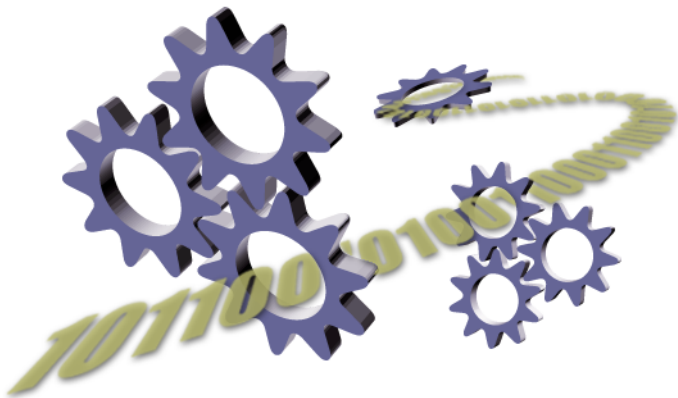


Event Store Redesign

Revision # 0.6.1



Contents

Preface	v
1 Shielding Persistence	1-1
2 Persistent Event Store (Current)	2-1
Sizes	2-2
3 Persistent Event Store (Proposed)	3-1
Proposed Changes	3-1
Store event headers with event	3-1
Store only references to renewed data?	3-4
Common objects	3-5
Contents of a common object	3-5
Matching events with common objects	3-6
Conclusions	3-6
Tag event	3-7
Deletion / garbage collection	3-8
Persisting mapping: data - which header	3-9
Reorganizing headers	3-9
Scalable collections	3-10
Collection --> db id mapping	3-10
Persistent event - others (not covered so far)	3-11
Avoiding resizing	3-11
Persistent tag	3-11
Sizes	3-12
4 Accessing Tags	4-1
TagTransient	4-1
A simple tag processing model	4-1

Filtering events with TagAttribute	4-2
'Fast' filtering	4-2
Re-organizing the classes	4-2
5 Summary.	5-1
Compatibility	5-1
Pros	5-1
Cons	5-2
Impact on users.	5-2
Deadlines	5-2
Progress	5-2
Manpower	5-3
A Example of Persistent Event in Current System	A-1
B Example of Persistent Event in New System	B-1
C Highlights of New Design	C-1
D Persistent Classes in Current System	D-1
E Persistent Classes in Proposed System	E-1

Preface

Main purpose of embarking on a redesign is to reduce disk space usage for the disk-resident components of the event store, in particular navigational component (col, tag, evt, evshdr). The redesign will allow to add some new important features, that were difficult to implement so far, due to two facts:

- They required changes to persistent schema.
- Persistent event store classes were not shielded behind transient interface, and were used throughout the BaBar code, making it very difficult to make any change to persistent classes.

The redesign should *not* be done at an expense of flexibility. All used features should be preserved, and performance should be no worse than in current system.

The project has been divided into several stages:

1. Shield persistent event store classes behind transient interface.
2. Introduce new persistent classes.
3. Plug the new persistent classes into the system. This has to be transparent and backwards compatible, with none, or minimal impact on users.

This document describes current event store design, features it provides, followed by description of the proposed, new system. Sizes, new features, strong and weak points of both systems are discussed.

1

Shielding Persistence

In the current system, persistent event store classes are exposed in many places in the BaBar software. To allow any modifications to these classes, they need to be hidden behind a wrapper. The wrapper will be able to deal with multiple versions of persistent classes, and automatically (and transparently) switch between them. Strategy pattern will be used to handle different types of persistent classes.

The following transient classes will be exposed to transient world: `BdbFastTagT`, `BdbTagT`, `BdbEventT`, `BdbAbsCollectionT<BdbEventT>*`. No code, other than event store classes, may directly use any of the event store persistent classes.

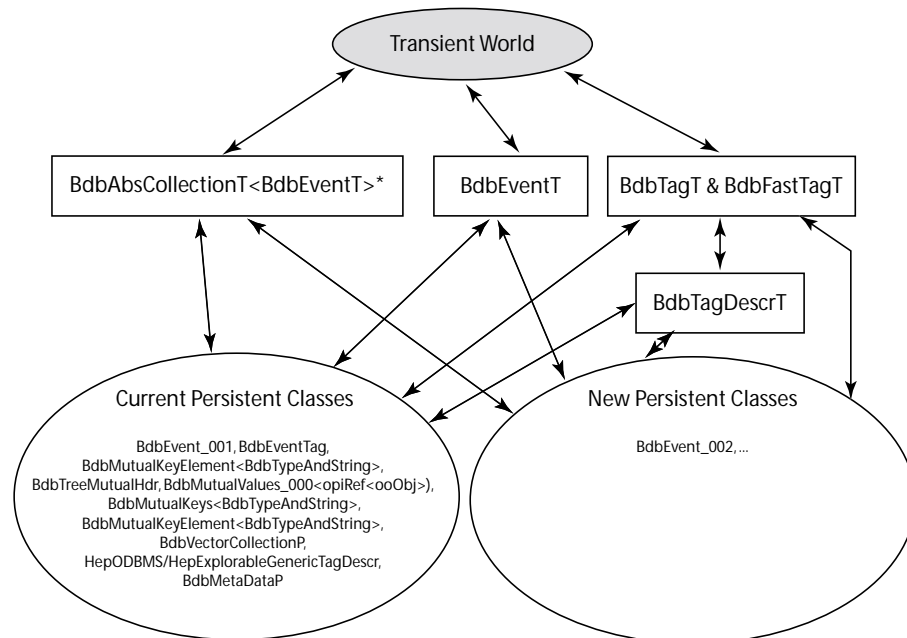


Figure 1-1. Shielding persistent event store

2

Persistent Event Store (Current)

Navigation for the event store is spread across 4 different databases: col, evt, evshdr and tag. The [Figure 2-1](#) shows the high-level physical structure of the components that are participating in the navigation.

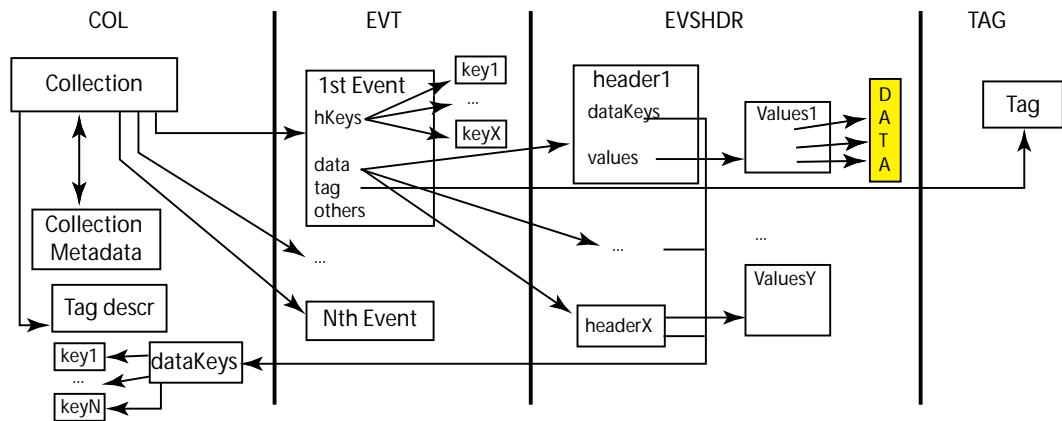


Figure 2-1. Simplified diagram of navigation in current event store

How does event store locate a single data object (e.g. a single BtaCandidate) for all, or some events in a given collection X? The collection is located via tree nodes. Iterating over all events (and possibly applying some filters) is the most popular method of accessing events. An event, in addition to identifier (EidContainer) and some other internal attributes is built of event headers and header keys. To access a data object, we need to know which header it belongs to (header name, also known as header key), name and type of data object (name and type is sometimes called data key).

