Data processing toolkit

Željko Vrba <zvrba@ifi.uio.no>

October 17, 2005

Abstract

We have designed a high-performance C++ toolkit for sequential processing of data files. It enables users to write their queries in a mostly declarative style. Along with sequential processing, it provides some features normally found in RDBMS, such as aggregate functions.

Contents

1 Introduction **1**

2 Stage reference **2**
2.1 I/O 3
2.2 Filtering 3
2.3 Windowing 3
2.4 Sorting and merging 4
2.5 Aggregation 4

3 Helper class reference **4**
3.1 FOs 4
3.2 Mapping implementations 4

4 Internals **5**
4.1 Stage basics 5
4.2 EOF condition 6
4.3 Aggregate functions 6
4.4 Mappings 7

5 Design rationale **7**
5.1 Stages 8
5.2 Mappings 8

6 Conclusion and future work **9**
6.1 Parallel computations 9
6.2 Porting to Phoenix-2 9
6.3 Mapping enhancements 9

1 Introduction

The goal of the toolkit is to encapsulate common algorithms encountered in processing of flat data files in a set of reusable components. That way the user has to write only parts of the code specific to their application, and "queries" are written in a mostly declarative style of functional composition. The toolkit is best applied to extremely large quantities of data (we have applied it to data files over 6 GB) of which only small amount needs to be resident in memory at any time.

The key concept in the toolkit is the *computation stage*. The computation stage has one or more *inputs* which themselves may be stages, and one *output*. Each stage encapsulates only the *behaviour* of an algorithm, which is then parametrized by user-supplied functions.

In its concepts, the toolkit is somewhat similar to data processing using UNIX pipes. Each program in the pipeline corresponds to one computation stage. The major differences between pipelines and our toolkit are summarized in Table 1 and are described in more detail below.

1. Programs in the pipeline have one input and one output. For example, it is not possible to pipe the output of two programs to a third one. This is not an inherent limitation, it is a consequence of the "linear" program composition syntax with the | operator, and the convention that a program reads its input from only one stream, namely `stdin`.

2. UNIX programs deal with *text-formatted* data. Each program in the pipeline has to parse the input text data into its in-memory representation and format it to text again on output. According to some measurements, the text-binary conversion process is 5 times slower than reading raw binary data directly into

```
size_t n = until_eof(
  filter(
    merge(
      typed_binary_istream<unsigned>(is1),
      typed_binary_istream<unsigned>(is2)),
    ptr_fun(is_even)),
  ptr_fun(print_int)) ();
```

Figure 1: An example computation

memory, and similarly for writing. This conversion takes place at every program in the pipeline.

For example, the toolkit provides a stage that performs the merging of two sorted sequences of items. The user has to provide as a parameter the function which decides which of the two items is lesser. Figure 1 shows an example stage composition which merges two sorted sequences of integers and outputs only the even integers. The actual code used to accomplish the computation is given in Figure 1. Missing from the example is code for `is_even` and `print_int` functions. Note the additional pair of parentheses in the invocation of the `until_eof` stage. The reason for this is given in later sections.

Also, we didn't provide a less-than comparison function in the invocation of `merge` because the toolkit uses `operator<` by default. If the overloaded operator for the type in question is defined, then everything works as expected. If not, a compile-time error is raised, and the user has a choice of either defining `operator<` or specifying a comparison function in merge invocation.

Every stage is *lazy*, that is, it does not perform any action unless it is explicitly told to do so. For example, the `merge` stage does not read any of its inputs unless *it* is told to produce the next item of the merged sequence.

The only exception is the `until_eof` stage – it is an implicit loop that executes a function on every element of the input stage until it signals an EOF (hence the name). The `until_eof` returns the number of processed elements.

Stages do not have to read data from files or other stages. It is allowed for the stage to have *no inputs*. Such a stage can be useful, for example, to generate an increasing sequence of integers or random numbers. We didn't implement any such stage that because we didn't need it for our computations.

The rest of this paper is organized as follows. Sections 2 and 3 give an overview of all stages and helper classes that come with the toolkit. These sections can be considered as a "user-manual" and assume a working knowledge of C++. Section 4 describes internals of the toolkit and assumes an advanced knowledge of C++, especially templates and STL. You should read this section if you intend to create your own computation stage compatible with the rest of the toolkit. In section 5 we give reasons for certain design decisions and in section **??** we describe our experiences on the concrete use-case.

