Chapter 5: Types and Dynamic Values

Bento provides a very powerful mechanism for transforming values during I/O, and for following indirect references. This chapter describes the way that types can be built to define such values, and explains how the library supports such types.

This entire chapter is new, so no change bars are used.

Usage Examples

The Bento type mechansims are probably best explained in terms of some usage examples.

External File

Suppose we would like to have a value that represents a file. When we do `CMWriteValueData` to the value, we want to actually perform I/O to the file.

The mechanism described in this chapter allows us to store a reference to the file in a value. When the value is `Used`, an I/O redirection is set up, without the application's being aware of it.

Note that this raises the thorny problem of platform-independent file references. Bento avoids this problem. It allows any number of different types of references, implemented by handers. Naturally, we intend to encourage definition of a standard platform independent file reference mechanism, but this is not required to use Bento.

Compressed Value

Suppose we would like to compress data as it is written to the value, and decompress it as it is read out. In addition to maintaining the data in the value itself, this compression may depend on a dictionary associated with the type of value. Furthermore, the compression routine may need to keep various state around, since the compression at any point may depend on what has already been written.

The mechanism described in this chapter allows us to give the value a type that causes the compression/decompression handler to be transparently invoked when the application does I/O. Again, this is an extensible mechanism, so that new compression algorithms (or more generally, arbitrary transformations) can be added without modifying the library.

Compressed, Format Converted Array

Suppose the value we are dealing with is actually an array of pixels. In addition to decompressing it, on a given platform we want to convert each pixel to a different format.

The mechanism described in this chapter allows us to take two (or more) data transformations, such as compression and format conversion, and compose them together. Just as the application does not need to be aware of the underlying transformations, the individual transformations do not need to be aware of each other.

All of the Above

Obviously, the next step is to put the compressed pixel array out in a file, and convert it to a different format when it is read in. This is all supported using exactly the same composition as used in the previous example. The interfaces to data transformations and I/O redirection are the same, so no special mechanism is required.

Stranger (But Still Useful) Examples

To briefly illustrate further where this leads, here are some more unusual examples:

- A value contains a query that is used to look information up in a database. The "I/O redirection" provides access to a table retrieved from the database.
- A value contains a file reference that is encrypted because it also holds the file-server password. A decryption stage is required before the I/O redirector can be applied to the file reference.
- A value contains a query that is used to generate a file reference, which then becomes the basis for a second level of I/O redirection.

But you get the idea.

Structured Types

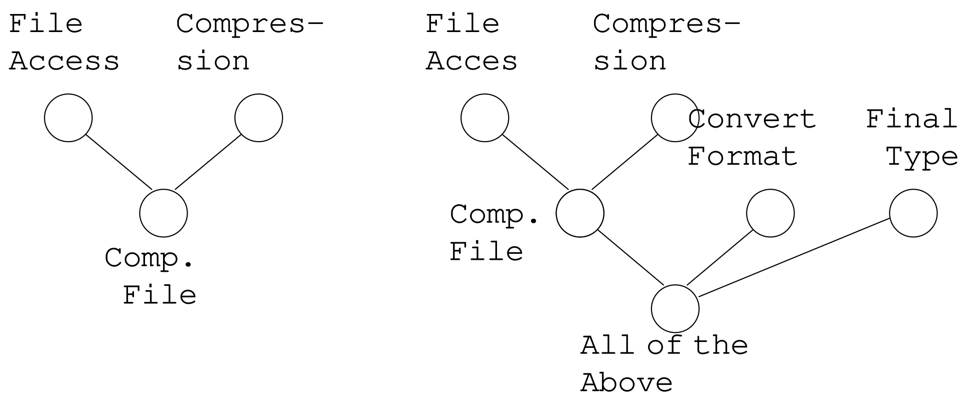
All of these examples are based on the types of the values involved. The examples depend on two aspects of Bento types.

First, every value handler is bound indirectly through the name of a type. Handlers are associated with type names through the `CMSetMetaHandler` operation. This association is session-wide. Then the handler is bound to a particular type in a given container through the name of that type. This binding is done when the container is opened.

Second, even in the simplest examples above, such as the value that is just an indirection to a file, or the the value that is just compressed, the value essentially has two types: the type visible to the application, which encodes the format of the data from the application's point of view, and the type used to find the appropriate handler for compression, I/O redirection, etc.

