## 1 Introduction

There are 3 data files for ORCV index, as opposed to 1 in the demo version:

* The header, with the number of vectors and the various options (n changes)
* Centroid normal values for each centroid (static)
* The vectors, one entry per vector (append to add new vectors)

The header and centroid normal are created during training, and the first version, with n=0 and a 0 length vector file, is known as the skeleton index.

## Index Header

This is an orcvhdr\_t structure, as follows:

**typedef struct orcvhdr {**

**uint32\_t n; // number of vectors (changes)**

**uint32\_t nc; // number of centroids (fixed)**

**uint32\_t code\_size; // code size in PQ format**

**uint32\_t code\_bytes; // code size in bytes**

**uint32\_t d; // vector dimensions**

**uint32\_t M; // seach index internal**

**uint32\_t efConstruction; // search index internal**

**float32\_t dmatch; // distance for vector match**

**float32\_t dnear; // distance for vector near match**

**uint8\_t do\_opq; // if vector rotation matrix is used**

**} orcvdhr\_t;**

After training this file must be created before an index service can be started. These files can easily be created from a variant of the test\_ … utilities, as most things are from Parser.h/opt.

## Centroid Normal

To understand this, look at index->write() and when it is called in the test\_ utility set.

**// Save centroid norms**

**write\_vector(output, centroid\_norms);**

**std::vector<float> centroid\_norms; ///< L2 square norms of coarse centroids**

This file is in vector format, which means a 4 byte length (hdr->nc), followed by that many floats, one per centroid.

## Vector File

There are various components for each vector:

* 8 byte EID, the global application ID for this vector
* 4 byte centroid ID
* Vector in PQ byte form (2 bytes for PQ length = 16)
* Vector Norm in PQ byte form (1 byte)

Note that there is no internal vector id as such. This file is an array, and the vector ID is the array index, which is just the file offset.

## Home Directory

The environment for an index service, includes a listen port, and a home directory. Inside the home directory, there are a set of files:

**sprintf(env.idxpath, "%s/idx.vec", env.home);**

**sprintf(env.hdrpath, "%s/hdr.vec", env.home);**

**sprintf(env.centidpath, "%s/centid.vec", env.home);**

**sprintf(env.cnormspath, "%s/cnorms.vec", env.home);**

**sprintf(env.pqpath, "%s/pq.vec", env.home);**

**sprintf(env.opqpath, "%s/opq.vec", env.home);**

**sprintf(env.normpqpath, "%s/normpq.vec", env.home);**

**sprintf(env.centpath, "%s/cent.vec", env.home);**

**sprintf(env.edgepath, "%s/edge.vec", env.home);**

**sprintf(env.nodepath, "%s/node.vec", env.home);**

Note that we probably want to make the file extension suffix exactly match the type of utility required to read or write.

With the exception of the index components defined above, all of the remaining files exist within the benchmarks themselves, and thus become part of the skeleton index file set. ORCV is merely standardizing the path names, instead of using command line options, but clearly, a test program can stuff whatever it wants into the orcv environment.