12

Thread architecture

This chapter covers

- Descriptions of ManifoldCF's Pull Agent threads
- How Pull Agent threads interact with repository connectors
- Transitions between job and document states, and under what conditions they occur
- Descriptions of ManifoldCF's Authority Service threads
- How Authority Service threads interact with authority connectors

This chapter, combined with Chapters 10 and 11, is mainly a description of how ManifoldCF actually works. Thus, what you get out of it as a reader will vary depending on what your needs are. Those who are looking only to integrate with the project, or add their own connectors, may get only limited additional help here towards their goals, although you will certainly be more effective at diagnosing any problems with any connectors you write if you have a strong notion of how ManifoldCF operates under the covers. On the other hand, if you are seriously considering a deeper involvement in ManifoldCF, you will certainly want to keep reading. You may even learn enough to become a contributor to the ManifoldCF project someday.

In this chapter, I'll explore the thread structure of ManifoldCF. I'll start with a presentation of how ManifoldCF threads work, and proceed to list and describe these threads, and the responsibilities of each one with respect to the transitions between job and document states.

12.1 ManifoldCF thread structure

Let's begin with an understanding of the various threads that perform tasks within the ManifoldCF Pull Agent and Authority Service. Together these constitute the *thread architecture* of the crawler. As we will learn, there is a reason for each of these threads to exist, and that reason has a lot to do with job states, document states, and the transitions between them.

12.1.1 Crawling stages and repository connectors

Overall, the ManifoldCF Pull Agent is built as a state machine. We've seen in Chapter 11 how the database is used to keep track of both a job's state and a document's state. Threads within the Pull Agent are individually responsible for transitioning jobs and documents to new states, and performing the appropriate crawling activities while they are at it.

ManifoldCF wound up with this architecture largely as a result of the need for it to be able to be shut down and restarted without losing track of what it was doing. This implies that all the individual threads base their activities mainly on the state that is stored in the database. As long as the database is managed in a careful manner, ManifoldCF is then able to claim the resilience it was designed for. Thus, the need for resilience leads very naturally to a state-based architecture. It is pretty reasonable to describe how ManifoldCF works by simply describing the states of jobs and documents, and the threads that take care of the transitions.

Before I talk about the responsibilities of individual Pull Agent persistent threads, we'll need to review the crawling process, especially the functionality inside of a repository connector that is used for crawling. We can look at a repository connector as implementing three pieces of crawling functionality. I'll simplify the discussion and use the following methods as exemplars: `addSeedDocuments()`, `getDocumentVersions()`, and `processDocuments()`. You should be familiar with the actual methods and their usage by now; we discussed these in considerable detail in Chapter 7.

The methods represented by `addSeedDocuments()` are responsible for providing starting documents for a job. For some kinds of repository, this may be the only way that documents are discovered; for others, a single document is all that is ever added. As you might expect, this method and its brethren are called only during the *seeding* phase of crawling.

The methods represented by `getDocumentVersions()` are responsible for obtaining a version description for a document or a list of documents. These methods directly support incremental crawling, but cannot cause documents to be indexed or new documents to be discovered. Conceptually, these methods belong to the *processing and discovery* phase of crawling. I will call the activity of obtaining document versions *versioning*.

The final set of methods are represented by `processDocuments()`. These methods are also called during the processing and discovery phase of crawling, and are responsible for indexing documents and discovering new documents (those that are referenced by the ones being processed).

Figure 12.1 pictorially represents the phases of crawling, and the repository connector method groups that are called in each. I've adapted this figure from figure 1.3, and flagged where the appropriate exemplar methods are used. I've also broken down the crawling phase of the cycle to separate out seeding from document discovery and processing.

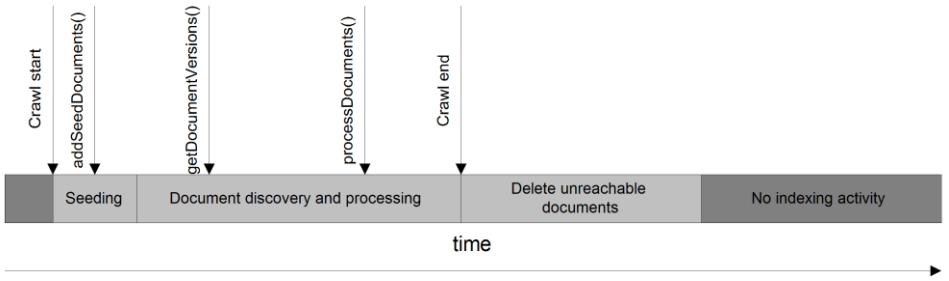

Figure 12.1 The full incremental crawling cycle, including seeding, discovery, and processing. This is not a continuous crawl.

A related diagram can be produced for continuous crawling. See figure 12.2.

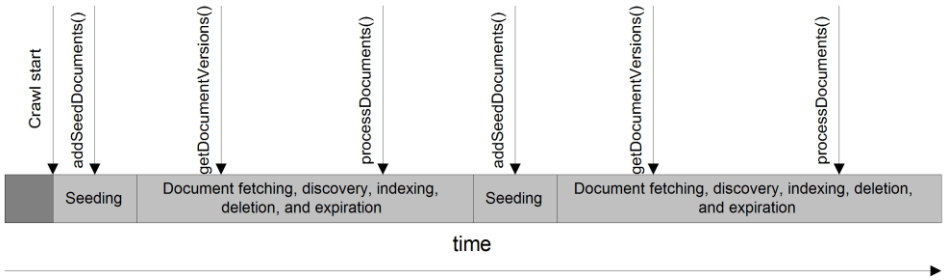

Figure 12.2 Continuous crawling, and the repository connector exemplar methods that are used for each stage. The seeding phases operate at the same time as fetching, discovery, etc. in this model.

During the design of ManifoldCF, it quickly became clear that there were not enough significant differences between these two patterns to justify the creation of a whole second set of states in order to deal with both models. Instead, a job's state determines the behavior, and the Pull Agent's persistent threads need to deal with both crawling models equally well. In rare cases, a thread exists solely to support one particular model, and is not needed at all for the other. In those cases, the differences in crawling model are represented by the creation of additional job and document states. For example, for nearly every ACTIVE_xxx job state, there is a corresponding ACTIVE_SEEDING_xxx variant of that

state, which captures the continuous crawl situation where seeding is occurring at the same time as crawling. We will be discussing exactly how that occurs later in this section.

12.1.2 Job states and transitions

Figure 12.3 is a diagram relating job states for a single job, and the conditions and transitions between them. Each transition is labeled with the conditions under which the transition takes place. I would have liked to include the persistent thread or threads responsible for making that transition, but space did not permit it. I will summarize the missing information in a subsequent table.

