NodeStore, A Case Study in Refactoring

Vincent Falco
OpenCoin, Inc.
vinnie.falco@gmail.com

Why?

Soon, our software including the **rippled** server and the client side wallet code will be in the spotlight on the global stage. Programmers, IT managers, maverick developers, and many more will be looking at everything that we are doing. We're building the most important piece of software that the world has ever seen. It deals with **money**, which makes the world go round!

If there was ever a piece of software for which readability, maintainability, and overall aesthetics was of paramount importance, it would be the software developed by OpenCoin. The Ripple payment system will help redefine society well beyond our lifetimes.

Every developer should remember that the world will be looking at their code.

What?

Code refactoring is a "disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior", undertaken in order to improve some of the *nonfunctional* attributes of the software. Advantages include improved code readability and reduced complexity to improve the maintainability of the source code, as well as a more expressive internal architecture or object model to improve extensibility:

> "By continuously improving the design of code, we make it easier and easier to work with. This is in sharp contrast to what typically happens: little refactoring and a great deal of attention paid to expediently adding new features. If you get into the hygienic habit of refactoring continuously, you'll find that it is easier to extend and maintain code."
> —Joshua Kerievsky, *Refactoring to Patterns*

Code is easier to understand and maintain when it is refactored.

The NodeStore

The `NodeStore` provides an interface that stores, in a persistent database, the collection of `NodeObject` that **rippled** uses as its primary representation of ledger items.

The original implementation used SQLite, which was quickly identified as a bottleneck for the performance of the `NodeStore`. To improve the performance, developers changed the original code to use LevelDB. The changes were made in hasty fashion, typical of fast paced startup environments, incurring a high level of technical debt (another explanation). It became clear that trying out a third backend using the same quick technique was not practical. To meet the business needs, it was necessary to refactor the `NodeStore` interface to support interchangeable backends before adding new features.

Evolving business needs are more easily met when code is already well designed.

Objective

Because the `NodeStore` plays such a crucial role in the operation and performance of the **rippled** server, we volunteered to improve the interface, to not only to fulfill the business requirements but also to serve as an example and reference for future development. The techniques applied in the refactoring of `NodeStore` should be documented and adopted as OpenCoin programming standards and promulgated throughout the corporate culture. Note that most of these refactoring approaches and techniques are applicable not only to the C++ code, but also the JavaScript code (although with accommodations made for language differences).

Source Material

Accompanying this presentation is the original source material for both the "before" and "after" version of the source code in question. The header and source files have been concatenated together, with each version in its own separate file.

Nouns and Verbs

Computers don't care about identifier names. But humans need them. The choice of names may seem a trivial exercise. But words shape the way we think and communicate ideas. Careful selection of identifier names facilitates thoughts and conversations about code. They also serve as a form of self-documentation (although they are not a substitute for proper documentation).

The original class was called `HashedObjectStore`. Since "node" was used in various places to describe a key/value based object, the name was changed to `NodeStore` for brevity and ease of pronunciation. Correspondingly, we have `NodeObject` instead of `HashedObject`.

It's easy to find plenty of advice on tips for naming variables.

Choosing the right names is just as important as having a correct implementation.

Original Code

Due to the fast paced nature of startup environments, the first iteration of code usually incurs significant technical debt. We begin with a review of the original code and the problems with it. These are marked in the program listing in "Addendum A" in this document, in bright green comments which begin with brackets "[ ]" with the note number inside:

1. Don't advertise unnecessary features. Only one place in the code base constructs a `HashedObjectStore`. Rather than passing the `cacheSize` and `cacheAge` parameters, it is easier to just put them in the constructor initializer list. An example of You Arent Gonna Need It.
2. There is no destructor for `HashedObjectStore`. If `mWriteSet` isn't empty we could be

left with unflushed data.

3. Exposing the function `isLevelDB` implies that calling code needs to care about the choice of backend. This adds an extra burden to anyone using the class.
4. Why are the backend-specific member functions public? We prefer to expose as little of a class as possible.
5. Pushing the responsibility of waiting for writes, onto the caller of the class is bad design. It should happen in the destructor.
6. By placing LevelDB-specific declarations in the header file, all the header material of LevelDB needs to be included just to compile the `NodeStore` class. This is an unwanted dependency.
7. The `std::vector` type is repeated, it should be given a descriptive name.
8. *ctor-initializers* should each be on their own line for legibility. This also encourages commenting on the line.
9. A magic number appears here and gets repeated.
10. We've introduced an undesirable dependency on `theConfig` which will complicate the unit test and add an unnecessary moving part.
11. Returning from the middle of the function is an unsustainable style. As a function grows, returning from the middle makes it more and more difficult to analyze.
12. The serialized format of a `HashedObject` is implied here but lacks any sort of annotation. Anyone mistakenly changing this code would break all existing databases.
13. Lack of comments hide an interesting fact, there are 4 bytes in the serialized format that get written out but are not read back in.
14. Redundant code, and even worse its the code that does serialization and defines the database format.
15. This batch write feature was added in a way that it could not be reused nor disabled if desired.
16. This unwanted dependency on both the `JobQueue` and the `Application` object means we can't write a unit test for `HashedObjectStore` without also initializing the `Application` object, itself a procedure ripe with problems.
17. A classic example of why returning from the middle is bad practice. The call to `canonicalize` is redundant. This example might seem trivial, but imagine if this function was the size of `RippleCalc`, and we needed to do many complex steps instead of just a call to `canonicalize`. One might even be tempted to use a "goto" statement in this situation...
18. We have duplicated code here because batch writing was never made into a class. What if we want a third backend?
19. More duplicated batch writing code.
20. Another dependency on `theConfig`. A proper unit test would want to run the tests with both values of `RUN_STANDALONE`, which is impossible because...
21. Global variables in the most unexpected place, and in a way that makes writing a unit test impossible.
22. Missing unit tests.

New Code

We refactored the entirety of `HashedObjectStore`, renaming it to `NodeStore`. The goal was to allow the backend to be chosen through a configuration. An additional goal was to bring the code up to the proposed OpenCoin programming standards to serve as a reference for future work. We review the changes in the program listing corresponding to the new code which is in "Addendum B" in this document with green comments and numbered brackets:

1. Javadoc style comments. These are automatically extracted by the Doxygen tool to produce the HTML formatted Ripple C++ API reference.
2. Magic numbers changed into a language constant with a descriptive name
3. Descriptive name given to a commonly used type
4. The deserialization logic is consolidated into a single well designed class
5. The `DecodedBlob` class can detect some errors in deserialization, a new feature. It was designed with the Single Responsibility Principle; to decode blobs. This is an instance of the larger principle of Responsibility Driven Design.
6. There is now only a single place in the program where `NodeObject` is constructed from serialized data
7. As per C++ best practices, this class exposes the least amount of interface needed to do the job
8. The `EncodedBlob` consolidates serialization of `NodeObject` into one place. This is the Single Responsibility Principle.
9. The `RecycledObjectPool` class was written and added to Beast (since it is generic) and allows the `NodeStore` to recycle the buffers required for serialization, reducing the load on the memory allocator (a resource which takes a global lock).
10. The `Scheduler` interface eliminates the previous dependency on the `Application` object and `JobQueue`, while still allowing the `NodeStore` implementation to fulfill a business requirement. Specifically, that batch writing occur on a separate managed thread.
11. The batch writing feature, a business requirement, is consolidated into its own class with a well defined interface which can be unit-tested independently of the rest of the system. This is an example of preferring *Composition over Inheritance*, or the Composite Reuse Principle.
12. The `Backend` interface is used by the `NodeStore`. Since the class is completely abstract, subclasses are free to choose their own implementations. Furthermore, no implementation details required by `Backend` subclasses are leaked into the header material seen by clients of `NodeStore`. This follows the Open/closed principle. The behavior of the `NodeStore` can be modified by creating a new `Backend`, which does not require modification of the `NodeStore` class itself.
13. Explicit return codes communicate information needed for business requirements, such as the existence of data known to be corrupted.
14. The comments for the `Backend` destructor make clear the responsibilities of derived classes, and what behaviors callers can expect. We use comments because C++ does not directly support all the features needed for Design by contract.
15. We chose to pass keys as `void*` instead of `uint256`, since it allows the caller more flexibility

in how the data is provided. A `uint256` can always be presented as a `void*` by passing `uint256::begin()`. However, there is no way to take a generic piece of memory and pass it as a `uint256` without invoking a constructor and a memory copy.

16. We chose *fetch* and *store* as the verbs used in member function names. Previously, there was *store*, *retrieve*, *fetch*, and *write*. Since the management of I/O resources is important for business requirements, we reserve the word *write* to indicate where I/O is taking place immediately.
17. The `NodeStore` lets the caller provide the `Scheduler`, instead of depending on the `Application` object.
18. The return value is well documented.
19. During the refactoring, it became clear that the storage required for the creation of the resulting `NodeObject` could be taken from the caller. Previously, the store operation resulting in the creation and destruction of three temporary memory buffers. This is now reduced to zero.
20. The format of the serialized `NodeObject` is defined in a comment.
21. We note that there are 4 bytes in the serialized output which are not well defined
22. Instead of repeating ugly calls to `boost::make_shared<>`, we simply give `NodeObject` a `createObject` function. This exemplifies Don't Repeat Yourself. We additionally made the `NodeObject` constructor effectively private, so that no one can create an object without going through our `createObject` function. This gives us a strong invariant: there's only one way to produce these objects, should we want to change how they are made.
23. We use Beast's `static_bassert` feature to detect unexpected architecture changes at compile time.
24. The `fetch` function implements the largest amount of Ripple-specific business logic, checking various caches and multiple backends. We found the numerous `return` statements in the middle of the function an impediment to achieving clarity in the understanding of the function. First, we rewrote the function have a single point of exit. During this refactoring, it was possible to eliminate the redundant calls to `canonicalize`. If this code needs to be changed, it will be easier as a result of the refactoring. Finally, we carefully annotated the thinking behind each important line of code so that if someone unfamiliar with the code base steps through it, they will have a profound understanding of it's behavior.
25. It was necessary to split some code into its own function to eliminate redundancy. This a Composed Method refactoring.
26. The Beast `ScopedPointer<>` container provides RAII management of a dynamically allocated object, ensuring that it gets deleted when the enclosing class is deleted.
27. Common code for unit tests is factored into its own class.
28. The unit test code provides its own `Scheduler` so we don't need to use the one from the `Application` object. Our hard work refactoring is now yielding dividends!
29. A set of basic tests make sure that none of the things we depend on in later unit tests break.
30. We unit test the `Backend` interface directly. This is now practical since `Backend` subclasses have no upstream dependencies.

