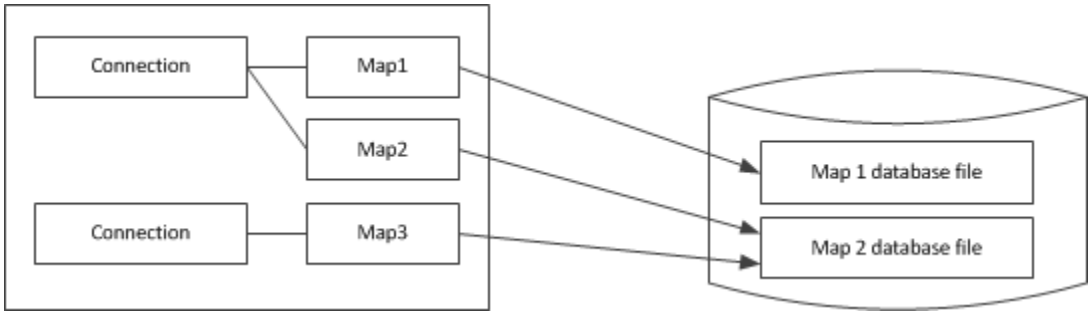
Freeze Map Concepts

On this page:

Freeze Connections

In order to create a Freeze map object, you first need to obtain a Freeze `Connection` object by connecting to a database environment.

As illustrated in the following figure, a Freeze map is associated with a single connection and a single database file. Connection and map objects are not thread-safe: if you want to use a connection or any of its associated maps from multiple threads, you must serialize access to them. If your application requires concurrent access to the same database file (persistent map), you must create several connections and associated maps.



Freeze connections and maps.

Freeze connections provide operations that allow you to begin a transaction, access the current transaction, get the communicator associated with a connection, close a connection, and remove a map index. See the Slice API reference for more information on these operations.

Using Transactions with Freeze Maps

You may optionally use transactions with Freeze maps. Freeze transactions provide the usual ACID (atomicity, concurrency, isolation, durability) properties. For example, a transaction allows you to group several database updates in one atomic unit: either all or none of the updates within the transaction occur.

You start a transaction by calling `beginTransaction` on the `Connection` object. Once a connection has an associated transaction, all operations on the map objects associated with this connection use this transaction. Eventually, you end the transaction by calling `commit` or `rollback`: `commit` saves all your updates while `rollback` undoes them. The `currentTransaction` operation returns the transaction associated with a connection, if any; otherwise, it returns nil.

Slice

```
module Freeze {

local interface Transaction {
    void commit();
    void rollback();
};

local interface Connection {
    Transaction beginTransaction();
    idempotent Transaction currentTransaction();
    // ...
};
};
```

If you do not use transactions, every non-iterator update is enclosed in its own internal transaction, and every read-write iterator has an associated internal transaction that is committed when the iterator is closed.

Using Transactions with C++

You must ensure that you either commit or roll back each transaction that you begin (otherwise, locks will be held by the database until they time out):

C++

```
ConnectionPtr connection = ...;

TransactionPtr tx = connection->beginTransaction();
try {

    // DB updates that might throw here...

    tx->commit();

    // More code that might throw here...

} catch (...) {
    try {
        tx->rollback();
    } catch (...) {
    }
    throw;
}
```

The outer try-catch blocks are necessary because, if the code encounters an exception, we must roll back any updates that were made. In turn, the attempt to roll back might throw itself, namely, if the code following the commit throws an exception (in which case the transaction cannot be rolled back because it is already committed).