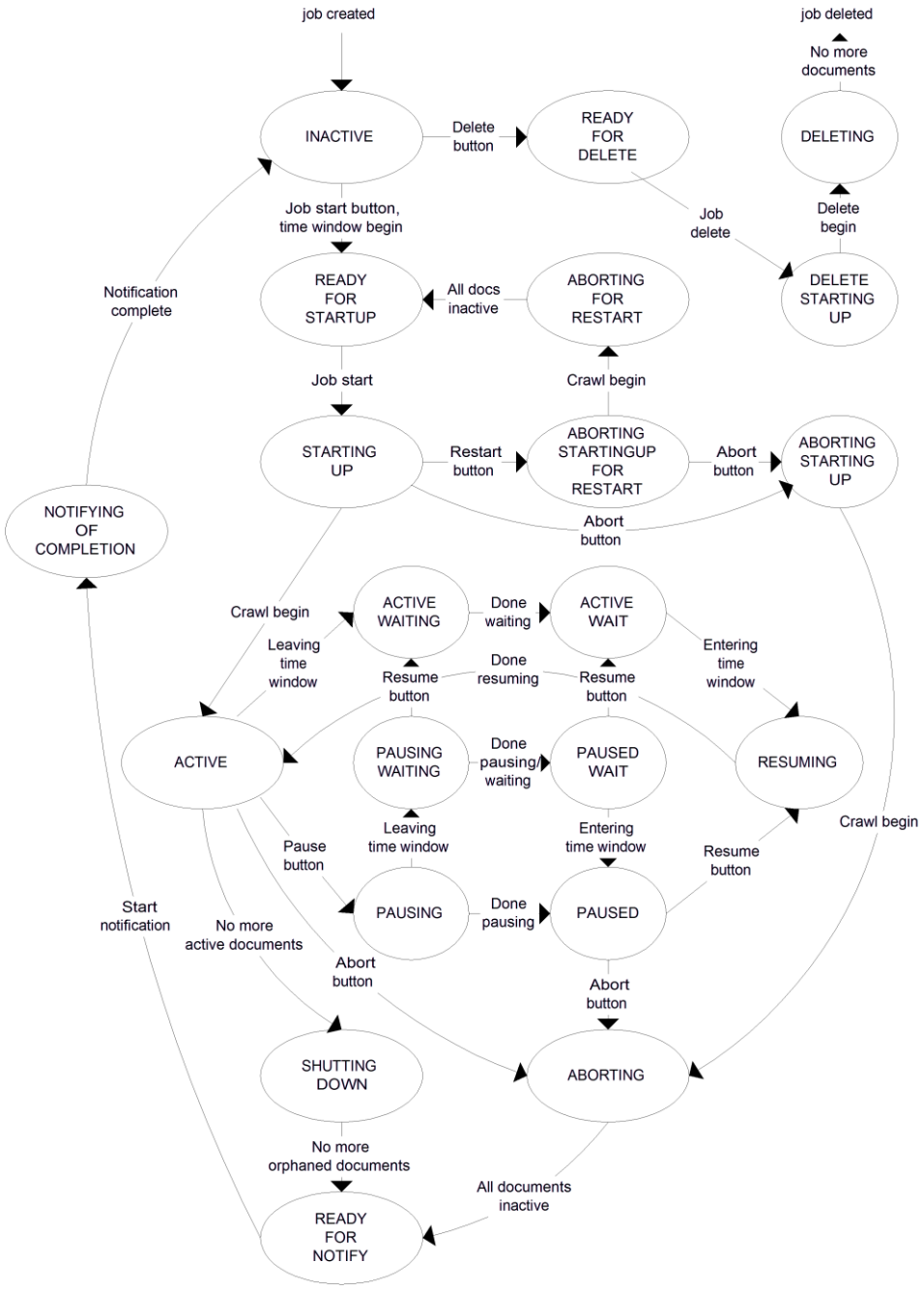

Figure 12.3 A diagram showing the principle job states and the transitions between them. This diagram is incomplete, as is explained in the text.

As a general rule, ManifoldCF is architected so that each state transition of this kind is made the unambiguous responsibility of a single thread or thread family, or are directly performed by an API or crawler UI operation. Figure 12.3 demonstrates this relationship to some extent. The same rule also happens to be true of the transitions between document states, which we'll get to in a moment. But why should it be done this way? For instance, why not just have one thread which performs most of the transitions?

The main reason for the multiple transition thread approach is, simply, performance. ManifoldCF does not want to stand in the way of a crawl executing as quickly as possible. Modern servers have many CPUs or cores, and ManifoldCF must therefore be structured in such a way as to be able to take advantage of all that parallelism to the greatest extent possible.

> **Remember** Each state transition does not merely represent a change of state for its own sake, but also represents actual crawling work being completed.

It is also true that it is easier to check for the correctness of a thread that performs a single task and transition, than it is to check for the correctness of a thread that performs multiple tasks. Thus, even if parallelism was not an issue, it would still be (in my opinion) a better design to dedicate each thread to a relatively simple task.

Let us now try to understand the states and tasks themselves. An astute reader may notice that many of the job states described in Chapter 11 are not present in figure 12.3. The reason is that the complexity of the diagram became too great for me to include them. To complete the picture, I've included what's missing from figure 12.3 in another diagram, figure 12.4.

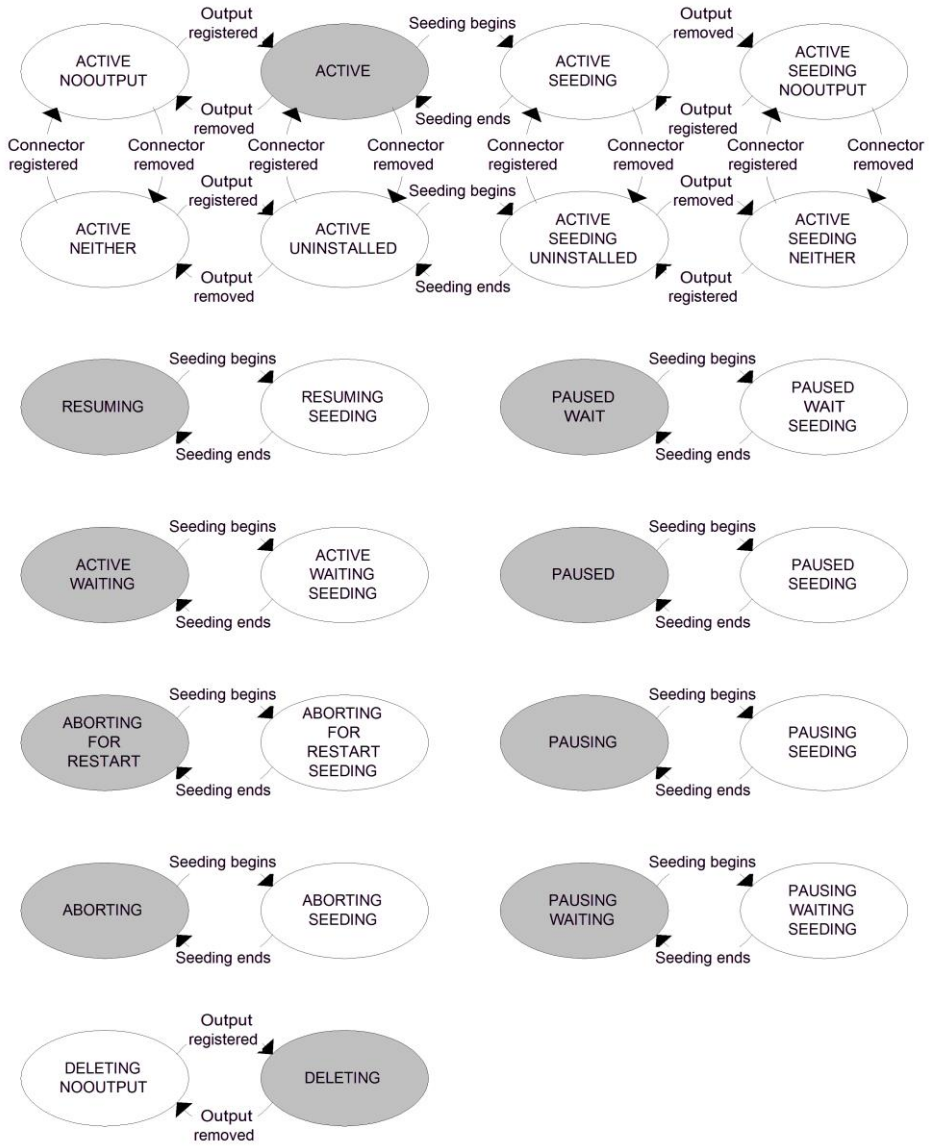

Figure 12.4 The job states and transitions that would not fit in figure 12.3. The grayed states are present in figure 12.3; the transitions to and from all related states are similar, with the exception that the condition that the state represents (e.g. ACTIVE_NOOUTPUT indicating no registered output connector) will be checked by any entering transition, and the proper target state chosen.

Clearly there's a great deal of complexity to understand here. It makes it easier to think about these related states as being a single, basic state that has been "split" in one or more dimensions. For example, a basic state (say ACTIVE), has many related states (ACTIVE_NOOUTPUT, ACTIVE_NEITHER, ACTIVE_UNINSTALLED, ACTIVESEEDING, etc.) which are similar except that they differ in a number of crucial dimensions, i.e. whether the output connector is currently registered, whether the repository connector is currently registered, and whether seeding is in progress. The transitions drawn give a clue as to which dimension is changing between the states.

