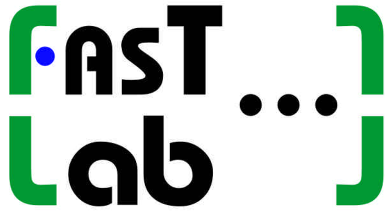

FASTLab Inc.
220 Santa Anita Rd.
Santa Barbara, California, 93105, USA
http://www.FASTLabInc.com

FASTLab Music Analysis Kernel (FMAK)
Version 2.0
Analysis and Clustering Utilities
Technical Documentation

Stephen Travis Pope
Alex Kouznetsov
September, 2004

TABLE OF CONTENTS

Introduction 2
The FMAK Analysis Driver 2
The Playlist Summary Application 3
The Batch Analysis Front-end 4
The FMAK Clusterer 4
The FMAK Database Extractor 7
Appendix A. SQL Database Table Formats 8
Appendix B. FC_IO DB-Writer File Formats 10
Appendix C. FMAK Database Setup and Management 13

Copyright FASTLab, Inc. 2004. Proprietary and confidential.

Introduction

This document describes the usage of the utility programs delivered with the FASTLab Music Analysis Kernel (FMAK) software library. The associated programs are:

(1) the analysis driver used to create rows in a music analysis database;

(2) the batch analysis front-end used for running the analysis driver over lists of music data files;

(3) the play-list summarizer application used for exporting song lists from Apple's iTunes music database for use with the batch analysis driver;

(4) the database clusterer, which finds clusters of similar feature collections in an FMAK database; and

(5() the database extractor, which selects the genre representatives from a database (after clustering) and produces a run-time flat file genre database dump.

The Appendices of the document describe the database management and file I/O formats used by these utilities.

The FMAK Analysis Driver

Overview

The FMAK analysis driver utility provided as a part of the FMAK package performs the necessary operations to load a sound file into memory, create the feature extraction data structures, call the FMAK library analysis functions, and store the results into a relational database. More information on the FMAK Library can be found in the FMAK Library Technical Documentation. The Analysis Driver is often run in batch mode over large sets of sound files, representing many CDs worth of material, however, single song analysis is supported as well.

Program Operation

The analysis driver program is executed from scripts with command-line options to configure the various analysis phases; these will determine the properties of the resulting feature collection.

The command-line options are:

```
Analyzer
    -s file_name                                # AIFF or WAV input file
                                                # window size and hop size for the RMS
    -r rmsWindow_size rmsHop_size               # time-domain analysis
    -f fftWindow_size fftLen fftHop_size        # for the FFT spectral analysis
    -l lpcWindow_size lpcOrder lpcHop_size      # for the LPC analysis
    -w fwtWindow_size fwtLen fwtHop_size        # for the wavelet analysis
    -i artist album title genre                 # 4 strings for meta-data
     -d [DIRECTORY]                             # Specify the directory for FC dump
     -t                                         # Displays the parsed command line and exits.
     -?                                         # Displays a usagemessage
    -p [host_name user_name password database_name]
                                                # PostgreSQL database parameters
```

Typical values for testing

```
Analyzer test.aiff -r 1024 4096 4 -f 4096 1024 8192 -l 2048 24 8192 -w 2048 1024 4096
```

or, to use all the defaults, just give the file name,

```
Analyzer test.aiff
```

or supply song metadata for batch database population

```
Analyzer -s song.aiff -i song_title artist_name album_name track
```

There are other options that are described in comments in the source code file Analysis-Driver.cpp. More details of the analysis process are given in the FMAK library technical documentation.

The Playlist Summary Application

For building a large FMAK music database, an interface has been developed between the Apple iTunes program and the FMAK analysis driver. This allows users to collect music archives in iTunes libraries and then produce a script for running the FMAK analysis driver over the entire iTunes library.