Code such as this is difficult to maintain: for example, an early return statement can cause the transaction to be neither committed nor rolled back. The `TransactionHolder` class ensures that such errors cannot happen:

C++

```
namespace Freeze {
    class TransactionHolder {
    public:
        TransactionHolder(const ConnectionPtr&);
        ~TransactionHolder();

        void commit();
        void rollback();

    private:
        // Copy and assignment are forbidden.
        TransactionHolder(const TransactionHolder&);
        TransactionHolder& operator=(const TransactionHolder&);
    };
}
```

The constructor calls `beginTransaction` if the connection does not already have a transaction in progress, so instantiating the holder also starts a transaction. When the holder instance goes out of scope, its destructor calls `rollback` on the transaction and suppresses any exceptions that the rollback attempt might throw. This ensures that the transaction is rolled back if it was not previously committed or rolled back and ensures that an early return or an exception cannot cause the transaction to remain open:

C++

```
ConnectionPtr connection = ...;

{ // Open scope

    TransactionHolder tx(connection); // Begins transaction

    // DB updates that might throw here...

    tx.commit();

    // More code that might throw here...

} // Transaction rolled back here if not previously
  // committed or rolled back.
```

If you instantiate a `TransactionHolder` when a transaction is already in progress, it does nothing: the constructor notices that it could not begin a new transaction and turns `commit`, `rollback`, and the destructor into no-ops. For example, the nested `TransactionHolder` instance in the following code is benign and does nothing:

C++

```
ConnectionPtr connection = ...;

{ // Open scope

    TransactionHolder tx(connection); // Begins transaction

    // DB updates that might throw here...

    { // Open nested scope

        TransactionHolder tx2(connection); // Does nothing

        // DB updates that might throw here...

        tx2.commit(); // Does nothing

        // More code that might throw here...

    } // Destructor of tx2 does nothing

    tx.commit();

    // More code that might throw here...

} // Transaction rolled back here if not previously
  // committed or rolled back.
```

Using Transactions with Java

You must ensure that you either commit or roll back each transaction that you begin (otherwise, locks will be held by the database until they time out):

Java

```
Connection connection = ...;

Transaction tx = connection.beginTransaction();
try {

    // DB updates that might throw here...

    tx.commit();

    // More code that might throw here...

} catch (java.lang.RuntimeException ex) {
    try {
        tx.rollback();
    } catch (DatabaseException e) {
    }
    throw ex;
}
```

The catch handler ensures that the transaction is rolled back before re-throwing the exception. Note that the nested try-catch blocks are necessary: if the transaction committed successfully but the code following the commit throws an exception, the rollback attempt will fail therefore we need to suppress the corresponding `DatabaseException` that is raised in that case.

Also use caution with early `return` statements:

Java

```
Connection connection = ...;

Transaction tx = connection.beginTransaction();
try {

    // DB updates that might throw here...

    if (error) {
        // ...
        return; // Oops, bad news!
    }

    // ...

    tx.commit();

    // More code that might throw here...

} catch (java.lang.RuntimeException ex) {
    try {
        tx.rollback();
    } catch (DatabaseException e) {
    }
    throw ex;
}
```

The early `return` statement in the preceding code causes the transaction to be neither committed nor rolled back. To deal with this situation, avoid early return statements or ensure that you either commit or roll back the transaction before returning. Alternatively, you can use a `finally` block to ensure that the transaction is rolled back:

Java

```
Connection connection = ...;

try {

    Transaction tx = connection.beginTransaction();

    // DB updates that might throw here...

    if (error) {
        // ...
        return; // No problem, see finally block.
    }

    // ...

    tx.commit();

    // More code that might throw here...

} finally {
    if (connection.currentTransaction() != null)
        connection.currentTransaction().rollback();
}
```

Iterating a Freeze Map

Iterators allow you to traverse the contents of a Freeze map. Iterators are implemented using Berkeley DB cursors and acquire locks on the underlying database page files. In C++, both read-only (`const_iterator`) and read-write iterators (`iterator`) are available. In Java, an iterator is read-write if it is obtained in the context of a transaction and read-only if it is obtained outside a transaction.

Locks held by an iterator are released when the iterator is closed (if you do not use transactions) or when the enclosing transaction ends. Releasing locks held by iterators is very important to let other threads access the database file through other connection and map objects. Occasionally, it is even necessary to release locks to avoid self-deadlock (waiting forever for a lock held by an iterator created by the same thread).