As the more complex examples show, these multiple types of a value need to be independent. This leads to a view of a value as having multiple, independent types. By analogy with C++ (an analogy we will explore in detail below) we call these "base types" of the value type. Base types can be added to and removed from any Bento type using the `CMAddBaseType()` and `CMRemoveBaseType()` operations documented in the API chapter.

Base types are normal types, and themselves may have base types. This could be useful, for example, when the combination of file access and decompression is used in a variety of different contexts. The two could be made base types of a new type, and then that new type could be used in various ways, including making it a base type of the "all of the above" type which adds format conversion. A picture may clarify this somewhat:



Base types will always form a tree rooted in the original type. If the same type is used as a base type in more than one place in the tree, the separate uses are treated as entirely separate types.

To understand how a given set of base types will behave, we must flatten them out into a linear string. This is done by performing a depth-first, post-order walk on the tree. Thus, in the case above, the final order is File Access, Compression, Comp. File, Convert Format, Final Type, All of the Above. Because Comp. File and All of the Above do not have handlers (let us assume) they will not have any effect on the value.

Dynamic Values

Now that we understand in a general way what Bento does with types, we need to look more closely at how this works. The remainder of this chapter is probably only of interest to people who want to know what it would take to write a value handler. You may understand some parts better if you read the appendix on handlers first.

The examples above indicate a number of constraints on our design. First of all, the application, and each handler, must always think that it is dealing with a "normal" value. Second, in several cases we saw that handlers might have a non-trivial amount of state to manage.

We address these constraints by giving each handler its own "private" value, called a **dynamic value**. Dynamic values are transient (ie. not persistent); they are created just to provide an environment for the handlers, and they are never written to the container, saved in the TOC, etc. However, they do have refnums and from the "outside" (ie. from any application code or handler code except the handler that "owns" them) they look exactly like normal values.

This implies that the normal value operations must be supported for each dynamic value. The operations are supported by the API passing each operation to the handler that “owns” the dynamic value. (Actually, a few operations are executed by the API; we will discuss those later.)

The following value operations can be supported by each value handler:

```
CMSize CMGetValueSize(CMValue value);

CMSize CMReadValueData(CMValue value,
                    CMPtr buffer,
                    CMCount offset,
                    CMSize maxSize);

void CMWriteValueData(CMValue value,
                    CMPtr buffer,
                    CMCount offset,
                    CMSize size);

void CMInsertValueData(CMValue value,
                    CMPtr buffer,
                    CMCount offset,
                    CMSize size);

void CMDeleteValueData(CMValue value,
                    CMCount offset,
                    CMSize size);

void CMGetValueInfo(CMValue value,
                    CMContainer *container,
                    CMObject *object,
                    CMProperty *property,
                    CMType *type,
                    CMGeneration *generation);

void CMSetValueType(CMValue value,
                    CMType type);

void CMSetValueGeneration(CMValue value,
                    CMGeneration generation);

void CMReleaseValue(CMValue);
```

When a dynamic value is spawned by `CMNewValue()` or `CMUseValue()`, the pointer to the top-most dynamic value header is returned as the refNum. Then, whenever the user passes a refnum to an API value routine, it checks to see if the refNum is a dynamic value. If it is, it initiates the call to the corresponding value handler. That may cause a search up the base value chain to look for the "inherited" value routine. In the limit, we end up using the original API value routine if no handler is supplied and we reach the "real" value in the chain. Thus the handler must be semantically identical to the corresponding API call.

These dynamic values only exist from creation during the `CMUseValue()` (discussed below) until until they are released by `CMReleaseValue()`. A dynamic value can have its own data, but this data is stored in the value's refCon rather than in the value data itself. Dynamic values do not have associated data in the normal sense.

Dynamic Value Creation

A dynamic value is created when a value is created by `CMNewValue()` or used by `CMUseValue()`, and the following two conditions occur:

1. The type or any of its base types have associated metahandlers registered by `CMSet-MetaHandler`().
2. The metahandlers support a Use Value Handler, and in addition for `CMNewValue()`, a New Value Handler.

The New Value Handlers are used to save initialization data for the Use Value Handlers. The Use Value Handlers are called to set up and return a refCon. Another metahandler address is also returned. This is used to get the address of the value operation handlers corresponding to the standard API CM... value routines mentioned above.

