Lotus Notes/Domino 7 application performance: Part 2: Optimizing database views

Raphael Savir
February 14, 2006

In part two of this article series, we explain how you can build views that are optimized for performance in your Notes/Domino applications.

View more content in this series

In "Lotus Notes/Domino 7 Application Performance, Part 1," we examined how you can improve the performance of Lotus Notes/Domino 7 applications through the efficient use of database properties and document collections. In part 2, we explain how you can build high-performing views. As in part 1, this article provides many code snippets that you can re-use and adapt to your own requirements.

Over many years of analyzing application performance issues, we found that views are frequently involved in both the problem and the solution. Often, view indexing is the issue. This article explains how this can happen and what you can do to troubleshoot and resolve this type of problem. But there is another kind of view performance problem that has been popping up more frequently over the past few years. This involves views that display reader access controlled documents. The performance problems seen in these views are often not indexing related, so we'll take a little time to discuss these separately.

This article assumes that you're an experienced Notes/Domino application developer.

Understanding view indexing (the Update task)

The first thing you have to know before troubleshooting a performance problem that may involve view indexing is how the indexing process works. Indexing is typically done by the Update task, which runs every 15 minutes on the Domino server. Technically, it is possible to tune that interval, but it involves renaming files, so it is rarely done.

When the Update task runs, it looks for every database on the server with a modified date more recent than the last time the Update task ran. Then the task refreshes the views in those databases. Based on our experience, it is reasonable to assume that it takes approximately 100 milliseconds to refresh a normal view in a production database in a production environment.
The logical question to ask is, "What flags a view as needing to be updated?" Every time any of the following occurs a view requires updating:

- Replication sends a document modification to a database
- A user saves or deletes a document (and then exits the database)
- The router delivers a document

The Update task is very liberal in how it determines if a view needs to be updated. For example, imagine that you're in your mail file and that you change the BCC field of a memo and nothing else. No view displays the contents of that field, so in fact, no view needs to be refreshed. But nevertheless, views that contain this memo will at least be examined simply because the server is not sure whether or not the edits you made will force a change in those views.

There is a twist to this: Users can force an immediate high-priority request to have a view refreshed by opening the view or by being in a view when they make a document change. Let's look at an example.

Suppose Alice opens the Contact Management database to view #1 at 9:02 AM. Suppose also that the Update task conveniently ran at 9:00:00 and will run again at 9:15:00, 9:30:00, and so on. Alice updates a couple of documents at 9:05 AM. She creates new documents, edits existing ones, or maybe deletes documents. In any case, the server immediately updates view #1 because she is in that view. It would be strange if you deleted a document and still saw it in the view, right? So that view is updated right away. But the other views are queued for updating at 9:15 AM.

Now imagine that Bob is also in our Contact Management database at 9:02 AM. He is working in view #1. At 9:05 AM, he sees the blue reload arrow in the upper left corner, telling him that the view has been updated. He can press the F9 key or click the reload arrow, and the view will quickly display the updated contents. He doesn't have to wait for a refresh.

Further, suppose Cathy is also in the database at 9:02 AM, but working in view #2. If she does not do anything to force a refresh in the interim (which is possible, but unlikely), then she sees the blue reload arrow at 9:15 AM when her view is refreshed by the Update task. More likely, though, Cathy makes some data updates of her own, scrolls through the view enough to force an update, or switches views. Any of these actions would force an immediate update.

At 9:15 AM, all the views that these clients have not already forced to refresh are refreshed, and we start all over again.

**Additional information on view indexing**

There are two additional pieces of information that we find helpful. The first is the full-text indexer. Typically, full-text indexes that are marked to update immediately are updated within seconds (or minutes if the server is busy). Full-text indexes that are hourly are updated (triggered) by the Update task at the appropriate interval. Full-text indexes that are daily are updated (triggered) by the Updall task when Updall runs at night.
The second additional point is that the developer can set indexing options in the view design properties. Many people misunderstand these to be settings that affect how the Update or Updall tasks will run. This is not so. The Update and Updall tasks will completely ignore these settings, updating views based solely on the criteria described in the preceding example.

