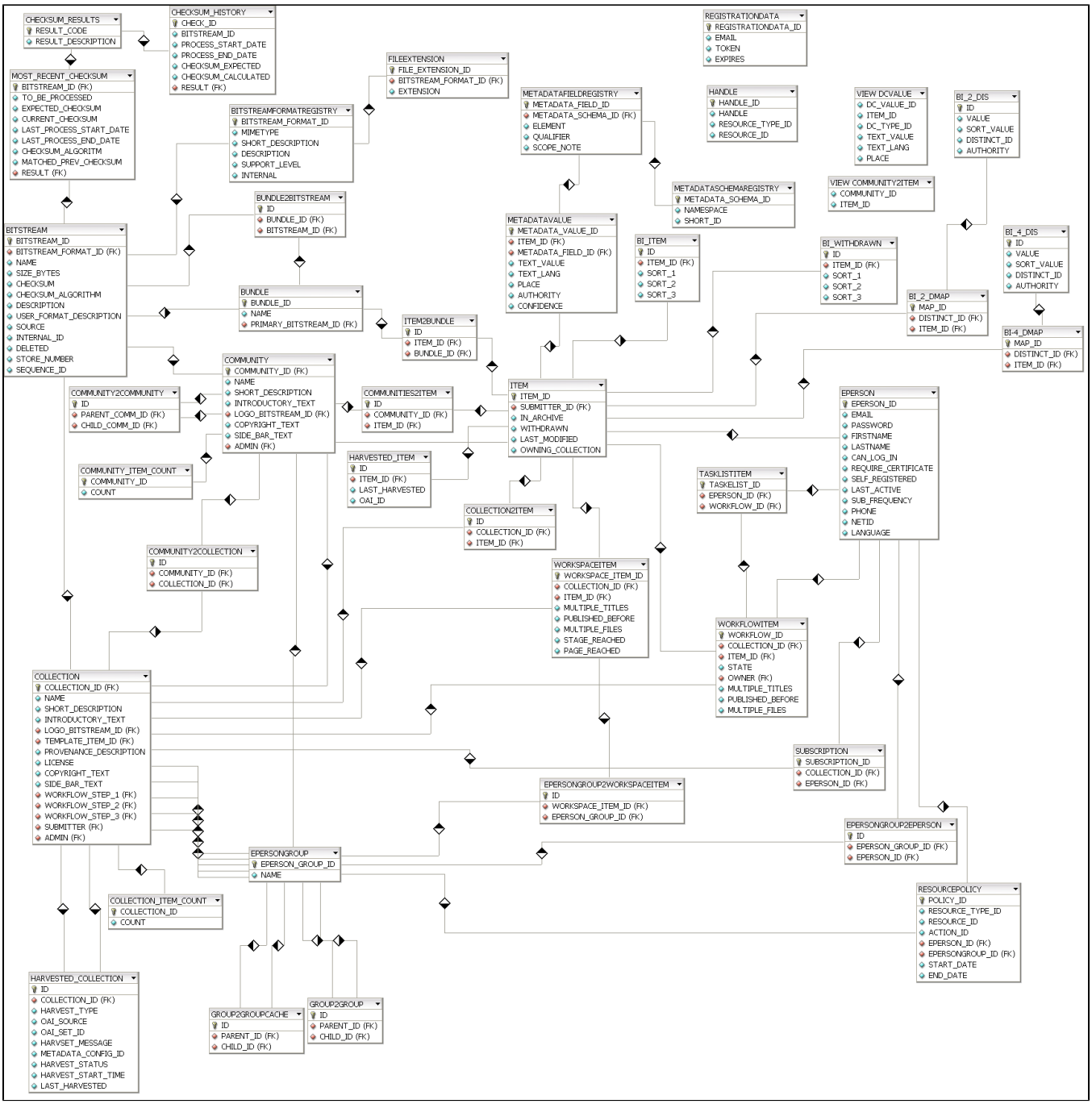
Storage Layer

In this section, we explain the storage layer: the database structure, maintenance, and the bistream store and configurations.

RDBMS / Database Structure

DSpace uses a relational database to store all information about the organization of content, metadata about the content, information about e-people and authorization, and the state of currently-running workflows. The DSpace system also uses the relational database in order to maintain indices that users can browse.



Most of the functionality that DSpace uses can be offered by any standard SQL database that supports transactions. Presently, the browse indices use some features specific to PostgreSQL and Oracle, so some modification to the code would be needed before DSpace would function fully with an alternative database back-end.

