LightningObjects

lightweight, high performance, non-relational C++ object persistence

Table of Contents

Introduction

LightningObjects is a persistence solution directly based on a high performance key/value store implementation, namely, LMDB (symas.com/lmdb). LMDB, written in C, consists of roughly 10000 lines of code and shines, beyond an impressive feature set, with extremely good performance figures. In fact, the „L“ in LMDB represents the „Lightning“ which is also inherited into the name of the C++ library in front of you. This, of course, despite the fact that we have sacrificed some of LMDB's performance on the altar of object-oriented convenience. At the same time, however, we strove to keep the door open to optimal performance by offering special-purpose API.

On top of LMDB, LO offers an extensive, easy to use, fully object-oriented layer that allows you to seamlessly integrate solid transactional persistent storage into your C++ application while still maintaining an extremely low footprint. While offering features like full polymorphism and different storage strategies, LO always seeks to choose the execution path with the lowest overhead to ensure optimal performance.

If you are looking for an extremely light-weight, rock-solid, embeddable and transactional high-performance C++ object store, your search may have reached its goal. Feel free to join us in making the best out of this library and using it to create other great software.

Here's a short list of LO's features:

- fully object-oriented with support for inheritance and polymorphism
- consists of approx. 3000 lines of code in header files and 1200 lines in CPP files (if this is a feature)
- fully template-based with minimal runtime overhead
- convenient standard and extensive special-purpose API
- flexible schema compatibility handling (2-way compatibility with fallbacks)
- fully extensible.

Also don't forget to read up on LMDB's feature set which is inherited (in good object-oriented fashion) by LO.

Code

You're probably thursting for code by now. Here's a scoop:

```
START_MAPPING(Person, name, age)
  MAPPED_PROP(Person, BasePropertyAssign, std::string, name)
  MAPPED_PROP(Person, BasePropertyAssign, unsigned, age)
END_MAPPING(Person)
```

The snippet above makes the class Person known to LO in the simplest manner. This is done in terms of macros which are provided in LO's headers.

```
KeyValueStore *kv = lmdb::KeyValueStore::Factory{".", "test"};
kv->putSchema<Person>();
```

This snippet creates a database in the current directory. The database file will have the name „test“. Then we declare the schema, which only consists of the Person class for now. This is done immediately after opening the database.

```
auto wtxn = kv->beginWrite();
Person p(„George“, 22);
ObjectKey key = wtxn->putObject(p);
wtxn->commit();
```

It had to come to this – a Person is saved in the LO store! We could have done a lot more between the begin and the end of the transaction, though, and all would have been treated in an ACID (Atomic, Consistent, Isolated, Durable) manner.

Architecture + Principles

LMDB, upon which LO is built, is a key/value store with a rather simple API. Code-wise, LO consists of a template-based front end, which does all the object-relational mapping, and a backend which interfaces with LMDB. As a result, in principle, it should be fairly easy to port LO over to another key/value store with a similar feature set.

Databases

LMDB allows to create several “logical” databases within the same physical file (called “environment” in LMDB speak). LO makes use of this by maintaining one logical database for metadata (the database “schema”) and another one for regular object data.

Database Schema

When opening a database, the application has to declare its schema to LO using the *putSchema* function:

```
kv->putSchema<Colored2DPoint,
    ColoredPolygon,
    player::SourceDisplayConfig,
    player::SourceInfo,
    SomethingWithALazyVector>();
```

Each class given to the *putSchema* call must be mapped (more on mappings in the next chapter). Several *putSchema* calls can be executed sequentially, and each call will compare the the mappings with schema already saved in the database, determine compatibility for existing mappings and save the new mappings. There are 3 levels of compatibility:

- write compatible. Data can be read and written without loss of information

- read compatible. Data can be read, but writing may result in loss of information (e.g., there are data fields declared in the saved schema which do not exist in the application schema)
- incompatible. Data cannot be safely read or written

The goal here is, rather than following a strict versioning approach, to allow 2-way (forward and backward) as well as cross (accessing databases written by other applications) database compatibility. The compatibility status is recorded on a per-class basis and can be used by the application to determine the further course of action (e.g., write protect the whole database or certain parts of it, create a writable copy of the database, etc.).

The schema database uses class names as keys.

