In-memory key-value store

User Manual & Reference Manual 2017

Index64 Embedded 6.0

www.index64.com

Content

1. User's manual

1.1. Overview sample

To use Index64 concurrent key-value store, add files to your source code and link your application with a shared libraries. Then you can instantiate any class of Index64. Depending on your Index64 version and on your OS, here are the files to add and the shared libraries to link:

Index64 / OS	File names
Include	#include "Index64.h"
Add files	NA
Redhat 64 bits	libindex64_rr.so
Ubuntu 64 bits	libindex64_ur.so
Windows 64 bits	Index64Wr.lib

The overview sample indexes null-terminated strings.

- It declares a global instance of I64MtString.
- It inserts key-value pairs into the index.
- It selects from the index, the values associated with the given keys.
- It deletes from the index, some key-value pairs.

Output of SampleOverview.cpp

```
>SampleOverviewW64d.exe

<KEYS       :VALUES       >
<key zero   :value zero   >
<key one    :value one    >
<key two    :value two    >
<key three  :value three  >
<key four   :value four   >
<key five   :value five   >

>_
```

SampleOverview.cpp 1 / 2

```
// --- BEGIN ----------------------------------------------------------------

// --------------------------------------------------------------------------
// Index64 concurrent key-value store
//
// SampleOverview.cpp
// Illustrate how to index null-terminated strings
// --------------------------------------------------------------------------

// --------------------------------------------------------------------------
#if defined(__linux__)
        #include <stddef.h>
        #include <stdio.h>
        #include <string.h>
        #define  MAXUINT16         ((unsigned short)~0)
        #define __int8  char
        #define __int16 short
        #define __int32 int
        #define __int64 long long
#else // windows
        #include <stddef.h>
        #include <stdio.h>
        #include <string.h>
        #define  MAXUINT16         ((unsigned short)~0)
#endif
// --------------------------------------------------------------------------

// --------------------------------------------------------------------------
// Index64 concurrent key-value store
#include "Index64.h"
// --------------------------------------------------------------------------

// --------------------------------------------------------------------------
// Index 64 object to index null-terminated strings with long values
I64MtString idx;
// --------------------------------------------------------------------------

// --------------------------------------------------------------------------
// Pairs of null-terminated strings
struct sData { char key[10]; char value[12]; } data[6] = { // 10 is the maximum key length in char
including trailing 0, 12 is the maximum value length in char including trailing 0
        { "key zero" , "value zero",  },
        { "key one"  , "value one",   },
        { "key two"  , "value two",   },
        { "key three", "value three", },
        { "key four" , "value four",  },
        { "key five" , "value five",  },
};
// --------------------------------------------------------------------------

// --------------------------------------------------------------------------
```

SampleOverview.cpp 2 / 2

```
// ------------------------------------------------------------------------

// ------------------------------------------------------------------------
int main(int, char**, char**) {
        // 10 is the maximum key length in char including trailing 0, 12 is the maximum value length
in char including trailing 0
        idx.setMaxKeyLen(10);
        idx.setMaxValLen(12);

        //
        // Index all null-terminated strings
        for(int i = 0; i < 6; i++)
                idx.insert(data[i].key, data[i].value, (tValSize)strlen(data[i].value));
        //
        // Retrieve all values
        printf("\n(KEYS      :VALUES      )\n");
        for(int i = 0; i < 6; i++) {
                char value[12] = "";
                if (idx.select(data[i].key, value) == eReturnCodeOk)
                        printf("(%-10s:%-12s)\n", data[i].key, value);
                else
                        printf("(%-10s:%12s)\n",  data[i].key, "no value");
        }
        printf("\n");
        //
        // Remove some null-terminated strings from the index
        idx.del(data[0].key);
        idx.del(data[2].key);
        idx.del(data[4].key);
}
// ------------------------------------------------------------------------

// --- END ----------------------------------------------------------------
```

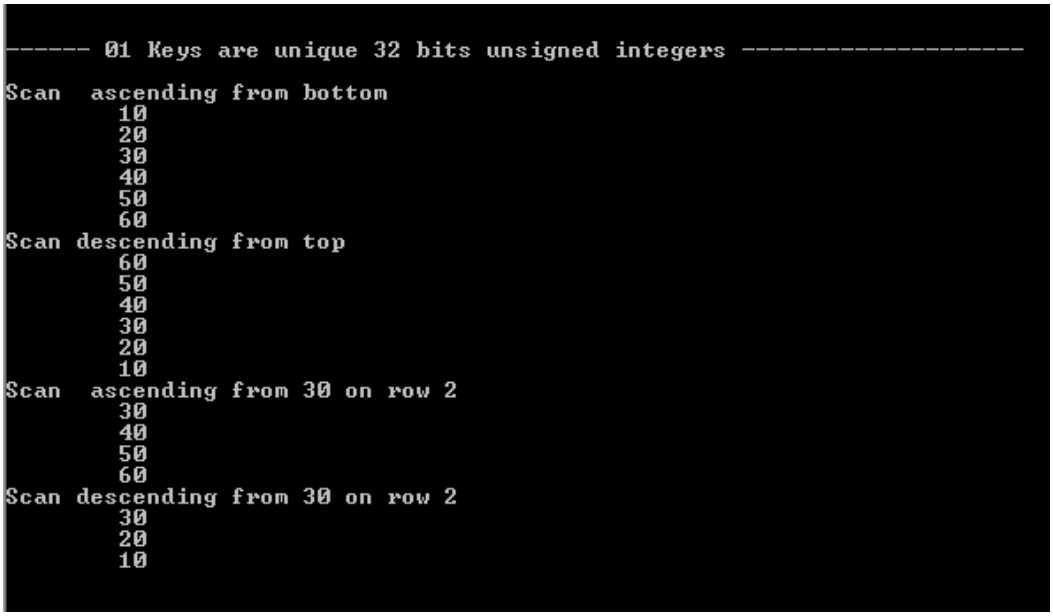
1.2. Full sample

The class used to declare an index depends on:

- The type of keys to index: signed or unsigned integer, float or double, character string or wide character string, compound data or variable length array of bytes.
- The value size: 8 bytes or long value.
- The uniqueness of keys.
- The ability of class to support a distributed or an embedded table. The API is the same in both cases.

See "§ 2.1 Index64 classes" for a table of all the classes available.

Output of Sample01_Unsigned32.cpp



For all the classes of Index64, the full sample illustrates calls to most of the functions of the API.

- It declares a local instance of a class of Index64.
- It inserts some key-value pairs into the index.
- It does a full scan ascending, then a full scan descending – see output above -.
- It updates all values of the index.
- It does a range scan ascending, then a range scan descending – see output above -.
- It deletes from the index, all key-value pairs.

Sample01_Unsigned32.cpp 1 / 2

```
// --- BEGIN ----------------------------------------------------------------

// --------------------------------------------------------------------------
// Index64 concurrent key-value store
//
// Sample_Class01_Unsigned32.cpp
// Illustrate how to index 32 bits unsigned integers
// --------------------------------------------------------------------------

// --------------------------------------------------------------------------
#if defined(__linux__)
        #include <stdio.h>
        #define  MAXUINT16         ((unsigned short)~0)
        #define __int8  char
        #define __int16 short
        #define __int32 int
        #define __int64 long long
#else // windows
        #include <stdio.h>
        #define  MAXUINT16         ((unsigned short)~0)
#endif
// --------------------------------------------------------------------------

// --------------------------------------------------------------------------
// Index64 concurrent key-value store
#include "Index64.h"
// --------------------------------------------------------------------------

// --------------------------------------------------------------------------
// Key size
#define dKeySize 4
// 32 bits unsigned integer
typedef unsigned int tKey;
//
// Value size
#define dValSize sizeof(struct sKeyUnsigned32Va)
// Table of unique 32 bits unsigned integers & values
const struct sKeyUnsigned32Va { tKey key; char text[12]; } keysUnsigned32Va[6] = {
        { 10, "ten",    },
        { 20, "twenty", },
        { 30, "thirty", },
        { 60, "sixty",  },
        { 50, "fifty",  },
        { 40, "forty",  },
};
// --------------------------------------------------------------------------

// --------------------------------------------------------------------------
```

```
// ---------------------------------------------------------------------------

// ---------------------------------------------------------------------------
void TestUnsigned32() {
        // Index 64 object to index unique 32 bits unsigned integers with long values
        I64MtUnsigned32 index;
        // Size of 32 bits unsigned integers is 4 bytes
        // Dont need to call index.setMaxKeyLen(4);
        // Set value size in bytes
        index.setMaxValLen(dValSize);
        //
        // Working variables
        tKey key; struct sKeyUnsigned32Va value; tInteger i;
        //
        // (...)
        //
        // Index keys and values
        for(i = 0; i < 6; i++)
                index.insert(keysUnsigned32Va[i].key, (tpValue)&keysUnsigned32Va[i], dValSize);
        //
        // Scan ascending all the keys
        printf("Scan  ascending from bottom\n");
        if (index.scanAscMin(key, (tpValue)&value) == eReturnCodeOk) do {
                printf("\t%u   %s\n", key, value.text);
        } while(index.scanAscNext(key, key, (tpValue)&value) == eReturnCodeOk);
        // Scan descending all the keys
        printf("Scan descending from top\n");
        if (index.scanDescMax(key, (tpValue)&value) == eReturnCodeOk) do {
                printf("\t%u   %s\n", key, value.text);
        } while(index.scanDescNext(key, key, (tpValue)&value) == eReturnCodeOk);
        //
        // Update values
        for(i = 0; i < 6; i++)
                // Old value parameter is optional with unique index as it is used for returning old
value
                index.update(keysUnsigned32Va[i].key, (tpValue)&keysUnsigned32Va[i], dValSize);
        //
        // Initialize a starting key
        int start = 2; const tKey& startKey = keysUnsigned32Va[start].key;
        // Scan ascending from the starting key
        printf("Scan  ascending from %u on row %d\n", startKey, start);
        if (index.scanAscFirst(startKey, key, (tpValue)&value) == eReturnCodeOk) do {
                printf("\t%u   %s\n", key, value.text);
        } while(index.scanAscNext(key, key, (tpValue)&value) == eReturnCodeOk);
        // Scan descending from the starting key
        printf("Scan descending from %u on row %d\n", startKey, start);
        if (index.scanDescFirst(startKey, key, (tpValue)&value) == eReturnCodeOk) do {
                printf("\t%u   %s\n", key, value.text);
        } while(index.scanDescNext(key, key, (tpValue)&value) == eReturnCodeOk);
        //
        // Delete keys and values
        for(i = 0; i < 6; i++)
                // Old value parameter is optional with unique index as it is used for returning old
value
                index.del(keysUnsigned32Va[i].key);
}
// ---------------------------------------------------------------------------

// --- END -----------------------------------------------------------------
```

