How heroes navigate the query minefield

Introduction

Hello and welcome to my talk. My name is James Palmer and I have been developing Progress applications for over 12 years, in a variety of scenarios.
I currently work for Inenco Group – an Energy Consultancy based in the North West of England. We have a 400GB Progress Database with around 200 users.
My aim today is to equip you with processes that you can take back to your place of work, wherever that is, to examine, evaluate and improve problem queries in your applications. This isn't an in depth look at how Progress selects indexes, or how it actually does the queries. That is beyond the scope of my expertise and knowledge.
The motivation for this talk comes on the back of me having recently taken on the role of part-time DBA for the company I work for. One of the most common things I come across is people saying that the system is running slow, please tune it. Whilst I have a list as long as my arm of activities to undertake from a DBA perspective, I can only improve things so much at the system level. If a user is running a query to retrieve all orders for last week for one customer and it's taking half a day, then it's probably more than just a system issue. There has to be a query issue. So I sit down and do some analysis as to what that user is actually doing and find that to retrieve last week's orders for one customer they are reading all orders for all customers: millions of records to return a handful to the user. The major performance win is going to be at a code level.
From reading common questions in forums I see that this is a problem that rears its head all too often, and people don't know how to determine what is going wrong, and are using bad practise in their queries, leading to disgruntled users who feel like they are wading through treacle.
Today I hope to give you an overview of practises and tools available in the community to help any developer to sit down in front of a poorly functioning screen and work out what's wrong, fix it and become a hero!

In very broad strokes, the process is:

1. Identify which query is bad;
2. Identify why the query is bad;
3. Fix the query;
4. Be a hero!

But how to do that effectively is rather more involved, as I'm sure you know, or else you wouldn't be here. Hence this talk!
As a slight aside, all links I refer to here, as well as any community applications I talk about are linked on my website - http://www.jdpalmer.co.uk

Identify which query is bad

Before we can look in detail at tools, we need to have an understanding of the various approaches we can take to identify where things are going wrong. This in itself depends on the context of the scenario. So we need to take each scenario in isolation.

We know exactly which query is wrong, but not why.

This is most likely to be an in-development problem. You've just written some code, and you're testing it. It takes longer than expected to complete. You can therefore pinpoint with accuracy exactly which query is the one that's broken, you just don't know why. We need an approach to deal with this.

- **Compile with Xref**

This is a really good starting point. It helps us to identify what index each part of a query is using, and also tells us if there are any WHOLE-INDEX reads going on. WHOLE-INDEX reads should, in general, be avoided unless necessary. As the name suggests, this is where the whole table is read. Obviously there are scenarios where you do want to read the whole table, but in a lot scenarios they are a good indicator that something is wrong. Particularly if your query has WHOLE-INDEX reads on multiple tables. Say you have 10000 Orders, and 20000 Order Lines, and you're joining both tables with a WHOLE-INDEX on both, you'll suddenly find yourself with two-hundred million reads to get your data.
Checking the indexes that are being used can help us to identify what's wrong too. In general we want to narrow down the results of our query as quickly as possible, and if that's not happening the way we expect it could be an indicator of a query problem.
Note: DO NOT USE USE-INDEX except in extremely mitigating circumstances. Progress knows how to select indexes much better than you do, and even if you are selecting a "better" index now, what happens when things change in 6 months? Also, by using USE-INDEX you are stopping Progress from bracketing on multiple indexes which in a lot of cases will make the query quicker. In order to get a different index selected, change your query predicate to make this happen.

- **What records are being accessed?**

We'll look at how to do this later on, but knowing what records are being accessed by a query is a really useful way of determining where something is wrong. You should have a fair knowledge of what numbers of records of each table you should be getting back. If you're actually accessing a whole lot more then something is possibly wrong.
Another indicator is if there are a lot more index accesses than table accesses for each table. It's not necessarily the case that this is due to bad index selection or a bad query, but it's an indicator to add to your arsenal.

We should now know which section(s) of the query are causing problems and can look at ways to improve them. I'll touch on some common mistakes people make at the end of this presentation.

