WFWG IFTB Notes

Skef Iterum

July 25, 2023

Hello everyone -

Today I'm talking about some Adobe work I've alluded to before, which might serve as a replacement for the range request specification. The current system is a prototype and definitely not "finished", although the part that would be part of a specification is fairly close.

It's easiest to explain this system in relation to the three other subsetting systems it draws ideas from. So I'll start with a quick review of static subsetting, the range request concept, and then just some of the basic, lower-level technology involved in subsetting a font.

Then I'll describe the file formats and client side of this new system, both of which are quite simple. And then, after a comparsion of IFTB with other options, I'll discuss the endoder and what remains to be done in that area. I'll end with a bit of miscellany and pointers to repositories and branches.

Before I go on want to thank Garret for his consultation over the past week. He was particularly helpful resolving some of my mistaken impressions of how range request IFT was supposed to work, and his input improved many sections of this talk.

1 Static subsetting

Static subsetting is one means of achieving something like incremental font transfer. The idea is to split a font into a collection of subset files, each of which contains the glyphs needed to render a particular subset of unicode codepoints.

In order to use those files a client also needs a mapping from each relevant code point to its corresponding subset file. On the web this mapping is provided using the unicode-range property in CSS. A browser will determine which files are needed to render a page or content, and download each file not already contained in its cache.

In order for static subsetting to be efficient, the mapping must take into account which glyphs are likely to be needed together. By grouping glyphs that tend to be used together into the same subsets, one reduces the number of subsets that need to be downloaded in *typical* cases.

So, three notes about this system, for future reference:

1. Both the glyph path data and the shaping data are subset, with corresponding reductions in file size.
2. All processing is done up-front, resulting in normal font files which can then be compressed at the highest levels. There is therefore no "server side" beyond some means of allowing the files to be downloaded. As a result the system is highly compatible with CDN caching.
3. The primary disadvantage of static subsetting is that any shaping data that applies *between* subsets is discarded. This fragments the font on the client side so that the overall behavior is different than using the original font. Another disadvantage is that data downloaded is less specific, and therefore larger, compared with a subset specific to a given page or content. And the path data for some glyphs will be downloaded multiple times.

2 Range request

The idea of range request is to use that HTTP technology to download specific byte ranges from a font rather than the whole file. In practice this will mean downloading all, or almost all, parts of the file other than the `glyf` (and possibly `gvar`) tables, or all but the CharStrings data in the case of a `CFF` font. After that, the client would use the shaping information together with `cmap` to determine which glyph paths are needed to render a page, and those would also be retrieved using range request, with the ranges determined by values in the `loca` table, subtables of `gvar`, or the `CFF` CharStrings index.

The font file made available for this kind of incremental transfer would generally be pre-processed to lower the total number of glyph path byte ranges to be requested. This pre-processing would use relative frequency data similar to that used for static subsetting, but the efficiency of the overall system is much less tied to that data: having more ranges to download has a modest effect on the total number of bytes uploaded and downloaded.

There are a number of things I want to say about the range request concept. I'll start with comparisons to static subsetting.

1. With range request, only glyph path data is subset. The full amount of shaping data is always downloaded.

2. Because the subsetting logic is on the client side, the font on the server side must either be uncompressed, or would need to be uncompressed by the *server* in order to get the right ranges. (Starting uncompressed is far preferable.) The data transferred can still be compressed by the server, but only dynamically and therefore at accordingly lower compression levels

3. Because all shaping data and all needed glyph paths are downloaded, the system does not suffer from the behavioral fragmentation of static subsetting.

4. Although range request *could* be compatible with `glyf`-table composite glyphs, this would substantially increase the needed number of round-trip requests needed. Composites are therefore prohibited in the current specification. (The specification also prohibits `CFF` subroutines, perhaps by analogy. This makes *some* sense given the overall goal of incremental transfer, but the complete ban seems unwarranted to me. If we do move forward with range request we should soften that.)

3 Concerns with Range Request

I have three primary worries about to range request *as specified*. The first concerns difficulties of implementation. With this system the steps are:

1. Download the shaping data.

2. Feed the content to be rendered through the local shaping engine to determine the full set of glyphs needed. (Note that shaping itself can sometimes be expensive, and that cost must be paid before the download of glyph data starts.)