When a `CMNewValue()` or `CMUseValue()` is almost done, a check is made on the value's type, and all of its base types (if any) to see if it has an associated registered metahandler. If it does it is called with a Use value operation type to see if a Use Value Handler exists for the type. If it does, we spawn the dynamic value.

The spawning is done by calling the Use Value Handler. The Use Value Handler is expected to set up a refCon to pass among the value handlers and a pointer to another metahandler. These are returned to `CMNewValue()` or `CMUseValue()` which does the actual creation of the dynamic value. The extensions are initialized, the metahandler pointer and refCon are saved. The pointer to the created dynamic value header is what `CMNewValue()` or `CMUseValue()` returns to the user as the refNum.

Now, when the user attempts to do a value operation using this refNum, we will use the corresponding handler routine in its place. The vector entries are set on first use of a value operation. If a handler for a particular operation is not defined for a value, its "base value" is used to get the "inherited" handler. This continues up the chain of base values, up to the original "real" value that spawned the base values from the `CMNewValue()` or `CMUseValue()`. Once found, we save the handler in the top layer vector (associated with the refNum) so we don't have to do the search again. Thus, as in C++, dynamic values may be "subclassed" via their (base) types.

Note that if we indeed do have to search up the base value chain then we must save the dynamic value refNum (pointer) along with the handler address. This is very much like C++ classes, where inherited methods are called and the appropriate "this" must also be passed.

Layering Dynamic Values

The best way to describe layering is in terms of C++. Say we have the following class types (using a somewhat abbreviated notation):

```
class Layer1 {// a base class
<layer1 data>// possible data (fields)
Layer1(<layer1 args>);// constructor to init the data
other methods...// value operations in our case
 };

class Layer2 {// another base class
<layer2 data>// possible data (fields)
Layer2(<layer2 args>);// constructor to init the data
other methods...// value operations in our case
};

class T: Layer1, Layer2 {// the class of interest!
<T data>// possible data (fields)
T(<T args>, <layer1 args>, <layer2 args>);
                                    // constructor to init the data
                    and bases
other methods...// value operations in our case
};
```

In Container Manager terminology, T is to be a registered type with other registered types as base types (classes). All type objects are created using the standard API call `CMRegisterType()`. Base types can be added to a type by using `CMAddBaseType()`. This defines a form of inheritance like the C++ classes.

Type T would be registered with its base types as follows:

```
layer1 = CMRegisterType(container, "Layer1");
layer2 = CMRegisterType(container, "Layer2");

t = CMRegisterType(container, "T");
CMAddBaseType(t, layer1);
CMAddBaseType(t, layer2);
```

For the t object, the global name property and value are created as usual by `CMRegisterType(container, "T")`. The `CMAddBaseType()` calls add the base types. These are recorded as the object ID's for each base type in the order created as separate value segments for a special “base type” property belonging to the type object.

As mentioned above, `CMNewValue()` or `CMUseValue()` spawn dynamic values if the original type or any of its base types have an associated Use Value Handler. Assume that was done for "T" in the above example. What happens is that `CMNewValue()` or `CMUseValue()` will look at its type object (t here) to see if the base type property is present. If it is, it will follow each type "down" to leaf types using a depth-first search.

In the example, "layer1" will be visited, then "layer2", and finally the original type "T" itself. If the "layer1" type object had base types of its own, they would be visited before using "layer1" itself. Hence the depth-first search down to the leaf types.

For each type processed, if it has a Use Value Handler of its own, it will be called to get a refCon and value handler metahandler.

Note that this scheme allows total freedom for the user to mix types. For example, type T1 could have base types T2 and T3. Alternatively, T1 could just have base type T2 and T2 have T3 as its base type!

Data For Dynamic Values

In the C++ class types shown above, note that each class could have its own data along with its own constructor. The T class has a constructor that calls the constructors of all of its base classes. We can carry this analogy with the Container Manager just so far! Here is where it starts to break down.

The problem here is that C++ class types are declared statically. A C++ compiler can see all the base classes and can tell what data gets inherited and who goes with what class. In the Container Manager, all "classes" (i.e., our type objects) are created dynamically! So the problem is we need some way to tell what data "belongs" to what type.

The solution is yet another special handler, which returns a format specification-called metadata. The handler is the Metadata Handler whose address is determined by the Container Manager from the same metahandler that returns the New Value and Use Value Handler addresses.