The indexing options affect what happens when a client opens a view. If the options are set to manual, for example, then the user gets a (possibly stale) version of the view very quickly. If the options are set to "Auto, at most every n hours" and if it has been more than the specified time interval, then the user will have to wait a moment while the view is updated, just as if it were a regular view with Automatic as the view indexing option. We will discuss how these indexing options can be used to help craft useful views later in this article.

**Quick tips on troubleshooting view indexing**

In a well-tuned environment, indexing should be relatively transparent; views and full-text indexes should be updated quickly. In an environment experiencing performance problems, you may see the following symptoms:

- Long delays when opening a database, opening a view, switching views, scrolling through a view, or saving a document
- Delays when opening a document that uses lookups (In fact, you may see the form pause at certain points as it waits for these lookups to compute.)
- Performance problems throughout the working day, but excellent performance during the off-hours
- Out-of-date full-text indexes

The following logging and debugging Notes.ini parameters can help you address these issues:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>log_update=1</td>
<td>Writes extra information to the log.nsf Miscellaneous Events view. This consists of one line of information every time the Update task starts updating a database, and a second line when the Update task finishes with that database. Each line has a time/date stamp, so by subtracting the times, you can get an approximate time (rounded to the nearest second) required to update that database.</td>
</tr>
<tr>
<td>log_update=2</td>
<td>Writes more information to the log. In addition to the data generated by log_update=1, this setting adds one line of information for each view in each database. Thus, a database with 75 views would have 77 lines -- one line to signal the start of the Update task against this database, 75 lines -- one for each view, and then a final line to denote the end of indexing in this database.</td>
</tr>
<tr>
<td>debug_nif=1</td>
<td>By far the most verbose way to collect information on indexing, debug_nif will write to a text file (which you specify using debug_outfile=c:\temp, for example). This can easily generate gigabytes of data per hour, so it should be used sparingly, if at all. The value of this debug variable is that it gives you the millisecond breakdown of all indexing activity, not just the Update task running every 15 minutes.</td>
</tr>
<tr>
<td>client_clock=1</td>
<td>Used on your client machine, not the server, this will write verbose information to a text file (which you specify using debug_outfile=c:\temp, for example), breaking down every client action you perform.</td>
</tr>
</tbody>
</table>
To troubleshoot view indexing problems, start by collecting some data from log.nsf by setting log_update=1 (if not already set) and allow the server to collect information on view updates for a day. Then review the log.nsf Miscellaneous Events view for that day and look for patterns. Some meaningful patterns that may emerge are:

- Very long update times for a specific database. This may indicate that the database has too much data being updated, too many complex views, or time/date-sensitive views (if the server is release 5 or earlier). The logical next step is to examine the business use and design of that database and perhaps to use log_update=2 for a day to pinpoint which views are problematic in that database.
- Very long update cycles. We've seen cycles that have lasted four to five hours, meaning that instead of passing through all modified databases every 15 minutes, the Update task may only do so two or three times every day. This may indicate general performance problems affecting all tasks, or it may indicate a very high user or data load on the server. The logical next step would be to assess the general state of the server and the business use on it.

Whenever you find long update times, it is helpful to note which other tasks are running concurrently and whether or not those tasks are behaving normally. It is also helpful to note if the slow indexing times are universal, only during business hours, or only during certain peak times, such as every two hours. It is rare that a single observation will give you all the information you need to solve your problems, but it usually puts you on the right path.

**View performance testing**

To test the performance of various features and ways of building a view, we created a database with 400,000 documents and ran a scheduled agent to update 4,000 documents every five minutes. We then built approximately 20 views, each with slightly different features, but each displaying all 400,000 documents, using five columns. We will list the details later, but the big picture looks like this:

- The size of your view will correlate strongly with the time required to rebuild and refresh that view. If your view doubles in size, so too will the time to rebuild or refresh that view. So, for example, if you have a 100 MB database that you expect will double in size every six months, and if you find that indexing takes approximately 30 seconds now, then you should anticipate that indexing will be 60 seconds in six months, then 120 seconds six months later, and so on.
- The biggest "performance killers" in terms of view indexing are categorized columns. Sort columns add a small amount of overhead as do some other features, such as the Generate unique keys option (more on this in the tips at the end of the article).