The application Playlist_Summary is a compiled AppleScript tool that processes an iTunes music library database and creates a text file of its contents for use with the FMAK batch analysis driver. To use the application, place it (the .app file) in your iTunes scripts folder (typically, this will be ~/Library/iTunes/Scripts). Then start the script from within the iTunes application using the scripts menu. The script will create a text file with listings for all of the songs in your library. The file format is shown the following example. All of the text up to the line that starts "%%" is a comment describing the file format; the following lines are the first entry in the database. For each song, the file name, artist name, album/CD name, track title, and musical genre (as given in iTunes), are each listed on separate lines. Song entries are separated by a blank line.

```
Playlist SUMMARY • Monday, May 3, 2004 • 3332 tracks

LOCATION
ARTIST
ALBUM
TRACK
GENRE

%%
/Volumes/BigNoise/EMA/iTunes/Compilations/Dance/01 Play The Song.aif
2 Fabiola
Dance Dance Dance
Play The Song
Electronica/Dance
```

...other songs follow here...

The Batch Analysis Front-end

A separate small program is included with the FMAK utilities to take the song listing file described above and call the analysis driver program for each entry in the content database (i.e., the iTunes library). The Batch_Processor program is usually executed from a UNIX command-line shell after the playlist summary file is dumped from iTunes. The shell command to do this looks like the following.

```
Batch_Processor database_summary_file_name.txt [path_to_FMAK_Analyzer]
```

In this command, the first argument is required and is the name of the playlist summary file; the second (optional) argument is the file path to the FMAK analysis driver; it defaults to "./" meaning that the batch processor expects a file in the local directory called Analyzer.

Note that this process can take quite a long time (up to several days) for large databases. It is often useful to split a playlist summary file into several pieces and run the analysis process on several computers at once.

The FMAK Clusterer

The FMAK database clusterer is a development/training tool designed to aid in finding the representative songs within the EMA database. The clusterer operates as a standalone executable and accesses the data in the SQL database that has been populated by the EMA database driver running the analysis on the set of all songs. The clusterer produces a cluster id label for each song that is stored back in the database table.

The clusterer is a full-featured multi-stage clustering algorithm implementation optimized for the specific requirements of EMA application:

- Optimized for large databases (uses a pre-clustering stage)
- Good performance with irregularly shaped clusters and over wide cluster size variation
- Low sensitivity to outliers

The EMA clusterer is almost entirely based on the CURE clustering algorithm as described in S. Guha, R. Rastogi, and K. Shim. “CURE: An efficient clustering algorithm for large databases.” In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 73--84, New York, 1998.

CURE employs a novel hierarchical clustering algorithm that adopts a middle ground between the centroid-based and the all-point extremes. In CURE, a constant number c of well-scattered points in a cluster are first chosen. The scattered points capture the shape and extent of the cluster. The chosen scattered points are next shrunk towards the centroid of the cluster by a fraction alpha. These scattered points after shrinking are used as representatives of the cluster. The clusters with the closest pair of representative points are the clusters that are merged at each step of CURE's hierarchical clustering algorithm.

The scattered points approach alleviates the shortcomings of both the all-points as well

as the centroid-based approaches. CURE is less sensitive to outliers since shrinking the scattered points toward the mean dampens the adverse effects due to outliers; outliers are typically further away from the mean and are thus shifted a larger distance due to the shrinking. Multiple scattered points also enable CURE to discover non-spherical clusters. For the centroid-based algorithm, the space that constitutes the vicinity of the single centroid for a cluster is spherical. Thus, it favors spherical clusters and splits the elongated clusters. On the other hand, with multiple scattered points as representatives of a cluster, the space that forms the vicinity of the cluster can be non-spherical, and this enables CURE to correctly identify such clusters.

Note that the kinds of clusters identified by CURE can be tuned by varying alpha between 0 and 1.CURE reduces to the centroid-based algorithm if alpha = 1, while for alpha = 0, it becomes similar to the all-points approach. CURE's hierarchical clustering algorithm uses space that is linear in the input size n and has a worst-case time complexity of O(n2 log n). For lower dimensions (e.g., two), the complexity can be shown to further reduce to O(n2). Thus, the time complexity of CURE is no worse than that of the centroid-based hierarchical algorithm.

The clustering algorithm starts with each input point as a separate cluster, and at each successive step merges the closest pair of clusters. In order to compute the distance between a pair of clusters, for each cluster, c representative points are stored. These are determined by first choosing c well-scattered points within the cluster, and then shrinking them toward the mean of the cluster by a fraction alpha. The distance between two clusters is then the distance between the closest pair of representative points – one belonging to each of the two clusters. Thus, only the representative points of a cluster are used to compute its distance from other clusters.