2 Stage reference

First we introduce some terminology that is consistently used for certain *concepts* throughout the toolkit.

Computation stage has already been introduced in the introduction. For brevity we will from now on use just the term "stage".

Function object (FO), in the context of this paper, is a structure with overloaded function call operator and `result_type` typedef which declares the return type of the function call operator. An ordinary function can be converted to FO by the `std::ptr_fun` function.

In the rest of this paper we write "function" when either an ordinary function or FO is acceptable.

Item is element obtained from the stage input.

Less-than ordering is a binary function taking two items of the same type and returning true if the first argument is less than the second. It should follow the same constraints as the "StrictWeakOrdering" STL concept.

Mapping is a data structure which supports mapping of keys of arbitrary type to values of arbitrary type, which need not be the same as the type of the key.

Predicate is a unary function taking a single argument and returning a boolean value.

issue addressed	pipelines	toolkit	advantage
connectivity I:O	1:1	many:1	allows more complex algorithms
data format	text	binary	no time lost in coversion from text

Table 1: Differences between pipelines and the toolkit

concept	type name
function object	`FunT`
item	`ItemT`
less-than ordering	`LessT`
mapping	`MapT`
predicate	`PredicateT`
computation stage	`StageT`
arbitrary type	`T`

Table 2: Template types corresponding to concepts

Sequence is a sequence of items obtained by repeated calls to some stage.

The division of stages in the following subsections is by their principal function and complexity. No code examples are given here: the toolkit is delivered with several test programs, at least one for each stage. These are simple enough to serve as examples.

In this section we give only "constructor forms" for each stage. The discussion about actual types of these expressions is deferred to the next section. Return and argument types in these forms are template types whose names resemble expected *concepts* for given arguments. Table 2 lists names that are consistently used for types of corresponding concepts.

2.1 I/O

Stages in this group perform I/O of items in binary mode.

`typed_binary_istream<ItemT>(std::istream &is)` stage reads items of type `ItemT` from the open input stream `is`.

`output_binary_record(std::ostream &os)` stage outputs binary records to the open output stream `os`. It can write items of *any type*.

`until_eof(const StageT &s, const FunT &f)` reads items from the stage `s` until it signals an EOF. The `f` FO is applied to every item, and its return value is discarded. Returns the number of times `f` was invoked.

2.2 Filtering

Filtering is the simplest operation. It always operates on one item at a time.

`StageT filter(const StageT &s, const PredicateT &pred)` returns only those items `it` from `s` for which `f(it)` is true.

`StageT mapcar(const StageT &s, const FunT &f)` applies the FO `f` to each item read from `s`. The type of items returned by this stage is the return type of the `f` FO. It need not be the same as the type of items returned by `s`.

2.3 Windowing

The windowing stage implements a *sliding window* of arbitrary size, within the limits of available memory. It cannot be instantiated directly, as it requires a corresponding *placer*. The full discussion is postponed until section 4. Here we describe its use within the library.

`StageT wsort(const StageT &s, size_t wsz)` is a windowed-sort stage that performs item comparison using `operator<` on items returned by `s`. It maintains a sorted linked list of `wsz` elements in memory with the smallest element at the *front*. When a new item arrives, an item is removed from the front, and a place for the new item is looked for from the *back*. If the item is not badly out of order, the search terminates quickly. The best case is when the item is in sorted order: only one comparison is performed, and the item is inserted at the back of the list.

`StageT wsort(const StageT &s, size_t wsz, const LessT &l)` is the same as above, except that it performs element comparison using the `LessT` FO instead of the `operator<`.

Let $a_1, a_2, \ldots, a_n$ be a sequence of elements. We say that two elements a_i and a_j with $i < j$ make an *inversion* of length $j - i$ if $a_i > a_j$. The output of `wsort` is sorted if `wsz` is at least as large as the longest inversion in the sequence.

2.4 Sorting and merging

`StageT merge(const StageT1 &s1, const StageT2 &s2)` merges two sorted sequences into one sorted sequence using the `operator<` for comparison. Both stages must return items of the same type.

`StageT merge(const StageT1 &s1, const StageT2 &s2, const LessT &l)` is the same as above, except that an explicit comparison function is provided.

`StageT inmemory_sort(const StageT &s)` and its pair `StageT inmemory_sort(const StageT &s, const LessT &l)` read *into memory* all available items from the stage `s` and return them in sorted order. For obvious reasons, this stage can't be used if the input stage produces an *infinite* sequence.

