# frovedis::dunordered_map<K,V>

## NAME

frovedis::dunordered_map<K,V> - a distributed unordered_map with key-type 'K' and value-type 'V' supported by frovedis

## SYNOPSIS

#include <frovedis.hpp>

### Constructors

dunordered_map ()
dunordered_map (const dunordered_map<K,V>& src)
dunordered_map (dunordered_map<K,V>&& src)

### Overloaded Operators

dunordered_map<K,V>& operator= (const dunordered_map<K,V>& src)
dunordered_map<K,V>& operator= (dunordered_map<K,V>&& src)

### Public Member Functions

template <class R, class F>
dunordered_map<K,R> map_values(const F& f);

template <class R, class U, class F>
dunordered_map<K,R> map_values(const F& f, const node_local<U>& l);

template <class R, class U, class W, class F>
dunordered_map<K,R> map_values(const F& f, const node_local<U>& l1,
    const node_local<W>& l2);

template <class R, class U, class W, class X, class F>
dunordered_map<K,R> map_values(const F& f, const node_local<U>& l1,
    const node_local<W>& l2, const node_local<X>& l3);

template <class R, class U, class W, class X, class Y, class F>
dunordered_map<K,R> map_values(const F& f, const node_local<U>& l1,
    const node_local<W>& l2, const node_local<X>& l3,
    const node_local<Y>& l4);

```
template <class R, class U, class W, class X, class Y, class Z, class F>
dunordered_map<K,R> map_values(const F& f, const node_local<U>& l1,
       const node_local<W>& l2, const node_local<X>& l3,
       const node_local<Y>& l4, const node_local<Z>& l5);

template <class R, class KK, class VV>
dunordered_map<K,R> map_values(R(*f)(KK, VV));

template <class R, class U, class KK, class VV, class UU>
dunordered_map<K,R> map_values(R(*f)(KK,VV,UU), const node_local<U>& l);

template <class R, class U, class W, class KK, class VV, class UU, class WW>
dunordered_map<K,R> map_values(R(*f)(KK,VV,UU,WW), const node_local<U>& l1,
       const node_local<W>& l2);

template <class R, class U, class W, class X,
     class KK, class VV, class UU, class WW, class XX>
dunordered_map<K,R> map_values(R(*f)(KK,VV,UU,WW,XX), const node_local<U>& l1,
       const node_local<W>& l2, const node_local<X>& l3);

template <class R, class U, class W, class X, class Y,
     class KK, class VV, class UU, class WW, class XX, class YY>
dunordered_map<K,R> map_values(R(*f)(KK,VV,UU,WW,XX,YY), const node_local<U>& l1,
       const node_local<W>& l2, const node_local<X>& l3,
       const node_local<Y>& l4);

template <class R, class U, class W, class X, class Y, class Z,
     class KK, class VV, class UU, class WW, class XX, class YY, class ZZ>
dunordered_map<K,R> map_values(R(*f)(KK,VV,UU,WW,XX,YY,ZZ), const node_local<U>& l1,
       const node_local<W>& l2, const node_local<X>& l3,
       const node_local<Y>& l4, const node_local<Z>& l5);

template <class F>
dunordered_map<K,V>& mapv(const F& f);

template <class U, class F>
dunordered_map<K,V>& mapv(const F& f, const node_local<U>& l);

template <class U, class W, class F>
dunordered_map<K,V>& mapv(const F& f, const node_local<U>& l1,
       const node_local<W>& l2);

template <class U, class W, class X, class F>
dunordered_map<K,V>& mapv(const F& f, const node_local<U>& l1,
       const node_local<W>& l2, const node_local<X>& l3);

template <class U, class W, class X, class Y, class F>
dunordered_map<K,V>& mapv(const F& f, const node_local<U>& l1,
       const node_local<W>& l2, const node_local<X>& l3,
       const node_local<Y>& l4);

template <class U, class W, class X, class Y, class Z, class F>
dunordered_map<K,V>& mapv(const F& f, const node_local<U>& l1,
       const node_local<W>& l2, const node_local<X>& l3,
       const node_local<Y>& l4, const node_local<Z>& l5);

template <class KK, class VV>
dunordered_map<K,V>& mapv(void(*f)(KK,VV));

template <class U, class KK, class VV, class UU>
dunordered_map<K,V>& mapv(void(*f)(KK,VV,UU), const node_local<U>& l);
```

```
template <class U, class W, class KK, class VV, class UU, class WW>
dunordered_map<K,V>& mapv(void(*f)(KK,VV,UU,WW), const node_local<U>& l1,
       const node_local<W>& l2);

template <class U, class W, class X,
     class KK, class VV, class UU, class WW, class XX>
dunordered_map<K,V>& mapv(void(*f)(KK,VV,UU,WW,XX), const node_local<U>& l1,
       const node_local<W>& l2, const node_local<X>& l3);

template <class U, class W, class X, class Y,
     class KK, class VV, class UU, class WW, class XX, class YY>
dunordered_map<K,V>& mapv(void(*f)(KK,VV,UU,WW,XX,YY), const node_local<U>& l1,
       const node_local<W>& l2, const node_local<X>& l3,
       const node_local<Y>& l4);

template <class U, class W, class X, class Y, class Z,
     class KK, class VV, class UU, class WW, class XX, class YY, class ZZ>
dunordered_map<K,V>& mapv(void(*f)(KK,VV,UU,WW,XX,YY,ZZ), const node_local<U>& l1,
       const node_local<W>& l2, const node_local<X>& l3,
       const node_local<Y>& l4, const node_local<Z>& l5);

template <class F> dunordered_map<K,V> filter(const F& f);
template <class KK, class VV> dunordered_map<K,V> filter(bool(*f)(KK,VV));

template <class F> dunordered_map<K,V>& inplace_filter(const F& f);
template <class KK, class VV> dunordered_map<K,V>& inplace_filter(bool(*f)(KK,VV));

void clear();
size_t size();

void put(const K& key, const V& val);
V get(const K& key);
V get(const K& key, bool& found);

dvector<std::pair<K,V>> as_dvector();
node_local<MAP<K,V>> as_node_local();
node_local<MAP<K,V>> moveto_node_local();
node_local<MAP<K,V>> viewas_node_local();
```