31. Now we can do performance testing in a controlled environment, to make sure we haven't introduced a problem, or to test new implementations.
32. We can test the `NodeStore` itself, with the additional caching and other business logic, without needing `theConfig` or the `Application` object to initialize.
33. Even the unit tests were written with best practices in mind. The `NodeStore` unit test exercises everything through a generic function that works with any backend.

What Can We do?

Code Should Be Easy To Understand

By paying attention to white space, choice of names, consistency in capitalization (capitalize the first letter for type names, lower case first letter for variables, and always use camel case), we can go a long way towards making the code readable. Breaking up long functions into smaller functions with very descriptive names also helps.

Comments are essential for describing what a class does, or for documenting the parameters or nuanced behaviors of public member functions. But comments are not always needed, it is possible to write readable code without comments if you follow these guidelines.

Code Should Be Written to Minimize the Time for Someone Else To Understand It

Each coding style or good coding practice might seem subtle or trivial on its own. But when you take them all together and apply their principles continuously over time, the results increase productivity and help to reliably achieve business goals. Here is a partial list of good programming practices:

- Write code in a clear, self-documenting style but use comments where necessary.
- Use Test Driven Development. It encourages designing interfaces first, before implementation.
- Don't Repeat Yourself, “D.R.Y.”. Put redundant code in a class so it can be re-used and unit tested.
- Expose as little of a class as possible. Prefer private over protected. Prefer protected over public. The smaller the interface footprint, the easier it is to write unit tests and comprehend the operation of the class. This is the Interface Segregation Principle.
- Make classes depend on as few external classes or routines as possible. Ideally, no dependencies.
- Write code that is first correct, and do not worry about optimizations until later. Changes that improve performance usually make the program harder to work with. This slows development. This would be a cost worth paying if the resulting software were quicker, but usually it is not. The performance improvements are spread all around the program, and each improvement is made with a narrow perspective of the program’s behavior.

The **rippled** repository and the Beast repository have a `CodingStyle.md` document which is a great starting point for writing well designed code.

Immediately Polish New Code

The best time to go over a piece of newly written working code and polish it up is right after it has been written and tested. This is the time to rename things to make sense, add or update the comments, eliminate unnecessary functions and redundant code, make implementation details private, streamline the exposed interface, and eliminate external dependencies.

If we leave this process for some unspecified point in the future, then it will be much more difficult to revise because the code and algorithms are no longer fresh in the mind of the programmer. A quick refactoring after writing the initial code gives us a large benefit for only a marginal cost. Waiting until later increases the cost, or defers it indefinitely if development is always in a "crisis mode."

The Rule of Three

All development times should be increased by a factor of three. Right now, we put in an hour of development and it ends there. Instead, for each hour of development we should add:

- An hour for refactoring, with a final peer review by another programmer. The first draft of code usually always needs some work to eliminate technical debt. Doing a refactoring pass when it is fresh in the mind of the author is ideal. Class names, types, and parameter names can be rethought. More comments can be added. Input from other developers can help at this point.
- An hour for writing unit tests. They say that the ratio of unit test code to business code is one to one. It follows that for each hour of development, an hour of unit test writing is necessary. During the writing of the unit test, it is likely that the inteface will be further refined to support the test. This is itself a form of factoring, and makes the newly written code more robust.

While it might appear counter-intuitive that increasing the time taken for development will decrease the time requirement for deliverables, remember that significant levels of technical debt impose a "tax" on development. It is the price that must continually be paid when a programmer has to work in functions with little or no documenting comments, that also does not adhere to good programming principles. The author notes that there is a considerable learning curve for gaining a sufficient understanding of core business logic code like the ripple calculation or the ledger closing process, to the extent that there is only one employee of sufficient proficiency to effect changes and fixes.

> **"If a shortened and bug-free release schedule was the goal of rapid development with an absence of refactoring, unit tests, or attention to good programming principles, then we have surely failed."**
>
> — *Vincent Falco*

Education

There is plenty of literature on good programming practices. Journal articles, columns, and even resources such as the #c++ or #boost channels on IRC's Freenode network. The author is always connected and regularly consults these resources with relevant questions in order to produce the highest quality code. For example:

```
<Vinnie_win> If I have a 32-bit signed integer and I want to
reinterpret the bits as a 32-bit unsigned integer without doing
any conversions, do I use static_cast<>, reinterpret_cast<>, or
either?

<adamm> << static_cast<unsigned>(-1);


<Vinnie_win> michlemken-1: According to StackExchange there's
```

```
no portable way to friend make_shared that works on all
compilers.
```

<michlemken-1> `Vinnie_win, ... template<...> friend ? std::make_shared(...)`

<PlasmaHH> `Vinnie_win: you can only befriend functions, not templates, so you have to use an instance of make_shared`

<Vinnie_win> `If I want to memcpy the contents of a std::string, without the null terminator, whats the 'proper' way to get a pointer to the beginning of the std::string in question?`

<japro> `C++98, C++11 also have the string::data() member`

<Vinnie_win> `I prefer data(). Because I am unconcerned with terminators.`

<tsimpson> `you'll still get the \0 in all likelihood`

<tsimpson> `but at least conceptually you're better off with data()`

<japro> `well yeah, you should probably be using std::copy instead of memcpy anyway`

OpenCoin developers should continually dedicate some time to studying the literature on good programming practices. These skills add significant value not only to the company but also the individual. It's a win-win.

The External Resources section contains a great starting point for learning more about writing well designed code.

Code Review, Code Audit

All code should be subjected to informal code reviews that are not overly time consuming, with the understanding that we're not looking to criticize anyone or play competitive games but instead to recognize that hearing the opinions of our colleagues can provide valuable insights. A code review could be as simple as looking over a header file containing interfaces just to look for code smells. We all have our own strengths, and a shared desire to make the best software possible. Peer reviewing code improves our process.

Furthermore, the open sourcing of the **rippled** software will introduce new requirements on every developer to enhance their code reviewing skills to work with outside contributors. Outside contributions must be subjected to a code audit process not only to ensure consistent quality but also to detect malicious intent.

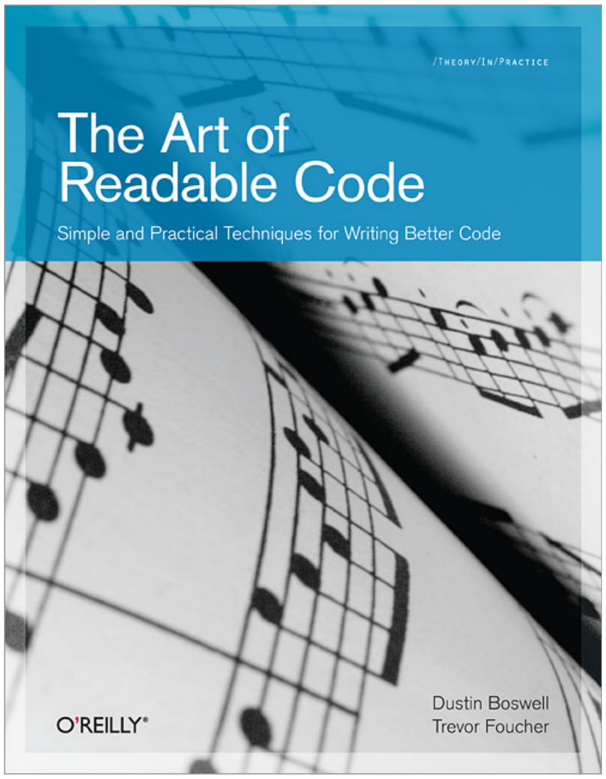
Benefits of Refactoring

- Improves the design of the software, reversing the increasing tendency of code to develop difficult to fix bugs.
- Helps you find bugs, and helps you program faster.
- Refactoring makes software easier to understand for everyone, instead of having only one person who can work in the software portions critical to the business.
- Well written code attracts higher quality contributions, and increase pride in workmanship.
- The techno-literate will more easily accept Ripple as the "Internet Payments" system that deals with money when they see well designed code.
- Code looks more professional once it is open sourced.

External Resources

As developers we should strive to grow our skills and knowledge. If you see this book lying around you should read it. Or you can order it for home:

The Art of Readable Code, Trevor Foucher and Dustin Boswell



Here's a list of more invaluable resources for improving code design skills:

- Good Programming Principles
- Refactoring: Improving the Design of Existing Code, Martin Fowler.
- Design Patterns: Elements of Reusable Object-Oriented Software, various
- C++ Coding Standards: 101 Rules, Guidelines, and Best Practices, Sutter and Alexandrescu
- Effective C++: 55 Specfic Ways to Improve Your Programs and Designs, Scott Myers
- More Effective C++: 35 New Ways to Improve Your Programs and Designs, Scott Myers
- The Mythical Man-Month: Essays on Software Engineering, Frederick P. Brooks Jr.
- Freenode IRC: **##c++**, **#boost**, and **##javascript** channels