Metadata is very similar to C `printf()` format descriptions, and is used for similar purposes. The next section will describe the metadata in detail. For now, it is sufficient to know that it tells `CMNewValue()` how to interpret its `"..."` parameters. The rest of this section will discuss how this is done to dynamically create data.

As with C++ classes, the data is created when a new value is created, i.e., with a `CMNewValue()` call. The data will be saved in the container, so `CMUseValue()` uses the type format descriptions to extract the data for each dynamic value layer.

`CMNewValue()` is defined as follows:

```
CMValue CMNewValue(CMObject object,
                   CMProperty property,
                   CMType type, ...);
```

The `"..."` is an arbitrary number of parameters used to create the data. Metadata, accessed from the Metadata Handler, tells `CMNewValue()` how to interpret the parameters just like a printf() format tells it how to use its arguments.

The **order** of the parameters is important! Because base types are done with a depth-first search through the types down to their leaves, the `CMNewValue()` `"..."` parameters **must** be ordered with the parameters for the first type in the chain occuring first in the parameter list. Note what's happening here is you are supplying all the constructor data just like T constructor class example above.

The way the data gets written is with a special handler, called the New Value Handler. After `CMNewValue()` calls the Metadata Handler, it uses the metadata to extract the next set of `CMNewValue()` `"..."` parameters. `CMNewValue()` then passes the parameters along in the form of a data packet to the New Value Handler. The New Value Handler is then expected to use this data, which it can extract with `CMScanData-Packet()` (see the handler appendix). Once it has the data, it can compute initialization values to **write** to its **base** value. It is the data written by the New Value Handler that the Use Value Handler will read to create its refCon.

Only `CMNewValue()` does this. The New Value Handler is only for new values, but the Use Value Handler is used by both `CMNewValue()` and `CMUseValue()`.

In the simplest case, with only one dynamic value, you can see that the data is written to the “real” value. Now if you layer another dynamic value on to this, the next chunk of data is written using that layer's base value and hence its handlers. The second layer will thus use the first layer’s handlers. That may or may not end up writing to the "real" value depending on the kind of layer it is. If it's some sort of I/O redirection handler (i.e., it reads and writes somewhere else), the second layer data will probably not go to the "real" value.

The Use Value Handler is called both for `CMNewValue()` and `CMUseValue()`. The Use Value Handler reads the data from its base value to create its refCon. If the user comes back the next day and does a `CMUseValue()`, only the Use Value Handler is called. Again it reads the data from its base value to construct the refCon and we're back as we were before in the `CMNewValue()` case.

Handler Contracts

It should be pointed out here that the Metadata and New Value Handlers will always be executed with a Container Manager running on some particular hardware (obviously). The data packet built from the `CMNewValue()` `"..."` parameters is stored as a function of the hardware implementation on which it is run (i.e., whatever the sizes are for bytes, words, longs, etc.). How it is stored is a function of the metadata returned from the Meta-data Handler. In other terms, the New Value Handler has a contract with both the Container Manager and the Metadata Handler on the meaning of the parameter data.

Note, however, it is **not** required that you be on the same hardware when you come back the next day and to the `CMUseValue()` that leads to the Use Value Handler call. The handler writer must keep this in mind. Specifically, the Use Value Handler **must** know the attributes (bytes size, big/little endian, etc.) of the data written out by the New Value Handler so it knows how to use that info. In other words, the Use Value Handler has a (separate) "contract" with its own New Value Handler on the meaning of the data written to the base value.

There is another, relatively minor, thing to keep in mind. That is that the value handlers for any one layer must take into account the size of its own data when manipulating additional data created by the handlers for `CMReadValueData()`, `CMWriteValueData()`, etc. This simply offsets the write and read value data operations by the proper amount. Remember all operations are on their base values. So if a New Value Handler writes data, this basically prefixes the "real" stuff being written by the handler operations.

Metadata

As mentioned above, the metadata directs CMNewValue() on how to interpret its "..." parameters to build data packets passed to New Value Handlers.

The format string is a sequence of characters containing data format specifications. Unlike `printf()`, anything other than the data format specifications are ignored. They are assumed to be comments.

The data format specifications indicate to `CMNewValue()` how to interpret its data initialization parameters. Each specification uses the next corresponding "..." parameter to `CMNewValue()`. This is similar to the behavior of `printf()`.