The c representative points attempt to capture the physical shape and geometry of the cluster. Furthermore, shrinking the scattered points toward the mean by a factor alpha gets rid of surface abnormalities and mitigates the effects of outliers. The reason for this is that outliers typically will be further away from the cluster center, and as a result, the shrinking would cause outliers to move more toward the center while the remaining representative points would experience minimal shifts. The larger movements in the outliers would thus reduce their ability to cause the wrong clusters to be merged. The parameter alpha can also be used to control the shapes of clusters. A smaller value of alpha shrinks the scattered points very little and thus favors elongated clusters. On the other hand, with larger values of alpha, the scattered points get located closer to the mean, and clusters tend to be more compact.

Optimizations

Analysis of not well-separated clusters in large-scale databases presents the problem of computational complexity. If the distance between a pair of clusters is small, then having a simpler solution of sampling a fraction for each of them may not enable the clustering algorithm to distinguish them. The reason for this is that the sampled points for a cluster may not be uniformly distributed and points across clusters may end up becoming closer to one another than points within the cluster. As the separation between clusters decreases and as clusters become less densely packed, samples of larger sizes are required to distinguish them. However, as the input size n grows, the computation that needs to be performed by the clustering algorithm could end up being fairly substantial due to the

O(n2 log n) time complexity.

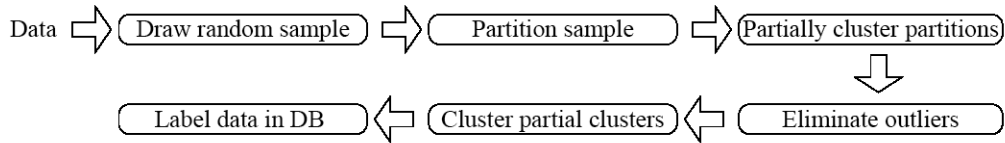

Figure 1. EMA Clusterer operation workflow.

Given the total number of input data points n, the basic idea is to partition the sample space into p partitions, each of size n/p. We then partially cluster each partition until the final number of clusters in each partition reduces to n/(p*q) for some constant q > 1. Alternatively, we could stop merging clusters in a partition if the distance between the closest pair of clusters to be merged next increases above a certain threshold. Once we have generated n/(p*q) clusters for each partition, we then run a second clustering pass on the n/q partial clusters for all the partitions (that resulted from the first pass). The advantage of partitioning the input is a substantial reduction in the computation complexity.

Running the Clusterer

The clusterer can be executed from the command line with following parameters:

```
Clusterer [REQUIRED_FLAGS] [OPTIONAL_FLAGS]
```

Where the following required and optional flags are supported:

-p [NUM]	Number of pre-clustering partitions, see ‘p’ parameter above
-q [NUM]	Pre-clustering partition factor, see ‘q’ parameter above
-k [NUM]	Desired number of clusters before stopping. Clusterer will continue merging clusters until k or less are left.
-c [NUM]	Number of well-scattered representatives per cluster, see ‘c’ parameter above.
-a [NUM]	Representative scaling factor, see ‘alpha’ parameter above
-u [STR]	Database user name
-s [STR]	Database password

(optional)

-t	Display parsed command parameters and exit. For testing purposes only.
-?	Display help message

Typical Values for these (for a database of approx. 3000 songs) are

```
Clusterer -p 5 -q 5 -k 30 -c 10 -a 1 -u user_name -s db_password -t
```

The specific values of these parameters are dependent on the size of database and more importantly, on the type of clustering result desired. More details on the clusterer implementation can be found in the CURE paper cited above.

Clusterer Database Interface

The clusterer uses the Analysis Results database for data input, labelling during cluster-

ing, and output. Clusterer reads the pertinent data from all song for which ClusteringStatus field is set to “X”. The default value for ClusteringStatus is “O”; such songs will be ignored by the clusterer. Therefore, prior to executing the clusterer, the appropriate subset of song must be marked with ClusteringStatus = “X”. For clustering, the selected database subset cannot contain songs with duplicate name and data as those will interfere with proper operation of K-D trees used for cluster merging.

