11

Data structures and Resource Management

This chapter covers

- ManifoldCF data structures
- ManifoldCF database tables
- How ManifoldCF manages resources

In this chapter, you will learn about the data structures that make up ManifoldCF. You will learn how these data structures are mapped to database tables, and that database tables are the only persistent data structures that ManifoldCF uses. You will see how this early design decision contributed greatly to ManifoldCF's resilience against interruption, restart, and errors – at the expense, to some degree, of raw crawling speed.

The information contained in this chapter is not essential for you to know in order to use or extend ManifoldCF. Clearly, you do not need to know about ManifoldCF's internal data structures in order to (for instance) use the Crawler UI, or write code that talks to the API Service. But when you set out to write connectors, you will find that although there is also no explicit connection between the connector code you will write and the data structures you will learn about here, the knowledge you gain may make you more effective in wisely using the constructs ManifoldCF makes available to connector writers. And you will understand, in much greater depth, some of the design tradeoffs that shaped ManifoldCF into its current form.

I will begin this chapter with a brief description of the persistent data storage requirements of ManifoldCF, and the technologies available to meet these requirements.

After this introductory section, I will move on to introduce the simplest persistent data structures: the ones that handle connector registration. We'll start simple in order to give you a chance to be introduced to the concepts before we move to more complex structures. Section by section, we'll work our way up to the critical data structures that are the core of ManifoldCF.

11.1 Philosophy

Where data structures are concerned, ManifoldCF has a very clear design philosophy. In this section we'll briefly examine this design philosophy, from the perspective of the choices that would be available to any software project. Then we'll examine the implications of ManifoldCF's chosen philosophy in more depth.

11.1.1 Ways of representing data structures

There are many ways to represent a data structure. Your standard college *Data Structures and Algorithms* class typically introduces you to such data structures at a low level, where you likely have learned that you choose your data structure based on the algorithm you can best use to achieve performance results consistent with the overall problem you are trying to solve.

I'm not intending to review any of that basic material here. Instead, I'm going to look at the choices of data structure with an eye towards rather different evaluation criteria – specifically, the requirements that come from running as part of a daemon process in a server environment. These are summarized below.

- *Resilience* – if the system or process is shut down suddenly, how difficult is it to resume operations?
- *Error tolerance* – if a problem occurs, does the system wind up in an inconsistent state, or can it resume without difficulty?
- *Memory bounding* – is it possible to set the amount of memory the process will need in advance, regardless of size of task, without trial-and-error experimentation?
- *Performance* – do the performance characteristics of the system scale in a manner that permits it to handle data sets of an interesting size?

We'll start by considering the simplest possible data representation model, and move onward from there.

IN-MEMORY STRUCTURES

Storing all data in the process memory or virtual memory has a very long history behind it. After all, this is the default way most computer programs do things, and it is by far the fastest. But this approach has some serious problems when we look at the four criteria we enumerated above. Let's see how it stacks up.

First, the resilience of pure in-memory structures is awful. Shutting the process down loses everything for data structures that are handled entirely in memory; the entire data structure must be recreated from scratch. This alone disqualifies a pure in-memory approach from being used for countless tasks.

Second, error tolerance is only as good as the programmer is. Coding for error tolerance may require explicit ad-hoc structures to be built, and great care must be taken to analyze all possible error conditions.

Third, a data structure represented entirely in memory is not going to be bounded in any way whatsoever; the structures will clearly grow as the size of the problem grows. The best that can be achieved is to limit the rate of growth.

The only category where in-memory data structures excel is in performance. Because you get to pick the data structure and the algorithm for working with the structure, you can make the structure perform as well as the best available algorithm will permit.

DISK-BACKED IN-MEMORY STRUCTURES

In the disk-backed model, an in-memory structure has a disk file associated with it. At the beginning of the process, the disk file is read, and periodically throughout the execution of the process, the disk file is updated with the current contents of the in-memory data structure. The hope is that it is then possible to restart the process from its disk file with a minimum of fuss.

So how does a structure like this stack up? Well, introducing the disk makes it at least **possible** to meet most of our goals in a reasonable manner, at the expense of placing more burdens on the programmer to do the right thing. For example, the programmer must write to the disk file at the correct times, and must also deal with the possibility of the process being shut down during the write of the disk file. There is also a startup penalty when the data is large, because all of the data must be copied into memory at startup time. But, of the four criteria we put forward, the only criteria that cannot be met with at all with this model is the memory bounding criteria, because all of the data must still fit in memory.

PURE DISK STRUCTURES

Storing the data structure entirely in a disk file is the next model we'll consider. In this model, a file containing the data is accessed and manipulated by the program. Thus, very little of the data is in the process memory at any given time.

A pure disk data model is well suited to controlling memory bounds, and can be made resilient and error tolerant, once again depending on the quality of the programmer. But, since it is often costly to seek to specific file data elements, or modify a file starting in the middle, this model will fail the performance criteria unless the programmer is very rigorous about how the data on disk is actually represented. By introducing extra disk files that contain indexes, performance just **might** be tolerable. By keeping files that contain transaction logs, a programmer can manage resilience and error tolerance with a relatively low performance penalty. So, in order to use a purely disk-based data structure, and still meet all of our criteria, the programmer needs to, in effect, write a database.

11.1.2 Implications of using a database for persistence

Using a database for a data structure meets the ManifoldCF goals of resilience (durability, in database parlance), error tolerance, memory bounding, and performance. For this reason, very early in the development of ManifoldCF the decision was made that ManifoldCF would use a database for all persistent storage.

However, it soon became clear that other factors also applied, and other data structure choices would need to be made. There are many different kinds and implementations of databases in the world today. Choosing the right technology and best implementation is not at all trivial. ManifoldCF's criteria for making this choice are as follows:

- Must work well in a multithreaded environment
- Needs to be powerful enough so that large result sets do not need to be pulled into memory
- Must be able to handle multiple indexes for each table
- Must be cheap

The last point is actually a critical one. Cost is often overlooked until far too late in a design process. If you are selling software or hardware for a competitive price, you often cannot afford to include (or require that your customer purchase) an expensive database license as well. In ManifoldCF's case, the initial requirements were even more drastic: the database had to be free.

SQL DATABASES

An SQL database is an implementation of the SQL language. This language describes relations, typically in the form of tables of multiple rows and columns, and allows for a rich set of abilities to process and retrieve that data, always in the form of a result set. Each result set also has multiple columns and multiple rows.

SQL databases in general provide many of the characteristics ManifoldCF is looking for. Resilience and fault tolerance are present because SQL has a concept of transactions built into it. The power of the SQL language makes it easy to structure queries so that large result sets do not need to be loaded into memory. And it is always possible to declare multiple indexes on a single table.

After that, however, it gets trickier. Many open-source databases exist, and they are cheap (or free). A partial list includes Apache Derby, HSqlDB, MySQL, and PostgreSQL. There are two flavors of such databases: the kind that can run as embedded software, and the kind that runs as a standalone process, and communicates with the database clients using socket connections. Some open-source databases permit both models, notably Derby and HSqlDB. However, in this author's experience, very few of these handle a multithreaded environment well. At one point fairly recently, HSqlDB would internally deadlock quite easily. Apache Derby used to also deadlock itself, but would recover after a timeout elapsed. While both databases have since released newer versions that do not have these problems, the ability to properly handle multithreadedness seems to be a criterion that separates the men from the boys.

PostgreSQL is an open-source, free database that meets all of the criteria. The main downside to PostgreSQL is that it cannot be run in embedded mode, which means that you will always need to manage at least one separate process in order to use it. Another issue with PostgreSQL is that new versions come out all of the time, and some have bugs that interfere with the operation of ManifoldCF. But, on the whole, PostgreSQL remains the best choice for ManifoldCF.

