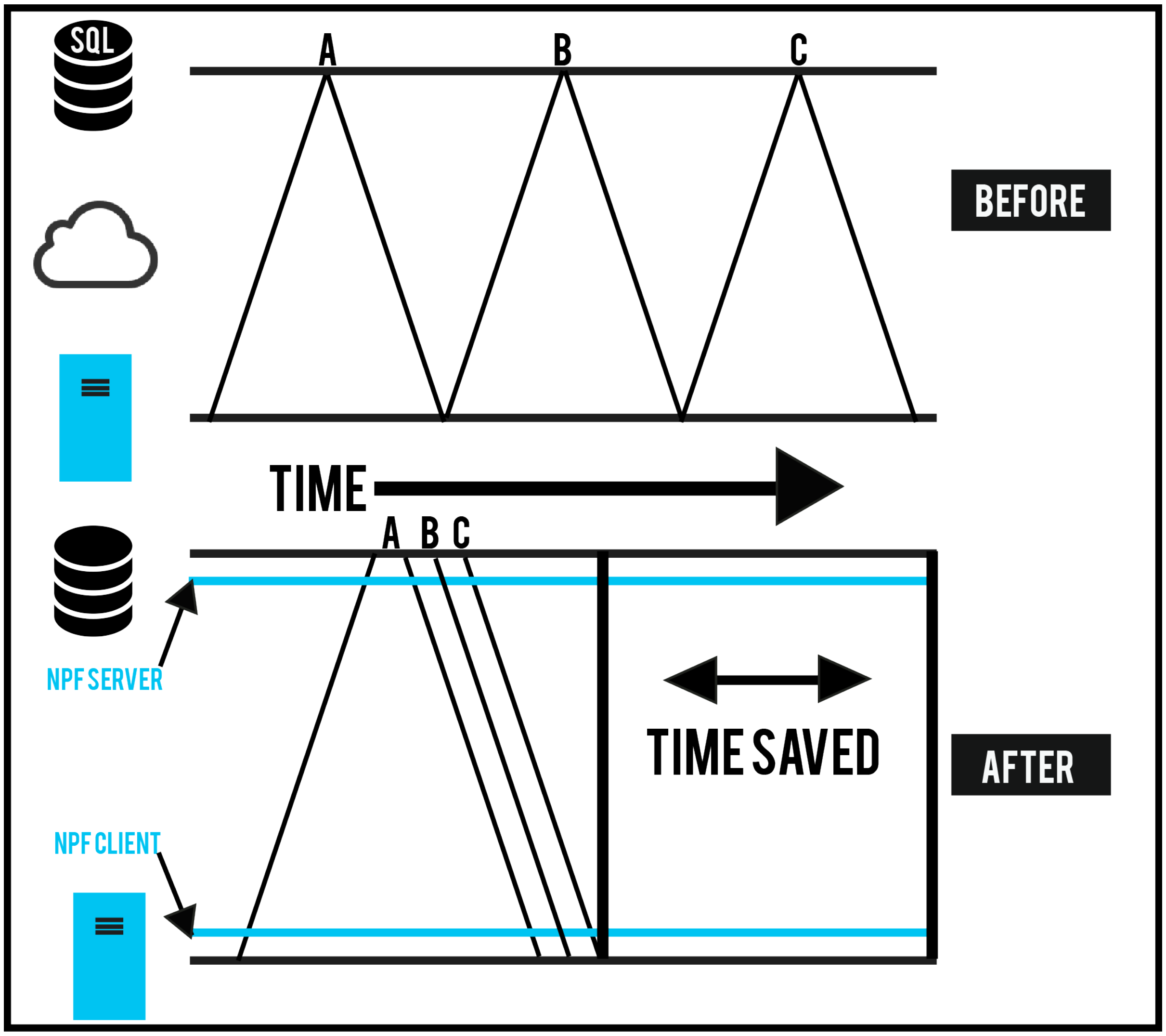
If an app takes over 100ms to respond then the user will notice the delay. On a 40ms WAN it would only take 3 queries to push you over that limit with network latency alone. In our investigations into geo-expansion and cloud migration projects we've routinely seen apps that issue 20, 40 even 60 queries each time a form is opened or a when a button is clicked. These apps become unusable over a high latency connection.