When clustering is complete, a number of different values may be assigned to the ClusteringStatus field:

-- After building from scratch, to have all records used by the clusterer, do this

```
update FeatureCollections set ClusteringStatus = 'X' where ClusteringStatus != 'X';
```

-- To look at the list of final cluster representatives, do this

```
select artist, title from FeatureCollections where ClusteringStatus = 'F';
```

-- Get min/avg/max of RMS statistics for average and peak feature tables using joins

```
SELECT max(RMS), avg(RMS), min(RMS) FROM FeatureTables, FeatureCollections
    where FeatureCollections.AvgFT = FeatureTables.oid;
SELECT max(RMS), avg(RMS), min(RMS) FROM FeatureTables, FeatureCollections
    where FeatureCollections.WeightedFT = FeatureTables.oid;
```

The FMAK Database Extractor

The final utility used by FMAKL applications is the database extractor; this is a small program that searches through the FMAK database for songs with the clustering status flag set to ‘F’ -- meaning that they have been identified as genre cluster representatives. The number of these genre representatives is chosen by the arguments to the clusterer. Once they are found, the DB extractor simply writes their feature tables out in flat binary format to a file named cluster_db.fmak. The command line for the DB extractor takes no arguments; it is executed simply as,

```
DBExtractor
```

Appendix A. SQL Database Table Formats

The following is the SQL table definition code for the FMAK database. Both the analysis driver and the clusterer use these database tables.

```
-- FMAK 2 Database Table Definition File

-- PostgresSql table definitions for FMAK databases

--
-- (C) Copyright FASTLab Inc. 2003. All rights reserved.
-- THIS IS UNPUBLISHED PROPRIETARY SOURCE CODE
-- The copyright notice above does not evidence any
-- actual or intended publication of such source code.


-- Feature Table Table
-- this stores the individual feature tables that make up a song's feature collection

DROP TABLE FeatureTables;

CREATE TABLE FeatureTables (
	Time real,			-- start time of the table
	Duration real,			-- the table's duration
	RMSWindowSize integer,		-- time-domain features
	RMS real,
	Peak real,
	LPRMS real,
	HPRMS real,
	ZeroCrossings integer,
	DynamicRange real,
	StereoWidth real,		-- spatial-domain features
	SurroundDepth real,
	CenterCorrelation real,
	FFTWindowSize integer,		-- spectral-domain features
	SpectralCentroid real,
	SpectralSlope real,
	SpectralVariety real,
	SpectralBands real[],		-- 4-band spectrum
	HiFreqBalance real,		-- hi-frequency properties
	HiFreqVariety real,
	HiFreqCorrelation real,
	STrackBirths real,		-- spectral track data
	STrackDeaths real,
	LPCWindowSize integer,		-- LPC data
	LPCresidual real,
	LTrackBirths real,
	LTrackDeaths real
);

GRANT ALL on FeatureTables TO fmak;
```

```
-- Feature Collection Table -- there is one of these per song

DROP TABLE FeatureCollections;

CREATE TABLE FeatureCollections (
        Name text,                        -- name string
        Album text,
        Artist text,
        Title text,
        Genre text,                       -- should be a list of related genres
        Version integer,                  -- version of the recording or analysis
        AnalyzedOn timestamp,
        Year integer,                     -- year of the recording
        Duration float,
        AvgFT integer,                    -- average and peak feature tables
        WeightedFT integer,
        TypicalFT integer,
        Tempo integer,                    -- average tempo
        Segments real[],                  -- segmentation data
        SegmentWeight real,
        VerseLength real,
        FirstVerseStart real,
        VerseWeight real,
        FadeIn real,                      -- fade in/out times
        FadeOut real,
        QuietSecs real,
        RepeatSecs real,
        ClusteringStatus char(1) DEFAULT 'O' CHECK (ClusteringStatus in ( 'O', 'X', 'P', 'R', 'C', 'F') ),
                                          -- used by the clusterer
        ClusterId int DEFAULT 0 -- used by the clusterer
);

GRANT ALL on FeatureCollections TO fmak;


-- Genre Table -- this is the network of musical genres built by the clusterer

DROP TABLE Genres;

CREATE TABLE Genres (
        Name text,                        -- name string
        Parent integer,                   -- parent's id
        siblings integer[],               -- path of [sibling-id -> distance weighting]
        siblingDistances real[]
);

GRANT ALL on Genres TO fmak;
```