Figures 1 and 2 demonstrate both the relationship between size and refresh time and the significant overhead that categorized columns add to your view performance. Figure 1 plots view index size against response time.
In both charts, the first and third categorized views are initially expanded, and the second and fourth views are initially collapsed. Collapsing your categorized views shows a small, but discernible savings in refresh time.

**Reader Names field**

Over the past few years, we have seen a dramatic increase in the number of critical situations related to Reader Names fields and views. Customers find that performance is unacceptable frequently throughout the day. They cannot understand why because the code reviews and pilot tests all went smoothly. But a few months after rollout to their entire department/division/company, they find that complaints about performance are dominating conversations with their employees, and something has to be done.

**Examples**

Our favorite examples are HR and Sales Force applications -- both of which typically have stringent Reader Names controls. The following table illustrates a hypothetical scenario for a company using a database of 400,000 documents.

<table>
<thead>
<tr>
<th>Title/role</th>
<th>Number of documents user can see (out of 400,000)</th>
<th>Percent of database</th>
</tr>
</thead>
<tbody>
<tr>
<td>Corporate HQ, CEO, CIO, Domino administrators, developers</td>
<td>400,000</td>
<td>100 percent</td>
</tr>
<tr>
<td>District Manager</td>
<td>4,000</td>
<td>1 percent</td>
</tr>
<tr>
<td>Manager</td>
<td>400</td>
<td>0.1 percent</td>
</tr>
</tbody>
</table>
Upper management and the Domino administrators and developers typically experience pretty good performance, which makes the first reports of poor performance easy to ignore. Performance problems are typically exacerbated by weaker connectivity (WAN versus LAN), which may also make the early complaints seem invalid. But after some critical number of complaints have been registered, someone sits down at a workstation with an employee and sees just how long it takes to open the database or to save a document, and then the alarm bells start ringing. Figure 3 shows how long it took a user to open a view in the sample database with 400,000 documents. All views were already refreshed. No other activity was taking place on the test server, and only one user at a time was accessing the server.

Figure 3. Time required to open a view in the sample database

Users are denoted by the percentage of documents they can see. If you cannot see a bar, it means that it’s very close to zero.

Before moving on to an explanation of Reader Names fields and performance, we want to leave you with one thought about Figure 3: You can quickly see why users of the flat (sorted, not categorized) views would have such radically different impressions of the performance of this application if they were 0.01 percent users compared to 100 percent users.

Understanding Reader Names

When an application uses Reader Names, it means that some or all documents have Reader Names fields -- that is, fields that are Summary Read Access. Summary means that the data can be displayed in views. Technically, you can create Reader Names fields and prevent them from having the summary flag, but then Lotus Domino cannot properly ensure security of the documents at the view level. Read Access means just that -- who can see this document? Typically, individual names, group names, and [roles] populate these kinds of fields. We encourage the use of [roles] over groups whenever possible for reasons of maintenance and control, but in terms of performance, these are all equivalent.

When a user opens a view that contains these reader access controlled documents, the server has to determine whether or not this user is allowed to see each document. The server is very methodical, starting at the top and checking each document. The process is similar to the following:
Lotus Domino examines document #1, determines that the user can see it, and then displays it.
Lotus Domino examines document #2, determines that the user can see it, and then displays it.
Lotus Domino examines document #3, determines that the user cannot see it, and then hides the document from the user.

The process continues until Lotus Domino examines all documents in the view. Of course, the view has to be refreshed first, otherwise the server wouldn’t know which document was #1 or what its Reader Names values were. Let’s pause in this explanation to consider a classic example of poor performance. Imagine for a moment that our database is a Sales Tracking application, and it has a By Revenue view that is sorted in descending order by size of contract. Imagine that the view has five columns with the first two columns being sorted (for example, Revenue and SalesRep). There are no categories. In this case, when a user opens the view (a view that potentially displays 400,000 documents), this user forces a refresh first, if needed, and then the server starts with document #1 and checks to see whether or not that document can be displayed to the user, then it checks documents #2, #3, and so on.