Table 12.1 ties all of these transitions into each responsible Pull Agent persistent thread. I have not included user actions in this list.

Table 12.1 The starting and ending states and names of the threads which handle each job-state transition, as labeled in figures 12.3 and 12.4.

Start state family	End state family	Transition	Thread
INACTIVE	READYFORSTARTUP	Time window begin	Job Start Thread
READYFORSTARTUP	STARTINGUP	Job start	Startup Thread
STARTINGUP, etc.	ACTIVE, etc.	Crawl begin	Startup Thread
ACTIVE	SHUTTINGDOWN	No more active documents	Finisher Thread
SHUTTINGDOWN	READYFORNOTIFY	No more orphaned documents	Job Reset Thread
READYFORNOTIFY	NOTIFYINGOFCOMPLETION	Notification starting	Job Notification Thread
NOTIFYINGOFCOMPLETION	INACTIVE	Notification complete	Job Notification Thread
ABORTING	READYFORNOTIFY	All documents inactive	Job Reset Thread
ACTIVEWAIT, etc.	ACTIVE, etc.	Time window begin	Job Start Thread
ACTIVE, etc.	ACTIVEWAIT, etc.	Time window end	Job Start Thread
ACTIVE, etc.	ACTIVESEEDING, etc.	Seeding begins	Seeding Thread
ACTIVESEEDING, etc.	ACTIVE, etc.	Seeding ends	Seeding Thread

READYFORDELETE	DELETESTARTINGUP	Job delete	Start Delete Thread
DELETESTARTINGUP	DELETING, etc.	Delete begin	Start Delete Thread

12.1.3 Document states and transitions

As I've indicated, there's another state diagram that you'll need to examine which describes the states and transitions for an individual document. These states are just as important as job states in the overall function of ManifoldCF. Figure 12.5 captures the possible states for an individual document, along with the threads which perform the state transitions. We will be discussing these threads and transitions in subsequent sections of this chapter.

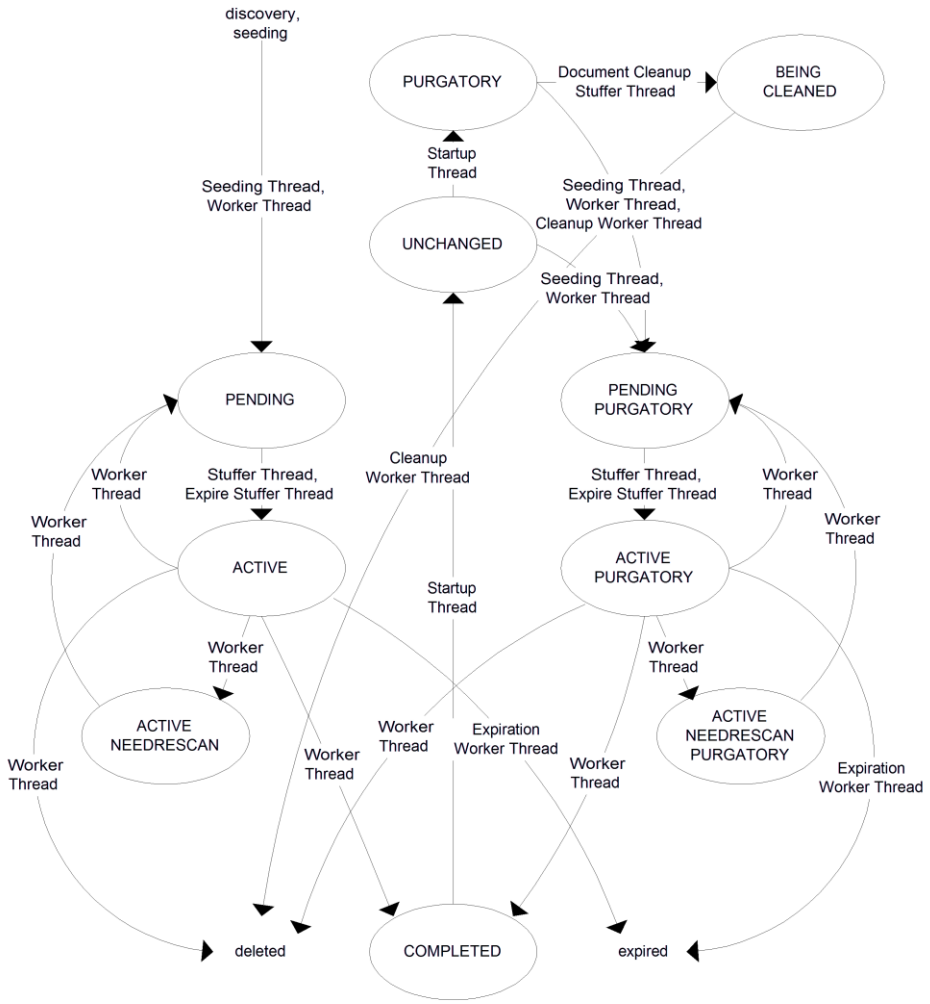

Figure 12.5 Document states and transitions involved in crawling. In this diagram, each transition is labeled with the name of the thread that performs that transition.

Now that we've seen the big picture of how all the job and document states are related, let's go into more depth and describe how each thread and thread family does its job.

12.2 Pull Agent document processing threads

Document processing is the collective name we use to describe the main activity of crawling. It includes fetching documents from the repository, and handing them to the target search engine.

In the Pull Agent, there are two types of threads involved in this activity. One type of thread identifies candidate documents and places them into an in-memory document queue. The second type of thread takes identified documents from the in-memory queue and processes them, either one at a time or in small batches. We'll describe these threads in detail next.

12.2.1 The Document Stuffer Thread

The Document Stuffer Thread performs the critical first step in the sequence of activities that is involved in processing any document. Its sole responsibility is to locate the documents that should be processed next, and to put these documents into an in-memory queue. The documents it identifies as needing to be processed are transitioned to a corresponding state that indicates that they are now the responsibility of a Worker Thread to manage. For example, if a document was in the PENDING state, the Document Stuffer Thread changes the document's state to ACTIVE, and then places the document into the in-memory queue. Later, a Worker Thread will pick up the document from the queue and process it. See figure 12.6.

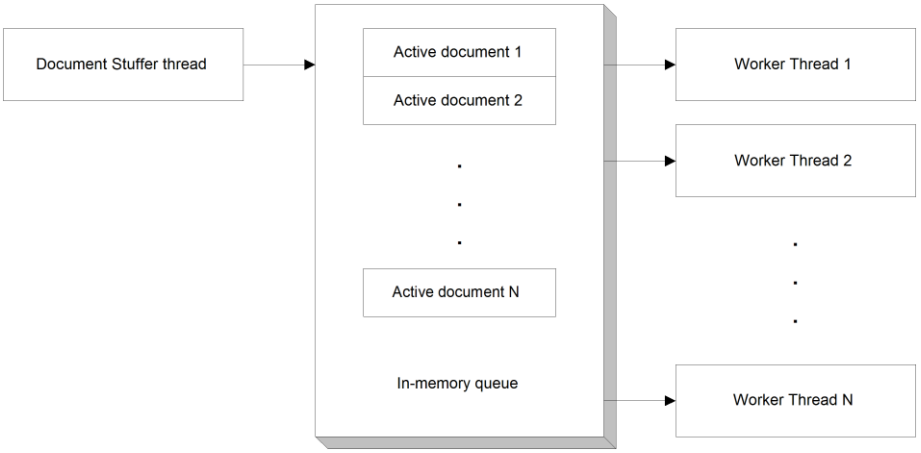

Figure 12.6 The relationship between the in-memory queue, the Document Stuffer thread, and the Worker Threads.