What about MySQL? Early on in ManifoldCF's development, the principle database used was indeed MySQL. However, MySQL's compatibility with SQL standards took a long time to materialize, and since then, the database has evolved away from the pure open-source model. Thus, the decision came very early on in ManifoldCF's development to switch from MySQL to PostgreSQL, before many of the current ManifoldCF features were complete. Later, MySQL got better, and ManifoldCF began supporting it once again fairly completely. In my experience, it still does not perform for ManifoldCF as well as PostgreSQL does, although it is a reasonably close second. We do support it as a first-tier database choice at this time.

By now it is no doubt becoming clear that choosing the right database was perhaps the most important decision that was made during ManifoldCF's development. A great deal of effort went into both the database choice, and the evaluation of that choice in real-world situations. Nevertheless, there remain other choices that may someday be a part of ManifoldCF's future.

NO-SQL DATABASES

There is another choice of database technology that has become increasingly popular in recent years. A No-SQL database typically is disk-based key/value store, something akin to multiple giant hash tables. Each hash table is the rough equivalent to a table, in standard SQL.

No-SQL databases have many of the attractive characteristics of SQL databases (e.g. resilience, error tolerance, performance, and multithread support), without some of the overhead and quirkiness that SQL brings to the picture. They are meant to be fast, resilient data stores that access data primarily via a key.

The place where No-SQL databases currently fall down for ManifoldCF is in the memory bounding constraint. The retrieval characteristics of No-SQL, while perfectly fine for some applications, are not well suited for ManifoldCF's needs. Specifically, ManifoldCF has a critical requirement to be able to read specific small subsets of potentially very large result sets. In SQL, this is done by creating an index that matches the ordering required for the retrieval, and thus the retrieval becomes effectively a read of the index itself. But as far as I am aware there aren't comparable structures in common No-SQL implementations that appear to permit this kind of functionality. This may change at any moment, if it hasn't already.

So, while it may be possible eventually to come up with a No-SQL-based way of doing what is needed, for the moment ManifoldCF is based firmly in the SQL world. We now understand what the reason for this design decision was, and what it gained. In the rest of this chapter, we'll look into the details of how the persistent data in ManifoldCF is managed, and the database tables where this persistent data is stored.

Browsing ManifoldCF database tables yourself

If you are using PostgreSQL, you have the ability to use a PostgreSQL utility called `psql` to examine what is in your database tables. This can be very helpful in understanding what ManifoldCF is doing. If you are using the default database and user names, then starting `psql` is as simple as typing the following command:

```
psql –U manifoldcf dbname
```

You will then need to supply the `manifoldcf` user's password, which by default is `local_pg_passwd`. Once inside, you may type a number of commands, such as `\dt` to list all the database tables, as well as entire queries if you want to see the contents of these tables. The command `\q` exits `psql`, and there is online help available by typing the command `\?`.

On Windows, `psql` can be usually found at `c:\Program Files\PostgreSQL\<version>\bin\psql.exe`.

11.2 Connector management

In this section we will explore the data structures used to manage connectors. We'll cover all three kinds of connectors – output connectors, authority connectors, and repository connectors – since connector management is very similar for each of the three.

It is not enough for someone to write a connector in order for ManifoldCF to be able to use it. Somehow, ManifoldCF needs to learn about the connector class's existence. There are also times when ManifoldCF needs to know the complete list of connectors available, along with some sort of human-readable description for each. (If you can't remember when this might happen, then I suggest you revisit Chapter 2.) In addition, there are also more functional reasons why ManifoldCF needs to know whether a connector's class is actually present or not: there job states that keep ManifoldCF from spinning its wheels uselessly if a connector that is required for a job is no longer present.

11.2.1 Registration and de-registration

The model that ManifoldCF uses to track connectors and authorization domains is a *registration* model. That is, you register your connector class with ManifoldCF, along with a human-readable description, and only then will ManifoldCF users be able to find it.

As you might guess by now, the list of connectors and their descriptions is kept in a persistent data structure – a database table. Each kind of connector has its own database table to keep track of registration information. See table 11.1 for a run-down of the tables used for each kind of connector.

Table 11.1 Database tables used for output connector registration, authority connector registration, repository connector registration, and authorization domain registration.

Type of connector	Database table name
Output connectors	`outputconnectors`
Authority connectors	`authconnectors`
Mapping connectors	`mapconnectors`
Repository connectors	`connectors`
Authorization domains	`authdomains`

> **Note** As ManifoldCF has evolved, its database schema has changed, and it has changed more than once. Rather than attempt to define a schema that would be good for all time, ManifoldCF's schema was designed to fit the task as it was known, with the full knowledge that schema changes would therefore need to take place. Since this need has been anticipated from the beginning, ManifoldCF's database schema installation logic also contains schema upgrade logic. What summaries of the schema I present in this book must also be considered impermanent for the same reason.

THE OUTPUTCONNECTORS, AUTHCONNECTORS, MAPCONNECTORS, AND CONNECTORS TABLES

The schema for each of these tables is identical, and is very simple and straightforward. See table 11.2 for a description of each column.

Table 11.2 Schema for the outputconnectors, authconnectors, mapconnectors, and connectors tables.

Column	Meaning
`classname`	This column contains the Java class name of the class that implements the connector.
`description`	The `description` column is the human-readable text that is provided whenever something about the connector is returned in the UI or API.

But, knowing the underlying tables and schema is not quite enough, because in order to understand how ManifoldCF manages this data, we also really need to understand what happens when we register or unregister a connector.

Connector class registration

When a connector is registered, the obvious entry is created within the appropriate connector table. In addition, the connector class itself is called to perform whatever it deems necessary to install itself. We have already discussed the connector class methods in Chapter 6. The method that is called is the connector's `install()` method.

Most connectors do not need any persistent data beyond what the framework already provides for its standard services. However, once in a while, a connector will need to maintain persistent data structures of its own. In these cases, the connector's `install()` method may use the framework's database services to create tables of the kind that the connector requires. As of this writing, the web connector is the only included ManifoldCF connector that uses this feature. The data that the web connector stores persistently are the robots.txt files from various servers, and the mappings between domain names and IP addresses.

The final thing that happens when a connector is registered is that other components of ManifoldCF are notified of the change. This is necessary, because jobs that have been paused because the needed connectors were uninstalled may then be resumed. We'll talk about this in more detail in a moment.

Connector class deregistration

A connector can also be deregistered, or uninstalled. When this occurs, the corresponding database table is updated. But, also, the connector class's `deinstall()` method is called. This allows the connector to clean up any persistent database tables that it created at installation time.

As you might guess, when a connector is deregistered, other components of ManifoldCF are notified. Jobs that are running that depend on the deregistered connector are put into a special state that corresponds to the fact that they are active, but they cannot make progress. If jobs were not modified in that way, then such jobs would continue to run, but would do nothing useful, and would instead generate reams of error messages. The special state can be reversed only if the connector is reregistered, or if a user stops the job manually.

The authdomains table

Authorization domains are kept track of in the `authdomains` database table, which has the schema described in table 11.3.

Table 11.3 Schema for the authdomains table

Column	Meaning
`domainname`	The name of the authorization domain
`description`	A description of the authorization domain

Now you should have a basic idea of how connector management works, and the persistent data structures that are behind it. Next, we'll look at how connection definitions are stored and managed.

11.3 Authority group management

Authority groups tie together repository connections and authority connections. Multiple repository connections can belong to a single authority group, as can multiple authority connections. But about the only thing that an authority group actually needs, as far as data is concerned, is a unique name. Authority group management is thus extraordinarily simple, so it is a good place to start.

11.3.1 Authority group database table

There is only one database table in which authority group information is stored. The name of the table is `authgroups`. See table 11.4.

Table 11.4 Schema for the authgroups table

Column	Meaning
`groupname`	This is the name of the authority group, which is limited to 32 characters.
`description`	This is the description of the authority group, which is limited to 255 characters.

11.3.2 Authority group usage and editing

Authority groups, while extremely simple, make use of the same sort of editing paradigm used widely in ManifoldCF.