Addendum A

`ripple_HashedObjectStore.cpp`

```
//------------------------------------------------------------------------------
/*
    Copyright (c) 2011-2013, OpenCoin, Inc.
*/
//==============================================================================

#ifndef RIPPLE_HASHEDOBJECTSTORE_H
#define RIPPLE_HASHEDOBJECTSTORE_H

/** Persistency layer for hashed objects.
*/
// VFALCO TODO Move all definitions to the .cpp
class HashedObjectStore : LeakChecked <HashedObjectStore>
{
public:
    /// [1] Unnecessary parameters, only 1 caller constructs this object
    HashedObjectStore (int cacheSize, int cacheAge);

    /// [2] No destructor. What happens if mWriteSet isn't empty?

    /// [3] Exposed implementation detail
    bool isLevelDB ()
    {
        return mLevelDB;
    }

    float getCacheHitRate ()
    {
        return mCache.getHitRate ();
    }

    bool store (HashedObjectType type, uint32 index, Blob const& data,
                uint256 const& hash)
    {
        if (mLevelDB)
            return storeLevelDB (type, index, data, hash);

        return storeSQLite (type, index, data, hash);
    }

    HashedObject::pointer retrieve (uint256 const& hash)
    {
        if (mLevelDB)
            return retrieveLevelDB (hash);

        return retrieveSQLite (hash);
    }

    /// [4] Were these supposed to be public?
    bool storeSQLite (HashedObjectType type, uint32 index, Blob const& data,
                      uint256 const& hash);
    HashedObject::pointer retrieveSQLite (uint256 const& hash);
    void bulkWriteSQLite (Job&);

    bool storeLevelDB (HashedObjectType type, uint32 index, Blob const& data,
                       uint256 const& hash);
    HashedObject::pointer retrieveLevelDB (uint256 const& hash);
    void bulkWriteLevelDB (Job&);

    /// [5] Are callers responsible for knowing about write batching?
    void waitWrite ();

    void tune (int size, int age);
    void sweep ()
    {
        mCache.sweep ();
        mNegativeCache.sweep ();
    }
    int getWriteLoad ();

    int import (const std::string& fileName);

private:
    /// [6] Everyone who needs NodeStore now needs to see leveldb declarations
    static HashedObject::pointer LLRetrieve (uint256 const& hash, leveldb::DB* db);
    static void LLWrite (boost::shared_ptr<HashedObject> ptr, leveldb::DB* db);
    static void LLWrite (const std::vector< boost::shared_ptr<HashedObject> >& set, leveldb::DB* db);

private:
    TaggedCache<uint256, HashedObject, UptimeTimerAdapter>  mCache;
    KeyCache <uint256, UptimeTimerAdapter> mNegativeCache;

    boost::mutex                mWriteMutex;
    boost::condition_variable   mWriteCondition;
    int                         mWriteGeneration;
    int                         mWriteLoad;

    /// [7] This type name is repeated often
    std::vector< boost::shared_ptr<HashedObject> > mWriteSet;
    bool mWritePending;
    bool mLevelDB;
    bool mEphemeralDB;
};

#endif

//==============================================================================

/// [8] ctor-initializers are hard to read, hard to add comments.
HashedObjectStore::HashedObjectStore (int cacheSize, int cacheAge) :
    mCache ("HashedObjectStore", cacheSize, cacheAge),
    mNegativeCache ("HashedObjectNegativeCache", 0, 120),
    mWriteGeneration (0), mWriteLoad (0), mWritePending (false), mLevelDB (false),
    mEphemeralDB (false)
{
    /// [9] A magic number
    mWriteSet.reserve (128);

    /// [10] Dependency on "theConfig"
    if (theConfig.NODE_DB == "leveldb" || theConfig.NODE_DB == "LevelDB")
        mLevelDB = true;
    else if (theConfig.NODE_DB == "SQLite" || theConfig.NODE_DB == "sqlite")
        mLevelDB = false;
    else
    {
        WriteLog (lsFATAL, HashedObject) << "Incorrect database selection";
        assert (false);
    }

    if (!theConfig.LDB_EPHEMERAL.empty ())
        mEphemeralDB = true;
}

void HashedObjectStore::tune (int size, int age)
{
    mCache.setTargetSize (size);
    mCache.setTargetAge (age);
}

void HashedObjectStore::waitWrite ()
{
    boost::mutex::scoped_lock sl (mWriteMutex);
    int gen = mWriteGeneration;

    while (mWritePending && (mWriteGeneration == gen))
        mWriteCondition.wait (sl);
```

```
}

int HashedObjectStore::getWriteLoad ()
{
    boost::mutex::scoped_lock sl (mWriteMutex);
    return std::max (mWriteLoad, static_cast<int> (mWriteSet.size ()));
}

// low-level retrieve
HashedObject::pointer HashedObjectStore::LLRetrieve (uint256 const& hash, leveldb::DB* db)
{
    std::string sData;

    leveldb::Status st = db->Get (leveldb::ReadOptions (),
                                  leveldb::Slice (reinterpret_cast<const char*>
                                  (hash.begin ()), hash.size ()), &sData);

    if (!st.ok ())
    {
        assert (st.IsNotFound ());

        /// [11] Returning from the middle of the function
        return HashedObject::pointer ();
    }

    /// [12] This defines the serialized format of a HashedObject
    const unsigned char* bufPtr = reinterpret_cast<const unsigned char*> (&sData[0]);
    uint32 index = htonl (*reinterpret_cast<const uint32*> (bufPtr));
    int htype = bufPtr[8];

    return boost::make_shared<HashedObject> (static_cast<HashedObjectType> (htype), index,
            bufPtr + 9, sData.size () - 9, hash);
}

// low-level write single
void HashedObjectStore::LLWrite (boost::shared_ptr<HashedObject> ptr, leveldb::DB* db)
{
    HashedObject& obj = *ptr;
    Blob rawData (9 + obj.getData ().size ());
    unsigned char* bufPtr = &rawData.front ();

    *reinterpret_cast<uint32*> (bufPtr + 0) = ntohl (obj.getIndex ());

    /// [13] Here, we're writing out 4 bytes that are not read back in
    *reinterpret_cast<uint32*> (bufPtr + 4) = ntohl (obj.getIndex ());
    * (bufPtr + 8) = static_cast<unsigned char> (obj.getType ());
    memcpy (bufPtr + 9, &obj.getData ().front (), obj.getData ().size ());

    leveldb::Status st = db->Put (leveldb::WriteOptions (),
                                  leveldb::Slice (reinterpret_cast<const char*>
                                     (obj.getHash ().begin ()), obj.getHash ().size ()),
                                  leveldb::Slice (reinterpret_cast<const char*> (bufPtr),
                                     rawData.size ()));

    if (!st.ok ())
    {
        WriteLog (lsFATAL, HashedObject) << "Failed to store hash node";
        assert (false);
    }
}

// low-level write set
void HashedObjectStore::LLWrite (const std::vector< boost::shared_ptr<HashedObject> >& set,
                                 leveldb::DB* db)
{
    leveldb::WriteBatch batch;

    BOOST_FOREACH (const boost::shared_ptr<HashedObject>& it, set)
    {
        const HashedObject& obj = *it;
        Blob rawData (9 + obj.getData ().size ());
        unsigned char* bufPtr = &rawData.front ();

        /// [14] Redundant code
        *reinterpret_cast<uint32*> (bufPtr + 0) = ntohl (obj.getIndex ());
        *reinterpret_cast<uint32*> (bufPtr + 4) = ntohl (obj.getIndex ());
        * (bufPtr + 8) = static_cast<unsigned char> (obj.getType ());
        memcpy (bufPtr + 9, &obj.getData ().front (), obj.getData ().size ());

        batch.Put (leveldb::Slice (reinterpret_cast<const char*>
                        (obj.getHash ().begin ()), obj.getHash ().size ()),
                   leveldb::Slice (reinterpret_cast<const char*>
                        (bufPtr), rawData.size ()));
    }

    leveldb::Status st = db->Write (leveldb::WriteOptions (), &batch);

    if (!st.ok ())
    {
        WriteLog (lsFATAL, HashedObject) << "Failed to store hash node";
        assert (false);
    }
}

bool HashedObjectStore::storeLevelDB (HashedObjectType type, uint32 index,
                                      Blob const& data, uint256 const& hash)
{
    // return: false = already in cache, true = added to cache
    if (!getApp().getHashNodeLDB ())
        return true;

    if (mCache.touch (hash))
        return false;

#ifdef PARANOID
    assert (hash == Serializer::getSHA512Half (data));
#endif

    HashedObject::pointer object =
        boost::make_shared<HashedObject> (type, index, data, hash);

    if (!mCache.canonicalize (hash, object))
    {
        /// [15] Non-reusable batch write feature
        boost::mutex::scoped_lock sl (mWriteMutex);
        mWriteSet.push_back (object);

        if (!mWritePending)
        {
            mWritePending = true;

            /// [16] Dependency on the JobQueue
            getApp().getJobQueue ().addJob (jtWRITE, "HashedObject::store",
                    BIND_TYPE (&HashedObjectStore::bulkWriteLevelDB, this, P_1));
        }
    }

    mNegativeCache.del (hash);
    return true;
}

void HashedObjectStore::bulkWriteLevelDB (Job&)
{
    assert (mLevelDB);
    int setSize = 0;

    while (1)
    {
        std::vector< boost::shared_ptr<HashedObject> > set;
        set.reserve (128);

        {
            boost::mutex::scoped_lock sl (mWriteMutex);

            mWriteSet.swap (set);
            assert (mWriteSet.empty ());
            ++mWriteGeneration;
            mWriteCondition.notify_all ();

            if (set.empty ())
            {
                mWritePending = false;
                mWriteLoad = 0;
                return;
            }

            mWriteLoad = std::max (setSize, static_cast<int> (mWriteSet.size ()));
            setSize = set.size ();
        }

        LLWrite (set, getApp().getHashNodeLDB ());

        if (mEphemeralDB)
            LLWrite (set, getApp().getEphemeralLDB ());
    }
}

HashedObject::pointer HashedObjectStore::retrieveLevelDB (uint256 const& hash)
{
    HashedObject::pointer obj = mCache.fetch (hash);

    if (obj || mNegativeCache.isPresent (hash) || !getApp().getHashNodeLDB ())
        return obj;

    if (mEphemeralDB)
    {
        obj = LLRetrieve (hash, getApp().getEphemeralLDB ());

        if (obj)
        {
            /// [17] Returning from the middle
            mCache.canonicalize (hash, obj);
            return obj;
        }
    }

    {
        LoadEvent::autoptr event (
            getApp().getJobQueue ().getLoadEventAP (jtHO_READ, "HOS::retrieve"));
        obj = LLRetrieve (hash, getApp().getHashNodeLDB ());

        if (!obj)
        {
            mNegativeCache.add (hash);
            return obj;
        }
    }

    /// [17] Leads to duplicated code
    mCache.canonicalize (hash, obj);

    if (mEphemeralDB)
        LLWrite (obj, getApp().getEphemeralLDB ());

    WriteLog (lsTRACE, HashedObject) << "HOS: " << hash << " fetch: in db";
    return obj;
}

bool HashedObjectStore::storeSQLite (HashedObjectType type, uint32 index,
                                     Blob const& data, uint256 const& hash)
{
    // return: false = already in cache, true = added to cache
    if (!getApp().getHashNodeDB ())
    {
        WriteLog (lsTRACE, HashedObject) << "HOS: no db";