# DESCRIPTION

`frovedis::dunordered_map<K,V>` can be considered as the distributed version of `std::unordered_map<K,V>`. Memory management is similar to unordered_map (RAII): when a dunordered_map is destructed, the related distributed data is deleted at the time. It is possible to copy or construct it from an existing dunordered_map. In this case, distributed data is also copied (if the source variable is an rvalue, the system tries to avoid copy).

In dunordered_map, each item (Key-Value pair) is distributed according to the hash value of the Key. In addition, the Key should be unique just like unordered_map (not multimap).

Usually, dunordered_map is created from a dvector (see manual of dvector), whose actual type should be `dvector<std::pair<K,V>>` by performing group_by_key() or reduce_by_key() like operations.

Like dvector, dunordered_map provides a global view of the distributed unordered map to the user. When operating on a dunordered_map, user can simply specify the intended operation to be performed on each Key of the dunordered_map (not on each local partition of the worker data). Thus it is simpler to handle a dunordered_map like an std::unordered_map, even though it is distributed among multiple workers. The next section explains functionalities on a dunordered_map in details.

## Constructor Documentation

### dunordered_map ()

This is the default constructor which creates an empty dunordered_map. But it does not allocate data, like normal container. See make_dunordered_map_allocate().

### dunordered_map (const `dunordered_map<K,V>`& src)

This is the copy constructor which creates a new dunordered_map with key-type K and value-type V by copying the distributed data from the input dunordered_map.

### dunordered_map (`dunordered_map<K,V>`&& src)

This is the move constructor. Instead of copying the input rvalue dunordered_map, it attempts to move the contents to the newly constructed dunordered_map. It is faster and recommended when input dunordered_map will no longer be needed.

## Overloaded Operator Documentation

### `dunordered_map<K,V>`& operator= (const `dunordered_map<K,V>`& src)

It copies the source dunordered_map object into the left-hand side target dunordered_map object of the assignment operator "=". After successful copying, it returns the reference of the target dunordered_map object.

### `dunordered_map<K,V>`& operator= (`dunordered_map<K,V>`&& src)

Instead of copying, it moves the contents of the source rvalue dunordered_map object into the left-hand side target dunordered_map object of the assignment operator "=". It is faster and recommended when source dunordered_map object will no longer be needed. It returns the reference of the target dunordered_map object after the successful assignment operation.

## Public Member Function Documentation

### map_values()

The map_values() function is used to specify the target operation to be mapped on each Key of a dunordered_map. It accepts a function or a function object (functor) and applies the same to each Key of the dunordered_map in parallel at the workers. Then a new dunordered_map is created from the return value of the function.