An even header is identified via a header name (a key). A header contains a list of pointers to data objects. Each data object is identified by a key, which consists of a string and type number. A list of data keys is contacted to identify position of pointer to data object in the array stored in a header. Data keys keep a mapping between the data object identifier (name and type) and offset of data object in the header. Data keys are stored in a collection database (see Figure 2-1). A manager of data keys (internally implemented as ooMap), also stored in the collection database provides a convenient access to data keys. The “final” result of the navigation for an event is a pointer to a data object. Data objects are stored in separate files, (e.g. aod, esd, raw, rec), marked in yellow on the Figure 2-1 above.

A complete diagram of an example event, including size overhead, can be found in [Appendix A](#).

Sizes

All sizes discussed in this document relate to

- real size occupied by an object or an attribute (for instance long --> 4 bytes)
- alignment between attributes
- overhead of a persistent object or variable size field

They do *not* include alignment between objects, or overhead of an empty container or a database.

Size overhead per each part for a typical event (produced by OPR) is presented in [Table 2-1](#) and [Table 2-2](#).

Table 2-1. Detailed size overhead

Element	Component	Size per event	Constant size
Collection	col	8	98
Collection metadata	col		2400
Tag descriptor	col		45000
Keys	col		118*9
Key1 +... + KeyN	col		48*9*5
Event	evt	300	
Key1 +... + KeyN	evt	52*9	
Header1 +... + HeaderM	evshdr	38*9	
Value1 +... + ValueM	evshdr	122*9	
Tag	tag	576	

Table 2-2. Per-component size overhead

Component	Size
col	$0.08 \text{ kB} * N^* + 52 \text{ kB}$
evt	$0.8 \text{ kB} * N$
evshdr	$1.5 \text{ kB} * N$
tag	$0.6 \text{ kB} * N$
Total	$2.9 \text{ kB} * N + 52 \text{ kB}$

* N - number of events

3

Persistent Event Store (Proposed)

The proposed changes are made around several ideas discussed below in details:

- keeping event headers with an event
- common objects
- tag events/tag collections.

They relay on standard techniques:

- reducing number of small persistent objects
- introducing new schema that well matches bulk data stored
- reducing duplication of same data
- preferring short refs (4 bytes) to refs (8 bytes), when applicable.
- removing unused (obsolete) attributes
- combining many vstrings into one.

The largest overhead is in evshdr and evt.

Proposed Changes

Store event headers with event

Event headers are currently stored separately from an event object. Each pair (key, value) consists of 2 persistent objects, as shown in [Figure 3-1](#).

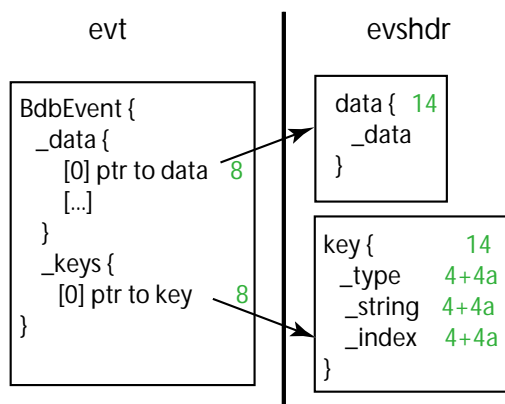


Figure 3-1. Simplified diagram of an event and its headers

Size overhead (without size of “useful” data) is indicated to the right of each object/attribute in green: 68 bytes. An event produced by OPR has 9 headers: emc, rec, trk, bta, svt, ifr, drc, dch, stateID.

The headers are stored in evshdr database separately from an event due to historical reasons. In the past, data for some headers (raw, rec) was big and rarely accessed in typical analysis. Putting headers in a separate database allowed to avoid overhead of reading it each time an event is accessed. Also, we were persisting *all* headers, not only these that are non-empty (for example we were persisting “sim” and “tru” headers for every event that was produced in OPR).

It would be very beneficial to store headers with an event (better clustering, faster access, reduction in space overhead). However, keeping headers with an event will not allow to “borrow” headers - feature broadly used in BaBar. Borrowing header is explained in [Figure 3-2](#) (red arrows indicate borrowing).

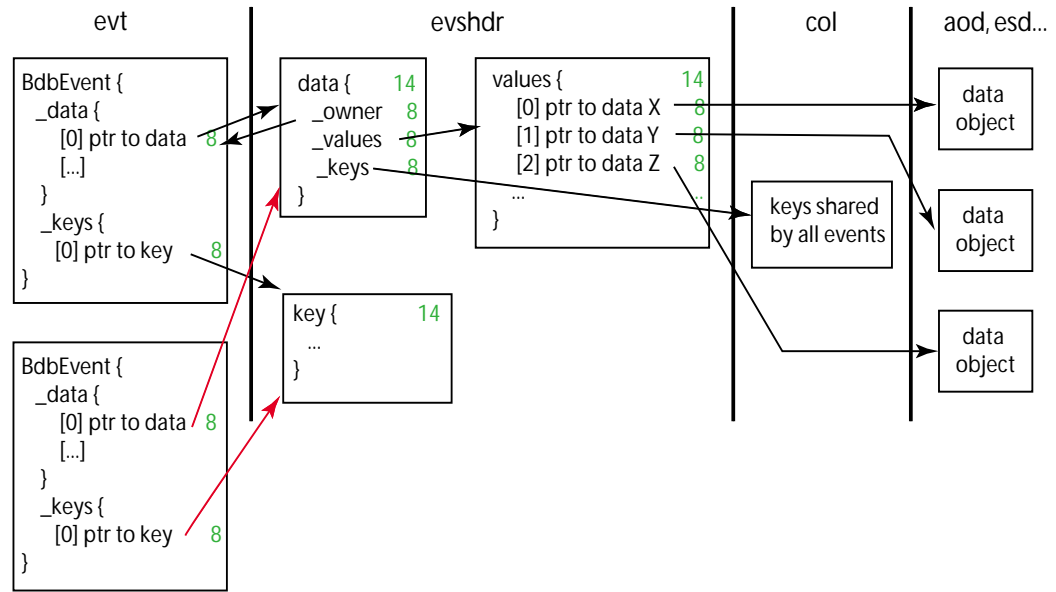


Figure 3-2. Borrowing headers

To store headers with an event and allow borrowing we can borrow data that belongs to a header. Instead of borrowing a header object, borrow data objects that the header points to. See Figure 3-3. Persistent schema would have to allow finding mapping between a header and data objects associated with the header. This is simple to implement and introduces negligible overhead. An example draft implementation is presented in Appendix C.