Object Data

Object data is maintained in a separate database which uniformly uses composite binary keys of the following form:

- ClassId. A 2-byte integer which identifies the class
- ObjectId. A 4-byte integer which identifies the object within the class
- PropertyId. A 2-byte integer which identifies individual object properties

Everything that goes into the object data space is keyed as described (struct name is `StorageKey`). However, despite the fact that there is a PropertyId, not every object property is saved under an individual key. Rather, there is a "primary" buffer (PropertyId 0), and an undetermined number of secondary buffers (PropertyId > 1). Each property has a storage strategy (declared as part of the mappings) which determines how it is stored. In principle, there are 3 property storage categories:

1. "shallow storage". The property value is serialized directly into the primary buffer
2. "shallow key". The property value (usually an object of a mapped type) is saved under a separate object key, and the key is also serialized into the primary buffer.
3. "property key". The property value is saved in a secondary buffer. The property value will usually belong to a collection and may itself consist of `StorageKey`s that refer to separately stored mapped objects.

The term "shallow" is sometimes substituted with "embedded".

Mappings

As already mentioned, the exact details about what (parts of) a class are to be made persistent and what storage strategy is to be applied need to be configured in the so-called mappings. This is done in a header file, normally using the LO-supplied macros. The following mapping features are available and will be explained later on in more detail:

- Inheritance (currently only single inheritance)
- Property mappings:

    - Simple values. Values that are serialized into the primary buffer without further structuring. All C primitive datatypes, but also extensible to new types
    - Mapped objects. References to other mapped objects. References can be by value or by shared_ptr.
    - vectors or sets of simple values
    - vectors of mapped objects. Again, references can by value or by shared_ptr
- Storage strategies:
    - Embedded (shallow) storage. Mandatory for simple values, available as an option for mapped object and vector of mapped objects.
    - Embedded Key storage. Available as an option for mapped object.
    - Property Key storage. Available as an option for vector of mapped objects. Mandatory for attached properties (see later)

Along with property storage strategies, there is also the concept of a property access strategy, which determines how the object property is read or written. The obvious alternatives here would be direct public access (all mapped properties are declared public) and function access (properties are accessed through functions). However, as of today, only direct access is implemented, which results in the requirement that all mapped properties be declared public.

In order to simplify mapping code, predefined property access / property storage pairs have been provided. Please consult the reference for details, and see them in use in the mapping examples below.

Examples

The first snippet shows a class mapping without inheritance, with 2 simple-valued properties:

```
START_MAPPING(FixedSizeObject, number1, numer2)
  MAPPED_PROP(FixedSizeObject, BasePropertyAssign, unsigned, number1)
  MAPPED_PROP(FixedSizeObject, BasePropertyAssign, unsigned, number2)
END_MAPPING(FixedSizeObject)
```

The next snippet shows a class mapping with the following property mappings:

1. property *displayConfig* holds a shared_ptr reference to an object of another mapped class, *SourceDisplayConfig*. It is mapped with the default strategy, namely, embedded key storage.
2. property *userOverlays* holds a vector of objects of a mapped class, namely, *Overlay*. It is mapped with the default strategy, namely, property key storage.

```
START_MAPPING(SourceInfo, displayConfig, userOverlays)
  MAPPED_PROP(SourceInfo, ObjectPtrPropertyAssign, SourceDisplayConfig, displayConfig)
  MAPPED_PROP(SourceInfo, ObjectPtrVectorPropertyAssign, Overlay, userOverlays)
END_MAPPING(flexis::player::SourceInfo)
```

The following snippet shows a class being declared as a subclass of another mapped class:

```
START_MAPPING_SUB(SomethingConcrete1, SomethingAbstract, description)
  MAPPED_PROP(SomethingConcrete1, BasePropertyAssign, std::string, description)
END_MAPPING_SUB(SomethingConcrete1, SomethingAbstract)
```

This final example shows the declaration of an abstract class:

```
START_MAPPING_A(SomethingAbstract, name)
  MAPPED_PROP(SomethingAbstract, BasePropertyAssign, std::string, name)
END_MAPPING(SomethingAbstract)
```

Object Ids

An obviously important issue is the handling of identifiers for persistent objects. As described further above, any persistent object is identified by an automatically assigned pair of ClassId / ObjectId (PropertyId only serves internal structuring of the object). The ClassId is uniquely assigned to each class declared to LO, and the ObjectId uniquely identifies each object created for a mapped class. The identifier is usually wrapped in an `ObjectKey`, which is returned or taken as parameter by many API functions.

