When it comes to application performance, faster is better. In part one of this series, we offer a tried-and-true method for troubleshooting application performance with a real-life example of how we applied this process to a customer’s application.

Over the years, we have seen many brilliant Domino applications. Inventive architecture, thoughtful user interfaces, precise code -- we’ve seen all of this and more. Of course, we’ve also seen Domino applications with performance problems. There are many causes for these problems from architecture to code inefficiencies. What has often made the situation difficult for the customer (and for anyone troubleshooting these problems) is limited access to and understanding of troubleshooting tools and techniques. In this two-part article series, we focus on indexing and coding techniques and on new tools to help users locate code trouble spots.

In part 1 of our article series, we explain a tried-and-true performance troubleshooting process along with an example of how following this process quickly narrowed down a vexing performance problem for a real-life customer application. We also cover some tips to help you optimize view indexing and agents. The second article of this series will cover new Lotus Notes/Domino 7 tools for monitoring and troubleshooting agent performance problems. This series is intended for experienced Notes application developers.

Identifying the problem

A slow application (or server) can trigger a series of escalating phone calls through an IT organization, including calls to Lotus/IBM support and possibly to consulting services. There are two ways to attack this problem. The first is to identify bottlenecks and to widen them. By definition, this is a continuous process, but usually there is a fairly clear point of diminishing returns. The second way is to architect and plan your application and server so that you don’t run into these bottlenecks in the first place (easier said than done). We deal primarily with the former approach here, but suggest ways to implement the latter as well.

We have found the following four steps useful in identifying bottlenecks.
Get details

When told that an application (or server) is slow, the first thing to do is to have a serious conversation with someone knowledgeable about the application and to ask some pointed and detailed questions. From many years of asking these questions, we found that three key areas, as follows, often yield meaningful answers:

1. Where is the problem (topographically and chronologically)? We want to determine if the slowdowns occur in the Domino application, the server, the network, and so on. Some questions that we found helpful are:
   - When the slowdown occurs, does it occur for all actions you take on that server or actions only taken in one application?
   - If all actions on the server are affected, then what about other servers? Even file servers?
   - If all actions on the network are affected, then is it isolated to people in your building, wing, or floor?
   - From a slightly different angle, do these problems occur at a particular time of day? Only on certain kinds of machines? Older machines? Machines with the same OS?

2. Where is the problem (actions)? We want to determine which areas of the application, or which specific user actions, are relevant to the performance problem. We want details, for example:
   - Does the slowdown affect you when you open the database? Open views? Scroll through views? If so, are all views or certain views affected? If only certain views are affected, identify at least one or two of these slow views.
   - Does the slowdown happen when you create, edit, save, or read documents? If so, specify the actions, and whether this holds true for documents of every form or only certain ones. And, of course, specify which forms.
   - Does it occur when you click a particular button? This is the dream scenario because if we can identify a specific button, then we have a particular piece of code to examine.

3. What changed? In nearly every instance, the customer thinks that nothing noteworthy has changed. If the customer knew something important had changed, he would have tracked down that change instead of talking to you! Some focused questions that help elicit facts may be:
   - Has this application (or server) always been slow?
   - If not, did it slow down when more users were added? More data?
   - Was new functionality recently added to the application?
   - Was the software or hardware recently upgraded?

Theorize

From the answers you receive in response to the preceding questions, start putting together one or more hypotheses about the problem. For example, hearing that the application is slow whenever someone works with a document of form Main may lead you to conclude that something about the code in that form needs work. You could then have an application developer review the form with an eye toward improving performance. Or perhaps the application is slow whenever users click a certain button in which case you know to look directly at that button code.
On the other hand, if you hear that performance seems to be slow frequently throughout the day when performing any operation and that the server as a whole seems to slow down, then that may indicate that server tasks are being overloaded. A common one to examine first is indexing. It is not uncommon for applications to be written in such a way as to cause frequent or even constant indexing, and as the load of users and data increases, the indexing work gets harder and harder. In fact, we have seen production servers in which the indexer worked constantly to keep the views updated. In those cases, there is little room to provide CPU cycles to user requests, and users often feel that those servers are unresponsive.

**Data collection**

In any case, wherever your nose leads you, you want to collect data to verify or dispute your theory. This is an iterative task, so ideally, you want steps one and two to take hours, not days.