Along with the function argument, map_values() can accept maximum of five distributed data of node_local type. This section will explain them in details.

```
dunordered_map<K,R> map_values(R(*f)(KK,VV));
```

Below are the points to be noted while using the above map_values() interface.

- it accepts only the function to be mapped on each key as an argument.

- the input function must accept a key parameter of type KK and a value parameter of type VV, where KK and VV must be same or compatible with K and V (the key and value type of the target dunordered_map).
- the return type, R can be anything. The value type of the resultant dunordered_map will be of the same type. The key type will remain same.

For example,

```
std::vector<int> func1(int k, std::vector<int>& v) {
  std::vector<int> tmp; for(auto& i: v) tmp.push_back(2*i); return tmp;
}
std::vector<float> func2(int k, std::vector<int>& v) {
  std::vector<float> tmp; for(auto& i: v) tmp.push_back(2*i); return tmp;
}
std::vector<float> func3(int k, std::vector<float>& v) {
  std::vector<float> tmp; for(auto& i: v) tmp.push_back(2*i); return tmp;
}

std::vector<std::pair<int,int>> v;
v.push_back(make_pair(1,100));
v.push_back(make_pair(2,200));
v.push_back(make_pair(1,300));
v.push_back(make_pair(2,400));

// m would be a dunordered_map<int,std::vector<int>>
auto m = make_dvector_scatter(v).group_by_key<int,int>();
auto m2 = m.map_values(func1); // ok, m2: dunordered_map<int,vector<int>>
auto m3 = m.map_values(func2); // ok, m3: dunordered_map<int,vector<float>>
auto m4 = m.map_values(func3); // error

// it is possible to chain the map_values calls
// ok, m5: dunordered_map<int,vector<float>>
auto m5 = m.map_values(func2).map_values(func3);
```

"m" is dunordered_map<int,vector<int>>,

func1() expects (int,vector<int>) -> OK and returns vector<int> -> OK. Resultant dunordered_map, "m2" becomes dunordered_map<int,vector<int>>.

func2() expects (int,vector<int>) -> OK and returns vector<float> -> OK (return value-type can differ). Resultant dunordered_map, "m3" becomes dunordered_map<int,vector<float>>.

func3() expects (int,vector<float>) -> vector<int> and vector<float> are incompatible, thus it will lead to a compilation error.

Result of "m.map_values(func2)" is dunordered_map<int,vector<float>> and func3() expects (int,vector<float>) -> thus no issues. func3() returns vector<float>, thus "m5" becomes dunordered_map<int,vector<float>>.

Note that, the key parameter "k" was not used in any of the above input functions for map_values(). But this is required to map the functions on each key of the source dunordered_map objects.

In the above case, functions accepting only two arguments (key and value) would be allowed to pass. If any other arguments are to be passed, different version of map_values() interface needs to be used. Frovedis supports map_values() interface which can accept a function with maximum of five arguments as follows.

```
dunordered_map<K,R> map_values(R(*f)(KK,VV,UU,WW,XX,YY,ZZ),
               const node_local<U>& l1,
               const node_local<W>& l2, const node_local<X>& l3,
               const node_local<Y>& l4, const node_local<Z>& l5);
```

When using the map_values() interface accepting function to be mapped with more than two arguments (arguments other than key and values), the below points are to be noted.

- the first argument of the map_values interface must be the function pointer to be mapped on the target dunordered_map.
- the key and value type of the dunordered_map and the type of the first two function arguments must be of the same or of compatible type.
- the other arguments of the map_values (apart from the function pointer) must be of distributed `node_local<T>` type, where "T" can be of any type and the corresponding function arguments should be of the same type.
- the return type, R can be anything. The value type of the resultant dunordered_map will be of the same type. The key time will remain same.

The mapping of the argument types of the map_values() call and the argument types of the function to be mapped on a dunordered_map, "um" will be as follows:

```
 func(key,val,x1,x2,x3,x4,x5);      um.map_values(func,l1,l2,l3,l4,l5);
 -----------------------------      ----------------------------------
    key: K, val: V                  dv: dunordered_map<K,V>
    x1: U                           l1: node_local<U>
    x2: W                           l2: node_local<W>
    x3: X                           l3: node_local<X>
    x4: Y                           l4: node_local<Y>
    x5: Z                           l5: node_local<Z>
```