A whole program is slow

I get this quite a lot, particularly with report extracts that people have written, usually without much understanding of Best Practise. The report is extracting 1000 lines of information, but it's taking all night to do it and it's having a knock-on effect on other extracts. How do we go about examining this?

- **Compile with Xref**

You can probably see a theme developing here, but having this is a great starting point. Compile the whole program with Xref and look for WHOLE-INDEX queries and correct index utilisation. Fix these as necessary and test again.

- **Run some sort of profiling**

Now we need to try and identify whereabouts in the code the problems are happening. So we run the code through some sort of profiling. I'll look at options later so I'll leave this vague for now. If the code really takes hours to run then you may find this harder to achieve, but there may be some benefit to splitting it down into chunks, or restricting the data volume somehow, at least as a starting point.

- **What records are being accessed?**

And we're back at this again as hopefully the profiling will help to identify around about where in the code the problems are occurring. We can narrow down to the particular block or query and hopefully run it in isolation, observing record access as before.

A user calls complaining their session is slow

This is something I come across quite frequently. Someone calls saying their session is slow, is the database under heavy load; is there something strange happening; that sort of thing. Sometimes there is something as a DBA I can pinpoint, but first of all I want to know what they're doing. They'll often give a vague description of a part of the system I don't really know. I need a quick way of finding what's going on. Thankfully since 10.1C there are a couple of options.

- **Statement Cache**

This is something that has to be enabled on the database, so you'll need to find a friendly DBA (yes they exist!) to assist you. You need to find the user number of the user complaining (The DBA will be able to assist you here also). Then in a Promon session select R&D, Option 1, Option 18; you really only want to have this on for the user in question and not for all. Certainly in some versions of Progress it is pretty buggy having it on all the time. There is an option in Promon to view the cache for a user (Option 7 on the Statement Caching Menu).

Here we see that user number 205 is currently running line 9655 in Internal Procedure AddRenewal in FullForecastAD.p. I can now very quickly find what code they are running. Alternatively you can view Statement Cache in ProTop or other tools available. More on that later.
As a note, the cache is updated each time the client session hits the database, so if the session isn't hitting database records then you're not going to get much joy from this approach.
Remember to turn it off again once you're done!

- **proGetStack**

If you can get on to a client machine then proGetStack <pid> in Windows (kill -USR1 <pid> on UNIX) can be easier to run than statement caching. Again this was made available in 10.1C. Find the PID of the process that's causing the problem and run it in a proenv session on that machine. It creates a protrace.<PID> file in the Start In directory of the client session. It contains a stack trace of where they currently are along with various environment settings such as PROPATH etc and the databases connected.

You can see here that I was currently at line 5652 of icmasstub.w, here are my startup parameters, here are persistent procedures and classes, the propath and the connected databases. All useful information for tracking down why something isn't working as expected.

- **What records are being accessed?**

Now we're pretty much back at the same scenario as before, but if the statement cache option didn't yield any results, or the cache was showing a vague pointer to where the problem's occurring, or you can't get your DBA on board, then you'll want to try and find what records they're accessing. Progress handily gives you the option of seeing what records each user is accessing, which can be a major help here. I'll go into this in more detail later.

If no records are being accessed, then you won't get a statement cache, and you won't get anything here, which would suggest that there's another issue in the code which has halted operation of the session or some such thing. It is unlikely to be a database problem, and you'll have to get as much info as possible from the user and work out what's going wrong the long way by trying to replicate the issue.

Once you've identified the cause of the problem you can follow steps as above to identify the broken query. Obviously speaking to the user and getting as much info from them is really important, as a particular combination of filters in a selection screen could be the cause.

Identify why the query is bad

So now we've identified the bad query we need to examine why it's bad, and for that there are a number of tools out there to do the job. Some of them are good, some of them are not so good, but they all have a part to play. So we'll look at a selection now. This isn't an exhaustive list by any stretch of the imagination. I'm sure there are plenty others out there. These are just the ones I use or am aware of.

Time based profiling (etime, mtime, etc)

Using time based query profiling can be really misleading. For example, the first time you run a query, the records are probably not in the buffer cache. You might run some code with timings, update the query a little, and run again and it will be quicker, but you haven't actually solved the problem.

