

Project 3

Generated by Doxygen 1.12.0

1 Class Index	1
1.1 Class List	1
2 File Index	3
2.1 File List	3
3 Class Documentation	5
3.1 Block Struct Reference	5
3.1.1 Detailed Description	6
3.1.2 Member Data Documentation	6
3.1.2.1 isAvailable	6
3.1.2.2 predecessorRBN	6
3.1.2.3 RBN	6
3.1.2.4 records	6
3.1.2.5 successorRBN	7
3.2 BlockBuffer Class Reference	7
3.2.1 Detailed Description	7
3.2.2 Constructor & Destructor Documentation	8
3.2.2.1 BlockBuffer()	8
3.2.3 Member Function Documentation	8
3.2.3.1 unpack_block()	8
3.2.4 Member Data Documentation	8
3.2.4.1 block_data	8
3.3 Buffer Class Reference	9
3.3.1 Detailed Description	10
3.3.2 Member Function Documentation	10
3.3.2.1 add_record()	10
3.3.2.2 dump_blocks()	10
3.3.2.3 get_blocks()	11
3.3.2.4 parse_csv_line()	11
3.3.2.5 process_blocks()	12
3.3.2.6 read_csv()	12
3.3.2.7 sort_records()	13
3.3.3 Member Data Documentation	13
3.3.3.1 blocks	13
3.3.3.2 records	13
3.4 FieldMetadata Struct Reference	14
3.4.1 Detailed Description	14
3.4.2 Member Data Documentation	14
3.4.2.1 name	14
3.4.2.2 typeSchema	15
3.5 HeaderRecord Class Reference	15
3.5.1 Detailed Description	17

3.5.2 Constructor & Destructor Documentation	17
3.5.2.1 HeaderRecord()	17
3.5.3 Member Function Documentation	17
3.5.3.1 addField()	17
3.5.3.2 getActiveListRBN()	18
3.5.3.3 getAvailListRBN()	18
3.5.3.4 getBlockSize()	18
3.5.3.5 getFields()	18
3.5.3.6 getFileStructureType()	18
3.5.3.7 getIndexFileName()	18
3.5.3.8 getIndexSchema()	19
3.5.3.9 getMinBlockCapacity()	19
3.5.3.10 getPrimaryKeyField()	19
3.5.3.11 getStaleFlag()	19
3.5.3.12 getVersion()	19
3.5.3.13 readHeader()	19
3.5.3.14 setActiveListRBN()	20
3.5.3.15 setAvailListRBN()	20
3.5.3.16 setBlockSize()	20
3.5.3.17 setFileStructureType()	20
3.5.3.18 setIndexFileName()	21
3.5.3.19 setIndexSchema()	21
3.5.3.20 setMinBlockCapacity()	21
3.5.3.21 setPrimaryKeyField()	22
3.5.3.22 setStaleFlag()	22
3.5.3.23 setVersion()	22
3.5.3.24 writeHeader()	22
3.5.4 Member Data Documentation	23
3.5.4.1 activeListRBN	23
3.5.4.2 availListRBN	23
3.5.4.3 blockCount	24
3.5.4.4 blockSize	24
3.5.4.5 fieldCount	24
3.5.4.6 fields	24
3.5.4.7 fileStructureType	24
3.5.4.8 headerSize	24
3.5.4.9 indexFileName	25
3.5.4.10 indexFileSchema	25
3.5.4.11 isStale	25
3.5.4.12 minBlockCapacity	25
3.5.4.13 primaryKeyField	25
3.5.4.14 recordCount	25

3.5.4.15 recordSizeBytes	26
3.5.4.16 sizeFormatType	26
3.5.4.17 version	26
3.6 Index Class Reference	26
3.6.1 Detailed Description	27
3.6.2 Member Function Documentation	27
3.6.2.1 processBlockData()	27
3.6.2.2 split()	28
3.7 mostStorage Struct Reference	29
3.7.1 Detailed Description	29
3.7.2 Member Data Documentation	29
3.7.2.1 county	29
3.7.2.2 latitude	30
3.7.2.3 longitude	30
3.7.2.4 other	30
3.7.2.5 state	30
3.7.2.6 zip_code	30
3.8 RecordBuffer Class Reference	31
3.8.1 Detailed Description	32
3.8.2 Constructor & Destructor Documentation	32
3.8.2.1 RecordBuffer()	32
3.8.3 Member Function Documentation	32
3.8.3.1 print_record()	32
3.8.3.2 unpack_record()	33
3.8.4 Member Data Documentation	33
3.8.4.1 city	33
3.8.4.2 latitude	33
3.8.4.3 longitude	33
3.8.4.4 record_data	33
3.8.4.5 state_id	34
3.8.4.6 zip_code	34
3.9 ZipCodeRecord Struct Reference	34
3.9.1 Detailed Description	35
3.9.2 Member Data Documentation	35
3.9.2.1 city	35
3.9.2.2 latitude	35
3.9.2.3 longitude	35
3.9.2.4 state_id	35
3.9.2.5 zip_code	35
4 File Documentation	37
4.1 C:/Users/mujah/OneDrive/Desktop/project/zip-code-group-project-2/p3/Block.cpp File Reference . . .	37

4.1.1 Function Documentation	38
4.1.1.1 createBlock()	38
4.1.1.2 createBlockFile()	39
4.1.1.3 dumpLogicalOrder()	40
4.1.1.4 dumpPhysicalOrder()	41
4.1.1.5 getBlockByRBN()	41
4.1.1.6 listMost()	42
4.1.1.7 parseBlockFile()	43
4.1.1.8 search()	44
4.1.1.9 splitZipLine()	45
4.1.2 Variable Documentation	46
4.1.2.1 availHeadRBN	46
4.1.2.2 blocks	46
4.1.2.3 listHeadRBN	46
4.2 Block.cpp	47
4.3 C:/Users/mujah/OneDrive/Desktop/project/zip-code-group-project-2/p3/Block.h File Reference	51
4.3.1 Detailed Description	52
4.3.2 Function Documentation	52
4.3.2.1 createBlock()	52
4.3.2.2 createBlockFile()	53
4.3.2.3 dumpLogicalOrder()	55
4.3.2.4 dumpPhysicalOrder()	55
4.3.2.5 listMost()	56
4.3.2.6 parseBlockFile()	56
4.3.2.7 search()	57
4.3.3 Variable Documentation	58
4.3.3.1 availHeadRBN	58
4.3.3.2 blocks	58
4.3.3.3 listHeadRBN	59
4.4 Block.h	59
4.5 C:/Users/mujah/OneDrive/Desktop/project/zip-code-group-project-2/p3/Buffer.cpp File Reference	59
4.6 Buffer.cpp	60
4.7 C:/Users/mujah/OneDrive/Desktop/project/zip-code-group-project-2/p3/Buffer.h File Reference	62
4.8 Buffer.h	63
4.9 C:/Users/mujah/OneDrive/Desktop/project/zip-code-group-project-2/p3/HeaderRecord.cpp File Reference	64
4.10 HeaderRecord.cpp	64
4.11 C:/Users/mujah/OneDrive/Desktop/project/zip-code-group-project-2/p3/HeaderRecord.h File Reference	66
4.12 HeaderRecord.h	67
4.13 C:/Users/mujah/OneDrive/Desktop/project/zip-code-group-project-2/p3/HeaderTest.cpp File Reference	68
4.14 HeaderTest.cpp	68
4.15 C:/Users/mujah/OneDrive/Desktop/project/zip-code-group-project-2/p3/Index.cpp File Reference	70

4.16 Index.cpp	70
4.17 C:/Users/mujah/OneDrive/Desktop/project/zip-code-group-project-2/p3/Index.h File Reference . . .	71
4.18 Index.h	72
4.19 C:/Users/mujah/OneDrive/Desktop/project/zip-code-group-project-2/p3/main.cpp File Reference . .	73
4.19.1 Function Documentation	73
4.19.1.1 main()	73
4.20 main.cpp	74
Index	77

Chapter 1

Class Index

1.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Block	Represents a single block in the blocked sequence set	5
BlockBuffer	A class to manage and process blocks of data	7
Buffer	A buffer class to manage ZipCodeRecords and process blocks of data	9
FieldMetadata	Metadata structure for field information in the header	14
HeaderRecord	Manages the header record for blocked sequence set files	15
Index	26
mostStorage	Represents geographical location information for a zip code	29
RecordBuffer	A class to manage and process individual records	31
ZipCodeRecord	34

Chapter 2

File Index

2.1 File List

Here is a list of all files with brief descriptions:

C:/Users/mujah/OneDrive/Desktop/project/zip-code-group-project-2/p3/ Block.cpp	37
C:/Users/mujah/OneDrive/Desktop/project/zip-code-group-project-2/p3/ Block.h Declaration of the Block structure and related global variables and functions for managing a blocked sequence set	51
C:/Users/mujah/OneDrive/Desktop/project/zip-code-group-project-2/p3/ Buffer.cpp	59
C:/Users/mujah/OneDrive/Desktop/project/zip-code-group-project-2/p3/ Buffer.h	62
C:/Users/mujah/OneDrive/Desktop/project/zip-code-group-project-2/p3/ HeaderRecord.cpp	64
C:/Users/mujah/OneDrive/Desktop/project/zip-code-group-project-2/p3/ HeaderRecord.h	66
C:/Users/mujah/OneDrive/Desktop/project/zip-code-group-project-2/p3/ HeaderTest.cpp	68
C:/Users/mujah/OneDrive/Desktop/project/zip-code-group-project-2/p3/ Index.cpp	70
C:/Users/mujah/OneDrive/Desktop/project/zip-code-group-project-2/p3/ Index.h	71
C:/Users/mujah/OneDrive/Desktop/project/zip-code-group-project-2/p3/ main.cpp	73

Chapter 3

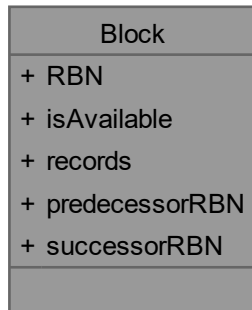
Class Documentation

3.1 Block Struct Reference

Represents a single block in the blocked sequence set.

```
#include <Block.h>
```

Collaboration diagram for Block:



Public Attributes

- int [RBN](#)
Relative [Block](#) Number (unique identifier for the block)
- bool [isAvailable](#)
Flag indicating whether the block is available.
- std::vector< std::string > [records](#)
Records stored in the block.
- int [predecessorRBN](#)
RBN of the predecessor block in the chain.
- int [successorRBN](#)
RBN of the successor block in the chain.

3.1.1 Detailed Description

Represents a single block in the blocked sequence set.

A block can either be part of the active list or the available list. It contains metadata such as predecessor and successor links and a list of records. Each block is uniquely identified by a Relative [Block](#) Number (RBN).

Definition at line [27](#) of file [Block.h](#).

3.1.2 Member Data Documentation

3.1.2.1 isAvailable

```
bool Block::isAvailable
```

Flag indicating whether the block is available.

Definition at line [29](#) of file [Block.h](#).

3.1.2.2 predecessorRBN

```
int Block::predecessorRBN
```

RBN of the predecessor block in the chain.

Definition at line [31](#) of file [Block.h](#).

3.1.2.3 RBN

```
int Block::RBN
```

Relative [Block](#) Number (unique identifier for the block)

Definition at line [28](#) of file [Block.h](#).

3.1.2.4 records

```
std::vector<std::string> Block::records
```

Records stored in the block.

Definition at line [30](#) of file [Block.h](#).

3.1.2.5 successorRBN

```
int Block::successorRBN
```

RBN of the successor block in the chain.

Definition at line 32 of file [Block.h](#).

The documentation for this struct was generated from the following file:

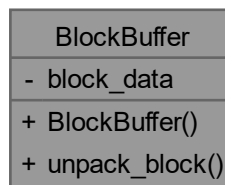
- C:/Users/mujah/OneDrive/Desktop/project/zip-code-group-project-2/p3/[Block.h](#)

3.2 BlockBuffer Class Reference

A class to manage and process blocks of data.

```
#include <Buffer.h>
```

Collaboration diagram for BlockBuffer:



Public Member Functions

- [BlockBuffer](#) (const std::unordered_map< std::string, [ZipCodeRecord](#) > &block)
A buffer class to manage individual blocks of data.
- std::vector< [ZipCodeRecord](#) > [unpack_block](#) () const
Unpacks a block into a vector of records.

Private Attributes

- std::unordered_map< std::string, [ZipCodeRecord](#) > [block_data](#)

3.2.1 Detailed Description

A class to manage and process blocks of data.

Definition at line 25 of file [Buffer.h](#).

3.2.2 Constructor & Destructor Documentation

3.2.2.1 BlockBuffer()

```
BlockBuffer::BlockBuffer (
    const std::unordered_map< std::string, ZipCodeRecord > & block) [explicit]
```

A buffer class to manage individual blocks of data.

Definition at line 14 of file [Buffer.cpp](#).

3.2.3 Member Function Documentation

3.2.3.1 unpack_block()

```
std::vector< ZipCodeRecord > BlockBuffer::unpack_block () const
```

Unpacks a block into a vector of records.

Returns

A vector of ZipCodeRecords contained in the block.

Definition at line 17 of file [Buffer.cpp](#).

Here is the caller graph for this function:



3.2.4 Member Data Documentation

3.2.4.1 block_data

```
std::unordered_map<std::string, ZipCodeRecord> BlockBuffer::block_data [private]
```

Definition at line 36 of file [Buffer.h](#).

