# SmartDir: An Intelligent Directory System

6.033 Design Project 1

**Colin Hong**

**February 26th, 2014**

## 1. Introduction

The concept of a dynamic database that stores filepaths according to specific criteria was first introduced in the mid 1990's, as a part of BeOS. However, it was not until 2004, when Apple released OS X 10.4 Tiger, that the feature known as "Smart Folders" was introduced to the mass population. When Windows Vista launched in 2006, it debuted a similar feature called "Smart Groups". [1]

Eight years later, there is no "Smart Folder/Group" feature implemented in the UNIX operating system. This report outlines the design of such a feature, called for internal purposes a *SmartDir* – Smart Directory.

## 2. Overview

A *SmartDir* directory allows the user to search a path for files that match a certain criteria: file size, time of creation or modification, file user ID, permission bits, extended attribute names, file names, etc. By creating a new directory containing the matching files, with symbolic links to the originals, the *SmartDir* allows the user to view and access the relevant files in a centralized location, without changing any original filepaths.

The user can access the SmartDir functionality by calling:

```
searchmount mountpoint searchpath [expressions]
```

There are many reasons why a user might want to create a SmartDir. One, if a user wants to group similar filetypes together for easy access. A *SmartDir* displays and allows access to the files, without displacing the files from their original location. Two, if a user wants to edit similar filetypes, without having to manually locate each file. An example of this would be readme files in individual programs; if a user wanted to access the readme files themselves, a *SmartDir* is much more straightforward than accessing each program directory and then accessing the readme file.

A sample use scenario can be seen on the next page in **Fig 1.**

**FIG 1.**

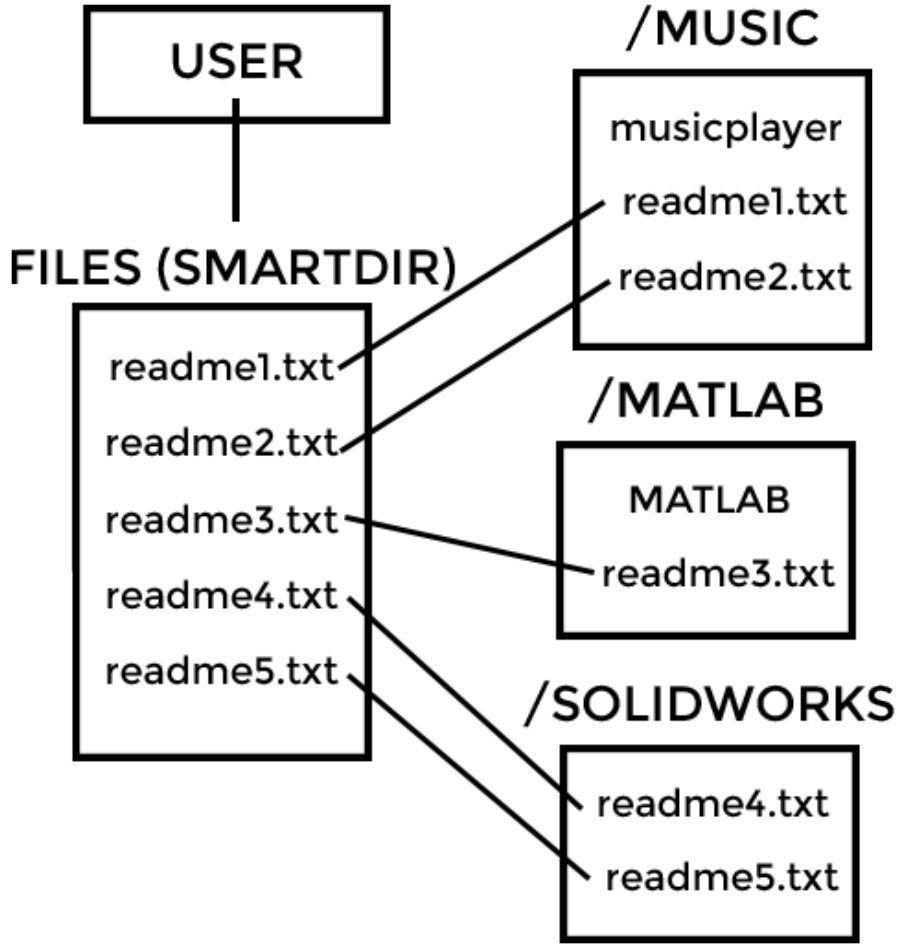


FIG 1. The user has created a SmartDir, with the expression "filename contains readme". The resulting SmartDir returns symbolic links to the files that meet the search criteria. The user can access the files from the SmartDir, rather than having to access each folder (/Music, /MATLAB, /SOLIDWORKS) individually.