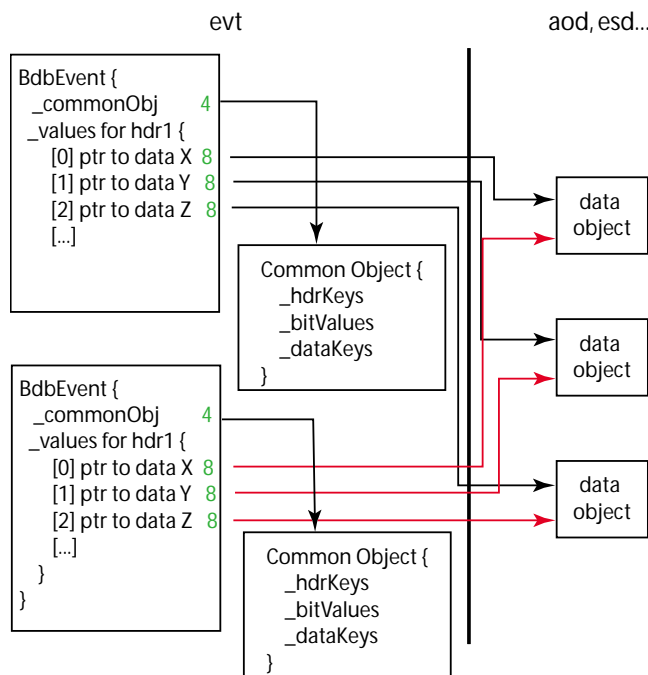


Figure 3-3. Borrowing data

Borrowing data instead of headers gives even more flexibility than borrowing headers. We could potentially borrow only part of data that belongs to a header, not necessarily all of it (which is always the case in the current system). And the most important - we eliminate 74 (out of 82) bytes of overhead per event header.

Why 82? Note, that the “data” attribute storing “values” (see [Figure 3-2](#)) introduces another 14 bytes of overhead on top of 68 already explained above. $68+14 = 82$ of pure overhead, without counting size of useful information stored in a header: pointers to data (aod, esd, raw, rec...).

Borrowing on data level rather than on header level can be hidden behind the transient interface, and it can be done transparently. Users will not be aware whether they borrow a header or data.

Another feature of event headers is, that new headers of any type can be added to an event without touching persistent schema. This feature will be preserved.

Also, because we are introducing new persistent class, we can now also store information whether a header is owned or borrowed. This will allow to add a new feature - deletion of collections/events (deletion would recover space in a database file, unlike in the current system, where deleted object is only marked as deleted, and still occupies space).

Store only references to renewed data?

In the current system, when we renew headers, a new event keeps *all* references to its headers (some references point to new header(s), the rest point to borrowed header(s)). In the new system each new event will keep all references to its headers' data (some will point to new data, the rest will point to borrowed data).

Suggestion: store only references to new data, and do not store any references to borrowed data. These pointers can be fetched from the original event.

Pro:

- Saving space (7.75 bytes per every pointer, on average ~300 bytes).

Cons:

- To get data for an event, an original event would have to be fetched. However, since the event that does not keep references to borrowed data will be significantly smaller now, many such events can fit on one page - so this feature will not increase I/O much.
- System will be more complex.

This feature could be made optional, that is, a user could choose whether pointers to borrow data should be stored or not.

Technical details of an example implementation are in [Appendix C](#).

We have considered all the pros and cons, and decided not to implement this option. The reason: savings are not worth the complexity.

Common objects

In the old system, thousands of objects stored together frequently have a lot in common, for instance run number, or id of platform used to store conditions. They tend to be the same, or change very infrequently. Instead of repeating the same set of values, the common values can be stored once (e.g. per container) and events would point to the object storing them (called a *common object*). The system would allow multiple common objects to coexist in the same container, in case common data is not the same for every event stored in that container.

Contents of a common object

The following data will be stored in a common object:

1. Header keys. Maps textual identifiers for headers(e.g. trk, rec, dch) to sets of data pointers in the event.
2. Data keys. Maps name and type pairs of data objects to their corresponding indices within sets of data pointers. As header keys, they do not change between events. In current system they are stored in a collection database.
3. Part of eid container:
 - `_eventPlatform`
 - `_eventPartitionMask`
 - `_condKeyPlatform`
 - `_condKeyPartitionMask`
 - `_configKey`
 - `_run`

The remaining attributes change frequently, and will be stored as part of an event.

4. State id list

Introducing common objects also allows to store full history of reprocessing (vector of "state ids"). Keeping that information persistently in the current system would introduce large space overhead. With common objects, the change in size will be unnoticeable.

5. Link (short reference) to tag a descriptor.

Tag descriptor is currently kept in a "col" database. This forces all events belonging to a collection to reference the same tag descriptor. It is not stored together with every tag due to space overhead (size of a single tag descriptor in the current system is over 45 kB). Such an arrangement has one big disadvantage: events from the same collection may not have different tag descriptors.

Introducing common object will allow us to store tag descriptor together with tag, and will allow for more than one tag descriptor per collection.

A single tag descriptor stores 558 strings, each string is stored as char [64]. Storing all strings in one vstring would reduce size to ~5.6 kB, assuming 10 bytes per string.

Due to size of a tag descriptor we will store it in a separate object, and keep a short reference to it from a common object. Should we need to create more than one common objects, we will avoid duplicating a tag descriptor (many common objects will point to a single tag descriptor).

Matching events with common objects

In the ideal case, a common object would be shared as much as possible. This means that no common objects would exist that have the same fields. In practice, it's expensive to search all possible common objects, so instead, we limit common objects to only be shared within a container. Within a container, we aim to share common objects among events produced in a job. This seems possible, because within a job, we can transiently store all the common objects used. We're unsure whether to attempt to share common objects that exist in the container before the run begins. In that case, reuse seems less likely to begin with. It's also unclear how to navigate existing persistent common objects, since our current design only links them to their respective events.

To reuse a common object, we must ensure that an existing common object has exactly the same fields. Doing a field-by-field equals-test is possible and not prohibitively expensive, as long as the total footprint of the common object is kept small. The downside of this is that the cost scales linearly with the number of common objects, and linearly with the size of the common object. We think it would be a better idea to store hashes of common objects. Then we compare only the hashes when deciding whether or not to link. While the cost of hashing can be larger than the cost of the simple field-compare, a hashed method scales more gracefully when more common objects are being compared, in both memory size and performance. The hashing algorithm has not been determined, but it is thought that a cryptographic hash is too expensive on performance, despite possible gain from its nice properties.

Conclusions

Common objects will be stored together with events that point to them (together = in the same container). This has important benefits:

- An event can use a short ref to reference its common object (4 bytes, full reference occupies 8).
- It reduces lock traffic (no extra containers need to be opened).
- It is natural to store where it belongs.