        return true;
    }

    if (mCache.touch (hash))
    {
        WriteLog (lsTRACE, HashedObject) << "HOS: " << hash << " store: incache";
        return false;
    }

    assert (hash == Serializer::getSHA512Half (data));

    HashedObject::pointer object = boost::make_shared<HashedObject> (type, index, data, hash);

    /// [18] Duplicated batch writing logic
    if (!mCache.canonicalize (hash, object))
    {
        //      WriteLog (lsTRACE, HashedObject) << "Queuing write for " << hash;
        boost::mutex::scoped_lock sl (mWriteMutex);
        mWriteSet.push_back (object);

        if (!mWritePending)
        {
            mWritePending = true;
            getApp().getJobQueue ().addJob (jtWRITE, "HashedObject::store",
                    BIND_TYPE (&HashedObjectStore::bulkWriteSQLite, this, P_1));
        }
    }

    //  else
    //      WriteLog (lsTRACE, HashedObject) << "HOS: already had " << hash;
    mNegativeCache.del (hash);

    return true;
}

/// [19] Duplicated batch writing code
void HashedObjectStore::bulkWriteSQLite (Job&)
{
    assert (!mLevelDB);
    int setSize = 0;

    while (1)
    {
        std::vector< boost::shared_ptr<HashedObject> > set;
        set.reserve (128);

        {
            boost::mutex::scoped_lock sl (mWriteMutex);
            mWriteSet.swap (set);
            assert (mWriteSet.empty ());
            ++mWriteGeneration;
            mWriteCondition.notify_all ();

            if (set.empty ())
            {
                mWritePending = false;
                mWriteLoad = 0;
                return;
            }

            mWriteLoad = std::max (setSize, static_cast<int> (mWriteSet.size ()));
            setSize = set.size ();
        }
        //      WriteLog (lsTRACE, HashedObject) << "HOS: writing " << set.size();

#ifndef NO_SQLITE3_PREPARE

        if (mEphemeralDB)
            LLWrite (set, getApp().getEphemeralLDB ());
        {
            Database* db = getApp().getHashNodeDB ()->getDB ();


            // VFALCO TODO Get rid of the last parameter "aux", which is set to !theConfig.RUN_STANDALONE
            //
            /// [20] Dependency on theConfig
            /// [21] Global variables make unit tests non-repeatable
            static SqliteStatement pStB (db->getSqliteDB (),
                    "BEGIN TRANSACTION;", !theConfig.RUN_STANDALONE);
            static SqliteStatement pStE (db->getSqliteDB (),
                    "END TRANSACTION;", !theConfig.RUN_STANDALONE);
            static SqliteStatement pSt (db->getSqliteDB (),
                    "INSERT OR IGNORE INTO CommittedObjects "
                    "(Hash,ObjType,LedgerIndex,Object) VALUES (?, ?, ?, ?);",
                        !theConfig.RUN_STANDALONE);

            pStB.step ();
            pStB.reset ();

            BOOST_FOREACH (const boost::shared_ptr<HashedObject>& it, set)
            {
                const char* type;

                switch (it->getType ())
                {
                case hotLEDGER:
                    type = "L";
                    break;

                case hotTRANSACTION:
                    type = "T";
                    break;

                case hotACCOUNT_NODE:
                    type = "A";
                    break;

                case hotTRANSACTION_NODE:
                    type = "N";
                    break;

                default:
                    type = "U";
                }

                pSt.bind (1, it->getHash ().GetHex ());
                pSt.bind (2, type);
                pSt.bind (3, it->getIndex ());
                pSt.bindStatic (4, it->getData ());
                int ret = pSt.step ();

                if (!pSt.isDone (ret))
                {
                    WriteLog (lsFATAL, HashedObject) <<
                        "Error saving hashed object " << ret;
                    assert (false);
                }

                pSt.reset ();
            }

            pStE.step ();
            pStE.reset ();
        }

#else

        static boost::format
        fAdd ("INSERT OR IGNORE INTO CommittedObjects "
              "(Hash,ObjType,LedgerIndex,Object) VALUES ('%s','%c','%u',%s);");

        Database* db = getApp().getHashNodeDB ()->getDB ();
        {
            ScopedLock sl (getApp().getHashNodeDB ()->getDBLock ());

            db->executeSQL ("BEGIN TRANSACTION;");

            BOOST_FOREACH (const boost::shared_ptr<HashedObject>& it, set)
            {
                char type;

                switch (it->getType ())
                {
                case hotLEDGER:
                    type = 'L';
                    break;

                case hotTRANSACTION:
                    type = 'T';
                    break;

                case hotACCOUNT_NODE:
                    type = 'A';
                    break;

                case hotTRANSACTION_NODE:
                    type = 'N';
                    break;

                default:
                    type = 'U';
                }

                db->executeSQL (boost::str (boost::format (fAdd)
                        % it->getHash ().GetHex () % type % it->getIndex ()
                        % sqlEscape (it->getData ())));
            }

            db->executeSQL ("END TRANSACTION;");
        }
#endif

    }
}

HashedObject::pointer HashedObjectStore::retrieveSQLite (uint256 const& hash)
{
    HashedObject::pointer obj = mCache.fetch (hash);

    if (obj)
        return obj;

    if (mNegativeCache.isPresent (hash))
        return obj;

    if (mEphemeralDB)
    {
        obj = LLRetrieve (hash, getApp().getEphemeralLDB ());

        if (obj)
        {
            mCache.canonicalize (hash, obj);
            return obj;
        }
    }

    if (!getApp().getHashNodeDB ())
        return obj;

    Blob data;
    std::string type;
    uint32 index;

#ifndef NO_SQLITE3_PREPARE
    {
        ScopedLock sl (getApp().getHashNodeDB ()->getDBLock ());

        static SqliteStatement pSt (getApp().getHashNodeDB ()->getDB ()->getSqliteDB (),
              "SELECT ObjType,LedgerIndex,Object FROM CommittedObjects WHERE Hash = ?;");

        LoadEvent::autoptr event (getApp().getJobQueue ().getLoadEventAP (jtDISK, "HOS::retrieve"));

        pSt.bind (1, hash.GetHex ());
        int ret = pSt.step ();

        if (pSt.isDone (ret))
        {
            pSt.reset ();
            mNegativeCache.add (hash);
            WriteLog (lsTRACE, HashedObject) << "HOS: " << hash << " fetch: not in db";
            return obj;
        }

        type = pSt.peekString (0);
        index = pSt.getUInt32 (1);
        pSt.getBlob (2).swap (data);
        pSt.reset ();
    }

#else

    std::string sql = "SELECT * FROM CommittedObjects WHERE Hash='";
    sql.append (hash.GetHex ());
    sql.append ("';");


    {
        ScopedLock sl (getApp().getHashNodeDB ()->getDBLock ());
        Database* db = getApp().getHashNodeDB ()->getDB ();

        if (!db->executeSQL (sql) || !db->startIterRows ())
        {
            sl.unlock ();
            mNegativeCache.add (hash);
            return obj;
        }

        db->getStr ("ObjType", type);
        index = db->getBigInt ("LedgerIndex");

        int size = db->getBinary ("Object", NULL, 0);
        data.resize (size);
        db->getBinary ("Object", & (data.front ()), size);
        db->endIterRows ();
    }
#endif

#ifdef PARANOID
    assert (Serializer::getSHA512Half (data) == hash);
#endif

    HashedObjectType htype = hotUNKNOWN;

    switch (type[0])
    {
    case 'L':
        htype = hotLEDGER;
        break;

    case 'T':
        htype = hotTRANSACTION;
        break;

    case 'A':
        htype = hotACCOUNT_NODE;
        break;

    case 'N':
        htype = hotTRANSACTION_NODE;
        break;

    default:
        assert (false);
        WriteLog (lsERROR, HashedObject) << "Invalid hashed object";
        mNegativeCache.add (hash);
        return obj;
    }

    obj = boost::make_shared<HashedObject> (htype, index, data, hash);
    mCache.canonicalize (hash, obj);

    if (mEphemeralDB)
        LLWrite (obj, getApp().getEphemeralLDB ());

    WriteLog (lsTRACE, HashedObject) << "HOS: " << hash << " fetch: in db";
    return obj;
}

int HashedObjectStore::import (const std::string& file)
{
    WriteLog (lsWARNING, HashedObject) << "Hashed object import from \"" << file << "\".";
    UPTR_T<Database> importDB (new SqliteDatabase (file.c_str ()));
    importDB->connect ();

    leveldb::DB* db = getApp().getHashNodeLDB ();
    leveldb::WriteOptions wo;

    int count = 0;

    SQL_FOREACH (importDB, "SELECT * FROM CommittedObjects;")
    {
        uint256 hash;
        std::string hashStr;
        importDB->getStr ("Hash", hashStr);
        hash.SetHexExact (hashStr);

        if (hash.isZero ())
        {
            WriteLog (lsWARNING, HashedObject) << "zero hash found in import table";
        }
        else
        {
            Blob rawData;
            int size = importDB->getBinary ("Object", NULL, 0);
            rawData.resize (9 + size);
            unsigned char* bufPtr = &rawData.front ();

            importDB->getBinary ("Object", bufPtr + 9, size);

            uint32 index = importDB->getBigInt ("LedgerIndex");
            *reinterpret_cast<uint32*> (bufPtr + 0) = ntohl (index);
            *reinterpret_cast<uint32*> (bufPtr + 4) = ntohl (index);

            std::string type;
            importDB->getStr ("ObjType", type);
            HashedObjectType htype = hotUNKNOWN;

            switch (type[0])
            {
            case 'L':
                htype = hotLEDGER;
                break;
            case 'T':
                htype = hotTRANSACTION;
                break;

            case 'A':
                htype = hotACCOUNT_NODE;
                break;

            case 'N':
                htype = hotTRANSACTION_NODE;
                break;

            default:
                assert (false);
                WriteLog (lsERROR, HashedObject) << "Invalid hashed object";
            }

            * (bufPtr + 8) = static_cast<unsigned char> (htype);

            leveldb::Status st = db->Put (wo,
                leveldb::Slice (reinterpret_cast<const char*> (hash.begin ()), hash.size ()),
                leveldb::Slice (reinterpret_cast<const char*> (bufPtr), rawData.size ()));

            if (!st.ok ())
            {
                WriteLog (lsFATAL, HashedObject) << "Failed to store hash node";
                assert (false);
            }

            ++count;
        }

        if ((count % 10000) == 0)
        {
            WriteLog (lsINFO, HashedObject) << "Import in progress: " << count;
        }
    }

    WriteLog (lsWARNING, HashedObject) << "Imported " << count << " nodes";
    return count;
}

/// [22] Where are the unit tests?
```