## 3. Design

The implementation of *SmartDir* is optimized to maximize speed first, and minimize space required second. This is because if *SmartDir* is not fast and responsive, *SmartDir* will see little adoption and use. Additionally, most users have excess on-disk memory, and do not use their RAM to capacity. For this reason, I decided to allow up to 500 MB of RAM space for the in-memory database. This will allow the program to run thousands of times faster than if it were completely on-disk. The time complexity analysis is explored further in section **4.**

### 3.1 Data Structures

The primary component of *SmartDir* is the three-part on-disk data structure, which consists of a directory inode *mountPoint,* array *nameList* and database *dataEntries*. The secondary component of *SmartDir* is the in-memory datastructure, which maintains the Directory Entry Pointers, or DEP, as well as a buffer representation of *nameList* and *dataEntries* datablocks for the next 100 files, default sorted by alphabetical. The number 200 was chosen because a typical Athena computer, with Terminal open and maximized, can display 100 output lines. An impatient user may quickly scroll through the output, requiring a buffer of 100 additional output lines to be maintained.

The SmartDir directory is associated with an inode, *mountPoint*, which contains information about the directory. There are also two datastructures: an array *nameList* and and a database *dataEntries*. Remember, our primary design objective is speed, not size efficiency. Therefore, we use two data structures, an on-disk directory and an in-memory representation.

#### 3.1.1 On-Disk Data Structure

The inode directory mountPoint is stored on the disk, along with the *nameList* and *dataEntries. nameList* contains a sorted list of file names within the directory. The database *dataEntries* contains filename keys, and filepath values.

The datablock array *nameList* is used to assist the *SmartDir* during the initialization. The filenames are default sorted in alphabetical order, but can be sorted by date modified, size, date created, etc. During the initialization step, we will call the indexes of *nameList* in order. The datablock database *dataEntries* simply maps the filenames to the symbolic filepaths.