The documentation for this class was generated from the following files:

- [C:/Users/mujah/OneDrive/Desktop/project/zip-code-group-project-2/p3/Buffer.h](#)
- [C:/Users/mujah/OneDrive/Desktop/project/zip-code-group-project-2/p3/Buffer.cpp](#)

3.3 Buffer Class Reference

A buffer class to manage ZipCodeRecords and process blocks of data.

```
#include <Buffer.h>
```

Collaboration diagram for Buffer:

Buffer
- blocks
- records
+ read_csv()
+ parse_csv_line()
+ process_blocks()
+ sort_records()
+ add_record()
+ get_blocks()
+ dump_blocks()

Public Member Functions

- bool [read_csv](#) (const std::string &csv_filename, size_t records_per_block)
Reads a CSV file and stores the records in the buffer.
- [ZipCodeRecord](#) [parse_csv_line](#) (const std::string &line) const
Parses a single line from the CSV file into a [ZipCodeRecord](#).
- void [process_blocks](#) ()
Processes the buffer block-by-block, unpacking records and fields.
- void [sort_records](#) ()
Sorts all records in the buffer by zip code.
- void [add_record](#) (size_t block_number, const [ZipCodeRecord](#) &record)
Adds a [ZipCodeRecord](#) to a specific block and the main records list.
- std::unordered_map< size_t, std::unordered_map< std::string, [ZipCodeRecord](#) > > [get_blocks](#) () const
Retrieves all blocks of [ZipCodeRecords](#).
- void [dump_blocks](#) () const
Prints the contents of each block for debugging purposes.

Private Attributes

- std::unordered_map< size_t, std::unordered_map< std::string, [ZipCodeRecord](#) > > [blocks](#)
- std::vector< [ZipCodeRecord](#) > [records](#)

3.3.1 Detailed Description

A buffer class to manage ZipCodeRecords and process blocks of data.

Definition at line 68 of file [Buffer.h](#).

3.3.2 Member Function Documentation

3.3.2.1 add_record()

```
void Buffer::add_record (  
    size_t block_number,  
    const ZipCodeRecord & record)
```

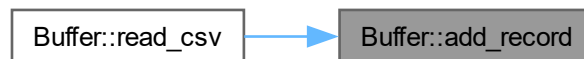
Adds a [ZipCodeRecord](#) to a specific block and the main records list.

Parameters

<i>block_number</i>	The block number to which the record should be added.
<i>record</i>	The ZipCodeRecord to be added.

Definition at line 157 of file [Buffer.cpp](#).

Here is the caller graph for this function:



3.3.2.2 dump_blocks()

```
void Buffer::dump_blocks () const
```

Prints the contents of each block for debugging purposes.

Definition at line 176 of file [Buffer.cpp](#).

3.3.2.3 get_blocks()

```
std::unordered_map< size_t, std::unordered_map< std::string, ZipCodeRecord > > Buffer::get_blocks () const
```

Retrieves all blocks of ZipCodeRecords.

Returns

A map where the key is the block number, and the value is a map of ZipCodeRecords.

`std::unordered_map<size_t, std::unordered_map<std::string, ZipCodeRecord>>` A map where the key is the block number and the value is a map of ZipCodeRecords within that block.

Definition at line 169 of file [Buffer.cpp](#).

3.3.2.4 parse_csv_line()

```
ZipCodeRecord Buffer::parse_csv_line (
    const std::string & line) const
```

Parses a single line from the CSV file into a [ZipCodeRecord](#).

Parameters

<i>line</i>	A string containing a single CSV line.
-------------	----------------------------------------

Returns

A parsed [ZipCodeRecord](#) object.

Parameters

<i>line</i>	A string containing a single CSV line.
-------------	----------------------------------------

Returns

[ZipCodeRecord](#) The parsed [ZipCodeRecord](#).

Definition at line 93 of file [Buffer.cpp](#).

Here is the caller graph for this function:



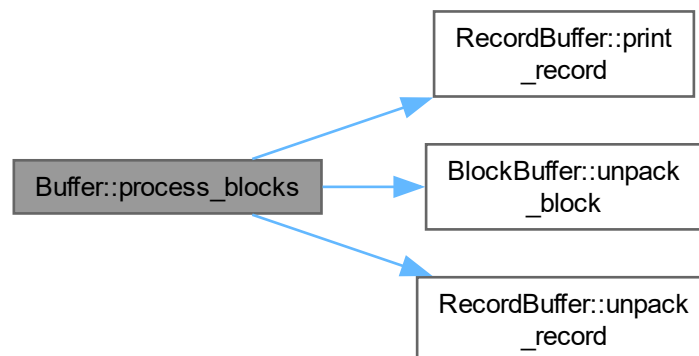
3.3.2.5 process_blocks()

```
void Buffer::process_blocks ()
```

Processes the buffer block-by-block, unpacking records and fields.

Definition at line 112 of file [Buffer.cpp](#).

Here is the call graph for this function:



3.3.2.6 read_csv()

```
bool Buffer::read_csv (
    const std::string & csv_filename,
    size_t records_per_block)
```

Reads a CSV file and stores the records in the buffer.

Parameters

<i>csv_filename</i>	The name of the CSV file to read.
<i>records_per_block</i>	The maximum number of records per block.

Returns

True if the CSV file was successfully read, false otherwise.

Parameters

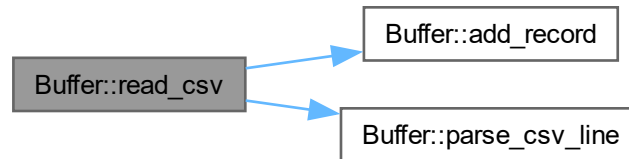
<i>csv_filename</i>	The name of the CSV file to read.
<i>records_per_block</i>	The maximum number of records per block.

Returns

true If the CSV file was successfully read and processed.
false If the file could not be opened or read.

Definition at line 59 of file [Buffer.cpp](#).

Here is the call graph for this function:



3.3.2.7 sort_records()

```
void Buffer::sort_records ()
```

Sorts all records in the buffer by zip code.

Definition at line 132 of file [Buffer.cpp](#).

3.3.3 Member Data Documentation

3.3.3.1 blocks

```
std::unordered_map<size_t, std::unordered_map<std::string, ZipCodeRecord> > Buffer::blocks  
[private]
```

Definition at line 115 of file [Buffer.h](#).

3.3.3.2 records

```
std::vector<ZipCodeRecord> Buffer::records [private]
```

Definition at line 118 of file [Buffer.h](#).

The documentation for this class was generated from the following files:

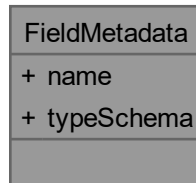
- C:/Users/mujah/OneDrive/Desktop/project/zip-code-group-project-2/p3/[Buffer.h](#)
- C:/Users/mujah/OneDrive/Desktop/project/zip-code-group-project-2/p3/[Buffer.cpp](#)

3.4 FieldMetadata Struct Reference

Metadata structure for field information in the header.

```
#include <HeaderRecord.h>
```

Collaboration diagram for FieldMetadata:



Public Attributes

- `std::string` [name](#)
Name or ID of the field.
- `std::string` [typeSchema](#)
Type and format information for the field.

3.4.1 Detailed Description

Metadata structure for field information in the header.

Definition at line 10 of file [HeaderRecord.h](#).

3.4.2 Member Data Documentation

3.4.2.1 name

```
std::string FieldMetadata::name
```

Name or ID of the field.

Definition at line 11 of file [HeaderRecord.h](#).

3.4.2.2 typeSchema

`std::string FieldMetadata::typeSchema`

Type and format information for the field.

Definition at line 12 of file [HeaderRecord.h](#).

The documentation for this struct was generated from the following file:

- [C:/Users/mujah/OneDrive/Desktop/project/zip-code-group-project-2/p3/HeaderRecord.h](#)

3.5 HeaderRecord Class Reference

Manages the header record for blocked sequence set files.

```
#include <HeaderRecord.h>
```

Collaboration diagram for HeaderRecord:

HeaderRecord
<ul style="list-style-type: none"> - fileStructureType - version - headerSize - recordSizeBytes - sizeFormatType - blockSize - minBlockCapacity - indexFileName - indexFileSchema - recordCount and 7 more...
<ul style="list-style-type: none"> + HeaderRecord() + writeHeader() + readHeader() + setFileStructureType() + setVersion() + setBlockSize() + setMinBlockCapacity() + setIndexFileName() + setIndexSchema() + setPrimaryKeyField() and 15 more...

Public Member Functions

- [HeaderRecord](#) ()
Default constructor for [HeaderRecord](#).
- bool [writeHeader](#) (std::ofstream &file)
Writes the header information to a file.
- bool [readHeader](#) (const std::string &filename)
Reads and parses header information from a file.
- void [setFileStructureType](#) (const std::string &type)
- void [setVersion](#) (const std::string &ver)
- void [setBlockSize](#) (int size)
- void [setMinBlockCapacity](#) (double capacity)
- void [setIndexFileName](#) (const std::string &name)
- void [setIndexSchema](#) (const std::string &schema)
- void [setPrimaryKeyField](#) (int field)
- void [setAvailListRBN](#) (int rbn)
- void [setActiveListRBN](#) (int rbn)
- void [setStaleFlag](#) (bool flag)
- void [addField](#) (const std::string &name, const std::string &schema)
Adds a new field definition to the header.
- std::string [getFileStructureType](#) () const
- std::string [getVersion](#) () const
- int [getBlockSize](#) () const
- double [getMinBlockCapacity](#) () const
- std::string [getIndexFileName](#) () const
- std::string [getIndexSchema](#) () const
- int [getPrimaryKeyField](#) () const
- int [getAvailListRBN](#) () const
- int [getActiveListRBN](#) () const
- bool [getStaleFlag](#) () const
- const std::vector< [FieldMetadata](#) > & [getFields](#) () const

Private Attributes

- std::string [fileStructureType](#)
Type of file structure.
- std::string [version](#)
Version of the file structure.
- int [headerSize](#)
Size of the header record in bytes.
- int [recordSizeBytes](#)
Number of bytes for record size integers.
- std::string [sizeFormatType](#)
Format type for sizes (ASCII/binary)
- int [blockSize](#)
Size of each block in bytes.
- double [minBlockCapacity](#)
Minimum block capacity (default 50%)
- std::string [indexFileName](#)
Name of the index file.
- std::string [indexFileSchema](#)
Schema information for the index file.

- int [recordCount](#)
Total number of records.
- int [blockCount](#)
Total number of blocks.
- int [fieldCount](#)
Number of fields per record.
- std::vector< [FieldMetadata](#) > [fields](#)
Metadata for each field.
- int [primaryKeyField](#)
Ordinal number of primary key field.
- int [availListRBN](#)
RBN link to block avail-list.
- int [activeListRBN](#)
RBN link to active sequence set list.
- bool [isStale](#)
Stale flag for header.

3.5.1 Detailed Description

Manages the header record for blocked sequence set files.

This class handles reading and writing header records that contain metadata about the file structure, block organization, and field definitions.

Definition at line 22 of file [HeaderRecord.h](#).

3.5.2 Constructor & Destructor Documentation

3.5.2.1 HeaderRecord()

```
HeaderRecord::HeaderRecord ()
```

Default constructor for [HeaderRecord](#).

Initializes all numeric members to sensible defaults

Definition at line 12 of file [HeaderRecord.cpp](#).

3.5.3 Member Function Documentation

3.5.3.1 addField()

```
void HeaderRecord::addField (  
    const std::string & name,  
    const std::string & schema)
```

Adds a new field definition to the header.

Parameters

<i>name</i>	Name or ID of the field
<i>schema</i>	Type and format information for the field

Definition at line 34 of file [HeaderRecord.cpp](#).

3.5.3.2 getActiveListRBN()

```
int HeaderRecord::getActiveListRBN () const [inline]
```

Definition at line 62 of file [HeaderRecord.h](#).

3.5.3.3 getAvailListRBN()

```
int HeaderRecord::getAvailListRBN () const [inline]
```

Definition at line 61 of file [HeaderRecord.h](#).

3.5.3.4 getBlockSize()

```
int HeaderRecord::getBlockSize () const [inline]
```

Definition at line 56 of file [HeaderRecord.h](#).

3.5.3.5 getFields()

```
const std::vector< FieldMetadata > & HeaderRecord::getFields () const [inline]
```

Definition at line 64 of file [HeaderRecord.h](#).

3.5.3.6 getFileStructureType()

```
std::string HeaderRecord::getFileStructureType () const [inline]
```

Definition at line 54 of file [HeaderRecord.h](#).

3.5.3.7 getIndexFileName()

```
std::string HeaderRecord::getIndexFileName () const [inline]
```

Definition at line 58 of file [HeaderRecord.h](#).

3.5.3.8 getIndexSchema()

```
std::string HeaderRecord::getIndexSchema () const [inline]
```

Definition at line 59 of file [HeaderRecord.h](#).

3.5.3.9 getMinBlockCapacity()

```
double HeaderRecord::getMinBlockCapacity () const [inline]
```

Definition at line 57 of file [HeaderRecord.h](#).

3.5.3.10 getPrimaryKeyField()

```
int HeaderRecord::getPrimaryKeyField () const [inline]
```

Definition at line 60 of file [HeaderRecord.h](#).

3.5.3.11 getStaleFlag()

```
bool HeaderRecord::getStaleFlag () const [inline]
```

Definition at line 63 of file [HeaderRecord.h](#).

3.5.3.12 getVersion()

```
std::string HeaderRecord::getVersion () const [inline]
```

Definition at line 55 of file [HeaderRecord.h](#).

3.5.3.13 readHeader()

```
bool HeaderRecord::readHeader (  
    const std::string & filename)
```

Reads and parses header information from a file.

Parameters

<i>filename</i>	Name of the file to read from
-----------------	-------------------------------

Returns

true if successful, false otherwise

Definition at line 95 of file [HeaderRecord.cpp](#).

3.5.3.14 setActiveListRBN()

```
void HeaderRecord::setActiveListRBN (  
    int rbn) [inline]
```

Definition at line 49 of file [HeaderRecord.h](#).

3.5.3.15 setAvailListRBN()

```
void HeaderRecord::setAvailListRBN (  
    int rbn) [inline]
```

Definition at line 48 of file [HeaderRecord.h](#).