Since maintaining ObjectKeys in the application can quickly become tedious, LO offers an alternative path along which ObjectKeys are maintained automatically and transparently. For this purpose, we use a feature of the C++ standard shared_ptr class, which allows storing a “custom deleter” inside the shared_ptr. Many LO APIs create such pointers or take them as argument. For this to work, all shared_ptrs that are newly created by the application and passed to the LO API must be created through either the *make_ptr* or the *make_obj* function. These functions do not create an ObjectId, but prepare the shared_ptr for receiving one later on – which would not be possible otherwise.

One special case exists for objects participating in the client side of a value-based relationship. Such objects are required to define an OBJECT_ID mapping for a property which is used by LO to store the ObjectId.

Persistent Object Lifetime

Generally speaking, persistent object lifetime starts with a *`put`* or *`save`* statement on any mapped object, and ends when it is explicitly deleted. In addition, there is a facility for “garbage-collecting” unreferenced objects, which will be presented in the following, along with subtle implications one should be aware of.

The garbage-collecting (also called "reference counting") facility comes into play when objects are placed inside other objects by means of single-valued or multi-valued relationships. As described in the Mapping section, such relationships come in different flavors:

1. value-based. Objects are copied into the relationship. Objects participating in such relationships are required to define an OBJECT_ID mapping.
2. shared_ptr-based. Objects are referenced through a shared_ptr, which also carries the object key

Additionally, there is the orthogonal aspect of embedded / keyed storage. With respect to lifetime, the following can be said:

1. For objects participating in a relationship with embedded storage, lifetime is implicitly managed because the objects live inside the enclosing object's shallow buffer and are thus deleted with the enclosing object. If such objects are multiply referenced (e.g., assigned to another object), a copy of the object is saved for each reference.
2. For objects participating in a value-based, keyed-storage relationship lifetime is not managed by LO in any way. This means that if an object is no longer referenced, e.g. because it was removed from a vector property, it is up to the application to ensure the object is also deleted from the KV store.
3. For objects participating in shared_ptr-based, keyed-storage relationships, lifetime management can be enabled by calling the `setRefCounting` function for the corresponding class. This will cause a separate key/value entry to be maintained for each object, which holds the reference count. In-memory, the reference count is maintained by the ObjectKey. When such an object is explicitly saved, the reference count is set to 1 (this can be suppressed if desired). Also, whenever such an object is placed into a shared_ptr relationship, its reference count is incremented. Equally, the reference count is decremented when the object is removed from a relationship. When the reference count falls to 0 while saving the reference, the object is automatically deleted. Also, objects loaded from the persistent store will be cached for the duration of a transaction, such that each object only resides in memory once. Finally, a top-level `deleteObject` will fail with an exception if the reference count is > 1

One ensuing implication of the above is that it should be avoided to place refcount-enabled objects in both shared_ptr-based and value-based relationships at the same time. Since the latter are not lifetime managed, it may happen that the object is deleted while still referenced in the value-based relationship.

Here's a code sequence demonstrating some of the above:

```
kv->setRefCounting<IFlexisOverlay>(); //normally done at DB startup. Also configures
                                      //subclasses RectangularOverlay and TimecodeOverlay

player::SourceInfo si;
RectangularOverlayPtr ro = kv::make_obj<RectangularOverlay>();
TimeCodeOverlayPtr  to = kv::make_obj<TimeCodeOverlay>();
si.userOverlays.push_back(to); //add to relationship
si.userOverlays.push_back(ro); //add to relationship

auto txn = kv->beginWrite();
txn->saveObject(ro);        //explicit save: refcount == 1
txn->saveObject(to, false); //explicit save: refcount suppressed
txn->saveObject(si, siKey); //implicit save: ro's refcount becomes 2, to's becomes 1
txn->commit();
```

The above example is for demonstration only. Otherwise, the last `saveObject` would have sufficed, the code shown sends `ro` and `to` to the datastore twice.

Collections

One special feature that to some degree goes beyond object mapping is that of *collections*. In LO speak, a collection is an unlimited-size, sequential structure that is stored under a top-level key (*StorageKey*) with a predefined ClassId and a unique ObjectId (called CollectionId in this context). A collection may consist of an unlimited number of *chunks*, with each chunk being assigned an individual PropertyId (aka ChunkId). Thus, unlimited-size containers can be realized with a limited memory footprint by only holding the currently accessed chunks in memory.