A data format specification begins with a "%" sign. Immediately following the % is a required data format descriptor, expressed as a sequence of characters. The data format descriptors are as follows (numbers in "[]" indicate notes following the format descriptor list):

`c`	A character or byte [1].
`d`	A short [1].
`l[d]`	A long (the "d" is optional) [1].
`[*]s`	A C string (i.e., null delimited). Optionally a "*" indicates that only the first n characters of the string are to be used. The "*" consumes an additional `CMNewValue()` "..." parameter of type `CMSize` [3, 4, 5].
`i`	An object, property, or type ID. Thus is defined as the same size as "ld" [1].
`p`	A pointer [1, 2].
`o`	A `CMObject`, `CMProperty`, or `CMType` object refNum. This is defined as the same size as "p" [1, 2].
`v`	A `CMValue` refNum. This is defined as the same size as "p" [1, 2].

<u>Notes:</u>

1. The `CMNewValue()` "..." parameters are converted to a packet of data using the hardware implementation defined sizes for bytes, words, longs, etc. as directed by the metadata returned from the Metadata Handler. Thus the Metadata Handler has a contract with the New Value Handler that `CMNewValue()` calls. The data that the New Value Handler writes to its base value is in terms of a "contract" it has with its Use Value Handler. It is the one that will read that base value data to create its refCon. If `CMUseValue()` is expected to be run on different hardware with different byte sizes, endianess, etc, then that is between the New Value Handler and its Use Value Handler. The Container Manager is independent of that.

2. Pointers can be passed to `CMNewValue()` to convey special information to the Use Value Handler. You shouldn't, of course, write these as data. RefNums can be passed to extract object ID's or other read value data. It is permissable to write object ID's to data. But this will put a restriction on such referenced objects that they shouldn't be moved or deleted.
3. For "`%*s`", the value corresponding to the "*" is copied to the packet data immediately in front of the string. This is somewhat (not quite) equivalent to "`%l%s`", where the `%l` is the length, n, and `%s` is a n byte string. Note however, this string is **not** null delimited.
4. Caution, the string will be **copied** from the string pointed to in the `CMNewValue()` **"..."** parameter list to the packet. It you intend to pass a pointer to the string, rather than the string itself, `%p` should be used. Frankly, `%s` will not be used much.
5. For symmetry `CMScanDataPacket()` returns the value of string length to an explict distinct parameter pointer. Thus the parameter pointer list passed to `CMScanDataPacket()` should be identical to the **"..."** parameters passed to a `CMNewValue()` **"..."** parameter list (at least the portion corresponding to this type).

The Metadata, New Value, and Use Value Handlers

The Metadata Handler is only needed for `CMNewValue()` so that the proper number of `CMNewValue()` **"..."** parameters can be placed into a data packet for the New Value Handler.

The Metadata Handler must have the following prototype:

```
CMMetaData metaData_Handler(CMType type);
```

where

`type` = the (base) type layer whose metadata is to be defined.

The Metadata Handler simply returns a C string containing the metadata using the format descriptions described above.

The type is passed as a convenience. It may or may not be needed. It is possible for a type object to contain **other** data for other properties. Types, after all, are ordinary objects. There is nothing prohibiting the creation of additional properties and their values. This fact could be used to add additional (static and private) information to a type to be used elsewhere. For example, the type could contain a compression dictionary.

Note, as in `printf()`, if the metadata handlers "lie" about the metadata format, or if there aren't enough parameters supplied to `CMNewValue()`, the results will be unpredictable!

The New Value Handler must have the following prototype:

```
CMBoolean newValue_Handler(CMValue baseValue,
                           CMType type,
                           CMDataPacket dataPacket);
```

where

`baseValue` = the base value which is to be used to write the refCon data for the Use Value Handler.

`type` = the type corresponding to this New Value Handler.

`dataPacket` = the pointer to the data packet, created from the `CMNewValue()` "..." parameters according the types metadata format description.

The type is passed again as a convenience just as in the Metadata Handler. It can also be used here to pass to `CMScanDataPacket()` to extract the dataPacket back into variables that exactly correspond to that portion of the `CMNewValue()` "..." parameters that correspond to the type. It is not required, however that `CMScanDataPacket()` be used.

