

Improving Data Access Performance with Data Caching

Chris Tacke

OpenNETCF Consulting

September 2007

Introduction

Performance of the data access layer of a mobile or embedded application is often viewed by developers in very broad strokes. When we begin designing a solution we consider the performance versus benefits of using DataReaders, DataAdapters, DataSets and the like. Most often we just make an early determination based on what our expected use and the generally accepted “best practice” is and then move on without much further thought.

Traditionally if we only wanted to ever read the data in a non-scrollable cursor, we’d use a DataReader. If we want to alter the data we’d use a DataSet. Version 2.0 of the Compact Framework brought the wonderful “blended” use DataAdapter, and many developers migrated to it and considered the variable of data access performance to be largely out of their hands beyond that point.

A recent plant floor automation system project that we worked on was a stark reminder that such assumptions are not true, and that performance considerations need to be looked at throughout the development of any solution.

Data Access as a Bottleneck

Like many projects, we didn’t really start to discover show-stopping performance problems in our recent endeavor until we were getting close to delivery of a full working system. Early tests during development usually miss the full impact of all of the interactions of a full running system, and it’s really hard, if not impossible, to guess what the actual execution performance of any solution will be until all of the major pieces are in place and cooperating. Sure, you might get hints at problem areas early on, but those are usually gross, easy-to-find-and-remedy problems usually contained within a single subsystem. It’s not until you have enough infrastructure in place for the system as a whole to start working as a unit that you start to see the larger-scale problems, and coincidentally it’s usually about the same time that you start delivering drops to your customer.

Our system collected event data from Programmable Logic Computers (PLCs) on the factory floor, aggregated that data into relational SQL CE tables, and then generated meaningful reports from that data through a light-weight ASP.NET Web Server running on Windows CE that we built (I’ll discuss more on that specific technology in a future article). The basic unit of measure that we used to judge system performance was the number of messages coming from PLCs that would could handle while still serving up reports.

The customer’s initial requirement included the ability to handle around 5 events per second, and early tests and proof-of-concept demonstrations had us running as many as 60 events per second so we were

confident that we were not just going to meet their requirements, but dazzle them with an order-of-magnitude-better performance. Then as we approached the time for final delivery of the system and we started tying all of the multitude of pieces together we suddenly found the system struggling to even handle 2 or 3 events per second, and new calculations from the customer we indicating a desire to handle not just 5 but 10 events per second.

My initial instinct based on the application's behavior and experience was that Garbage Collection was the problem, but as I outlined in a previous article [insert link] that turned out not to be the case. After heavily instrumenting the code I determined that a large amount of our cycle execution time was being spent in data access (a cycle being the receipt of an event as an array of bytes, parsing that event and then storing the results in a table) .

Part of the parsing and storage involved looking up foreign key integers from lookup tables (the events come in with "denormalized" text data, so we have to look up the ID based on text) and after instrumenting the code to even finer granularity I found that even these simple reads were taking a *lot* longer than one would expect.

Profiling Lookup Table Access Speed

Since I knew that simple lookups were taking a long time (tens to hundreds of milliseconds) I knew immediately that we needed to try to decrease this, but how? We needed the data from the lookup table to do subsequent inserts and/or updates. The answer was to generate our own data caching mechanism at the table level.

Most data entities in the solution have been abstracted to single class definitions and a search on the table was done through methods like GetCustomerByID or GetIDFromCustomerName. We were doing a lot of the latter type – getting an ID based on other fields, but the caching technique we're going to cover in this article works well for any of them.

It's a lot easier to talk about performance gains if we can quantify the performance before the "fix" and again after the fix, so before we move on to the solution, let's take a look at some post-mortem investigation data I gathered.

I created a simple application that creates a SQL CE database with a single table (our simulated lookup table) and then puts just 2 rows into the table. With only 2 rows we remove the possible effects of any table indexing, since a lookup is going to require two fetches at most.

