RawDatum - A Light-Weight In-Core Value-Semantic Type for Hypersheet and Datalayer

Rishi Wani
UI Infrastructure

March 28, 2011

Abstract

The current vocabulary type used by Datalayer and Hypersheet, `rdft::Value`, frequently becomes a CPU and memory bottleneck. This paper presents `RawDatum`, a replacement vocabulary type for use in Hypersheet and Datalayer.

1 Acknowledgement

Thanks to Paul Sader, who started this project under the name of `ftlval_value`.

2 Introduction

`rdft::Value` is the primary vocabulary type for Hypersheet and Datalayer. This (fully) value-semantic type is used as input and output in many Datalayer and Hypersheet functions. `Value` frequently dominates both the memory footprint and the runtime performance of Hypersheet and Datalayer due to its widespread usage. The following sections discuss in detail the implementation of `Value` and its limitations when used inside Hypersheet and Datalayer. In particular, we show how `RawDatum` which is an in-core value semantic type, overcomes these limitations without losing the current functionality of `Value`. We also introduce `Datum` which restores the full value-semantics of `Value`. Note that for exposition purposes only, all implementation sizes mentioned in this paper assume MSVC 2008 targetting a 32 bit platform.

3 Usage of Value

`Value` is used in the byte-code evaluation engine and worksheet inside Hypersheet. `Value` is also used in the data pipeline and template manager inside

Datalayer.

The byte-code evaluation engine inside Hypersheet evaluates the compiled byte-code. The result of this evaluation is a `Value` object. `Value` objects are also passed as arguments to functions inside the byte-code evaluation engine.

A worksheet inside Hypersheet is a two-dimensional object that models an Excel worksheet. Each worksheet uses two `Value` objects per cell to store its current and previous values.

A Datalayer pipeline consists of a model and one or more views, each of which are represented by a two-dimensional tables. Every cell inside the model holds a `Value` object. Each cell inside a view may hold a `Value` object too. The data flow between the model and the views, and among the views, is also in terms of `Value` objects.

The template manager inside datalayer applies different visual templates on the data inside the views and drives a two-dimensional grid, which displays this decorated data to the user. All the properties for a cell in a view are cached inside `Value` objects by the template manager.

4 Implementation of Value

`Value` is currently implemented using the `bdeut_Variant` class template instantiated on all the types supported by Hypersheet and Datalayer. Figure 1 shows some implementation details of `Value`.

```
class Error {
  int d_code;
  // ...
};

class Value {
  typedef bdeut_Variant<
    int,
    bsls_Types::Int64,
    double,
    bdet_DatetimeInterval,
    bdet_TimeTz,
    bdet_DatetimeTz,
    bdet_DateTz,
    bsl::string,
    bool,
    bsl::vector<Value>,
    Error
  > ValueType;

  ValueType d_data;
  // ...
};
```

Figure 1: Implementation details of Value

The size of a `Value` object is the size of the contained `d_data` data member, which is the sum of the sizes listed in Figure 2.

Description	**Bytes**
maximum of the sizes of all the types that may potentially be stored in `d_data`, which is the size of `bsl::string` and `bsl::vector<T>`	16
size of the discriminator to tell which type is currently held	1
size of pointer to an object derived from `bslma::Allocator`, which is required for any type that dynamically allocates memory	4
any padding added by the compiler to align the data	3
Total	**24**

Figure 2: Size of `Value`

The size of `Value` comes out to be 24 bytes on MSVC 2008. This is typically true for any 32-bit platform. The size of `Value` is approximately 32 bytes on a 64-bit platform. The size of `bsl::string` will increase to 32 bytes in the near future due to an optimization, in turn increasing the size of `Value` to 40 bytes.

5 Limitations of Value

The memory footprint of Datalayer and Hypersheet is dominated by `Value` objects and runtime instrumentation has shown that Hypersheet and Datalayer are the biggest memory bottlenecks in many Bloomberg applications. The vast majority of `Value` objects used in Hypersheet and Datalayer hold a `double`, an `int` or a `bool` value. The size of `Value` is proportionately much larger than the sizes of these types. Thus, a siginificant amount of memory is wasted in `Value` objects that store `double`, `int` or `bool` values. For example, a worksheet with 10,000 rows and 100 columns (which is not a rare scenario) has 1 million cells in it. If 90 percent of these cells hold `double` values, approximately 10 megabytes of memory is wasted in the 900,000 cells (considering that the size of `double` is 8 bytes).

The full value semantics of `Value` also creates performance problems. Hypersheet uses a stack based byte-code evaluation engine, where arrays of `Value` objects are frequently created and destroyed. The objects in these arrays do not need to be initialized. Despite the fact that the default constructed `Value` objects are never read in this case, we must pay for their initialization and subsequent destruction. The destruction cost is even higher than initialization, as the destructor cannot be inlined by the compiler. `Value` objects are copied extensively in the evaluation engine. The assignment of `Value` objects is a performance bottleneck, since `Value` objects cannot be copied trivially. Each assignment incurs the overhead of a function call, and may involve memory allocation and deallocation, as well.

