Lotus. Developer Domain
The Technical Resource for Lotus software
LDD Today

Level: Intermediate
Works with: Notes/Domino
Updated: 01-May-2003

by Tara Hall
with Raphael Savir

In **part one** of this two part series about application performance tuning, we looked at database, view, and form properties that can affect how well your applications perform. By now, you may have enabled or disabled some properties and seen some improvement. But there's more that you can do to improve application performance. In part two of this series, we look at coding practices that can performance. We also examine some common LotusScript methods to see which ones perform best under different conditions. This article assumes that you are an experienced Notes/Domino application developer familiar with LotusScript.

Coding front to back

When it comes to coding, let's make some simple distinctions between what we refer to as front-end code and back-end code. Front-end code is code that is executed from an application's user interface (UI), for instance, manually running an agent from the Actions menu. Problems with this type of code are easy to diagnose because you can readily identify through manual testing when your code is running well and when it's running poorly. Performance problems with front-end code are often associated with a field, button, agent—some UI element that you can change. As suggested in the first article, a good test environment can help you to isolate the problem.

Back-end code is code executed behind the scenes. This includes scheduled agents, which can sometimes be a source of systemic performance-related problems. For instance, you may have a scheduled agent that removes save/replication conflicts from a database, and you may find that the agent is completing its task in an unusual amount of time or that it is consuming an inordinate amount of system resources to complete the task. To test back-end code, review your logs for aberrations, such as long time lapses to complete a task. When the back-end code that you are testing is an agent, check the agent log to find out how well the agent is executing its tasks.

Now that we've differentiated front-end code from back-end code, let's look at some common coding mistakes that affect both.

Temporary variables

One of the most common mistakes that developers make when it comes to coding is failing to use temporary variables as placeholders for data that is expensive to retrieve. The most obvious example is code that relies upon the results of an @DbLookup formula. If your code references data that you lookup more than once, set the data to a temporary variable. Now you can reference it as many times as needed, including:

- When checking for error conditions
- When parsing the data into smaller units (for example, a multi-value list)
- When sorting the data

Another classic example is having a document with a lot of data and users who can sort the data in different ways by clicking a button. This functionality may be intended to mimic the sort-on-the-fly view functionality, but within a large table in a document. This is definitely an opportune situation to set your large arrays of information to temporary variables, and then to sort the temporary variables. The difference in performance can be startling. In one embarrassing moment in the author's past, code that took over a minute to execute was thankfully reduced to sub-second performance by the use of temporary variables.

Finally, a third example is having code that searches for a view of a specific name in a database. You may be tempted to loop through the db.views property, but you can save time by first setting a temporary variable, like viewLIST = db.views. Then you can iterate through the temporary variable with excellent performance.

Computed fields

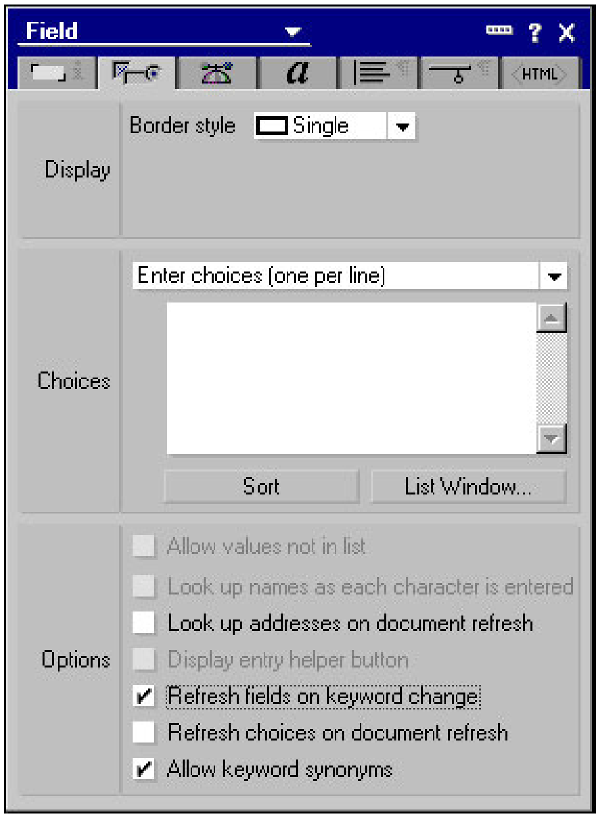
Limiting the frequency of computations in a document can improve performance. To state the obvious, the more computations executed in a document, the slower the performance. Whenever you open a document in read mode, certain computations occur. When you open a document in edit mode or switch from read to edit mode, other computations occur. You should have a sense of what percentage of the time your documents are opened in read versus edit mode so that you know what fields/computations to reduce. Here are some examples for each case:

- *If the documents are usually read, not edited*
  In this case, @DbLookup and @DbColumn formulas are triggered, so code fields containing these formulas to resist execution in read mode. For example, you can bracket the formula in an if statement that checks for @IsDocBeingEdited. Keyword fields are prime candidates for this code because they often contain @DbLookup or @DbColumn formulas. For example, a keyword field called kList may then have the following formula:

  ```
  @If(@IsDocBeingEdited; @DbColumn("Notes"; ""; ViewName; 1); kList)
  ```

  Note that in the Else condition we leave the contents of the field as is so that in read mode, whatever value has already been selected is displayed, but no further computation takes place. Alternatively, you could add the code to a button that users click to execute the code. Note that in read mode, Computed for Display fields compute, so be sure that those don't have expensive formulas.

- *If the documents are often read, and then switched to edit mode*
  In this case, make sure that any code you have prevented from executing (as described earlier) is executed upon switching to edit mode. For example, keyword fields that have @DbLookup or @DbColumn formulas are prime candidates for suppression when a user opens the document in read mode. But if the user switches the document to edit mode, use a PostModeChange event that forces a document refresh (for example, if source.editmode then call source.refresh). In addition, select the keyword field option "Refresh choices on document refresh." With that option selected, when a user switches from read mode to edit mode, the document refreshes once automatically and forces the keyword fields to re-evaluate.



Display
Border style: Single

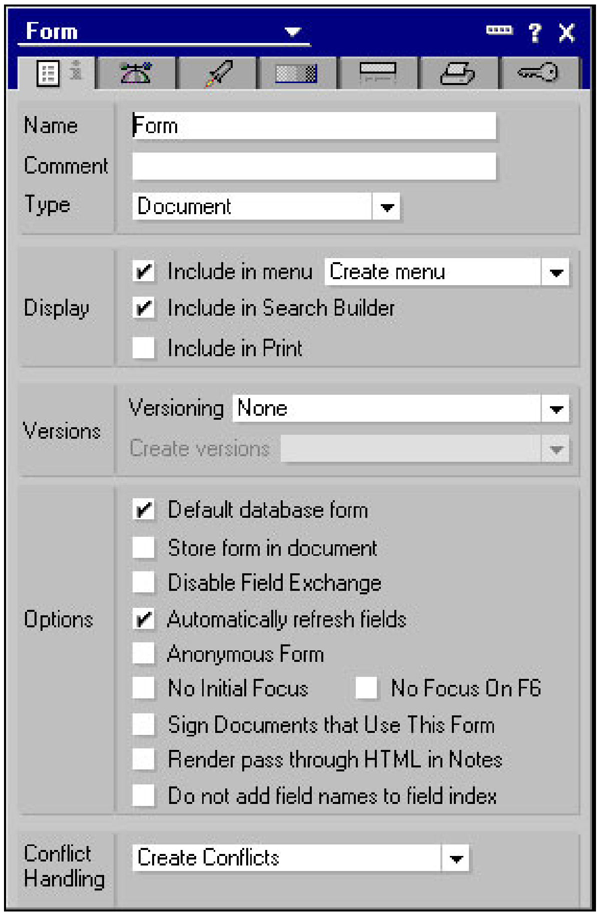
Choices
Enter choices (one per line)
Sort
List Window...

Options
- [ ] Allow values not in list
- [ ] Look up names as each character is entered
- [ ] Look up addresses on document refresh
- [ ] Display entry helper button
- [x] Refresh fields on keyword change
- [ ] Refresh choices on document refresh
- [x] Allow keyword synonyms

- *If the documents are often opened in edit mode*
  In this case, you may want to move as much of your expensive (in terms of performance) code into buttons so that the frequent edits are not bogged down. This assumes that even when editing the document, most users don't need to change all the keyword fields, for example.