1.3. Advanced sample

The advanced sample implement concurrent accesses and the CAS function.

To illustrate the ability of class to supports concurrent accesses in reading and writing, four threads are involved:

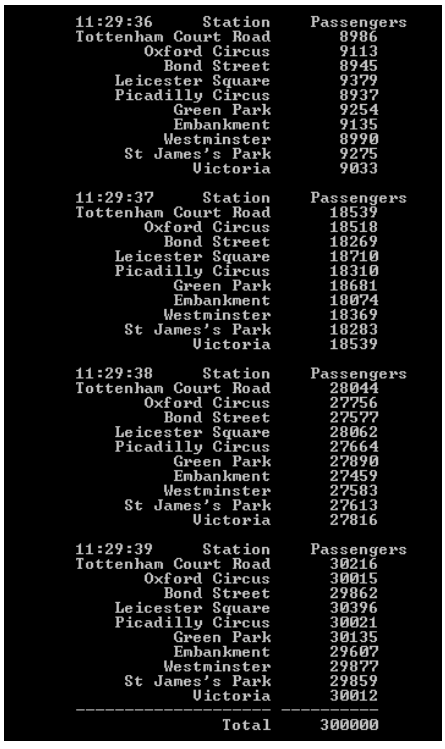
- threads 1, 2, 3 continuously update the values
- thread 4 read the values to reflect the changes

The keys represent stations in a transportation network. The values associated with each key represent the number of persons coming in the stations.

The CAS function is designed to preserve data integrity with ongoing concurrent update on the same value. CAS stands for compare and swap. To use CAS first make a local copy of the value to be modified. With the local copy evaluate the new value. Call CAS with both the local copy and the new value. The result is one of:

- Ok the update is done.
- Inaccurate old value the present value associated with the key is returned in the local copy. With the up-to-date local copy, evaluate the new value and call CAS again.

Output of SampleAdvanced.cpp



```
11:29:36     Station    Passengers
Tottenham Court Road        8986
       Oxford Circus        9113
         Bond Street        8945
    Leicester Square        9379
    Picadilly Circus        8937
          Green Park        9254
          Embankment        9135
         Westminster        8990
      St James's Park        9275
            Victoria        9033

11:29:37     Station    Passengers
Tottenham Court Road       18539
       Oxford Circus       18518
         Bond Street       18269
    Leicester Square       18710
    Picadilly Circus       18310
          Green Park       18681
          Embankment       18074
         Westminster       18369
      St James's Park       18283
            Victoria       18539

11:29:38     Station    Passengers
Tottenham Court Road       28044
       Oxford Circus       27756
         Bond Street       27577
    Leicester Square       28062
    Picadilly Circus       27664
          Green Park       27890
          Embankment       27459
         Westminster       27583
      St James's Park       27613
            Victoria       27816

11:29:39     Station    Passengers
Tottenham Court Road       30216
       Oxford Circus       30015
         Bond Street       29862
    Leicester Square       30396
    Picadilly Circus       30021
          Green Park       30135
          Embankment       29607
         Westminster       29877
      St James's Park       29859
            Victoria       30012
-------------------- ----------
               Total      300000
```

SampleAdvanced.cpp 1 / 4

```
// --- BEGIN ----------------------------------------------------------------

// --------------------------------------------------------------------------
// Index64 concurrent key-value store
// SampleAdvanced.cpp
// Illustrate distributed table and concurrent threads
// --------------------------------------------------------------------------

// --------------------------------------------------------------------------
#if defined(__linux__)
        #include <pthread.h>
        #include <stdio.h>
        #include <stdlib.h>
        #include <time.h>
        #include <unistd.h>
        #define  MAXUINT16         ((unsigned short)~0)
        #define __int8  char
        #define __int16 short
        #define __int32 int
        #define __int64 long long
        #define  dSleep(x)        sleep(x)
        #define  tThread                                    pthread_t
        #define  dThreadBegin(Handle_, StartAddress_, Arg_) \
pthread_create(&Handle_, &Att, (void*(*)(void*))StartAddress_, Arg_)
        #define  dThreadEnd(RetVal_)                        pthread_exit((void*)RetVal_)
        #define  dThreadWait(Handle_)                       pthread_join(Handle_, 0)
#else // windows
        #define  WIN32_LEAN_AND_MEAN
        #define  __attribute__(n)
        #include <stdio.h>
        #include <stdlib.h>
        #include <process.h>
        #include <time.h>
        #include <windows.h>
        #define  dSleep(x)        Sleep(1000*x)
        #define  tThread                                    void*
        #define  dThreadBegin(Handle_, StartAddress_, Arg_) \
Handle_ = (void*)_beginthreadex(0, 0x3FF, (unsigned int(__stdcall*)(void*))StartAddress_, Arg_, 0, 0)
        #define  dThreadEnd(RetVal_)                        _endthreadex(RetVal_)
        #define  dThreadWait(Handle_)                       WaitForSingleObject(Handle_, INFINITE)
#endif
// --------------------------------------------------------------------------

// --------------------------------------------------------------------------
// Index64 concurrent key-value store
#include "Index64.h"
// --------------------------------------------------------------------------

// --------------------------------------------------------------------------
// Number of passengers by station: a key is a station, a value is a number of passengers
I64MtUnsigned32 ActualPassengers;
// A value is a number of passengers and a number of passengers is an unsigned __int64
#define dValSize sizeof(unsigned __int64)
// --------------------------------------------------------------------------

// --------------------------------------------------------------------------
// Stations
const char* StationsNames[10] = {
        "Tottenham Court Road",
        "Oxford Circus",
        "Bond Street",
        "Leicester Square",
        "Picadilly Circus",
        "Green Park",
        "Embankment",
        "Westminster",
        "St James's Park",
        "Victoria",
};
```

```
// ----------------------------------------------------------------------------

// ----------------------------------------------------------------------------// Report the situation
as long as there is a writer thread running
bool Writer1isRunning = 0;
bool Writer2isRunning = 0;
bool Writer3isRunning = 0;
// ----------------------------------------------------------------------------

// ----------------------------------------------------------------------------
// Report the situation
void ReportActualPassengers(void*) {
        // Total number of passengers
        tInteger TotalPassengers = 0;
        //
        // Report the situation
        for(;;) {
                //
                // Let happen some changes
                dSleep(1);
                //
                // Format the timestamp
                time_t Now; time(&Now); struct tm Today = *localtime(&Now);
                char FormatedTime[12]; strftime(FormatedTime, 128, "%H:%M:%S", &Today);
                printf("\n\t%s%12s    %10s\n", FormatedTime, "Station", "Passengers");
                //
                // List all stations
                TotalPassengers = 0; for(unsigned __int32 Station = 0; Station < 10; Station++) {
                        //
                        // Retrieve the number of passengers that came at the station
                        unsigned __int64 Passengers;
                        ActualPassengers.select(Station, (tpValue)&Passengers);
                        //
                        // Display station's name and actual passenger number
                        printf("\t%20s %10Lu\n", StationsNames[Station], Passengers);
                        //
                        // Total number of passengers
                        TotalPassengers += Passengers;
                }
                //
                // Report the situation as long as there is a writer thread running
                if (Writer1isRunning || Writer2isRunning || Writer3isRunning) { } else break;
        }
        //
        // Display the total number of passengers
        printf("\t%20s %10s\n", "--------------------", "----------");
        printf("\t%20s %10Lu\n\n", "Total", TotalPassengers);
        //
        // End the worker thread
        dThreadEnd(0);
}
// ----------------------------------------------------------------------------

// ----------------------------------------------------------------------------
```

```
// ----------------------------------------------------------------------------

// ----------------------------------------------------------------------------
// Simulate incoming passengers
void SimulateIncomingPassengers(bool* WriterIsRunning) {
        // Local copy of ActualPassengers values
        unsigned __int64 localActualPassengers[10]; for(unsigned __int32 Station = 0; Station < 10;
Station++) localActualPassengers[Station] = 0;
        //
        // A value is a number of passengers and a number of passengers is an unsigned __int64
        tValSize valSize = dValSize;
        //
        // Simulate incoming passengers
        for(unsigned __int32 Repeat = 0; Repeat < 10000000; Repeat++) {
                //
                // Chose a random station
                unsigned __int32 Station = rand() % 10;
                //
                // Set the new number of passengers
                unsigned __int64 newActualPassengers = localActualPassengers[Station] + 1;
                //
                // CAS succeed when localActualPassengers[Station] is correct
                while(ActualPassengers.CAS(
                        Station,
                        (tpValue)&newActualPassengers, dValSize,
                        (tpValue)&localActualPassengers[Station], &valSize
                ) == eReturnCodeInaccurateOldValue) {
                        // If localActualPassengers[Station] is incorrect
                        // CAS return eReturnCodeInaccurateOldValue
                        // and set localActualPassengers[Station] to the correct value
                        newActualPassengers = localActualPassengers[Station] + 1;
                }
                //
                // When CAS succeed, update the local copy of ActualPassengers
                localActualPassengers[Station] = newActualPassengers;
        }
        //
        // End the worker thread
        *WriterIsRunning = false; dThreadEnd(0);
}
```