Separately, `Value` does not have a trivial constructor (or destructor), so it cannot be used inside types such as unions, which impose such restrictions.

6 RawDatum

6.1 Requirements for RawDatum

`RawDatum` must satisfy the requirements shown in Figure 3.

(1) Represent the same values as `Value`

(2) Balance the smallest memory footprint against the cost of heap allocation for some types

(3) Be bitwise copyable

(4) Have trivial initialization, assignment, and destruction

Figure 3: Requirements for `RawDatum`

6.2 Implementation of RawDatum

In most applications, at least 80 percent of `Value` objects are of `double`, `int` or `bool` type; `RawDatum` objects must not require indirect (heap) allocation to represent values of these core types. `double` is the largest core type. Since its size is 8 bytes, this is the smallest possible size for `RawDatum`. `double` uses all of the 8 bytes, so we need a way to fit values of other core types within the 8 bytes and differentiate between a non-`double` value and a `double` value. Let us first examine the structure of an IEEE `double` value.

IEEE 754 `double` format uses 1 bit for storing sign, 11 bits for storing exponent and the remaining 52 bits for storing fraction part of a double-precision floating point number. Figure 4 shows the IEEE 754 double format.

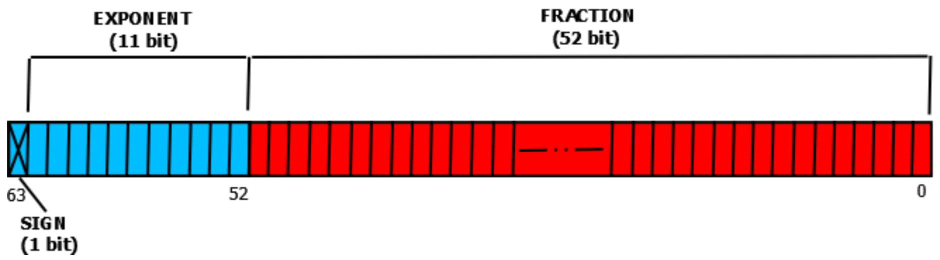


Figure 4: IEEE 754 double format

On a big-endian machine, the addresses of the bytes containing the sign and exponent bits are lower than the addresses of the bytes containing the fraction(mantissa) bits; while on a little-endian machine, the sign and exponent bits are in the higher address space than the fraction bits.

The IEEE Standard for Floating-Point Arithmetic (IEEE 754) states that if all bits in the exponent part of a `double` value are 1, then the value is one of the following special `double` values.

- Infinity - fraction part is 0
- Negative Infinity - fraction part is 0 and sign bit is 1
- NaN - fraction part is a non-zero number
- Negative NaN - fraction part is a non-zero number and sign bit is 1

Figure 5 shows special `double` values on two of the compilers that we use. As shown in figure, each platform sets just one specific bit in the fraction part of a `double` to indicate that the value is a NaN (apart from setting all exponent bits to 1). When the NaN bit is set to 0, then the double value is infinity. This special NaN bit is part of the byte that also contains 4 bits of the exponent. The other bits of the fraction part are not used for representing a NaN or an infinity value.

	EXPONENT			FRACTION					
NIBBLES	**63-60**	**59-56**	**55-52**	**51-48**	**47 - - 16**	**15-12**	**11-8**	**7-4**	**3-0**
QNaN	7	f	f	8	0 - - 0	0	0	0	0
SNaN	7	f	f	4	0 - - 0	0	0	0	0
INF	7	f	f	0	0 - - 0	0	0	0	0
-INF	f	f	f	0	0 - - 0	0	0	0	0

(a) Special double values on GNU C++

	FRACTION					EXP	FRA	EXP	
NIBBLES	**63-60**	**59-56**	**55-52**	**51-48**	**47 - - 16**	**15-12**	**11-8**	**7-4**	**3-0**
QNaN	0	0	0	0	0 - - 0	f	8	7	f
SNaN	0	1	0	0	0 - - 0	f	8	7	f
INF	0	0	0	0	0 - - 0	f	0	7	f
-INF	0	0	0	0	0 - - 0	f	0	f	f
IND	0	0	0	0	0 - - 0	f	8	f	f

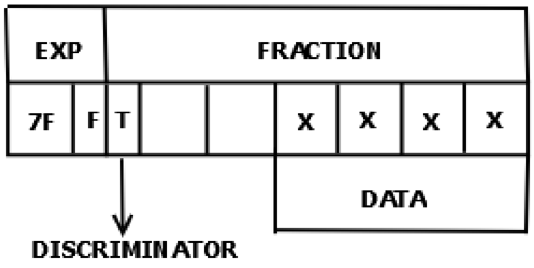
(b) Special double values on MS Visual Studio