Addendum B

ripple_NodeStore.cpp

```
//------------------------------------------------------------------------------
/*
    Copyright (c) 2011-2013, OpenCoin, Inc.
*/
//==============================================================================

#ifndef RIPPLE_NODESTORE_H_INCLUDED
#define RIPPLE_NODESTORE_H_INCLUDED

// Javadoc comments are added to all public classes, member functions,
// type definitions, data types, and global variables (which we should
// minimize the use of.
//
// A Javadoc comment is introduced with an extra asterisk following the
// beginning of a normal C++ style comment, or by using a triple forward slash.
//
// Structure of a Javadoc comment:

/** Brief one line description.

    A more detailed description, which may be broken up into multiple
    paragraphs. Doxygen commands are prefixed with the at-sign '@'. For
    example, here's a formatted code snippet:

    @code

    int main (int argc, char** argv)
    {
        return 0;
    }

    @endcode

    You can also add a note, or document an invariant:

    @note This appears as its own note.

    @invariant This must not be called while holding the lock.

    When documenting functions, you can use these Doxygen commands
    to annotate the parameters, return value, template types.

    @param  argc    The number of arguments to the program.
    @param  argv    An array of strings argc in size, one for each argument.

    @return The return value of the program, passed to to the enclosing process.
*/

/** Functions can be documented with just the brief description, like this */

/// Here's the alternate form of a brief description.

//------------------------------------------------------------------------------

/// [1] Javadoc comment explaining the class

/** Persistency layer for NodeObject

    A Node is a ledger object which is uniquely identified by a key, which is
    the 256-bit hash of the body of the node. The payload is a variable length
    block of serialized data.

    All ledger data is stored as node objects and as such, needs to be persisted
    between launches. Furthermore, since the set of node objects will in
    general be larger than the amount of available memory, purged node objects
    which are later accessed must be retrieved from the node store.

    @see NodeObject
*/
class NodeStore
{
public:
    /// [2] Magic numbers moved to an enumeration instead of repeated
    enum
    {
        // This is only used to pre-allocate the array for
        // batch objects and does not affect the amount written.
        //
        batchWritePreallocationSize = 128
    };

    /// [3] Commonly used type is given a descriptive name
    typedef std::vector <NodeObject::Ptr> Batch;

    //--------------------------------------------------------------------------

    /// [4] Deserialization format made explicit

    /** Parsed key/value blob into NodeObject components.

        This will extract the information required to construct a NodeObject. It
        also does consistency checking and returns the result, so it is possible
        to determine if the data is corrupted without throwing an exception. Not
        all forms of corruption are detected so further analysis will be needed
        to eliminate false negatives.

        @note This defines the database format of a NodeObject!
    */
    class DecodedBlob
    {
    public:
        /** Construct the decoded blob from raw data. */
        DecodedBlob (void const* key, void const* value, int valueBytes);

        /// [5] New feature, detecting corrupted data

        /** Determine if the decoding was successful. */
        bool wasOk () const noexcept { return m_success; }

        /// [6] Centralized place for creating NodeObject from serialized data

        /** Create a NodeObject from this data. */
        NodeObject::Ptr createObject ();

        /// [7] Data members hidden, all interface is through just 3 members
    private:
        bool m_success;

        void const* m_key;
        LedgerIndex m_ledgerIndex;
        NodeObjectType m_objectType;
        unsigned char const* m_objectData;
        int m_dataBytes;
    };

    //--------------------------------------------------------------------------

    /// [8] Serialization format made explicit
```

```
/** Utility for producing flattened node objects.

    These get recycled to prevent many small allocations.

    @note This defines the database format of a NodeObject!
*/
struct EncodedBlob
{
    /// [9] Added a Beast class to assist with optimal usage of memory
    typedef RecycledObjectPool <EncodedBlob> Pool;

    void prepare (NodeObject::Ptr const& object);

    void const* getKey () const noexcept { return m_key; }

    size_t getSize () const noexcept { return m_size; }

    void const* getData () const noexcept { return m_data.getData (); }

private:
    void const* m_key;
    MemoryBlock m_data;
    size_t m_size;
};

//------------------------------------------------------------------------------

/// [10] Abstract interface to eliminate an unwanted dependency

/** Provides optional asynchronous scheduling for backends.

    For improved performance, a backend has the option of performing writes
    in batches. These writes can be scheduled using the provided scheduler
    object.
*/
class Scheduler
{
public:
    /** Derived classes perform scheduled tasks. */
    struct Task
    {
        virtual ~Task () { }

        /** Performs the task.

            The call may take place on a foreign thread.
        */
        virtual void performScheduledTask () = 0;
    };

    /** Schedules a task.

        Depending on the implementation, this could happen
        immediately or get deferred.
    */
    virtual void scheduleTask (Task* task) = 0;
};

//------------------------------------------------------------------------------

/// [11] Batch writing logic refactored into its own class.

/** Helps with batch writing.

    The batch writes are performed with a scheduled task. Use of the
    class it not required. A backend can implement its own write batching,
    or skip write batching if doing so yields a performance benefit.

    @see Scheduler
*/
// VFALCO NOTE I'm not entirely happy having placed this here,
//             because whoever needs to use NodeStore certainly doesn't
//             need to see the implementation details of BatchWriter.
//
class BatchWriter : private Scheduler::Task
{
public:
    /** This callback does the actual writing. */
    struct Callback
    {
        virtual void writeBatch (Batch const& batch) = 0;
    };

    /** Create a batch writer. */
    BatchWriter (Callback& callback, Scheduler& scheduler);

    /** Destroy a batch writer.

        Anything pending in the batch is written out before this returns.
    */
    ~BatchWriter ();

    /** Store the object.

        This will add to the batch and initiate a scheduled task to
        write the batch out.
    */
    void store (NodeObject::Ptr const& object);

    /** Get an estimate of the amount of writing I/O pending. */
    int getWriteLoad ();

private:
    void performScheduledTask ();
    void writeBatch ();
    void waitForWriting ();

private:
    typedef boost::recursive_mutex LockType;
    typedef boost::condition_variable_any CondvarType;

    Callback& m_callback;
    Scheduler& m_scheduler;
    LockType mWriteMutex;
    CondvarType mWriteCondition;
    int mWriteGeneration;
    int mWriteLoad;
    bool mWritePending;
    Batch mWriteSet;
};

//------------------------------------------------------------------------------

/// [12] Backend interface is completely abstract

/** A backend used for the store.

    The NodeStore uses a swappable backend so that other database systems
    can be tried. Different databases may offer various features such
    as improved performance, fault tolerant or distributed storage, or
    all in-memory operation.

    A given instance of a backend is fixed to a particular key size.
*/
```

```
class Backend
{
public:
    /** Return codes from operations. */

    /// [13] Well defined return codes to make results explicit

    enum Status
    {
        ok,
        notFound,
        dataCorrupt,
        unknown
    };

    /// [14] Behavior of member functions is defined in the comments

    /** Destroy the backend.

        All open files are closed and flushed. If there are batched writes
        or other tasks scheduled, they will be completed before this call
        returns.
    */
    virtual ~Backend () { }

    /** Get the human-readable name of this backend.

        This is used for diagnostic output.
    */
    virtual std::string getName() = 0;

    /** Fetch a single object.

        If the object is not found or an error is encountered, the
        result will indicate the condition.

        @note This will be called concurrently.

        @param key A pointer to the key data.
        @param pObject [out] The created object if successful.

        @return The result of the operation.
    */

    /// [15] We choose void* over uint256

    virtual Status fetch (void const* key, NodeObject::Ptr* pObject) = 0;

    /** Store a single object.

        Depending on the implementation this may happen immediately
        or deferred using a scheduled task.

        @note This will be called concurrently.

        @param object The object to store.
    */

    /// [16] Standardized naming conventions. fetch, store.

    virtual void store (NodeObject::Ptr const& object) = 0;

    /** Store a group of objects.

        @note This function will not be called concurrently with
              itself or @ref store.
    */
    virtual void storeBatch (Batch const& batch) = 0;

    /** Callback for iterating through objects.

        @see visitAll
    */
    struct VisitCallback
    {
        virtual void visitObject (NodeObject::Ptr const& object) = 0;
    };

    /** Visit every object in the database

        This is usually called during import.

        @note This routine will not be called concurrently with itself
              or other methods.

        @see import
    */
    virtual void visitAll (VisitCallback& callback) = 0;

    /** Estimate the number of write operations pending. */
    virtual int getWriteLoad () = 0;
};

//------------------------------------------------------------------------------

/** Factory to produce backends.
*/
class BackendFactory
{
public:
    virtual ~BackendFactory () { }

    /** Retrieve the name of this factory. */
    virtual String getName () const = 0;

    /** Create an instance of this factory's backend.

        @param keyBytes The fixed number of bytes per key.
        @param keyValues A set of key/value configuration pairs.
        @param scheduler The scheduler to use for running tasks.

        @return A pointer to the Backend object.
    */
    virtual Backend* createInstance (size_t keyBytes,
                                     StringPairArray const& keyValues,
                                     Scheduler& scheduler) = 0;
};

//------------------------------------------------------------------------------

/** Construct a node store.

    Parameter strings have the format:

    <key>=<value>['|'<key>=<value>]

    The key "type" must exist, it defines the choice of backend.
    For example
        `type=LevelDB|path=/mnt/ephemeral`

    @param backendParameters The parameter string for the persistent backend.
    @param fastBackendParameters The parameter string for the ephemeral backend.
    @param cacheSize ?
    @param cacheAge ?