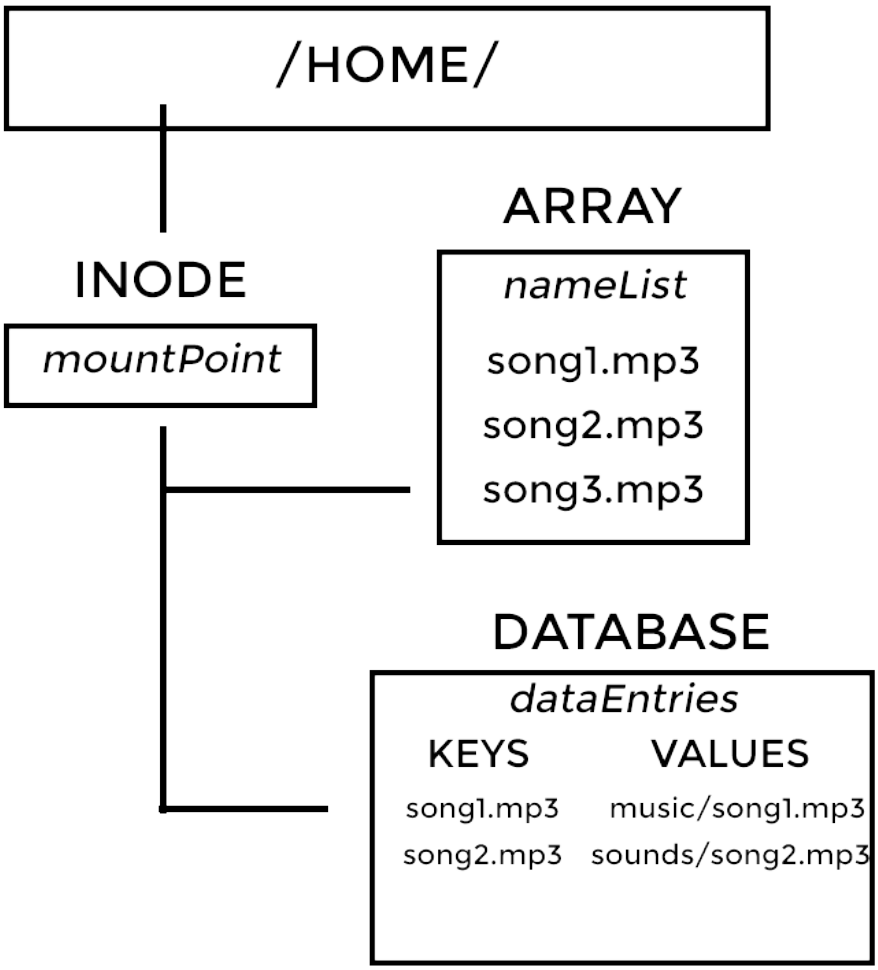


**FIG 2:** Visual Representation of the mountPoint inode and the two on-disk data structures, nameList and dataEntries. When the directory inode *mountPoint* is accessed initially, it calls upon nameList and dataEntries to create the DEP objects. The implementation is explored further in **3.2**

### 3.1.2 In-Memory Data Structure

When searchmount is initially called, or the directory mountPoint is accessed, *SmartDir* creates DirEntryPointers for each file contained within the directory. These DEPs are then accessed to provide information on the filename or symbolic filepath. The DEPs provide the actual functionality of the *SmartDir* from the user's perspective, i.e. when the user clicks on a represented file, *SmartDir* calls the DEP to provide the symbolic filepath to access said file. Therefore, it is necessary to store these DEPs in-memory, to allow quick and repeated access without appearing to "lag".

An in-memory representation of *dataEntries* and *nameList* is also created for the subsequent 100 files that may be called. If the user wishes to see past the current 100 files, then *SmartDir* can create the DEP objects using in-memory data, without having to make costly calls to the disk to access the information. *SmartDir* then loads the next subsequent 100 *dataEntries* and *nameList* indexes that may be called.

### 3.2 Implementation

This section will describe the implementation of the *SmartDir* directory system, and provides sample codes of the created API methods.

#### 3.2.1: Initialization

The *SmartDir* directory system is initially created through the command:

```
searchmount mountpoint searchpath [expressions]
```

This calls createDirectory(mointPoint, searchPath, expression), which in turn creates the directory inode *mountPoint* in the current directory, and applies the expression(s) along searchPath.

First, createDirectory(*mointPoint*, searchPath, expression) creates the database *dataEntries* by calling openDatabase(*mountPoint*), and creates the array *nameList*. Then, createDirectory() makes a find() call on searchPath using the given expression(s). For each returned file, createDirectory() uses the given database API to put(*mountPoint*, filename, filepath) in *dataEntries*, and add the filename to *nameList*. The given database API methods are covered in section **3.2.3.** Here is a roughly chronological order of the initialization call.

1. `searchmount mountpoint searchpath [expressions]`
2. `createDirectory(mountPoint, searchPath, expressions)`
    a. *dataEntries*
    b. *nameList*
3. `find() searchpath [expressions]`
    a. `put(mountpoint, filename, filepath) -> dataEntries`
    b. `add(filename) -> nameList`

createDirectory() also initializes an inotify kernel pointing to each subdirectory of searchPath, to monitor modifications to the searchPath directory. This will be further explained in **3.3.5**.

#### 3.2.2 DirEntryPointer

*SmartDir* requires the creation of a new class, DirEntryPointer, or DEP for short, to store variables. These variables are:

- filename: the name of the file eg: “song1.mp3”
- filepath: the path of the file eg: “Desktop/music/song1.mp3”

- index: the index of the file within *namefile* eg: 0, 1, 2…
- parent: the pointer to the database

We initialize DEP with:

```
DEP(filename, filepath, index, parent)
```

### 3.2.3 Given Database API Methods

The implementation of *SmartDir* makes use of the following database API methods.