Figure 5: Special double values on different compilers

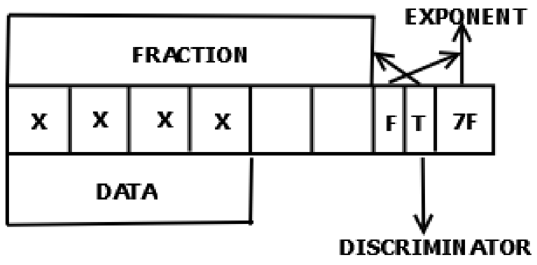
It is thus possible to store data in the rest of the bits of the fraction part and differentiate it from a regular NaN or an infinity value using a discriminator. Note that, MS Visual Studio has a special indeterminate (IND) value for representing a NaN resulting from dividing 0 by 0.

`RawDatum` uses 4 bits in the fraction part to differentiate between a special `double` value and a non-`double` value and also among the non-`double` values. Thus, apart from NaN and infinity double values, 14 different types can be represented using the 4 bit discriminator, which is sufficient to represent all the types that `Value` supports.

We have 6 bytes available to store data, allowing us to store at most 4-byte-aligned data in the fraction part. Values of 4-byte-aligned fundamental types like `int` and `bool`, and the user-defined type `Error` (which is just a wrapper around an `int` error code). Values of larger fixed length types such as `bsls_PlatformUtil::Int64`, `bdet_Date`, `bdet_Time`, `bdet_Datetime` and `bdet_DatetimeInterval` are allocated externally and held by pointer. Variable length types like arrays of objects and character strings may also allocated externally. Figure 6 shows the bits used for storage in a `RawDatum` object on different platforms.



(a) Double bits used for storage on a big endian machine

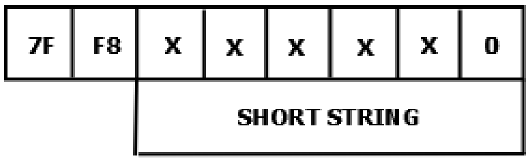


(b) Double bits used for storage on a little endian machine

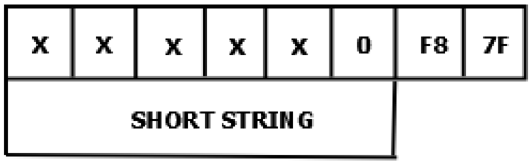
Figure 6: Double bits used for storage on different platforms

6.3 Short String Optimization

Strings with a short length are commonly used inside Hypersheet and Datalayer applications. One example is the security mnemonics that are 3 or 4 characters in length. Such strings can be stored inline instead of allocating them externally. Since character strings do not require alignment, the extra 2 bytes in the fraction part can be used along with the already available 4 bytes to store null-terminated strings with 6 or fewer characters in length directly in the `RawDatum` object. The exact bytes used for storage depend on the endianness of the platform as shown in Figure 7.



(a) Short-string optimization on a big endian machine



(b) Short-string optimization on a little endian machine

Figure 7: Short-string optimization on different platforms

6.4 RawDatumFactory

`RawDatum` has only trivial constructors. Hence factory methods are needed to create `RawDatum` objects. We also need factory method to destroy `RawDatum` objects as `RawDatum` has a trivial destructor and objects of some types stored inside `RawDatum` may contain externally allocated memory. `RawDatum` objects are not deep copied by the default copy-assignment operator. Hence factory method is needed to deep-copy `RawDatum` objects. The utility 'class' `RawDatumFactory` provides static methods for creating and destroying `RawDatum` objects, allocating and deallocating memory used in `RawDatum` objects as required with the help of an externally supplied allocator. `RawDatumFactory` also provides a method for creating deep copies of `RawDatum` objects.

7 Core Implementation

Figure 8 shows the core definition of `RawDatum` on MS Visual Studio 2008. Note that asserts and other defensive checks are omitted from the code samples for clarity.

```
class RawDatum {
  static const unsigned int EXPONENT_OFFSET    = 6;
  static const unsigned int DATA_OFFSET        = 0;
  static const unsigned int SHORTSTRING_OFFSET = 0;
  static const unsigned int SHORT_OFFSET       = 4;

  union {
    unsigned char d_data[8];
    double        d_dummy;
  };

  enum DataType {
    // primary discriminator
      INF                = 0  // +/- infinity value
      //..
    , SHORTSTRING        = 4  // short string value
    , STRING             = 5  // string value
    , ARRAY              = 6  // array value
    , DATE               = 7  // date value
      //...
    , INTEGER            = 11 // integer value
    , INTEGER64          = 12 // 64-bit integer value
    , CLONEABLE          = 13 // object implementing the
                              // dlcp::Cloneable protocol
    , USERDEFINED        = 14 // pointer to a user defined object (void *)
    , EXTENDED           = 15 // other types that cannot be discriminated
                              // using the existing 4 bit discriminator
    , REAL               = 16 // double value
      //...
  };

  enum ExtendedType {
    // secondary discriminator for additional types
    NAN2 = 0 // NaN double value
    // ..
  };

  //...
};
```