A drawback:

- In systems like OPR, where we split a single run into many containers (one container per client), a 100-client run will result in 100 common objects. Luckily, that still remains insignificant, especially if the number of events is large.

It is very important to understand what to store in common objects. One could think that since common object is shared by many events, its size does not matter much. In addition to the drawback mentioned above, if one bit of data in common object changes, it will force the system to generate a new common object. Given that size of a common object is significant (measured in kilobytes) data stored in common objects should be very stable across hundreds of events, or common objects will fire back and blow out size of evt component.

The code creating and storing persistent objects will “cache” only one (most recently used) common object. Given what we will store in common object,s caching a single instance should suffice.

Tag event

A lot of data is produced by “skims”. A skim job often creates a new tag, which currently forces us to create a new event object:

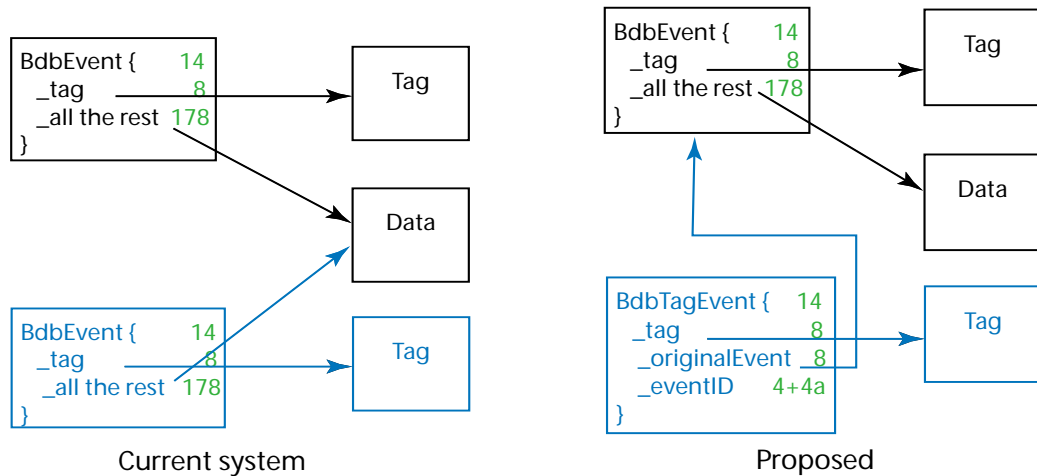


Figure 3-4. Creating new tag - current and proposed

Figure 3-4 shows how current system works, and how the proposed system would (suggestion A). Blue indicates data produced by skims, green sizes. In order to rewrite tag, now we have to pay 178 bytes of extra overhead (to duplicate all other data contained in BdbEvent). The proposed scheme is based on a special event type “BdbTagEvent”.

The new “BdbTagEvent” would contain a pointer to its predecessor event (8 bytes overhead), and a pointer to the new tag. This gives us 170 (178-8) bytes of saving - a BdbTagEvent is 80% smaller than a regular event.

The overhead of following the extra link would be unnoticeable: we would need to read a page that contains skim event, and a page that contains the original event (compare to reading one page that contains a whole event). However, given that the skim event is very small, many events could be clustered on the same page, spreading overhead of reading this page across hundreds of skim events.

Suggestion B: Introduce a new type of collection, “skim collection”. This would allow to save another 14 bytes of overhead associated with an overhead of a persistent object (BdbTagEvent).

Disadvantage: some attributes of event class would have to be exposed to the new skim collection class.

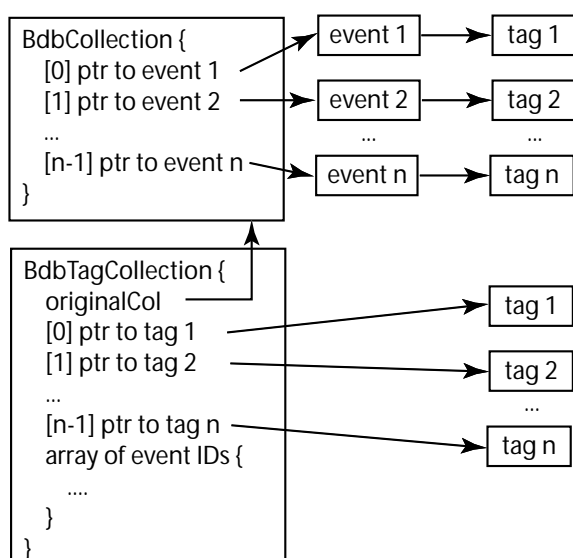


Figure 3-5. Creating new tag - TagCollection approach

This model could be easily expanded to allow for skimming multiple input collections, and storing output in one output collection (e.g. by introducing an array of original collections). A job iterating through BdbTagCollection would follow its pointers to get tag, and get corresponding event from an appropriate original collection.

Decision: “Suggestion A” will be implemented.

Deletion / garbage collection

True delete which recovers space will be supported. However, even if we track which objects are owned and which are borrowed, we have no way of insuring that deletion does not leave any dangling pointers in the federation (somebody else might be pointing at deleted data).

Often borrowing and deletion are not used in the same federation. Either there is a lot of borrowing (production environment) and no deletion, or users want to delete collections, but they do not use borrowing feature. Solution: introduce a new switch that would allow user to configure a federation as "allow-delete" or "allow-borrow". One or the other, but never both. As far as we can tell, this will give a nice flexibility and robustness: if a user wants to treat a federation as a scratch area, it can create/delete collections, but not borrow. If a user wants to borrow, he/she may not delete. Extracting interesting collections to a separate federation and deleting the old one still may be used in such case to recover space. Also, this significantly simplifies the new event store system, as we would not have to keep track of who owns what.

A production federation would obviously be configured as allow-borrow. Users would extract data to their private federations using deep copy, and would have a choice how to configure their federations as needed.

It would not be allowed to re-configure federation from allow-borrow to allow-delete. It would be configurable until the first event is inserted, and then it would be fixed forever. We could allow a one-way switch from allow-delete to allow-borrow, after events are in the system, but not the other way around.

Persisting mapping: data - which header

In current system the mapping which data element belongs to which header is kept in a simple table that is not stored persistently. It is stored only in our code repository (CVS), and we rely on the fact that data fetched from database correspond to the table in the code. This is inflexible and dangerous. Ideally, we should persist this mapping.

This appears to be very simple, and at almost no cost! How would it work? For each data key we are currently storing persistently the following info: name, type of persistent class and index (position in header). If we store one more number representing offset of the header (the owner) in the header list, we will be able to find which data key belongs to which header (persistently store the mapping). Since this will be stored in common object, overhead is practically zero.

How would it work? A transient table (similar to what we have now) would be used to determine mapping when we build persistent event from transient info. Once an event is persisted, we would always use the persistent mapping, not the transient one. This would allow transient mapping to be reorganized at any time, without any danger of getting inconsistent results.