SampleAdvanced.cpp 4 / 4

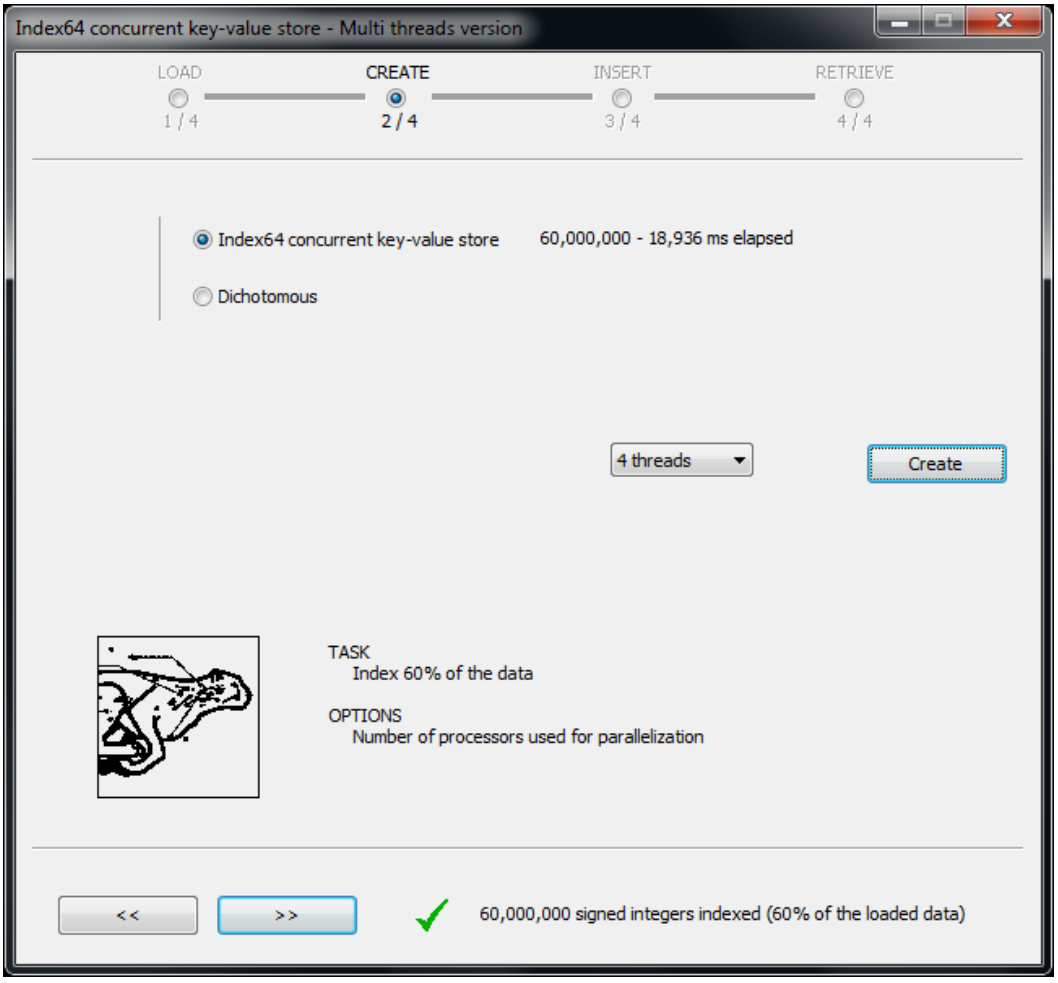
```
// ------------------------------------------------------------------------

// ------------------------------------------------------------------------
// Smaller is the number of stations, higher is the probability of collision of two different threads
int main(int, char**, char**) {
        // Linux only
        #if defined(__linux__)
        pthread_attr_t Att;
        pthread_attr_init(&Att);
        pthread_attr_setdetachstate(&Att, PTHREAD_CREATE_JOINABLE);
        pthread_attr_setstacksize(&Att, 0x3FF);
        #endif
        //
        // Reset the random serie
        srand((unsigned int)time(0)); // Use always a new seed to get always a new serie
//      srand(3391453417);            // Use always the same seed to get always the same serie
        //
        { // Initialize the application
                // A value is a number of passengers and a number of passengers is an unsigned __int64
                ActualPassengers.setMaxValLen(dValSize);
                // Set the initial number of passengers
                unsigned __int64 initialActualPassengers = 0;
                // Initialize ActualPassengers:
                // keys are station number, values are number of passengers
                for(unsigned __int32 Station = 0; Station < 10; Station++)
                   ActualPassengers.insert(Station, (tpValue)&initialActualPassengers, dValSize);
        }
        //
        // Start the worker threads
        tThread writer1; dThreadBegin(writer1, SimulateIncomingPassengers, &Writer1isRunning);
        Writer1isRunning = true;
        tThread writer2; dThreadBegin(writer2, SimulateIncomingPassengers, &Writer2isRunning);
        Writer2isRunning = true;
        tThread writer3; dThreadBegin(writer3, SimulateIncomingPassengers, &Writer3isRunning);
        Writer3isRunning = true;
        tThread reader1; dThreadBegin(reader1, ReportActualPassengers, 0);
        //
        // Let run the worker threads
        dThreadWait(writer1);
        dThreadWait(writer2);
        dThreadWait(writer3);
        dThreadWait(reader1);
}
// ------------------------------------------------------------------------

// --- END ----------------------------------------------------------------
```

1.4. Compilation, Link and Run

Tool	Option	Description	Example
Compiler	Include directories	Path to: "index64.h"	-I../../../index64/include /I"..\..\..\index64\include"
Linker	Library directories	Path to: "libindex64_rr.so"	-L../../../index64/lib /LIBPATH:"..\..\..\index64\link"
	Library dependencies	Name of the library file	-lindex64_rr "Index64Wr.lib"
System	Path variable	Path to: "libindex64_rr.so"	cp libindex64_rr.so /usr/bin copy Index64Wr.dll %windir%\system32\