The Use Value Handler is called for both the `CMUseValue()` and `CMNewValue()` cases. If its companion New Value Handler wrote data to its base value, the Use Value Handler will probably read the data to create its refCon. The refCon will be passed to all value handlers. The Use Value Handler returns its refCon along with another metahandler address that is used to get the value handler addresses. These are used to create the dynamic value.

The Use Value Handler should have the following prototype:

```
CMBoolean useValue_Handler(CMValue baseValue,
                           CMType type,
                           CMMetaHandler *metahandler,
                           CMRefCon *refCon);
```

where

`baseValue` = the base value which is to be used to write the refCon data for the Use Value Handler.

`type` = the type corresponding to this New Value Handler.

`metahandler` = a pointer to the value operations metahandler which is **returned** by the Use Value Handler to its caller.

`refCon` = a reference constant built by the Use Value Handler and **returned** to its caller.

The baseValue and type are identical to the ones passed to the New Value Handler. The type may or may not be needed in the Use Value Handler. Like the Use Value Handler, it could be used to supply additional information from other properties.

It is expected that the Use Value Handler will read data from its base value to construct its refCon. The refCon is then returned along with a pointer to another metahandler that is used by the Container Manager to get the addresses of the value operations.

Note, both the New Value and Use Value Handlers return a `CMBoolean` to indicate success or failure. Failure means (or it is assumed) that the handlers reported some kind of error condition or failure. As documented, error reporters are not supposed to return. But in case they do, we use the `CMBoolean` to know what happened. It should return 0 to indicate failure and nonzero for success.

Value Operation Handlers

The value operation routines can do a `CMGetValueRefCon()` on the value passed to get at the refCon set up by the Use Value Handler. This provides a communication path among the value handlers. Further, the value handler should usually do its operations in terms of their base value, which can be accessed using CMGetBaseValue().

There is one exception to this rule; the release handler. A set of one or more dynamic value layers are spawned as a result of a single `CMUseValue()` or `CMNewValue()`. The layers result from the specified type haveing base types. From the caller's point of view s/he is doing one `CMUseValue()` or `CMNewValue()` with no consideration of the base types. That implies that the returned dynamic value should have a single `CMRelaseValue()` done on it. The handlers have no business doing `CMReleaseValue()` on their base value. This is detected and treated as an error.

A count is kept by the Container Manager of every `CMUseValue()` and `CMNewValue()`. Calling `CMReleaseValue()` reduces this count by one. When the last release is done on the dynamic value (its count goes to 0), the release handler will be called. It is the Container Manager who calls the release handler for all the layers, not the handler. The Container Manager created them as a result of the original type; it is therefore responsible for releasing them.

The reason the Container Manager is so insistent on forcing a release for every use of a dynamic value is mainly to enforce cleanup. Most value operation handlers will, at a minimum, use a refCon that was memory allocated by the Use Value Handler. Release handlers are responsible for freeing that memory. In another example, if any files were open by the Use Value Handler, the releases would close those files.

If all a value operation does is get its base value and call back the API routine to do its operation (again except for the release handler), then what it is basically doing is invoking the “inherited” value operation. In this case, the value operation could be **omitted** entirely by having the metahandler return `NULL` when asked for that operation. The Container Manager uses that as the signal to search up the dynamic value inheritance chain to find the first metahandler that **does** define the operation. In the limit, it will end up using the original “real” value.

Possible Limitations On Value Operations

Value I/O operations are basically stream operations. That is, you read or write a chunk of stuff linearly from a specified offset. In addition, Bento provides insert and delete operations (`CMInsertValueData()` and `CMDeleteValueData()`).

Insert and delete can cause problems because base types may want to do certain transformations on their data that depend on what has occurred previously in that stream of data. For example, encryption using a cyclic key, or compression generally cannot be done simply by looking at a chunk of data starting at some random offset. A cyclic key encryption can be deterministic if you can always determine where to start in the key as a function of offset. But you can see that inserts and deletes will change the offsets of following data. You would not know where to start in the key.

What all this means is that certain data transformations only make sense if you are willing to refuse to support the insert/delete operations. Basically only data transformations that are position independent can be supported with the full set of value operations.

Even simple I/O to a file may create problems, since most file systems do not support inserts and deletes in the middle of a file. If you do want to support inserts and deletes, then you should consider the potential for data intensive and/or computationally intensive operations.