3.5.3.16 setBlockSize()

```
void HeaderRecord::setBlockSize (  
    int size) [inline]
```

Definition at line 43 of file [HeaderRecord.h](#).

Here is the caller graph for this function:



3.5.3.17 setFileStructureType()

```
void HeaderRecord::setFileStructureType (  
    const std::string & type) [inline]
```

Definition at line 41 of file [HeaderRecord.h](#).

Here is the caller graph for this function:



3.5.3.18 setIndexFileName()

```
void HeaderRecord::setIndexFileName (  
    const std::string & name) [inline]
```

Definition at line 45 of file [HeaderRecord.h](#).

Here is the caller graph for this function:



3.5.3.19 setIndexSchema()

```
void HeaderRecord::setIndexSchema (  
    const std::string & schema) [inline]
```

Definition at line 46 of file [HeaderRecord.h](#).

Here is the caller graph for this function:

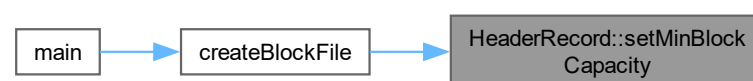


3.5.3.20 setMinBlockCapacity()

```
void HeaderRecord::setMinBlockCapacity (  
    double capacity) [inline]
```

Definition at line 44 of file [HeaderRecord.h](#).

Here is the caller graph for this function:



3.5.3.21 setPrimaryKeyField()

```
void HeaderRecord::setPrimaryKeyField (  
    int field) [inline]
```

Definition at line 47 of file [HeaderRecord.h](#).

Here is the caller graph for this function:



3.5.3.22 setStaleFlag()

```
void HeaderRecord::setStaleFlag (  
    bool flag) [inline]
```

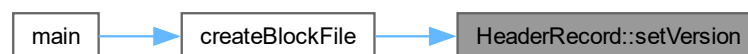
Definition at line 50 of file [HeaderRecord.h](#).

3.5.3.23 setVersion()

```
void HeaderRecord::setVersion (  
    const std::string & ver) [inline]
```

Definition at line 42 of file [HeaderRecord.h](#).

Here is the caller graph for this function:



3.5.3.24 writeHeader()

```
bool HeaderRecord::writeHeader (  
    std::ofstream & file)
```

Writes the header information to a file.

Writes the header information to an already open file stream.

Parameters

<i>file</i>	ofstream of the file to write to
-------------	----------------------------------

Returns

true if successful, false otherwise

Parameters

<i>file</i>	Reference to an open output file stream
-------------	-----------------------------------------

Returns

true if successful, false otherwise

Definition at line 48 of file [HeaderRecord.cpp](#).

Here is the caller graph for this function:



3.5.4 Member Data Documentation

3.5.4.1 activeListRBN

```
int HeaderRecord::activeListRBN [private]
```

RBN link to active sequence set list.

Definition at line 82 of file [HeaderRecord.h](#).

3.5.4.2 availListRBN

```
int HeaderRecord::availListRBN [private]
```

RBN link to block avail-list.

Definition at line 81 of file [HeaderRecord.h](#).

3.5.4.3 blockCount

```
int HeaderRecord::blockCount [private]
```

Total number of blocks.

Definition at line 77 of file [HeaderRecord.h](#).

3.5.4.4 blockSize

```
int HeaderRecord::blockSize [private]
```

Size of each block in bytes.

Definition at line 72 of file [HeaderRecord.h](#).

3.5.4.5 fieldCount

```
int HeaderRecord::fieldCount [private]
```

Number of fields per record.

Definition at line 78 of file [HeaderRecord.h](#).

3.5.4.6 fields

```
std::vector<FieldMetadata> HeaderRecord::fields [private]
```

Metadata for each field.

Definition at line 79 of file [HeaderRecord.h](#).

3.5.4.7 fileStructureType

```
std::string HeaderRecord::fileStructureType [private]
```

Type of file structure.

Definition at line 67 of file [HeaderRecord.h](#).

3.5.4.8 headerSize

```
int HeaderRecord::headerSize [private]
```

Size of the header record in bytes.

Definition at line 69 of file [HeaderRecord.h](#).

3.5.4.9 indexFileName

```
std::string HeaderRecord::indexFileName [private]
```

Name of the index file.

Definition at line 74 of file [HeaderRecord.h](#).

3.5.4.10 indexFileSchema

```
std::string HeaderRecord::indexFileSchema [private]
```

Schema information for the index file.

Definition at line 75 of file [HeaderRecord.h](#).

3.5.4.11 isStale

```
bool HeaderRecord::isStale [private]
```

Stale flag for header.

Definition at line 83 of file [HeaderRecord.h](#).

3.5.4.12 minBlockCapacity

```
double HeaderRecord::minBlockCapacity [private]
```

Minimum block capacity (default 50%)

Definition at line 73 of file [HeaderRecord.h](#).

3.5.4.13 primaryKeyField

```
int HeaderRecord::primaryKeyField [private]
```

Ordinal number of primary key field.

Definition at line 80 of file [HeaderRecord.h](#).

3.5.4.14 recordCount

```
int HeaderRecord::recordCount [private]
```

Total number of records.

Definition at line 76 of file [HeaderRecord.h](#).

3.5.4.15 recordSizeBytes

```
int HeaderRecord::recordSizeBytes [private]
```

Number of bytes for record size integers.

Definition at line 70 of file [HeaderRecord.h](#).

3.5.4.16 sizeFormatType

```
std::string HeaderRecord::sizeFormatType [private]
```

Format type for sizes (ASCII/binary)

Definition at line 71 of file [HeaderRecord.h](#).

3.5.4.17 version

```
std::string HeaderRecord::version [private]
```

Version of the file structure.

Definition at line 68 of file [HeaderRecord.h](#).

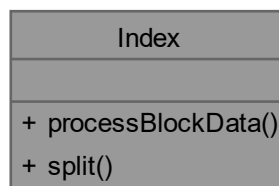
The documentation for this class was generated from the following files:

- C:/Users/mujah/OneDrive/Desktop/project/zip-code-group-project-2/p3/[HeaderRecord.h](#)
- C:/Users/mujah/OneDrive/Desktop/project/zip-code-group-project-2/p3/[HeaderRecord.cpp](#)

3.6 Index Class Reference

```
#include <Index.h>
```

Collaboration diagram for Index:



Public Member Functions

- void [processBlockData](#) (const string &inputFileName, const string &outputFileName)
Splits a string into tokens based on a specified delimiter.
- std::vector< std::string > [split](#) (const std::string &line, char delimiter)
Processes block data from an input file and organizes it into an output file.

3.6.1 Detailed Description

Definition at line 8 of file [Index.h](#).

3.6.2 Member Function Documentation

3.6.2.1 processBlockData()

```
void Index::processBlockData (  
    const string & inputFileName,  
    const string & outputFileName)
```

Splits a string into tokens based on a specified delimiter.

Processes block data from an input file and organizes it into an output file.

Parameters

<i>line</i>	The input string to be split.
<i>delimiter</i>	The character used as the delimiter for splitting the string.

Returns

A vector containing the tokens extracted from the input string.

This method reads data from an input file, extracts and processes relevant information, and writes the results into an output file. Each valid block and zip code pair is stored in the output file in the format "Block,Zip Code".

Parameters

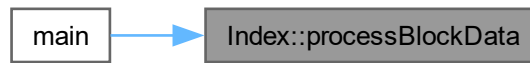
<i>inputFileName</i>	The name of the input file containing block data.
<i>outputFileName</i>	The name of the output file where processed data will be saved.

Definition at line 36 of file [Index.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:



3.6.2.2 split()

```

vector< string > Index::split (
    const std::string & line,
    char delimiter)
  
```

Processes block data from an input file and organizes it into an output file.

Splits a string into tokens based on a specified delimiter.

This method reads data from an input file, extracts and processes relevant information, and writes the results into an output file. Each valid block and zip code pair is stored in the output file in the format "Block,Zip Code".

Parameters

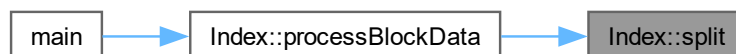
<i>inputFileName</i>	The name of the input file containing block data.
<i>outputFileName</i>	The name of the output file where processed data will be saved.
<i>line</i>	The input string to be split.
<i>delimiter</i>	The character used as the delimiter for splitting the string.

Returns

A vector containing the tokens extracted from the input string.

Definition at line 16 of file [Index.cpp](#).

Here is the caller graph for this function:



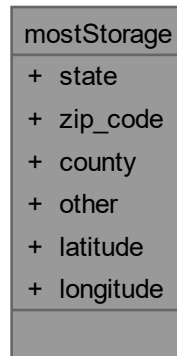
The documentation for this class was generated from the following files:

- C:/Users/mujah/OneDrive/Desktop/project/zip-code-group-project-2/p3/[Index.h](#)
- C:/Users/mujah/OneDrive/Desktop/project/zip-code-group-project-2/p3/[Index.cpp](#)

3.7 mostStorage Struct Reference

Represents geographical location information for a zip code.

Collaboration diagram for mostStorage:



Public Attributes

- `std::string state`
- `std::string zip_code`
- `std::string county`
- `std::string other`
- `double latitude`
- `double longitude`

3.7.1 Detailed Description

Represents geographical location information for a zip code.

This struct stores detailed geographical data including state, zip code, latitude, and longitude coordinates

Definition at line 189 of file [Block.cpp](#).

3.7.2 Member Data Documentation

3.7.2.1 county

```
std::string mostStorage::county
```

Definition at line 190 of file [Block.cpp](#).

3.7.2.2 latitude

```
double mostStorage::latitude
```

Definition at line 191 of file [Block.cpp](#).

3.7.2.3 longitude

```
double mostStorage::longitude
```

Definition at line 192 of file [Block.cpp](#).

3.7.2.4 other

```
std::string mostStorage::other
```

Definition at line 190 of file [Block.cpp](#).

3.7.2.5 state

```
std::string mostStorage::state
```

Definition at line 190 of file [Block.cpp](#).

3.7.2.6 zip_code

```
std::string mostStorage::zip_code
```

Definition at line 190 of file [Block.cpp](#).

The documentation for this struct was generated from the following file:

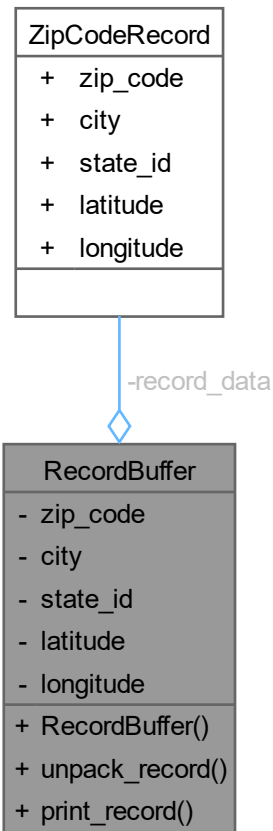
- [C:/Users/mujah/OneDrive/Desktop/project/zip-code-group-project-2/p3/Block.cpp](#)

3.8 RecordBuffer Class Reference

A class to manage and process individual records.

```
#include <Buffer.h>
```

Collaboration diagram for RecordBuffer:



Public Member Functions

- `RecordBuffer` (const `ZipCodeRecord` &record)
A buffer class to manage individual records.
- void `unpack_record` ()
Unpacks fields from the record into individual attributes.
- void `print_record` () const
Prints the contents of the record.

Private Attributes

- [ZipCodeRecord](#) `record_data`
- `std::string` `zip_code`
- `std::string` `city`
- `std::string` `state_id`
- `double` `latitude`
- `double` `longitude`

3.8.1 Detailed Description

A class to manage and process individual records.

Definition at line 42 of file [Buffer.h](#).

3.8.2 Constructor & Destructor Documentation

3.8.2.1 RecordBuffer()

```
RecordBuffer::RecordBuffer (  
    const ZipCodeRecord & record) [explicit]
```

A buffer class to manage individual records.

Definition at line 30 of file [Buffer.cpp](#).

3.8.3 Member Function Documentation

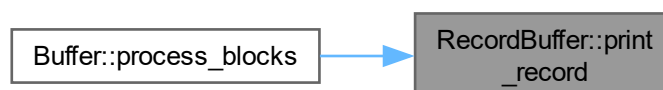
3.8.3.1 print_record()

```
void RecordBuffer::print_record () const
```

Prints the contents of the record.

Definition at line 41 of file [Buffer.cpp](#).

Here is the caller graph for this function:



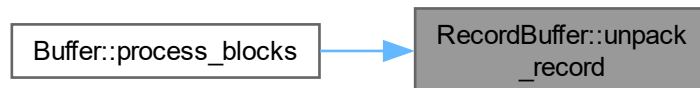
3.8.3.2 unpack_record()

```
void RecordBuffer::unpack_record ()
```

Unpacks fields from the record into individual attributes.

Definition at line 33 of file [Buffer.cpp](#).

Here is the caller graph for this function:



3.8.4 Member Data Documentation

3.8.4.1 city

```
std::string RecordBuffer::city [private]
```

Definition at line 59 of file [Buffer.h](#).

3.8.4.2 latitude

```
double RecordBuffer::latitude [private]
```

Definition at line 61 of file [Buffer.h](#).

3.8.4.3 longitude

```
double RecordBuffer::longitude [private]
```

Definition at line 62 of file [Buffer.h](#).

3.8.4.4 record_data

```
ZipCodeRecord RecordBuffer::record_data [private]
```

Definition at line 57 of file [Buffer.h](#).

3.8.4.5 state_id

```
std::string RecordBuffer::state_id [private]
```

Definition at line 60 of file [Buffer.h](#).

3.8.4.6 zip_code

```
std::string RecordBuffer::zip_code [private]
```

Definition at line 58 of file [Buffer.h](#).