Figure 8: RawDatum on MS Visual Studio 2008

The `d_data` member in `RawDatum` is an array of 8 `unsigned char` values. Any `double` value (including NaN and infinity values) can be stored inside `d_data`. Values that are not of type `double` are stored inline or by pointer. The bits in `d_data` that correspond to the exponent part of a `double` value are set to 1, with the discriminator indicating the type of value stored. Note that this approach will not work on 64 bit platform because pointers are 8 bytes instead of 4 bytes in size. We will have a separate implementation for 64 bit platform.

Figure 9 shows a part of the public interface of `RawDatumFactory`.

```
class RawDatumFactory {
  static const unsigned short DOUBLE_MASK = 0x7ff0U;
  static const unsigned int MASK_OFFSET = 4;
  static const unsigned long long NAN_BIT = 0x0008000000000000;
public:
  // ..
  static RawDatum createInteger(int value);
  static RawDatum createDouble(double value);
  static RawDatum createString(const char      *value,
                               bslma_Allocator *basicAllocator);
  static RawDatum createArray(const RawDatum  *values,
                              int              len,
                              bslma_Allocator *basicAllocator);
  static RawDatum createUninitializedArray(
                                       RawDatum       **result,
                                       int              len,
                                       bslma_Alocator  *basicAllocator);
    static RawDatum createDate(const bdet_Date& value);
    static RawDatum createDatetime(
                               const bdet_Datetime&  value,
                               bslma_Allocator      *basicAllocator);
    static RawDatum createCloneable(dlcp::Cloneable *data, int type);
    static RawDatum createUdt(void *data, int type);
    // ..
    static RawDatum copyRawDatum(const RawDatum&      value,
                                    bslma_Allocator *basicAllocator);
    static void destroyRawDatum(const RawDatum&      rawdatum,
                                   bslma_Allocator *basicAllocator);
};
```

Figure 9: Public interface of RawDatumFactory

`RawDatumFactory` has static methods to create, copy and destroy various types of `RawDatum` objects.

Figure 10 shows how a `RawDatum` object having an integer value is created.

```
RawDatum RawDatumFactory::createInteger(int value)
{
  BSLMF_ASSERT(8 == sizeof(RawDatum));
  BSLMF_ASSERT(4 >= sizeof(int));

  RawDatum result;
  static const unsigned int MASK = (DOUBLE_MASK | INTEGER) * 0x10000;
  new (result.d_data + MASK_OFFSET) unsigned int(MASK);
  new (result.d_data + DATA_OFFSET) int(value);
  return result;
}
```

Figure 10: Creating a RawDatum having an integer value

An integer value is representative of all the 4 byte sized types of values that can be stored directly inside a `RawDatum` object.

Figure 11 shows how a `RawDatum` object having a double value is created.

```
RawDatum RawDatumFactory::createDouble(double value)
{
  BSLMF_ASSERT(sizeof(double) == sizeof(RawDatum));

  RawDatum result;
  if (!(value == value)) {
    static const unsigned short MASK = DOUBLE_MASK | EXTENDED;
    new (result.d_data + EXPONENT_OFFSET) unsigned short(MASK);
    // Copy the type of the extended type into the remaining two bytes.
    new (result.d_data + SHORT_OFFSET) short(NAN2);
    // Initialize the remaining uninitialized bytes.
    new (result.d_data + DATA_OFFSET) int(0);
  }
  else {
    new (result.d_data) double(value);
  }
  return result;
}
```

Figure 11: Creating a RawDatum having a double value

A double value is just copied into the 8 byte array. If the double value is a Infinity value, the bits corresponding to the discriminator part in the value will match the discriminator for infinity. If the double value is a NaN value, the discriminator is set to 'EXTENDED' and the remaining two bytes are used to indicate that it is a NaN value. The NaN value is not directly stored in the array. Only an indication that the value is a NaN value is stored in the array.