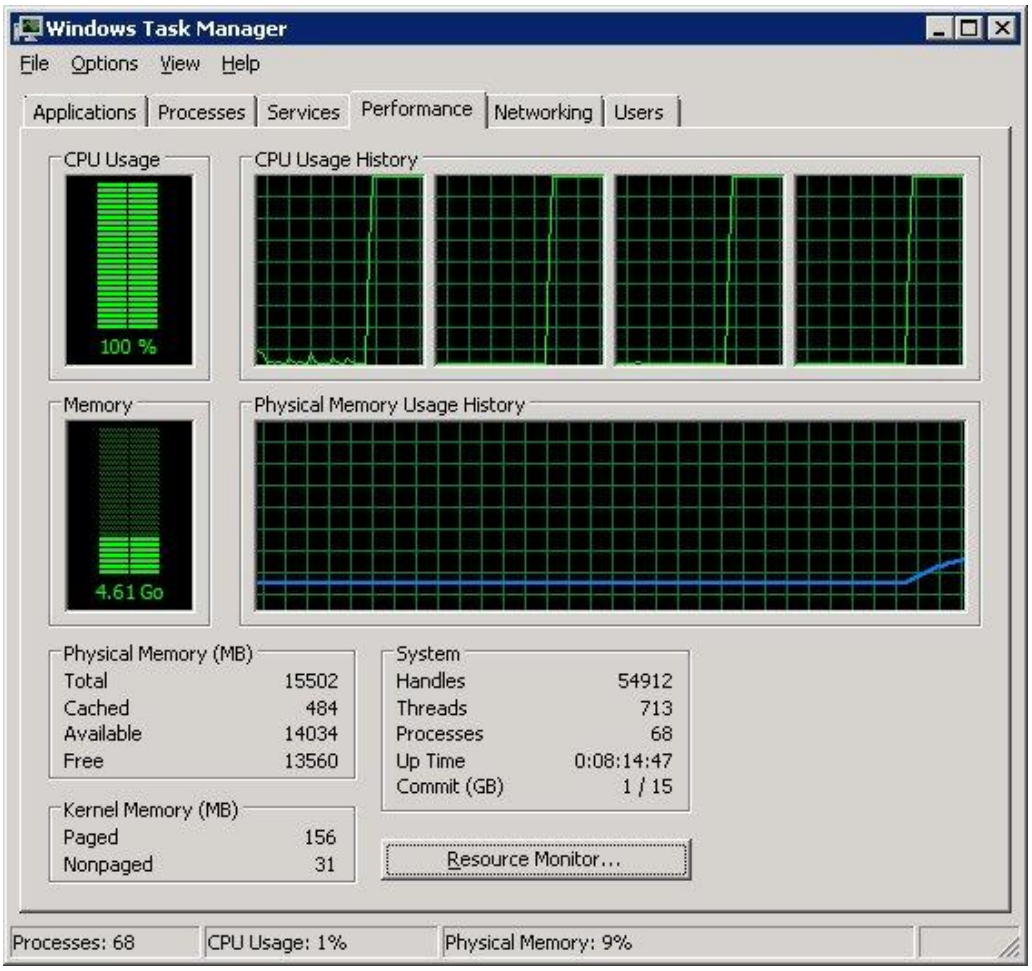
1.5. Graphic demo

Within the "Stable release" package, a demo program is available for these three 64 bits platforms: redhat, ubuntu and windows. The demo program contains Index64 embedded version. To observe the operation of threads, under windows launch the demo program and the task manager together, under linux launch the demo program and "top".



The above screenshot shows an example of running the demo program on a machine with 4 cores. To maximize the CPU usage, the data processing is launched with 4 threads.

The following screenshot shows the task manager while the demo program is running.



As Index64 uses a lock-free algorithm, the CPU usage reaches 100% with 4 threads running in parallel inside a 4 cores CPU.

2. Reference manual

2.1. Index64 classes

Each Index64 class may instantiate an index object depending on the data type. Prefix I64Cl stands for client classes of multi nodes version. Prefix I64Mt stands multi thread classes of embedded version respectively.

The API is homogeneous among all Index64 classes.

Data type	Index64 class
unsigned integer 32 bits	I64MtUnsigned32
signed integer 32 bits	I64MtSigned32
Float	I64MtFloat
unsigned integer 64 bits	I64MtUnsigned64
signed integer 64 bits	I64MtSigned64
Double	I64MtDouble
null-terminated string	I64MtString
null-terminated wide character string	I64MtWideChar

There are three Index64 classes which accept concatenated data of any kind.

Data type	Index64 class
Compound data or multi columns	I64MtMulti

2.2. Anti-fragmentation mechanism

Index64 classes automatically allocate blocks of memory to store key-value pairs. To avoid fragmentation, these blocks are always of the same size: 50 M. When blocks of memory are released, they are kept in a list of free memory blocks available for future allocations. This list supports concurrent accesses.

When you know in advance the number of key value pairs that will be stored, you can get an estimation of the needed amount of memory with a call to the estimateMemSize() function.

For the embedded version, with a call to the function setMinQuota1(), you can trigger allocation of memory for a key value store before inserting any K-V pairs. This allocation will still be made with blocks of 50 M.

2.3. Query and manipulation functions

The query functions allow to:

- select a value associated with a key,
- scan keys and values.

See "§ 1.2 Full sample" for a demonstration of all the scan functions for all the Index64 classes.

The manipulation functions allow to:

- insert or delete key-value pairs,
- update values.

The manipulation function upsert allows to:

- insert a key-value pair, if the key is not already inside the index,
- update a value, if the key is already inside the index.

```
enum eReturnCode scanAscMin(
        OUT unsigned __int32& firstKey,
        OUT tpValue pFirstValue = 0, OUT tValSize* pFirstValLen = 0);
```

Return the minimum key-value pair. Enough space must be allocated to pFirstValue.

```
enum eReturnCode scanAscFirst(
        IN unsigned __int32 fromKey, OUT unsigned __int32& firstKey,
        OUT tpValue pFirstValue = 0, OUT tValSize* pFirstValLen = 0);
```

Return the lowest key-value pair above or equal to the input key.
(fromKey <= firstKey)
Enough space must be allocated to pFirstValue.

```
enum eReturnCode scanAscNext(
        IN unsigned __int32 fromKey, OUT unsigned __int32& nextKey,
        OUT tpValue pNextValue = 0, OUT tValSize* pNextValLen = 0);
```

Return the lowest key-value pair above and not equal to the input key.
(fromKey < nextKey)
There is no need to call scanAscFirst before calling scanAscNext. Enough space must be allocated to pNextValue.

enum eReturnCode **scanDescMax**(
OUT unsigned __int32& firstKey,
OUT tpValue pFirstValue = 0, OUT tValSize* pFirstValLen = 0);

Return the maximum key-value pair. Enough space must be allocated to pFirstValue.

enum eReturnCode **scanDescFirst**(
IN unsigned __int32 fromKey, OUT unsigned __int32& firstKey,
OUT tpValue pFirstValue = 0, OUT tValSize* pFirstValLen = 0);

Return the highest key-value pair below or equal to the input key.
(firstKey <=fromKey)
Enough space must be allocated to pFirstValue.

enum eReturnCode **scanDescNext**(
IN unsigned __int32 fromKey, OUT unsigned __int32& nextKey,
OUT tpValue pNextValue = 0, OUT tValSize* pNextValLen = 0);

Return the highest key-value pair below and not equal to the input key.
(firstKey < fromKey)
There is no need to call scanDescFirst before calling scanDescNext. Enough space must be allocated to pNextValue.

enum eReturnCode **select**(
IN unsigned __int32 key,
OUT tpValue pValue = 0, OUT tValSize* pValLen = 0);

Return the value associated with the key. Enough space must be allocated to pValue.

enum eReturnCode **insert**(
IN unsigned __int32 key, IN tpValue pNewValue, IN tValSize newValLen);

Insert a key-value pair into index. Only one value can be associated with any key.

enum eReturnCode **CAS**(
IN unsigned __int32 key, IN tpValue pNewValue, IN tValSize newValLen,
INOUT tpValue pOldValue, INOUT tValSize* pOldValLen);

Update the value associated with the key if pOldValue points to the previous value that was associated with the key. Otherwise enough space must be allocated to pOldValue and the previous value that was associated with the key is returned.

enum eReturnCode **update**(
IN unsigned __int32 key, IN tpValue pNewValue, IN tValSize newValLen,
OUT tpValue pOldValue = 0, OUT tValSize* pOldValLen = 0);

Update the value associated with the key. If pOldValue is set, enough space must be allocated to pOldValue and the previous value that was associated with the key is returned.

enum eReturnCode **upsert**(
IN unsigned __int32 key, IN tpValue pNewValue, IN tValSize newValLen,
OUT tpValue pOldValue = 0, OUT tValSize* pOldValLen = 0);

If the key is not already inside the index, this function inserts the key-value pair into index. pOldValue is ignored. If the key is already inside the index, this function update the value associated with the key. If pOldValue is set, enough space must be allocated to pOldValue and the previous value that was associated with the key is returned.

enum eReturnCode **del**(
IN unsigned __int32 key,
OUT tpValue pOldValue = 0, OUT tValSize* pOldValLen = 0);

Delete the key-value pair from the index. If pOldValue is set, enough space must be allocated to pOldValue and the value that was associated with the key is returned.

2.4. Bulk functions

These functions are a convenient facility provided to take advantage of multi or many cores processors.

Before calling a bulk function, set up a pool of thread via a call to bulkAllocateThreads(). The number of threads in the pool can be increased at a later time by another call to bulkAllocateThreads() with the new total number of threads needed in the pool. The pool is maintained alive for subsequent uses by another bulk function of the same index. It is not shared among index. It is released with a call to bulkReleaseThreads() or by the index destructor. The pool of thread is required to call bulkInsert() or bulkDelete(), the pool is not required to make concurrent calls of query and manipulation functions.