Depending upon your theory, use different tools to collect data. In any release of Lotus Domino, you have Notes.ini logging at your disposal, and this can be tremendously valuable. You can turn on `log_update=1` if you think that view indexing is problematic. Putting this value into the Server Configuration document pushes it into the server's Notes.ini file, and you can let it remain there for a day before pulling it out again (because this setting does not normally write too much information to the log file, many customers leave this setting in place indefinitely). Then examine the Miscellaneous Events view of the server's `log.nsf` for that day to see what the indexer is working on and for how long. You may see entries such as the following:

<table>
<thead>
<tr>
<th>Date</th>
<th>Time</th>
<th>Event Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>01/04/2005</td>
<td>11:35:47 AM</td>
<td>Updating views in Merkle\CM.nsf</td>
</tr>
<tr>
<td>01/04/2005</td>
<td>11:36:22 AM</td>
<td>Finished updating views in Merkle\CM.nsf</td>
</tr>
<tr>
<td>01/04/2005</td>
<td>12:02:17 PM</td>
<td>Updating views in Merkle\CM.nsf</td>
</tr>
<tr>
<td>01/04/2005</td>
<td>12:02:39 PM</td>
<td>Finished updating views in Merkle\CM.nsf</td>
</tr>
<tr>
<td>01/04/2005</td>
<td>12:36:04 PM</td>
<td>Updating views in Merkle\CM.nsf</td>
</tr>
<tr>
<td>01/04/2005</td>
<td>12:36:15 PM</td>
<td>Finished updating views in Merkle\CM.nsf</td>
</tr>
</tbody>
</table>

**NOTE:** Typically, you would see hundreds, if not thousands, of lines of other logging information between these six lines, so search and then copy/paste the information into a text file so that you can collect the relevant lines together like this for easier analysis.

Each of the Domino tasks has similar kinds of logging abilities, whether it is replication or agent manager, so it is fairly easy to collect data on any of these tasks.

**Analyze and repeat**

Analyze the data you have collected, determine whether or not you are on track, and then repeat the process of asking questions, theorizing, collecting data, and analyzing that data.

**Troubleshooting example**

Over the course of many years, we have had the opportunity to review many dozens of customer applications to assist in optimizing for performance. A majority of the performance problems we have seen have fallen into the following few categories:

- Form design
This is a topic for another day. In general, you need to optimize performance for edit and read modes, keeping in mind typical usage.

- **Agents**
  We cover troubleshooting and some code tips in this article.

- **View indexing**
  We cover troubleshooting and some code tips here as well.

The following is an example from an actual real customer problem: The customer reported that users were experiencing slow performance in all activities in an application throughout the day. There were times of good performance, but generally the busier times of day corresponded to poor performance. Further questioning revealed that virtually any user actions could be fast or slow, which made us disinclined to pursue the code behind the forms, agents, and various other database actions. We felt confident that there was at least one server task that was probably hogging the server resources. The fact that users reported that opening and scrolling through views was often very slow clinched it. We decided to pursue view indexing.

Using log_update=1, the customer collected indexing data for a day, and then we looked through it and pulled out all the lines that related to their database, Merkle\CM.nsf. Six of these lines are shown earlier in the article. For search purposes, we like to use pdating views in xyz.nsf, where xyz.nsf is the path and file name of the application we're interested in. By dropping the leading u, we don't have to worry about case-sensitivity.

What we found interesting about these six lines is that they indicated that the update task was taking 30 minutes to complete its cycle. Normally, the update task runs every 15 minutes, and this is non-configurable on your Domino server. If you see that it takes more than 15 minutes for the update task to return in its cycle to a particular database, then only two explanations are possible: The database had no activity for a period of time, so the update task skipped it for one or more cycles. This is exactly what you would expect at night and on weekends, but for a busy database in the middle of the day, it is not likely to be true.

The update task is taking more than 15 minutes to finish updating all the views in all the databases that need updating. In this case, you will typically see sluggish performance during these times because the server is working hard to fulfill the view indexing needs.

In this customer's case, we were confident that the latter was true, and this was valuable information for them. But it didn't solve the problem. The next step was to start the four-step process of questions, hypotheses, data collection, and analysis all over again.