The documentation for this class was generated from the following files:

- C:/Users/mujah/OneDrive/Desktop/project/zip-code-group-project-2/p3/[Buffer.h](#)
- C:/Users/mujah/OneDrive/Desktop/project/zip-code-group-project-2/p3/[Buffer.cpp](#)

3.9 ZipCodeRecord Struct Reference

```
#include <Buffer.h>
```

Collaboration diagram for ZipCodeRecord:



Public Attributes

- std::string [zip_code](#)
- std::string [city](#)
- std::string [state_id](#)
- double [latitude](#)
- double [longitude](#)

3.9.1 Detailed Description

Definition at line 11 of file [Buffer.h](#).

3.9.2 Member Data Documentation

3.9.2.1 city

```
std::string ZipCodeRecord::city
```

Definition at line 13 of file [Buffer.h](#).

3.9.2.2 latitude

```
double ZipCodeRecord::latitude
```

Definition at line 15 of file [Buffer.h](#).

3.9.2.3 longitude

```
double ZipCodeRecord::longitude
```

Definition at line 16 of file [Buffer.h](#).

3.9.2.4 state_id

```
std::string ZipCodeRecord::state_id
```

Definition at line 14 of file [Buffer.h](#).

3.9.2.5 zip_code

```
std::string ZipCodeRecord::zip_code
```

Definition at line 12 of file [Buffer.h](#).

The documentation for this struct was generated from the following file:

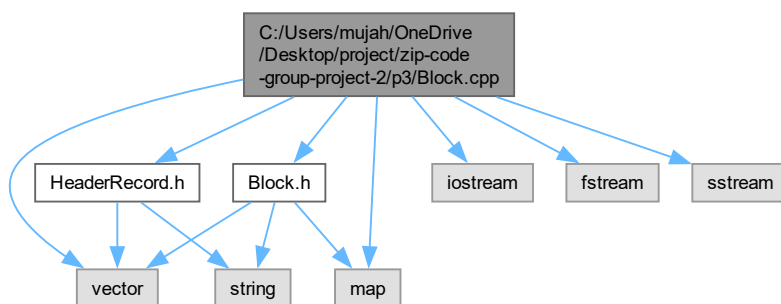
- C:/Users/mujah/OneDrive/Desktop/project/zip-code-group-project-2/p3/[Buffer.h](#)

Chapter 4

File Documentation

4.1 C:/Users/mujah/OneDrive/Desktop/project/zip-code-group-project-2/p3/Block.cpp File Reference

```
#include "Block.h"
#include <iostream>
#include <fstream>
#include <sstream>
#include <vector>
#include <map>
#include "HeaderRecord.h"
Include dependency graph for Block.cpp:
```



Classes

- struct [mostStorage](#)

Represents geographical location information for a zip code.

Functions

- bool `createBlockFile` (const std::string &inputFile, const std::string &outputFile, size_t BLOCK_SIZE)
Creates a block file from an input CSV file.
- void `parseBlockFile` (const string &blockFile)
Parses a block file and populates the global map of blocks.
- void `dumpPhysicalOrder` ()
Dumps all blocks in physical order.
- void `dumpLogicalOrder` ()
Dumps all blocks in logical order starting from the active list head.
- void `listMost` ()
Finds and lists the extreme points (easternmost, westernmost, northernmost, southernmost) for each state.
- std::vector< std::string > `splitZipLine` (const std::string &str)
Splits a string containing zip codes separated by "-z" delimiter.
- `Block` * `getBlockByRBN` (int requestedRBN)
Retrieves a block by its Relative `Block` Number (RBN)
- void `search` (const std::string &str, const std::string &indexName)
Searches for a specific zip code in the block file and index file.
- void `createBlock` (int RBN, bool isAvailable, const vector< string > &records, int predecessorRBN, int successorRBN)
Creates a new block and inserts it into the global map.

Variables

- map< int, `Block` > `blocks`
Global map of blocks indexed by Relative `Block` Number (RBN).
- int `listHeadRBN` = -1
Head of the active block list (RBN).
- int `availHeadRBN` = -1
Head of the available block list (RBN).

4.1.1 Function Documentation

4.1.1.1 createBlock()

```
void createBlock (
    int RBN,
    bool isAvailable,
    const vector< string > & records,
    int predecessorRBN,
    int successorRBN)
```

Creates a new block and inserts it into the global map.

This function initializes a new block with the provided details and inserts it into the `blocks` map. It also updates the global head pointers for the active and available block lists as needed.

Parameters

<i>RBN</i>	Relative <code>Block</code> Number of the new block.
<i>isAvailable</i>	Flag indicating whether the block is available (true) or active (false).

<i>records</i>	List of records to store in the block.
<i>predecessorRBN</i>	RBN of the predecessor block in the chain.
<i>successorRBN</i>	RBN of the successor block in the chain.

Definition at line 470 of file [Block.cpp](#).

Here is the caller graph for this function:



4.1.1.2 createBlockFile()

```

bool createBlockFile (
    const std::string & inputFile,
    const std::string & outputFile,
    size_t BLOCK_SIZE)

```

Creates a block file from an input CSV file.

This function reads an input CSV file, divides its data into fixed-size blocks, and writes those blocks into a new output file.

Parameters

<i>inputFile</i>	Path to the input CSV file.
<i>outputFile</i>	Path to the output block file.
<i>BLOCK_SIZE</i>	Maximum size of each block in bytes.

Returns

True if the file was successfully created, false otherwise.

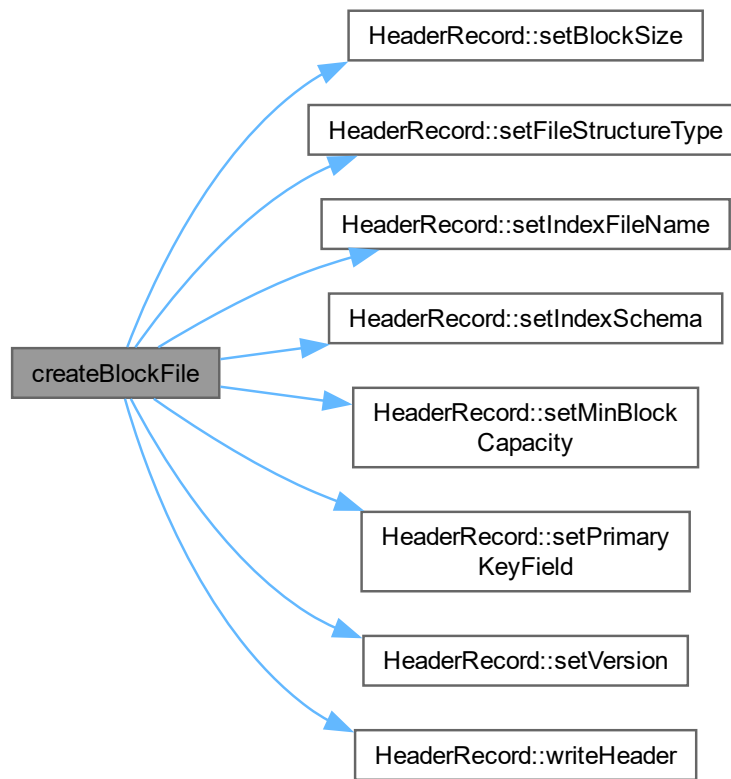
< Current block number being written

< Current size of the block in bytes

< Records for the current block

Definition at line 43 of file [Block.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:



4.1.1.3 dumpLogicalOrder()

```
void dumpLogicalOrder ()
```

Dumps all blocks in logical order starting from the active list head.

Dumps blocks in logical order starting from the active list head.

This function follows the logical chain of blocks using their successor links and prints the details of each block in sequence. < Start from the logical list head

< Move to the next block in the chain

Definition at line 170 of file [Block.cpp](#).

Here is the caller graph for this function:



4.1.1.4 dumpPhysicalOrder()

```
void dumpPhysicalOrder ()
```

Dumps all blocks in physical order.

Dumps blocks in physical order based on their RBNs.

This function iterates through all blocks stored in the global `blocks` map and prints their details in ascending order of their RBNs.

Definition at line 153 of file [Block.cpp](#).

Here is the caller graph for this function:



4.1.1.5 getBlockByRBN()

```
Block * getBlockByRBN (  
    int requestedRBN)
```

Retrieves a block by its Relative [Block](#) Number (RBN)

This function searches the global blocks map for a block with the specified RBN. It returns a pointer to the block if found, or nullptr if the block does not exist.

Parameters

<i>requestedRBN</i>	The Relative Block Number of the block to retrieve
---------------------	--------------------------------------------------------------------

Returns

Block* Pointer to the block if found, nullptr otherwise

Note

Uses the global `blocks` map to perform the lookup

Warning

Returns nullptr if the block is not found

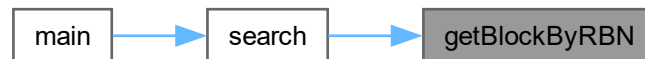
See also

[blocks](#)

[Block](#)

Definition at line 339 of file [Block.cpp](#).

Here is the caller graph for this function:



4.1.1.6 listMost()

```
void listMost ()
```

Finds and lists the extreme points (easternmost, westernmost, northernmost, southernmost) for each state.

This function processes a collection of blocks containing location records, identifying the extreme geographical points for each state based on longitude and latitude coordinates.

The function performs the following steps:

- Iterates through all blocks and their records
- Extracts state, ZIP code, latitude, and longitude information
- Tracks the extreme points for each state
- Stores the results in a map of state to extreme locations
- Prints out the extreme point ZIP codes for each state

Note

Assumes records are in a specific order:

- Record 1: ZIP code
- Record 3: State
- Record 5: Latitude
- Record 6: Longitude

Precondition

Requires a global `blocks` container with records

Postcondition

Prints extreme point information for each state

Definition at line 218 of file [Block.cpp](#).

Here is the caller graph for this function:

**4.1.1.7 parseBlockFile()**

```
void parseBlockFile (  
    const string & blockFile)
```

Parses a block file and populates the global map of blocks.

This function reads a block file, splits its content into blocks, and populates the `blocks` map with their respective details.

Parameters

<i>blockFile</i>	Path to the block file to parse.
------------------	----------------------------------

< Extracted RBN of the block

< Records in the block

Definition at line 120 of file [Block.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:



4.1.1.8 search()

```
void search (  
    const std::string & str,  
    const std::string & indexName)
```

Searches for a specific zip code in the block file and index file.

This function performs the following steps:

1. Opens the index file and block file
2. Searches for the given zip code in the index file
3. If found, retrieves the corresponding block
4. Parses the block records to extract and display matching record details

Parameters

<i>str</i>	The zip code to search for
<i>indexName</i>	The name of the index file containing zip code to RBN mappings

Precondition

Requires a valid index file and block file to be present

Postcondition

Prints the details of the matching record or a "not found" message

Note

Uses [mostStorage](#) struct to store and display record information

Assumes a specific record structure within each block

See also

[mostStorage](#)

[Block](#)

Definition at line 375 of file [Block.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:

**4.1.1.9 splitZipLine()**

```
std::vector< std::string > splitZipLine (  
    const std::string & str)
```

Splits a string containing zip codes separated by "-" delimiter.

Parameters

<i>str</i>	Input string containing zip codes
------------	-----------------------------------

Returns

`vector<string>` Vector containing individual zip codes

Processes a string containing multiple zip codes separated by "-z" delimiter, handles special cases like strings starting with "-z" and empty segments

Definition at line 295 of file [Block.cpp](#).

Here is the caller graph for this function:



4.1.2 Variable Documentation

4.1.2.1 availHeadRBN

```
int availHeadRBN = -1
```

Head of the available block list (RBN).

Stores the RBN of the first block in the available (free) list.

Definition at line 30 of file [Block.cpp](#).

4.1.2.2 blocks

```
map<int, Block> blocks
```

Global map of blocks indexed by Relative [Block](#) Number (RBN).

This map stores all blocks, where the key is the RBN, and the value is the block object.

Definition at line 16 of file [Block.cpp](#).

4.1.2.3 listHeadRBN

```
int listHeadRBN = -1
```

Head of the active block list (RBN).

Stores the RBN of the first block in the active (logical) sequence.

Definition at line 23 of file [Block.cpp](#).