Parameters of the bulk functions are:

- firstThread the number of the first thread to run among allocated threads.
- nbThreads the number of threads to run.
- nbKeys the number of key-value pairs to insert or to delete.
- pKey pointer to the first key.
- sizeToNextKey number of bytes between two keys.

Optional parameters of the bulk functions are:

- pNewValue/pOldValue pointer to the first value to insert/delete.
- sizeToNextValue number of bytes between two values.
- pNewValLen/pOldValLen pointer to the length of the first value to insert/delete.
- sizeToNextValLen number of bytes between two lengths.
- pResult pointer to the first result.
- sizeToNextResult number of bytes between two results.

Maximum number of threads accepted is 65,535 however for best performance do not exceed hardware capacity. See "§ 1.3 Advanced sample" for a demonstration of all the bulk functions.

enum eReturnCode **bulkInsert**(
IN tNbThreads firstThread, IN tNbThreads nbThreads, IN tInteger nbKeys,
IN const unsigned __int32* pKey, IN tDeltaSize sizeToNextKey,
IN const char* pNewValue = 0, IN tDeltaSize sizeToNextValue = 0,
IN const tValSize* pNewValLen = 0, IN tDeltaSize sizeToNextValLen = 0,
OUT tResult* pResult = 0, IN tDeltaSize sizeToNextResult = 0);

Insert key-value pairs into the index. If pOldValue is not set, key-&key pairs are used instead.

```
// Result is equivalent to
for(int i = 0; i < nbKeys; i++) {
   *(tResult*)((char*)pResult + i*sizeToNextResult) = I.insert(
      *(unsigned __int32*)((char*)pKey + i*sizeToNextKey),
      *(tValue*)((char*)pNewValue + i*sizeToNextValue),
      *(tValSize*)((tValSize*)pNewValLen + i*sizeToNextValLen));
}
```

enum eReturnCode **bulkDelete**(
IN tNbThreads firstThread, IN tNbThreads nbThreads, IN tInteger nbKeys,
IN const unsigned __int32* pKey, IN tDeltaSize sizeToNextKey,
OUT char* pOldValue = 0, IN tDeltaSize sizeToNextValue = 0,
OUT tValSize* pOldValLen = 0, IN tDeltaSize sizeToNextValLen = 0
OUT tResult* pResult = 0, IN tDeltaSize sizeToNextResult = 0);

Delete key-value pairs from the index. If pOldValue is set, enough space must be allocated to pOldValue and the values that were associated with the keys are returned.

```
// Result is equivalent to
for(int i = 0; i < nbKeys; i++) {
   *(tResult*)((char*)pResult + i*sizeToNextResult) = I.del(
      *(unsigned __int32*)((char*)pKey + i*sizeToNextKey),
      (tValue*)((char*)pOldValue + i*sizeToNextValue),
      (tValSize*)((char*)pOldValLen + i*sizeToNextValLen));
}
```

2.5. Other functions

These functions manage the index itself. They require exclusive access to the index.

Distinct categories are:

- Key length definition: for classes dealing with long keys,
- Value length definition: for classes dealing with long values,
- File management: to save and restore all the key-value pairs of an index,
- Threads and memory resources: to set pool of threads, to set optional memory quota, to clear the index,
- Static functions: to convert return code into message, to get version details and to estimate memory consumption.

2.5.1. Key length

For I64MtString, I64MtWideChar and I64MtMulti classes, set the maximum key length through a call to setMaxKeyLen().

enum eReturnCode **setMaxKeyLen**(IN tKeySize maxKeyLen);
Set maximum key length, only when the index is cleared.

enum eReturnCode **getMaxKeyLen**(OUT tKeySize& maxKeyLen);
Get maximum key length.

2.5.2. Value length

For I64MtUnsigned32, I64MtSigned32, I64MtFloat, I64MtUnsigned64, I64MtSigned64, I64MtDouble, I64MtString, I64MtWideChar and I64MtMulti classes, set the maximum value length through a call to setMaxValLen().

enum eReturnCode **setMaxValLen**(IN tValSize& maxValLen);
Set maximum value length, only when the index is cleared.

enum eReturnCode **getMaxValLen**(OUT tValSize& maxValLen);
Get maximum value length.

2.5.3. Load and save

enum eReturnCode **load**(IN const char* fileName);
Restore into the index all the key-value pairs previously saved in a file.

enum eReturnCode **save**(IN const char* fileName);
The key-value pairs of the index are saved in a file. The file name may begin with a relative or full pathname.

2.5.4. Threads and memory resources

enum eReturnCode **bulkAllocateThreads**(IN tNbThreads nbThreads);
Create a pool of threads or extend it to contain the requested number of threads. It is only useful for bulkInsert() and bulkDelete() since bulkInsert() and bulkDelete() need a pool of thread but won't create it or modify it.

enum eReturnCode **bulkReleaseThreads**();
Release the pool of thread if there is one. It is automatically called by the destructor.

enum eReturnCode **setMinQuota1**(IN tIndexSize minMemSize);
Memory to store key-value pair is allocated automatically. However you can control precisely the memory consumption by calling setMinQuota1(). This might be of importance for very big indexes. To cancel the mimimum quota, call setMinQuota1(0) or rely on the index destructor. When the minimum quota is full and if there is no maximum quota, the index allocates new memory blocks as needed.

enum eReturnCode **setMinQuota2**(IN void* buffer, IN tIndexSize bufferSize);
Memory to store key-value pair is allocated automatically. However you can force the index to use an existing memory space by calling setMinQuota2(). To cancel the mimimum quota, call setMinQuota2(0, 0). This function does not allocate memory. When the minimum quota is full and if there is no maximum quota, the index allocates new memory blocks as needed.

enum eReturnCode **setMaxQuota**(IN tIndexSize maxMemSize);
Prevent memory allocation to exceed a maximum amount. You can set both a minimum quota and a maximum quota. To cancel the maximum quota, call setMaxQuota(0).

enum eReturnCode **getAllocatedMemSize**(
    OUT tIndexSize* usedMemSize,
    OUT tIndexSize* leftMemSize);
Return the exact memory needed by this index and the total memory allocated minus the exact memory needed for this index.
totalMemSize = *usedMemSize + *leftMemSize;

enum eReturnCode **clear**();
Empty the index. The allocated memory is released except the minimum quota if there is one.

2.5.5. Static functions

static const char* **getI64ReturnText**(IN enum eReturnCode returnCode);
Give the message corresponding to a returnCode. See "§ 3 Return codes" for a complete description.

static const struct sVersion& g**etVersion**();
Provide a set of information about the Index64 concurrent key-value store itself: name and version of the library, Redhat/Ubuntu/Windows binary, 32/64 platform, Debug/Release, maximum number of threads. Contact the editor at index64.com, to increase the maximum number of threads of your licence.

static enum eReturnCode **estimateMemSize**(
IN tKeySize maxKeyLen,
IN tValSize maxValLen,
IN tInteger nbKeys,
OUT tIndexSize* pRecommendedMemSize = 0,
OUT tIndexSize* pMinMemSize = 0,
OUT tIndexSize* pMaxMemSize = 0,
OUT tInteger* pMaxNbKeys = 0);
Estimate the memory size that will be allocated to index n key-value pairs.

The maxKeyLen parameter exists only with character string keys, wide character string keys, fixed length array keys and multi-column keys. The maxKeyLen parameter is also the first parameter to the function setMaxKeyLen() or returned by the function getMaxKeyLen().

The maxValLen parameter exists only with Index64 classes I64MtUnsigned32, I64MtSigned32, I64MtFloat, I64MtUnsigned64, I64MtSigned64, I64MtDouble, I64MtString, I64MtWideChar and I64MtMulti. The maxValLen parameter is also the first parameter to the function setMaxValLen() or returned by the function getMaxValLen().

The nbKeys parameter gives the value of n.

The exact amount of memory required does not only depend on the number of key-value pairs to index. It depends also on the repartition of the keys. So the estimate function gives a recommended memory amount, a minimum memory amount and a maximum memory amount. Any of which can be use with setMinQuota1() or setMinQuota2().

The pMaxNbKeys returned parameter is the highest value accepted as nbKeys assuming the available memory is 0x FFFF FFFF FFFF FFFF bytes.

2.6. Multi columns reference

I64MtMulti is an Index64 classes that manage compound data. Compound data will be created from the original data using a set of per-type concatenation functions. Extract functions will do the reverse operation, meaning restoring the original data from a compound data. Extract functions are provided for convenience reasons as they are rarely useful.

For a demonstration of append and extract functions, see file "Sample10_Multi.cpp" of the "§ 1.2 Full sample".

2.6.1. Append functions

Here are the three steps to append data to a compound key:

- Initialize the compound key by calling appendToMultiBegin() function.
- Append fields one by one by using the appropriate append function for each of them.
- Terminate the compound key by calling appendToMultiEnd() function.