I then created very basic code that would get a row from the table based on the row identifier (so `SELECT * FROM Table WHERE RowID = N`). The architecture of our large solution was such that most data transactions created their own connection to the database, so I added code to my test application that used the same behavior. To smooth out the results, the application would create the connection, open it, run the SQL and then close the connection. It did this 100 times and output the average time it took to perform the operation. The results from running the test on three separate test devices are below (the actual production device being the first in the list):

Device	Processor	Mean Execution (ms)*
OLDI SAM-L8	800MHz Geode x86	7.5 to 9.5
iCOP eBox 2300	200MHz Vortex86	47 to 52
Dell Axim x51	416MUz Intel PXA270	38 to 62

*Ranges are given because we ran several iterations of the 100-run tests

Intuition and experience (and other developers) told me that the first step in improving these numbers would be to reuse the connection instead of creating and opening it every time. So I modified the test application to create a global connection and use it without closing for all of the queries. The results for those tests are below. Note that these numbers exclude the first run which creates and opens the connection, so these numbers reflect purely the time to get one row of data from a table containing only two rows of data.

Device	Processor	Mean Execution (ms)
OLDI SAM-L8	800MHz Geode x86	7.1 to 14.8
iCOP eBox 2300	200MHz Vortex86	51 to 52
Dell Axim x51	416MUz Intel PXA270	43 to 61

Now if you go back and look at the two result tables you'll probably be as surprised as I was (and yes, I double- and triple-checked these numbers).

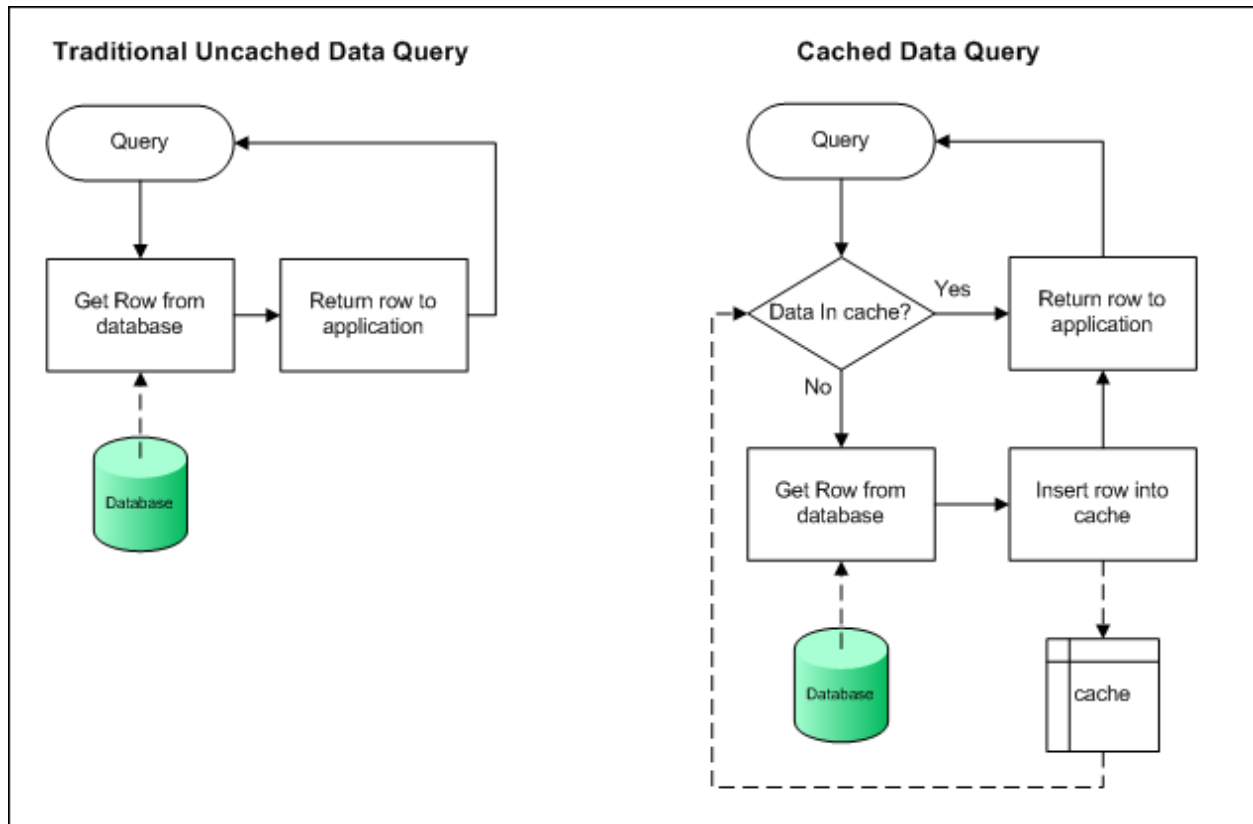
Lesson 1: *There's no performance benefit whatsoever to keeping the connection created and opened through the run of your application.*