- DBPointer openDatabase(String filename): opens a database at the specified location on disk; database is created if it doesn't exist.
- put(DBPointer p, String key, String value): stores the specified key/value in the specified database. If the key already exists, the previous value is overwritten.
- String get(DBPointer p, String key): returns the value of the specified key in the database.
- remove(DBPointer p, String key): deletes the specified key from the database

### 3.2.4 Implemented API Methods

The implementation of *SmartDir* requires the operating system to invoke a collection of methods to create and read contents of the directory.

getFirstDirectoryEntry(name) returns DEP

Get the first element of the *nameList* array. Armed with the filename, get the filepath by calling a get() on the database *dataEntries* using the filename as the key. Returns the DEP associated with the first file within the *Smart Group* directory *mountPoint*.

```
filename = name.nameList.getindex(0)

filepath = get(name.dataEntries, filename) //from database API

return DEP(filename, filepath, 0, name)
```

readNextDirectoryEntry(DEP prevFile) returns DEP

Given a DEP prevFile, call the next element in the *nameList* array. This is done by calling prevFile's index aspect, and iterating by one. Next, gets the corresponding filename through an index call to *nameList.* Armed with the filename, get the filepath by calling a get() on the database *dataEntries* using the filename as the key. Returns the DEP that is the next directory entry after prevFile.

```
index = prevFile.index + 1

filename = prevFile.parent.nameList.getindex(index)

filepath = get(prevFile.parent.dataEntries, filename)

return DEP(filename, filepath, index, prevFile.parent)
```

filename(DEP entry) returns String

Given a DEP, makes a call to the DEP's filename attribute. Returns the name of the file represented by directory entry entry.

```
return entry.filename
```

readSymbolicLink(DEP entry) returns String

Given a DEP, makes a call to the DEP's filepath attribute. Returns the target of a symbolic link stored in the directory entry entry.

```
return entry.filepath
```

### 3.3 System Interaction

This section briefly details how the operating system accesses files in the *SmartDir*.

When the OS wants to begin iterating over files in the *SmartDir*, the OS will begin by calling getFirstDirectoryEntry, then repeatedly calling readNextDirectoryEntry. This will create the relevant DEPs according to how the nameList is sorted (default: alphabetical). The calls must access on-disk memory, but create the DEPs in RAM. The time complexity of these actions is analyzed in section **4.**

For example, a user calls ls and wants to print the contents of the selected *SmartDir*. *SmartDir* iterates through the DEPs, calling filename() and readSymbolicLink() to print the name and paths of the entries.

### 3.4 inotify

This section explains the implementation of the inotify structure into *SmartDir* to provide updated, proper filepaths.

In order to ensure the *SmartDir* directory stays up to date, an inotify kernel is initialized to the each subdirectory of the searchPath directory. Every time search path is modified, the kernel reports a message, such as IN_CREATE. The following are the inotify reports that *SmartDir* cares about:

- `IN_CREATE`: the monitored directory had a file added to it. *SmartDir* applies the expression stored in the inode *mountPoint* to the file, and updates the datablocks *nameList* and *dataEntries* if need be.
- `IN_DELETE`: the monitored directory had a filed deleted from it. *SmartDir* checks if the file was stored in the database *dataEntries*, and updates the datablocks *nameList* and *dataEntries* if need be.
- `IN_MOVED_FROM`: The monitored directory had a file moved out of it. *SmartDir* acts as if this was an IN_DELETE notification.
- `IN_MOVED_TO`: The monitored directory had a file moved into it. *SmartDir* acts as if this was an IN_CREATE notification.

- `IN_DELETE_SELF`: the monitored directory was deleted. The *SmartDir* acts as if every file within the monitored directory had an IN_DELETE notification.
- `IN_MOVE_SELF`: the searchPath directory was moved itself. Update the inode pointer in *mountPoint* to reference the new searchPath directory path.

Several clarifications:

- If a file is moved from one monitored directory to another (i.e. moved subdirectories but still within the searchpath), the implementation of *SmartDir* deletes the file and re-adds it.
- If the searchPath directory is moved itself, we made the conscious decision to shift the *SmartDir* filepaths to the new filepaths, as most likely the user was reorganizing and wanted to maintain their *SmartDir* directories.
- If an inotify node watches a directory included in multiple *SmartDir* searchPaths, each with their own expression, then each *SmartDir* directory applies the notification to itself.

## 4. Performance Analysis

As previously stated in section **3**, our primary consideration when implementing *SmartDir* is performance. This section will first analyze the bounded time of the process, then apply the process to several real-world examples,

### 4.1 Initialization Performance

The performance of the initialization call is bounded by O(n*2*x+C), where is the number of files added to the *SmartDir* directory, x is the time required to read/write to the on-disk memory, and C is a constant time required to create the directory inode and empty datablocks.