Compound keys must be built with care because I64MtMulti can't detect wrong data. Using the dedicated append functions is mandatory for normal functioning.

```
// Compound key needs 2 bytes more than the sum of its fields
typedef char tColumnMulti[ 2 + 4 + 10 + 4 ];
// Data made of 3 fields and a compound key
struct sDataRow {
            int Column1;     //  4 bytes exactly
           char Column2[10]; // 10 bytes maximum
   unsigned int Column3;     //  4 bytes exactly
   tColumnMulti ColumnMulti; // Compound key
};
// 3 lines of data
struct sDataRow data[3] = {
   {  1, "red",   125, },
   { -1, "green", 105, },
   {  1, "blue",  125, },
};
// Make compound keys using append functions
for(int i = 0; i < 3; i++) {
   void* p;
   appendToMultiBegin         (p, &data[i].ColumnMulti);
   appendToMultiFromSigned32  (p, &data[i].Column1    );
   appendToMultiFromString    (p, &data[i].Column2    );
   appendToMultiFromUnsigned32(p, &data[i].Column3    );
   appendToMultiEnd           (p, &data[i].ColumnMulti);
}
// Index compound keys
I64MtMulti I; I.setMaxKeyLen(20); I.setMaxValLen(sizeof(sDataRow));
printf("Insert all lines\n");
for(int i = 0; i < 3; i++) {
   eReturnCode rc = I.insert(&data[i].ColumnMulti, (tpValue)&data[i], sizeof(sDataRow));
   printf("\tLine %d, result %2u %s\n", i, rc, I. getI64ReturnText(rc));
}
```

static enum eReturnCode **appendToMultiBegin**(OUT void* OUT & toMulti, OUT void* baseData);
toMulti is a working variable that will be passed along all the append functions of the 3-steps process mentioned earlier. baseData is the address of the target compound field.

static enum eReturnCode
appendToMultiFromUnsigned32(OUT void* INOUT & toMulti, IN const void* fromUnsigned32);
toMulti is the working variable initialized by the appendToMultiBegin and passed along all the append functions of the 3-steps process mentioned earlier. fromUnsigned32 is the address of the unsigned __int32 to append to the target compound field.

static enum eReturnCode
appendToMultiFromSigned32(OUT void* INOUT & toMulti, IN const void* fromSigned32);
toMulti is the working variable initialized by the appendToMultiBegin and passed along all the append functions of the 3-steps process mentioned earlier. fromSigned32 is the address of the signed __int32 to append to the target compound field.

static enum eReturnCode
appendToMultiFromFloat(OUT void* INOUT & toMulti, IN const void* fromFloat);
toMulti is the working variable initialized by the appendToMultiBegin and passed along all the append functions of the 3-steps process mentioned earlier. fromFloat is the address of the float to append to the target compound field.

static enum eReturnCode
appendToMultiFromUnsigned64(OUT void* INOUT & toMulti, IN const void* fromUnsigned64);
toMulti is the working variable initialized by the appendToMultiBegin and passed along all the append functions of the 3-steps process mentioned earlier. fromUnsigned64 is the address of the unsigned __int64 to append to the target compound field.

static enum eReturnCode
appendToMultiFromSigned64(OUT void* INOUT & toMulti, IN const void* fromSigned64);
toMulti is the working variable initialized by the appendToMultiBegin and passed along all the append functions of the 3-steps process mentioned earlier. fromSigned64 is the signed __int64 to append to the target compound field.

static enum eReturnCode
appendToMultiFromDouble(OUT void* INOUT & toMulti, IN const void* fromDouble);
toMulti is the working variable initialized by the appendToMultiBegin and passed along all the append functions of the 3-steps process mentioned earlier. fromDouble is the address of the double to append to the target compound field.

static enum eReturnCode
appendToMultiFromString(OUT void* INOUT & toMulti, IN const void* fromString);
toMulti is the working variable initialized by the appendToMultiBegin and passed along all the append functions of the 3-steps process mentioned earlier. fromString is the address of the string to append to the target compound field.

static enum eReturnCode
appendToMultiFromWideChar(OUT void* INOUT & toMulti, IN const void* fromWideChar);
toMulti is the working variable initialized by the appendToMultiBegin and passed along all the append functions of the 3-steps process mentioned earlier. fromWideChar is the wide character string to append to the target compound field.

static enum eReturnCode
appendToMultiFromBytes(OUT void* INOUT & toMulti, IN const void* fromBytes, IN tKeySize bytes);
toMulti is the working variable initialized by the appendToMultiBegin and passed along all the append functions of the 3-steps process mentioned earlier. fromBytes is the array of bytes to append to the target compound field. bytes is the number of bytes of this array.

static enum eReturnCode **appendToMultiEnd**(NA void* IN & toMulti, OUT void* baseData);
toMulti is the working variable initialized by the appendToMultiBegin and passed along all the append functions of the 3-steps process mentioned earlier. baseData is the address of the target compound field. If this function is not called, the target compound field will be considered as empty.

2.6.2. Extract functions

Here are the two steps to extract original data from a compound key:

- Initialize the compound key by calling extractFromMultiBegin() function.
- Extract fields one by one by using the appropriate extract function for each of them.

The following example continues the example of § 2.6.1 Append functions.

```
// Retrieve compound keys
tColumnMulti ColumnMulti; tValue v; printf("Scan lines in ascending order\n");
if (I.scanAscMin(ColumnMulti, v) == eReturnCodeOk) do {
   // Display fields using extract functions
            int Column1;     //  4 bytes exactly
           char Column2[10]; // 10 bytes maximum
   unsigned int Column3;     //  4 bytes exactly
   const void* p;
   extractFromMultiBegin       (p, &ColumnMulti);
   extractFromMultiToSigned32  (p, &Column1    );
   extractFromMultiToString    (p, &Column2    );
   extractFromMultiToUnsigned32(p, &Column3    );
   printf("\tLine %d is { %2d, %-10s, %3u, },\n",
      (struct sDataRow*)v - &data[0], Column1, Column2, Column3 );
} while(I.scanAscNext(ColumnMulti, ColumnMulti, v) == eReturnCodeOk);
```

static enum eReturnCode
extractFromMultiBegin(IN const void* OUT & fromMulti, IN const void* baseData);
fromMulti is a working variable that will be passed along all the extract functions of the 2-steps process mentioned earlier. BaseData_ is a variable pointing the original compound field.

static enum eReturnCode
extractFromMultiToUnsigned32(IN const void* INOUT & fromMulti, OUT void* toUnsigned32);
fromMulti is the working variable initialized by the extractFromMultiBegin and passed along all the extract functions of the 2-steps process mentioned earlier. toUnsigned32 is the address where extracted data will be placed.

static enum eReturnCode
extractFromMultiToSigned32(IN const void* INOUT & fromMulti, OUT void* toSigned32);
fromMulti is the working variable initialized by the extractFromMultiBegin and passed along all the extract functions of the 2-steps process mentioned earlier. toSigned32 is the address where extracted data will be placed.

static enum eReturnCode
extractFromMultiToFloat(IN const void* INOUT & fromMulti, OUT void* toFloat);
fromMulti is the working variable initialized by the extractFromMultiBegin and passed along all the extract functions of the 2-steps process mentioned earlier. toFloat is the address where extracted data will be placed.

static enum eReturnCode
extractFromMultiToUnsigned64(IN const void* INOUT & fromMulti, OUT void* toUnsigned64);
fromMulti is the working variable initialized by the extractFromMultiBegin and passed along all the extract functions of the 2-steps process mentioned earlier. ToUnsigned_ is the address where extracted data will be placed.

static enum eReturnCode
extractFromMultiToSigned64(IN const void* INOUT & fromMulti, OUT void* toSigned64);
fromMulti is the working variable initialized by the extractFromMultiBegin and passed along all the extract functions of the 2-steps process mentioned earlier. toSigned64 is the address where extracted data will be placed.

static enum eReturnCode
extractFromMultiToDouble(IN const void* INOUT & fromMulti, OUT void* toDouble);
fromMulti is the working variable initialized by the extractFromMultiBegin and passed along all the extract functions of the 2-steps process mentioned earlier. toDouble is the address where extracted data will be placed.

static enum eReturnCode
extractFromMultiToString(IN const void* INOUT & fromMulti, OUT void* toString);
fromMulti is the working variable initialized by the extractFromMultiBegin and passed along all the extract functions of the 2-steps process mentioned earlier. toString is the address where extracted data will be placed.

static enum eReturnCode
extractFromMultiToWideChar(IN const void* INOUT & fromMulti, OUT void* toWideChar);
fromMulti is the working variable initialized by the extractFromMultiBegin and passed along all the extract functions of the 2-steps process mentioned earlier. toWideChar is the address where extracted data will be placed.

static enum eReturnCode
extractFromMultiToBytes(IN const void* INOUT & fromMulti, OUT void* toBytes, IN tKeySize bytes);
fromMulti is the working variable initialized by the extractFromMultiBegin and passed along all the extract functions of the 2-steps process mentioned earlier. toBytes is the address where extracted data will be placed. bytes is the number of bytes of the array to extract.

3. Return codes

3.1. Texts

Each Index64 classes function returns a code. It can be transformed into text with the getI64ReturnText() static function.