4.2 Block.cpp

[Go to the documentation of this file.](#)

```

00001 #include "Block.h"
00002 #include <iostream>
00003 #include <fstream>
00004 #include <sstream>
00005 #include <vector>
00006 #include <map>
00007 #include "HeaderRecord.h"
00008
00009 using namespace std;
00010
00016 map<int, Block> blocks;
00017
00023 int listHeadRBN = -1;
00024
00030 int availHeadRBN = -1;
00031
00043 bool createBlockFile(const std::string& inputFile, const std::string& outputFile, size_t BLOCK_SIZE) {
00044     ifstream inFile(inputFile);
00045     ofstream outFile(outputFile);
00046     if (!inFile.is_open() || !outFile.is_open()) {
00047         cerr << "Error: Could not open input or output file: " << inputFile << " | " << outputFile << endl;
00048         return false;
00049     }
00050
00051     HeaderRecord header;
00052
00053     // Set basic header information
00054     header.setFileStructureType("blocked_sequence_set");
00055     header.setVersion("1.0");
00056     header.setBlockSize(512); // Default block size
00057     header.setMinBlockCapacity(0.5); // 50% minimum capacity
00058     header.setIndexFileName("headerTest.idx");
00059     header.setIndexSchema("key:string,rbn:int");
00060
00061     // Set primary key field (zip_code is field 0)
00062     header.setPrimaryKeyField(0);
00063
00064     // First write the header
00065     if (!header.writeHeader(outFile)) {
00066         std::cerr << "Failed to write header to output file" << std::endl;
00067         return false;
00068     }
00069
00070     size_t blockNumber = 1;
00071     size_t currentBlockSize = 0;
00072     vector<string> blockRecords;
00073
00074     string line;
00075     getline(inFile, line); // Skip header
00076     while (getline(inFile, line)) {
00077         size_t lineSize = line.size() + 1; // Include newline character
00078         if (currentBlockSize + lineSize > BLOCK_SIZE) {
00079             // Write the current block to the output file
00080             outFile << blockNumber << ":";
00081             for (size_t i = 0; i < blockRecords.size(); i++) {
00082                 outFile << blockRecords[i];
00083                 if (i < blockRecords.size() - 1) outFile << ",";
00084             }
00085             outFile << "\n";
00086
00087             blockRecords.clear();
00088             currentBlockSize = 0;
00089             blockNumber++;
00090         }
00091
00092         blockRecords.push_back(line);
00093         currentBlockSize += lineSize;
00094     }
00095
00096     // Write the last block if there are remaining records
00097     if (!blockRecords.empty()) {
00098         outFile << blockNumber << ":";
00099         for (size_t i = 0; i < blockRecords.size(); i++) {
00100             outFile << blockRecords[i];
00101             if (i < blockRecords.size() - 1) outFile << ",";
00102         }
00103         outFile << "\n";
00104     }
00105
00106     inFile.close();
00107     outFile.close();
00108 }

```

```

00109     return true;
00110 }
00111
00120 void parseBlockFile(const string& blockFile) {
00121     ifstream inFile(blockFile);
00122     if (!inFile.is_open()) {
00123         cerr << "Error: Could not open block file: " << blockFile << endl;
00124         return;
00125     }
00126
00127     string line;
00128     while (getline(inFile, line)) {
00129         size_t colonPos = line.find(':');
00130         int RBN = stoi(line.substr(0, colonPos));
00131         string recordsPart = line.substr(colonPos + 1);
00132
00133         vector<string> records;
00134         stringstream recordStream(recordsPart);
00135         string record;
00136         while (getline(recordStream, record, ',')) {
00137             records.push_back(record);
00138         }
00139
00140         // Create a block using the parsed data
00141         createBlock(RBN, false, records, -1, -1);
00142     }
00143
00144     inFile.close();
00145 }
00146
00153 void dumpPhysicalOrder() {
00154     cout << "Dumping Blocks by Physical Order:\n";
00155     for (const auto& [RBN, block] : blocks) {
00156         cout << "RBN: " << RBN << " ";
00157         for (const string& record : block.records) {
00158             cout << record << " ";
00159         }
00160         cout << "\n";
00161     }
00162 }
00163
00170 void dumpLogicalOrder() {
00171     cout << "Dumping Blocks by Logical Order:\n";
00172     int currentRBN = listHeadRBN;
00173     while (currentRBN != -1) {
00174         const Block& block = blocks[currentRBN];
00175         cout << "RBN: " << currentRBN << " ";
00176         for (const string& record : block.records) {
00177             cout << record << " ";
00178         }
00179         cout << "\n";
00180         currentRBN = block.successorRBN;
00181     }
00182 }
00189 struct mostStorage {
00190     std::string state, zip_code, county, other;
00191     double latitude;
00192     double longitude;
00193 };
00218 void listMost() {
00219     int currentRBN = listHeadRBN;
00220     int recordPart = 0;
00221     int testnum = 0;
00222     mostStorage current, easternmost, westernmost, northernmost, southernmost;
00223     std::map<string, std::vector<mostStorage> sorted_directions;
00224
00225     for (const auto& [RBN, block] : blocks) {
00226         bool initialized = false;
00227
00228         for (const string& record : block.records) {
00229             recordPart++;
00230             if (recordPart == 1) {
00231                 current.zip_code = record;
00232             }
00233
00234             if (recordPart == 3) {
00235                 current.state = record;
00236             }
00237
00238             if (recordPart == 5) {
00239                 current.latitude = std::stod(record);
00240             }
00241
00242             if (recordPart == 6) {
00243                 current.longitude = std::stod(record);
00244                 if (!initialized) {
00245                     easternmost = current;

```



```

00246         westernmost = current;
00247         northernmost = current;
00248         southernmost = current;
00249         initialized = true;
00250     }
00251     if ( current.longitude < easternmost.longitude ) {
00252         easternmost = current;
00253     }
00254     if ( current.longitude > westernmost.longitude ) {
00255         westernmost = current;
00256     }
00257     if ( current.latitude > northernmost.latitude ) {
00258         northernmost = current;
00259     }
00260     if ( current.latitude < southernmost.latitude ) {
00261         southernmost = current;
00262     }
00263     recordPart=0;
00264     sorted_directions[ current.state ] = { easternmost, westernmost, northernmost,
southernmost };
00265 }
00266
00267
00268
00269
00270     }
00271     }
00272
00273
00274
00275
00276
00277     cout << "State: " << "Easternmost: " << "westernmost: " << "northernmost: " << "southernmost: " << endl;
00278     for (const auto& [state, locations] : sorted_directions) {
00279         if (locations.size() == 4) { // Ensure we have all 4 directional records
00280             cout << state << ", "
00281                 << locations[0].zip_code << ", " // Easternmost
00282                 << locations[1].zip_code << ", " // Westernmost
00283                 << locations[2].zip_code << ", " // Northernmost
00284                 << locations[3].zip_code << "\n"; // Southernmost
00285         }
00286     }
00287 }
00295 std::vector<std::string> splitZipLine(const std::string& str) {
00296     std::vector<std::string> result;
00297     size_t start = 0;
00298     size_t end = str.find("-", start);
00299
00300     // Skip the first empty part if string starts with "-"
00301     if (start == end) {
00302         start += 2; // length of "-"
00303         end = str.find("-", start);
00304     }
00305
00306     while (end != std::string::npos) {
00307         // Add the part between current position and next "-"
00308         if (end - start > 0) {
00309             result.push_back(str.substr(start, end - start));
00310         }
00311         start = end + 2; // Skip over "-"
00312         end = str.find("-", start);
00313     }
00314
00315     // Add the last part if there's anything left
00316     if (start < str.length()) {
00317         result.push_back(str.substr(start));
00318     }
00319
00320     return result;
00321 }
00322
00323
00339 Block* getBlockByRBN(int requestedRBN) {
00340     // Check if the block exists in the global blocks map
00341     auto it = blocks.find(requestedRBN);
00342
00343     if (it != blocks.end()) {
00344         // Block found, return a pointer to the block
00345         return &(it->second);
00346     } else {
00347         // Block not found
00348         std::cerr << "Block with RBN " << requestedRBN << " not found." << std::endl;
00349         return nullptr;
00350     }
00351 }
00352
00375 void search(const std::string& str, const std::string& indexName){

```

```

00376     mostStorage current;
00377     bool notfound = true;
00378     std::string correct_line;
00379     std::ifstream file2(indexName); // Open the file add name later
00380     if (!file2.is_open()) {
00381         std::cerr << "Error opening file: index.txt " << std::endl;
00382         return;
00383     }
00384     std::ifstream file3("block.txt"); // Open the file add name later
00385     if (!file3.is_open()) {
00386         std::cerr << "Error opening file: block.txt " << std::endl;
00387         return;
00388     }
00389     std::string strcopy = str;
00390
00391     std::string rbn, zipcode, line;
00392     int recordPart = 0;
00393     //int i = 5;
00394     getline( file2, line );
00395     line = "";
00396     while ((file2 > zipcode > rbn) && !file2.eof()) { // reads word by word
00397         if(zipcode == str){
00398             int block = std::stoi(rbn);
00399             cout << "Zipcode: " << zipcode << " is at " << block << endl;
00400             Block* myBlock = getBlockByRBN(block);
00401             for (const string& record : myBlock->records) {
00402                 recordPart++;
00403                 if(recordPart == 1){
00404                     current.zip_code = record;
00405
00406                 }
00407                 if(recordPart == 2){
00408                     current.other = record;
00409
00410                 }
00411                 if(recordPart == 3){
00412                     current.state = record;
00413                 }
00414                 if(recordPart == 4){
00415                     current.county = record;
00416
00417                 }
00418                 if(recordPart == 5){
00419                     current.latitude = std::stod(record);
00420                 }
00421                 if(recordPart == 6){
00422                     current.longitude = std::stod(record);
00423
00424                 }
00425                 if(current.zip_code == zipcode){
00426                     cout << current.zip_code << " " << current.other << " " << current.state << "
" << current.county
00427                     << " " << current.latitude << " " << current.longitude << " " << endl;
00428                     notfound = false;
00429                     break;
00430                 }
00431                 // current.longitude = std::stod(record);
00432                 /* if (!initialized) {
00433                     easternmost = current;
00434                     westernmost = current;
00435                     northernmost = current;
00436                     southernmost = current;
00437                     initialized = true;*/
00438
00439                 recordPart=0;
00440             }
00441         }
00442     }
00443
00444     // break;
00445 }
00446
00447
00448
00449 if(notfound){
00450     cout << str << " was not found in the file." << endl;
00451 }
00452 file2.close();
00453 }
00454
00455
00456 void createBlock(int RBN, bool isAvailable, const vector<string>& records, int predecessorRBN, int
successorRBN) {
00471     Block block;
00472     block.RBN = RBN;
00473     block.isAvailable = isAvailable;

```

```

00474     block.records = records;
00475     block.predecessorRBN = predecessorRBN;
00476     block.successorRBN = successorRBN;
00477
00478     blocks[RBN] = block;
00479
00480     // Update the global head pointers
00481     if (!isAvailable && listHeadRBN == -1) {
00482         listHeadRBN = RBN;
00483     }
00484     if (isAvailable && availHeadRBN == -1) {
00485         availHeadRBN = RBN;
00486     }
00487 }

```

4.3 C:/Users/mujah/OneDrive/Desktop/project/zip-code-group-project-2/p3/Block.h File Reference

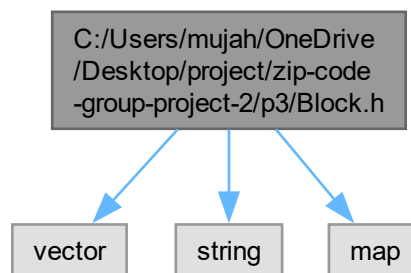
Declaration of the [Block](#) structure and related global variables and functions for managing a blocked sequence set.

```

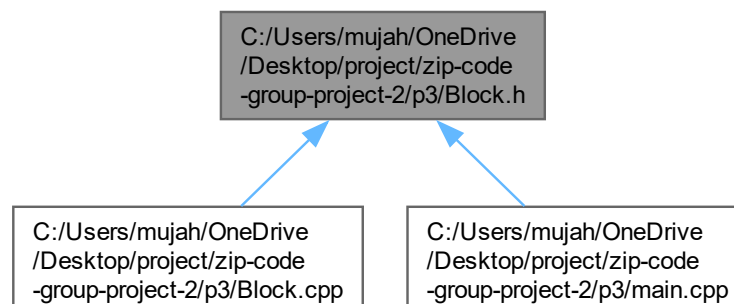
#include <vector>
#include <string>
#include <map>

```

Include dependency graph for Block.h:



This graph shows which files directly or indirectly include this file:



Classes

- struct [Block](#)
Represents a single block in the blocked sequence set.

Functions

- void [dumpPhysicalOrder](#) ()
Dumps blocks in physical order based on their RBNs.
- void [dumpLogicalOrder](#) ()
Dumps blocks in logical order starting from the active list head.
- void [createBlock](#) (int RBN, bool isAvailable, const std::vector< std::string > &records, int predecessorRBN, int successorRBN)
Creates a new block and inserts it into the global map.
- void [parseBlockFile](#) (const std::string &blockFile)
Parses a block file and populates the global map of blocks.
- bool [createBlockFile](#) (const std::string &inputFile, const std::string &outputFile, size_t BLOCK_SIZE=512)
Creates a block file from an input CSV file.
- void [listMost](#) ()
Finds and lists the extreme points (easternmost, westernmost, northernmost, southernmost) for each state.
- void [search](#) (const std::string &str, const std::string &indexName)
Searches for a specific zip code in the block file and index file.

Variables

- std::map< int, [Block](#) > [blocks](#)
Global map of blocks indexed by Relative [Block](#) Number (RBN).
- int [listHeadRBN](#)
Head of the active block list (RBN).
- int [availHeadRBN](#)
Head of the available block list (RBN).

4.3.1 Detailed Description

Declaration of the [Block](#) structure and related global variables and functions for managing a blocked sequence set.

This file defines the structure of a block and declares global variables and functions used to manage a sequence of blocks for a blocked file system. It supports operations such as dumping blocks in physical or logical order.

Date

11/21/2024

Definition in file [Block.h](#).

4.3.2 Function Documentation

4.3.2.1 createBlock()

```
void createBlock (
    int RBN,
    bool isAvailable,
    const std::vector< std::string > & records,
    int predecessorRBN,
    int successorRBN)
```

Creates a new block and inserts it into the global map.

Parameters

<i>RBN</i>	Relative Block Number of the new block.
<i>isAvailable</i>	Flag indicating whether the block is available (true) or active (false).
<i>records</i>	List of records to store in the block.
<i>predecessorRBN</i>	RBN of the predecessor block in the chain.
<i>successorRBN</i>	RBN of the successor block in the chain.

This function initializes a new block with the provided parameters and adds it to the global map.

4.3.2.2 createBlockFile()

```
bool createBlockFile (  
    const std::string & inputFile,  
    const std::string & outputFile,  
    size_t BLOCK_SIZE)
```

Creates a block file from an input CSV file.

This function reads an input CSV file, divides the data into blocks of a specified size, and writes the blocks to an output file.

Parameters

<i>inputFile</i>	Path to the input CSV file.
<i>outputFile</i>	Path to the output block file.
<i>BLOCK_SIZE</i>	Maximum size of each block in bytes (default is 512).

Returns

True if successful, false otherwise.

This function reads an input CSV file, divides its data into fixed-size blocks, and writes those blocks into a new output file.

Parameters

<i>inputFile</i>	Path to the input CSV file.
<i>outputFile</i>	Path to the output block file.
<i>BLOCK_SIZE</i>	Maximum size of each block in bytes.