Consider this simple query on the sports2000 database:

The first time I ran this (on a database that had just been started up) it took a little over six seconds. I then ran it again straight away and it took 1.5 seconds to complete. Why? Well most likely because the first time I ran it the records were loaded into the primary buffer, meaning that the second time I ran it the data was read from memory rather than disk – much quicker. If this had been a problem query and I'd just made some tweaks to it and rerun it, I might be fooled into thinking I'd solved the problem when in fact I hadn't.
Time based profiling will also include the time taken for a user to enter data, or time spent with messages on the screen so is unreliable.

Application Profiler (Time based profiling, plus)

The application profiler is a well-kept secret provided by Progress for profiling applications. It mainly uses time based profiling, but the way it presents the information, and because it excludes think time means it is the exception to my rule of not using time based profiling. If you have a piece of application code that runs slowly, but you're not sure which particular piece of the code is the bottleneck you can run it through the profiler to help you establish which part is taking the time. It will also identify how often various blocks and procedures are run meaning you can really tune up your code very quickly.
Progress has put in some serious work on the Profiler in 11.6 and Nischal Reddy is giving a talk specifically on this in the next time slot. I won't therefore go into any more detail on this now in the interests of time. In the meantime there is a better option from the community that I will be looking at in more detail later on.

Hidden start up parameters

Progress has provided a couple of client startup parameters to help with tracking down bad queries. They are undocumented as they may be removed from the product at any point, but as yet that hasn't happened. The parameters in question are –zqil and –zqilv. These parameters will ensure that query information is output to the database log file. There is quite an overhead on them so they are not recommended for use in production. Additionally they can very quickly fill up the database log file. –zqilv is slightly more verbose than –zqil.
These parameters are probably of most use in situations where you don't have the source code available, but there is a problem with the query, as you can identify indexes used, and, more usefully, WHOLE-INDEX reads.
Unfortunately there is also a lot of noise that is of little interest, as we have other more useful tools available in most cases.
If you are really interested in this approach then there is a brief knowledgebase article (number 21216) on the subject.

Table and Index Stat tables

By far the best way, in my opinion, to establish why a query is bad is to look at the table and index activity for that query.
As a slight aside, in order to get accurate stats from these VSTs you need to set the –tablerangesize and –indexrangesize parameters correctly on your database. This enables Progress to collect stats about all the tables and indexes in your database. By default they are set to 50, and unless you have a very lean system this won't be enough at all. There's some simple code for calculating the minimum level to set these at on my website under Tips and Tricks.
The two tables _TableStat and _IndexStat provide you with information about table and index reads across the whole database. It's not hard to write code to query these over a specific timeframe, but if you're looking in production with 200 users connected and working you're not going to see much useful information.
That's where _UserTableStat and _UserIndexStat come into play. They are stats for each user connected to the database, meaning you can see what each user is up to. This can be hugely beneficial for all sorts of reasons. We looked at why this information could be useful earlier in the session.
You can roll your own code to query these stats, and it's probably a good learning exercise at some stage so you know what you're looking at. If you do, be aware that these 2 tables don't have indexes that allow you to query by user number directly. There is a trick though, and full details are on the Tips and Tricks section of my website.
Thankfully there are a number of tools out there that have already cracked this particular nut, and in the interests of getting up and running using this approach I would recommend using one of them. More on that shortly.

Client Logging

Client logging was added to the product in 9.1D01 and has been carried over ever since. It gives you the ability to log all sorts of things on the client in a specific log directory. There is a whole raft of logging available, but for the purposes of this presentation we are interested primarily in the QryInfo logging.
There are two ways of starting up client logging. One is with client startup parameters, or you can do it programmatically using the LOG-MANAGER handle.

LOG-ENTRY-TYPES is a comma delimited list of values, where the number after the colon is the level of logging for that entry type. Further details are in the documentation.

QryInfo yields all sorts of information about the query just run, but most usefully there is information on each table and the index used. There is also information available on the exact number of records read for each table, as well as the actual number of records returned to the client.
The nice thing about Client Logging that you can't easily get from other sources is the details of indexes used by dynamic queries.