The authority group manager allows access and editing of authority group definitions using a *paper object* model. In a paper object model, a java class instance represents an in-memory picture of the persistent object. This class, an instance of which is called a *paper object*, has no logic whatsoever for filling the object in from the database tables it is stored it. Instead, an associated *manager* class provides the necessary methods for loading the object and saving it when it is changed. The editing cycle in the paper object model, therefore, involves loading or creating a paper object using the proper manager class, editing the paper object, and then saving it (once again using the manager class). See figure 11.1.

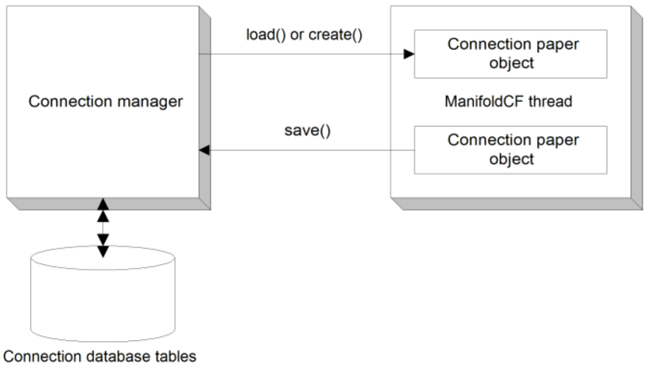

Figure 11.1 The flow for using or editing a paper object. The paper object metaphor is quite useful for editing complex objects, and for obtaining self-consistent views of such objects.

And that's all there is to how ManifoldCF deals with authority groups. Let's now look at connection definitions.

11.4 Connection definition management

Connection definitions – that is, output connection definitions, authority connection defintions, and repository connection definitions – are also stored persistently. All connection definitions consist of, at least, a name, a description, a connector class, and some connector class-dependent configuration information. But connection definition management is somewhat more complicated than connector management, because each kind of connection definition has some data that is unique to it. For example, a repository connection definition has information about throttle bins that an authority connection definition doesn't need.

11.4.1 Connection definition database tables

The database tables that are used to store each kind of connection definition are summarized in table 11.5.

Table 11.5 Database tables used for each connection type.

Connection type	Database tables
Output connection definitions	`outputconnections`
Authority connection definitions	`authconnections`
Mapping connection	`mapconnections`

definitions	
Repository connection definitions	`repoconnections`, `throttlespec`, `repohistory`

Notice that repository connection definitions have three tables involved in their management, where output and authority connection definitions only have one. Why is that? Simply, there is a lot more data associated with repository connection definitions than with the other kinds. We have mentioned throttle specifications already, and that is what the `throttlespec` table is for. But in addition, as you might recall, all the history information that is retained for history reports is also associated with repository connection definitions. The `repohistory` table is where this information is kept.

THE OUTPUTCONNECTIONS TABLE

The schema for the `outputconnections` table is captured in table 11.6.

Table 11.6 Schema for outputconnections table

Column	**Meaning**
`connectionname`	This is the name of the connection definition, which must be unique.
`description`	This column, if non-null, contains a long description for the connection definition.
`classname`	The content of this column is the class name of the connector for the connection definition.
`configxml`	This column contains the connection definition's XML configuration information, whose form depends on the connector specified.
`maxcount`	This column contains an integer, which is the maximum number of allowed connection instances, per JVM.

THE AUTHCONNECTIONS TABLE

The schema for the `authconnections` table is described in table 11.7.

Table 11.7 Schema for authconnections table

Column	**Meaning**
`authorityname`	This is the name of the connection definition, which must be unique.
`description`	This column, if non-null, contains a long description for the connection

	definition.
classname	The content of this column is the class name of the connector for the connection definition.
configxml	This column contains the connection definition's XML configuration information, whose form depends on the connector specified.
mappingname	If present, the name of the mapping connection that must be evaluated before this authority connection is invoked.
authdomainname	This column contains the name of the authorization domain for which this authority connection should be invoked. A blank value represents the "default" domain.
groupname	This column contains the required authority group name for the authority.
maxcount	This column contains an integer, which is the maximum number of allowed connection instances, per cluster.

THE MAPCONNECTIONS TABLE

The mapconnections table's schema is similar in most ways to that of the authconnections table. See table 11.8.

Table 11.8 Schema for the mapconnections table

Column	**Meaning**
mappingname	This is the name of the connection definition, which must be unique.
description	This column, if non-null, contains a long description for the connection definition.
classname	The content of this column is the class name of the connector for the connection definition.
configxml	This column contains the connection definition's XML configuration information, whose form depends on the connector specified.
mappingname	If present, the name of the mapping connection that must be evaluated before this mapping connection is invoked.
maxcount	This column contains an integer, which is the maximum number of allowed connection instances, per cluster.

THE REPOCONNECTIONS TABLE

The repoconnections table has much in common with the outputconnections and authconnections table, but has an additional column which describes the repository connection definition's affinity with an authority connection definition. See table 11.9.

Table 11.9 Schema for the repoconnections table

Column	Meaning
connectionname	This is the name of the connection definition, which must be unique.
description	This column, if non-null, contains a long description for the connection definition.
classname	The content of this column is the class name of the connector for the connection definition.
configxml	This column contains the connection's XML configuration information, whose form depends on the connector used.
maxcount	This column contains an integer, which is the maximum number of allowed connection instances, per JVM.
groupname	This column contains the name of the authority group which secures all documents indexed from this repository connection definition, if any.

THE THROTTLESPEC TABLE

The throttlespec database table, as we have already discussed, contains information about the throttles we have associated with each repository connection definition. Its schema is described in table 11.10. There is one throttle in this database table per row.

Table 11.10 The schema of the throttlespec database table

Column	Meaning
ownername	This column contains the name of the repository connection definition that owns each database table row.
description	This value is the text that describes what the throttle is for.
matchstring	This value is the regular expression that is used to describe the class of throttle bins to which this throttle applies. It must be unique.
throttle	This column contains a floating point value, in documents per millisecond, for the maximum average rate of document fetches if this throttle applies.

THE REPOHISTORY TABLE

Finally, the `repohistory` table contains all of the history records that have been accumulated for all repository connection definitions. This table may grow quite large. Its schema is described in table 11.11.

Table 11.11 The schema of the repohistory database table

Column	**Meaning**
`id`	This column contains a unique numerical identifier.
`owner`	This column contains the name of the repository connection definition that owns each database table row.
`activitytype`	The `activitytype` column contains a string value corresponding to the kind of activity recorded for each record.
`entityid`	This column has a string value, which depends on the kind of activity type the record contains, that describes the entity involved in the activity.
`datasize`	This column contains a numeric value, representing the number of bytes involved in the activity.
`resultcode`	The resultcode column contains an activity-type-dependent string describing the class of result for the activity. For example, a value of "OK" may represent successful completion for some kinds of activity, while "ERROR" may indicate a failure.
`resultdesc`	This column contains a verbose summary of the result, meant to be read by a person, rather than queried upon.
`starttime`	This is the time, in milliseconds since Jan 1, 1970, when the activity began.
`endtime`	This is the time, in milliseconds since Jan 1, 1970, when the activity completed.

As you can see, a history record is represented by exactly one row in the `repohistory` database table.

Now that we've had a look at the underlying persistent storage for connection definitions, let's explore how ManifoldCF actually manages them.

11.4.2 Connection definition usage and editing

In order to make it easy for components within ManifoldCF to use or change connection definition information, the ManifoldCF framework defines an internal component for each kind of connection definition. These components are the Output Connection Manager, the Authority Connection Manager, and the Repository Connection Manager. Each manager component is solely responsible for managing the data for all associated connection definitions.

The connection definition managers allow access and editing to connection definitions using a paper object model just as we've already described for authority groups. But since connection definitions are much more complex than authority groups, the paper object for a connection is correspondingly more complex.

The next topic will be the management of jobs.

11.5 Job definition management

When you create a job in ManifoldCF, it is saved persistently in the database. But what you may not consider is that there are really two aspects of a job: the *definition* of the job, and the *operational status* of the job.