Collection definitions exist for the following types of values:

1. mapped objects. Objects are serialized into the chunk buffer. Only shallow storage is supported, i.e. object properties with embedded key or property key storage are ignored.
2. Simple values. All values for which ValueTraits are defined. This covers the primitive types of the C++ language, but may be extended to user-supplied types
3. Raw values. This option provides the best performance for large datasets. Primitive types that have fixed size across the ILP32, LLP64 and LP64 memory models are supported. This covers floating point types and the integer types *short*, *int* and *long long*.

Using Collections

A collection is initially created with a put statement:

```
vect.push_back(PersonPtr(new Person("Rudi")));
vect.push_back(PersonPtr(new Person("Sabine")));
collectionId = wtxn->putCollection(vect);
```

The lines above create an object collection with one chunk containing 2 *Person* objects (which is untypical, considering the collection API is designed for large datasets).

The collection can be extended using append statements:

```
vect.push_back(PersonPtr(new Person("George")));
vect.push_back(PersonPtr(new Person("Harry")));
wtxn->appendCollection(collectionId, vect);
```

This will create another chunk for the collection identified by *collectionId*.

Another way to extend a chunked collection is by using an appender:

```
auto appender = wtxn->appendCollection<Person>(collectionId, 1024*1024);
for(int i=0; i<10; i++) {
  stringstream ss;
  ss << "Test_" << i;
  appender->put(PersonPtr(new Person(ss.str())));
}
```

One special feature of an appender is the second parameter supplied when creating it (1024*1024 in the example above), which determines the chunk size.

Similar API exist for reading a chunked collection.

Raw Data Collections

Raw data collections deserve special mentioning because they offer the unique capability of true zero-copy persistent data access. Already, when writing to a collection, memory overhead is minimized by preallocating memory from the database, which is directly mapped to disk by LMDB. With raw data collections, it is even possible to request preallocated chunk space from LO and put data into it from within the application, thus again removing a level of copying. Finally, copying during write can be completely avoided by using the LMDB WRITE_MAP (advanced) configuration option.

The snippet below shows a data chunk being allocated and written:

```
const double *vp = nullptr;
collectionId = wtxn->putDataCollection(&vp, 1000);
for (unsigned i = 0; i < 1000; i++)
  vect[i] = 1.44 * i;
```

When reading from a raw data collection, copying can be avoided by requesting data along the chunk boundaries. If a read request does not cross chunk boundaries, a pointer is returned that directly references the memory-mapped disk region that holds the data. This region will usually be write-protected by the OS, such that every attempt to alter data directly will result in a segmentation violation. Also, the pointer is only valid until the end of the transaction.

Other Specialties

Attached Properties

Attached properties are a means to extend the persistent storage of a given class without changing the class itself. Regular property mappings can be declared for “virtual” properties, i.e. properties that only exist in the form of a property ID.

Iterator Properties

Iterator properties are a means to transparently embed persistent functionality into an application model without making the application model aware of KV's presence. By application model we mean the classes that model the application problem domain, which are normally not concerned with persistence aspects. Sometimes, however it is impractical to delegate all persistence-related code to one central location.

As an example, you may want an application model class to iterate over a persistent collection – which is only possible using special LO API. To achieve this, the application would declare a suitable interface, like:

```
template <class T> class ObjectHistory {
public:
  virtual T& getHistoryValue(uint64_t bufferPos) = 0;
};
```

Then, you would define an LO-aware implementation class which inherits from both the application interface and the LO-provided class *KVPropertyBackend*.

```
template<typename T>
struct KVObjectHistory : public ObjectHistory<T>, public KVPropertyBackend
{
  T t;
  T& getHistoryValue(uint64_t bufferPos) override {
    return t;
  }
};
```

The class that holds the LO-enabled property might look like this:

```
struct SomethingWithAnObjectIter
{
  std::string name;
  shared_ptr<ObjectHistory<CollectionElement>> history;
};
```

And finally, you would declare the mapping like so:

```
START_MAPPING(SomethingWithAnObjectIter, history)
  MAPPED_PROP_ITER(SomethingWithAnObjectIter, CollectionIterPropertyAssign,
CollectionElement, KVObjectHistory, ObjectHistory, history)
END_MAPPING(SomethingWithAnObjectIter)
```

This will cause LO to replace the contents of the *history* property with a ready-initialized *KVObjectHistory* object. This object receives the ObjectId of the collection and has access to the full LO API. Still, the application model is kept clean without references to the underlying persistence technology.