For example,

```
std::vector<int> func1(int k, std::vector<int>& v, int n) {
  std::vector<int> tmp; for(auto& i: v) tmp.push_back(n*i); return tmp;
}

// let's consider "m" is a dunordered_map<int,vector<int>>
// key-value type of "m" and type of the first two arguments of func1() -> Ok
// But third argument of the map_values() is simply "int" type,
// thus it will lead to an error.
auto m1 = m.map_values(func1, 2); // error

// broadcasting "2" to all workers to obtain node_local<int>.
// m2: dunordered_map<int,vector<int>>
auto m2 = m.map_values(func1, broadcast(2)); // Ok
```

Thus there are limitations on map_values() interface. It can not accept more than five distributed parameters. And also all of the parameters (except function pointer) have to be distributed before calling map (can not pass non-distributed parameter).

These limitations of map_values() can be addressed with the map_values() interfaces accepting functor (function object), instead of function pointer. This section will explain them in details.

Below are the points to be noted when passing a functor (function object) in calling the map_values() function.

- the first argument of the map_values() interface must be a functor definition.
- the key-value type of the dunordered_map must be same or compatible with the type of the first two arguments of the overloaded "operator()" of the functor.
- apart from the functor, the map_values() interface can accept a maximum of five distributed node_local objects of any type as follows.

  dunordered_map map_values(const F& f, const node_local& l1,
  const node_local& l2, const node_local& l3,
  const node_local& l4, const node_local& l5);

Where U, W, X, Y, Z can be of any type and the corresponding arguments of the overloaded "operator()" must be of the same or compatible type.

- the functor itself can have any number of data members of any type and they need not to be of the distributed type and they must be specified with "SERIALIZE" macro. If the functor does not have any data members, then the "struct" definition must be ended with "SERIALIZE_NONE" macro.
- the return type, R of the overloaded "operator()", can be anything. The value-type of resultant dunordered_map would be of the same type. The key-type will remain same. But the value-type needs to be explicitly defined while calling the map_values() interface.

For example,

```
struct foo {
  foo() {}
  foo(int n_): n(n_) {}
  std::vector<int> operator() (int k, std::vector<int>& v) {
    std::vector<int> tmp; for(auto& i: v) tmp.push_back(n*i); return tmp;
  }
  int n;
  SERIALIZE(n)
};

// let's consider "m" is a dunordered_map<int,vector<int>>
auto m1 = m.map_values(foo(2)); // error in type deduction
auto m2 = m.map_values<vector<int>>(foo(2)); // ok
```

In the above call of map_values(), it is taking a function object with "n" value as 2. Since it is the value for initializing the member of the function object, it can be passed like a simple constructor call.

"m" is `dunordered_map<int,vector<int>>` and map_values() is called with only functor definition (operator() accepting int and `vector<int>`). Thus it will be fine. Return type is of operator() is `vector<int>` which can be of any type and needs to be explicitly mentioned while calling the map_values() function like `map<vector<int>>()` (otherwise some compiler errors might be encountered).

Like map_values() with function pointer, map with function object can also accept up to five distributed node_local objects of any type.

Using function object is a bit faster than using a function, because it can be inline-expanded. On SX, it might become much faster, because in the case of function pointer, the loop cannot be vectorized, but using function object makes it possible to vectorize the loop.

## mapv()

The mapv() function is also used to specify the target operation to be mapped on each key of the dunordered_map. It accepts a void returning function or a function object (functor) and applies the same to each key of the dunordered_map in parallel at the workers. Since the applied function does not return anything, the mapv() function simply returns the reference of the source dunordered_map itself in order to support method chaining while calling mapv().

Like map_values(), mapv() has exactly the same rules and limitations. It is only different in the sense that it accepts non-returning (void) function or function object. It can not be mapped on a function which returns something other than "void".

For example,

```
void func1(int k, std::vector<int> v) {
  for(auto i=0; i<v.size(); ++i) v[i] *= 2; // updates on temporary v local to func1()
}
void func2(int k, std::vector<int>& v) {
  for(auto i=0; i<v.size(); ++i) v[i] *= 2; // in-place update
}
std::vector<int> func3(int k, std::vector<int> v) {
  std::vector<int> tmp; for(auto& i: v) tmp.push_back(2*i); return tmp;
}

// let's consider "m" is a dunordered_map<int,vector<int>>
m.mapv(func1); // Ok, but "m" would remain unchanged.
m.mapv(func2); // Ok, all the values of "m" associated with a key would be doubled.
m.mapv(func3); // error, func3() is a non-void function

// method chaining is allowed (since mapv returns reference to
// the source dunordered_map)
auto r = dv.mapv(func2).map_values(func3); // Ok
```

Here the resultant dunordered_map "r" will be of `<int,vector<int>>` type and all its values associated with a particular key will contain 4 times of the initial values. While mapping func2() on the keys of "m", its associated values will be doubled in-place and the mapv() will return the reference of the updated "m" on which the next map_values() function will apply the function func3() to double values associated with each key once again (not in-place) and will return a new `dunordered_map<int,vector<int>>`.