The first kind of data structure, a job's definition, includes everything about a job that a user might directly set. This includes parameters such as the job's description, whether it's a continuous job or not, the job's schedule, document specification information, output specification information, etc. That's what this section is about.

11.5.1 Job definition database tables

The database tables that are used to store job definitions are summarized in table 11.12.

Table 11.12 Job definition database tables

Table	Purpose
`jobs`	This is the primary table for job definitions. It contains basic information about a job, such as its identifier and its description.
`schedules`	The `schedules` table has a list of schedule records for each job.
`jobhopfilters`	This table stores the maximum hop count values, per link type, for each job.

Let's look at each of these tables in turn.

THE JOBS TABLE

The `jobs` table contains information for both a job's definition **and** for a job's operational status. But we're not yet ready to discuss the operational status data, so in table 11.13 I've summarized only the definition-related columns in this table.

Table 11.13 The schema of the jobs database table, including only those columns relating to job definition

Column	Meaning

id	This column contains each job's unique numeric identifier.
description	A human-readable description for each job can be found in this column.
connectionname	This column contains the name of the repository connection definition each job uses.
outputname	This column contains the name of the output connection definition each job uses.
docspec	In this column, an XML document is kept, which is each job's document specification. The document specification form is determined by the kind of repository connection definition that the job uses.
outputspec	This column contains an output specification, which is an XML document, whose form depends on the output connection definition that each job uses.
startmethod	This column contains the start method specified for each job, which determines whether the job is started automatically by a schedule, or whether it must be started by hand.
priority	In this column, each job's priority value is stored, as an integer between 1 and 10.
hopcountmode	This column contains a enumerated value, specifying whether each job deletes unreachable documents from the job, or keeps unreachable documents for the time being, or retains unreachable documents forever.
type	This column contains the type specified for each job, which signals whether the job is a continuous one or a scan-once one.
intervaltime	For each continuous job, this column may contain an interval in milliseconds, which represents the minimum time between assessments of the same document. A blank value means an infinite interval.
maxintervaltime	For each continuous job, this column may contain an interval in milliseconds, which represents the maximum time between assessments of the same document. A blank value means no upper limit.
reseedinterval	For each continuous job, this column may contain an interval in milliseconds, which represents the minimum time between job seeding operations. A blank value means an infinite interval.
expirationtime	For each continuous job, this column may contain an interval in milliseconds, which is the minimum time from a document's origination until the time it should be expired. A blank value means an infinite interval.

As you can see, each `jobs` database table record corresponds to exactly one job. It's also not a great surprise that all of these column values and switches correspond directly to fields you have already seen in both the crawler UI and in the API service.

THE SCHEDULES TABLE

As we discussed at some length in Chapter 2, the way a schedule is set up for a job in ManifoldCF is through a set of schedule records. Each schedule record, as it appears in the crawler UI, describes some set of starting times, and an optional maximum run time.

These records are all stored pretty much in the way you would expect in the `schedules` database table. There is one database table record per schedule record. Table 11.14 describes the schema for this table.

Table 11.14 The schema of the schedules database table

Column	Meaning
`ownerid`	This is the numeric identifier of the job that owns each row.
`ordinal`	This column contains an integer which determines the ordering of the scheduling records that belong to each job.
`yearlist`	This column contains either a comma-separated list if year values, or "*" as a wildcard value.
`monthofyear`	This column contains either a comma-separated list of month values, starting at zero, or "*" as a wildcard value.
`dayofmonth`	This column contains either a comma-separated list of day-of-month values, starting at zero, or "*" as a wildcard value.
`dayofweek`	This column contains either a comma-separated list of day-of-the-week values, starting at zero (Sunday), or "*" as a wildcard value.
`hourofday`	Similar to dayofweek, except describes hour-of-day values, starting at zero (midnight).
`minutesofhour`	Similar to dayofweek, except describes minutes of the hour.
`timezone`	This column contains a string describing the timezone for each record.
`windowlength`	If not null, this column contains a numeric value in milliseconds describing the duration of each record's scheduled time window.

THE JOBHOPFILTERS TABLE

The schema of the `jobhopfilters` database table is described in table 11.14. As mentioned in table 11.8, this database table contains the limits on hop counts for each job.

We discussed how hop counts can be used to limit the documents belonging to a job in Chapter 2.

Table 11.14 The schema of the jobhopfilters database table

Column	Meaning
`ownerid`	This is the numeric identifier of the job that owns each row.
`linktype`	This column contains the type of link that has the limit. The types of links that exist are determined by the job's underlying repository connector.
`maxhops`	This column contains an integer, which describes the maximum number of hops from a seed of the specified link type that is allowed for a document that belongs to the owning job.

In this table, there is one record per job per type of link. Some jobs, therefore, will not have any rows in this table, because their underlying connector does not support any kinds of links at all.

11.5.2 Management of job definitions

Management of job definitions within ManifoldCF is fairly straightforward, and uses a paper object model that is similar in many ways to the model used to access or edit authority groups or various kinds of connections. The diagram is similar, but I've presented it regardless, if for no other reason than to establish the proper terminology. See figure 11.2.

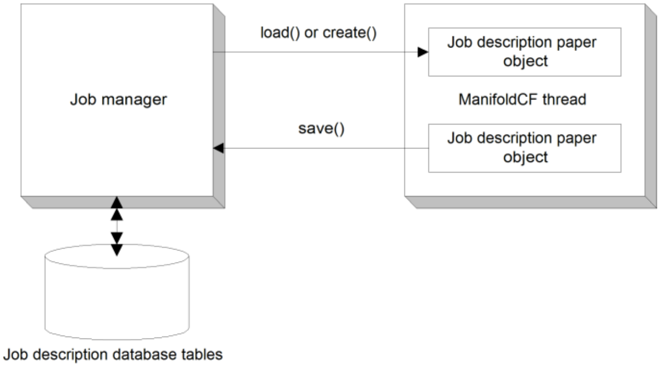
Job manager
load() or create()
Job description paper object
ManifoldCF thread
save()
Job description paper object
Job description database tables

Figure 11.2 The paper-object model is used to manage job definitions.

With the job definition data, we've made it to the end of the data structures that are manifestly visible when you define connections and jobs using ManifoldCF crawler UI and the ManifoldCF API Service. Now, the interesting part begins; we're going to describe data structures that we've only hinted at so far. These are the data structures that ManifoldCF actually uses to do crawling.

11.6 Job operational status management

As you may recall, there are two kinds of job-related data structures in ManifoldCF. The first kind has to do with job definitions, which we covered in the previous section. The second kind of job data is a job's operational status, which is the focus of this section. The operational status data structures include, among other things, the following:

- Each job's state
- When the last time document seeding occurred for the job
- The documents that belong to the job
- The links between the documents that belong to the job
- The minimum hop count from the seeds for each document
- Data that is carried down from a parent document to a child document
- Information controlling ordering of document fetches

11.6.1 Job operational status database tables

There are many database tables that contribute towards maintaining a job's operational status. These database tables are summarized in table 11.15.

Table 11.15 The database tables that maintain job operational status information

Table	**Purpose**
`jobs`	The `jobs` table includes the job's state, its start time, its end time, any errors that caused it to abort, and the last time a seeding request was made for the job.
`jobqueue`	This table contains all the documents, so far discovered, which belong to each job, including their states, their priority, and the earliest time they should be fetched.
`intrinsiclink`	The `intrinsiclink` table contains information about the direct relationships between documents that belong to a job. This data is used to calculate the hop count values, which are the distances from seed documents.
`hopcount`	This table records the hop counts for each link type for each document that belongs to every job.
`hopdeletedeps`	The `hopdeletedeps` table contains the dependencies between a link

	between documents and saved hop count values.
`carrydown`	This table keeps a record of connector-specific data that is saved when a parent document is processed, and is made available to the parent's child documents for their use.
`prereqevents`	For every document, this table contains the set of event names that will prevent the document from being processed.
`events`	This table contains the events that are currently active, and thus will block processing of the documents that depend on that event to be completed.

We'll now examine each of these database tables in some detail.