To improve ease of use and make self-deadlocks less likely, Freeze often closes iterators automatically. If you close a map or connection, associated iterators are closed. Similarly, when you start or end a transaction, Freeze closes all the iterators associated with the corresponding maps. If you do not use transactions, any write operation on a map (such as inserting a new element) automatically closes all iterators opened on the same map object, except for the current iterator when the write operation is performed through that iterator. In Java, Freeze also closes a read-only iterator when no more elements are available.

There is, however, one situation in C++ where an explicit iterator close is needed to avoid self-deadlock:

- you do not use transactions, and
- you have an open iterator that was used to update a map (it holds a write lock), and
- in the same thread, you read that map.

Read operations in C++ never close iterators automatically: you need to either use transactions or explicitly close the iterator that holds the write lock. This is not an issue in Java because you cannot use an iterator to update a map outside of a transaction.

Recovering from Freeze Map Deadlocks

If you use multiple threads to access a database file, Berkeley DB may acquire locks in conflicting orders (on behalf of different transactions or iterators). For example, an iterator could have a read-lock on page P1 and attempt to acquire a write-lock on page P2, while another iterator (on a different map object associated with the same database file) could have a read-lock on P2 and attempt to acquire a write-lock on P1.

When this occurs, Berkeley DB detects a deadlock and resolves it by returning a "deadlock" error to one or more threads. For all non-iterator operations performed outside any transaction, such as an insertion into a map, Freeze catches such errors and automatically retries the operation until it succeeds. (In that case, the most-recently acquired lock is released before retrying.) For other operations, Freeze reports this deadlock by raising `Freeze::DeadlockException`. In that case, the associated transaction or iterator is also automatically rolled back or closed. A properly written application must expect to catch deadlock exceptions and retry the transaction or iteration.

Key Sorting for Freeze Maps

Keys in Freeze maps and indexes are always sorted. By default, Freeze sorts keys according to their Ice-encoded binary representation; this is very efficient but the resulting order is rarely meaningful for the application. Starting with Ice 3.0, Freeze offers the ability to specify your own comparator objects so that you can customize the traversal order of your maps. Note however that the comparator of a Freeze map should remain the same throughout the life of the map. Berkeley DB stores records according to the key order provided by this comparator; switching to another comparator will cause undefined behavior.

Key Sorting for Freeze Maps in C++

In C++, you specify the name of your comparator objects during code generation. The generated map provides the standard features of `std::map`, so that iterators return entries according to the order you have defined for the main key with your comparator object. The `lower_bound`, `upper_bound`, and `equal_range` functions provide range-searches (see the definition of these functions on `std::map`).

Apart from these standard features, the generated map provides additional functions and methods to perform range searches using secondary keys. The additional functions are `lowerBoundFor`*`Member`*, `upperBoundFor`*`Member`*, and `equalRangeFor`*`Member`*, where *`Member`* is the name of the secondary-key member. These functions return regular iterators on the Freeze map.

Key Sorting for Freeze Maps in Java

In Java, you supply comparator objects (instances of the standard Java interface `java.util.Comparator`) at run time when instantiating the generated map class. The map constructor accepts a comparator for the main key and optionally a collection of comparators for secondary keys. The map also provides a number of methods for performing range searches on the main key and on secondary keys.

Indexing a Freeze Map

Freeze maps support efficient reverse lookups: if you define an index when you generate your map (with `slice2freeze` or `slice2freezej`), the generated code provides additional methods for performing reverse lookups. If your value type is a structure or a class, you can also index on a member of the value, and several such indexes can be associated with the same Freeze map.

Indexed searches are easy to use and very efficient. However, be aware that an index adds significant write overhead: with Berkeley DB, every update triggers a read from the database to get the old index entry and, if necessary, replace it.

If you later add an index to an existing map, Freeze automatically populates the index the next time you open the map. Freeze populates the index by instantiating each map entry, so it is important that you register the object factories for any class types in your map before you open the map.

Note that the index key comparator of a Freeze map index should remain the same throughout the life of the index. Berkeley DB stores records according to the key order provided by this comparator; switching to another comparator will cause undefined behavior.

See Also

- Using a Freeze Map in C++
- Using a Freeze Map in Java