## filter()

Some specific values from a dunordered_map can be filtered out with the help of filter() function. It accepts a function or a function object specifying the condition on which the value is to be filtered out from the dunordered_map. The type of the function arguments must be same or compatible with the key-value type of the dunordered_map and the function must return a boolean value (true/false).

```
dunordered_map<K,V> filter(const F& f);
dunordered_map<K,V> filter(bool(*f)(KK,VV));
```

On success, it will return a new dunordered_map of same key-value type containing the filtered out elements.

For example,

```
bool is_even(int k, std::vector<int>& v) { return k%2 == 0; }

// let's consider "m" is a dunordered_map<int,vector<int>>
// r will be the resultant dunordered_map<int,vector<int>> containing only
// the values for the keys with even numbers in "m".
auto r = m.filter(is_even);
```

### inplace_filter()

Like filter(), this function can also be used to filter out some specific values from a dunordered_map. But in this case the filtration happens in-place, i.e., instead of returning a new dunordered_map, this function aims to update the source dunordered_map by keeping only the filtered out values in it.

Like filter(), it also accepts a function or a function object specifying the condition on which the value is to be filtered out from the dunordered_map. The type of the function arguments must be same or compatible with the key-value type of the dunordered_map and the function must return a boolean value (true/false).

```
dunordered_map<K,V>& inplace_filter(const F& f);
dunordered_map<K,V>& inplace_filter(bool(*f)(KK,VV));
```

On success, the source dunordered_map will be updated with only the filtered out values in-place and this function will return a reference to the updated dunordered_map.

For example,

```
bool is_even(int k, std::vector<int>& v) { return k%2 == 0; }

// let's consider "m" is a dunordered_map<int,vector<int>> containing both
// even and odd keys. it will contain only the values associated with even
// keys after the below in-place filtration.
m.inplace_filter(is_even);
```

### clear()

In order to remove the existing elements and clear the memory space occupied by a dunordered_map, clear() function can be used. It returns void.

### size()

This function returns the size of the distributed unordered_map, i.e., the number of unique keys present in the source dunordered_map as "size_t" parameter.

For example,

```
std::vector<std::pair<int,int>> v1;
v1.push_back(make_pair(1,100));
v1.push_back(make_pair(2,200));
v1.push_back(make_pair(3,300));
v1.push_back(make_pair(4,400));

std::vector<std::pair<int,int>> v2;
v2.push_back(make_pair(1,100));
```

```
v2.push_back(make_pair(2,200));
v2.push_back(make_pair(1,300));
v2.push_back(make_pair(2,400));

std::cout << make_dvector_scatter(v1).group_by_key<int,int>.size(); // -> 4
std::cout << make_dvector_scatter(v2).group_by_key<int,int>.size(); // -> 2
```

**put()**

This function can be used to modify a value associated with an existing key or insert a value with a new key in the source dunordered_map. It has the below
signature:

```
void put(const K& key, const V& val);
```

It allows user to perform simple map assignment like operation “m[key] = val”, where “m” is a distributed unordered_map. But such an operation should not be performed within a loop in order to avoid poor loop performance.

Here “key” can be either ‘an existing key’ or ‘a new key’ and “val” is the intended value ‘to be modified with’ or ‘to be inserted in’ the map. Types of the given key and value must be same or compatible with the key-value types of the source dunordered_map.

For example, if “m” is a `dunordered_map<int,int>`, then “m.put(2,5)” will either modify the value associated with key “2” as “5” or insert a new key “2” with associated value “5”.

**get()**

This function can be used to get the value associated with a given key in the source dunordered_map.

On success, if the given key exists, it returns the associated value of type “V”. But if the key does not exist, it returns the default value of type “V” (i.e., V()). It has the below signature:

```
V get(const K& key);
```

It is equivalent to an indexing operation “m[key]”, performed on a distributed unordered_map, “m”. But such an operation should not be used within a loop in order to avoid poor loop performance.