The Document Stuffer Thread cannot "stuff" all of the eligible documents from the `jobqueue` table all at once. If it tried to do that, it would need to read a resultset possibly consisting of millions of rows into memory. So instead, it stuffs some number that is a multiple of the number of Worker Threads at one time, up to some limit (which is also related to the number of Worker Threads). Since it is always dealing with a subset of eligible documents, the order in which the "stuffing" occurs will matter. That is why the Document

Stuffer Thread deals with documents only in an order that is determined by *document priority*.

A document's priority is assigned to it when the document is added to the job queue. It is a simple numeric value, which is based on a function of the document bins the underlying repository connector indicates that it belongs to. The document bins are important because the ideal assignment of a document's priority has a deep underlying relationship with how ManifoldCF and the repository connector perform throttling, and the document's throttling bins are the way in which this is done. All documents which belong to the same throttling bin will throttle each other, while documents belonging to different bins will not. Document priorities are assigned in a way that takes these relationships into account.

In addition, there are a number of conditions that the Document Stuffer Thread obeys to determine the eligibility of a document for "stuffing". For instance, the job that the document belongs to must be in an active state, such as ACTIVE or ACTIVESEEDING. Also, the current time must exceed the minimum time allowed for the document to be fetched (the `jobqueue` table's `checktime` field). But we're still not done – there are other restrictions too. Let's go through the additional restrictions one at a time.

EVENT-GATED RESTRICTIONS

If you recall the discussion in Chapter 11 about events, you may recall that documents can be blocked from being processed if they depend on a specific event, and that event is in play. This restriction is enforced by the Document Stuffer Thread, which temporarily overlooks documents where this restriction applies. The way that these documents are overlooked is by means of the query that the Document Stuffer Thread uses to identify documents to be handed to the in-memory queue.

> **Note** The Document Stuffer Thread restriction is necessary, but is not sufficient, to enforce event gating. Connector code must also play a part, because there is nothing that ManifoldCF can do to stop active document processing within a connector once it has begun. We've already discussed how to code connectors properly to enforce event gating in Chapter 7.

THE SAME DOCUMENT IN A DIFFERENT JOB

Another restriction that must be enforced is when the very same document appears in another job. If that document is either being processed by that job, or is in the process of being deleted by that job, then any processing in the current job must be delayed.

You can see why this is by thinking in terms of what ManifoldCF will do with the document during processing. The document might well be in the process of being deleted in one job and indexed in another, for instance. Since we cannot control the order in which these things happen, it's not predictable what will happen unless the case is considered specially. For this reason, the Document Stuffer Thread makes sure that all the documents it "stuffs" are not active anywhere else at the time they are prepared for processing.

AVERAGE FETCH RATE THROTTLING

One of the many features of the Document Stuffer Thread is the ability of the thread to limit the average fetch rate of documents according to the way the corresponding throttles have been defined. You might think that this job would naturally be a responsibility of Worker Threads, but remember that any time there is a disconnect between the rate that Worker Threads process documents and the Document Stuffer Thread adds them to the in-memory queue, there is a bottleneck. In the case of an average fetch rate throttle being implemented by the Worker Threads, the bottleneck would take the form of Worker Thread starvation, because all the Worker Threads would be occupied with waiting, busily doing nothing. Thus, crawling progress for other documents would be blocked.

On the other hand, if the Document Stuffer Thread largely handles throttling, then no such starvation condition can develop. Thus, this is the approach that ManifoldCF uses. The Document Stuffer Thread calculates the time interval between "stuffing" attempts, and limits the number of documents from each bin that are added to the in-memory queue to be consistent with any throttles that have been specified.

A statistical approach is used to determine the maximum document count limit for each time stuffing takes place, because the time between stuffing attempts is short. That is, the desired number of documents is calculated as a floating point value, and the fractional number of documents is compared against a random number between zero and one to determine whether or not an additional document should be included.

Because throttling occurs per document bin, the maximum document count limit cannot be enforced by the `jobqueue` document stuffing query itself. Instead, it is enforced by result filtering, as the database resultset from the query is processed. This has some interesting repercussions, which we'll discuss next.

DOCUMENT PRIORITIZATION AND BLOCKING DOCUMENTS

As we discussed in Chapter 11, all documents in the `jobqueue` database table have a document priority value, which represents the order in which they are read from the job queue and handed off to Worker Threads to be processed, dependent events, which can block documents from being identified for queuing, and specific earliest processing time stamps, which can also prevent documents from being selected for the in-memory queue. All of these exclusions and restrictions are part of the selection criteria for the query that the Document Stuffer Thread performs, so that documents that do not meet the criteria are excluded from the document selection query's result set.

Fetch-rate throttling is special; it cannot work in the same way as all of the other document restrictions because, as we have discussed, there is no way to apply query constraints based on throttling bins. Documents over the fetch limit are therefore filtered from the selection query's result set afterwards. But that cannot be all there is to it. What happens if, because of fetch rate throttling, a document cannot be processed at that time? It will not have its state changed to ACTIVE, and so will remain eligible for queuing. Thus, it will be eligible to appear in the resultset the next time the stuffing query is executed, and

could well be returned again and again, in many subsequent "stuffing" cycles, even though there is little hope of it being processed right away. If enough such documents were to accumulate, the Document Stuffer Thread would be unable to make any progress whatsoever, even on documents whose bins were not being throttled. Such documents are called *blocking documents*, because they block the proper functioning of the Document Stuffer Thread. ManifoldCF solves the problem by changing the document priority of any documents that are returned by the "stuffing" query but are subsequently skipped due to rate throttling. They are assigned a document priority that places them at the end of the line for their particular document bin. See figure 12.7.

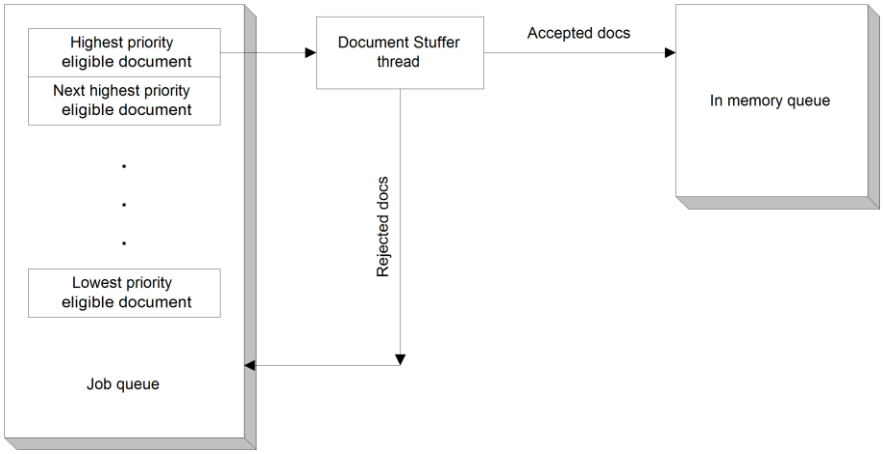

Figure 12.7 Document flow including path for throttling rejection. Documents that are filtered out of the document selection query results are re-prioritized to the bottom of the job queue.

PENDING vs. PENDINGPURGATORY

The documents that the Document Stuffer Thread identifies must either be in the PENDING state or the PENDINGPURGATORY state. The PENDING state is used for a document that has just been seeded or discovered, but has never been successfully processed. The PENDINGPURGATORY state applies to documents which have been processed successfully at least once. The distinction is maintained so that when document cleanup occurs after the end of an incremental crawl, it will be possible to unambiguously identify which documents have been potentially indexed in a previous job run. Please refer to figure 12.2.

When the Document Stuffer Thread selects documents for processing, it therefore must move them to states which preserve the distinction between PURGATORY and non-PURGATORY. Thus, documents in the PENDING state move to the ACTIVE state, and documents in the PENDINGPURGATORY state move to the ACTIVEPURGATORY state.

12.2.2 Worker Threads

I've mentioned Worker Threads several times before in this book. Now it is time to define what they are, and what they do.