The *org.dspace.storage.rdbms* package provides access to an SQL database in a somewhat simpler form than using JDBC directly. The main class is *DataseManager*, which executes SQL queries and returns *TableRow* or *TableRowIterator* objects. The *InitializeDatabase* class is used to load SQL into the database via JDBC, for example to set up the schema.

All calls to the *Database Manager* require a DSpace *Context* object. Example use of the database manager API is given in the *org.dspace.storage.rdbms* package Javadoc.

The database schema used by DSpace is created by SQL statements stored in a directory specific to each supported RDBMS platform:

- PostgreSQL schemas are in *[dspace-source]/dspace/etc/postgres/*
- Oracle schemas are in *[dspace-source]/dspace/etc/oracle/*
  The SQL (DDL) statements to create the tables for the current release, starting with an empty database, aer in *database_schema.sql*. The schema SQL file also creates the two required e-person groups (*Anonymous* and *Administrator*) that are required for the system to function properly.

Also in *[dspace-source]/dspace/etc/[database]* are various SQL files called *database_schema_1x_1y*. These contain the necessary SQL commands to update a live DSpace database from version 1.*x* to 1.*y*. Note that this might not be the only part of an upgrade process: see Updating a DSpace Installation for details.

The DSpace database code uses an SQL function *getnextid* to assign primary keys to newly created rows. This SQL function must be safe to use if several JVMs are accessing the database at once; for example, the Web UI might be creating new rows in the database at the same time as the batch item importer. The PostgreSQL-specific implementation of the method uses *SEQUENCES* for each table in order to create new IDs. If an alternative database backend were to be used, the implementation of *getnextid* could be updated to operate with that specific DBMS.

The *etc* directory in the source distribution contains two further SQL files. *clean-database.sql* contains the SQL necessary to completely clean out the database, so use with caution! The Ant target *clean_database* can be used to execute this. *update-sequences.sql* contains SQL to reset the primary key generation sequences to appropriate values. You'd need to do this if, for example, you're restoring a backup database dump which creates rows with specific primary keys already defined. In such a case, the sequences would allocate primary keys that were already used.

Versions of the ***.sql*** files for Oracle are stored in *[dspace-source]/dspace/etc/oracle*. These need to be copied over their PostgreSQL counterparts in *[dspace-source]/dspace/etc* prior to installation.

Maintenance and Backup

When using PostgreSQL, it's a good idea to perform regular 'vacuuming' of the database to optimize performance. This is performed by the *vacuumdb* command which can be executed via a 'cron' job, for example by putting this in the system *crontab*:

```
# clean up the database nightly
40 2 * * * /usr/local/pgsql/bin/vacuumdb --analyze dspace > /dev/null 2>&1
```

The DSpace database can be backed up and restored using usual methods, for example with *pg_dump* and *psql*. However when restoring a database, you will need to perform these additional steps:

- The *fresh_install* target loads up the initial contents of the Dublin Core type and bitstream format registries, as well as two entries in the *epersongroup* table for the system anonymous and administrator groups. Before you restore a raw backup of your database you will need to remove these, since they will already exist in your backup, possibly having been modified. For example, use:

  ```
  DELETE FROM dctyperegistry;
  DELETE FROM bitstreamformatregistry;
  DELETE FROM epersongroup;
  ```

- After restoring a backup, you will need to reset the primary key generation sequences so that they do not produce already-used primary keys. Do this by executing the SQL in *[dspace-source]/dspace/etc/update-sequences.sql*, for example with:

  ```
  psql -U dspace -f  [dspace-source]/dspace/etc/update-sequences.sql
  ```

  Future updates of DSpace may involve minor changes to the database schema. Specific instructions on how to update the schema whilst keeping live data will be included. The current schema also contains a few currently unused database columns, to be used for extra functionality in future releases. These unused columns have been added in advance to minimize the effort required to upgrade.

Configuring the RDBMS Component

The database manager is configured with the following properties in *dspace.cfg*:

db.url	The JDBC URL to use for accessing the database. This should not point to a connection pool, since DSpace already implements a connection pool.
db.driver	JDBC driver class name. Since presently, DSpace uses PostgreSQL-specific features, this should be *org.postgresql.Driver*.
db. username	Username to use when accessing the database.
db. password	Corresponding password ot use when accessing the database.

Bitstream Store

DSpace offers two means for storing content. The first is in the file system on the server. The second is using SRB (Storage Resource Broker). Both are achieved using a simple, lightweight API.