ProTop

ProTop is an absolutely brilliant tool, primarily for DBAs. It was created by White Star Software, DBAppraise, Consultingwerk and Dot.r Limited. It is completely free to use and provides all sorts of useful information and metrics for your database. In my opinion it should be installed on every server that is serving Progress databases.
One thing it does well is to provide information on a user's usage of tables and indexes within the database, so it's great for tracking down bad queries.
We talked about Statement Caching earlier, and ProTop is an excellent way of viewing this information in real time. Run ProTop, Enter U for User Information Viewer, Select # to enter the user number and voila.

Here we see that user 212 is reading a rather large number of Recommendation records, and that it's happening in a program called CheckRenewalsV2.p at line 6595. This last information is from the Statement Cache we discussed earlier. From here we could go to the code, find that line and work out why the user is reading so many records.
The difficulty with ProTop is that whilst it provides all sorts of really good information, it runs best in shared memory on the server and that isn't practical for developers to use in many cases.

Session Trace

Finally we have Session Trace created by Keith Sudbury (or TheMadDBA). It is essentially an extension to the Application Profiler we looked at earlier. In addition to the time based profiling though, it also provides table and index activity information.
It's as easy to switch on as the Application Profiler using some simple code. In order to do this you need to place SessionTrace.cls and include/SessionTrace.i in a location within your propath. Then run the following code:

The UseProfiler option tells SessionTrace to also enable the Application Profiler and collate that information. It does mean that there's more of an overhead to the data gathering, but I feel the information it provides is worth the slightly longer run time.
When you're done with your tracing, all you need is to

We have this set against a hotkey in our application so we can run Session Trace output from any session at any time and then grab the output for analysis; really helpful if it's hard to reproduce a problem other than on a client machine.
The example above will output a JSON file to load into the GUI viewer. Alternatively you can use WriteHTML(*filename*) to output an HTML report if the GUI viewer isn't practical for you.
The DB stats that are written to the output are for the current session only which is something to be aware of if you run code on an AppServer. We are in that position, but I have the luxury of having a local copy of the application to run against, which means that I know I'm the only person connecting to that database at any time. So I can rely on the stats for the whole database for the duration of the test meaning I can profile reads on the AppServer, even if I can't profile what the code is doing. Keith has kindly added a setting to force full DB stats rather than user stats.

Another option you have is to track various variables and their values.
This can be really helpful for tracking down what value various variables have during query execution. This has to be set within the code that's running though so is only useful if you can edit the source before executing the trace.
Another feature that Keith has just introduced at the time of writing this is the ability to have the trace facility running and only to produce information on queries that run over certain thresholds. This could be really useful in an AppServer environment to tackle the occasional problem query more proactively. As it's a new feature I haven't had the chance to play with it yet.
The GUI viewer is really feature rich. The best bit is that you can pretty much change the order and size of everything displayed, and even save that off as your default layout for the future.
You can import an existing profile file, run some existing code, or even write some code into a scratchpad to execute and profile.
So I want to profile a piece of code that seems to be running slower than it should:

You can see that the user is able to give a comma delimited list of Countries and we want to pull out all Order, Order Lines and Items. I can see that someone has attempted a fix already by caching Items in a temp-table. The eagle-eyed among you will see very quickly what the problem is here, but it's a good example for SessionTrace to see what's happening and why the query is slower than expected.
You can also see that I'm a bit worried about the values of Country that we are being supplied so I'm adding a Debug to the trace for that.
I can now load up the resultant Trace file into the viewer and get to work on analysing the output.