```

```
        @param scheduler The scheduler to use for performing asynchronous tasks.

        @return A pointer to the created object.
    */
    /// [17] Scheduler passed in
    static NodeStore* New (String backendParameters,
                           String fastBackendParameters,
                           Scheduler& scheduler);

    /** Destroy the node store.

        All pending operations are completed, pending writes flushed,
        and files closed before this returns.
    */
    virtual ~NodeStore () { }

    /** Add the specified backend factory to the list of available factories.

        The names of available factories are compared against the "type"
        value in the parameter list on construction.

        @param factory The factory to add.
    */
    static void addBackendFactory (BackendFactory& factory);

    /// [18] Comments make the return value clear.

    /** Fetch an object.

        If the object is known to be not in the database, isn't found in the
        database during the fetch, or failed to load correctly during the fetch,
        `nullptr` is returned.

        @note This can be called concurrently.

        @param hash The key of the object to retrieve.

        @return The object, or nullptr if it couldn't be retrieved.
    */
    virtual NodeObject::pointer fetch (uint256 const& hash) = 0;

    /// [19] An optimization allows the caller's buffer to be taken over

    /** Store the object.

        The caller's Blob parameter is overwritten.

        @param type The type of object.
        @param ledgerIndex The ledger in which the object appears.
        @param data The payload of the object. The caller's
                    variable is overwritten.
        @param hash The 256-bit hash of the payload data.

        @return `true` if the object was stored?
    */
    virtual void store (NodeObjectType type,
                        uint32 ledgerIndex,
                        Blob& data,
                        uint256 const& hash) = 0;

    /** Import objects from another database.

        The other NodeStore database is constructed using the specified
        backend parameters.
    */
    virtual void import (String sourceBackendParameters) = 0;

    /** Retrieve the estimated number of pending write operations.

        This is used for diagnostics.
    */
    virtual int getWriteLoad () = 0;

    // VFALCO TODO Document this.
    virtual float getCacheHitRate () = 0;

    // VFALCO TODO Document this.
    //        TODO Document the parameter meanings.
    virtual void tune (int size, int age) = 0;

    // VFALCO TODO Document this.
    virtual void sweep () = 0;

};

#endif

//==============================================================================

NodeStore::DecodedBlob::DecodedBlob (void const* key, void const* value, int valueBytes)
{
    /// [20] Format of the data is explained in a comment

    /*  Data format:

        Bytes

        0...3       LedgerIndex     32-bit big endian integer
        4...7       Unused?         An unused copy of the LedgerIndex
        8           char            One of NodeObjectType
        9...end                     The body of the object data
    */

    m_success = false;
    m_key = key;
    // VFALCO NOTE Ledger indexes should have started at 1
    m_ledgerIndex = LedgerIndex (-1);
    m_objectType = hotUNKNOWN;
    m_objectData = nullptr;
    m_dataBytes = bmax (0, valueBytes - 9);

    if (valueBytes > 4)
    {
        LedgerIndex const* index = static_cast <LedgerIndex const*> (value);
        m_ledgerIndex = ByteOrder::swapIfLittleEndian (*index);
    }

    /// [21] There are 4 bytes unaccounted for

    // VFALCO NOTE What about bytes 4 through 7 inclusive?

    if (valueBytes > 8)
    {
        unsigned char const* byte = static_cast <unsigned char const*> (value);
        m_objectType = static_cast <NodeObjectType> (byte [8]);
    }

    if (valueBytes > 9)
    {
        m_objectData = static_cast <unsigned char const*> (value) + 9;

        switch (m_objectType)
        {
        case hotUNKNOWN:
```

```
        default:
            break;

        case hotLEDGER:
        case hotTRANSACTION:
        case hotACCOUNT_NODE:
        case hotTRANSACTION_NODE:
            m_success = true;
            break;
        }
    }
}

NodeObject::Ptr NodeStore::DecodedBlob::createObject ()
{
    bassert (m_success);

    NodeObject::Ptr object;

    if (m_success)
    {
        Blob data (m_dataBytes);

        memcpy (data.data (), m_objectData, m_dataBytes);

        /// [22] No more boost::make_shared<> everywhere

        object = NodeObject::createObject (
            m_objectType, m_ledgerIndex, data, uint256 (m_key));
    }

    return object;
}

//------------------------------------------------------------------------------

void NodeStore::EncodedBlob::prepare (NodeObject::Ptr const& object)
{
    m_key = object->getHash ().begin ();

    // This is how many bytes we need in the flat data
    m_size = object->getData ().size () + 9;

    m_data.ensureSize (m_size);

    /// [23] Architecture assumptions are checked at compile time

    // These sizes must be the same!
    static_bassert (sizeof (uint32) == sizeof (object->getIndex ()));

    {
        uint32* buf = static_cast <uint32*> (m_data.getData ());

        buf [0] = ByteOrder::swapIfLittleEndian (object->getIndex ());
        buf [1] = ByteOrder::swapIfLittleEndian (object->getIndex ());
    }

    {
        unsigned char* buf = static_cast <unsigned char*> (m_data.getData ());

        buf [8] = static_cast <unsigned char> (object->getType ());

        memcpy (&buf [9], object->getData ().data (), object->getData ().size ());
    }
}

//==============================================================================

NodeStore::BatchWriter::BatchWriter (Callback& callback, Scheduler& scheduler)
    : m_callback (callback)
    , m_scheduler (scheduler)
    , mWriteGeneration (0)
    , mWriteLoad (0)
    , mWritePending (false)
{
    mWriteSet.reserve (batchWritePreallocationSize);
}

NodeStore::BatchWriter::~BatchWriter ()
{
    waitForWriting ();
}

void NodeStore::BatchWriter::store (NodeObject::ref object)
{
    LockType::scoped_lock sl (mWriteMutex);

    mWriteSet.push_back (object);

    if (! mWritePending)
    {
        mWritePending = true;

        m_scheduler.scheduleTask (this);
    }
}

int NodeStore::BatchWriter::getWriteLoad ()
{
    LockType::scoped_lock sl (mWriteMutex);

    return std::max (mWriteLoad, static_cast<int> (mWriteSet.size ()));
}

void NodeStore::BatchWriter::performScheduledTask ()
{
    writeBatch ();
}

void NodeStore::BatchWriter::writeBatch ()
{
    int setSize = 0;

    for (;;)
    {
        std::vector< boost::shared_ptr<NodeObject> > set;

        set.reserve (batchWritePreallocationSize);

        {
            LockType::scoped_lock sl (mWriteMutex);

            mWriteSet.swap (set);
            assert (mWriteSet.empty ());
            ++mWriteGeneration;
            mWriteCondition.notify_all ();

            if (set.empty ())
            {
                mWritePending = false;
                mWriteLoad = 0;

                // VFALCO NOTE Fix this function to not return from the middle
                return;
            }

            // VFALCO NOTE On the first trip through, mWriteLoad will be 0.
            //             This is probably not intended. Perhaps the order
            //             of calls isn't quite right
            //
            mWriteLoad = std::max (setSize, static_cast<int> (mWriteSet.size ()));

            setSize = set.size ();
        }

        m_callback.writeBatch (set);
    }
}

void NodeStore::BatchWriter::waitForWriting ()
{
    LockType::scoped_lock sl (mWriteMutex);
    int gen = mWriteGeneration;

    while (mWritePending && (mWriteGeneration == gen))
        mWriteCondition.wait (sl);
}

//==============================================================================

class NodeStoreImp
    : public NodeStore
    , LeakChecked <NodeStoreImp>
{
public:
    NodeStoreImp (String backendParameters,
                  String fastBackendParameters,
                  Scheduler& scheduler)
        : m_scheduler (scheduler)
        , m_backend (createBackend (backendParameters, scheduler))
        , m_fastBackend (fastBackendParameters.isNotEmpty ()
            ? createBackend (fastBackendParameters, scheduler) : nullptr)
        , m_cache ("NodeStore", 16384, 300)
        , m_negativeCache ("NoteStoreNegativeCache", 0, 120)
    {
    }

    ~NodeStoreImp ()
    {
    }

    //------------------------------------------------------------------------------

    /// [24] Complex business logic receives extra attention from comments

    NodeObject::Ptr fetch (uint256 const& hash)
    {
        // See if the object already exists in the cache
        //
        NodeObject::Ptr obj = m_cache.fetch (hash);

        if (obj == nullptr)
        {
            // It's not in the cache, see if we can skip checking the db.
            //
            if (! m_negativeCache.isPresent (hash))
            {
                // There's still a chance it could be in one of the databases.

                bool foundInFastBackend = false;

                // Check the fast backend database if we have one
                //
                if (m_fastBackend != nullptr)
                {
                    obj = fetchInternal (m_fastBackend, hash);

                    // If we found the object, avoid storing it again later.
                    if (obj != nullptr)
                        foundInFastBackend = true;
                }

                // Are we still without an object?
                //
                if (obj == nullptr)
                {
                    // Yes so at last we will try the main database.
                    //
                    {
                        // Monitor this operation's load since it is expensive.
                        //
                        // VFALCO TODO Why is this an autoptr? Why can't it just be a plain old object?
                        //
                        // VFALCO NOTE Commented this out because it breaks the unit test!
                        //
                        //LoadEvent::autoptr event (getApp().getJobQueue ()
                        //   .getLoadEventAP (jtHO_READ, "HOS::retrieve"));

                        obj = fetchInternal (m_backend, hash);
                    }

                    // If it's not in the main database, remember that so we
                    // can skip the lookup for the same object again later.
                    //
                    if (obj == nullptr)
                        m_negativeCache.add (hash);
                }

                // Did we finally get something?
                //
                if (obj != nullptr)
                {
                    // Yes it so canonicalize. This solves the problem where
                    // more than one thread has its own copy of the same object.
                    //
                    m_cache.canonicalize (hash, obj);

                    if (! foundInFastBackend)
                    {
                        // If we have a fast back end, store it there for later.
                        //
                        if (m_fastBackend != nullptr)
                            m_fastBackend->store (obj);

                        // Since this was a 'hard' fetch, we will log it.
                        //
                        WriteLog (lsTRACE, NodeObject) << "HOS: " << hash << " fetch: in db";