Ok, so things aren't quite what we expected, but I created a meaningful test that provided baseline numbers that I needed to improve. The next step was to implement caching and then quantify the results to see if the improvement in performance warrants the extra time it takes to implement the caching itself.

Implementing a Data Cache

So what exactly is a data cache and how do we go about creating one? The concept (and really the implementation) is quite simple. You start by creating a data type (could be a struct or a class) that describes a row of data. That entity can be a subset of the columns in a table if your lookup query typically only looks at a couple fields. You then create a searchable collection to hold a series of these entities. In my test application I used a Hashtable because it makes lookups by a key value very easy. This collection of entities is the cache.

Using the cache is simple. When a table lookup is needed, instead of going right to the database you first look in the cache to see if the row is there. If it is, you return the data from the cache. If it's not, you get it from the database, store it in the cache and then return the data. In this way the cache grows as data is queried to spread out the load time over many operations. Another mechanism would be to populate the cache with all of the lookup table rows on creation. This method would mean that all lookups, including the first will be as fast as possible, but you have to pay up front to fill it.



I added a cache to the test application and re-ran the tests. The results follow. Again these numbers omit the first run which was skewed high because we had 1 table lookup and 99 cached instead of 100 cache reads.

Device	Processor	Mean Execution (ms)
OLDI SAM-L8	800MHz Geode x86	0.0193 to 0.0201
iCOP eBox 2300	200MHz Vortex86	0.1014 to 0.1164
Dell Axim x51	416MUz Intel PXA270	0.0975 to 0.1437

Yes, again those numbers are correct – the decimal points are not in the wrong place.

Lesson 2: *Cache reads are two orders of magnitude faster than a database read.*

Of course I expected the cache reads to be faster, but I wasn't expecting this much of a gap. In fact I have no explanation for why the difference is so large. The database that the test application created was in RAM, not persistent storage, so both the database read and the cache read are coming from RAM.

Conclusion

So what did we learn from this test? First, that the "general knowledge" that reusing a connection is faster appears to be completely false and second that caching data has some seriously large

performance advantages. Of course you still have to weigh those advantages against the advantages of getting the data from the database.

Your cached data is not going to get updated when a table is updated, so it probably not a good idea for tables that have data that changes a lot. The cache maintenance code gets more complex if you have code that deletes or updates the lookup table.

You can't join or query your cached data with SQL. If the data is not a typical lookup table and is often used in joins, then it's probably not a good candidate for caching.

The data cache takes up memory. If you have a really large lookup table (say a lookup of every product a large retail store has on its shelves for example) then a cache might not be a good idea, though you could modify the cache maintenance code to hold maybe the last 100 or so rows read, so frequently queried rows would often be in the cache and fast to read.

So is caching the answer to every database performance woe that might plague you in a project? Obviously not, but it is an important tool that you can use to improve some areas of your application. It is simply another tool that you now have and it's going to be up to you to recognize if and when it's applicable to a problem you're working on, but at least now you know it exists. And, as I was told many times by a cartoon as a kid, knowing is half the battle. Go Joe!