THE JOBS TABLE

We've already talked a bit about the schema of the jobs database table. This table has one row per job. Don't forget, though, that this table fulfills two duties – partly as a place for job definition data, which we've already explored, and partly as the residence of some very important job status information. Table 11.16 describes the part of the schema pertaining to the latter.

Table 11.16 The schema of the jobs table, including only columns that contain job status information

Column	**Meaning**
`id`	This column contains each job's unique numeric identifier, as already discussed.
`status`	In this column, you will find each job's status. The values this can have, and the transitions between them, will be explained in more detail in a subsequent subsection.
`processid`	When the above status represents a transient condition, this field contains the process identifier of the process in charge of completing the job's transition to a permanent status.
`starttime`	This column contains the time (in milliseconds since January 1, 1970) that each job was last started. If the job was never run, this column is blank.
`endtime`	If a job is not running, this column contains the time (in milliseconds since January 1, 1970) that the job last successfully completed. If the job was aborted, or has never been run, this column is blank.
`errortext`	If a job is not running, and was aborted due to error, this column contains the human-readable text of the error that caused the job to abort.
`windowend`	If this column is not empty, it means that the corresponding job was started

	on a schedule, and the value in the column is the time (in milliseconds since January 1, 1970) that the job execution window ends.
`lastchecktime`	Every time a seeding operation is done, starting and ending times are sent to the connector method which is required to return changed documents. This column contains the milliseconds since January 1, 1970 of the last time a seeding operation was performed on the corresponding job. If no such operation was ever performed, then this column contains a value of zero.
`lasttime`	If a job is on a schedule, this column contains the time (in milliseconds since January 1, 1970) that ManifoldCF should start looking at for the next available scheduling window. The value in this column is updated when the job is completed or is aborted, either manually or due to an error.
`reseedtime`	For an adaptive job, this is the time it will next be reseeded.
`failtime`	If any job-related activity resulted in a `ServiceInterruption` being thrown, this field may contain the time, in milliseconds since January 1, 1970, when the job will abort if repeated `ServiceInterruptions` are thrown.
`failcount`	If any job-related activity resulted in a `ServiceInterruption` being thrown, this field may contain the number of remaining times that a `ServiceInterruption` can occur before the job will be aborted.

As you can see, ManifoldCF keeps everything it needs to start and run jobs in the database. This permits it to recover very gracefully should the agents process be stopped or aborted.

THE JOBQUEUE TABLE

Now, finally, we are ready to describe the database table which is really at the core of ManifoldCF's crawling ability – the `jobqueue` table. What does this table do, and why is it so important?

Simply put, the `jobqueue` table is the data structure that functions as the document queue for ManifoldCF. One consistent feature of all crawlers, regardless of technology or model, is a document queue. Documents are placed on the queue either during *seeding*, or as they are *discovered*. As-yet unprocessed documents are taken off the queue and processed, one at a time.

As you might expect, it is the performance of the document queue that determines the performance of the overall crawler. For this reason, many other crawlers adopt special custom data structures for their document queues. Indeed, ManifoldCF is the only crawler I am aware of today that uses a SQL database for this purpose.

As we discussed in Chapter 1, ManifoldCF's philosophy with regards to performance is that performance is usually limited by external factors; reliability and living within a memory constraint are therefore of more paramount importance than raw speed. Nevertheless, a poor choice of data structure for the document queue could still have crippled ManifoldCF. Therefore, great care was expended in making sure that the `jobqueue` table would perform well given the tasks it needed to perform.

It turns out that the essential `jobqueue` query that must run quickly (and also be bounded in memory consumption) is the one that locates documents that are ready for processing. This query has unique challenges, because there's a decent chance that most of the documents in a job's document queue would be returned from this query, and those could well number in the millions. In addition, getting the document ordering right is crucial – but in no way would it be acceptable to read all the eligible document records into memory at once and only afterwards sort them.

In order for this all to work, ManifoldCF relies on two key database features that can be found in many SQL databases (but by no means all of them). The first feature is the ability to perform a *limit* query. This is a query that returns only the first *N* records matching a search criterion, and does not return the rest. This cannot mean, either, that the database loads all the matching rows into memory and then just returns the first ones, because that would solve nothing; the problem of memory overconsumption would appear in the database instead of in ManifoldCF, but it would still appear nevertheless. Thus, some cleverness must exist to actually load only the rows that are needed. The second feature, which is of critical importance, involves the ability to read rows from a database resultset *in index order*. That is, when the results of a query are ordered by a column that has an index built on it, the database reads the index directly, rather than sort the results in memory.

Taken together, it is possible with these two features to construct a query that can return some fixed number of documents, all of which have the "right stuff", and send them off to be processed, without causing undue delay or an out of memory condition. With all this in mind, table 11.17 describes the schema of the `jobqueue` database table. Each record in the table represents a single document which belongs to a single job. Documents that appear in more than one job must therefore have more than one record in the `jobqueue` table; they are still considered to represent the same document if their jobs share the same underlying repository connection definition, and their document identifiers are the same.

Table 11.17 The schema of the jobqueue database table

Column	Meaning
`id`	This column contains a unique numeric identifier for each document for each job.
`jobid`	The `jobid` column contains the numeric identifier of the job that owns each document.
`docid`	This column contains an unlimited-length string that identifies each document in the context of the repository connection definition from which it came. From the framework perspective, it is immaterial what the docid column actually

	contains; it is only the repository connector that knows or cares.
`dochash`	The `dochash` column is the SHA hash of the `docid` column, and is actually used as the unique identifier for each document, within all jobs that share the same repository connection definition.
`status`	This column contains a single character representing a document's state. Status values include representations for "pending", "active", and "done", among many others. These will be described in a later subsection.
`processid`	If the status above is a transient status, this field will contain the process identifier of the process in charge of transitioning this document record to a permanent status.
`seedingprocessid`	If the status above is a transient seeding status, then this field will contain the process identifier of the process in charge of completing the seeding and leaving the seeding state.
`docpriority`	The `docpriority` column contains a floating point number upon which an index has been created, and serves to order a job's documents in such a way as to process documents in the order that achieves the maximum document throughput over time, given the throttling in effect. A later subsection will describe how document priority is assigned.
`priorityset`	This column contains a time, in milliseconds since January 1, 1970, that is the time when the document's priority was assigned. This is used to control when a document's priority is reset, which is done if a document is determined to be blocking the queue, and its last reprioritization was earlier than some point in the past.
`isseed`	The `isseed` column contains a single character enumerated value which describes whether the corresponding document is not a seed document, was formerly a seed document, or is a new seed document. Whether a document is a seed document is up to the repository connection definition used by the corresponding job.
`checktime`	This column contains the time, in milliseconds since January 1, 1970, that is the earliest time the corresponding document may be processed. This column is used to control the average document fetch rate, as part of throttling, and also as part of ManifoldCF's fetch error handling strategy. If empty, then the document may be processed at any time.
`checkaction`	This column contains a single-character representation of the action that should be undertaken when the document is next processed. The value can either represent the actions "fetch" or "expire".

failcount	The failcount column contains a number describing the total remaining times a fetch of the corresponding document may subsequently occur before the fetch is considered to have failed. If this column is empty, there is no limit.
failtime	The failtime column contains a time (in milliseconds since January 1, 1970) that is the time limit for fetch attempts before the fetch is considered to have failed. If this column is empty, there is no limit.

There is a lot in this table to digest and understand – and the descriptions above beg more questions sometimes than they seem to answer. But please have patience – the goal of this chapter is only to explore the data structures and their management. I will be covering the most important algorithms involving this table in Chapter 12. For now, though, let's try to stay focused, and continue with our description of the job operational status data structures.

THE INTRINSICLINK TABLE

The purpose of the intrinsiclink database table is to keep track of the relationships between documents in each job. This is necessary if there is a possibility of filtering documents by hop count. As you may recall from Chapter 2, the ability to filter documents in this way is a function of each job's repository connector; only specific connectors actually support this functionality. As of this writing, the only included connectors that deal with hop counts are the web connector and the file system connector.