SRB is purely an option but may be used in lieu of the server's file system or in addition to the file system. Without going into a full description, SRB is a very robust, sophisticated storage manager that offers essentially unlimited storage and straightforward means to replicate (in simple terms, backup) the content on other local or remote storage resources.

The terms "store", "retrieve", "in the system", "storage", and so forth, used below can refer to storage in the file system on the server ("traditional") or in SRB.

The *BitstreamStorageManager* provides low-level access to bitstreams stored in the system. In general, it should not be used directly; instead, use the *Bitst ream* object in the content management API since that encapsulated authorization and other metadata to do with a bitstream that are not maintained by the *BitstreamStorageManager*.

The bitstream storage manager provides three methods that store, retrieve and delete bitstreams. Bitstreams are referred to by their 'ID'; that is the primary key *bitstream_id* column of the corresponding row in the database.

As of DSpace version 1.1, there can be multiple bitstream stores. Each of these bitstream stores can be traditional storage or SRB storage. This means that the potential storage of a DSpace system is not bound by the maximum size of a single disk or file system and also that traditional and SRB storage can be combined in one DSpace installation. Both traditional and SRB storage are specified by configuration parameters. Also see Configuring the Bitstream Store below.

Stores are numbered, starting with zero, then counting upwards. Each bitstream entry in the database has a store number, used to retrieve the bitstream when required.

At the moment, the store in which new bitstreams are placed is decided using a configuration parameter, and there is no provision for moving bitstreams between stores. Administrative tools for manipulating bitstreams and stores will be provided in future releases. Right now you can move a whole store (e.g. you could move store number 1 from */localdisk/store* to */fs/anotherdisk/store* but it would still have to be store number 1 and have the exact same contents.

Bitstreams also have an 38-digit internal ID, different from the primary key ID of the bitstream table row. This is not visible or used outside of the bitstream storage manager. It is used to determine the exact location (relative to the relevant store directory) that the bitstream is stored in traditional or SRB storage. The first three pairs of digits are the directory path that the bitstream is stored under. The bitstream is stored in a file with the internal ID as the filename.

For example, a bitstream with the internal ID *12345678901234567890123456789012345678* is stored in the directory:

```
(assetstore dir)/12/34/56/12345678901234567890123456789012345678
```

The reasons for storing files this way are:

- Using a randomly-generated 38-digit number means that the 'number space' is less cluttered than simply using the primary keys, which are allocated sequentially and are thus close together. This means that the bitstreams in the store are distributed around the directory structure, improving access efficiency.
- The internal ID is used as the filename partly to avoid requiring an extra lookup of the filename of the bitstream, and partly because bitstreams may be received from a variety of operating systems. The original name of a bitstream may be an illegal UNIX filename.
  When storing a bitstream, the *BitstreamStorageManager* DOES set the following fields in the corresponding database table row:

- *bitstream_id*
- *size*
- *checksum*
- *checksum_algorithm*
- *internal_id*
- *deleted*
- *store_number*
  The remaining fields are the responsibility of the *Bitstream* content management API class.

The bitstream storage manager is fully transaction-safe. In order to implement transaction-safety, the following algorithm is used to store bitstreams:

1. A database connection is created, separately from the currently active connection in the current DSpace context.
2. An unique internal identifier (separate from the database primary key) is generated.
3. The bitstream DB table row is created using this new connection, with the *deleted* column set to *true*.
4. The new connection is _commit_ted, so the 'deleted' bitstream row is written to the database
5. The bitstream itself is stored in a file in the configured 'asset store directory', with a directory path and filename derived from the internal ID
6. The *deleted* flag in the bitstream row is set to *false*. This will occur (or not) as part of the current DSpace *Context*.
   This means that should anything go wrong before, during or after the bitstream storage, only one of the following can be true:

- No bitstream table row was created, and no file was stored
- A bitstream table row with *deleted=true* was created, no file was stored
- A bitstream table row with *deleted=true* was created, and a file was stored
  None of these affect the integrity of the data in the database or bitstream store.

Similarly, when a bitstream is deleted for some reason, its *deleted* flag is set to true as part of the overall transaction, and the corresponding file in storage is *not* deleted.

The above techniques mean that the bitstream storage manager is transaction-safe. Over time, the bitstream database table and file store may contain a number of 'deleted' bitstreams. The *cleanup* method of *BitstreamStorageManager* goes through these deleted rows, and actually deletes them along with any corresponding files left in the storage. It only removes 'deleted' bitstreams that are more than one hour old, just in case cleanup is happening in the middle of a storage operation.