If the update task is taking approximately 30 minutes to complete, we can look in the log file at the same Miscellaneous Events document to get a picture of why it's taking so long. One possibility is that we'll see a tremendous number of databases and views. So many that, although each one is indexed very quickly, there is just no way to index them all without taking 30 minutes to do so. This may lead us to ask whether or not the server has been overloaded with databases.

Another possibility is that there aren't so many databases and views, but they are very time consuming. If this were our theory, we might change our logging to log_update=2, which will
record a time stamp for each view that is indexed, instead of at the database level. Then we could
examine the specific views that appear to be taking so long to index. This setting will generally
write a tremendous amount of data to the log.nsf and therefore, should be turned off, or at least
reduced to log_update=1, as soon as possible.

And yet a third possibility is that there is simply a tremendous amount of data changes taking place
throughout the day.

It is within reason to get answers to the implied questions here from our logging as well as from the
customer directly. They may not have a sense of what quantity of data is too much, but they can
likely tell you how the application is being used and give a rough approximation of the volume of
data being changed on a daily basis.

In the case of our customer, increased logging pointed to a handful of views taking a very long time
to index. We examined those views (some not even in their primary application) and found that
they were coded in such a way as to force a complete view rebuild each time they were accessed.
Because they had @Now or @Today in their selection or column formulas, every time a user
opened these views, the view had to be rebuilt from scratch, a very time consuming process.
Worse still, this was a Domino 5 server, and in release 5, these time/date sensitive views are
updated whenever the update task refreshes any views in the database.

Knowing this, the customer recoded these views so that they met their business needs without
time/date sensitive views. And performance improved markedly right away. The total time spent
on the problem was approximately five days of our time, plus another 10 to 20 days for their
developers to code and test the successful new approach.

**Code tips**

From many years of observation, as well as testing, we have come up with a handful of very
common and critical coding practices that can easily be done well, or poorly, in terms of
performance. Here are a few:

- Time/date sensitive views
- Click-sort view columns
- Getting a collection of documents in an agent

**Time/date sensitive views**

A view that has @Now or @Today in the view selection formula or in any column formula can be
referred to as a time/date sensitive view. From a performance perspective, the implication is that
when this view is refreshed for any reason, it is completely rebuilt. Rebuilding a view may take
50 to 100 seconds, whereas that same view without the @Now or @Today formula may take just
100 ms to refresh. This is a monumental difference when you consider that views are refreshed
hundreds of times per day.

For example, suppose a user opens a view; the view is refreshed. The user edits and saves a
document while in a view; it’s refreshed. The user sees the blue reload arrow lit up and clicks it; the
view is refreshed. In each of these cases, if the view has to be rebuilt instead of merely refreshed, it can be quite a wait for that user and also quite a load on the server.

**Using @Now and @Today**

When @Now or @Today are used in selection formulas, they typically limit the size of the view collection. This reduces the time to refresh and the time to rebuild the view. When @Now or @Today are used in a column formula, they do not similarly restrict the view index size. Therefore, although in theory there is no distinction between using @Now or @Today in selection formulas versus column formulas, in practice it is often the case that the worst performers are those that have these functions in their column formulas.

In Lotus Notes/Domino 6, these views are completely ignored by the update task (see the explanation of the update task in previous section). This means that the view is not exerting a drag by virtue of its existence. Only by user action does the view cause problems in Lotus Notes/Domino 6.

Also, in Lotus Notes/Domino 6, the indexing options (refresh view automatically; auto, at most every n hours; or manually) work well, which means that you can make a time/date sensitive view refresh manually, and it will open immediately for users without trying to refresh and therefore, without extensive work by the server. When the indexing options are set this way, the view is refreshed only when a user presses F9 or clicks the blue reload arrow. Normal views would also be refreshed every 15 minutes by the update task, but because the update task skips these time/date sensitive views, not even that process will cause a refresh (or rebuild).

It may not be a good idea to have a view that is always out of date, but the fact that it's a viable option in Lotus Notes/Domino 6 is in fact a very good thing. If you need to have the view refreshed every so often, you can run an agent to update the view or set the view's indexing option to auto, at most every n hours.

**Click-sort view columns**

This is a brilliant feature, even though it's not new to Lotus Notes/Domino 6. It allows your users to re-sort a view based on any column which you have so marked. And it's fast, providing such a quick re-sort for users that it's easy to believe that the feature is free in terms of performance. But it's not.