2.5 Aggregation

Aggregation is the most complex operation currently present in the toolkit. It is best described with the help of the following SQL statement:

```
SELECT AVG(a-b), c FROM some_table
GROUP BY c;
```

`AVG` is an SQL aggregate function. As such, it has a *state* consisting of the current sum and the number of items. The *initial state* has both elements set to 0. For each different value of the `c` column, a separate `AVG` state is kept. When all items have been processed, the aggregate function returns its *final value*, namely, the total sum divided by the total number of items.

`StageT aggregate<AggregateT, MapT>(const StageT &s, const KeyT &k, const ValueT &v)` creates an aggregate function. The key FO `k` and the value FO `v` are evaluated for each item returned by the stage `s`. `MapT` is searched for the key returned by `k`. If the key isn't found, a new state is instantiated, otherwise the found state is updated with the value returned by `v`.

`AggregateT` is a template class that actually computes the aggregate value. `MapT` is a template class that is used to associate keys returned by the key function with aggregate function state corresponding to the key.

Returning to the SQL example above, the key function would return the `c` field, while the value function would return `a-b`. `AggregateT` corresponds to the `AVG` function. Each time `AVG` is invoked with an item and a state, it adds the item's value to the sum and increases the number of items in the state. `MapT` cannot be directly related to anything in the SQL query – it is a detail hidden in the implementation of the database.

The toolkit comes with several predefined aggregate functions:

- `max_a` and `min_a` calculate the maximum, and respectively the minimum of values.
- `count_a` counts the number of seen items, irrespective of their value.

There are also three map implementations: `std_map`, `bdb_map` and partial specialization of `bdb_map` that correctly handles `char*` values, and two useful FOs: `identity_f` and `constant_f`. All of these classes are described in the next section.

3 Helper class reference

The following classes are *not* data-processing stages. Still, they are crucial for the functionality of some stages.

3.1 FOs

`identity_f()` is a non-template FO that operates on values of *any type*, simply returning what was received in its only argument.

`constant_f<T>(const T &v)` internally stores a copy of `v`. Its function call accepts arguments of *any type* and irrespective of its value always returns `v`.

3.2 Mapping implementations

Mappings are internally used by aggregates to associate keys returned by the key function with states. All data held by the mapping, both keys and values, is stored *by value*, with no interpretation. This means that both keys and values must be copy-constructible. Also, the mapping itself must be copy-constructible. Within the toolkit, the mapping is copied only when it is empty, so this doesn't present a performance problem.

If the programmer is not careful, value semantics can have surprising effects when pointers are used

as keys or values: the value of the *pointer itself* is used, and *not the value it points to*.

The mapping interface is discussed in the Internals section, here we describe just the provided mappings and make the reader aware of some possible pitfalls.

`intkey_map<KeyT, ValueT>` is implemented on top of `std::vector` where keys *must* be unsigned integers. The used memory is proportional to the largest key. If keys are sparse and numerically large, this may lead to unacceptable memory usage, so another implementation should be used in this case. Referencing a non-existent large key will not cause reallocation – it happens only when inserting values.

`std_map<KeyT, ValueT>` is implemented on top of `std::map`. For this map, the key type *must* be less-than comparable with the standard operator `<`. If string values are desired for, then `std::string` should be specified for `ValueT`. The standard string class is constructible from `char*` and makes a copy of it.

`bdb_map<KeyT, ValueT>` is backed by the BerkeleyDB database (BDB) using BTree access method and default comparison function which is a byte-for-byte lexicographical comparison. This implementation is suitable for very large number of items which do not fit all in memory.

`bdb_map<KeyT, char*>` is a partial specialization of the above general mapping that correctly stores null-terminated `char*` strings as values. A *copy* of the string, instead of the bare pointer is stored.

Currently, the toolkit provides no way, except by modifying the toolkit source itself, to change either the access method from BTree to Hash, or to set the comparison function.

The default comparison function in BDB may not be optimal for a particular key type, but will work correctly *provided that there is no padding within the key type*. If the key type is a `struct`, padding bytes inserted for alignment purposes may appear between members. These bytes have *undefined content* unless explicitly set. With padding, the default comparison function may return *inconsistent results.* One of the following things can be done to eliminate inconsistencies:

- Manually `memset` the key structure to 0 before filling in the relevant fields. This way padding bytes always have a known value which is never changed once set.
- Structure packing.