Returns

True if the file was successfully created, false otherwise.

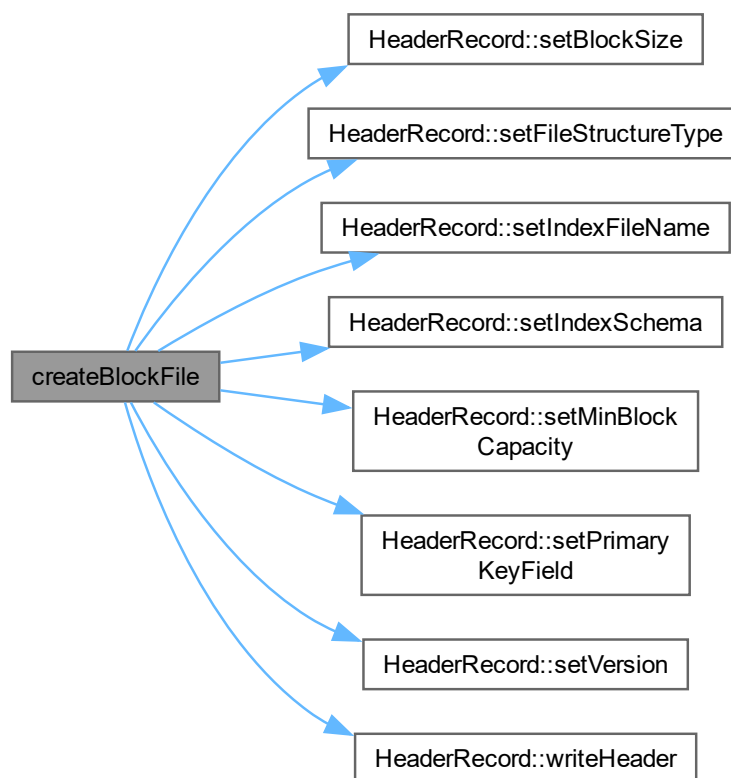
< Current block number being written

< Current size of the block in bytes

< Records for the current block

Definition at line 43 of file [Block.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:



4.3.2.3 dumpLogicalOrder()

```
void dumpLogicalOrder ()
```

Dumps blocks in logical order starting from the active list head.

This function follows the logical chain of blocks using their successor links and prints details of each block.

Dumps blocks in logical order starting from the active list head.

This function follows the logical chain of blocks using their successor links and prints the details of each block in sequence. < Start from the logical list head

< Move to the next block in the chain

Definition at line 170 of file [Block.cpp](#).

Here is the caller graph for this function:



4.3.2.4 dumpPhysicalOrder()

```
void dumpPhysicalOrder ()
```

Dumps blocks in physical order based on their RBNs.

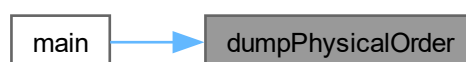
This function iterates over all blocks in ascending order of their RBNs and prints their details. Available blocks are explicitly marked.

Dumps blocks in physical order based on their RBNs.

This function iterates through all blocks stored in the global `blocks` map and prints their details in ascending order of their RBNs.

Definition at line 153 of file [Block.cpp](#).

Here is the caller graph for this function:



4.3.2.5 listMost()

```
void listMost ()
```

Finds and lists the extreme points (easternmost, westernmost, northernmost, southernmost) for each state.

This function processes a collection of blocks containing location records, identifying the extreme geographical points for each state based on longitude and latitude coordinates.

The function performs the following steps:

- Iterates through all blocks and their records
- Extracts state, ZIP code, latitude, and longitude information
- Tracks the extreme points for each state
- Stores the results in a map of state to extreme locations
- Prints out the extreme point ZIP codes for each state

Note

Assumes records are in a specific order:

- Record 1: ZIP code
- Record 3: State
- Record 5: Latitude
- Record 6: Longitude

Precondition

Requires a global `blocks` container with records

Postcondition

Prints extreme point information for each state

Definition at line 218 of file [Block.cpp](#).

Here is the caller graph for this function:



4.3.2.6 parseBlockFile()

```
void parseBlockFile (  
    const std::string & blockFile)
```

Parses a block file and populates the global map of blocks.

This function reads a block file, extracts block information, and populates the global `blocks` map.

Parameters

<i>blockFile</i>	Path to the block file to parse.
------------------	----------------------------------

4.3.2.7 search()

```
void search (
    const std::string & str,
    const std::string & indexName)
```

Searches for a specific zip code in the block file and index file.

This function performs the following steps:

1. Opens the index file and block file
2. Searches for the given zip code in the index file
3. If found, retrieves the corresponding block
4. Parses the block records to extract and display matching record details

Parameters

<i>str</i>	The zip code to search for
<i>indexName</i>	The name of the index file containing zip code to RBN mappings

Precondition

Requires a valid index file and block file to be present

Postcondition

Prints the details of the matching record or a "not found" message

Note

Uses [mostStorage](#) struct to store and display record information
Assumes a specific record structure within each block

See also

[mostStorage](#)
[Block](#)

Definition at line 375 of file [Block.cpp](#).

Here is the call graph for this function:



Here is the caller graph for this function:



4.3.3 Variable Documentation

4.3.3.1 availHeadRBN

```
int availHeadRBN [extern]
```

Head of the available block list (RBN).

Stores the RBN of the first block in the available (free) list.

Definition at line 30 of file [Block.cpp](#).

4.3.3.2 blocks

```
std::map<int, Block> blocks [extern]
```

Global map of blocks indexed by Relative [Block](#) Number (RBN).

This map stores all blocks, with the RBN as the key and the corresponding block as the value.

This map stores all blocks, where the key is the RBN, and the value is the block object.

Definition at line 16 of file [Block.cpp](#).

4.3.3.3 listHeadRBN

```
int listHeadRBN [extern]
```

Head of the active block list (RBN).

Stores the RBN of the first block in the logical (active) sequence.

Stores the RBN of the first block in the active (logical) sequence.

Definition at line 23 of file [Block.cpp](#).

4.4 Block.h

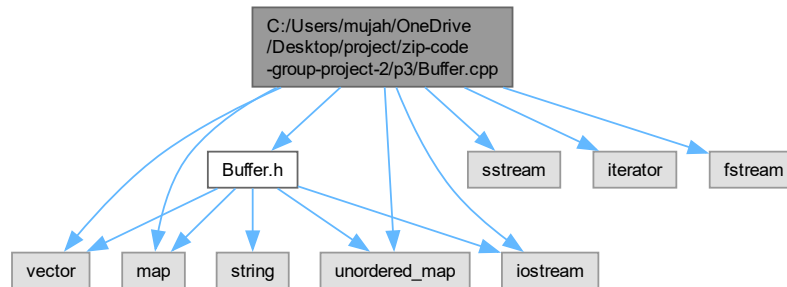
[Go to the documentation of this file.](#)

```
00001
00012 #ifndef BLOCK_H
00013 #define BLOCK_H
00014
00015 #include <vector>
00016 #include <string>
00017 #include <map>
00018
00027 struct Block {
00028     int RBN;
00029     bool isAvailable;
00030     std::vector<std::string> records;
00031     int predecessorRBN;
00032     int successorRBN;
00033 };
00034
00040 extern std::map<int, Block> blocks;
00041
00047 extern int listHeadRBN;
00048
00054 extern int availHeadRBN;
00055
00062 void dumpPhysicalOrder();
00063
00069 void dumpLogicalOrder();
00070
00082 void createBlock(int RBN, bool isAvailable, const std::vector<std::string>& records, int
    predecessorRBN, int successorRBN);
00083
00091 void parseBlockFile(const std::string& blockFile);
00092
00103 bool createBlockFile(const std::string& inputFile, const std::string& outputFile, size_t BLOCK_SIZE =
    512);
00104
00105
00106
00107 void listMost();
00108
00109 void search(const std::string& str, const std::string& indexName);
00110
00111 #endif // BLOCK_H
```

4.5 C:/Users/mujah/OneDrive/Desktop/project/zip-code-group-project-2/p3/Buffer.cpp File Reference

```
#include "Buffer.h"
#include <iostream>
#include <sstream>
#include <iterator>
#include <fstream>
#include <map>
```

```
#include <vector>
#include <unordered_map>
Include dependency graph for Buffer.cpp:
```



4.6 Buffer.cpp

[Go to the documentation of this file.](#)

```
00001 #include "Buffer.h"
00002 #include <iostream>
00003 #include <sstream>
00004 #include <iterator>
00005 #include <fstream>
00006 #include <map>
00007 #include <vector>
00008 #include <unordered_map>
00009
00013 // BlockBuffer class method definitions
00014 BlockBuffer::BlockBuffer(const std::unordered_map<std::string, ZipCodeRecord>& block)
00015     : block_data(block) {}
00016
00017 std::vector<ZipCodeRecord> BlockBuffer::unpack_block() const {
00018     std::vector<ZipCodeRecord> records;
00019     for (const auto& entry : block_data) {
00020         records.push_back(entry.second);
00021     }
00022     return records;
00023 }
00024
00025
00029 // RecordBuffer class method definitions
00030 RecordBuffer::RecordBuffer(const ZipCodeRecord& record)
00031     : record_data(record) {}
00032
00033 void RecordBuffer::unpack_record() {
00034     zip_code = record_data.zip_code;
00035     city = record_data.city;
00036     state_id = record_data.state_id;
00037     latitude = record_data.latitude;
00038     longitude = record_data.longitude;
00039 }
00040
00041 void RecordBuffer::print_record() const {
00042     std::cout << "ZipCode: " << zip_code
00043               << ", City: " << city
00044               << ", State: " << state_id
00045               << ", Latitude: " << latitude
00046               << ", Longitude: " << longitude
00047               << std::endl;
00048 }
00049
00050
00059 bool Buffer::read_csv(const std::string& csv_filename, size_t records_per_block) {
00060     std::ifstream file(csv_filename);
00061     if (!file.is_open()) {
00062         std::cerr << "Failed to open file: " << csv_filename << std::endl;
00063         return false;
00064     }
}
```

```

00065
00066     std::string line;
00067     std::getline(file, line); // Skip the header line
00068
00069     size_t block_number = 0;
00070     size_t record_count = 0;
00071
00072     while (std::getline(file, line)) {
00073         ZipCodeRecord record = parse_csv_line(line);
00074         add_record(block_number, record);
00075
00076         if (++record_count >= records_per_block) {
00077             block_number++;
00078             record_count = 0;
00079         }
00080     }
00081
00082     file.close();
00083     std::cout << "CSV loaded into the buffer successfully." << std::endl;
00084     return true;
00085 }
00086
00093 ZipCodeRecord Buffer::parse_csv_line(const std::string& line) const {
00094     std::stringstream ss(line);
00095     std::string token;
00096     ZipCodeRecord record;
00097
00098     std::getline(ss, record.zip_code, ',');
00099     std::getline(ss, record.city, ',');
00100     std::getline(ss, record.state_id, ',');
00101     std::getline(ss, token, ',');
00102     record.latitude = std::stod(token);
00103     std::getline(ss, token, ',');
00104     record.longitude = std::stod(token);
00105
00106     return record;
00107 }
00108
00112 void Buffer::process_blocks() {
00113     for (const auto& block_entry : blocks) {
00114         size_t block_number = block_entry.first;
00115         const auto& block = block_entry.second;
00116
00117         BlockBuffer block_buffer(block);
00118         std::vector<ZipCodeRecord> records = block_buffer.unpack_block();
00119
00120         std::cout << "Processing Block " << block_number << std::endl;
00121         for (const auto& record : records) {
00122             RecordBuffer record_buffer(record);
00123             record_buffer.unpack_record();
00124             record_buffer.print_record();
00125         }
00126     }
00127 }
00128
00132 void Buffer::sort_records() {
00133     std::map<std::string, ZipCodeRecord> sorted_records;
00134
00135     for (const auto& record : records) {
00136         sorted_records[record.zip_code] = record;
00137     }
00138
00139     std::cout << "Records sorted by Zip Code:" << std::endl;
00140     for (const auto& entry : sorted_records) {
00141         const auto& record = entry.second;
00142         std::cout << "ZipCode: " << record.zip_code
00143             << ", City: " << record.city
00144             << ", State: " << record.state_id
00145             << ", Latitude: " << record.latitude
00146             << ", Longitude: " << record.longitude
00147             << std::endl;
00148     }
00149 }
00150
00157 void Buffer::add_record(size_t block_number, const ZipCodeRecord& record) {
00158     blocks[block_number][record.zip_code] = record;
00159     records.push_back(record);
00160 }
00161
00169 std::unordered_map<size_t, std::unordered_map<std::string, ZipCodeRecord> Buffer::get_blocks() const {
00170     return blocks;
00171 }
00172
00176 void Buffer::dump_blocks() const {
00177     for (const auto& block : blocks) {
00178         std::cout << "Block " << block.first << " contains the following ZipCodeRecords:" << std::endl;
00179         for (const auto& record_pair : block.second) {

```

```

00180         std::cout << "ZipCode: " << record_pair.second.zip_code
00181                     << ", City: " << record_pair.second.city
00182                     << ", State: " << record_pair.second.state_id
00183                     << ", Latitude: " << record_pair.second.latitude
00184                     << ", Longitude: " << record_pair.second.longitude
00185                     << std::endl;
00186     }
00187 }
00188 }

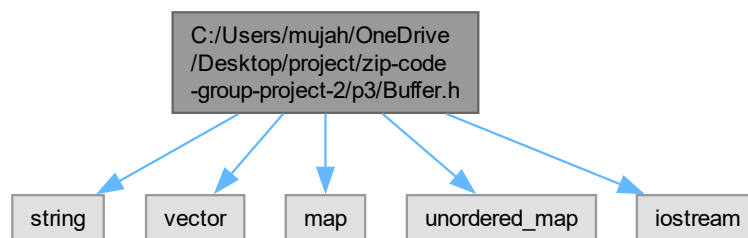
```