You may think that the suggestions above require more computations, not fewer computations. In one sense, that is correct. Only take these steps to avoid expensive computations such as @Db formulas and don't bracket a simple @ProperCase, for example, with an @If(@IsDocBeingEdited). It's not worthwhile.

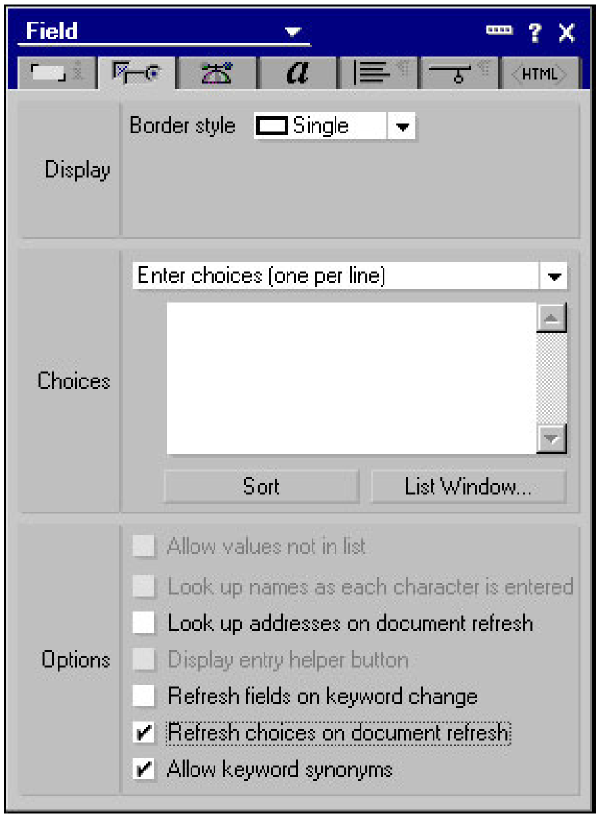
Refreshing field values
You can set field values to refresh automatically by selecting the Automatically refresh field option on the Form Info tab of the Form Properties box.



Name	Form
Comment	
Type	Document
Display	✓ Include in menu: Create menu
	✓ Include in Search Builder
	☐ Include in Print
Versions	Versioning: None
	Create versions
Options	✓ Default database form
	☐ Store form in document
	☐ Disable Field Exchange
	✓ Automatically refresh fields
	☐ Anonymous Form
	☐ No Initial Focus ☐ No Focus On F6
	☐ Sign Documents that Use This Form
	☐ Render pass through HTML in Notes
	☐ Do not add field names to field index
Conflict Handling	Create Conflicts

Doing so adversely affects your performance because every time a user moves his cursor through the form fields all the previous fields are recomputed. The purpose of this heavy computation is primarily to check for Input Translation and Input Validation formulas, but, in fact, all code is executed.

If you need to refresh the keyword lists in your computed fields when a user selects a specific value, select the "Refresh fields on keyword change" option on the Control tab of the Field Properties box that we mentioned earlier. For example, suppose you have multiple keyword fields on a form and the values of keyword fields two, three, and four differ depending on the value that the user selects in keyword field one. In that case, use the "Refresh fields on keyword change" option. This is just like pressing the F9 key, but it's done automatically every time the value in this keyword field is changed. This offers better performance over the "Automatically refresh fields" option on the Form Properties box because the document is only refreshed when the value changes in this one field, instead of refreshing every time any value is changed. Note that you must set the field option "Refresh choices on document refresh" for the other keyword fields. Any keyword fields that do not need to participate in this more dynamic relationship do not need that option set, so they do not refresh when the first keyword has its value changed.