Reorganizing headers

Current organization of headers (split into rec, trk, bta, svt, ifr, drc, dch) has a lot of legacy behind, and does not reflect the way we are using the system. We are now dealing with two types: "micro" and "mini", and possibly, this would be a better split. We will discuss it with other users of our system, estimate cost and benefits, make a decision and then update this section....

Scalable collections

A collection stores references to events, often thousands of them. In the current system we use regular varrays of references. This means that when a collection is accessed, the whole varray is brought to memory from disk, no matter if we read one event out of thousands, or all of them. **Suggestion:** use paged varrays. They are brought to memory one page at a time - a clear benefit in cases when we deal with large collections, especially if not all events are accessed. **Create a new type "BdbPagedCollectionP", make it a default (now a default is BdbVectorCollectionP).**

Collection --> db id mapping

We will add new metadata objects that will keep mapping between a collection and database ids. Each container with event object will have one or more common dbid objects that will keep information which dbids are associated with events that point to it. A central registry of such common objects kept as part of a (new) collection metadata will allow to quickly find which dbids are associated with a given collection. This scheme will work for owned and borrowed events.

This feature will allow to quickly identify all databases associated with a collection. Currently, to find the mapping, one has to scan every single event, including event headers. It is time consuming (can take many hours), and I/O intensive.

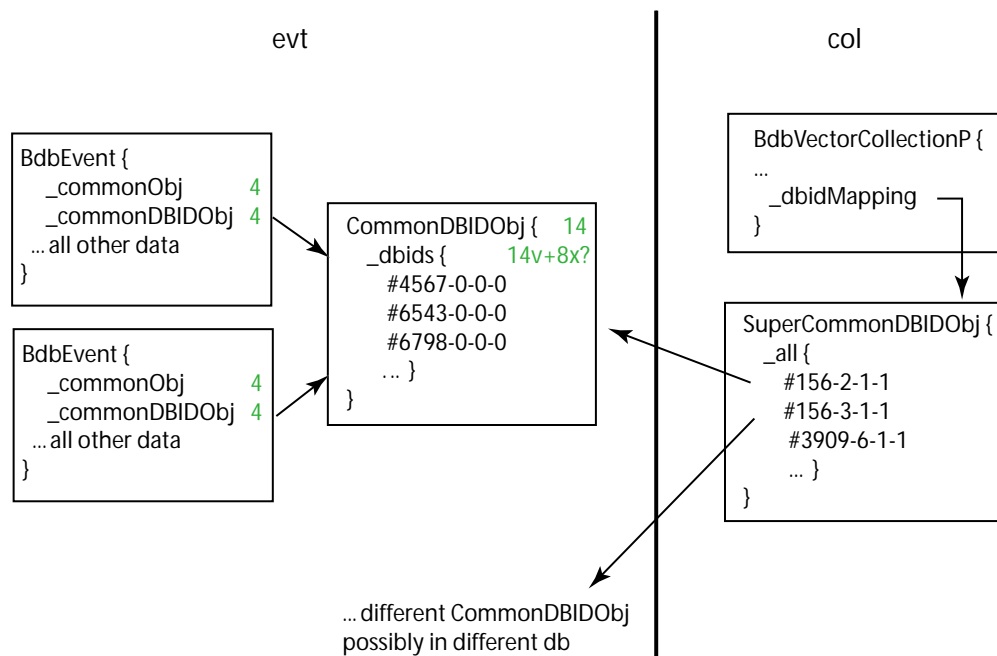


Figure 3-6. Collection to db id mapping

Unused collection metadata objects will be removed (strings starting with "Bdb xxx files:"). This was introduced a long time ago, but was never used/filled. There is no code that would fill it.

Persistent event - others (not covered so far)

The following attributes of event are obsolete and will be removed

- `_isDeleted`. Instead, we will provide a real deletion.
- `_number`
- `_eventID` - redundant field.

Avoiding resizing

In the current system headers are added to an event one by one, and the number of headers is not known when an event is created. This results in resizing a varray that stores header information each time we add a new header. This has implications in both persistent layout of object (less efficient, holes), and performance of reading. To avoid it, we could:

- initially resize a varray to a size equals to a page size (this would force a varray to go to a separate page)
- keep adding headers to that varray
- Once all headers are inserted, copy the varray to transient world, delete the “large” varray, and create a new one with a correct size, then load it with the data caches transiently.

We need to test this.

Another approach would be to pay penalty for the very first event, and pre-size varrays for next events using the size of varray from the previous event (average? max?)

Persistent tag

There is not much we can easily do to store tag data more efficiently. Right now, one tag consists of elements as indicated in [Table 3-1](#):

Table 3-1. Tag elements, and their size overhead

type	No elements	Overhead [bytes]
d_Float	64	14+4*64
d_Double	0	4
d_ULong	44	14+44*4
d_Short	0	4
d_Boolean	0	4
d_Char	64 (on average)	14+64

This totals to 550 bytes. Ideas:

- Do not store empty arrays (double, short, boolean) - there is no support for these types anyway in the transient layer (booleans are packed into chars). This would save 12 bytes.
- Review usage of types (e.g. used short if long not needed, maybe some elements not needed?).

These decisions can *not* be made by the database group. [Jim Smith promised to help with this, any other volunteers?].

Sizes

Expected sizes for navigational components in the proposed system (assuming we would implement features as explained above) are shown in [Table 3-2](#) and [Table 3-3](#).

Table 3-2. Detailed size overhead

Element	Component	Size per event	Constant size
Collection	col	8	?
Collection metadata	col	0	?
Common object	evt	0	10000?
Event	evt	446	0
Tag	tag	576	0

Table 3-3. Per-component size overhead

Component	Size
col	0.08 kB * N + ? kB
evt	0.45 kB * N + 14 kB (?)
evshdr	0
tag	0.6 kB * N
Total	1.1 kB * N + 14 kB (?)

* N - number of events

Most savings are in evt+evshdr: 2.3 kB --> 0.45 kB. *Proposed size < 20% of original size.*

Similarly, "BdbTagEvents" will occupy 80% less space, as described earlier.

4

Accessing Tags

It is quite apparent that the tag related packages have evolved considerably since their inception. They contain some very useful and interesting features for accessing and modifying event tags. But the class structure is rather disjoint, and some functionality is replicated. A common persistent tag wrapper would help.

TagTransient

One of the core classes is TagTransient. This class contains no persistent code and provides a transient tag model by storing tag values in map data structures using the tag field names as keys. Instances of TagTransient may be 'locked'. Locking does not prevent tag values from being overwritten, but it does stop attempts to add new fields. BdbTagTransient is a subclass of TagTransient and keeps references to the persistent tag and tag descriptor. When a tag value is first read through BdbTagTransient, it is obtained from the persistent tag, and then held in TagTransient for subsequent reads. BdbTagTransient is used extensively in the analysis framework, and is normally exchanged between modules by referring to its abstract superclass AbsEventTag.

A simple tag processing model

This case assumes that we do not perform any tag filtering when reading in persistent events. A single BdbTagConverter object is responsible for tag transient<-->persistent conversion.