Figure 12 shows how a `RawDatum` object having a character string value is created.

```
RawDatum RawDatumFactory::createString(const char      *value,
                                       bslma_Allocator *basicAllocator)
{
  BSLMF_ASSERT(8 == sizeof(RawDatum));
  BSLMF_ASSERT(1 == sizeof(char));
  BSLS_ASSERT(basicAllocator);

  RawDatum result;
  const int SHORTSTRING_SIZE = 6;
  std::size_t i = 0;
  while (value[i] != 0 && i < SHORTSTRING_SIZE) {
    result.d_data[SHORTSTRING_OFFSET + i] = value[i];
    ++i;
  }
  // Check for short string.
  if (i < SHORTSTRING_SIZE) {
    static const unsigned short MASK = DOUBLE_MASK | SHORTSTRING;
    new (result.d_data + EXPONENT_OFFSET) unsigned short(MASK);
    // Initialize rest of the bytes to 0 along with null terminator.
    while (i < SHORTSTRING_SIZE) {
      result.d_data[SHORTSTRING_OFFSET + i] = 0;
      ++i;
    }
    return result;
  }

  static const unsigned int MASK = (DOUBLE_MASK | STRING) * 0x10000;
  new (result.d_data + MASK_OFFSET) unsigned int(MASK);
  int len = std::strlen(value);
  // Allocate extra bytes for length of the string to be stored before
  // the string itself.
  void *mem = allocate(len + 1 + sizeof(int), basicAllocator);
  *(int*)(mem) = len;
  char *data = (char*)(mem) + sizeof(int);
  std::memcpy(data, value, len + 1);
  new (result.d_data + DATA_OFFSET) void*(mem);
  return result;
}
```

Figure 12: Creating a RawDatum having a string value

`createString` takes an allocator to allocate a copy of the passed in character string, if the string length is greater than 6 bytes (including the terminating null character). Otherwise, the string is stored directly inside the `RawDatum` object. Note that the length of the string is stored in the begining, if the string is allocated on the heap.

Figure 13 shows how a `RawDatum` object having an array of uninitialized `RawDatum` values is created.

```
RawDatum RawDatumFactory::createUninitializedArray(
                                       RawDatum         **result,
                                       int                len,
                                       bslma_Allocator   *basicAllocator)
{
  BSLMF_ASSERT(8 == sizeof(RawDatum));
  BSLS_ASSERT(result);
  BSLS_ASSERT(0 < len);
  BSLS_ASSERT(basicAllocator);

  RawDatum rslt;
  static const unsigned int MASK = (DOUBLE_MASK | ARRAY) * 0x10000;
  new (rslt.d_data + MASK_OFFSET) unsigned int(MASK);
  void *mem = allocate(sizeof(RawDatum) * (len + 1), basicAllocator);
  *result = ((RawDatum*)mem) + 1;
  *(int*)(mem) = len;
  new (rslt.d_data + DATA_OFFSET) void*(mem);
  return rslt;
}
```

Figure 13: Creating a RawDatum having an array of uninitialized values

`RawDatum` object with an array of uninitialized values is created and returned. Address of the array inside the newly created `RawDatum` object is also returned.

Figure 14 shows how a `RawDatum` object having an array of `RawDatum` values is created.

```
RawDatum RawDatumFactory::createArray(const RawDatum   *values,
                                      std::size_t       len,
                                      bslma_Allocator *basicAllocator)
{
  BSLMF_ASSERT(8 == sizeof(RawDatum));
  BSLS_ASSERT(values);
  BSLS_ASSERT(0 < len);
  BSLS_ASSERT(basicAllocator);

  RawDatum result;
  static const unsigned int MASK = (DOUBLE_MASK | ARRAY) * 0x10000;
  new (result.d_data + MASK_OFFSET) unsigned int(MASK);
  void *mem = allocate(sizeof(RawDatum) * (len + 1), basicAllocator);
  RawDatum *toValues = ((RawDatum*)mem) + 1;
  // Track the allocated memory and destroy it if any of the allocations
  // inside the for loop throws.
  ArrayAutoDtor adtor((RawDatum*)(mem),
                      toValues,
                      toValues,
                      basicAllocator);
  for (int i = 0; i < len; ++i) {
      CopyVisitor cv(&toValues[i], basicAllocator);
      values[i].apply(cv);
      adtor.moveEnd();
  }
  *(int*)(mem) = len;
  new (result.d_data + DATA_OFFSET) void*(mem);
  adtor.release();
  return result;
}
```

Figure 14: Creating a RawDatum having an array of values

The passed in array is deep-copied to create the new `RawDatum` object. The length of the array is stored in the begining. Note that this method uses a proctor class `ArrayAutoDtor` to automatically destroy all the initialized `RawDatum` objects if an out-of-memory exception is thrown while allocating one of the objects. It also releases memory allocated for the entire array.

Figure 15 shows how a `RawDatum` object having a date value is created.

```
RawDatum RawDatumFactory::createDate(const bdet_Date& value)
{
  BSLMF_ASSERT(8 == sizeof(RawDatum));
  BSLMF_ASSERT(4 >= sizeof(bdet_Date));

  RawDatum result;
  static const unsigned int MASK = (DOUBLE_MASK | DATE) * 0x10000;
  new (result.d_data + MASK_OFFSET) unsigned int(MASK);
  new (result.d_data + DATA_OFFSET) bdet_Date(value);
  return result;
}
```

Figure 15: Creating a RawDatum having a date value

A `bdet_Date` value is directly stored inside the `RawDatum` object. Similarly, a `bdet_Time` value is also stored inline.