You can see I've got 4 Grids enabled. Session snapshots which is all the snapshots loaded into memory. That gives a brief overview of the activity of each session which is useful if you've run the same Ad-hoc code over and over in the session whilst tweaking it. Here we've just got the one file loaded so it's less useful, but could help you see straight off that something is wrong.
Then we have Line timing for the snapshot. This is in essence the Application Profiler output that has been captured.
In a lot of cases this is enough information to find a lot of the problems. It certainly is in our case. I can see that fn-CacheItems() is being run far too many times when it should in fact only run once. We can see also what percent of the snapshots execution time each line took up. This is really useful for pinpointing exactly where the query is going bad.
Sometimes, as explained before, execution time isn't enough and we need to look at the effort on the database. The 3rd Grid I have enabled is the Table Activity one. Here you can see each table that is read and the number of records read. I know that table Item has 55 records, but it is showing 3108 reads. Alarm bells should definitely be going off now. In fact we spent over 75% of the snapshot's effort reading the Item table which should actually hardly be read.
The 4th Grid is the Custom Debug Information one. We could in theory have put anything in here, but in this case it confirms the Countries we are selecting are ok. But it would be quite useful to see if a variable has a very weird value here.
From running this simple tool I can home in directly on the problem in my code and fix it. It saves me looking in two or three places at once, and is really flexible. Also, Keith Sudbury is actively enhancing the application and is open to feature requests. There's a link to this on my website. Go take a look and see if it suits.

Summary

We've seen a number of tools available to us for evaluating queries. Some of them provide compile time information on the queries, others run time information. Of the run time information we've seen that some are time based, and others are effort based. All in all I would heavily recommend an effort based approach to query evaluation as it is not impacted by other external factors. That being said, if you can combine the two approaches in one then it's even better.
Which tools are available to you is up to you, and/or the structures in place. But I would personally strongly recommend implementing Keith Sudbury's Session Trace tool in your workplace as it provides both effort and time based profiling in one package. I would also reiterate my recommendation for installing ProTop on your database server.

Fix the query

Now to spend some time looking at some common query mistakes, and how to possibly fix them, as well as looking at a few more general hints and tips.

Function on the left of the predicate

Consider the following query:

It's a very simplistic example, but as soon as there is any type of function on the left hand side of the predicate, you get a whole index read.
In the Sports2000 database this gives 15 results, but reads all 56 records to get there. I come across a lot of this sort of example, although often more subtle than this and it leads to a lot of wasted effort.
You really don't want to be doing this sort of thing in production as it will provide terrible performance in a situation where there's more than just a handful of records.
Another example of this sort of query is something like

This is a fairly contrived example, but the idea is that you're allowing the user to input some options for Carrier and then returning all orders for that Carrier. This is automatically a whole index read. A fix for this debacle depends on what you are trying to do, but it's highly likely a dynamic query will serve the purpose you need. More on this later.
Let me also state right now, CAN-DO should not be used for this sort of thing. Use the LOOKUP function. CAN-DO is designed for access validation of Users. It has a load of special characters and cases that will mean it doesn't do what you expect and can cause very strange results. Check the help files for the command to see what I mean.

NOT or NE in the predicate

A NOT or a NE in the clause will always result in a WHOLE-INDEX read.

This query reads the whole family table whatever you do. Avoid it. Again depending on your requirements the solution will vary, but try and turn the negativity into positivity! Easily done with a logical like here, but could be a bit more tricky with a different data type.

Matches in the Clause

Any query with MATCHES in it will result in the whole table being read, whatever the pattern you're matching with. Progress is unable to select an index.

This behaviour is unlike SQL which will select an index based on the pattern given. Out of interest, and something new developers often fall foul of when they come over from the SQL world, Progress does not do cost based optimisation of queries. The index utilisation

for static queries is done at compile time. For dynamic queries it's slightly different in that by definition the indexes have to be resolved at run time, but the rules for the indexes will be the same as the compile time selection.
So as tempting as it might seem to give the user the option to provide a partial match for their enquiry screen, it's a recipe for disaster. By all means give them a BEGINS option, so long as the field you're allowing them to query on is in an index.

Change the table order

Sometimes the best way to improve a query is as simple as changing the order of the tables in the query around a bit. Often it's a case of trying a few different options to find the best one, but sometimes you can spot the options very quickly. Consider the following query:

It reads a whopping number of records for 234 results:

If we swap this around to read OrderLine first

We get the following reads:

And if we swap this around to read the Item first

So you see just playing around with the order of the tables we can hugely affect the reads the query does.