Appendix B. FC_IO DB-Writer File Formats

For run-time applications (such as the Expert Mastering Assistant), it is often required to store FMAK feature collections in flat binary files. A special file format has been designed and implemented for this purpose. The FC_IO file format uses two data structures, one that maps onto a FeatureTable, and one for storing FeatureCollections. These each map directly onto the C++ class definitions, as shown in the C data structure declarations below.

```
// The data structure used to store feature tables in flat binary files.
// These structures are all that is visible to the run-time application.
// They are structs rather than classes so we can use them with fread/fwrite.

typedef struct {
    FeatureDatum mRMS;                  // Rectangular-windowed RMS amplitude
    FeatureDatum mPeak;                 // Max sample amplitude
    FeatureDatum mLPRMS;                // RMS amplitude of LP-filtered signal
    FeatureDatum mHPRMS;                // RMS amplitude of HP-filtered signal
    FeatureDatum mZeroCrossings;        // Count of zero crossings
    FeatureDatum mDynamicRange;         // RMS dynamic range of sub-windows
    FeatureDatum mPeakIndex;            // RMS peak sub-window index
    FeatureDatum mTempo;                // RMS/FWT instantaneous tempo estimate
    FeatureDatum mTimeSignature;        // Time signature guess
    FeatureDatum mBassPitch;            // Bass pitch guess in Hz
    FeatureDatum mBassNote;             // Bass note (MIDI key number) guess
    FeatureDatum mBassDynamicity;       // Bass note dynamicity (size of histogram)
    FeatureDatum mStereoWidth;          // L/R difference
    FeatureDatum mSurroundDepth;        // Front/Surround difference
    FeatureDatum mCenterDistinction;    // Center vs. L/R sum difference
    FeatureDatum mBandSpectrum[4];      // 2.5-octave FFT data (4 points -- spectral bands)
    FeatureDatum mSpectralCentroid;     // Spectral centroid measure
    FeatureDatum mSpectralSlope;        // Spectral slope measure
    FeatureDatum mSpectralVariety;      // Inter-frame spectral variety measure
    FeatureDatum mSTrackBirths;         // Spectral peak track births
    FeatureDatum mSTrackDeaths;         // Spectral peak track deaths
    FeatureDatum mLPCResidual;          // LPC residual level (noisiness)
    FeatureDatum mLPCPitch;             // Pitch estimate
    FeatureDatum mLTrackBirths;         // LPC formant peak track births
    FeatureDatum mLTrackDeaths;         // LPC formant peak track deaths
} FeatureTableStruct;

// The data structure used to store feature collections in flat binary files

typedef struct {
    char * mName, * mArtist, * mTitle, * mAlbum, * mGenre;
    FeatureDatum mSampleRate;
    FeatureDatum mDuration;
    FeatureDatum mNumSegments;
    FeatureDatum mSegmentWeight;
```

```
        FeatureDatum mVerseLen;
        FeatureDatum mFirstVerse;
        FeatureDatum mQuietSections;
        FeatureDatum mLoudSections;
        FeatureDatum mRepeatSections;
        FeatureDatum mFadeIn;
        FeatureDatum mFadeOut;
        FeatureTableStruct * mAvgData;          // the average and peak feature tables
        FeatureTableStruct * mWeightedData;
} FeatureCollectionStruct;
```