3. Download the data for the needed glyphs.

4. Use that data to finish the rendering process.

For this to work the shaping engine must be altered to work with a partial font file, and then either modified to store state during the gap between shaping and rendering or shaping must be run twice. This may be straightforward for systems that already use HarfBuzz has the shaping engine, but other systems could require significant restructuring. It is also different work than is needed to support the patch-subset specification.

The second worry is the number of round-trips needed. It sounds to me like a minimum of 3 are needed given current technology: One for the shaping data, one for glyph path data, and oene before that for the CORS pre-flight that must precede a requests for multiple ranges. And if the font has a gvar table that will almost always add *another* round trip.

(Just to clarify: I think there's been some suggestion that the CORS preflight might be rolled into the initial request. I don't think that's true in the narrow sense — that request will either itself be for multiple ranges, which would need its own pre-flight, or would be for a single range, which would *not* need a pre-flight but also *not* avoid the need for a later one.
Still, perhaps the initial request and a separate preflight preparing for a later multi-range-request could be made in parallel – I can see how that might work.)

However, even getting down to the minimum two or three serial requests involves choosing between of two imperfect strategies. One is to guess a length of the initial file that will include all of the shaping data but not too much of the glyph data. The other is to start loading the font data while monitoring the stream as it comes in, and then to close the connection when all of the data needed has arrived. This second idea sounds attractive but the IFT specification repository has an issue filed — seventy-four — outlining the problems with it.

So all in all, I would say that the technology is complex to adopt and has some significant performance drawbacks.

4 Subsetting technology

I also want to briefly discuss the lower-level process of subsetting a font so that I can establish a bit of terminology. Subsetting is often a two stage process:

1. The first stage takes a set of codepoints and layout features and determines the set of glyphs needed to accurately render those codepoints and features.

2. The second stage goes back through the various tables, including shaping data and the tables where glyph path data is stored, and reduces their contents so that only glyphs in the first stage list are mentioned.

Note that parts of this second stage are optional. It is possible to "retain" GIDs by inserting blank (or minimal) entries for omitted glyphs in the glyph path tables. That gives one the option of leaving the shaping and compositing data unmodified.

However, I bring all this up in order to highlight the *first* stage, and the function — in something like the mathematical sense of "function" — that it implies. This is a map from the input parameters (a set of Unicode codepoints and a set of layout features) to a set of glyphs.

This is a simple and familiar enough idea. However, I'm not aware of a specific term for it. As I'll need that term later, I'm going to call it the "glyph closure function", and call the set of glyphs mapped by a particular combination of codepoints and features the "glyph closure" of that combination.

5 Binned Incremental Font Transfer

With this background I can now talk about this new system, which I've been calling "Binned Incremental Font Transfer" or "IFTB".

I'll start with a brief description of the two IFTB font file formats. Conveniently, these are almost identical to the already existing formats they correspond to: OTF (uncompressed) and WOFF2 (compressed). The only differences between the IFTB formats and their OpenType counterparts are:

1. The new formats have sparse glyph path data, with some empty (with `glyf`, `gvar`, or `CFF2`) or minimal (with `CFF`) entries for some glyphs. The other tables are taken directly from the font, or perhaps unpacked and repacked for purposes of optimization.

2. There is an extra "IFTB" table containing various data, including a mapping from input parameters (which as usual are a set of unicode codepoints and a set of layout features) to a set of bin indices. I'll talk more about bins in a moment.

3. The OTF version of the file is “IFTB” rather than “OTTO” or `0x00010000`, in order to avoid accidentally using one of these files without appropriate client-side processing.
4. There are some additional requirements on (and suggestions about) table and subtable ordering.

Note that as with the OTF/TTF and WOFF2 formats, you can move between the two IFTB font formats using `woff2_decompress` and (a very slightly modified, but still entirely conforming version of) `woff2_compress`. Accordingly, I have been using the suffixes `_iftb.otf`, `_iftb.ttf`, and `_iftb.woff2` for these files.

6 Bins

Now, about bins:

Each GID in an IFTB-encoded font is assigned to one or more bins. There is an array of bin indices in the IFTB table indexed by GID that encodes most of that mapping. (So that bin *must* include that glyph; but other bins *might also* include it.)

There is also a set of per-layout-feature IFTB subtables that map bins to bins. (The specifics of that bin-to-bin mapping are a little in flux right now, at least in my head, but I’ll talk about it in general terms in a moment.)