Figure 16 shows how a `RawDatum` object having a date and time value is created.

```
RawDatum RawDatumFactory::createDatetime(
                                 const bdet_Datetime&  value,
                                 bslma_Allocator      *basicAllocator)
{
  BSLMF_ASSERT(8 == sizeof(RawDatum));
  BSLS_ASSERT(basicAllocator);

  RawDatum result;
  static const unsigned int MASK = (DOUBLE_MASK | DATETIME) * 0x10000;
  new (result.d_data + MASK_OFFSET) unsigned int(MASK);
  void *mem = allocate(sizeof(bdet_Datetime), basicAllocator);
  const bdet_Datetime *copy = new (mem) bdet_Datetime(value);
  new (result.d_data + DATA_OFFSET) void*(mem);
  return result;
}
```

Figure 16: Creating a RawDatum having a date and time value

A date and time value is representative of all the fixed length types of values that are externally allocated.

Figure 17 shows how a `RawDatum` object having a `dlcp::Cloneable` object is created.

```
RawDatum RawDatumFactory::createCloneable(dlcp::Cloneable *data,
                                          int              type)
{
  BSLMF_ASSERT(8 == sizeof(RawDatum));
  BSLS_ASSERT_SAFE(data);
  BSLS_ASSERT_SAFE(0 <= type && type <= 65535);

  RawDatum result;
  static const unsigned short MASK = DOUBLE_MASK | CLONEABLE;
  new (result.d_data + EXPONENT_OFFSET) unsigned short(MASK);
  new (result.d_data + DATA_OFFSET) dlcp::Cloneable*(data);
  // Copy the specified 'type' into the remaining two bytes.
  new (result.d_data + SHORT_OFFSET) short(type);
  return result;
}
```

Figure 17: Creating a RawDatum having a dlcp::Cloneable object

An object that implements the `dlcp::Cloneable` protocol can be cloned by calling the `clone` method on it. It also has a virtual destructor and needs to be destroyed when the `RawDatum` object is destroyed. The 'type' parameter can have value between 0 and 65535 and is used to provide type information regarding the derived class object.

Figure 18 shows how a `RawDatum` object having a user-defined object (denoted by a void*) is created.

```
RawDatum RawDatumFactory::createUdt(void *data, int type)
{
  BSLMF_ASSERT(8 == sizeof(RawDatum));
  BSLMF_ASSERT(4 == sizeof(void*));
  BSLS_ASSERT_SAFE(data);
  BSLS_ASSERT_SAFE(0 <= type && type <= 65535);

  RawDatum result;
  static const unsigned short MASK = DOUBLE_MASK | USERDEFINED;
  new (result.d_data + EXPONENT_OFFSET) unsigned short(MASK);
  new (result.d_data + DATA_OFFSET) void*(data);
  // Copy the specified 'type' into the remaining two bytes.
  new (result.d_data + SHORT_OFFSET) short(type);
  return result;
}
```

Figure 18: Creating a RawDatum having a user-defined object

A user-defined object is represented by an opaque pointer (void*). The 'type' parameter can have value between 0 and 65535 and is used to provide type information regarding the user-defined object.

Figure 19 shows how a `RawDatum` value is copied from an existing `RawDatum` object.

```
RawDatum RawDatumFactory::copyRawDatum(const RawDatum&  value,
                                       bslma_Allocator *basicAllocator)
{
  BSLS_ASSERT(basicAllocator);
  RawDatum result;
  CopyVisitor cv(&result, basicAllocator);
  value.apply(cv);
  return result;
}
```

Figure 19: Copying a RawDatum value from another RawDatum object

`copyRawDatum` uses a visitor to copy a particular type of `RawDatum` object.

Figure 20 shows how the memory allocated within an `RawDatum` object is deallocated.

```
void RawDatumFactory::destroyRawDatum(const RawDatum&  value,
                                      bslma_Allocator *basicAllocator)
{
  BSLS_ASSERT(basicAllocator);
  const RawDatum::DataType type = rawdatum.type();
  switch (type) {
    case CLONEABLE: {
      dlcp::Cloneable *object =
          *(dlcp::Cloneable**)(&value.d_data[DATA_OFFSET]);
      basicAllocator->deleteObject(object);
    } break;
    case RawDatum::ARRAY: {
      ArrayRef values = value.theArray();
      for (bsls_Types::size_type i = 0; i < values.size(); ++i) {
        destroyRawDatum(values.d_data()[i], basicAllocator);
      }
    } // Fall through to release memory allocated for the entire array.
    case RawDatum::STRING:
    case RawDatum::DATETIME:
    case RawDatum::DATETIME_INTERVAL:
    case RawDatum::INTEGER64: {
      // NOTE: Call destructor of a type,
      // if the destructor is not trivial.
      void **mem = (void**)(&value.d_data[RawDatum::DATA_OFFSET]);
      deallocate(*mem, basicAllocator);
    } break;
  }
}
```