Code	Text
eReturnCodeAlignment	Multi-threads instances must be 16 bytes aligned
eReturnCodeClearBefore	Call clearIndex() before
eReturnCodeCpuCompatibility	Current cpu does not support multi-threads version
eReturnCodeCreateExistingTable	The table already exists
eReturnCodeExcCountOfTable	Can't operate when the table is exclusively referenced
eReturnCodeFileName	File name or file access error
eReturnCodeFileNameLen	File name length must be < 512
eReturnCodeFileSize	File size does not fit
eReturnCodeFromKey	From key is required for scan functions
eReturnCodeInaccurateOldValue	Old value was inaccurate
eReturnCodeKeyExists	Key already exists
eReturnCodeKeyNotFollowed	Key not followed
eReturnCodeKeyNotFound	Key not found
eReturnCodeKeyTooLong	Maximum key length is 65535
eReturnCodeKeyTooShort	Minimum key length is 1
eReturnCodeKeyType	The received key is of another key type
eReturnCodeMaxKeyLen	Key length is greater than MaxKeyLen
eReturnCodeMaxKeyLenBefore	Call setMaxKeyLen() before
eReturnCodeMaxTable	Too much tables
eReturnCodeMaxThread	Too much data threads
eReturnCodeMaxValLen	Value length is greater than MaxValLen
eReturnCodeMinMaxProxy	Check 0< nodes <=dMaxNodes, 0< threads <=dI64MaxThreads
eReturnCodeMinQuota1Set	Call setMinQuota1(0) before
eReturnCodeMinQuota2Set	Call setMinQuota2(0, 0) before
eReturnCodeNetConnect	Connect error
eReturnCodeNetPoll	Poll error or timeout
eReturnCodeNetRecv	Recv error
eReturnCodeNetSend	Send error
eReturnCodeNetSocket	Socket error
eReturnCodeNoKeyLen	No need to call setMaxKeyLen() for this class
eReturnCodeNoParam	The client didn't ask for this paramater
eReturnCodeNoTable	The table doesn't exist
eReturnCodeNotConnected	Recreate the I64Client object
eReturnCodeNotEnoughMemory	Not enough memory
eReturnCodeOk	Success
eReturnCodeOnGoingExclusive	On going call to an exclusive function, try later
eReturnCodeOnGoingShared	On going call to a shared function, try later
eReturnCodeOtherCommand	The received message has another command
eReturnCodePendingRequest	Can't operate while there is a pending request
eReturnCodeProxyNotReady	Recreate the I64Proxy object
eReturnCodeQuotaBufferSize0	Buffer size can't be null
eReturnCodeQuotaBufferSize1	Buffer size has to be null
eReturnCodeQuotaMaxToConst	Set maximum size to 0 or >= 0x1000
eReturnCodeQuotaMaxToMin	Respect maximum size >= minimum size
eReturnCodeQuotaMaxToMin2	Respect maximum size >= buffer size
eReturnCodeQuotaMin	Set minimum size to 0 or >= 0x1000
eReturnCodeQuotaModulus	Set the minimum or maximum size to a modulus of 50 Mb
eReturnCodeRefCountOfTable	Can't operate when the table is already referenced
eReturnCodeRequiredOldValue	Old value is required for CAS function
eReturnCodeReserved	Code reserved
eReturnCodeTableNameLength	Table name length must be < 64
eReturnCodeTableType	The table name exists with another type
eReturnCodeThreadBusy	At least one thread is already busy
eReturnCodeThreadPoolSize	Thread range outside thread pool
eReturnCodeTooManyKeysOrValues	Too many keys or too many values
eReturnCodeTooManyThreads	Thread parameter is too high. Upgrade your version.
eReturnCodeWKeyTooLong	Maximum wide char key length is 16384/32768
eReturnCodeZeroBytes	Minimum bytes value is 1

3.2. Descriptions

3.2.0. eReturnCodeAlignment

Any non static function: compiler alignment option or alignment of embedding object is different from 16 bytes alignment.

3.2.1. eReturnCodeClearBefore

setMaxKeyLen(), setMinQuota1(), setMinQuota2(), setMaxQuota(): the index must be empty to perform the operation.

3.2.2. eReturnCodeCpuCompatibility

Any non static function: few IA64 cpu don't support multi-threads version.

3.2.3. eReturnCodeCreateExistingTable

createTable(): the table exists with the same type, the same key length and the same value length.

3.2.4. eReturnCodeExcCountOfTable

A table constructor or a call to createTable() can't operate because another thread is currently running the function dropTable(), clear(), load() or save() on the same table.

3.2.5. eReturnCodeFileName

load() save(): the file name is incorrect, the file is missing or there is no access right to the file.

3.2.6. eReturnCodeFileNameLen

load() save(): check if the size of the file name is < 512.

3.2.7. eReturnCodeFileSize

load() save(): the file may be truncated or corrupted.

3.2.8. eReturnCodeFromKey

scanAscFirst(), scanAscNext(), scanDescFirst(), scanDescNext(), scanAscFirstDistinct (), scanAscNextDistinct (), scanDescFirstDistinct (), scanDescNextDistinct (): these scan functions start from a key. The from key parameter cannot be zero.

3.2.9. eReturnCodeInaccurateOldValue

CAS(): the compare and swap function compare the old value associated with the key to an expected value. Here the compare has failed and the swap didn't occur.

3.2.10. eReturnCodeKeyExists

insert(), bulkInsert(), update(): for unique index, the key is already indexed. For non-unique index, the key-value pair is already indexed.

3.2.11. eReturnCodeKeyNotFollowed

scanAscFirst(): for unique index, neither the key nor a greater key are indexed. For non-unique index, neither the key-value pair nor a greater key-value pair are indexed.

scanAscNext(): for unique index, no greater key is indexed. For non-unique index, no greater key-value pair is indexed.

scanDescFirst(): for unique index, neither the key nor a smaller key are indexed. For non-unique index, neither the key-value pair nor a smaller key-value pair are indexed.

scanDescNext(): for unique index, no smaller key is indexed. For non-unique index, no smaller key-value pair is indexed.

scanAscFirstDistinct(): neither the key nor a greater key are indexed.

scanAscNextDistinct(): no greater key is indexed.

scanDescFirstDistinct(): neither the key nor a smaller key are indexed.

scanDescNextDistinct(): no smaller key is indexed.

scanAscMin(), scanDescMax(),scanAscMinDistinct(), scanDescMaxDistinct (): the index is empty.

3.2.12. eReturnCodeKeyNotFound

select(), selectFirst(), selectNext(), update(),del(), bulkDelete(): for unique index, the key is not indexed. For non-unique index, the key-value pair is not indexed.

3.2.13. eReturnCodeKeyTooLong

appendToMultiFromString(), appendToMultiFromWideChar(), extractFromMultiToString(), extractFromMultiToWideChar(): for character string and wide character string, maximum length in byte is 65535.

3.2.14. eReturnCodeKeyTooShort

appendToMultiFromString(), appendToMultiFromWideChar(), extractFromMultiToString(), extractFromMultiToWideChar(): empty character string or wide character string can't be indexed.

setMaxKeyLen(), insert(), upsert(), bulkInsert(): for I64MtString, I64MtWideChar and I64MtMulti, empty key can't be indexed.

3.2.15. eReturnCodeKeyType

Scan extractors: the key type of the parameter is not the same as the key type of the table.

3.2.16. eReturnCodeMaxKeyLen

insert(), upsert(), bulkInsert(): for I64MtString, I64MtWideChar and I64MtMulti, the length of the key parameter is greater than the maximum length set by a previous call to setMaxKeyLen().

3.2.17. eReturnCodeMaxKeyLenBefore

insert(), upsert(), bulkInsert(): for I64MtString, I64MtWideChar and I64MtMulti, a call to setMaxKeyLen () is required before indexing any key.

3.2.18. eReturnCodeMaxTable

createTable(): the number of existing tables has reached the maximum value.

3.2.19. eReturnCodeMaxThread

`I64Client` constructor: the number of running thread has reached the maximum value.

3.2.20. eReturnCodeMaxValLen

insert(), update(), upsert(), bulkInsert():for I64Unsigned32 ... I64Multi, the parameter NewValLen_ is greater than the maximum length set by a previous call to setMaxValLen().

3.2.21. eReturnCodeMinMaxProxy

`I64Proxy constructor`: there must be at least one node and one thread, and maximum dMaxNodes nodes and dI64MaxThreads threads.

3.2.22. eReturnCodeMinQuota1Set

setMinQuota2(): call setMinQuota1(0) before calling setMinQuota2.

3.2.23. eReturnCodeMinQuota2Set

setMinQuota1(): call setMinQuota2(0, 0) before calling setMinQuota1.

3.2.24. eReturnCodeNetConnect

`I64Proxy constructor`: at least one node in the cluster is not started or there is a network failure.

3.2.25. eReturnCodeNetPoll

`There is a` network failure.

3.2.26. eReturnCodeNetRecv

`There is a` network failure.