It might be easiest to think of a bin as analogous to the set of glyphs in a static subset: Some glyphs are included because they correspond to unicode codepoints that tend to get used together, and others are included because they are substituted for the first sort under certain conditions of shaping or “compositing” (narrowly in the sense of composite TTF glyphs, or more loosely as with references in `COLR` substables).

Note that bin index `0` is special: this indicates the bin associated with the initial IFTB encoded font. The path data for glyphs in bin `0` is always present in any valid copy or augmentation of the font.

7 Chunks

Each bin other than 0 has an associated chunk file. As with the font files, there are two formats for a chunk: The uncompressed format starts with tag "IFTC" and the compressed format starts with tag "IFTZ". Other than that, the only difference between the two formats is that all content of an IFTZ chunk file after a short header is Brotli-compressed.

A Chunk file contains glyph path data for every glyph assigned to its corresponding bin. There can be one or two subtables of path data: two in the case of a variable TTF font with both glyf and gvar entries, one otherwise. Information in the header indicates which GIDs correspond to which path data entries as well as the destination table or tables.

8 Client-side operation

Client-side operation with IFTB is quite simple compared with the range request system:

1. In the first stage the client downloads the initial IFTB-encoded file. This will presumably be the woff2-like format in almost all cases. The type of file can be recognized in the OTF version field of the woff2 header (or by the first four bytes of the uncompressed file).

2. Next the client takes its set of needed codepoints and maps these to GIDs using the `cmap` table in the downloaded font file. `cmap` must include either a format 12 or format 4 subtable. If both are present the format 12 subtable must be used.

3. The client then computes, in several stages, a set of bin indexes to download.

   First it maps each GID added in the last step to a chunk index using the array in the IFTB table and adds that to a "chunk set".

   Then it sorts its list of needed layout feature tags and looks for each in the list of feature-specific IFTB subtables. When a tag is present, it uses the chunk-to-chunk mapping to add further chunk indices to the set. (Note that there is no need to go through the feature-specific subtables to compute a "chunk closure" — the chunk list only needs to be processed once per feature as long as the feature tags are examined in order.)

Not every layout feature available in the font will have a subtable. Typically only larger, optional features will have extra mappings. The encoder chooses whether a given layout feature will be encoded separately.)

In the last stage the client takes the list of needed bin indices and subtracts the list of bins that have already been added to the font. (That set has its own bitmap in the IFTB table.)

4. The list of needed chunks now in hand, the client initiates the download of each corresponding chunk file. The IFTB table also contains a relative URI "template" for chunk files, stored as a string. This template is filled in with characters corresponding to digits of the bin index encoded as hexidecimal, producing a relative URI for a specific chunk. That URI is then made absolute relative to the initial font's URI, providing the download URI.

5. As downloaded chunks arrive they are uncompressed and the glyph data they contain is staged in an intermediate data-structure. When the last chunk is unpacked that structure is used to add the glyph data to the decompressed font image.

6. Then the chunk bitmap in the IFTB table is updated to reflect the bins that have been integrated.

7. Finally, the OTF version is set to "OTTO" or 0x00010000, the sfnt-header checksums are updated, and the font is ready for use.

Augmentation proceeds pretty much the same way, except of course that the file has already been downloaded.

9 Comparisons

Before moving on I'll point out some the features and aspects of performance of IFTB relative to the other systems.

The compression *level* with IFTB is the same as with static subsetting, and better than that of patch-subset or range request, because the files are compressed in advance. Note, however, the impact of that level is offset by the fact that IFTB data is compressed into separate bundles while the data for those systems is compressed as a single unit. (Patch-subset has the additional advantage that it uses the previous file as a Brotli "dictionary" for the next file.)

The glyph "granularity" of IFTB is similar to that of static subsetting and can be similarly tuned. This is coarser grained than the other systems because the encoding is not specific to a parameter set, so a client will typically download data for more glyphs compared with patch-subset or range request. The data for some glyphs may be downloaded multiple times.

In contrast with static subsetting, all shaping data from the original font is downloaded from the start, making that part of the system more similar to range request than the other technologies.

Finally, in terms of client-side *interface* IFTB is very close to patch-subset, which seems like a significant advantage. Supporting range-request would mean additional work over patch-subset, with calls needed in different contexts. Except for the very first stages (downloading the initial IFTB file or noting the CSS directive to use patch-subset) the paramterizations and upper-level interfaces of IFTB and patch-subset are close enough to be easily integrated into a single library. And the additional C++ code for IFTB beyond WOFF2 and Brotli decoders — which are already present in browsers — will probably add up to less than 2000 lines. (It is around 1000 now).