Each row in the intrinsiclink database table represents a relationship of a certain type between a parent document (which is the owner of the relationship), and a child document (which is the document the parent document references in some way). The types of references that are possible are determined by the underlying connector, and are called *link types*. Thus, a connector that supports no link types cannot have any associated rows in the intrinsic link database table.

The schema of the intrinsiclink database table is described in table 11.18.

Table 11.18 The schema of the intrinsiclink database table.

Column	**Meaning**
jobid	This column contains the numeric identifier of the job that owns the row.
parentidhash	The parentidhash column contains the SHA hash of the parent document identifier.
childidhash	The childidhash column contains the SHA hash of the child document identifier.
linktype	This column contains the type of link the row represents.
isnew	The isnew column is used to mark rows as being newly added, and not yet fully

	reflected in other tables (such as `hopcount` and `hopdeletedeps`).

THE HOPCOUNT TABLE

The `hopcount` database table maintains a cache of the calculated hop counts for each link type for each document for each job. Each row in the table corresponds to a single link type for a single document in a single job. If you want to see the schema, it is described in table 11.19.

Table 11.19 The schema of the hopcount database table.

Column	Meaning
`id`	This column is a unique numeric identifier for the database table row.
`jobid`	The `jobid` column is the numeric identifier of the job that owns each database row.
`parentidhash`	This column is the SHA hash value of the document identifier that this database row pertains to.
`linktype`	The `linktype` column is a string which describes what type of link the cached distance applies to. The types of links are dependent on the underlying repository connector that the job uses.
`deathmark`	This column contains a single-character value which represents three different states for the row. The states are “queued”, meaning that the distance is required but has not yet been calculated, “normal”, meaning that the actual distance is indeed what this record reflects, and “deleting”, meaning that the record is invalid and will be destroyed unless it is re-queued for evaluation before that happens.
`distance`	This is the numeric value representing the shortest distance from a seed document to the current document for the link type in question. For a seed document, the distance recorded will be 0.

THE HOPDELETEDEPS TABLE

The `hopdeletedeps` database table keeps track of the links that affect specific `hopcount` table rows. In other words, the purpose of the `hopdeletedeps` table is to figure out what cached `hopcount` table rows must be removed should a link between any parent document and child document that was previously recorded in the `intrinsiclinks` table be removed.

The schema of the hopdeletedeps table is described in table 11.20.

Table 11.20 The schema of the hopdeletedeps table

Column	Meaning
`ownerid`	This is the id of the row in the `hopcount` database table that the record belongs to.
`jobid`	This column contains the numeric identifier of the job that the record belongs to.
`linktype`	The `linktype` column contains the type of link upon which the hopcount row depends. Together, the `linktype`, `parentidhash`, and `childidhash` columns contain the complete description of the link.
`parentidhash`	This column contains the SHA hash of the parent document identifier, which is a part of the link description.
`childidhash`	This column contains the SHA hash of the child document identifier, which is part of the link description.

THE CARRYDOWN TABLE

Some of the job-related database tables in ManifoldCF exist to provide direct support to connector implementations. The `carrydown` table is one such. Its purpose is to provide a general means of passing data from parent documents to child documents within the same job, and its functionality is exposed to connectors by means of an internal java interface that appropriate repository connector methods have access to. We have already discussed repository connector methods that deal with carry-down information in Chapter 7. As a real-world example, the RSS connector uses the carry-down feature to transmit title, publication date, and source feed information from feed documents, which contain descriptions and document URLs, to their children, which are the documents that are actually indexed.

Only certain connectors will ever need to make use of this functionality. But where it is needed, it is extremely useful and powerful. Table 11.21 describes the schema of this table.

Table 11.21 Schema of the carrydown database table

Column	Meaning
`jobid`	This column contains the numeric identifier of the job that the carrydown row belongs to.
`parentidhash`	The `parentidhash` column contains the SHA hash of the parent document identifier of the carry-down datum.
`childidhash`	The `childidhash` column contains the SHA hash of the child document identifier of the carry-down datum. Each piece of carry-down data is specific to a particular child document.
`dataname`	This column contains the name of the datum, of which the corresponding data

	value is a portion.
`datavalue`	This column contains an unlimited-length value, which is one of the (possibly multiple) values for the datum.
`datavaluehash`	The `datavaluehash` column contains the SHA hash of the `datavalue` column, and is used to determine uniqueness. If by chance more than one data value for a given datum has the same hash value, only one of them will be carried down.
`isnew`	This column is used to help ManifoldCF figure out what rows represent new values and deleted values after an update. The values in the column represent a “base” state, a “new” state, and an “existing” state.

Each chunk of carry-down information is called a *datum*, and it has a name (which is provided by the connector code when it uses the carry-down service), via which it is both saved and later accessed. Since each datum can be multi-valued, one row in the `carrydown` table potentially corresponds to only one piece of a complete carry-down datum. Furthermore, each data value must be unique, given the job, parent document, child document, and datum name.

THE PREREQEVENTS TABLE

In some circumstances, the fetch of a document must be prevented until the proper conditions are in place to permit it to be fetched correctly. A prime example of this need is when a web crawler accesses a site which is secured by *session-based authentication*.

Session-based authentication uses a cookie stored by your browser to control access to content. The way the cookie is typically set up involves visiting a series of pages, which we will call *login pages*, where a registered user can supply the appropriate credentials. Only when this process is completed can the user see the protected content.

What this means for ManifoldCF is that it is possible, with the very same URL, to fetch either the correct content, or data that is not interesting. A crawler will not get the proper content unless or until the login sequence as been completed. It’s also possible for sessions to expire, which then requires that the login sequence be repeated. With either of these situations, the ManifoldCF web connector must not try to fetch protected content while there is a login sequence in progress.

Just as in the case of carry-down data, ManifoldCF provides a service to its connectors which is intended to permit them to restrict the processing of documents if the wrong conditions exist. These situations are labeled *events* in ManifoldCF parlance. An event is *asserted*, and when it is, all documents that are listed as being dependent on that event are temporarily blocked from being processed. When the event is over, the restriction is removed, and the blocked documents then become available for processing once again.

Tying this back to a real-world example, the start of a site's login sequence would constitute the beginning of an event. The event is given a name: "Enter:mysite.com", for instance. Prior to the even taking starting, all documents which require that the login sequence be completed before they are processed should have been added to the document queue including a dependency on that same event. These documents will not be considered for processing so long as the event is in progress. Once the event is completed, the documents that were off-limits suddenly are available for processing once again.

The `prereqevents` database table is used to record the event dependencies, if any, for all documents in the `jobqueue` database table. See table 11.22 for a description of its schema.

Table 11.22 The schema of the prereqevents database table

Column	Meaning
`owner`	This column contains the numeric identifier of the `jobqueue` database table row that owns the event dependency.
`eventname`	The eventname column contains a string that is the name of the event that the `jobqueue` database table row depends on.

Each row in the `prereqevents` table describes exactly one event dependency for one document (as described by a `jobqueue` table row).

THE EVENTS TABLE

The `events` table is the place where events that are currently active are noted. Each row represents an event that is in play; when the event is complete, the corresponding row will be deleted from the table. Table 11.23 describes the schema of this table.

Table 11.23 The schema of the events database table

Column	Meaning
`name`	The name column is the name of an event that is currently in play.

EVENT SCOPE

You may wonder why the schema of the `events` table does not include a job identifier or other means of binding the event to some other entity within ManifoldCF. But, which entity should it be bound to? A good case could be made to make events be job-specific, but an equally good case could be made to have every event be specific to a repository connection definition. Because there is no clear specificity, ManifoldCF is designed to permit flexibility in this regard. It presumes the name of the event will be appropriately qualified (with a prefix) in a manner consistent with its intended scope. The qualification of the event name is handled by the framework.

It is therefore possible to have events whose scope is global, events whose scope is repository connection definition specific, and events which are job specific. The choice of scope is up to the connector designer, and depends completely on the connector's intended use.