Worker Threads implement part of the processing and discovery phase of crawling. There are many of them – in fact, you can control exactly how many by setting the `properties.xml` parameter `org.apache.manifoldcf.crawler.threads`. Each worker thread picks up a set of documents from an in-memory document queue, where sets of documents have been placed by the Document Stuffer Thread. Those documents are known to be one of the several variants of the ACTIVE state, because that's the state that the Document Stuffer Thread sets to remove the documents from consideration for subsequent fetching.

Once it receives a set of documents, a Worker thread uses the appropriate underlying connector's `getDocumentVersions()` and (if appropriate) `processDocuments()` method families to do versioning, indexing, and discovery. Of course, it is possible that either of these processing steps will fail for a specific document. When that happens, it is the Worker Thread's responsibility to restore the document's state to what it was prior to the Document Stuffer thread's changes. If the attempt succeeds, however, the Worker Thread must advance the document state to the appropriate follow-up state.

This all sounds straightforward enough, but there are a number of complicating factors that are potentially involved. We'll discuss these next.

ACTIVE vs. ACTIVEPURGATORY

The document that a Worker Thread is processing started out in one of two states, before the Document Stuffer thread got to it and added it to the in-memory queue. It was either in the PENDING state, or in the PENDINGPURGATORY state.

Both of these states indicate that the document should be processed, provided that the job itself is active. As we discussed earlier, the PENDING state is used for documents which have never before been processed successfully, and the PENDINGPURGATORY state means that the document has indeed been successfully processed at least once. When the Document Stuffer Thread identifies documents for processing and places them in the in-memory document queue, it needs to preserve the distinction between these two starting states. Thus, PENDING documents are thus marked ACTIVE, while PENDINGPURGATORY documents get marked ACTIVEPURGATORY. See figure 12.8.

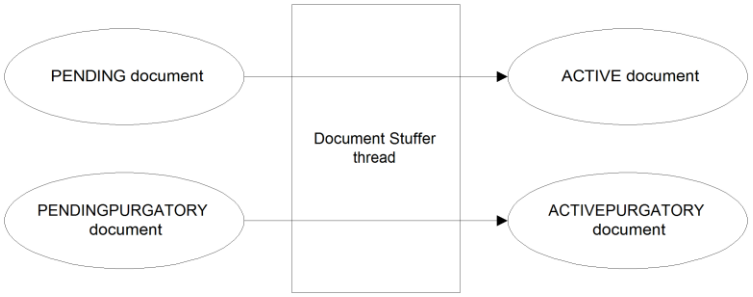

Figure 12.8 Document state transitions performed by the Document Stuffer thread. A separation is maintained between the "PURGATORY" vs. "non-PURGATORY" character of the documents being stuffed.

A Worker Thread therefore needs to be cognizant of the fact that the documents it is handed may start out in either the ACTIVE or the ACTIVEPURGATORY states. Should the processing succeed, the starting state does not matter – the Worker Thread will set a final state consistent with the crawling model being used. For a continuous crawl, PENDINGPURGATORY is the new state choice, along with a `checktime` field value consistent with the parameters of the crawl. This will determine when the document is next processed. For a non-continuous crawl, successful processing of a document results in the document being put into the COMPLETED state.

It is possible, of course, for the processing of a document to fail. Should this happen, the starting state of the document determines the final state. The Worker Thread places the document back into either the PENDING or PENDINGPURGATORY state, corresponding to a starting state of ACTIVE or ACTIVEPURGATORY. The `checktime` field is also given a value consistent with the kind of processing error that occurred. For example, if a `ServiceInterruption` exception was thrown by the underlying connector, the `checktime` field value will be set in accord with the retry time specified in the exception. The document will then be retried at some point in the future, when the Document Stuffer Thread once again identifies the document as being ready for processing. If continuous errors occur, then the document may be retried indefinitely. However, the repository or output connector may instead choose to provide ManifoldCF with a cutoff time, or maximum number of retries, after which the Worker Thread will abort the job, and record an appropriate error as part of the job's state.

CARRY-DOWN INFORMATION AND RESCANNING

The situation gets even more complicated when carry-down information is considered. If you recall, in Chapter 7 we first discussed carry-down information and explained how it transfers data from parent documents to child documents. Thus, processing a child document can depend on the results of processing its parent. The dependency implies that the way that a

child is passed to the target search index becomes dependent on the order in which processing occurs.

The way ManifoldCF deals with this dependency is to make sure that any changes to carry-down data for a child document cause the document to be processed again, if the document has already been processed. This is very easy to do if the child document is in the PENDING, PENDINGPURGATORY, or COMPLETED states – the document should be put into either PENDING or PENDINGPURGATORY again. But what should happen if the document is already ACTIVE, and is in the in-memory document queue?

You can see the problem. One Worker Thread may already be working on the child document, while another one is working on the parent, and is therefore in the process of potentially changing the carry-down information that the child will need for correct processing. There is no way for ManifoldCF to know what order this is going to happen in. Thus, this represents a classic race condition.

ManifoldCF solves this problem by introducing two additional document states. These states are ACTIVENEEDRESCAN and ACTIVENEEDRESCANPURGATORY. When a document's parent carry-down information is changed and the document is in either the ACTIVE or ACTIVEPURGATORY state, then the state of the document is changed immediately by the Worker Thread that is doing the changes to ACTIVENEEDRESCAN or ACTIVENEEDRESCANPURGATORY, respectively. Then, when the Worker Thread that was working on the child document is done processing it, it notices the changed state, and insures that the document is queued for a rescan. It does this by setting the document's state to PENDING or PENDINGPURGATORY, accordingly, instead of COMPLETED, thus insuring that the child document will be processed at least once more to resolve any carry-down changes that may (or may not) be needed. Similarly, when carry-down information for a child document changes and the child document is in the COMPLETED state, its state is updated to the PENDINGPURGATORY state, which causes the child document to be reprocessed with the changed carry-down information in effect. See figure 12.9 for a pictorial depiction of the relationships between documents and Worker Threads for this process.

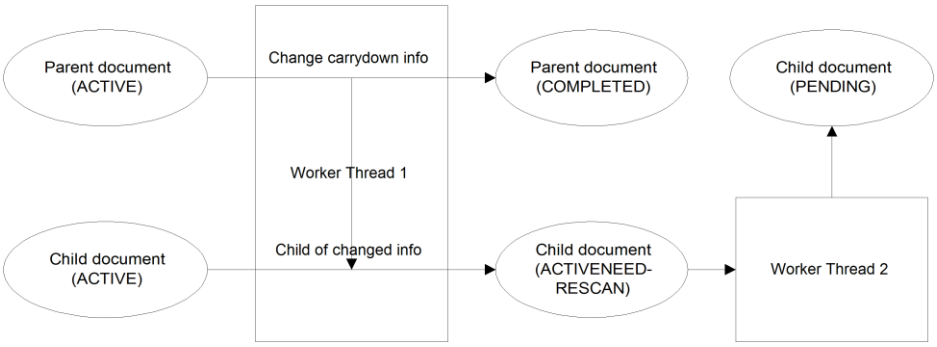

Figure 12.9 Diagram of what happens to an ACTIVE child document when the parent's carry-down information is changed. Note that the child document winds up in a different final state as a result of the change. The child document will eventually be reprocessed due to carry-down changes.

> **Note** It is up to the repository connector be sure that the version string it calculates for the document will be different when any of the incoming carry-down data which could affect indexing differs. If this is not true, then the repository connector has a bug, by definition.

12.3 Pull Agent document expiration threads

For continuous jobs in ManifoldCF, documents often need to be removed from the job and from the target search engine as they expire. This activity we call *expiration*.

Two types of Pull Agent persistent threads are involved in expiration. The first type identifies documents that should be expired, and places them on an in-memory queue. The second type takes documents off that queue, and performs the expiration itself.

If this logic seems familiar, that's because the very same model is used for document processing, which we described in a previous section. The details, however, differ in many respects. We'll discuss these threads next.

12.3.1 The Expire Stuffer Thread

The Expire Stuffer Thread is, in many ways, much like the Document Stuffer Thread in character. Its purpose is to identify documents that are ready for expiration, and hand these documents off to Expiration Worker Threads, which perform the actual deed.