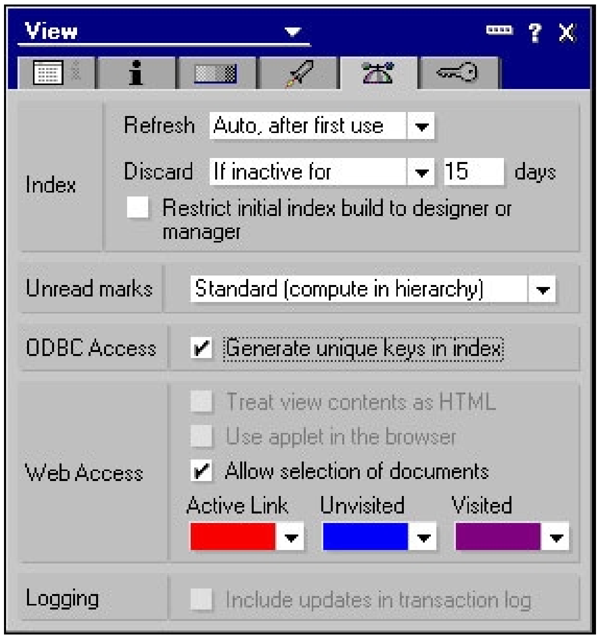
- [ ] Allow values not in list
- [ ] Look up names as each character is entered
- [ ] Look up addresses on document refresh
- [ ] Display entry helper button
- [ ] Refresh fields on keyword change
- [x] Refresh choices on document refresh
- [x] Allow keyword synonyms

Using the Computed when composed field type
The Computed when composed field type calculates the values for a field when a user creates a document. You can use a Computed when composed field to inherit values, or if a field is going to be set by some other code, but you want it to remain inert. For example, a field named OriginalSubject in a response document contains the formula Subject. When a user creates a response document, this field inherits from the main document selected and never again computes. Another example is a field called DateClosed that is set by code behind an action bar button. Because we never want this field to change its own value, we set it to be Computed when composed and use the formula DateClosed. This lets it act as a placeholder formula. It only tries to compute when first created (and in this case, let us suppose that there is no inheritance at work) and thereafter only takes whatever value is forced into it. Note that Computed when composed fields apply the correct data type to values pushed into them as long as the user saves the document.

You may ask what the difference is between using a Computed when composed field and a Computed field in these two examples. The primary difference is simply that a Computed field computes every time the document is edited, refreshed, and saved, even if it has no real work to perform. If there are just a handful of easy formulas as above, then this field type makes little difference to your application's performance. However, if there are many such fields in your forms, you will surely see a difference in performance. Because there is no benefit to making the field Computed, in this example, your users may as well get better performance. Truly a case of getting something for nothing!

Cache and nocache parameters
The cache and nocache parameters apply to all of the @Db formulas. When you specify the cache parameter, values for the formulas are stored in a cache for easy retrieval. When you specify the nocache parameter, values for the formulas are not stored in a cache, so each lookup comes from the database. Too often, developers overuse the nocache parameter and more resources are used to retrieve data from the database rather than the cache. Don't make the mistake of determining the use of the nocache parameter on how important you think the data is.

Instead, consider how often the data changes: The more frequently the data changes, the more you may want to use the nocache parameter. For data that changes infrequently, use the cache parameter. For example, suppose you have a discussion database in which users can specify keywords that create new categories. (That is, each time a user creates a new topic, he or she can specify any category for that topic.) In this case, the data may be relatively unimportant, but you nevertheless need to use the nocache parameter so that a user who enters two or three main topics in a row sees his new categories reflected immediately in the keyword field of the next main topic. To prevent poor performance, use the nocache parameter with the "Generate unique keys in index" option for ODBC Access, which is discussed in part one of this article series. Remember the "Generate unique keys in index" option maintains a smaller view index by listing only unique categories.



Index
Refresh: Auto, after first use
Discard: If inactive for 15 days
☐ Restrict initial index build to designer or manager

Unread marks: Standard (compute in hierarchy)

ODBC Access: ✓ Generate unique keys in index

Web Access
☐ Treat view contents as HTML
☐ Use applet in the browser
✓ Allow selection of documents
Active Link: 🟥 Unvisited: 🟦 Visited: 🟪

Logging: ☐ Include updates in transaction log

For a counter example, suppose you have a lookup to salary information. This is vitally important information, but typically salary changes occur at most once every few months in the best of times, making this data a good candidate for caching.