With the description of the events database table, we've finally covered all of the core database tables that participate in making a job run in ManifoldCF.

11.6.2 Job states

The `jobs` database table has a column called `status`. As we have seen, this column contains critical information about the state of a job. In Chapter 12, we'll learn more about the transitions between them, and which threads perform those transitions. First, though, let's explore what these job states are, and what they mean.

> **Note** In chapters 2 and 3, we've already seen job states as they are displayed in the Crawler UI, and as they are returned by the API Service. The actual underlying states within the jobs table, however, are much richer than what we've seen so far. Many of the states are collapsed together because they provide no useful insights to a user of ManifoldCF.

Table 11.24 summarizes the full complement of job states and their meanings.

Table 11.24 Job states and their meanings

State	Meaning
`INACTIVE`	The job is not running.
`ACTIVE`	The job is active, and is within a valid scheduling window.
`PAUSED`	The job is within a valid scheduling window, but has been paused manually.
`PAUSING`	The job has been paused but has not yet fully stopped running.
`ACTIVEWAITING`	The job is active but is leaving a valid scheduling window.
`PAUSINGWAITING`	The job was pausing and is also leaving a valid scheduling window.

`RESUMING`	The job has been resumed from a pause or a wait, but has not yet started running again.
`SHUTTINGDOWN`	The job has finished crawling, and is in the cleanup of unreachable documents phase.
`ACTIVEWAIT`	The job is active, but is not within a valid scheduling window.
`PAUSEDWAIT`	The job has been manually paused, and is also not within a valid scheduling window.
`ABORTING`	The job is in the process of aborting, either due to error or because of a manual abort.
`STARTINGUP`	The job is preparing to start, by setting all document states back to a starting condition, and by performing the initial seeding.
`STARTINGUPMINIMAL`	A minimal job run is preparing to start.
`DELETESTARTINGUP`	The job is preparing to delete, by setting all document states to an appropriate starting condition.
`ABORTINGSTARTINGUP`	The job is aborting from a STARTINGUP state, either due to error or due to manual intervention.
`READYFORSTARTUP`	Due to scheduling or due to manual intervention, the job is now ready to go through the start up sequence.
`READYFORSTARTUPMINIMAL`	Due to scheduling or manual intervention, the job is ready to now to go through the minimal job start up sequence.
`READYFORDELETE`	The job has been manually deleted, and the job is ready to begin the job delete sequence.
`ACTIVESEEDING`	The job is running, and is in a valid schedule window, and a seeding cycle is also underway.
`PAUSINGSEEDING`	This state is like PAUSING but with seeding going on as well.
`ACTIVEWAITINGSEEDING`	This state is like ACTIVEWAITING but with seeding going on as well.
`PAUSINGWAITINGSEEDING`	This state is like PAUSINGWAITING but with

	seeding going on as well.
`RESUMINGSEEDING`	This state is like RESUMING but with seeding going on as well.
`ABORTINGSEEDING`	The job is aborting, and a seeding cycle is also underway.
`PAUSEDSEEDING`	The job has been paused, and is in a valid schedule window, and a seeding cycle is also underway.
`ACTIVEWAITSEEDING`	The job is running, but is not in a valid schedule window, and a seeding cycle is underway.
`PAUSEDWAITSEEDING`	The job is paused, but is not in a valid schedule window, and a seeding cycle is also underway.
`ABORTINGFORRESTART`	The job has been aborted, and will restart when the abort is complete.
`ABORTINGFORRESTARTMINIMAL`	The job has been aborted, and will undergo a minimal restart when the abort is complete.
`ABORTINGFORRESTARTSEEDING`	The job has been aborted for restart, and a seeding cycle is also underway.
`ABORTINGFORRESTARTSEEDINGMINIMAL`	The job has been aborted for restart, and will be restarting as a minimal job, and a seeding cycle is also underway.
`ABORTINGSTARTINGUPFORRESTART`	The job has been aborted for restart while in the STARTINGUP state.
`ABORTINGSTARTINGUPFORRESTARTMINIMAL`	The job has been aborted for minimal restart while in the STARTINGUP state.
`READYFORNOTIFY`	The job is ready for the output connector to be notified that the job completed.
`NOTIFYINGOFCOMPLETION`	The output connector is being notified that the job is completed.
`ACTIVE_UNINSTALLED`	The job is active, but the associated repository connector is not installed.
`ACTIVESEEDING_UNINSTALLED`	The job is active and is being seeded, but the associated repository connector is not installed.

`ACTIVE_NOOUTPUT`	The job is active, but the output connector is uninstalled.
`ACTIVESEEDING_NOOUTPUT`	The job is active and is being seeded, but the output connector is uninstalled.
`ACTIVE_NEITHER`	The job is active, but neither the repository connector nor the output connector is installed.
`ACTIVESEEDING_NEITHER`	The job is active, and is being seeded, but neither the repository connector nor the output connector is installed.
`READYFORDELETE_NOOUTPUT`	The job is ready to have its documents deleted from the index, but the output connector is uninstalled.

While this state description is helpful on some level, it's a bit hard to see at this point why, exactly, all these states must exist. But this will become much clearer when we describe what each crawler thread does, and how ManifoldCF avoids inconsistent states. We'll cover that in great depth in Chapter 12.

11.6.3 The document identifier

In Chapter 7, we briefly described what a document identifier is. At that time, I described this as a connection-type-specific value, which is used to uniquely describe a document. But there is a lot more to it than that.

DOCUMENT IDENTIFIER SCOPE

A document identifier is both created and interpreted by a specific repository connection definition – the one that underlies the job that owns the `jobqueue` table record where the identifier is found. However, since the same repository connection definition can be used for multiple jobs, you might expect that this identifier would be the same when used to describe the same document in different jobs. Indeed, ManifoldCF expects this to be the case, and includes special logic to handle the same document in multiple jobs in a reasonable way – and it uses the document identifier and repository connection definition name as the means of determining sameness.

THE DOCUMENT IDENTIFIER HASH VALUE

Actually, what I have just told you is not completely true. ManifoldCF actually uses the SHA hash of the document identifier to determine sameness, rather than the identifier itself. This is necessary because it is not possible to create indexes in most databases on a column whose data can be of unlimited length. Neither PostgreSQL nor Apache Derby supports such functionality; for PostgreSQL, attempting to index a column value greater than a certain rather obscure length throws a database exception.

So, what are the implications of using a hash value, rather than the thing itself, as a unique database key? First, recognize that the chances of a collision are remote. Secure hashes like SHA are designed specifically to make the chance of two strings having the same hash value be vanishingly small. But even so, it's bad design practice to build software that has a known non-zero chance of failing utterly, so ManifoldCF **always** treats the document identifier hash as the actual key, for every table, and for every internal data structure. Thus, if two documents indeed wind up having different identifiers but the same hash value, the document that "wins" is the first one that was discovered – and thus one of them will never be indexed, or considered to be part of the job's documents at all.

DOCUMENT IDENTIFIER INTERPRETATION

We've talked about how the document identifier's form is opaque to the ManifoldCF framework. But that's not very helpful – what kinds of form should we expect?

The whole purpose of having a document identifier at all is to allow the connector software to efficiently find out certain details about the document that the identifier describes. So, for a web or RSS connection definition, we'd expect that the document identifier would be a URL. For a repository such as Documentum, on the other hand, the document identifier will be whatever it is that Documentum uses as a document identifier.

11.6.4 Document states

Just as jobs have states, so do individual documents, as we have seen in table 11.17. These states keep track of where the document actually is in its lifecycle. Table 11.25 describes these states.

Table 11.25 The states that a document can have in the jobqueue database table

State	**Meaning**
`PENDING`	The document has never been processed, and is eligible to be processed for the first time.
`ACTIVE`	The document has been handed to a worker thread for processing.
`COMPLETE`	The document has been processed, and no further work is needed on it.
`UNCHANGED`	The document has not yet been processed on this job run, but was completed on a previous job run.
`PENDINGPURGATORY`	The document was processed in one of the previous job runs, and has been discovered in the current job run, but has not yet been processed.