This is because the process must apply the given expression to every file within searchPath. Then, for every match, it must add the filename to the on-disk array *nameList* and on-disk database *dataEntries*, in O(n*2*x) time. The constant time that comes from the initial inode and datablock creation is small enough that it won't affect the performance time.

Thus, the performance time is O(n*.02s), because a random disk I/O takes ~10ms.

### 4.2 First-Access Performance

The initialization performance only covers the time taken by creating the *SmartDir* directory on-disk. In order for the user to access any of the files, the operating system must apply the given method API's to create the first 100 DEPs. This system interaction is bounded by min(2s, n*.02s), because each DEP creation takes 2 random disk I/O calls, which each take ~10ms.

### 4.3 Subsequent Performance

After the directory is initialized, and the DEPs are created, all subsequent operations take negligible amounts of time. Any operation or access of the DEPs takes ~.000001s, due to DRAM load. Subsequent DEPs (if there are more files in the directory than 100) are loaded during CPU downtime, ie in-between inputs, so as to ensure there is always a buffer zone of preloaded DEPs onto the RAM.

### 4.4 Performance Applied to Real World Situations

Thus, the performance of the *SmartDir* is n*.02s+min(2s, n*.02s), where m is the total number of files in searchPath, and n is the number of matching files. In order to put this into context, the performance of *SmartDir* is evaluated on four different workloads:

1. A large searchPath with a rare expression
2. A small searchPath with a common expression
3. A large searchPath with a common expression
4. A small searchPath with a rare expression

The following assumptions are made:

- A large searchPath, such as /user, is ~100,000 files
- A small searchPath, such as /MITwork, is ~1,000 files
- A rare expression is "filename contains .tex"
- A common expression is "(filebody || filename) contains Apple"

| Case | searchPath | files | expression | Matches | Initialization (seconds) | DEP load (seconds) | Total Time |
|---|---|---|---|---|---|---|---|
| 1 | /user | ~100,000 | ".tex" | 19 | .38 | .38 | .76 |
| 2 | /MITwork | ~1,000 | "Apple" | 17 | .34 | .34 | .68 |
| 3 | /user | ~100,000 | "Apple" | 1510 | 30.2 | 2 | 32.2 |
| 4 | /MITwork | ~1,000 | ".tex" | 2 | .04 | .04 | .08 |

### 4.5 Additional Time-Performance

The find() call itself and the inotify kernels add additional time to the process. However, the find() call takes ~3 seconds even for a system search, so it does not add a significant amount of time. The inotify kernels depends on the complexity of the file organizational structure, i.e. how many subdirectories exist. This is difficult to analyze, but inotify kernels are stored in-memory and thus shouldn't add significant time.

### 4.5 Space Performance

The size of the RAM dedicated to *SmartDir* while *SmartDir* is being accessed (and after access until needed by a more urgent program) is 500MB. The size of the

directory inode & the stored list & database take negligible amounts of on-disk space, because they only store string values & filepaths.

## 5. Exceptions

If two files with the same file name match the given expression, then the process appends the respective parent directory to the beginning of the name within *nameList* and *dataEntries*. This allows the user to differentiate them easily within the directory listing.

## 5. Conclusion

A *SmartDir* directory allows the user to search a path for files that match a certain criteria, and store the symbolic links. *SmartDir* allows the user to view and access the relevant files in a centralized location, without changing any of the original filepaths.

The described implementation allows for persistence through reboot, and the 100 file in-memory buffer allows the user to quickly call directory listings. This system has been optimized for performance. The use of the *nameList* array and *dataEntries* database take up very little space, and leave the original files unmodified. The performance of the searched directories is not seriously inhibited, as the only interaction with them is the find() call, which typically runs in nano seconds.

Further research and testing should be done to determine the ideal dedicated RAM size. 500MB was our safe estimate to ensure proper performance, but in reality much less should be required, depending on the size of the DEP objects.

## 6. Collaboration & References

Credit is given to Brad Eckert for contribution to the API method design.

[1] http://gigaom.com/2010/01/13/the-smart-mac-smart-folders-in-os-x/