You are presumably reading this article via a Web browser, and you’re perhaps familiar with the fact that browsers render portions of a page quickly (text) and other portions (such as graphics) more slowly. But a Domino server does not serve up views this way. It doesn’t send any portion of the view data to the user until it has what it considers a full screen of information. For practical purposes, this may be, for instance, 60 rows of data. You can test this yourself with a Notes client by opening a flat view (no categorized columns) and pressing the down arrow key. At first, the view scrolls quickly. Then there is a pause. This is when the server is dishing up another few KB of data for you. Then the view scrolls quickly again for another 60 or so rows. This repeats as long as you hold down the arrow key.

Back to our user: It turns out that he can see document #1, but perhaps he cannot see the next 10,000 documents. Why? Maybe he can see only 40 documents in the whole database. Because the Domino server is not going to show him anything until it can build about 60 rows of information (or until it gets all the way to the bottom of the view), it’s going to be a long wait.

This is exactly what users experience. They wait for minutes sometimes, while the server checks each document. In the case of a user who can see so few documents that they don’t fill the screen, the server goes through the entire view before sending the user the partial screen of data that he is allowed to see.

Some readers are aware of this functionality. Some are now screaming in pain. You can imagine that as someone who routinely analyzes applications for performance problems that this kind of problem would be one that we spot quickly and look to resolve with workarounds.

One workaround is to make the view categorized. The reason this helps performance is that the categories will always be displayed, and they take up a row of data. So in our example of a view By Revenue, the user would quickly see dozens of categories displaying, say, Revenue and SalesRep. Clicking the twistie to open any category results in a quick round trip to the server to
determine that the underlying document is not viewable for this user, but that’s a far cry from having to examine 400,000 documents in one go.

On the other hand, finding that virtually every twistie the user clicks expands, but doesn’t show any documents, may be frustrating. In our example, there are some other possible workarounds, but we would assert that a view showing all documents by revenue is probably not a view that users with access to only 0.01 percent of the database should have access to. It is a contradiction of their access. But there are many cases in which it makes business sense to have many users accessing a view that contains more data than they should see. At the bottom of this section, we list some tips for building fast views in databases with reader access controlled documents.

We have two closing thoughts about Reader Names and views. First, in a normal production environment, it’s rare to get such clear data as we have in Figure 3. You’re much more likely to see that the performance problems caused by Reader Names cause a host of other performance delays, such as agents, view indexing, and so on. These problems may cause other problems. For example, you may notice that the views are slow for everyone and deduce that the problem is view indexing -- when the underlying problem is that Reader Names are forcing long computations by the server when some users open views.

Second, our tests (and Figure 3) represent an absolutely best-case scenario. The reason is that we had only one user at a time. We could duplicate the volume of data and even indexing issues with our scheduled agents, but our tests could not show the spiraling performance problem that occurs when hundreds of users simultaneously try to access views with reader access controlled documents. If your application uses Reader Names, you absolutely have to pay attention to view performance if you want to avoid a crisis.

Performance enhancing tips with reader access controlled documents

The following are some tips for making applications/views that perform well even with reader access controlled documents:

- **Embedded view using Show Single Category.** This is the winner, hands down. If your data is structured so that users can see all the documents in a category, then you can display just the contents of that category very quickly to the user. In some cases, it may make sense to let the user switch categories, in which case you have to consider whether or not he can see the contents of the other categories. But in most cases, the view would be something like My Sales and would show all the sales documents for the current user. The caveat for this kind of view is that the user interface for the Notes client is not quite as nice as the native view display. For Web browsers, it is just as good, and we have never seen a reason not to use this kind of view for Web browser applications. In fact, the performance is so good that it’s faster to open one of these with reader access controlled documents than to open a native view without reader access controlled documents!

- **Local Replicas.** For a sales tracking database, many companies use local replication to ensure that their sales reps can use the database when disconnected. This is a great solution in many ways, but reader access controlled documents can be tricky when their Reader
Names values change, and they need to disappear from some local replicas and appear on others.

- **Shared, Private on first use views.** This is an elegant solution for Notes clients, but there are some drawbacks. First, it cannot be used for browsers. Second, the views need either to be stored locally, which can be problematic for some customers, or on the server, which can be a performance and maintenance problem of its own. Third, as the design of the application changes, it can be tricky updating the design of these (now) private views. And fourth, some customers have experienced performance problems, which may be related to having large numbers of these private views being created and used.