`ACTIVEPURGATORY`	The document was processed in one of the previous job runs, and has been discovered in the current job run, and has been handed to a worker thread for processing.
`PURGATORY`	The document was processed in one of the previous job runs, but has not been discovered yet in the current job run.
`ELIGIBLEFORDELETE`	The job is in the DELETING state, and the document is waiting to be handed to a delete thread for removal.
`BEINGDELETED`	The document has been handed to a delete thread for removal.
`BEINGCLEANED`	The job is in the SHUTTINGDOWN state and the document is being cleaned up.
`ACTIVENEEDRESCAN`	The document was being processed by a worker thread in the ACTIVE state, but during that time carry-down data changed, so the document needs to be processed again before the job can be completed.
`ACTIVENEEDRESCANPURGATORY`	The document was being processed by a worker thread in the ACTIVEPURGATORY state, but during that time carry-down data changed, so the document needs to be processed again before the job can be completed.
`HOPCOUNTREMOVED`	The document has been excluded from the crawl because its hop count is too large, but it remains in the `jobqueue` table so that the hop distance to it is not lost.

Once again, this description only hints at the reasons and mechanism behind document state changes. But don't worry - we will be talking a lot more about such things in Chapter 12.

We're now done discussing the job operational status data structures! There's only one major data structure left: the data structure that coordinates incremental indexing.

11.7 Incremental indexing

We've looked in some detail now at most of the data structures that ManifoldCF uses to make crawling work. Incremental indexing, though, requires structures and data of its own. Specifically, we need somehow to keep track of what documents were indexed, so we know whether we'll need to index them again or not.

In some places, a hint of what's needed for incremental indexing has already appeared. Recall that in table 11.16, the `lastchecktime` column of the `jobs` database table contained the last time the repository connector's seeding operation was called, so that each seeding operation could behave in as incremental a way as possible. That's potentially helpful, but it isn't enough, because not all repositories support that kind of functionality.

So, in addition, ManifoldCF also keeps track of the last indexing operation that was performed for every document, along with some statistics about how often that document has changed in the past.

11.7.1 Incremental indexing database tables

The database schema that supports incremental indexing is fairly straightforward. There is one database table, called `ingeststatus`, whose purpose is to track this information. This table has a record for each repository connection definition and document identifier. Table 11.26 describes the schema.

Table 11.26 The schema of the ingeststatus database table

Column	**Meaning**
`id`	This column contains a unique numeric identifier.
`dockey`	The `dockey` column contains an amalgamation of the "identifier class" and the "identifier hash". The interpretation of these values is theoretically dependent on the agent that submitted the indexing request for the document. For the pull agent, the identifier class is the name of the repository connection definition, and the identifier hash is the document identifier hash value.
`docuri`	This column contains the URI for the document, or is empty if the document is not currently indexed. The URI is the considered to be the document identifier within the target search engine.
`urihash`	The `urihash` column contains the SHA hash of the value in the `docuri` column.
`connectionname`	This column contains the name of the output connection definition that was used to index the document.
`authorityname`	The `authorityname` column contains the name of the authority group that has been selected to secure the document.
`lastversion`	This is the version string that was calculated for the document by the repository connector.
`lastoutputversion`	This column contains the version string that was calculated for the document by the output connector.
`firstingest`	This column contains the time (in milliseconds since January 1, 1970) of the first index operation that was performed on the document.
`lastingest`	This column contains the time (in milliseconds since January 1, 1970) of the

	most recent index operation that was performed on the document.
changecount	The changecount column contains the number of changes to the document that took place since the first index operation.

That's it! Next, we'll look in detail into some of the information stored in this table.

11.7.2 Document URIs

The docuri field of the ingeststatus database table deserves a more extended explanation. For every document, the repository connector can generate a URI that is meant to permit access to the document. It is this URI that is saved.

Document URI scope

In the incremental ingestion manager, ManifoldCF presumes that this URI is used as the primary key for the document in the search engine. That means that whenever a new index request comes along that has the same URI as an earlier request, also with the same output connection definition, then no matter where it came from, even if it came from a different repository connection definition entirely, it will replace the existing index entry. Thus, the scope of the document URI is bound solely to the output connection definition.

The document URI hash value

Because the document URI is effectively the primary key of the document, we once again find ourselves needing to create a database index based on this key. But, as we have discussed, it is not possible to reliably create such database indexes on values that can be unlimited in length. (URIs, although technically limited to 4096 bytes, still can exceed the size cutoff for being indexed in PostgreSQL.)

We therefore use the same trick we used for dealing with the unlimited-length document identifier: we take the SHA hash value of it, and use that as the primary key instead. Thus, there is a remote possibility that two documents that have different URIs might inadvertently collide. A collision in this case effectively means that only one of these two documents can be indexed at the same time, and whichever document was indexed last is the one that will be kept.

11.7.3 Versioning information

You may have noticed that there are two version strings that appear in the ingeststatus table. One of them is stored in the lastversion column, and the other is stored in the lastoutputversion column. What are these strings, anyway?

As far as the incremental index manager is concerned, these strings are completely opaque; it does not know what is inside them, and doesn't care. They are strictly used for comparison, to compare one version of the document against another.

There are **two** strings because there are two sources of information that affect how a document is indexed. There is information that comes from the repository connection – often, from the repository itself, but also potentially from the document specification included in the job, or from the configuration information used to configure the connection. Then, there is information that affects indexing that comes by way of the output connection – which, in all likelihood, comes from the job's output specification, or the output connection's configuration information.

The versioning information stored in the `ingeststatus` table is used to make decisions about whether a document needs to be processed again, or not. As we've discussed, it is a connector's responsibility to form each version string, but it is the ManifoldCF framework that uses the old and new version strings to make a determination as to how to proceed.

11.7.4 Update frequency information

You may have noticed that the `ingeststatus` database table keeps track of how often a document is updated, over what period of time. This has a very specific purpose.

On large, continuous crawls, ManifoldCF may periodically refetch a document, if it is configured to do so. This sounds like a good idea, but there are serious problems with this approach.

Imagine now what will happen if all documents are refetched on a fixed schedule – say, once a day. On the first day of such a crawl there is no problem – the crawler happily chugs along and brings in nothing but new documents at a reasonable rate. But you would notice that at the beginning of the second day crawling progress comes to a halt; little additional new content gets crawled.

Why does this happen? The crawler ceases to make obvious progress because it must in fact repeat itself. All the documents it fetched before must be fetched again, for no other reason than to determine if any changes occurred in them.

What can be done to fix this problem? One thing we could do is to change the minimum refetch interval to be large enough so that the crawler only begins to repeat itself after all the documents have been indexed. But using that approach involves knowing how long it will take to fetch all the documents in advance, at least roughly – which is a very hard thing to know, if you are crawling the web. So, that approach has limited utility. But what else can we do?

The answer is that we can dynamically adjust the interval between checks of the document, based on the frequency with which the document changes. Documents that never change will therefore be checked less and less often, while documents that seem to change a lot get checked much more frequently.

Now you can see what the update frequency information stored in the `ingeststatus` table is for. It is there to allow for the dynamic adjustment to the refetch schedule for a document. ManifoldCF actually implements an exponential backoff algorithm based on the change frequency, starting with the minimum time interval for rescan, which you can set independently for every job.

11.8 Summary

Congratulations, you've finally made it through the most technically demanding chapter of this book! I hope you've learned something about how ManifoldCF manages data in general, and also a great deal about how database tables are employed both to keep track of definitional information, as well as operational status information. We've looked a bit into how ManifoldCF internally manages connection definitions and jobs, and how it pools connector class instances.

In the next chapter, we'll go even deeper, and describe the threads and algorithms that ManifoldCF uses to perform crawls. We'll look hard at how ManifoldCF schedules documents for fetching, how it transitions documents and jobs between states, and how it handles continuous crawling.