The C++ methods to write out a feature collection into a flat binary file is given below as an example. The reader side is quite simple, and is included in the FMAK source file FC_IO.cpp**.**

```
#define FC_IO_WRITE_STRING(MEMBER_NAME) \
            fprintf(output, "%s\n", mFeatureCollection->MEMBER_NAME)
#define FC_IO_WRITE_DATUM(MEMBER_NAME) \
            num_written = fwrite( & mFeatureCollection->MEMBER_NAME,  \
                    sizeof(FeatureDatum), 1, output); \
            if (num_written != 1) result = false;

// Writer method

bool FC_IO :: write_FC() {
        if (mDirectory.empty())
            mDirectory += FC_IO_DEFAULT_NAME;
        FILE * output = fopen(mDirectory.c_str(), "w");
        if (output == NULL) {
            printf("Error opening output file\n");
            return false;
        }
        size_t num_written;
        bool result = true;
                                // write the string header fields on separate lines
        FC_IO_WRITE_STRING(mName);
        FC_IO_WRITE_STRING(mArtist);
        FC_IO_WRITE_STRING(mTitle);
        FC_IO_WRITE_STRING(mAlbum);
        FC_IO_WRITE_STRING(mGenre->mName);
                                // write the feature collection numerical data
        FC_IO_WRITE_DATUM(mDuration);
        FC_IO_WRITE_DATUM(mSampleRate)
        FC_IO_WRITE_DATUM(mSegmentWeight)
        FC_IO_WRITE_DATUM(mVerseLen)
        FC_IO_WRITE_DATUM(mFirstVerse)
        FC_IO_WRITE_DATUM(mQuietSections)
        FC_IO_WRITE_DATUM(mLoudSections)
        FC_IO_WRITE_DATUM(mRepeatSections)
```

```
	FC_IO_WRITE_DATUM(mFadeIn)
	FC_IO_WRITE_DATUM(mFadeOut)
				// now write the peak and average feature tables in their compact format
	if (result)
		result = this->write_FT(mFeatureCollection->mWeightedData, output);
	if (result)
		result = this->write_FT(mFeatureCollection->mAvgData, output);
	fclose(output);
	return result;
}

// Write a feature table as a binary structure

bool FC_IO :: write_FT(const FeatureTable & ft, FILE * output) {
	FeatureTableStruct out_fv;
	out_fv.mRMS = ft.mRMS;
	out_fv.mPeak = ft.mPeak;
	out_fv.mLPRMS = ft.mLPRMS;
	out_fv.mHPRMS = ft.mHPRMS;
	out_fv.mZeroCrossings = (float) ft.mZeroCrossings;
	out_fv.mDynamicRange = ft.mDynamicRange;
	out_fv.mPeakIndex = ft.mPeakIndex;
	out_fv.mTempo = ft.mTempo;
	out_fv.mTimeSignature = ft.mTimeSignature;
	out_fv.mBassPitch = ft.mBassPitch;
	out_fv.mBassNote = (float) ft.mBassNote;
	out_fv.mBassDynamicity = ft.mBassDynamicity;
	out_fv.mStereoWidth = ft.mStereoWidth;
	out_fv.mSurroundDepth = ft.mSurroundDepth;
	out_fv.mCenterDistinction = ft.mCenterDistinction;
	for (unsigned i = 0; i < 4; i++)
		out_fv.mBandSpectrum[i] = ft.mBandSpectrum[i];
	out_fv.mBandSpectrum[4] = ft.mBandSpectrum[0];
	out_fv.mSpectralCentroid = ft.mSpectralCentroid;
	out_fv.mSpectralSlope = ft.mSpectralSlope;
	out_fv.mSpectralVariety = ft.mSpectralVariety;
	out_fv.mSTrackBirths = ft.mSTrackBirths;
	out_fv.mSTrackDeaths = ft.mSTrackDeaths;
	out_fv.mLPCResidual = ft.mLPCResidual;
	out_fv.mLPCPitch = ft.mLPCPitch;
	out_fv.mLTrackBirths = ft.mLTrackBirths;
	out_fv.mLTrackDeaths = ft.mLTrackDeaths;

	size_t num_written = fwrite( & out_fv, sizeof(FeatureTableStruct), 1, output);
	if (num_written == 1)
		return TRUE;
	else
		return FALSE;
}
```

Appendix C. FMAK Database Setup and Management

Installing and using Postgresql

See the excellent step-by-step instructions at

http://developer.apple.com/internet/opensource/postgres.html

Setup commands after installing postgresql

```
sudo mkdir /usr/local/pgsql/data
chown postgres /usr/local/pgsql/data
sudo -u postgres initdb -D /usr/local/pgsql/data
sudo -u postgres createdb FMAK
sudo -u postgres createuser -d -a -P fmak
    use password "KAMF"
```

Load the table definitions file (in this directory) into the DB

```
psql -d FMAK -f FMAK_Tables.sql
```

To use the stored-procedures you also have to create the plpgsql language:

```
sudo -u postgres createlang -d FMAK plpgsql
```

Testing

After running some analysis, you can look at the DB contents with the commands

```
psql -d FMAK
```

and type SQL

```
select * from FeatureTables;
```

or

```
select * from FeatureCollections;
```

type ctrl-d to exit pgsql

Maintenance

For routine maintenance

```
vacuumdb -d FMAK
```

For backing up the database

```
cd /usr/local/pgsql
sudo tar cvfz ~/db_backup.tgz data
```