Figure 20: Deallocating memory in a RawDatum

If the value stored inside a `RawDatum` object is allocated externally, it is released using the passed in `Allocator` object. If the passed in `RawDatum` object contains an array, `destroyRawDatum` is called on each `RawDatum` object inside the array. For `RawDatum` object with `dlcp::Cloneable` object, the virtual destructor of the `dlcp::Cloneable` object is invoked. Note that `RawDatum` objects are bitwise copyable. Only one of the copies of the same `RawDatum` object can be destroyed. The rest of those copies then become invalid and should not be used or destroyed. It is an undefined behavior to call any accessors or `destroyRawDatum` on these copies. It is also undefined behavior to pass in an uninitialized or partially initialized array to this function. All arrays created using createUninitializedArray should be completely initialized before destroying them.

Figure 21 shows how we get the type of value stored inside a `RawDatum` object.

```
RawDatum::DataType RawDatum::type() const
{
  if (0x7f == d_data[EXPONENT_OFFSET + 1] &&
      0xf0 == (d_data[EXPONENT_OFFSET] & 0xf0)) {
        return static_cast<DataType>(d_data[EXPONENT_OFFSET] & 0x0f);
  }
  return REAL;
}
```

Figure 21: Getting type of value stored inside a RawDatum

We check if the bits representing the exponent part of a `double` are set to 1. If so, then we return the discriminator value stored in the nibble in the fraction part. Otherwise, we treat the number as a `double` value.

Figure 22 shows how we get the type of value stored as an extended type inside a `RawDatum` object.

```
RawDatum::ExtendedType RawDatum::extendedType() const
{
  BSLS_ASSERT_SAFE(EXTENDED == type());
  return static_cast<ExtendedType>
            (*(const short*)(&d_data[SHORT_OFFSET]));
}
```

Figure 22: Getting extended type of value stored inside a RawDatum

This is only invoked when the type of the value stored is `EXTENDED`. In such a case, the value stored in the 2 bytes (not used to store data) is returned as the extended type.

Figure 23 shows how an integer value is extracted from an `RawDatum` object.

```
bool RawDatum::isInteger() const
{
  return (INTEGER == type());
}

int RawDatum::theInteger() const
{
  BSLS_ASSERT_SAFE(isInteger());
  return *(const int*)(&d_data[DATA_OFFSET]);
}
```

Figure 23: Retrieving an integer value from a RawDatum

Figure 24 shows how a double value is extracted from an `RawDatum` object.

```
bool RawDatum::isDouble() const
{
  const DataType t = type();
  if (REAL == t || INF == t) {
    return true;
  }
  return (EXTENDED == t && NAN2 == extendedType());
}

double RawDatum::theDouble() const
{
  BSLS_ASSERT_SAFE(isDouble());
  const DataType t = type();
  return ((REAL == t || INF == t) ? *(const double*)(d_data) :
              bsl::numeric_limits<double>::quiet_NaN());
}
```

Figure 24: Retrieving a double value from a RawDatum

Notice that, more than one discriminator value can indicate a `double` value. Also notice that a NaN value is stored as using another discriminator, with the original discriminator having the value 'EXTENDED'.

Figure 25 shows how a character string value is extracted from a `RawDatum` object. The value is returned as a `bdeut_StringRef` object (which also contains the length of the string).

```
bool RawDatum::isString() const
{
  const DataType t = type();
  return (SHORTSTRING == t || STRING == t);
}

bdeut_StringRef RawDatum::theString() const
{
  BSLS_ASSERT_SAFE(isString());
  if (SHORTSTRING == type()) {
    const char *data = (const char*)(&d_data[SHORTSTRING_OFFSET]);
    return bdeut_StringRef(data, strlen(data));
  }
  const char *data = *(const char**)(&d_data[DATA_OFFSET]);
  return bdeut_StringRef(data + sizeof(int), *(const int*)data);
}
```

Figure 25: Retrieving a string value from a RawDatum

Figure 26 shows how an array of values is extracted from a `RawDatum` object.

```
bool RawDatum::isArray() const
{
  return (ARRAY == type());
}

ArrayRef RawDatum::theArray() const
{
  BSLS_ASSERT_SAFE(isArray());
  const RawDatum *arr = *(const RawDatum**)(&d_data[DATA_OFFSET]);
  ArrayRef ref(arr + 1, *(const int*)arr);
  return ref;
}
```

Figure 26: Retrieving an array of values from a RawDatum

Note that the array and its size are returned as an `ArrayRef` object, which is a value-semantic wrapper around the array and its size.