One solution is to rewrite the application: Make it less chatty, ideally only having a single call to the database for each operation. Make calls to the database asynchronously. Develop an app server that makes the calls to the database and ensure the client-app server traffic is not chatty. Maybe even load data in the background in anticipation of what the user will do next. The problem is a rewrite can be very expensive.

NitroPrefetch solves the network latency problem by predicting what queries are coming up and issuing them to the database a fraction of a second before they are needed. Let's see how that works:

The top half of the diagram below shows an application issuing three queries to the database, A, B and C, with time going to the right. Here A, B, and C represent a single operation for the application, for example in an application with a graphical interface the position marked "start" could be when the user clicks a button and "end" is when the screen refreshes. But equally this applies to applications without GUIs, "start" could be when an API call is made and "end" when the result to the API call is sent back to the caller. The point is that the responsiveness of the application is constrained by the time taken to issue these three queries in series.

The bottom half of the diagram shows how NPF (NitroPrefetch) speeds up the response time of the 3 queries. NPF has a client and server component that together act as a proxy for the SQL traffic (using the TDS protocol). NPF server monitors the requests that are issued by the application and makes a model of how the application issues queries through it's various workflows.



Once NPF Server has seen a workflow the machine learning algorithms are able to make predictions of the exact queries that are coming up when the workflow next comes up. It predicts the TSQL statement or what stored procedures will be used, what the parameters will be, and what order the queries will be in.

In the diagram, NitroPrefetch has noticed the sequence A,B,C before so that now when it sees query A again it can predict that B and then C will come next. Once it has received the response to query A it can issue query B. Likewise once it receives the response to query B it can issue query C. This way the queries are issued to the database as fast as any one connection can handle them, the only network round trips happening are the sub-millisecond ones between NPF Server and SQL Server. The responses to B and C are streamed to NPF Client which queues them up and providing the application does indeed request B then C it will be given the appropriate responses.

If the predictions are wrong and a different query from B follows A then the responses to both B and C are thrown away. The application therefore has to wait for the round trip to retrieve the response from the database, much as if NPF was not there.

However discarding an incorrectly predicted query may not be enough, if the predicted query changes the state of the connection (for example with a Use statement) and the prediction is wrong, NPF issues another query to restore the state of the connection to how it was before the predicted query was issued. Also writes (such as an Insert) are not prefetched at all.

You might be thinking that these incorrect predictions increase the load on the database, and you'd be right but it is worth noting that incorrect predictions are quite rare when using NPF. The algorithms are specialised for predicting deterministic processes (the code in the application) and actually stop making wrong predictions once it has learnt how the application works.

The algorithms learn very quickly. For workflows without variation such as when you first start the application, NPF only needs to see the workflow once before it can achieve a high prefetch rate. Other workflows may have parameter value variations. For example displaying a Customer Information form will be a workflow that varies depending on exactly which customer it is displaying the information for. For workflows such as this, NPF will generally provide some benefit the second time the workflow is run but will not be up to full speed until the NPF has fully learnt how the various parameters are used. This generally takes NPF server seeing the workflow 2 or 3 times but can be up to 6 times for very complicated workflows.

The model of the workflows is held in NPF Server so only one user need experience an non-accelerated operation. Once that one user has completed the operation, all users will then have that operation accelerated. On a system with a large number of users it it becomes rare to experience a non-accelerated operation on the day that the learning starts, and typically non-existent thereafter.

NPF is always getting the data fresh from the database so it is never more than a fraction of a seconds old. The queries are never reordered so the latest response is always at least as up to date as the previous queries. This ensures that NPF maintains SQL's consistency model. Everything SQL Server guarantees, NPF also guarantees. This means it won't introduce bugs and the app does not need re-writing. The configuration is minimal, just tell NPF server how to connect to the database) and so an accelerated connection can be set up minutes.