Expiration is the process by which documents that are no longer in some way "current" are deleted from both the `jobqueue` database table and the target search engine. The time that a document will expire (or, indeed, if it will expire at all) is determined by the configuration of the crawling job, and the implementation of the underlying repository connector. As we have discussed, expiration only takes place for jobs that are continuous. The connector furnishes the document's *origination time*, which is the connector's best guess at when the document was authored. The expiration interval that you provide to the job is added to the document's origination time to calculate a default *expiration time.* But, the connector also has the option of directly overriding the current document's default expiration time, if it so chooses to do so.

The document expiration process itself does not involve the repository connector much. Many of the connector-specific document decisions that the Document Stuffer Thread needs to make – mainly involving assignment of documents to document bins – are not applicable to the Expire Stuffer Thread. This thread has no need for throttling, and correspondingly there is no document prioritization. The expiration family of threads uses an architecture that is otherwise quite similar to the worker family of threads that we've just finished discussing in the previous sections. Documents that are selected for expiration by the Expire Stuffer Thread get placed onto an in-memory queue and their states are changed

accordingly. Then they are acted upon by one of the several Expiration Threads, which actually do the deed and clean up afterwards.

The Expire Stuffer Thread considers many conditions to be sure a document is actually eligible for expiration. The job must be in one of the ACTIVE states, as is true for the Document Stuffer Thread as well. The documents themselves must be marked as needing expiration, as we have discussed, and the current time must be greater than the `checktime` field for the document. (The `checktime` field would have been set the last time the document was processed, based on what the connector determined was the document's time of origin.) In addition, the Expire Stuffer Thread also needs to be careful not to expire documents that are currently active elsewhere. Let's discuss how that is done.

DEALING WITH CROSS-JOB DOCUMENTS

Just as in the case of the Document Stuffer Thread, the Expiration Stuffer Thread must identify documents that belong to more than one job, and handle these specially. For example, a document may be ready to be expired in one job, but is still active in another. The Expire Stuffer Thread deals with this situation by skipping over all documents that also exist in other jobs, and are in any kind of active state, such as ACTIVE, BEINGCLEANED, or BEINGDELETED. This exclusion is performed as part of the query that identifies candidate documents for expiration.

However, even if a document is identified as being safe to process, there is another expiration-related concern. If the document is part of any other job which shares the same output connection, even if the document is not active at the moment in that job, the document should not be deleted from the target search engine. Doing so would remove a valid document belonging to the other job. Thus, an expiring document should not be deleted from the target search index if **any** job still considers it valid. But its `jobqueue` entry must still be deleted for the job where expiration is taking place. The Expire Stuffer Thread therefore passes a signal in the queue to whatever Expiration Worker Thread receives the document for expiration, signalling whether the document should be removed from the target search index or not.

PENDING VS. PENDINGPURGATORY

Like the Document Stuffer Thread, the Expire Stuffer Thread makes a distinction between the document states of PENDING and PENDINGPURGATORY. When it selects a document for expiration, the Expire Stuffer Thread places the document into the corresponding ACTIVE and ACTIVEPURGATORY states. This maintenance of states allows the Expiration Worker Threads to return the document to its original state, should it prove impossible to perform the expiration.

12.3.2 Expiration Worker Threads

Expiration Worker Threads are responsible for accepting documents from the Expire Stuffer Thread for expiration. There are usually a number of these threads; the default is ten, but

this number can be changed using the `properties.xml` property `org.apache.manifoldcf.crawler.expirethreads`.

An Expiration Worker Thread that accepts a document may first attempt to delete the document from the target search engine, if the Expire Stuffer Thread signaled that it should. If this is successful, the Expiration Worker Thread will then delete the document from the `jobqueue`, `carrydown`, `intrinsiclink`, and `hopcount` tables. Once the document is removed from these data structures, it is no longer present as far as ManifoldCF is concerned, and thus has no document state.

ACTIVE vs. ACTIVEPURGATORY

When the Expiration Worker Thread pulls a document off of the in-memory expiration queue, the document it receives is guaranteed to be in either the ACTIVE or the ACTIVEPURGATORY state, depending on what its state was when the Expire Stuffer Thread selected it. When the document is successfully expired, it no longer has any state. But what happens if the document cannot be expired successfully? After all, Expiration Worker Threads can receive transient errors from the target search engine when they try to remove expired documents.

When an Expiration Worker Thread receives such an error, the documents it was trying to delete from the search index are restored to the queue, and given either the PENDING or PENDINGPURGATORY state, with a `checktime` value indicating when the document will again be eligible for expiration. Thus, as the job continues to run, the Expire Stuffer Thread will eventually select the document again, and try to delete it from the search engine at that time. This cycle may continue indefinitely, ending only when the document is successfully removed from the target search engine. Or, the output connector may instead signal to ManifoldCF to stop trying after a certain number of tries or a certain period of time. In that case, the Expiration Worker Thread will abort the job, and log an appropriate error.

12.4 Pull Agent document cleanup threads

An activity known as *document cleanup* takes place at the end of every non-continuous job run. Document cleanup is the process of removing documents from the job queue and target search index that were identified and processed on a previous run of the job, but were not encountered during the current job run.

The threads and data structures that perform document cleanup are analogous to the threads and data structures we've seen for document processing and expiration. Two types of Pull Agent threads, and an in-memory queue, are used to perform document cleanup. One thread type identifies documents that should be cleaned up, and puts them into the in-memory queue. The second thread type takes documents from that queue and cleans them up. We'll describe the function of these threads in detail next.

12.4.1 Document Cleanup Stuffer Thread

When a job enters the SHUTTINGDOWN state, the documents that are in the PURGATORY state must be deleted from the job. As we discussed in Chapter 1, this is because those

documents are no longer reachable from the seed documents, and ManifoldCF no longer considers these to be part of the job.

For document cleanup, ManifoldCF uses an architecture that's very similar to the architecture it uses for document expiration. There is a stuffer thread – the Document Cleanup Stuffer Thread – which identifies documents that are in the PURGATORY state that have a `checktime` field before the current time. Then, it changes the state of identified documents to BEINGCLEANED, and places them into an in-memory queue. One of many Document Cleanup Worker Threads then takes the document off the queue and acts on it in an appropriate manner.

THE PURGATORY STATE

The Document Cleanup Stuffer Thread only selects documents for stuffing that are in the PURGATORY state. This is the state that documents which were formerly COMPLETED at the end of a job run are put into at the start of the subsequent job run. Documents that remain in the PURGATORY state at the end of the next job run have, by definition, become unreachable, and are thus eligible to be cleaned up.

Because there is only one possible starting state, the Document Cleanup Stuffer Thread does not have to make any decisions as far as the state it assigns to the document when it "stuffs" the document. The only state it can enter is the BEINGCLEANED state. This is unlike the Expire Stuffer Thread or the Document Stuffer Thread, which may get documents in one of two initial states, and must preserve the distinction between them.

DEALING WITH CROSS-JOB DOCUMENTS

When a document exists in two or more jobs at the same time, the same sorts of race condition concerns apply to the Document Cleanup Stuffer Thread as apply to the other stuffer threads we've discussed. The Document Cleanup Stuffer Thread must therefore avoid stuffing documents that are already active in some way. However, there is another case the Document Cleanup Stuffer Thread needs to handle – specifically, the question of how to actually process documents that are shared across jobs.

A shared document clearly must always be deleted from the job's `jobqueue` database table rows. Deleting the document from the target search engine is, however, a different matter. If another job owns the document as well, that job probably would rather not have the document ripped out from underneath it. It's easy to see that the actual search index deletion should only take place when the document is owned by only *one* job. This logic is quite similar to that used by the Expire Stuffer Thread. The Document Cleanup Stuffer Thread thus decides for each document whether it should be deleted from the target search engine or not, and communicates this information to the Document Cleanup Worker Threads along with the document in the in-memory cleanup queue.