10 CDN Caching

As with static subsetting there is no server-side to the system other than a network file-store. It is therefore fully compatible with CDN caching and significantly more cachable than patch-subset, given that the chunk files are statically computed and independently loadable.

I should probably clarify what I mean when I use "compatible with caching" as a relative term. Any HTTP-based protocol can be compatible with CDN caching, broadly speaking, as long as it conforms to certain restrictions (GET requests (or QUERY requests soon), a URL limit length, etc.) What I'm getting at is the question of how likely it is that a needed content will be cached, which (to oversimplify) depends on how many different encoded files correspond to a source font file. With static subsetting this is the number of files initially created. With IFTB it is the number of chunks. With range request it depends on the patterns of bytes typically downloaded and how the CDN caches data for range requests. Of the systems discussed, patch-subset has the worst caching profile: The number of initial files corresponds to the distinct sets of input parameters, which is related to the number of distinct contents, and the number of augmentation files corresponds to distinct combinations of current and new parameters, which is related to the *browsing paths between* contents.

To again oversimplify, whether IFTB or patch-subset is a better choice for a given *client* will therefore generally depend on narrowest network path between the client and the “original” server or servers.

When the so-called “last mile” is narrow or congested, as with many phone networks, the client will generally be better off with patch-subset, as that minimizes both round-trips and the number of bits that need to be transmitted over that last link.

When the smallest link is earlier in the network, which choice makes more sense will likely depend on how widely a font is used (or, in some cases, whether a provider is taking extra steps to keep its files cached). Frequently loaded font or chunk files are likely to be cached, and there is likely to be a CDN cache at, or closely connected to, the start of the last mile. Clients that have a clear path to their CDN cache may be able to download many more bytes per second from it than from a more distant server, providing an advantage to IFTB (or Range Request, for that matter).

When the needed IFTB files are not cached, the CDN must retrieve them from the original server, which gives patch-subset the advantage in virtue of its generally smaller payload.

11 Shaping and rendering behavior

As I’ve described IFTB so far it probably sounds like the system evolved from static subsetting more than the other technologies, although that is not how we arrived at the design. Why, then, is this coming up now instead of in, say, 2018?

The answer has to do with an additional question, which is how IFTB is able to provide the same *behavior* as the original font, as long as the client-side chunk integration process is implemented correctly.

Note that:

- Patch-subset provides the same behavior by carefully calculating a self-consistent subset and building the relevant tables back up around it.

- Range-request provides the same behavior by tracing through the relevant font tables to arrive at the full list of needed glyphs (after downloading all the shaping data).
- And, as noted earlier, static subsetting simply *doesn't* provide the same behavior.

12 The IFTB encoder

We know that with IFTB all shaping data for the font is downloaded from the start, and can assume that each chunk file will be correctly encoded, so that it has the data for all GIDs associated with its bin. With that in mind, the answer about behavior comes in the form of a "closure requirement" on the encoding process. This is the core of the design:

- The set of glyphs contained in set of bins computed from the input parameters — that is, the codepoint and layout feature sets — must be a superset of the glyph closure for those parameters.

I hope it's self evident that meeting this requirement ensures that the processed font will have the same behavior as a font produced using patch-subset.

13 Encoding options

The requirement is one thing, meeting it is another. The encoder I have developed for this initial prototype is quite primitive, although it performs fairly well with a surprising number of existing CJK fonts. It is constructed around meeting the requirement, and it seems to do so correctly according to one randomization test that I've written and tried out with some fonts, and also in our live testing.

I'll talk more about how and when it performs poorly in a moment.

Here are some aspects of encoding:

1. When a glyph has a Unicode codepoint but is never the product of a substitution or composition, it can just be added to whatever bin makes sense for Codepoint grouping. This is the easiest case.

2. When a glyph has no codepoint and is only substituted under one circumstance, it can just be assigned to the bin of a glyph it substitutes for. (This is usually *the* glyph it substitutes for, but with a ligature substitution there can be multiple options.)