This cleanup can be invoked from the command line via the *Cleanup* class, which can in turn be easily executed from a shell on the server machine using */dspace/bin/cleanup*. You might like to have this run regularly by *cron*, though since DSpace is read-lots, write-not-so-much it doesn't need to be run very often.

Backup

The bitstreams (files) in traditional storage may be backed up very easily by simply 'tarring' or 'zipping' the *assetstore* directory (or whichever directory is configured in *dspace.cfg*). Restoring is as simple as extracting the backed-up compressed file in the appropriate location.

Similar means could be used for SRB, but SRB offers many more options for managing backup.

It is important to note that since the bitstream storage manager holds the bitstreams in storage, and information about them in the database, that a database backup and a backup of the files in the bitstream store must be made at the same time; the bitstream data in the database must correspond to the stored files.

Of course, it isn't really ideal to 'freeze' the system while backing up to ensure that the database and files match up. Since DSpace uses the bitstream data in the database as the authoritative record, it's best to back up the database before the files. This is because it's better to have a bitstream in storage but not the database (effectively non-existent to DSpace) than a bitstream record in the database but not storage, since people would be able to find the bitstream but not actually get the contents.

With DSpace 1.7 and above, there is also the option to backup both files and metadata via the AIP Backup and Restore feature.

Configuring the Bitstream Store

Both traditional and SRB bitstream stores are configured in *dspace.cfg*.

Configuring Traditional Storage

Bitstream stores in the file system on the server are configured like this:

```
assetstore.dir =  [dspace]/assetstore
```

(Remember that *[dspace]* is a placeholder for the actual name of your DSpace install directory).

The above example specifies a single asset store.

```
assetstore.dir =  [dspace]/assetstore_0
assetstore.dir.1 = /mnt/other_filesystem/assetstore_1
```

The above example specifies two asset stores. assetstore.dir specifies the asset store number 0 (zero); after that use assetstore.dir.1, assetstore.dir.2 and so on. The particular asset store a bitstream is stored in is held in the database, so don't move bitstreams between asset stores, and don't renumber them.

By default, newly created bitstreams are put in asset store 0 (i.e. the one specified by the assetstore.dir property.) This allows backwards compatibility with pre-DSpace 1.1 configurations. To change this, for example when asset store 0 is getting full, add a line to *dspace.cfg* like:

```
assetstore.incoming = 1
```

Then restart DSpace (Tomcat). New bitstreams will be written to the asset store specified by *assetstore.dir.1*, which is */mnt/other_filesystem/assetstore_1* in the above example.

Configuring SRB Storage

The same framework is used to configure SRB storage. That is, the asset store number (0..n) can reference a file system directory as above or it can reference a set of SRB account parameters. But any particular asset store number can reference one or the other but not both. This way traditional and SRB storage can both be used but with different asset store numbers. The same cautions mentioned above apply to SRB asset stores as well: The particular asset store a bitstream is stored in is held in the database, so don't move bitstreams between asset stores, and don't renumber them.

For example, let's say asset store number 1 will refer to SRB. The there will be a set of SRB account parameters like this:

```
srb.host.1 = mysrbmcathost.myu.edu
srb.port.1 = 5544
srb.mcatzone.1 = mysrbzone
srb.mdasdomainname.1 = mysrbdomain
srb.defaultstorageresource.1 = mydefaultsrbresource
srb.username.1 = mysrbuser
srb.password.1 = mysrbpassword
srb.homedirectory.1 = /mysrbzone/home/mysrbuser.mysrbdomain
srb.parentdir.1 = mysrbdspaceassetstore
```

Several of the terms, such as *mcatzone*, have meaning only in the SRB context and will be familiar to SRB users. The last, *srb.parentdir.n*, can be used to used for addition (SRB) upper directory structure within an SRB account. This property value could be blank as well.

(If asset store 0 would refer to SRB it would be *srb.host* = ..., *srb.port* = ..., and so on (*.0* omitted) to be consistent with the traditional storage configuration above.)

The similar use of *assetstore.incoming* to reference asset store 0 (default) or 1..n (explicit property) means that new bitstreams will be written to traditional or SRB storage determined by whether a file system directory on the server is referenced or a set of SRB account parameters are referenced.

There are comments in dspace.cfg that further elaborate the configuration of traditional and SRB storage.