12.4.2 Document Cleanup Worker Threads

The Document Cleanup Worker Threads correspond fairly closely in design and function to the Expiration Worker Threads, which we've already discussed elsewhere. There are usually

a number of these threads; the default is ten, but you can change this value using the `properties.xml` property `org.apache.manifoldcf.crawler.cleanupthreads`.

When a Document Cleanup Worker Thread pulls a document from the cleanup queue for action, it may first attempt to delete the document from the search index, if the Document Cleanup Stuffer Thread signaled that it should. If it succeeds at this, then it removes the document's entry from the `jobqueue`, `hopcount`, `intrinsiclink`, and `carrydown` tables, just like the Expiration Worker Thread does when it expires a document.

THE BEINGCLEANED STATE

For the document to be in the cleanup queue at all, the document's state must already have been set to BEINGCLEANED by the Document Cleanup Stuffer Thread. The document remains in this state until either it is deleted from the target index and from the job queue, or an error occurs during the deletion process. Prior to being given the BEINGCLEANED state, the document was in the PURGATORY state. So it is quite clear what must be done to handle any errors that might come up when the Document Cleanup Worker Thread attempts to delete the document from the search index – it should put the document back into the PURGATORY state, and set an appropriate value in the document's `checktime` field.

SIDE EFFECTS OF DELETING CARRY-DOWN INFORMATION

There is another side effect that might happen when a document is cleaned. It is possible that removing the document might change carry-down information for a child document which remains part of the job.

A similar thing takes place when a document is processed by either a Worker Thread or an Expiration Worker Thread. As you recall, we discussed how changes to carry-down information when a job is running may cause some previously completed documents to return to the PENDINGPURGATORY state. But, in the case of document cleanup, the job is no longer even ACTIVE – it is, by definition, in the SHUTTINGDOWN state. If, during cleanup, carry-down information for a document is changed, the Document Cleanup Worker Thread puts the document back into the PENDINGPURGATORY state. Later, when all the PURGATORY documents have been removed, the Job Reset Thread will notice that PENDINGPURGATORY documents exist, and cause the job to re-enter the ACTIVE state, instead of changing its state to INACTIVE, as originally planned. This cycle will repeat until all processing is complete.

12.5 Pull Agent document deletion threads

When a job is deleted in ManifoldCF, all the documents belonging to the job must be removed from the target search engine. This process is called *document deletion*, and the Pull Agent uses threads and data structures similar to those used for document processing, expiration, and cleanup to perform it.

For document deletion, there are, once again, two types of threads and an in-memory queue involved. The documents to be deleted are identified and placed onto the queue by

the first type of thread. The documents are actually deleted by the second type of thread. We'll describe these threads next.

12.5.1 The Document Delete Stuffer Thread

When a job is deleted, the documents belonging to the job must possibly be removed from the index, and must definitely be removed from the `jobqueue` database table and related database tables. By now, it will come as no surprise to you to learn that document deletion of this kind is managed by a stuffer thread and a pool of worker threads. The Document Delete Stuffer Thread is the thread that performs the "stuffing", and the Document Deletion Worker Threads are the ones that perform the actual deletion of the document from the search engine.

THE ELIGIBLEFORDELETE STATE

At the time a job deletion sequence begins, ManifoldCF puts all documents in the job into the ELIGIBLEFORDELETE state. Thus, the Document Delete Stuffer Thread need only look for documents in that state to find documents that are eligible for deletion.

Like all stuffer threads, the Document Delete Stuffer Thread runs all the time, and selects documents for stuffing based on what's available. To qualify, a document must be in one of the specified states, and the job to which the document belongs must be in the DELETING state. Also, the `checktime` column value for the document must be less than the current time. This makes it possible to handle delete errors by setting the state of the documents that received the errors back to ELIGIBLEFORDELETE, but with a later `checktime` value.

The Document Delete Stuffer Thread changes the state of all documents that it selects from the ELIGIBLEFORDELETE state to the BEINGDELETED state. This reflects the fact that these documents are already in the in-memory document deletion queue, and should not be selected again.

DEALING WITH CROSS-JOB DOCUMENTS

For the Document Delete Stuffer Thread, the same issues exist concerning documents that are shared among jobs as do for the other stuffer threads we've already discussed. In order to avoid race conditions, the Document Delete Stuffer Thread must avoid selecting documents that are active in any other job. And the same situation occurs that we've described for the Expire Stuffer Thread and Document Cleanup Stuffer Thread, with respect to documents that are not active but are nevertheless shared: the document should not be deleted from the target search engine unless it is the last reference to that document.

12.5.2 Document Deletion Worker Threads

Document Deletion Worker Threads accept documents from the in-memory document deletion queue for removal. These documents have been placed on the queue by the Document Delete Stuffer Thread.

There are usually multiple Document Deletion Worker Threads running at one time. You can in fact control how many ManifoldCF starts by setting a value in the `properties.xml`

file called `org.apache.manifoldcf.crawler.deletethreads`. The default value, if there is no property set, is ten.

A Document Deletion Worker Thread can do only two things with a document. The first thing is to delete the document from the target search index. The second is to remove the document from the `jobqueue`, `carrydown`, `intrinsiclink`, and `hopcount` database tables.

It turns out that it is much more efficient to remove all job-related rows at the end of the document deletion process than to clean them up a document at a time. So the Document Deletion Worker Thread takes a shortcut: it only updates the `jobqueue` database table as it goes along, because it knows there is no chance that the job will ever become active once more. It nevertheless updates the `jobqueue` record because that is the only way to keep track of the state of the document.

If the deletion of the document from the target search engine fails, then the document is returned to the ELIGIBLEFORDELETE state in the `jobqueue` database table, with a `checktime` column value consistent with whatever the target search engine suggests for a retry time. If the error occurs repeatedly, then the document will be retried indefinitely. There is no option to give up, because the decision to delete a job is irrevocable in ManifoldCF, and thus document deletion must go to completion in one way or another.

12.6 Pull Agent job transition threads

In addition to the four families of threads we've just covered that deal with documents in bulk, there are a number of miscellaneous threads in the Pull Agent that take care of transitioning jobs from one state to another. In general, these threads identify jobs that require a transition, and perform that transition. We'll go through these threads one at a time.

12.6.1 The Job Start Thread

As you can see from table 12.1, the Job Start Thread is responsible for performing schedule evaluation, and using a job's schedule to decide when to activate it or put it to sleep. Its logic involves the following steps:

1. Identify jobs that are INACTIVE, or in any of the WAIT states
2. Evaluate the schedule for those jobs to decide if the job should be started or made active
3. If the schedule so indicates, then change the job's state to either READYFORSTARTUP, or a corresponding non-WAIT state
4. Identify jobs that are in any of the active non-WAIT states
5. Evaluate the schedule for those jobs to decide if the job should be paused
6. If the schedule so indicates, then change the job's state to the corresponding WAIT state

For example, suppose that a job is in the INACTIVE state, but it is supposed to be started on a schedule. The Job Start Thread will then examine this job every ten seconds or so, and figure out of the start of a schedule window has passed since the last time the job's schedule was evaluated. (The time of last schedule evaluation is kept in the `jobs` table in the `lasttime` column). If it has, then the job is moved to the READYFORSTARTUP state, where other threads then execute the job startup sequence.

This thread is also responsible for putting jobs that execute past the end of their schedule window into an appropriate WAIT state. The WAIT states for a job are ACTIVEWAIT, PAUSEDWAIT, ACTIVEWAITSEEDING, and PAUSEDWAITSEEDING. These correspond to the non-WAIT states ACTIVE, PAUSED, ACTIVESEEDING, and PAUSEDSEEDING. The WAIT states behave identically in every way with their non-WAIT brethren, with the exception that they do no crawling, because by definition they imply that the current time is not within the job's scheduling window.

The Job Start Thread also reverses the assignment of a job to a WAIT state, once it enters a new scheduling window. It does this by noticing if the beginning of a schedule window was crossed since the last time the Job Start Thread checked the job's schedule.