With other glyphs things are more complicated, and it is easier, for now, to talk about the options available to an encoder. Assume an encoder that starts by forming fine-grained "proto-bins" and then tries to resolve each circumstance in which the closure requirement will *not* be met. In doing so it has these three options:

1. Merge two proto-bins into one. If a glyph is "needed" in more than one bin, merging them together results in only needing it in one. The main disadvantages of merging are that bins can wind up too coarse-grained, or (probably more importantly) it can wind up containing too many unrelated codepoints, which works against the goal of grouping like-with-like.

2. Duplicate the glyph across multiple bins. This avoids the disadvantages of merging but can (and often will) increase the number of bits that need to be downloaded in a given scenario.

3. Put the glyph in bin 0. This avoids the disadvantages of the other two alternatives at the cost of increasing the size of the initial font file. Because many glyphs are rarely needed, adding those to the initial file wastes bandwidth and reduces performance for many users. However, bin 0 is an important fallback for *complex* cases: if some circumstance is just too complicated for an encoder to figure out, it can always punt by putting the glyph into bin 0.

14 The prototype encoder

My current prototype encoder *never* duplicates a glyph. It starts by forming proto-bins informed by glyph frequency data (provided in a config file) and does some chunk merging. Then it looks for cases that are at all complicated and dumps glyphs into bin 0 to meet the closure requirement. Indeed, the current version is *so* simple that "is substituted for another glyph and also has its own codepoint" counts as "complicated". These will generally wind up in bin 0.

As a result, fonts with more complicated composition or shaping patterns wind up with larger initial files, sometimes much larger.

I've only looked at the current problems briefly, but it appears that across existing CJK fonts the biggest sources of trouble are the `aalt` and `nalt` layout features, which can be extensive when a font designer specifies those thoroughly. Because these are rarely used in practice (at least in web contexts), this may allow for a sort of hack: just always duplicate the glyphs for those features in additional bins, perhaps even just one bin per feature to make things easy. The result will be a few extra chunks that are almost never downloaded, combined with getting those features out of the way of the rest of the encoding analysis. I have been meaning to add the logic for this but haven't had an opportunity.

15 A better encoder

Based on the performance of my very simple encoder, and an understanding of the three options (bin merging, glyph duplication, and bin 0), I am tentatively convinced that this system can work well in almost all relevant CJK applications and probably other applications (such as Emoji fonts).

And of course the specification itself only needs to refer to the closure requirement; it need not and probably should not explain how to meet it. So all that remains to write that specification (as far as I can tell) is to rethink some details of the IFTB layout feature chunk-to-chunk mapping, as I think my current design is too restrictive.

Of course, *arguing* for IFTB as replacement for range request and an alternative to patch-subset would be a lot easier with a good encoder already in hand. Even if we were all convinced that a better encoder is just a matter of putting in the work, we still face the problem of accurately estimating IFTB performance, as estimates based on the prototype encoder won't look great for some fonts.

On that subject, I want to be clear today that while I was given the opportunity to develop these ideas to this point, and I can continue to participate in discussions here and do more work on the specification side, this is not my current project and probably will not be my primary project in the short or mid term. In particular, I don't expect to be directly contributing to a better encoder implementation anytime very soon. I can talk about ideas, of course, and I do have some of those.

16 Preloading

One potential difficulty of the “chunk file” convention is the overhead of downloading chunk files, as in some cases many chunks might be needed. From our initial testing this doesn’t seem like a huge problem as long as the server implements HTTP 2. But it may be desirable to download fewer total files in the normal case.

When I described the client-side operation of the system I noted that the IFTB table contains a map of which chunks are already integrated. This is how we maintain state for augmentation, but nothing about the code restricts it to that case. If the initial file already comes with chunks 8 and 12 in addition to chunk 0, the client side will never download chunks 8 and 12.

This enables what I have termed “preloading”. In addition to the more or less “blank” initial file that results from IFTB encoding, one can also create initial files that already contain whatever chunks one desires. One obvious desirable possibility, which Adobe has experimented with, is an initial file with the chunks for all high-frequency codepoints of a script, such as Japanese. Creating that file is just a matter of doing on the server-side what is normally done on the client side, and then compressing the result. (Recall that an uncompressed IFTB font file is almost a TTF or OTF, and that a compressed IFTB font file is just a WOFF2 encoding of that TTF or OTF.)

Preloading decreases the total number of bytes that need to be downloaded, because any glyphs duplicated among the preloaded chunks will no longer be duplicated, and because it avoids the overhead of chunk headers. It also means that all preloaded data is compressed together, at the highest Brotli level, regaining some of the compression advantage relative to patch-subset.