Dynamic Queries

Consider a screen where the user is given the option of filling in various bits of information. We then return the results that match those criteria. This can lead to pretty awful query statements, trying to handle all the different scenarios that a user could select. We have quite a lot of these spaghetti queries in our system because it's a legacy system that's been around 20 odd years. In more modern releases of Progress we can mitigate against these queries by introducing dynamic queries. Many of you have probably used these extensively, but just in case there are folks who haven't, and we get a lot of users who haven't in the forums, I'll go through a simple example of why they're so useful.
Consider the sports2000 database. We want a selection screen for orders which have a line over a chosen value, from a chosen country, and city, with a particular carrier.
To represent that in a static query you're going to end up with something like this:

Look familiar? We are obliged to use every table, and to have complex ways of dealing with the possibility that the selections may or may not be there.
If we convert this to a dynamic query we see that we can greatly simplify the actual query run. In fact, if the user doesn't provide certain criteria at all we don't need to even include that piece of the query. In this example, if the user doesn't specify a minimum price we can bypass querying the OrderLine table completely!

Now it's simple to run the query.

Fn-GetTables() is a little routine for parsing the query string and working out the buffers in use. There's a copy of the code for that on my website under Tips and Tricks so I won't reproduce it here for the sake of time.

If we run it for all cities starting with 'B', with a carrier of Standard Mail we get 118 results.
The static query read records as follows:

The dynamic query read as follows:

2012 total reads in the former, vs 1893 in the latter. In this particular case it hasn't made a huge amount of difference because of the indexes used and the lack of actual data available in sports2000, but you can see how it could make a massive difference in a large database with a lot of different filters.
It's also a lot, lot easier to maintain!
If we take out all the filters we see a much bigger difference: nearly 4000 less reads. This scenario in a real database would be massively different for the user.

Remember, as always in Progress, if you create it then you should also delete it. When you are finished with a dynamic query, close the query object and delete the query associated with the handle. This ensures you don't get a memory leak which in itself can have a huge impact on performance if lots of them are left hanging around.
As a slight note, whilst preparing this section of the presentation I was having some real problems with the dynamic queries. They were reading a lot more records than I was expecting. In the end I tied this down to the –rereadnolock client startup parameter. This is a really important parameter to have on but in multi table dynamic queries it leads to extra reads. There is a way to stop this though. There is an attribute of CACHE you can set on the query handle, which specifies how many records of the query to hold in memory. It's a good idea to play around with this if the performance of your dynamic queries isn't what you expect. As always refer to the documentation for more information.

Using ranges to avoid a dynamic query

People will go to great lengths to avoid dynamic queries. A lot of this is for legacy reasons as I've already stated. It can also be because dynamic queries are considered to be too difficult to implement. The thought process behind anything dynamic can be a little tricky to start with.
As a result you get some sickening solutions to the problem. Some of you may well recognise code like this:

Awful! Guaranteed to pretty much bomb on every execution. Don't do it. Take the time. Write a dynamic query. You'll thank me! Particularly when you're asked to add another filter to the screen.

BREAK BY on the wrong field

BREAK BY can be a real help. It has its place. But it can have a detrimental impact on performance. Only use it when necessary.
Here we see an example where the BREAK BY is on the wrong table:
Here are the reads for this.
If we then change it to be on the CustNum of Customer we get:

So we see that having the BREAK-BY on the wrong table greatly increases the number of reads on both tables.

Logic in the Wrong Place

It's very simple to get logic in the wrong place, and sometimes this can lead to really slow query execution. I see a lot of code where inside a query we test for something and then say next if that test isn't met. Now sometimes this is necessary in order to force the compiler to select a better index, but using NEXT should be the exception not the norm. And if you are going to do it, make sure it's as soon after the query predicate as possible so as not to do unnecessary work.
Consider the following:

Here we're doing a big chunk of work before we decide to NEXT on the query. The resulting reads are:

If we rework this test into the query predicate itself then we get the following:

This is a huge improvement.

Table-Scan

I mentioned earlier that there are cases where a full table scan is necessary. If you're in a situation like that and you don't mind what order your records come back, then the TABLE-SCAN option might be for you. It was introduced in version 11. So long as your database table resides in a Type II storage area (why on earth it wouldn't in a version 11 database is beyond me!), then this will retrieve the records in the most efficient manner without using an index, The records are returned in the order they are found. This could help to speed up a whole table read, in some cases significantly.