For example, if “m” is a `dunordered_map<int,int>` and its associated value with key “2” is “5” and there is no entry for the key “3”, then

```
auto r = m.get(2); // "r" will contain 5
auto x = m.get(3); // "x" will contain 0 (considering default integer value)
```

But it might happen that for a key “4” the associated value itself is “0”. Then,

```
 // y will contain 0, but it would be unknown whether the key "4" exists.
auto y = m.get(4);
```

In that case, a special interface of “get()” with the below signature is provided:

```
V get(const K& key, bool& found);
```

The second boolean parameter will be reflected (passed-by-reference) based on the existence of the given key. For example,

```
bool flag = false;
auto y = m.get(4,flag); // y: 0, flag: true  -> key "4" exists with value "0"
auto z = m.get(3,flag); // z: 0, flag: false -> key "3" does not exist
```

**as_dvector()**

A dunordered_map can be considered as the distributed version of the std::unordered_map containing the key-value pairs. Now in order to convert a `dunordered_map<K,V>` to a `dvector<std::pair<K,V>>`, member function as_dvector() can be used on the source dunordered_map. The source dunordered_map will remain unchnaged after the dvector conversion. The signature of the function is as follows:

```
dvector<std::pair<K,V>> as_dvector();
```

For example, if there is a `dunordered_map<int,int>` "m" containing the below elements:

```
1: 100
2: 200
3: 300
4: 400
```

Then,

```
auto dv = m.as_dvector(); // dv: dvector<std::pair<int,int>> (copy)
auto v = dv.gather(); // v: vector<int> -> {(1,100),(2,200),(3,300),(4,400)}
```

Note that, there is no gather() method provided on a dunordered_map. When gathering of the data will be required, it needs to be converted to a dvector object first and then gather() on the converted dvector object can be called.

**as_node_local()**

This function can be used to convert a `dunordered_map<K,V>` to a `node_local<MAP<K,V>>`, where MAP can be either a 'std::map' or a 'std::unordered_map' depending upon the user configuraton (USE_ORDERED_MAP macro is defined or not) in config.hpp file. While converting to the node_local (see manual entry for node_local) object it copies the entire elements of the source dunordered_map. Thus after the conversion, source dunordered_map will remain unchanged. The signature of the function is as follows:

```
node_local<MAP<K,V>> as_node_local();
```

**moveto_node_local()**

This function can be used to convert a `dunordered_map<K,V>` to a `node_local<MAP<K,V>>`, where MAP can be either a 'std::map' or a 'std::unordered_map' depending upon the user configuraton (USE_ORDERED_MAP macro is defined or not) in config.hpp file. While converting to the node_local object, it avoids copying the data. Thus the source dunordered_map will become invalid after the conversion. This is faster and recommended to use when source dunordered_map will no longer be used in a user program. The signature of the function is as follows:

```
node_local<MAP<K,V>> moveto_node_local();
```

#### viewas_node_local()

This function can be used to create a view of a `dunordered_map<K,V>` as a `node_local<MAP<K,V>>`, where MAP can be either a 'std::map' or a 'std::unordered_map' depending upon the user configuraton (USE_ORDERED_MAP macro is defined or not) in config.hpp file. Since it is about just creation of a view, the data in source dunordered_map is neither copied nor moved. Thus it will remain unchanged after the view creation and any changes made in the source dunordered_map will be reflected in its node_local view as well and the reverse is also true. The signature of the function is as follows:

```
node_local<MAP<K,V>> viewas_node_local();
```

### Public Global Function Documentation

#### `dunordered_map<K,V>` make_dunordered_map_allocate()

**Purpose**
This function is used to allocate empty unordered_map instances with key-type "K" and value-type "V" at the worker nodes to create a valid empty `dunordered_map<K,V>` at master node.

The default constructor of dunordered_map, does not allocate any memory at the worker nodes. Whereas, this function can be used to create a valid empty dunordered_map with allocated zero-sized map memory at worker nodes.

Note that, the intended key-value types needs to be explicitly mentioned while calling this function.

For example,

```
dunordered_map<int,int> m1; // empty dunordered_map without any allocated memory

// empty dunordered_map with allocated memory
auto m2 = make_dunordered_map_allocate<int,int>();

m1.put(1,5); // error, can't insert key-value pair in map (it is not valid)
m2.put(1,5); // Ok, a key "1" with associated value "5" will be inserted
```

**Return Value**
On success, it returns the allocated `dunordered_map<K,V>`.

## SEE ALSO

dvector, node_local