From the perspective of performance, every little click-sortable arrow is like building a new view much like the original, but sorted differently. So, if you have a view that is 10 MB in size, and you have four columns that are marked click-sortable (ascending or descending), then your view will now weigh in at approximately 50 MB, and it will take five times as long to index as the original. If those four columns have both ascending and descending arrows, then that's a total of eight arrows, so the view will now be approximately 90 MB and will take approximately nine times as long to index as the original.

You should use this feature judiciously. Marking up all your view columns with up/down arrows is not necessarily making your application any easier to use, but it certainly will make it slower. In particular, you want to make sure that hidden lookup views, which are only used by back-end code, do not have any click-sortable columns because that is a waste.
Getting a collection of documents

Over many years of analyzing customer applications, we’ve seen that agent code (and code in form events, such as PostOpen and QueryClose, in particular) most often gets a collection of documents and then reads from, or writes to, those documents. It turns out that how you go about getting that collection of documents is critical to the performance of the code you write.

To emphasize this point, it may seem natural that getting the appropriate collection of documents would be trivial in terms of performance and that the size of the collection and what you do with it is all that counts. But, on the contrary, how you get that collection can often be a sizable, and highly variable, piece of the puzzle in terms of code optimization. We describe some different techniques, roughly in descending order of performance.

If you get a collection of documents and you need to read some values that can appear in a view, then getting a handle to the appropriate NotesViewEntry or NotesViewEntryCollection using view.GetEntryByKey or view.GetAllEntriesByKey, and then reading the appropriate ColumnValues is the fastest way to get your data. You may wonder whether or not the impact of having to index this view may negate your performance gains when your code executes. Our experience is that it’s not uncommon to see lookup views with several columns of information index (refresh) in less than 100 ms. Therefore, we don’t think it’s really a concern. Certainly, you do want to make such a lookup view, or views, as streamlined as possible, but the savings from having such a view far outweigh the minor cost of keeping it updated.

If you need to read data which cannot, for whatever reason, be reasonably displayed in a view, then you need to get a handle to the back-end documents so that you can access their NotesItems directly. In that case, you can use NotesDocument or NotesDocumentCollection using view.GetDocumentByKey or view.GetAllDocumentsByKey.

**NOTE:** If you are "walking the view" by using set doc = view.GetNextDocument ( doc ), then you want to also use view.AutoUpdate = False, for two reasons. First, it greatly improves speed. Second, it allows you to change a document, even remove it, without having your code fail when you proceed to the next document.

In terms of performance, getting a handle to these back-end Notes documents using db.ftSearch is just as fast as accessing the documents via a view. If your key is from a search in rich text fields, then this is really your only option. Of course, fast performance using this method requires an updated full-text index. Other caveats for using db.ftSearch include the slightly more complicated syntax, and the fact that you are limited in the size of your result set unless you use Notes.ini variables to override that maximum.

From a performance perspective, using db.Search to return a small collection of documents is a poor performer; however, it offers the advantage of performing searches without concern for a view or full-text index being built, making ad hoc queries easy. This means that you can perform these searches against a database owned by other people in which you have no ability to create views or a full-text index. Also, if the collection size is large, db.Search becomes competitive. Once the collection is approximately 5 to 10 percent of the number of documents in the database, it
becomes the fastest method for getting a collection. That means that in a database of 100,000 documents, if your search returns more than approximately 5,000 documents, then using db.Search will actually be your best performer.

**Conclusion**

We hope that the techniques illustrated in this article give you a greater understanding of the process of troubleshooting performance problems and that the four-step process itself can be a useful guide for you in the future. If the indexing and code tips help you create more performance friendly applications, then perhaps it will be a long time before you need to use this process. But if you find yourself needing to troubleshoot performance problems in your applications, look for part 2 in this series on Lotus Notes/Domino 7 performance troubleshooting tools.
Related topics

- The developerWorks: Lotus articles, "Application Performance Tuning, Part 1" and "Application Performance Tuning, Part 2" offer additional insight on how you can make your Notes/Domino applications perform better.
- See the Maximizing Domino Performance White Paper to learn more about improving the performance of your Domino applications (and the servers on which they run).
- Consult the Domino Designer help for complete information about creating Notes/Domino applications.

© Copyright IBM Corporation 2005
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)