3.2.27. eReturnCodeNetSend

`There is a` network failure.

3.2.28. eReturnCodeNetSocket

`I64Proxy constructor`: at least one node in the cluster is not started or there is a network failure.

3.2.29. eReturnCodeNoKeyLen

setMaxKeyLen(): key length is implicitly 4 for I64Unsigned32, I64Signed32 and I64Float. Key length is implicitly 8 for I64Unsigned64, I64Signed64 and I64Double.

3.2.30. eReturnCodeNoParam

Extractors: the boolean bringBackValue was set to false in the request thus there is no returned value in the response.

3.2.31. eReturnCodeNoTable

A table constructor doesn't create nor recreate a table. Call createTable() to solve this error.

3.2.32. eReturnCodeNotConnected

insert(), update(), del(), bulkInsert(), bulkDelete(): there is no more memory.

3.2.33. eReturnCodeNotEnoughMemory

insert(), update(), del(), bulkInsert(), bulkDelete(): there is no more memory.

3.2.34. eReturnCodeOk

Any function: the function call has succeeded.

3.2.35. eReturnCodeOnGoingExclusive

Any function: the function call will return this code if there is an exclusive function in progress.

3.2.36. eReturnCodeOnGoingShared

Any exclusive function: the function call will return this code if there is a shared function in progress. "§3.3 Return codes by functions" give the list of exclusive functions.

3.2.37. eReturnCodeOtherCommand

Extractors: the extractor doesn't apply to the current response. For example, you can't apply to a response of a delete request, an extractor that is intended to a response of a select request.

3.2.38. eReturnCodePendingRequest

dropTable(), clear(), load(), save(): these functions can't operate because there is at least one pending request. Meaning an asynchronous request hasn't got its response yet.

3.2.39. eReturnCodeProxyNotReady

`I64Client` constructor: the Proxy is not ready. Rebuild the Proxy then rebuild the client.

3.2.40. eReturnCodeQuotaBufferSize0

setMinQuota2(): buffer size can't be null. Both parameters must be non null.

3.2.41. eReturnCodeQuotaBufferSize1

setMinQuota2(): buffer size has to be null. Both parameters must be null.

3.2.42. eReturnCodeQuotaMaxToConst

setMaxQuota(): set maximum size to 0 or >= 0x1000.

3.2.43. eReturnCodeQuotaMaxToMin

setMinQuota1(), setMaxQuota(): check maximum size >= minimum size.

3.2.44. eReturnCodeQuotaMaxToMin2

setMinQuota2(), setMaxQuota(): check maximum size >= buffer size.

3.2.45. eReturnCodeQuotaMin

setMinQuota1(): set minimum size to 0 or >= 0x1000.

3.2.46. eReturnCodeQuotaModulus

setMinQuota1(), setMinQuota2(), setMaxQuota(): the size parameter has to be a multiple of 50 M.

3.2.47. eReturnCodeRefCountOfTable

dropTable(), clear(), load(), save(): these functions can't operate because another thread has currently access to the same table.

3.2.48. eReturnCodeRequiredOldValue

CAS(): the length of the expected value cannot be zero.

3.2.49. eReturnCodeReserved

Not used by current Index64 version.

3.2.50. eReturnCodeTableNameLength

Table constructor or createTable(): length of the table name should not exceed 64.

3.2.51. eReturnCodeTableType

Table constructor or createTable(): the table already exists in the cluster with the same name but with another type, another key size, or another value size. For example, an attempt to create the table TEST with the class I64ClUnsigned32 return this error, if there is in the cluster a table TEST that was made with the class I64ClString.

3.2.52. eReturnCodeThreadBusy

bulkInsert(), bulkDelete():the ranges of threads [firstThread ; firstThread+ nbThreads] are overlapping between parallel calls to functions bulkInsert(), bulkDelete().

3.2.53. eReturnCodeThreadPoolSize

bulkInsert(), bulkDelete(): check that (firstThread+ nbThreads) < n where n is the number of allocated threads via a call to the function bulkAllocateThreads().

3.2.54. eReturnCodeTooManyKeysOrValues

estimateMemSize(): too many keys or too many values. The number of key-value pairs must not exceed the returned parameter pMaxNbKeys.

3.2.55. eReturnCodeTooManyThreads

Any shared function: the function call will return this code if the number of functions in progress would exceeds the maximum value allowed for your version. Upgrade your version, accepted number of threads can be up to 65,535. "§3.3 Return codes by functions" give the list of shared functions.

3.2.56. eReturnCodeWKeyTooLong

setMaxKeyLen(): for I64MtWideChar, the maximum wide char key length is 32768 on windows and 16384 on Linux.

3.2.57. eReturnCodeZeroBytes

appendToMultiFromBytes(), extractFromMultiToBytes(): bytes can't be zero.

3.3. Return codes by functions 1

		append / extract	setMinQuota1	setMinQuota2	setMaxQuota	allocate / release threads	getMaxKeyLen	setMaxKeyLen	clear	getAllocatedMemSize	estimateMemSize	load / save	scan	select	insert	CAS	update / upsert	del	bulkInsert	bulkDelete
0	eReturnCodeOk	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
1	eReturnCodeKeyExists														x	x	x		x	
2	eReturnCodeKeyNotFound													x		x	x	x		x
3	eReturnCodeKeyNotFollowed												x							
4	eReturnCodeNotEnoughMemory		x			x									x	x	x	x	x	x
5	eReturnCodeClearBefore		x	x	x		x	x												
6	eReturnCodeZeroBytes	x																		
7	eReturnCodeKeyTooShort	x						x												
8	eReturnCodeKeyTooLong	x																		
9	eReturnCodeMaxKeyLen														x	x	x		x	
10	eReturnCodeWKeyTooLong							x												
11	eReturnCodeMaxKeyLenBefore														x	x	x		x	
12	eReturnCodeNoKeyLen							x												
13	eReturnCodeMaxValLen														x	x	x		x	
14	eReturnCodeRequiredOldValue															x				
15	eReturnCodeInaccurateOldValue															x				
16	eReturnCodeFromKey												x							
17	eReturnCodeQuotaModulus		x	x	x															
18	eReturnCodeQuotaMin		x																	
19	eReturnCodeQuotaMaxToConst				x															
20	eReturnCodeQuotaMaxToMin		x		x															
21	eReturnCodeQuotaMaxToMin2			x	x															
22	eReturnCodeQuotaBufferSize0			x																
23	eReturnCodeQuotaBufferSize1			x																
24	eReturnCodeMinQuota1Set			x																
25	eReturnCodeMinQuota2Set		x																	
26	eReturnCodeTooManyKeysOrValues										x									
27	eReturnCodeFileName											x								
28	eReturnCodeFileNameLen											x								
29	eReturnCodeFileSize											x								
30	eReturnCodeCpuCompatibility		x	x	x	x	x	x	x	x		x	x	x	x	x	x	x	x	x
31	eReturnCodeAlignment		x	x	x	x	x	x	x	x		x	x	x	x	x	x	x	x	x
32	eReturnCodeTooManyThreads					x							x	x	x	x	x	x	x	x
33	eReturnCodeThreadPoolSize																		x	x
34	eReturnCodeThreadBusy																		x	x
35	eReturnCodeOnGoingExclusive		x	x	x	x	x	x	x			x	x	x	x	x	x	x	x	x
36	eReturnCodeOnGoingShared		x	x	x	x	x	x	x			x								

3.4. Return codes by functions 2

		Proxy constructor	Client constructor	stopPipelining / echo	Table constructor	getMax KeyLen / ValLen	create / drop table	clear	getAllocatedMemSize	estimateMemSize	load / save	scan	select	insert	CAS	update / upsert	del	getVersion	Listener / Extractors
37	eReturnCodeNetSocket	x																	
38	eReturnCodeNetConnect	x																	
39	eReturnCodeNetSend	x		x			x	x	x	x	x	x	x	x	x	x	x	x	
40	eReturnCodeNetRecv	x					x	x	x	x	x	x	x	x	x	x	x	x	x
41	eReturnCodeNetPoll	x					x	x	x	x	x	x	x	x	x	x	x	x	x
42	eReturnCodeMinMaxProxy	x																	
43	eReturnCodeMaxThread		x																
44	eReturnCodeMaxTable						x												
45	eReturnCodeProxyNotReady		x																
46	eReturnCodeNotConnected				x		x			X								x	
47	eReturnCodeTableType				x		x												
48	eReturnCodeTableNameLength				x		x												
49	eReturnCodeCreateExistingTable						x												
50	eReturnCodeNoTable				x	x	x	x	x	x	x	x	x	x	x	x	x	x	
51	eReturnCodeExcCountOfTable						x	x			x								
52	eReturnCodeRefCountOfTable				x		x												
53	eReturnCodePendingRequest						x	x			x								
54	eReturnCodeKeyType																		x
55	eReturnCodeOtherCommand																		x
56	eReturnCodeNoParam																		X
57	eReturnCodeReserved																		

End of document