The same reasoning holds for the BDB default hash function.

In our data analysis programs we have adopted the second approach, since structure packing is on most compilers a once-only directive attached to the `struct` declaration. The first case is more error-prone, since the programmer has to remember to explicitly `memset` the key structure to 0 before giving it to BDB, or create a default constructor that does this.

4 Internals

This section, combined with the distributed toolkit source, should give sufficient information to readers that wish to extend the toolkit.

4.1 Stage basics

A stage which returns values of type `RET` has to provide the following *minimal interface*:

```
struct stage {
  typedef RET result_type;
  RET operator()();
};
```

The function call operator does not take any arguments. Inputs to the stage are given in its constructor (not shown here, as it is stage-dependant) which stores them in member variables. Then the `operator()` invokes the input stages as neccessary.

`result_type` typedef makes it possible to *compose* stages, as in the example in the previous section. Given this typedef, it is trivial for the stage to find out return types of its inputs.

To make stage composition possible, the constructor has to take arbitrary stages as inputs and store them into member variables. This requires the use of templates. For example, the `merge` stage has the following interface:

```
template<typename StageT,
  typename LessT =
    std::less<StageT::result_type> >
struct merge_c {
```

```
  typedef typename StageT::result_type
    result_type;

  StageT stage1, stage2;
  LessT  less;

  merge_c(const StageT &s1,
    const StageT &s2,
    const LessT &l) :
  stage(s1), stage(s2), less(l)
  { /* other initialization code */ }
};
```

Few points need to be emphasized:

- The stage is a template class, parametrized by the types of its input stages. In this case, both input stages must deliver items of *the same type*, therefore only *one* template parameter (`StageT`) for *two* inputs.
- The stage class has an additional policy parameter – the less-than ordering function type (`LessT`) which defaults to the standard `operator<`.
- The constructor takes input stages by const reference. This is just a minor optimization.
- More importantly, explicit instantiation of the `merge_c` stage would be painful as template type deduction from its constructor arguments doesn't work for classes.

Therefore, *constructor functions* are provided:

```
template<typename StageT1, typename StageT2>
inline merge_c<StageT1, StageT2>
merge(const StageT1 &p1, const StageT2 &p2)
{
  return merge_c<StageT1, StageT2>(p1, p2);
}

template<typename StageT1, typename StageT2,
  typename LessT>
inline merge_c<StageT1, StageT2, LessT>
merge(const StageT1 &p1, const StageT2 &p2,
  const LessT &l)
{
  return merge_c<StageT1, StageT2,
    LessT>(p1, p2, l);
}
```

These functions are defined in the same header as the `merge_c` class. Because of the `inline` specifier, there won't be any linker errors even when included in different compilation units. The first function instantiates the `merge_c` class with the default ordering function, while the second takes it as an additional parameter. The reason two functions are needed is because template functions do not allow default template arguments.

4.2 EOF condition

The toolkit defines a `stop_iteration` exception, which is an *empty* class. Stages that want to detect the EOF condition of its inputs, so that it doesn't propagate further, must catch this exception and handle it appropriately.

An example of EOF handling and signaling can be found in the implementation of the already mentioned merge algorithm.

4.3 Aggregate functions

The anatomy of an aggregate function is best shown in the example of the `sum_a` aggregate, which sums all its inputs.

```
template<typename ValueT>
struct sum_a {
  typedef ValueT result_type;

  struct state_type {
    ValueT sum;

    state_type() { }
    state_type(const ValueT &v) : sum(v)
    { }
  };

  state_type &operator()(state_type &state,
  const ValueT &v) {
    state.sum += v;
    return state;
  }

  ValueT operator()(
  state_type const &state) {
    return state.sum;
  }
};
```

Here are some points to note:

- An aggregate is a template function. The template type denotes the type of items that are *input* to the aggregate.
- The `result_type` typedef must be present. It needs not be the same as the `ValueT`. For example, in the `count_a` aggregate it is always `size_t`, independent of the type of input items.
- The aggregate function must define an inner `state_type` which holds the current state of the computation. In this case, it is the current sum of elements.
- *Two* overloaded variants of the function call operator.

The `state_type` must provide both a default constructor and a constructor which takes a single argument of type `ValueT`. It must also be copy-constructible. Here the constructor value is passed by const reference as an optimization, and the compiler-generated copy constructor (i.e. byte copy) is OK in this case. The default constructor is needed because instances of `state_type` are held in mappings, which must be able to default construct their values.