During the tag input phase of a job, the BdbTagConverter object creates a BdbTagTransient instance, and loads it with the persistent tag data. Loading is optional, and the user may choose to skip individual tag fields. Next, new tag fields that have been declared in advance are appended to the transient tag with associated

default values. In order to guarantee that a tag field is available in the transient tag, analysis modules must pre-declare fields in advance with a default value. If the tag field has already been read from the persistent tag during loading, the loaded value takes precedence.

Modules that wish to declare tag fields in advance must inherit from the `TagAccessor` class. Using `TagAccessor`, a tag field may either be declared during module initialization (`beginJob()`), or declaration may be deferred until the first event is read in. By deferring, modules wait until conditions information become available. A tag field may only be declared once. All instances of `TagAccessor` refer to a `TagDescriptor` singleton.

`TagDescriptor` is subclassed from `TagTransient` and has a similar interface, but will only allow a tag to be set once. When the construction of the transient tag is complete, it is locked to prevent any further addition of tag fields. Modules access the transient tag through `BdbTagTransient`, which is shared via a pointer to `AbsEventTag`. If the job requires event output, `BdbTagConverter` creates a new persistent tag object and loads it using the values in the transient tag.

Filtering events with `TagAttribute`

Rather than process each event in turn and evaluate it's tag, there is the possibility of filtering out events during the event input phase using the tag as a discriminant. The filter input modules, such as `TagFilterInputByName`, inherit from `BdbEventInput`. They examine specific tags bits to determine if the event should be filtered out or not. The `TagAttribute` class is a convenient way of accessing specific tag bits. A `TagAttribute` instance is initialized by supplying references to the persistent tag and persistent descriptor along with the name of the field. Most importantly, `TagAttribute` contains template assignment operators.

Filter input modules build instances of these attributes in order to check tag bit values. Each time the next persistent event is evaluated, the `TagAttribute` instances are bound to the event's persistent tag & descriptor.

'Fast' filtering

The important thing to note about `TagAttribute` is that it does not deal with the transient tag. 'Fast' filtering modules also use `TagAttribute` because there is no transient tag overhead.

Re-organizing the classes

The key constraint with the class diagram is that we have two class structures that deal with access to the persistent tag, but they do not have a common ancestor. Some

modules need to access the tag and store values in transient objects which may be shared by other modules and output to a new persistent tag later. Other modules wish to use the tag to filter events and do not need to values in memory.

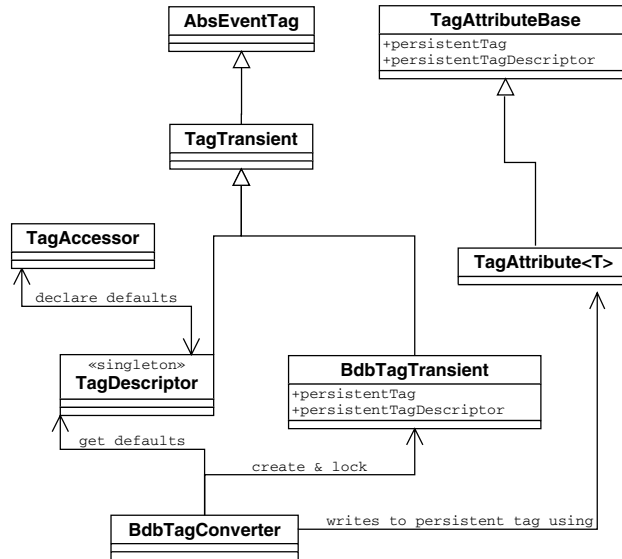


Figure 4-1. Accessing tags - current class structure

The revised class diagram contains only one class that directly accesses the persistent tag & descriptor. BdbFastTagT does not perform any caching, that task is left to TagTransient. BdbTagT unifies the persistent and transient tag access. TagAttribute still has the same functionality without any transient overhead, and most importantly the persistent references are encapsulated in only one class.

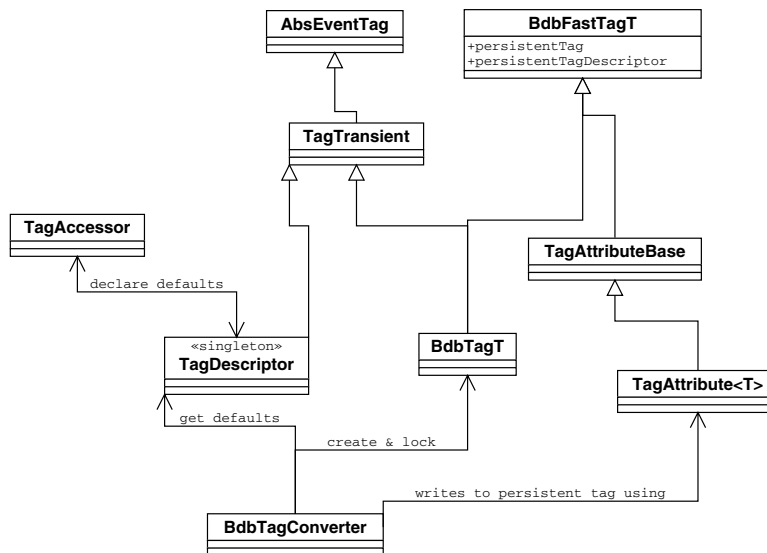


Figure 4-2. Accessing tags - proposed class structure

5

Summary

Compatibility

Changes will be transparent to users, with one exception: old releases will not be able to use new persistent classes.

Each time a system will deal with “old” type event (reprocessing, skimming), a product will be stored in “old” format.

Mixing “old” and “new” events in one collection will be supported.

Mixing “old” and “new” collections in a single tree collection will be supported.

No migration will be necessary.

Pros

- Size of navigational part of event ~70-80% smaller.
- Borrowing data objects, instead of headers.
- Common objects.
- Tag descriptors stored with tag, more than one tag descriptor per collection allowed.
- Mapping of data objects to headers is stored persistently.
- Scalable collections.
- Deleting data will recover space in database.
- Efficient collection metadata will allow to find col name -> db mapping momentarily, no need to scan all event headers.

- Less load on servers (less data to read/write, no evshdr --> less files opened)
- Faster - data clustered more efficiently.
- More scalable - puts less load on servers.
- Code tuned for performance in many places, impossible to list in this document.

Cons

- Old releases will not be able to work with new persistent classes.
- Transient interface will have to deal with different types of persistent classes. The implementation will use a strategy pattern: each strategy object will be responsible for dealing with one persistent class. Switching between strategies will be efficient. We do not expect to introduce performance overhead larger than 1%. It will be surpassed by performance boost due to better organization of data and tuning in code.
- Introducing broad changes in the low level code - lots of work to productize.

Impact on users

Old releases will not be able to use new persistent classes.

We expect all changes to be transparent - hidden behind the wrapper that we control. Users will not be forced to migrate any of their code. No data will need to be migrated. All changes are backwards compatible.