Figure 27 shows how a date value is extracted from an `RawDatum` object.

```
bool RawDatum::isDate() const
{
  return (DATE == type());
}

bdet_Date RawDatum::theDate() const
{
  BSLS_ASSERT_SAFE(isDate());
  return *(const bdet_Date*)(&d_data[DATA_OFFSET]);

}
```

Figure 27: Retrieving a date value from a RawDatum

Figure 28 shows how a date and time value is extracted from a `RawDatum` object.

```
bool RawDatum::isDatetime() const
{
  return (DATETIME == type());
}

const bdet_Datetime& RawDatum::theDatetime() const
{
  BSLS_ASSERT(isDatetime());
  return **(const bdet_Datetime**)(&d_data[DATA_OFFSET]);
}
```

Figure 28: Retrieving a date and time value from a RawDatum

Figure 29 shows how a `dlcp::Cloneable` object is extracted from a `RawDatum` object.

```
bool RawDatum::isCloneable() const
{
  return (CLONEABLE == type());
}

Cloneable RawDatum::theCloneable() const
{
  BSLS_ASSERT_SAFE(isCloneable());
  return (*(dlcp::Cloneable**)(&d_data[DATA_OFFSET]),
          *(const short*)(&d_data[SHORT_OFFSET]));
}
```

Figure 29: Retrieving a dlcp::Cloneable object from a RawDatum

Note that the `dlcp::Cloneable` object and its type are returned as a `Cloneable` object, which is a value-semantic wrapper around the object and its type.

Figure 30 shows how a user-defined object is extracted from a `RawDatum` object.

```
bool RawDatum::isUdt() const
{
  return (USERDEFINED == type());
}

Udt RawDatum::theUdt() const
{
  BSLS_ASSERT_SAFE(isUdt());
  return Udt(*(void**)(&d_data[DATA_OFFSET]),
             *(const short*)(&d_data[SHORT_OFFSET]));
}
```

Figure 30: Retrieving a user-defined object from a RawDatum

Note that the user-defined object and its type are returned as a `Udt` object, which is a value-semantic wrapper around the object and its type.

8 Datum

The `Datum` class is used to wrap the `RawDatum` and create a full value-semantic type on top of it. `Datum` has constructors for creating different `RawDatum` values. These constructors take an optional pointer to an `Allocator` and use the `RawDatumFactory` class to construct the `RawDatum` object. The destructor for `Datum` class releases any allocated memory inside the contained `RawDatum` object using the `RawDatumFactory` class. Figure 31 shows the core definition of `Datum`.

```
class Datum {
  RawDatum          d_data;
  bslma_Allocator  *d_alloc_p;

public:
  explicit Datum(bslma_Allocator *basicAllocator = 0);
  explicit Datum(int value, bslma_Allocator *basicAllocator = 0);
  explicit Datum(double value, bslma_Allocator *basicAllocator = 0);
  explicit Datum(bdeut_StringRef   value,
                 bslma_Allocator *basicAllocator = 0);
  Datum(const ArrayRef& values, bslma_Allocator *basicAllocator = 0);
  explicit Datum(const bdet_Date&   value,
                 bslma_Allocator   *basicAllocator = 0);
  //...
  explicit Datum(const RawDatum&   value,
                 bslma_Allocator *basicAllocator = 0);
  explicit Datum(const Datum&      original,
                 bslma_Allocator *basicAllocator = 0);
  ~Datum();

  RawDatum rawdatum() const;
};
```

Figure 31: Datum

`d_alloc_p` stores address of the user-supplied `Allocator` object, which is used to allocate and release memory used by values that are not stored directly inside the `RawDatum` object.

9 Conclusion

`RawDatum` represents the same values as `Value`. Values of the most frequently used types like `double`, `int`, and `bool` are stored inline. All larger non-primitive and variable length types, that are less frequently used, are allocated externally. As shown in Figure 2, `bsl::string` contributes 16 bytes to the size of `Value`. By eliminating the use of `bsl::string` and managing the character array manually, 8 bytes of memory per `RawDatum` object is saved in case of string values. 8 more bytes of memory is saved per `RawDatum` object by not storing the allocator. String with a short length (which are frequently used), are stored inline to avoid external allocation. Thus, `RawDatum` satisfies the first two requirements from Figure 3.

`RawDatum` is a light-weight, in-core, value-semantic type. It has trivial constructor, destructor, copy-constructor and copy-assignment operator. `RawDatum` is bitwise copyable and any allocated memory is shared among `RawDatum` instances on copy-construction and copy-assignment. Thus, `RawDatum` also satisfies the last two requirements from Figure 3.

`Datum` restores the full value-semantics of `RawDatum`. Thus, users of `Value` can use `RawDatum` if efficiency is an issue and they can use `Datum` if full value-semantics is needed.

We assert that `RawDatum` reduces the memory footprint of Hypersheet and Datalayer applications, while also being less CPU-intensive than `Value`. We will back this claim up with a section on performance tests.