When the aggregate receives its first input, its state is initialized with the 1-argument constructor. In this case, the current sum is set to be equal to the first element seen.

The two-argument form of `operator()` is used while the computation is in progress. It receives the current state *by reference*, updates it and returns the *same reference*. The 0-argument form of `operator()` delivers the *final result* of the computation,

4.4 Mappings

Mapping implementation must be a template class parametrized by the key and value type, here written as `KeyT` and `ValueT`. It must implement the following public methods:

- `ValueT *get(const KeyT&)` retrieves a pointer to the *modifiable* value associated with the key. NULL pointer is returned if the key doesn't exist. Each call to this function *invalidates* all pointers returned previously.
- `void put(const KeyT&, const ValueT*)` replaces or creates the key with the value. Current implementation does not support multiple values for one key.
- `bool next(const KeyT**, const ValueT**)` returns pointers to the "next" key-value pair in the mapping. Returns `false` when all records are exhausted. The records are traversed in the order *defined by the implementation*.

All returned pointers are *owned by the mapping class itself* and should not be freed by the user code.

5 Design rationale

We realized that all operations we need to perform over the data are simple *algorithms*, like filtering out elements by some criteria. We need the ability to *compose* different algorithms to achieve different computations, for example sorting items by some criteria after filtering out the unwanted ones.

In designing our toolkit, the STL has given a good starting point. However, with STL it is not possible to compose functions. Even with the SGI's extensions `compose1st` and `compose2nd` the invocation is hard to read.

In the following subsections, we describe reasons for the design decisions taken during the development of the toolkit.

There exist far better programming languages than C++ for writing functional code, like Ocaml, LISP and Python. Our requirement is an efficient compiler to native code and easy interfacing with native C++ data types. Ocaml fits both requirements, but we are not experienced enough in Ocaml to start any project with it. There are many LISP implementations with efficient compilers, but they fail the requirement of easy interfacing to native data types. Python fails the requirement of efficient compiler: it was measured to be several *times* slower than an equivalent C++ program, with much time spent in converting native data types to Python types.

5.1 Stages

The toolkit heavily relies on the functional programming style. Closures are simulated by C++ function objects, and (static) function polymorphism is achieved through the use of templates.

The key concept in the toolkit is the *stage*. A stage is a *nullary function* which, when called, returns the next item from the data stream. When there is no more data in the stream, an *exception* is raised.

We have decided that stages should be *passive*, i.e. they should deliver the data on demand instead of actively supplying the data to its consumers. Doing it the other way around would create big problems with stages that first consume their whole input, and then produce an output, e.g. aggregates.

Stages return all data *by value* instead of by reference. This certainly causes some in-memory data copying overhead, but we believe that the gained *safety* is much more important – there are no dangling pointers or references to worry about, or even getting incorrect results because some shared value has been inadvertently modified. This way, the programmer also does not have to worry much about const-correctness since all data is copied and safe to modify without concern that some other function may be referencing the same value. Also, the overhead of data copying is somewhat compensated by the absence of dynamic memory management using `new` and `delete`.

If stages were to return *pointers* to items, it would be easy to mark the end of the data by returning the NULL pointer. But this is not the case and, unlike some languages, there is no "universal value" (e.g. `nil` in LISP or `option` type in Ocaml) which could be used to mark the end of the stream. We were faced with one of the following choices to resolve the problem:

- Make each value a `std::pair` where the first item in the pair is a value returned by the stage, and the second item is `bool`. If the second item is `false`, then there is no more data in the stream and the value of the first item should be considered *unknown*.
- Deliberately disregard the popular C++ coding recommendation of not using exceptions for control-flow by *throwing an exception* when there is no more data in the stream.

We have chosen the second approach based on the premise that the first will unnecessarily clutter the code with many extra end-of-file (EOF) checks and its propagation to "upper-level" stages. In retrospect, this has shown to be an excellent decision. The code is clean and very easy to follow. Many stages do not even check for the EOF condition but just let it propagate upwards. Using the first approach, every stage would explicitly have to check in several places for end-of-stream conditions of *its* stages and correctly propagate the EOF state. The user of the library is not aware of exceptions since they are handled entirely within the library code.