4.7 C:/Users/mujah/OneDrive/Desktop/project/zip-code-group-project-2/p3/Buffer.h File Reference

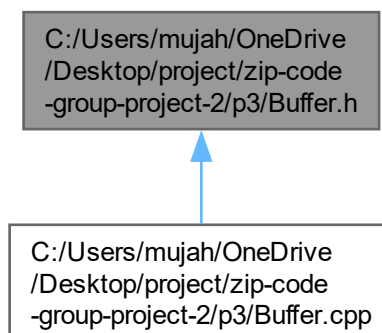
```

#include <string>
#include <vector>
#include <map>
#include <unordered_map>
#include <iostream>
Include dependency graph for Buffer.h:

```



This graph shows which files directly or indirectly include this file:



Classes

- struct [ZipCodeRecord](#)
- class [BlockBuffer](#)
 - A class to manage and process blocks of data.*
- class [RecordBuffer](#)
 - A class to manage and process individual records.*
- class [Buffer](#)
 - A buffer class to manage ZipCodeRecords and process blocks of data.*

4.8 Buffer.h

[Go to the documentation of this file.](#)

```

00001 #ifndef BUFFER_H
00002 #define BUFFER_H
00003
00004 #include <string>
00005 #include <vector>
00006 #include <map>
00007 #include <unordered_map>
00008 #include <iostream>
00009
00010 // Define a struct to represent a zip code record.
00011 struct ZipCodeRecord {
00012     std::string zip_code;
00013     std::string city;
00014     std::string state_id;
00015     double latitude;
00016     double longitude;
00017 };
00018
00019 // Forward declaration of the Buffer class
00020 class Buffer;
00021
00025 class BlockBuffer {
00026 public:
00027     explicit BlockBuffer(const std::unordered_map<std::string, ZipCodeRecord>& block);
00028
00033     std::vector<ZipCodeRecord> unpack_block() const;
00034
00035 private:
00036     std::unordered_map<std::string, ZipCodeRecord> block_data;
00037 };
00038
00042 class RecordBuffer {
00043 public:
00044     explicit RecordBuffer(const ZipCodeRecord& record);
00045
00049     void unpack_record();
00050
00054     void print_record() const;
00055
00056 private:
00057     ZipCodeRecord record_data;
00058     std::string zip_code;
00059     std::string city;
00060     std::string state_id;
00061     double latitude;
00062     double longitude;
00063 };
00064
00068 class Buffer {
00069 public:
00076     bool read_csv(const std::string& csv_filename, size_t records_per_block);
00077
00083     ZipCodeRecord parse_csv_line(const std::string& line) const;
00084
00088     void process_blocks();
00089
00093     void sort_records();
00094
00100     void add_record(size_t block_number, const ZipCodeRecord& record);
00101
00106     std::unordered_map<size_t, std::unordered_map<std::string, ZipCodeRecord> get_blocks() const;
00107
00111     void dump_blocks() const;

```

```

00112
00113 private:
00114     // Map where the key is the block number, and the value is a map of ZipCodeRecords in the block.
00115     std::unordered_map<size_t, std::unordered_map<std::string, ZipCodeRecord> blocks;
00116
00117     // A flat list of all ZipCodeRecords, used for sorting and other operations.
00118     std::vector<ZipCodeRecord> records;
00119 };
00120
00121 #endif // BUFFER_H

```

4.9 C:/Users/mujah/OneDrive/Desktop/project/zip-code-group-project-2/p3/HeaderRecord.cpp File Reference

```
#include "HeaderRecord.h"
```

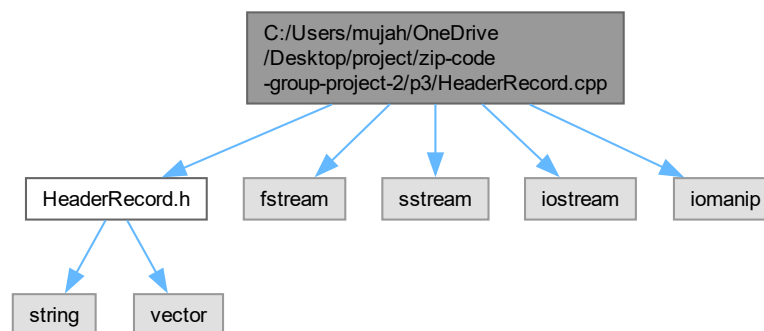
```
#include <fstream>
```

```
#include <sstream>
```

```
#include <iostream>
```

```
#include <iomanip>
```

Include dependency graph for HeaderRecord.cpp:



4.10 HeaderRecord.cpp

[Go to the documentation of this file.](#)

```

00001 #include "HeaderRecord.h"
00002 #include <fstream>
00003 #include <sstream>
00004 #include <iostream>
00005 #include <iomanip>
00006
00012 HeaderRecord::HeaderRecord()
00013     : headerSize(0)
00014     , recordSizeBytes(-1)
00015     , blockSize(512) // Default block size of 512 bytes
00016     , minBlockCapacity(0.5) // Default 50% minimum capacity
00017     , recordCount(40933) // Record count of input data
00018     , blockCount(3679)
00019     , fieldCount(6) // Default 6 as all used zipcode data has 6 parameters
00020     , primaryKeyField(0)
00021     , availListRBN(-1)
00022     , activeListRBN(-1)
00023     , isStale(false) {
00024     fileStructureType = "blocked_sequence_set";
00025     version = "1.0";
00026     sizeFormatType = "ASCII";

```



```

00027 }
00028
00034 void HeaderRecord::addField(const std::string& name, const std::string& schema) {
00035     FieldMetadata field;
00036     field.name = name;
00037     field.typeSchema = schema;
00038     fields.push_back(field);
00039     fieldCount = fields.size();
00040 }
00041
00048 bool HeaderRecord::writeHeader(std::ofstream& file) {
00049     if (!file.is_open()) {
00050         std::cerr << "Error: File stream is not open" << std::endl;
00051         return false;
00052     }
00053
00054     // Lambda to write length-indicated field
00055     auto writeField = [&file](const std::string& value) {
00056         std::string lengthStr = std::to_string(value.length());
00057         if (lengthStr.length() < 2) lengthStr = "0" + lengthStr;
00058         file << lengthStr << value << ",";
00059     };
00060
00061     // Write main header fields
00062     writeField(fileStructureType);
00063     writeField(version);
00064     writeField(std::to_string(headerSize));
00065     writeField(std::to_string(recordSizeBytes));
00066     writeField(sizeFormatType);
00067     writeField(std::to_string(blockSize));
00068     writeField(std::to_string(static_cast<int>(minBlockCapacity * 100)));
00069     writeField(indexFileName);
00070     writeField(indexFileSchema);
00071     writeField(std::to_string(recordCount));
00072     writeField(std::to_string(blockCount));
00073     writeField(std::to_string(fieldCount));
00074     writeField(std::to_string(primaryKeyField));
00075     writeField(std::to_string(availlistRBN));
00076     writeField(std::to_string(activeListRBN));
00077     file << (isStale ? "1" : "0") << "\n";
00078
00079     // Write field metadata
00080     for (const auto& field : fields) {
00081         writeField(field.name);
00082         writeField(field.typeSchema);
00083         file << "\n";
00084     }
00085
00086     return true;
00087 }
00088
00095 bool HeaderRecord::readHeader(const std::string& filename) {
00096     std::ifstream file(filename);
00097     if (!file.is_open()) {
00098         std::cerr << "Error: Unable to open file for reading: " << filename << std::endl;
00099         return false;
00100     }
00101
00102     std::string line;
00103     if (std::getline(file, line)) {
00104         std::stringstream ss(line);
00105
00106         // Lambda to read length-indicated field
00107         auto readField = [](std::stringstream& ss) -> std::string {
00108             std::string lenStr;
00109             lenStr.resize(2);
00110             if (!ss.read(&lenStr[0], 2)) return "";
00111
00112             if (!std::isdigit(lenStr[0]) || !std::isdigit(lenStr[1])) {
00113                 throw std::runtime_error("Invalid length indicator");
00114             }
00115
00116             int length = std::stoi(lenStr);
00117             std::string value;
00118             value.resize(length);
00119             if (!ss.read(&value[0], length)) return "";
00120
00121             if (ss.peek() == ',' ss.ignore());
00122             return value;
00123         };
00124
00125         try {
00126             // Read main header fields
00127             fileStructureType = readField(ss);
00128             version = readField(ss);
00129             headerSize = std::stoi(readField(ss));
00130             recordSizeBytes = std::stoi(readField(ss));

```

```

00131         sizeFormatType = readField(ss);
00132         blockSize = std::stoi(readField(ss));
00133         minBlockCapacity = std::stoi(readField(ss)) / 100.0;
00134         indexFileName = readField(ss);
00135         indexFileSchema = readField(ss);
00136         recordCount = std::stoi(readField(ss));
00137         blockCount = std::stoi(readField(ss));
00138         fieldCount = std::stoi(readField(ss));
00139         primaryKeyField = std::stoi(readField(ss));
00140         availListRBN = std::stoi(readField(ss));
00141         activeListRBN = std::stoi(readField(ss));
00142
00143         std::string staleStr;
00144         ss >> staleStr;
00145         isStale = (staleStr == "1");
00146
00147         // Read field metadata
00148         fields.clear();
00149         for (int i = 0; i < fieldCount && std::getline(file, line); i++) {
00150             std::stringstream fieldSS(line);
00151             FieldMetadata metadata;
00152             metadata.name = readField(fieldSS);
00153             metadata.typeSchema = readField(fieldSS);
00154             fields.push_back(metadata);
00155         }
00156     }
00157     catch (const std::exception& e) {
00158         std::cerr << "Error parsing header: " << e.what() << std::endl;
00159         return false;
00160     }
00161 }
00162
00163 file.close();
00164 return true;
00165 }

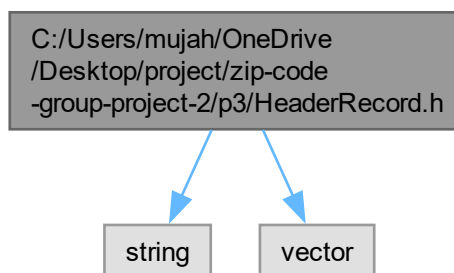
```

4.11 C:/Users/mujah/OneDrive/Desktop/project/zip-code-group-project-2/p3/HeaderRecord.h File Reference

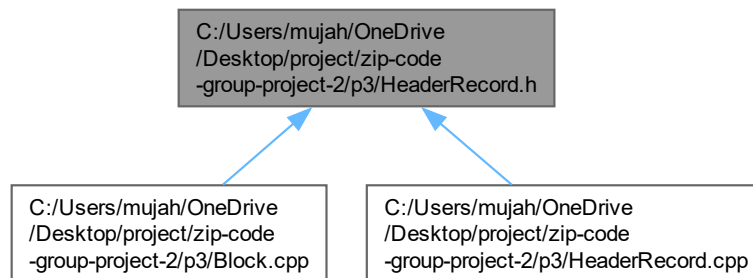
```
#include <string>
```

```
#include <vector>
```

Include dependency graph for HeaderRecord.h:



This graph shows which files directly or indirectly include this file:



Classes

- struct [FieldMetadata](#)
Metadata structure for field information in the header.
- class [HeaderRecord](#)
Manages the header record for blocked sequence set files.

4.12 HeaderRecord.h

[Go to the documentation of this file.](#)

```

00001 #ifndef HEADER_RECORD_H
00002 #define HEADER_RECORD_H
00003
00004 #include <string>
00005 #include <vector>
00006
00010 struct FieldMetadata {
00011     std::string name;
00012     std::string typeSchema;
00013 };
00014
00022 class HeaderRecord {
00023 public:
00024     HeaderRecord();
00025
00031     bool writeHeader(std::ofstream& file);
00032
00038     bool readHeader(const std::string& filename);
00039
00040     // Setters
00041     void setFileStructureType(const std::string& type) { fileStructureType = type; }
00042     void setVersion(const std::string& ver) { version = ver; }
00043     void setBlockSize(int size) { blockSize = size; }
00044     void setMinBlockCapacity(double capacity) { minBlockCapacity = capacity; }
00045     void setIndexFileName(const std::string& name) { indexFileName = name; }
00046     void setIndexSchema(const std::string& schema) { indexFileSchema = schema; }
00047     void setPrimaryKeyField(int field) { primaryKeyField = field; }
00048     void setAvailListRBN(int rbn) { availListRBN = rbn; }
00049     void setActiveListRBN(int rbn) { activeListRBN = rbn; }
00050     void setStaleFlag(bool flag) { isStale = flag; }
00051     void addField(const std::string& name, const std::string& schema);
00052
00053     // Getters
00054     std::string getFileStructureType() const { return fileStructureType; }
00055     std::string getVersion() const { return version; }
00056     int getBlockSize() const { return blockSize; }
00057     double getMinBlockCapacity() const { return minBlockCapacity; }
00058     std::string getIndexFileName() const { return indexFileName; }
00059     std::string getIndexSchema() const { return indexFileSchema; }
  
```

```

00060     int getPrimaryKeyField() const { return primaryKeyField; }
00061     int getAvailListRBN() const { return availListRBN; }
00062     int getActiveListRBN() const { return activeListRBN; }
00063     bool getStaleFlag() const { return isStale; }
00064     const std::vector<FieldMetadata>& getFields() const { return fields; }
00065
00066 private:
00067     std::string fileStructureType;
00068     std::string version;
00069     int headerSize;
00070     int recordSizeBytes;
00071     std::string sizeFormatType;
00072     int blockSize;
00073     double minBlockCapacity;
00074     std::string indexFileName;
00075     std::string indexFileSchema;
00076     int recordCount;
00077     int blockCount;
00078     int fieldCount;
00079     std::vector<FieldMetadata> fields;
00080     int primaryKeyField;
00081     int availListRBN;
00082     int activeListRBN;
00083     bool isStale;
00084 };
00085
00086 #endif // HEADER_RECORD_H