12.6.2 The Finisher Thread

The Finisher Thread's sole purpose is to identify when an ACTIVE job has completed, and put it into the SHUTTINGDOWN state. The way the Finisher Thread does this is by periodically checking whether the job is in an ACTIVE state, and whether there are any active or pending documents corresponding to the job in the `jobqueue` database table. If no such documents are found, then the Finisher Thread puts the job into the SHUTTINGDOWN state. At this point, other threads take care of the post-crawl document cleanup. When the cleanup is complete, another thread has the responsibility of returning the job to the INACTIVE state. We'll discuss that thread now.

12.6.3 The Job Reset Thread

The purpose of the Job Reset Thread is to identify jobs in which crawling has been completed, or jobs that have been aborted with no remaining active documents, and transition these jobs to the READYFORNOTIFY state.

The Job Reset Thread thus periodically queries for jobs that are in the SHUTTINGDOWN or ABORTING states. For jobs in the SHUTTINGDOWN state, they are eligible for transition to the READYFORNOTIFY state only when there are no remaining documents in the PURGATORY or BEINGCLEANED states. For jobs in the ABORTING state, the criteria are that the job can have no remaining ACTIVE documents.

After the transition to READYFORNOTIFY has been made, another thread handles the actual output connection notification. We'll discuss that next.

12.6.4 The Job Notification Thread

The Job Notification Thread's purpose is to notify the output connector when a job is done, so that the connector may choose to act on this knowledge.

Output connector notification is a requirement of Solr. In Solr, documents must be committed to be indexed or removed. Without some kind of signal (or notification) at the end of a job, there would be no automatic way for this commit operation to be performed.

The Job Notification Thread periodically looks for jobs that are in the READYFORNOTIFY state. It transfers these jobs to the NOTIFYINGOFCOMPLETION state, and then attempts to perform the notification operation. If the notification was successful, the Job Notification Thread then changes the state of those jobs to INACTIVE.

If the operation is unsuccessful, then the transition to INACTIVE is still made, because output connection notification is considered a nicety, not a strict requirement.

12.6.5 The Seeding Thread

The Seeding Thread is only helpful for jobs that are running continuously. Its purpose is to periodically reseed the job's `jobqueue` table with seed documents. This functionality is there so as to allow a user to change the job's seeds on the fly, without stopping and restarting the job.

Each continuous job has its own seeding interval, which is set in the crawler UI or via the API service. The Seeding Thread keeps track of the last time that seeding took place for each job (it's kept in the jobs table in the `lastchecktime` column), and periodically assesses each active, continuous job to decide if seeding should be done. If the decision is made to go ahead, then the job's ACTIVE state is changed to the appropriate SEEDING variant, and seeding begins. When seeding is complete, the job's state is changed back to the appropriate non-SEEDING state.

12.6.6 The Start Delete Thread

The Start Delete Thread periodically identifies jobs that are in the READYFORDELETE state. It transitions each such job to the DELETESTARTINGUP state, and then prepares the documents in the job for deletion. When that is done, it places the job into the DELETING state. Other threads then take over to perform the document deletion phase.

The preparation for a job entering the DELETING state involves moving all of its documents to the ELIGIBLEFORDELETE state. At this time the `checktime` column for all of the job's documents is also set to zero, which makes all documents in the job ready to be deleted right away. If these preparatory changes to the `jobqueue` database table were not made, then it might be the case that a document would already have a value in the `checktime` column that was inconsistent with immediate deletion.

With the Start Delete thread, we've completed our exploration of the threads used by the Pull Agent. Next, we'll move on to the Authority Service, and look at the threads used there.

12.7 Authority Service threads

The Pull Agent is not the only entity in ManifoldCF that uses persistent threads. The Authority Service also has a need for parallelism, and handles this need by maintaining a pool of threads. We'll describe the Authority Service threads in this section.

12.7.1 Request processing threads

As you no doubt recall, the Authority Service is a web application that responds to requests for a user's access tokens. Web applications run within Java application servers. Incoming web requests are typically handled by the application server by being dispatched to a request processing thread, which the application server usually pulls out of a pool that it keeps.

So here I have fooled you. I stated above that we'd talk only about ManifoldCF persistent threads, but instead I'm talking about threads that are managed by an application server! But that's okay, because these threads are still persistent threads as we've defined them, and there is no restriction on using them exactly as if they originated within ManifoldCF itself.

A request processing thread essentially is the agent by which a request is transmitted into the Authority Service servlet. The thread is thus responsible for the following activities:

1. Parsing the request
2. Adding an authorization request for each defined authority to an in-memory queue
3. Waiting for all of the authorization requests to complete
4. Assembling the output response

So, how are the actual authorization requests performed? That's the job of the Auth Check Threads. We'll talk about them next.

12.7.2 Auth Check Threads

The access tokens for any user may come, in part, from each authority connection that has been created in ManifoldCF. The Authority Service aggregates these disparate tokens into one list.

In a complex environment, there may be many such authority connections. It's not uncommon at all to find enterprises where there are a dozen in use at once. However, if we are not very careful, this could easily result in an unacceptable degradation of search performance, because each search request will need to obtain the necessary access tokens before it can even begin. If we tried to get these tokens from one authority connection at a time, then the system would get slower and slower the more authority connections were added.

Thus, in order to obtain such tokens quickly, the Authority Service uses a separate persistent thread to obtain the user's tokens for each authority connection. These threads are called Auth Check Threads, and they obey all the rules for persistent threads described in Chapter 5. See figure 12.10.

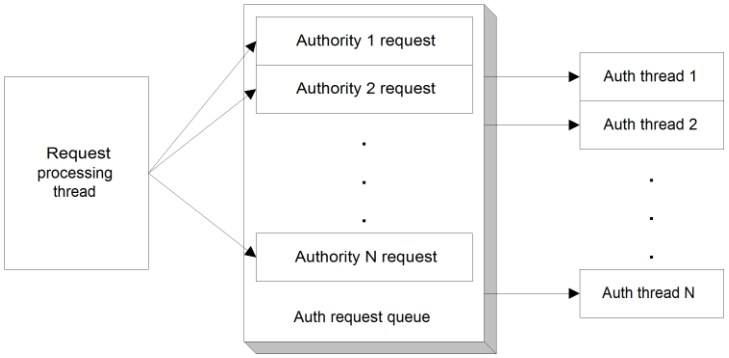

Figure 12.10 Processing a request involves the request processing thread producing an authority request for each valid authority, and adding these to the auth request queue. A pool of Auth Check Threads then services the authority requests. When done, the request processing thread amalgamates the results from each authority request and returns them.

A fixed-sized pool of Auth Check Threads is created when the Authority Service starts. The default number of threads is ten, but you can change this through the `properties.xml` parameter `org.apache.manifoldcf.authorityservice.threads`.

Each Auth Check Thread waits for an authorization check request to appear on the in-memory queue mentioned earlier. When it finds one, it invokes the specified authority connection on behalf of the requested user, and receives either a list of user access tokens, or an error. In either case, the Auth Check Thread fills in the authorization check request with the response, and goes back to look for another.

There's very little else that needs to be said about Authority Service threads. Because there is no need for the Authority Service to have states or persistence, there are no state transitions involved. This keeps the function of the Authority Service relatively simple, which I am sure you will consider welcome news after looking at the complexities of the Pull Agent.

12.8 Summary

In this final chapter, we've learned a great deal about the internal thread architecture of ManifoldCF, and how it actually does what it sets out to do. We've looked in depth at the Pull Agent persistent threads involved in managing jobs and documents, and hopefully clarified exactly how transitions occur between job and document states. We've also begun to explore the true reasons each job or document state exists. And, last, we looked briefly at the logic within the Authority Service, and noted how much less complex it is than the logic within the Pull Agent.

My one regret for this chapter is that it does not include detailed descriptions of some of the more interesting algorithms ManifoldCF uses to crawl documents. The algorithms used for load balancing, document prioritization, and hop count computation are all worthy subjects, which must unfortunately wait for a future book to do them justice.

Nevertheless, I hope you have enjoyed reading ManifoldCF in Action, and will find it useful in whatever endeavors await you.