Stages can be *composed*, that is, a stage A can have another stage, B, as *its* stage. This makes it simple to construct complicated queries from simple ones. Composition is made possible because every stage must have an internal `result_type` typedef.

5.2 Mappings

Since mappings are intended as helper classes, expected to be used often in different circumstances, we do not want to throw exceptions – they clutter the code much more than `if()` when there are many checks. Rather, we return pointers or boolean status codes.

The `get` function returns a pointer so that it can be used in functional-style code. It cannot return a reference since there are no NULL references in C++ and, as explained before, there is no universal value which could be returned to signify the *absence* of value.

The `put` function takes a value pointer only for interface consistency with `get`, so that values are always passed as pointers.

The `next` function has to return *two* values: the key and the associated value. This is impossible in C++. Instead of splitting the return value in two parts, the function's return value and one pointer argument, we return both values by pointer arguments. Modifying the mapping structure while traversing it can have disasterous effects, depending on the implementation, so we disallow it by returning const pointers to keys and values. At the same time, no object copying is performed during the traversal.

An alternative design would return a pair with two const pointers with the key pointer set to

NULL if there are no more elements.

In our code, one approach doesn't have any advantage over the other.

6 Conclusion and future work

We have successfully used the toolkit to analyze a large amount of Apache web server access logs – about 160 million variable sized records. The records are of varying length because of the URL string.

The toolkit is not designed to efficiently handle variable-length data. In our analyses, we are often not interested in the URLs themselves, but only whether they are different. Therefore, we replaced the URL in the record with its *MD5 hash "folded" to 64 bits*, thus obtaining fixed-length records.

The MD5 hash is 128 bits long, which is too much for the amount of data that we are dealing with: on the order of 10 million different URLs. By xoring 64-bit halves of the MD5 hash, we obtain the "folded" hash with a still very small probability of collision for our amount of data.

We have written a Python program that parses the Apache common log format and packs all relevant information, including the hash of the URL, into a fixed-length binary record. It is very slow, but this work is done only once, and Python is much more suitable for text-processing than C++.

As we need the possibility to analyze individual URLs, the script also inserts the folded MD5 hash as the key and the real URL as the value in a BDB database. In some computations, this database is accessed by the `bdb_map` partial specialization for null-terminated string values. The folded MD5 hash collision checking is not implemented, although this script is the right place to do so.

The toolkit is far from complete, even though it excellently serves the purpose in its current form. Rough measurements show that the toolkit can outperform the PostgreSQL RDBMS by a factor of 10. The following subsections describe some possible enhancements that we currently have in mind.

6.1 Parallel computations

In some cases it would be convenient to compute two values in parallel. One example is computing both the minimum and the maximum value in some data set, in one pass. Currently, the user has to write their own aggregate function, although both `min_a` and `max_a` are provided by the library.

To this end, there are two classes in the `stage_parallel.h` header: `parallel2_c` and `replicate_c`. The replicate class replicates its input in n copies before consuming the next input from its provider. In combination with the parallel composition stage, it could be used to accomplish the goal stated in the previous paragraph. However, there are two problems:

1. The replicator needs to be *named* (stored in a variable) to be able to pass it as an input to more than one stage. More on this problem in the next subsection.

2. The way the aggregate algorithm is now implemented, it first consumes its whole input before producing output.

The second point is a problem because the first aggregate in the parallel composition stage will consume its whole input – i.e. both copies of each item produced by the replicator. Elegantly solving this problem, also taking into account that performance should not suffer, is not straightforward.

6.2 Porting to Phoenix-2

Storing a replicator in a variable is currently very cumbersome, if not impossible, unless one wishes to use the `typeof` extension that some C++ compilers offer. This is because it can have a very complex type. If the toolkit were ported to the Phoenix-2 framework, we could use its scoped `lambda` expressions where the compiler would deduce the type by itself.

In fact, an initial implementation of the toolkit was implemented within the Phoenix-2 framework. However, at that time, Phoenix-2 was still in developement and we encountered some show-stopping bugs. So we decided to implement the toolkit independently.

6.3 Mapping enhancements

Generic API should be devised so that mappings could deal with multiple values per key.

BDB-backed mappings should be redesigned to allow defining comparison and hash functions. Ideally, the template would, at the point of instantiation, discover by itself whether `operator<` had already been overloaded for the key type in question and generate the code to set the comparison function automatically. Similarly, it could discover whether some `hash` static member function existed and generate the code to set the hash function automatically.