Local caching

From a DBA perspective, small static tables that have a lot of activity on them are perfect candidates for putting into something called the secondary buffer pool. It's essentially a second server side cache meaning those records are read from memory rather than from disk every time they're accessed.
Consider a query in a client server environment that reads the same records over and over again; particularly if they're reasonably static tables. Each record that is read has to be brought across the network from the database each time it is used.
So can we somehow cache things locally? Yes. Create a temp-table for the records which you prepopulate before running your queries. If you're really canny you can define the temp-table with only the fields you actually need, and then populate them using the FIELDS phrase to reduce the amount of data pulled across the wire. You could even use the TABLE-SCAN option discussed earlier to make the prepopulation even quicker.
In newer versions of Progress you can also cache the records in a class.
This is a very trivial example, but it hopefully illustrates the point. We want to get all Shipped Orders and Purchase Orders for Golf items and do something with them.

This reads as follows:

If on the other hand I cache the records for Item in a Temp Table first I get a slightly better picture.

You can see that the reads are near identical, except for the Item table where we're reading over 3000 less records over the network. And the network traffic we are reading is reduced as well due to the fact I used the FIELDS phrase to just pull back what I need.

OR in the predicate

Having lots of ORs in a query can lead to poor index selection by the compiler. In a lot of cases you'll get away with it, but even if you do, the query can be pretty nasty to maintain. It's much better to populate a temp table with the information you need and add that to the join. Look at this query in the system I work with every day:

This will return all Meter Sites for Electricity and Water; 268097 of them in fact. Reads were rather higher:

A whole read of MeterSite is the culprit because Progress wasn't able to use any meaningful index on ProductGroup. Index reads were even higher:

We can actually use the principal we saw a moment ago with local caching to help us a great deal here. We can create a temp table containing the information we need and use that in the query to greatly reduce record reads and improve performance a great deal.

The original query took around 28 seconds to complete. The new one less than 5.
We still read a lot of MeterSites but it's a good start.

If we then go a step further and flip the predicate we get some fantastic improvements.

AppServer

Another way of reducing the amount of data coming across the network is to send the query to an AppServer process. A lot of our enquiry screens have a test in them. If certain primary filters are provided then the query is executed locally because we know that a good index will be hit. But if that core information isn't present we send the query string over to the AppServer which then executes the query on a shared memory connection to the database, builds a table of results and then passes that back to the client. You have to be a little careful if the query results in a massive result set as that can cause memory issues, but in most cases it's fine and will most likely be significantly quicker.

Be a Hero!

All of this talk of how to fix broken queries is really helpful at getting a foothold in being a hero for your users, but putting out code that is potentially broken or underperforming and waiting for customers to complain about it is a bad scenario to be in. You wouldn't do it with anything else deliberately, you have processes for testing and so on for code. You need to be proactive in your query tuning BEFORE shipping your code. Just saying "Oh it runs fine in development" isn't good enough. Development runs on different hardware, has fewer users, probably has less data, etc.
The processes you can implement will vary from situation to situation, but I would suggest that it should contain the following steps:

1. All code to be compiled with XREF before shipping. The XREF should be checked for WHOLE-INDEX reads and complex queries should be sanity checked on index selection.
2. Code should be profiled. Either with the application profiler, or with Session Trace. Session Trace negates the necessity to do step 3 as the two are combined into one. We should check the profiling to check that blocks of code aren't being run unnecessarily and check the sections of code that take a long time to complete (as a % of total execution time) that they aren't causing bottlenecks.
3. Table and Index reads should be examined to check that the number of records being read versus the number returned to the user is acceptable.

Taking the extra time to do this testing will mean that the chances of getting THAT phone call will be greatly reduced. Your customers will be happy, and you'll be a hero. Maybe not one that everyone talks about, because actually they won't notice you – they will just be happy, getting on with their job, quickly and efficiently. But you'll be an unsung hero, and those are the best kind.

Thank you for listening to me. I hope you have found it useful. If there are any questions or comments then now is your chance. Alternatively feel free to catch me at any point for a further chat.

Questions?