The possibility of preloading may not look much different than just putting all those glyphs in bin 0 in the first place, and for single-language fonts it isn’t. When a font is designed for use with *multiple* scripts and/or languages, it is of course possible to IFTB-encode that font for each relevant language separately. However, the configuration for my prototype encoder also allows optimization for multiple scripts and/or languages simultaneously (within reason). One can then produce preloaded and compressed initial font files with the high-frequency bins corresponding to each language.

This saves server disk space, of course, in that the chunk set is common across all the languages. However, far more importantly for performance, the fact that requests from the users of the various scripts target the same set of chunk files, making them that much more cachable.

In any case, current statistics suggest that a thoroughly preloaded Japanese IFTB font file could render a significant percentage of contents without loading *any* additional chunks, reducing the needed number of round-trips to one. We expect the same will be true of other CJK languages.

17 Miscellany

Before moving on to the discussion, there are a few things I want to note about the current implementation and documentation in case anyone runs into them.

1. The current prototype actually has two means of retrieving chunks to choose between. One is downloading the chunk files as I described. The other is to download chunks from a "range file", which is a single file of all compressed chunks concatenated. The current draft IFTB table also contains an array of chunk offsets into that range file, similar to loca or a CFF index. The idea is that one could do a range request to get all chunk data from a single file rather than downloading separate files.

   I included this option because when I started the prototype I wasn't sure which mechanism would be best. At this point it looks like chunk files will work better. The main disadvantage of downloading many files is overhead, but HTTP 2 mostly resolves that problem. And it appears to be very unlikely one would run into a server that does a good job with multi-range requests but a poor job serving multiple files. (I could be wrong about this.) We have already discussed the CORS preflight problem with range request. The `multipart-byterange` format used for multi-range-requests is also a bit hokey, although I did write parser code for it that doesn't need to scan through the range data and it seems to work fine.

2. The GID to chunk map in the current IFTB table is just an array of chunk indexes. This is a design decision on my part that could be revisited.

   The main way of compressing the *whole* table would be to reorder it to put like chunks in sequence, but that means reordering GIDS, which would work against the reordering typically done to optimize `cmap` tables, which have somewhat different constraints.

Short of that extreme, one could only include those GIDS with unicode mappings, as those are the only entries that will normally be checked. I decided against this because a "sophisticated" client might want to add particular glyphs by GID for its own reasons, so it seemed best to provide a full mapping in the format.

3. The prototype must currently be linked against a slightly modified version of HarfBuzz. The modifications are at https://github.com/skef/harfbuzz/tree/chunkmods and are modest. Most have to do with enforcing IFTB table and subtable ordering requirements. I also added an interface to retrieve the offset to the CharStrings Index of a CFF table, for my convenience. (This branch hasn't been touched since February and is about a thousand commits behind the current HarfBuzz main branch, but rebasing shouldn't be difficult.)
4. The prototype must also be linked against a slightly modified version of the WOFF2 library. The modifications are at https://github.com/skef/woff2/tree/iftb_changes and are trivial. The existing library always added tables in tag order when encoding and I just needed an option to retain the ordering of the source file. The specification only imposes a few ordering restrictions and the library's standard decoder will happily decode my IFTB table ordering.

18 Resources

The prototype repository is https://github.com/adobe/binned-ift-reference. I got approval to use the W3C Software License for the public release.

The prototype handles the requirements with git subrepositories. For now it has a simple Makefile-driven build, because higher-level stuff can get confused when you're building against modified versions of common libraries. The encoder can be a bit "fragile" for that reason. The wasm code only includes standard bits of the WOFF2 and Brotli libraries and is not fragile.

The prototype also includes a browser demo with its own `README`. It is not sophisticated — it doesn't even augment — but it can probably be quickly adapted to match the current patch-subset demo. The WASM and Javascript are included so that they don't need to be compiled.

Ages ago I wrote up some documentation in a March fork of the IFT spec that predates the division into the two protocols. That branch is at https://github.com/skef/IFT/tree/iftb. This is meant much more as a set of notes than a serious attempt at a PR, especially in its current form. Still, I think it's fairly accurate relative to the prototype and explains things I haven't here, including the IFTB table and chunk file layouts.

19 Questions/Discussion