Deadlines

A deadline agreed with BaBar management: in a test release by the end of December 2002. Since some features initially were not part of the redesign (skim events, deleting events, dbid metadata), they may be finalized after December (by the end of first quarter of 2003).

Progress

As of Aug 02, 2002 we have

- introduced a wrapper that shields persistent event store classes from transient world, this required cleaning a lot of transient code that deals with tags (tests in progress)
- made a detailed breakdown of navigational component sizes

- came up with a new design.

To be done:

- Implement new persistent classes. Testing. (Aug, Sep).
- Teach the wrapper how to deal with multiple versions of persistent classes. Testing. (Oct).
- Insure backwards compatibility (reprocessing events of “old” type, skimming “old” events”, mixing old and new events in one collection). Testing. (Nov, 1/2 Dec).
- Re-test everything (1/2 Dec, possibly Jan’03).
- Document and productise (Jan-Mar’03).

Manpower

~16 FTE-months: May-Sep 1 FTE-month each month (Yemi/Jacek). Oct’02-Feb’03: Daniel 100%, Yemi 80%, Jacek 25%.

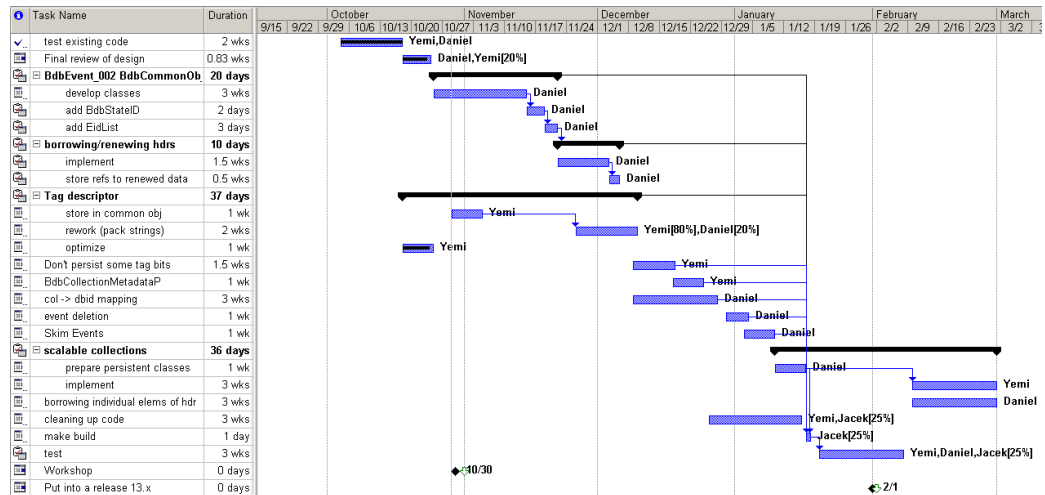


Figure 5-1. Tentative schedule

B

Example of Persistent Event in New System

Zoom to see details.

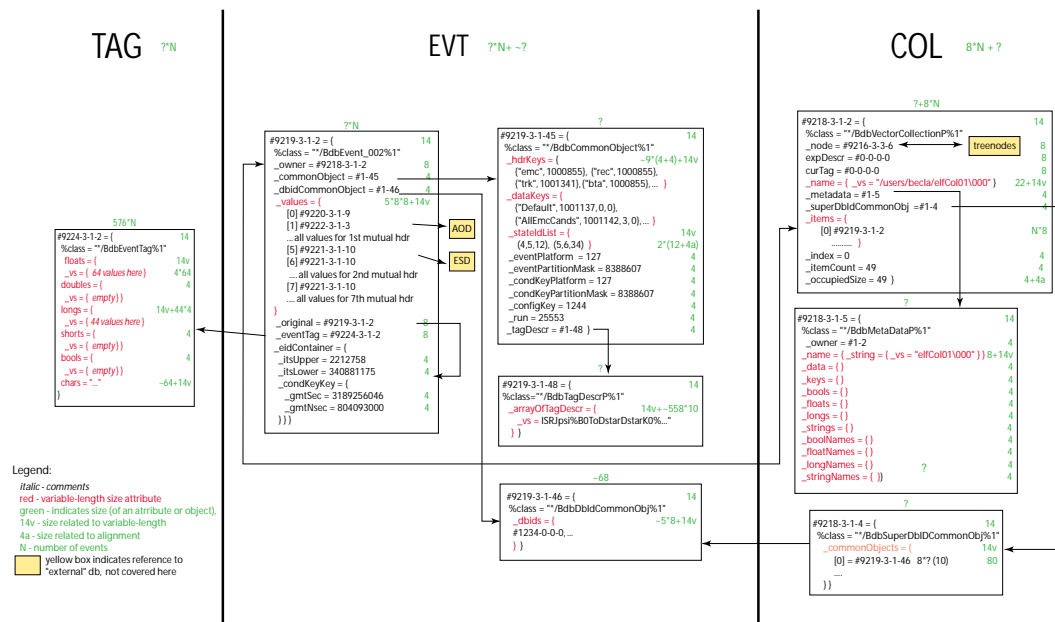


Figure B-1.



Highlights of New Design

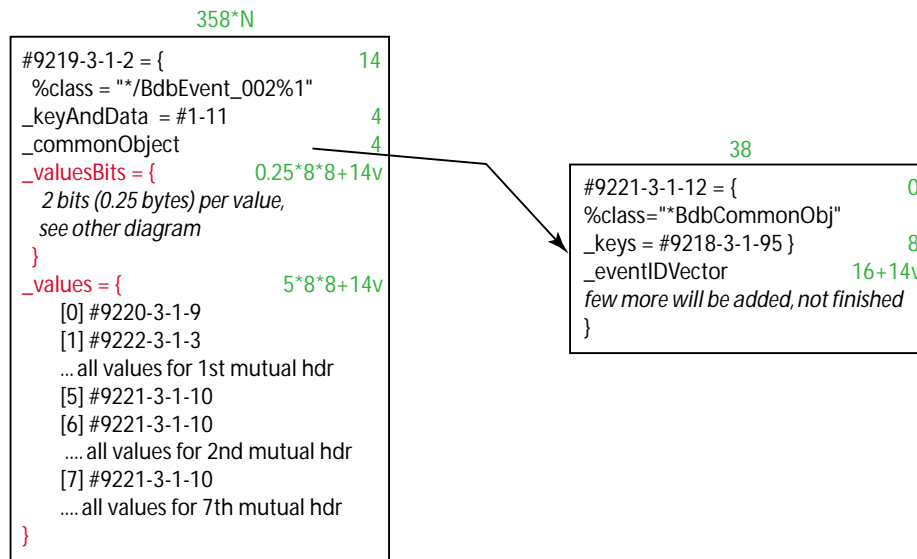


Figure C-1.

_valueBits will carry the following information (2 bits per one reference to data object):

- 10 - invalid
- 01 - borrowed
- 11 - owned

00 would indicate the end of data for current header.