LotusScript methods

Lotus conducted tests to determine which commonly used LotusScript methods performed best in terms of getting a collection of documents—the most frequently performed task in virtually any piece of LotusScript code. In this section, we compare the following commonly used LotusScript methods:

- db.FTSearch
- db.Search
- view.GetAllDocumentsByKey
- view.GetDocumentByKey

In the tests, databases of differing sizes (10,000, 100,000, and 1,000,000 documents) were used to see how well each method performed.

db.FTSearch method
db.FTSearch returns a collection of documents based upon a full text search of a database. It performs well, but requires a current full text index and perhaps a steeper learning curve for syntax mastery. In addition, depending upon server Notes.ini settings, there may be limits placed upon the size of the returned collection. Of course, if your search is based upon the contents of a rich text field, then this is your only viable option!

db.Search method
db.Search returns a collection of documents based upon a database search using what is essentially a view selection formula. This is a relatively poor performer for small collections in large databases. For example, if you have 100,000 documents in your database and you only need to find five or ten documents, you may want to avoid using db.Search. On the other hand, it requires no full text index, and no pre-built views, so it can be a very handy search method. If, for example, you are searching against a database over which you have little control, this may be your only reliable choice.

view.GetAllDocumentsByKey method
Since Release 5, this method has been the fastest way to retrieve a collection of documents. The only downside is the need to have already built the relevant views. However, as long as you streamline your view design and avoid the use of expensive time/date sensitive formulas (as discussed in the **previous article** in this series), the impact of these views on performance and disk space should be minimal, and the performance of code utilizing view.GetAllDocumentsByKey to get collections of documents from these views will be very fast.

In general, when iterating through the collection of documents retrieved using any of these methods, your code should use

```
set doc = DocumentCollection.GetNextDocument ( doc )
```

instead of

```
set doc = DocumentCollection.GetNthDocument ( i )
```

where i increments from one to DocumentCollection.count. For small collections—and for code run in isolation, like a scheduled agent—the performance degradation is minimal, but for large collections—or for code run by many users simultaneously—there is a performance cost which makes GetNth an unwise choice. The GetNth method is typically reserved for cases in which you want to pick and choose documents out of the collection, not for simply iterating through the whole collection.

view.GetDocumentByKey method
This is the only method that does not get a collection of documents in memory. Instead, view.GetDocumentByKey uses an already built view index as its collection and gets one document at a time in the view. Used in conjunction with view.AutoUpdate = False, this method is very fast and doesn't require the memory to hold potentially large collections of documents.

Note: view.AutoUpdate = False is used primarily to avoid error messages when getting a handle to the next document in a view if the previous document has been removed from the view, but it also improves performance substantially for the agent running. When changing data in documents, you may see significant improvement in your views with view.AutoUpdate = False.

Events, shared elements, and more

Here are a few additional programming tips to keep in mind:

- *Pay attention to the number of events in a form and don't "overcode."*
  When removing code, be careful to completely remove it. Don't just remark it out, or partially erase the code. You can tell whether an event thinks it has code by whether the circle/squiggle is filled in or is empty.

- *Shared elements are slightly worse performers; however, they compensate for this poor performance by being used in multiple places.*
  Consider carefully when to use shared elements to save yourself some work and when to repeat an element to increase performance.

- *If you implement error checking, make sure that the checking stops when it encounters an error.*
  In case of careful programming, this ensures that your code does not "leak" by continuing to execute when it logically should end.

- *Large subforms are poor performers.*
  A large subform can affect application performance. If you do not use a large subform many times in an application, consider repeating the fields of the subform in each form rather than using the subform.

- *Use fewer fields.*
  Having fewer fields in a document is relevant to performance, more so than the size of the document. Having fewer fields with more data, such as multi-value fields, rather than more fields with less data, improves application performance. For more traditional programmers new to Notes/Domino application development, this may be counter-intuitive, but testing validates this concept clearly.

- *Use view.Autoupdate=False to prevent a view from refreshing.*
  As described earlier, using the view.GetDocumentByKey method in conjunction with this property can be an excellent performer.

- *Use the StampAll method to modify a large collection of documents at once.*
  This method works best when you need to stamp a large document collection with a static value, such as the current time/date or a flag set to a value.

- *The ForAll statement is the fastest way to iterate through a loop.*

- *Fixed arrays are better performers than dynamic arrays.*
  Dynamic arrays are slightly worse performers than fixed arrays, but dynamic arrays are sized appropriately, so weigh the two considerations before you decide on fixed or dynamic arrays.

Conclusion

We hope that the tips in this article series have been helpful to you and that you soon see improvement in your application performance as a result of implementing these practices. We want to hear about your best practices for application performance tuning, so if you have any tips that you want to share with the greater Notes/Domino application developer community, **submit your tips to us**.