- **Categorized views.** As seen in Figure 3, categorized views can be very fast to open with respect to Reader Names. They are bigger and slower to index, but typically they eliminate the Reader Names performance issue. The real caveat here is that users may find these views to be unfriendly, a label no one wants to have on their application.

The final tip concerns something to avoid. The feature "Don’t show empty categories" could, in theory, be used very successfully with the preceding tip to make a categorized view that would only display the categories containing documents that a user can see. However, in practice, it will result in a view with performance characteristics akin to a flat view, so it is probably a feature to avoid if performance is important.

**General view performance tips**

Here are some tips about view performance, regardless of whether or not reader access controlled documents are present.

**Time/date formulas**

Using @Now or @Today in a column or selection formula forces a view rebuild every time the view is refreshed. Therefore, if you use this kind of formula, consider making the view refresh Manually or "Auto, at most every n hours." That way, when users open the view, they will not have to wait for a view rebuild (or rarely so, in the latter case). The downside to this is that the view may be out-of-date because it opened without trying to refresh. Consider the contents of these views and how fast they change to determine whether or not you can use these indexing options safely.

**Use click-sort judiciously**

Click-sort is a brilliant feature, one that we use in most public views in our applications. But it’s worth checking your hidden lookup views to be sure that no columns have click-sort enabled because having it increases the disk space and indexing requirements without adding any functionality for your users. On this same topic, consider using ascending or descending, but not both, when you use this feature to save on disk space and indexing time without impairing functionality. We would argue that the functionality is preferable when there is only one arrow because it is an on-off toggle rather than a three-way toggle.

**Generate unique keys in index**

One of the lesser known performance enhancing features, the "Generate unique keys in index" option for lookup views can dramatically improve lookup times. When selected, the view...
discards all duplicate entries. For example, if you have 400,000 documents all displaying only CompanyName, and there are only 1,000 unique company names, then selecting this feature results in only the first document for each company being held in the view index. Although there is a slight overhead to this feature, it is easily outweighed by the fact that your view may dramatically decrease in size. One warning is that if you are displaying the results of a multi-value field, you need to use a code trick to avoid losing data.

For example, imagine a database has 100,000 documents, all containing a multi-value field called Industry. You want to look up a unique list of industries already selected so that the Industry drop-down list can display the values when users create new documents. If you use this unique keys feature in a view that has a single column displaying Industry, it is possible that a new document containing the values Automotive and Air would not display because another document containing Automotive was already being displayed in the view. Thus, your new value of Air would never be picked up by the lookup formula.

To avoid this problem, use a formula such as @Implode(Industry; "~") in the column formula, and then the drop-down field lookup formula may be @Explode(@DbColumn("Notes"; ""; "(LookupViewName)"; 1); "~"). Although you will have some minor duplication of data in your view, you will be guaranteed no data loss.

Color profiles

If you use color profiles (as in the mail template), then any change to that color profile document necessitates a view rebuild for any views that reference that color profile.

Default view

Whenever your application is opened, your default view is opened. Always be sure that this is a view that opens quickly. An interesting example of this is your mail file. For many users, the view that opens by default is the Inbox (folder). If a user is not in the habit of cleaning out her Inbox and if it has many thousands of documents, then it will be slower to open her mail file than if she regularly cleaned out her Inbox and if the view/folder had only dozens or hundreds of documents. Incidentally, this is why some companies have a policy that old documents in the Inbox are deleted periodically. It forces users to move those documents into other folders, thus improving performance both for that user and the server.

Conclusion

The conclusion we hope you draw from this article series is that application performance is something you can influence. With thoughtful application of common sense (and perhaps some of the information and tips in this article), you can keep the performance of your applications at a level that is more than acceptable for your users.
Related topics

- Read part 1 of this article series, "Lotus Notes/Domino 7 application performance: Part 1: Database properties and document collections."
- In addition, the two-part article series, "Application Performance Tuning, Part 1" and "Application Performance Tuning, Part 2," offers valuable information about tuning application performance in Lotus Notes/Domino 5 and 6.
- The article, "New features in Lotus Notes and Domino Designer 7.0," describes new features introduced in Lotus Domino Designer 7.0.

© Copyright IBM Corporation 2006
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)