For example, consider 4 headers, first header points to 3 data objects, second to 5 data objects, third to 2 data objects, and fourth to 6 data objects. 3rd data object of the

second header is renewed. 4rd data object for the last header is invalid. All other headers are borrowed. Here is how this would be represented in `_valuesBits` vector:

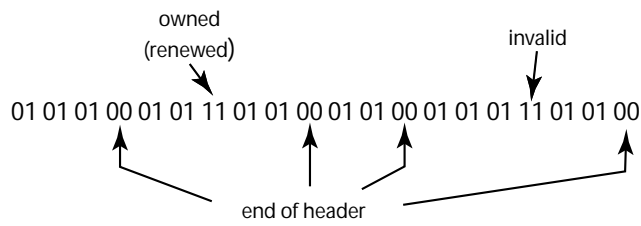


Figure C-2.

In this example, we would store only one pointer (the one that is owned), instead of 16.

D

Persistent Classes in Current System

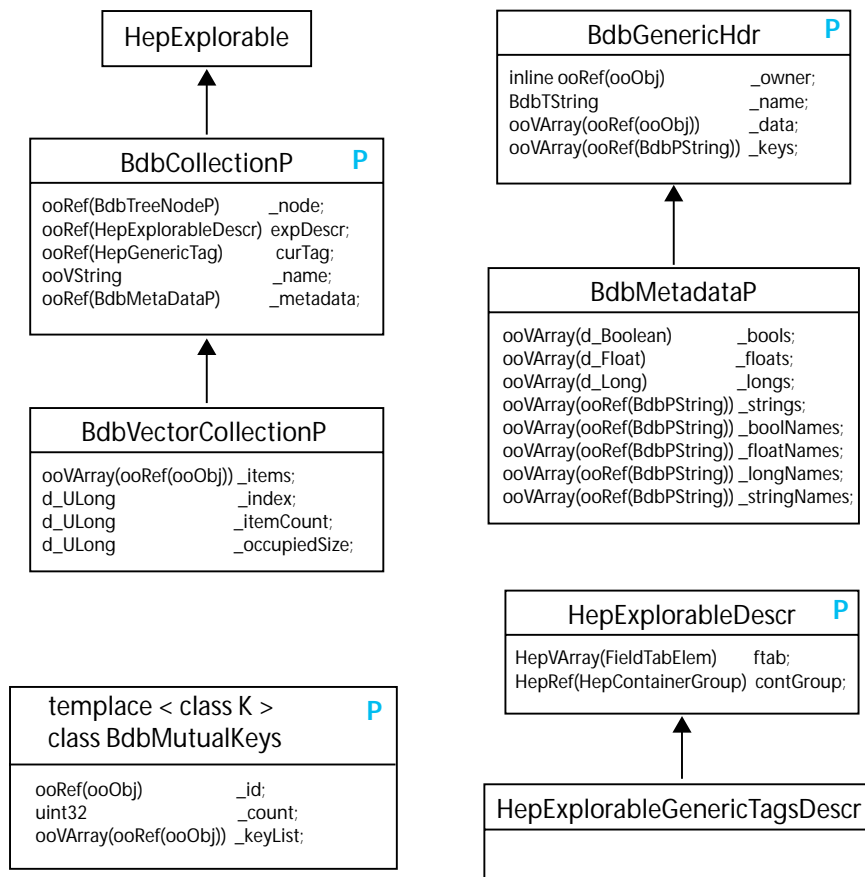


Figure D-1.

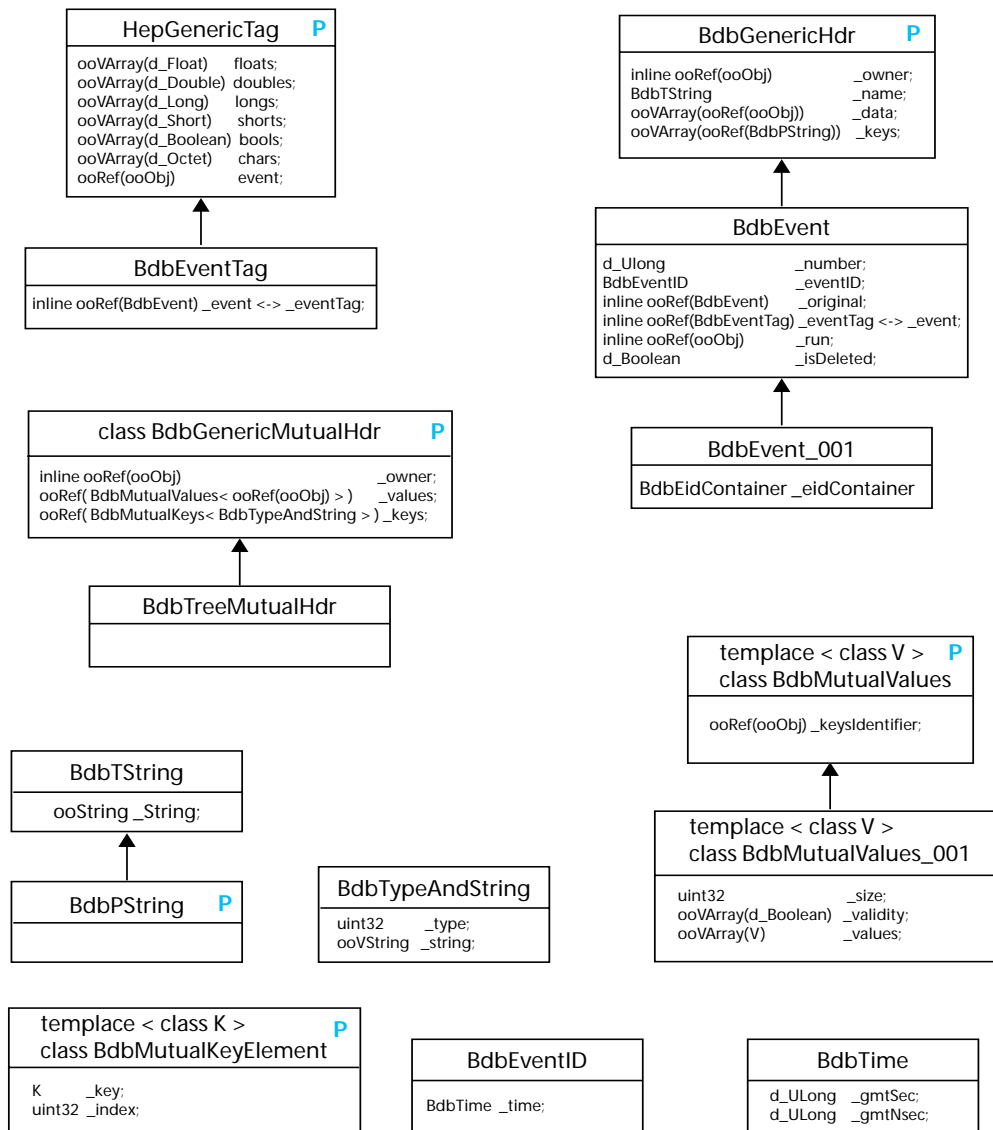


Figure D-2.



Persistent Classes in Proposed System

[this is not available yet]