                    }
                }
            }
            else
            {
                // hash is known not to be in the database
            }
        }
        else
        {
            // found it!
```

```cpp
        }

        return obj;
    }

    /// [25] Function split up as needed to avoid redundant code

    NodeObject::Ptr fetchInternal (Backend* backend, uint256 const& hash)
    {
        NodeObject::Ptr object;

        Backend::Status const status = backend->fetch (hash.begin (), &object);

        switch (status)
        {
        case Backend::ok:
        case Backend::notFound:
            break;

        case Backend::dataCorrupt:
            // VFALCO TODO Deal with encountering corrupt data!
            //
            WriteLog (lsFATAL, NodeObject) << "Corrupt NodeObject #" << hash;
            break;

        default:
            WriteLog (lsWARNING, NodeObject) << "Unknown status=" << status;
            break;
        }

        return object;
    }

    //------------------------------------------------------------------------------

    void store (NodeObjectType type,
                uint32 index,
                Blob& data,
                uint256 const& hash)
    {
        bool const keyFoundAndObjectCached = m_cache.refreshIfPresent (hash);

        // VFALCO NOTE What happens if the key is found, but the object
        //             fell out of the cache? We will end up passing it
        //             to the backend anyway.
        //
        if (! keyFoundAndObjectCached)
        {
        #if RIPPLE_VERIFY_NODEOBJECT_KEYS
            assert (hash == Serializer::getSHA512Half (data));
        #endif

            NodeObject::Ptr object = NodeObject::createObject (
                type, index, data, hash);

            if (!m_cache.canonicalize (hash, object))
            {
                m_backend->store (object);

                if (m_fastBackend)
                    m_fastBackend->store (object);
            }

            m_negativeCache.del (hash);
        }
    }

    //------------------------------------------------------------------------------

    float getCacheHitRate ()
    {
        return m_cache.getHitRate ();
    }

    void tune (int size, int age)
    {
        m_cache.setTargetSize (size);
        m_cache.setTargetAge (age);
    }

    void sweep ()
    {
        m_cache.sweep ();
        m_negativeCache.sweep ();
    }

    int getWriteLoad ()
    {
        return m_backend->getWriteLoad ();
    }

    //------------------------------------------------------------------------------

    void import (String sourceBackendParameters)
    {
        class ImportVisitCallback : public Backend::VisitCallback
        {
        public:
            explicit ImportVisitCallback (Backend& backend)
                : m_backend (backend)
            {
                m_objects.reserve (batchWritePreallocationSize);
            }

            ~ImportVisitCallback ()
            {
                if (! m_objects.empty ())
                    m_backend.storeBatch (m_objects);
            }

            void visitObject (NodeObject::Ptr const& object)
            {
                if (m_objects.size () >= batchWritePreallocationSize)
                {
                    m_backend.storeBatch (m_objects);

                    m_objects.clear ();
                    m_objects.reserve (batchWritePreallocationSize);
                }

                m_objects.push_back (object);
            }

        private:
            Backend& m_backend;
            Batch m_objects;
        };

        //------------------------------------------------------------------------------

        ScopedPointer <Backend> srcBackend (createBackend (sourceBackendParameters, m_scheduler));

        WriteLog (lsWARNING, NodeObject) <<
            "Node import from '" << srcBackend->getName() << "' to '"
```

```
                                 << m_backend->getName() << "'.";

        ImportVisitCallback callback (*m_backend);

        srcBackend->visitAll (callback);
    }

    //------------------------------------------------------------------------------

    static NodeStore::Backend* createBackend (String const& parameters, Scheduler& scheduler)
    {
        Backend* backend = nullptr;

        StringPairArray keyValues = parseKeyValueParameters (parameters, '|');

        String const& type = keyValues ["type"];

        if (type.isNotEmpty ())
        {
            BackendFactory* factory = nullptr;

            for (int i = 0; i < s_factories.size (); ++i)
            {
                if (s_factories [i]->getName ().compareIgnoreCase (type) == 0)
                {
                    factory = s_factories [i];
                    break;
                }
            }

            if (factory != nullptr)
            {
                backend = factory->createInstance (NodeObject::keyBytes, keyValues, scheduler);
            }
            else
            {
                Throw (std::runtime_error ("unknown backend type"));
            }
        }
        else
        {
            Throw (std::runtime_error ("missing backend type"));
        }

        return backend;
    }

    static void addBackendFactory (BackendFactory& factory)
    {
        s_factories.add (&factory);
    }

    //------------------------------------------------------------------------------

private:
    static Array <NodeStore::BackendFactory*> s_factories;

    Scheduler& m_scheduler;

    /// [26] Beast containers reduce programmer errors

    // Persistent key/value storage.
    ScopedPointer <Backend> m_backend;

    // Larger key/value storage, but not necessarily persistent.
    ScopedPointer <Backend> m_fastBackend;

    // VFALCO NOTE What are these things for? We need comments.
    TaggedCache <uint256, NodeObject, UptimeTimerAdapter> m_cache;
    KeyCache <uint256, UptimeTimerAdapter> m_negativeCache;
};

Array <NodeStore::BackendFactory*> NodeStoreImp::s_factories;

//------------------------------------------------------------------------------

void NodeStore::addBackendFactory (BackendFactory& factory)
{
    NodeStoreImp::addBackendFactory (factory);
}

NodeStore* NodeStore::New (String backendParameters,
                           String fastBackendParameters,
                           Scheduler& scheduler)
{
    return new NodeStoreImp (backendParameters,
                             fastBackendParameters,
                             scheduler);
}

//==============================================================================

/// [27] We factor common unit test code into its own class

// Some common code for the unit tests
//
class NodeStoreUnitTest : public UnitTest
{
public:
    // Tunable parameters
    //
    enum
    {
        maxPayloadBytes = 1000,
        numObjectsToTest = 1000
    };

    // Shorthand type names
    //
    typedef NodeStore::Backend Backend;
    typedef NodeStore::Batch Batch;

    /// [28] The unit test provides its own version of Scheduler

    // Immediately performs the task
    struct TestScheduler : NodeStore::Scheduler
    {
        void scheduleTask (Task* task)
        {
            task->performScheduledTask ();
        }
    };

    // Creates predictable objects
    class PredictableObjectFactory
    {
    public:
        explicit PredictableObjectFactory (int64 seedValue)
            : m_seedValue (seedValue)
        {
        }

        NodeObject::Ptr createObject (int index)
        {
            Random r (m_seedValue + index);

            NodeObjectType type;
            switch (r.nextInt (4))
            {
            case 0: type = hotLEDGER; break;
            case 1: type = hotTRANSACTION; break;
            case 2: type = hotACCOUNT_NODE; break;
            case 3: type = hotTRANSACTION_NODE; break;
            default:
                type = hotUNKNOWN;
                break;
            };

            LedgerIndex ledgerIndex = 1 + r.nextInt (1024 * 1024);

            uint256 hash;
            r.nextBlob (hash.begin (), hash.size ());

            int const payloadBytes = 1 + r.nextInt (maxPayloadBytes);

            Blob data (payloadBytes);

            r.nextBlob (data.data (), payloadBytes);

            return NodeObject::createObject (type, ledgerIndex, data, hash);
        }

    private:
        int64 const m_seedValue;
    };

public:
    NodeStoreUnitTest (String name, UnitTest::When when = UnitTest::runAlways)
        : UnitTest (name, "ripple", when)
    {
    }

    // Create a predictable batch of objects
    static void createPredictableBatch (Batch& batch, int startingIndex, int numObjects, int64 seedValue)
    {
        batch.reserve (numObjects);

        PredictableObjectFactory factory (seedValue);

        for (int i = 0; i < numObjects; ++i)
            batch.push_back (factory.createObject (startingIndex + i));
    }

    // Compare two batches for equality
    static bool areBatchesEqual (Batch const& lhs, Batch const& rhs)
    {
        bool result = true;

        if (lhs.size () == rhs.size ())
        {
            for (int i = 0; i < lhs.size (); ++i)
            {
                if (! lhs [i]->isCloneOf (rhs [i]))
                {
                    result = false;
                    break;
                }
            }
        }
        else
        {
            result = false;
        }

        return result;
    }

    // Store a batch in a backend
    void storeBatch (Backend& backend, Batch const& batch)
    {
        for (int i = 0; i < batch.size (); ++i)
        {
            backend.store (batch [i]);
        }
    }

    // Get a copy of a batch in a backend
    void fetchCopyOfBatch (Backend& backend, Batch* pCopy, Batch const& batch)
    {
        pCopy->clear ();
        pCopy->reserve (batch.size ());

        for (int i = 0; i < batch.size (); ++i)
        {
            NodeObject::Ptr object;

            Backend::Status const status = backend.fetch (
                batch [i]->getHash ().cbegin (), &object);

            expect (status == Backend::ok, "Should be ok");

            if (status == Backend::ok)
            {
                expect (object != nullptr, "Should not be null");

                pCopy->push_back (object);
            }
        }
    }

    // Store all objects in a batch
    static void storeBatch (NodeStore& db, NodeStore::Batch const& batch)
    {
        for (int i = 0; i < batch.size (); ++i)
        {
            NodeObject::Ptr const object (batch [i]);

            Blob data (object->getData ());

            db.store (object->getType (),
                      object->getIndex (),
                      data,
                      object->getHash ());
        }
    }

    // Fetch all the hashes in one batch, into another batch.
    static void fetchCopyOfBatch (NodeStore& db,
                                  NodeStore::Batch* pCopy,
                                  NodeStore::Batch const& batch)
    {
        pCopy->clear ();
        pCopy->reserve (batch.size ());

        for (int i = 0; i < batch.size (); ++i)
        {
            NodeObject::Ptr object = db.fetch (batch [i]->getHash ());
            if (object != nullptr)
                pCopy->push_back (object);
        }
    }
};

//------------------------------------------------------------------------------

/// [29] A suite of basic tests needed for the rest of the tests

// Tests predictable batches, and NodeObject blob encoding
//
class NodeStoreBasicsTests : public NodeStoreUnitTest
{
public:
    typedef NodeStore::EncodedBlob EncodedBlob;
    typedef NodeStore::DecodedBlob DecodedBlob;

    NodeStoreBasicsTests () : NodeStoreUnitTest ("NodeStoreBasics")
    {
    }

    // Make sure predictable object generation works!
    void testBatches (int64 const seedValue)
    {
        beginTest ("batch");

        Batch batch1;
        createPredictableBatch (batch1, 0, numObjectsToTest, seedValue);

        Batch batch2;
        createPredictableBatch (batch2, 0, numObjectsToTest, seedValue);

        expect (areBatchesEqual (batch1, batch2), "Should be equal");

        Batch batch3;
        createPredictableBatch (batch3, 1, numObjectsToTest, seedValue);

        expect (! areBatchesEqual (batch1, batch3), "Should be equal");
    }

    // Checks encoding/decoding blobs
    void testBlobs (int64 const seedValue)
    {
        beginTest ("encoding");

        Batch batch;
        createPredictableBatch (batch, 0, numObjectsToTest, seedValue);

        EncodedBlob encoded;
        for (int i = 0; i < batch.size (); ++i)
        {
            encoded.prepare (batch [i]);

            DecodedBlob decoded (encoded.getKey (), encoded.getData (), encoded.getSize ());

            expect (decoded.wasOk (), "Should be ok");

            if (decoded.wasOk ())
            {
                NodeObject::Ptr const object (decoded.createObject ());

                expect (batch [i]->isCloneOf (object), "Should be clones");
            }
        }
    }

    void runTest ()
    {
        int64 const seedValue = 50;

        testBatches (seedValue);

        testBlobs (seedValue);
    }
};

static NodeStoreBasicsTests nodeStoreBasicsTests;

//------------------------------------------------------------------------------

/// [30] Backend test skips NodeStore and tests the backend directly

// Tests the NodeStore::Backend interface
//
class NodeStoreBackendTests : public NodeStoreUnitTest
{
public:
    NodeStoreBackendTests () : NodeStoreUnitTest ("NodeStoreBackend")
    {
    }

    //--------------------------------------------------------------------------

    void testBackend (String type, int64 const seedValue)
    {
        beginTest (String ("NodeStore::Backend type=") + type);

        String params;
        params << "type=" << type
               << "|path=" << File::createTempFile ("unittest").getFullPathName ();

        // Create a batch
        NodeStore::Batch batch;
        createPredictableBatch (batch, 0, numObjectsToTest, seedValue);
        //createPredictableBatch (batch, 0, 10, seedValue);

        {
            // Open the backend
            ScopedPointer <Backend> backend (
                NodeStoreImp::createBackend (params, m_scheduler));

            // Write the batch
            storeBatch (*backend, batch);

            {
                // Read it back in
                NodeStore::Batch copy;
                fetchCopyOfBatch (*backend, &copy, batch);
                expect (areBatchesEqual (batch, copy), "Should be equal");
            }

            {
                // Reorder and read the copy again
                NodeStore::Batch copy;
                UnitTestUtilities::repeatableShuffle (batch.size (), batch, seedValue);
                fetchCopyOfBatch (*backend, &copy, batch);
                expect (areBatchesEqual (batch, copy), "Should be equal");
            }
        }

        {
            // Re-open the backend
            ScopedPointer <Backend> backend (
                NodeStoreImp::createBackend (params, m_scheduler));

            // Read it back in
            NodeStore::Batch copy;
            fetchCopyOfBatch (*backend, &copy, batch);
            // Canonicalize the source and destination batches
            std::sort (batch.begin (), batch.end (), NodeObject::LessThan ());
            std::sort (copy.begin (), copy.end (), NodeObject::LessThan ());
            expect (areBatchesEqual (batch, copy), "Should be equal");
        }
    }

    void runTest ()
    {
        int const seedValue = 50;

        testBackend ("keyvadb", seedValue);

        testBackend ("leveldb", seedValue);

        testBackend ("sqlite", seedValue);

        #if RIPPLE_HYPERLEVELDB_AVAILABLE
        testBackend ("hyperleveldb", seedValue);
        #endif

        #if RIPPLE_MDB_AVAILABLE
        testBackend ("mdb", seedValue);
        #endif
    }

private:
    TestScheduler m_scheduler;
};

static NodeStoreBackendTests nodeStoreBackendTests;

//------------------------------------------------------------------------------

/// [31] Timing tests to pick the "winner" and prevent regressions

class NodeStoreTimingTests : public NodeStoreUnitTest
{
public:
    enum
    {
        numObjectsToTest     = 20000
    };

    NodeStoreTimingTests ()
        : NodeStoreUnitTest ("NodeStoreTiming", UnitTest::runManual)
    {
    }

    class Stopwatch
    {
    public:
        Stopwatch ()
        {
        }

        void start ()
        {
            m_startTime = Time::getHighResolutionTicks ();
        }

        double getElapsed ()
        {
            int64 const now = Time::getHighResolutionTicks();

            return Time::highResolutionTicksToSeconds (now - m_startTime);
        }

    private:
        int64 m_startTime;
    };

    void testBackend (String type, int64 const seedValue)
    {
        String s;
        s << "Testing backend '" << type << "' performance";
        beginTest (s);

        String params;
        params << "type=" << type
               << "|path=" << File::createTempFile ("unittest").getFullPathName ();

        // Create batches
        NodeStore::Batch batch1;
        createPredictableBatch (batch1, 0, numObjectsToTest, seedValue);
        NodeStore::Batch batch2;
        createPredictableBatch (batch2, 0, numObjectsToTest, seedValue);

        // Open the backend
        ScopedPointer <Backend> backend (
            NodeStoreImp::createBackend (params, m_scheduler));

        Stopwatch t;

        // Individual write batch test
        t.start ();
        storeBatch (*backend, batch1);
        s = "";
        s << "  Single write: " << String (t.getElapsed (), 2) << " seconds";
        logMessage (s);

        // Bulk write batch test
        t.start ();
        backend->storeBatch (batch2);
        s = "";
        s << "  Batch write:  " << String (t.getElapsed (), 2) << " seconds";
        logMessage (s);

        // Read test
        Batch copy;
        t.start ();
        fetchCopyOfBatch (*backend, &copy, batch1);
        fetchCopyOfBatch (*backend, &copy, batch2);
        s = "";
        s << "  Batch read:   " << String (t.getElapsed (), 2) << " seconds";
        logMessage (s);
    }

    void runTest ()
    {
        int const seedValue = 50;

        testBackend ("keyvadb", seedValue);

        testBackend ("leveldb", seedValue);

        #if RIPPLE_HYPERLEVELDB_AVAILABLE
        testBackend ("hyperleveldb", seedValue);
        #endif
        #if RIPPLE_MDB_AVAILABLE
        testBackend ("mdb", seedValue);
        #endif

        testBackend ("sqlite", seedValue);
    }

private:
    TestScheduler m_scheduler;
};

//------------------------------------------------------------------------------

/// [32] Direct tests of the NodeStore with various backends

class NodeStoreTests : public NodeStoreUnitTest
{
public:
    NodeStoreTests () : NodeStoreUnitTest ("NodeStore")
    {
    }

    void testImport (String destBackendType, String srcBackendType, int64 seedValue)
    {
        String srcParams;
        srcParams << "type=" << srcBackendType
                  << "|path=" << File::createTempFile ("unittest").getFullPathName ();

        // Create a batch
        NodeStore::Batch batch;
        createPredictableBatch (batch, 0, numObjectsToTest, seedValue);

        // Write to source db
        {
            ScopedPointer <NodeStore> src (NodeStore::New (srcParams, "", m_scheduler));

            storeBatch (*src, batch);
        }

        String destParams;
        destParams << "type=" << destBackendType
                   << "|path=" << File::createTempFile ("unittest").getFullPathName ();

        ScopedPointer <NodeStore> dest (NodeStore::New (
            destParams, "", m_scheduler));

        beginTest (String ("import into '") + destBackendType + "' from '" + srcBackendType + "'");

        // Do the import
        dest->import (srcParams);

        // Get the results of the import
        NodeStore::Batch copy;
        fetchCopyOfBatch (*dest, &copy, batch);

        // Canonicalize the source and destination batches
        std::sort (batch.begin (), batch.end (), NodeObject::LessThan ());
        std::sort (copy.begin (), copy.end (), NodeObject::LessThan ());
        expect (areBatchesEqual (batch, copy), "Should be equal");

    }

    void testBackend (String type, int64 const seedValue)
    {
        beginTest (String ("NodeStore backend type=") + type);

        String params;
        params << "type=" << type
               << "|path=" << File::createTempFile ("unittest").getFullPathName ();

        // Create a batch
        NodeStore::Batch batch;
        createPredictableBatch (batch, 0, numObjectsToTest, seedValue);

        {
            // Open the database
            ScopedPointer <NodeStore> db (NodeStore::New (params, "", m_scheduler));

            // Write the batch
            storeBatch (*db, batch);

            {
                // Read it back in
                NodeStore::Batch copy;
                fetchCopyOfBatch (*db, &copy, batch);
                expect (areBatchesEqual (batch, copy), "Should be equal");
            }

            {
                // Reorder and read the copy again
                NodeStore::Batch copy;
                UnitTestUtilities::repeatableShuffle (batch.size (), batch, seedValue);
                fetchCopyOfBatch (*db, &copy, batch);
                expect (areBatchesEqual (batch, copy), "Should be equal");
            }
        }

        {
            // Re-open the database
            ScopedPointer <NodeStore> db (NodeStore::New (params, "", m_scheduler));

            // Read it back in
            NodeStore::Batch copy;
            fetchCopyOfBatch (*db, &copy, batch);
            // Canonicalize the source and destination batches
            std::sort (batch.begin (), batch.end (), NodeObject::LessThan ());
            std::sort (copy.begin (), copy.end (), NodeObject::LessThan ());
            expect (areBatchesEqual (batch, copy), "Should be equal");
        }
    }

public:
    void runTest ()
    {
        int64 const seedValue = 50;

        //
        // Backend tests
        //

        /// [33] The NodeStore tests exercise all the backends

        testBackend ("keyvadb", seedValue);

        testBackend ("leveldb", seedValue);

        testBackend ("sqlite", seedValue);

    #if RIPPLE_HYPERLEVELDB_AVAILABLE
        testBackend ("hyperleveldb", seedValue);
    #endif

    #if RIPPLE_MDB_AVAILABLE
```

```
        testBackend ("mdb", seedValue);
    #endif

        //
        // Import tests
        //

        //testImport ("keyvadb", "keyvadb", seedValue);

        testImport ("leveldb", "leveldb", seedValue);

    #if RIPPLE_HYPERLEVELDB_AVAILABLE
        testImport ("hyperleveldb", "hyperleveldb", seedValue);
    #endif

    #if RIPPLE_MDB_AVAILABLE
        testImport ("mdb", "mdb", seedValue);
    #endif

        testImport ("sqlite", "sqlite", seedValue);
    }

private:
    TestScheduler m_scheduler;
};

static NodeStoreTests nodeStoreTests;

static NodeStoreTimingTests nodeStoreTimingTests;
```