```

4.13 C:/Users/mujah/OneDrive/Desktop/project/zip-code-group-project-2/p3/HeaderTest.cpp File Reference

4.14 HeaderTest.cpp

[Go to the documentation of this file.](#)

```

00001 // #include "HeaderRecord.h"
00002 // #include <iostream>
00003 // #include <fstream>
00004 // #include <string>
00005 // #include <vector>
00006
00007 // /**
00008 //  * @brief Creates a sample CSV file with zip code data
00009 //  * @param filename Name of the file to create
00010 //  * @return true if successful, false otherwise
00011 //  */
00012 // bool createSampleCSV(const std::string& filename) {
00013 //     std::ofstream file(filename);
00014 //     if (!file.is_open()) {
00015 //         std::cerr << "Error: Unable to create sample CSV file: " << filename << std::endl;
00016 //         return false;
00017 //     }
00018
00019 //     // Write header row
00020 //     file << "zip_code,city,state,latitude,longitude\n";
00021
00022 //     // Write some sample data
00023 //     std::vector<std::string> sampleData = {
00024 //         "12345, Springfield, IL, 39.7817, -89.6501",
00025 //         "23456, Riverside, CA, 33.9533, -117.3961",
00026 //         "34567, Lakewood, OH, 41.4819, -81.7984",
00027 //         "45678, Maplewood, MN, 44.9530, -93.0275",
00028 //         "56789, Oakland, CA, 37.8044, -122.2711"
00029 //     };
00030
00031 //     for (const auto& record : sampleData) {
00032 //         file << record << "\n";
00033 //     }
00034
00035 //     file.close();
00036 //     return true;
00037 // }
00038
00039 // /**
00040 //  * @brief Creates a new file with header and copies CSV data
00041 //  * @param csvFile Original CSV file
00042 //  * @param outputFile Output file with header
00043 //  * @param header HeaderRecord object

```

```

00044 // * @return true if successful, false otherwise
00045 // */
00046 // bool createFileWithHeader(const std::string& csvFile, const std::string& outputFile, HeaderRecord&
    header) {
00047 //     // First write the header
00048 //     if (!header.writeHeader(outputFile)) {
00049 //         std::cerr << "Failed to write header to output file" << std::endl;
00050 //         return false;
00051 //     }
00052 //     std::ofstream file(outputFile);
00053 //     if (file.is_open()) {
00054 //         if (!header.writeHeader(file)) {
00055 //             // Handle error
00056 //         }
00057 //         // Continue using the file stream for other operations
00058 //         file.close();
00059 //
00060 //         // Now append the CSV data
00061 //         std::ifstream inFile(csvFile);
00062 //         std::ofstream outFile(outputFile, std::ios::app); // Open in append mode
00063 //
00064 //         if (!inFile.is_open() || !outFile.is_open()) {
00065 //             std::cerr << "Error opening files for copying data" << std::endl;
00066 //             return false;
00067 //         }
00068 //
00069 //         std::string line;
00070 //         getline(inFile, line); // Skip the CSV header
00071 //
00072 //         // Copy the rest of the file
00073 //         while (getline(inFile, line)) {
00074 //             outFile << line << "\n";
00075 //         }
00076 //
00077 //         inFile.close();
00078 //         outFile.close();
00079 //         return true;
00080 //     }
00081 //
00082 // int main() {
00083 //     const std::string csvFilename = "headerTest.csv";
00084 //     const std::string outputFilename = "headerTest_with_header.dat";
00085 //
00086 //     // Create sample CSV file
00087 //     std::cout << "Creating sample CSV file..." << std::endl;
00088 //     if (!createSampleCSV(csvFilename)) {
00089 //         std::cerr << "Failed to create sample CSV file" << std::endl;
00090 //         return 1;
00091 //     }
00092 //
00093 //     // Create and configure header
00094 //     std::cout << "Configuring header record..." << std::endl;
00095 //     HeaderRecord header;
00096 //
00097 //     // Set basic header information
00098 //     header.setFileStructureType("blocked_sequence_set");
00099 //     header.setVersion("1.0");
00100 //     header.setBlockSize(512); // Default block size
00101 //     header.setMinBlockCapacity(0.5); // 50% minimum capacity
00102 //     header.setIndexFileName("headerTest.idx");
00103 //     header.setIndexSchema("key:string,rbn:int");
00104 //
00105 //     // Add field definitions
00106 //     // header.addField("zip_code", "string(5)");
00107 //     // header.addField("city", "string(64)");
00108 //     // header.addField("state", "string(2)");
00109 //     // header.addField("latitude", "decimal(8,4)");
00110 //     // header.addField("longitude", "decimal(8,4)");
00111 //
00112 //     // Set primary key field (zip_code is field 0)
00113 //     header.setPrimaryKeyField(0);
00114 //
00115 //     // Create the output file with header
00116 //     std::cout << "Creating output file with header..." << std::endl;
00117 //     if (!createFileWithHeader(csvFilename, outputFilename, header)) {
00118 //         std::cerr << "Failed to create output file with header" << std::endl;
00119 //         return 1;
00120 //     }
00121 //
00122 //     // Verify by reading back the header
00123 //     std::cout << "\nVerifying header by reading it back..." << std::endl;
00124 //     HeaderRecord readHeader;
00125 //     if (readHeader.readHeader(outputFilename)) {
00126 //         std::cout << "Header verification successful!\n" << std::endl;
00127 //         std::cout << "File structure type: " << readHeader.getFileStructureType() << std::endl;
00128 //         std::cout << "Version: " << readHeader.getVersion() << std::endl;
00129 //         std::cout << "Block size: " << readHeader.getBlockSize() << std::endl;

```

```

00130 //      std::cout << "Index file: " << readHeader.getIndexFileName() << std::endl;
00131 //      std::cout << "Primary key field: " << readHeader.getPrimaryKeyField() << std::endl;
00132 //      std::cout << "Number of fields: " << readHeader.getFields().size() << std::endl;
00133
00134 //      std::cout << "\nField definitions:" << std::endl;
00135 //      const auto& fields = readHeader.getFields();
00136 //      for (size_t i = 0; i < fields.size(); i++) {
00137 //          std::cout << i << ": " << fields[i].name << " (" << fields[i].typeSchema << ")" << std::endl;
00138 //      }
00139 //  } else {
00140 //      std::cerr << "Failed to read back header" << std::endl;
00141 //      return 1;
00142 //  }
00143
00144 //      std::cout << "\nTest completed successfully!" << std::endl;
00145 //      return 0;
00146 //  }

```

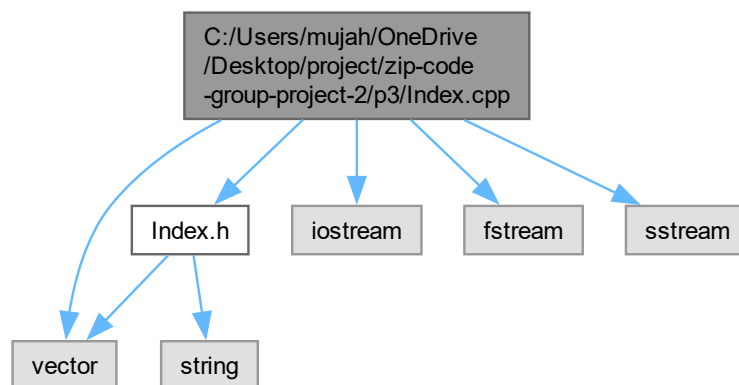
4.15 C:/Users/mujah/OneDrive/Desktop/project/zip-code-group-project-2/p3/Index.cpp File Reference

```

#include "Index.h"
#include <iostream>
#include <fstream>
#include <sstream>
#include <vector>

```

Include dependency graph for Index.cpp:



4.16 Index.cpp

[Go to the documentation of this file.](#)

```

00001 #include "Index.h"
00002 #include <iostream>
00003 #include <fstream>
00004 #include <sstream>
00005 #include <vector>
00006
00007 using namespace std;
00008
00016 vector<string> Index::split( const string& line, char delimiter ) {
00017     vector<string> tokens;

```

```

00018     string token;
00019     stringstream ss( line );
00020     while ( getline( ss, token, delimiter ) ) {
00021         tokens.push_back( token );
00022     }
00023     return tokens;
00024 }
00025
00036 void Index::processBlockData( const string& inputFileName, const string& outputFileName ) {
00037     ifstream inputFile( inputFileName );
00038     if ( !inputFile.is_open() ) {
00039         cerr << "Error: Could not open " << inputFileName << endl;
00040         return;
00041     }
00042
00043     ofstream outputFile( outputFileName );
00044     if ( !outputFile.is_open() ) {
00045         cerr << "Error: Could not open " << outputFileName << endl;
00046         return;
00047     }
00048
00049     outputFile << "Block,Zip Code\n";
00050     string line;
00051     getline( inputFile, line );
00052     line = "";
00053     while ( getline( inputFile, line ) ) {
00054         if ( line.empty() ) continue;
00055
00056         // Check if the line contains a colon; if not, skip it
00057         size_t colonPos = line.find( ':' );
00058         if ( colonPos == string::npos ) {
00059             // If no colon, perform additional validation
00060             vector<string> fields = split( line, ',' );
00061             if ( fields.size() < 2 || !isdigit( fields[ 0 ][ 0 ] ) ) {
00062                 // Skip malformed lines
00063                 continue;
00064             }
00065         }
00066         else {
00067             // Process lines with valid block:data format
00068             string block = line.substr( 0, colonPos ); // Block number
00069             string data = line.substr( colonPos + 1 ); // Rest of the data
00070
00071             // Split data into fields
00072             vector<string> fields = split( data, ',' );
00073
00074             // Extract zip codes (skip 5 fields for each)
00075             for ( size_t i = 0; i < fields.size(); i += 6 ) {
00076                 if ( !fields[ i ].empty() && isdigit( fields[ i ][ 0 ] ) ) {
00077                     string zipCode = fields[ i ];
00078                     outputFile << zipCode << "," << block << "\n";
00079                 }
00080             }
00081         }
00082     }
00083
00084     inputFile.close();
00085     outputFile.close();
00086
00087     cout << "Data successfully organized and saved to '" << outputFileName << "'.\n";
00088 }
00089

```

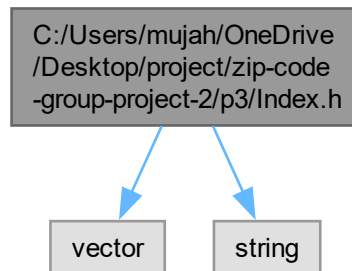
4.17 C:/Users/mujah/OneDrive/Desktop/project/zip-code-group-project-2/p3/Index.h File Reference

```

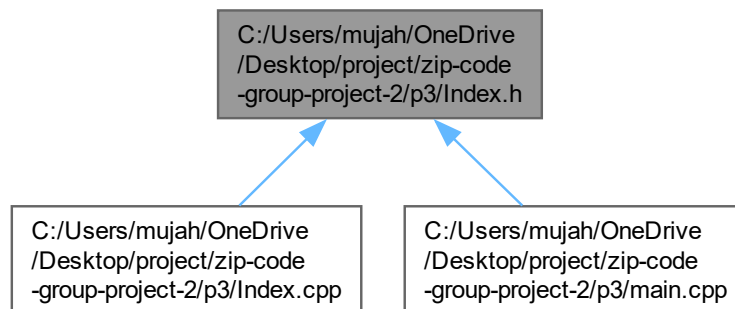
#include <vector>
#include <string>

```

Include dependency graph for Index.h:



This graph shows which files directly or indirectly include this file:



Classes

- class [Index](#)

4.18 Index.h

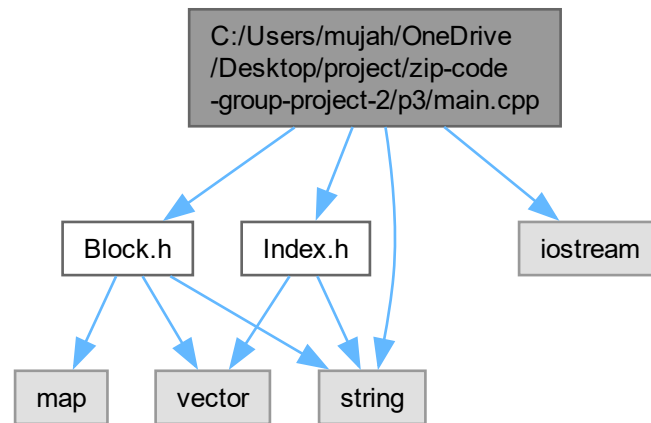
[Go to the documentation of this file.](#)

```

00001 #ifndef INDEX_H
00002 #define INDEX_H
00003 #include <vector>
00004 #include <string>
00005
00006 using namespace std;
00007
00008 class Index {
00009 public:
00017 void processBlockData( const string& inputFileName, const string& outputFileName );
00028 std::vector<std::string> split( const std::string& line, char delimiter );
00029 };
00030
00031 #endif
  
```


4.19 C:/Users/mujah/OneDrive/Desktop/project/zip-code-group-project-2/p3/main.cpp File Reference

```
#include "Block.h"
#include "Index.h"
#include <iostream>
#include <string>
Include dependency graph for main.cpp:
```



Functions

- `int main ()`
Main function to interactively manage blocks.

4.19.1 Function Documentation

4.19.1.1 main()

```
int main ()
```

Main function to interactively manage blocks.

This function provides an interactive menu-driven interface for managing and querying blocks. It performs the following steps:

1. Creates a block file from an input CSV file.
2. Parses the block file to populate the global `blocks` map.
3. Enters an infinite loop providing the user with the following options:
 - Dump all blocks in physical order.

- Dump all blocks in logical order.
- Query a specific block by its RBN.
- Exit the program.

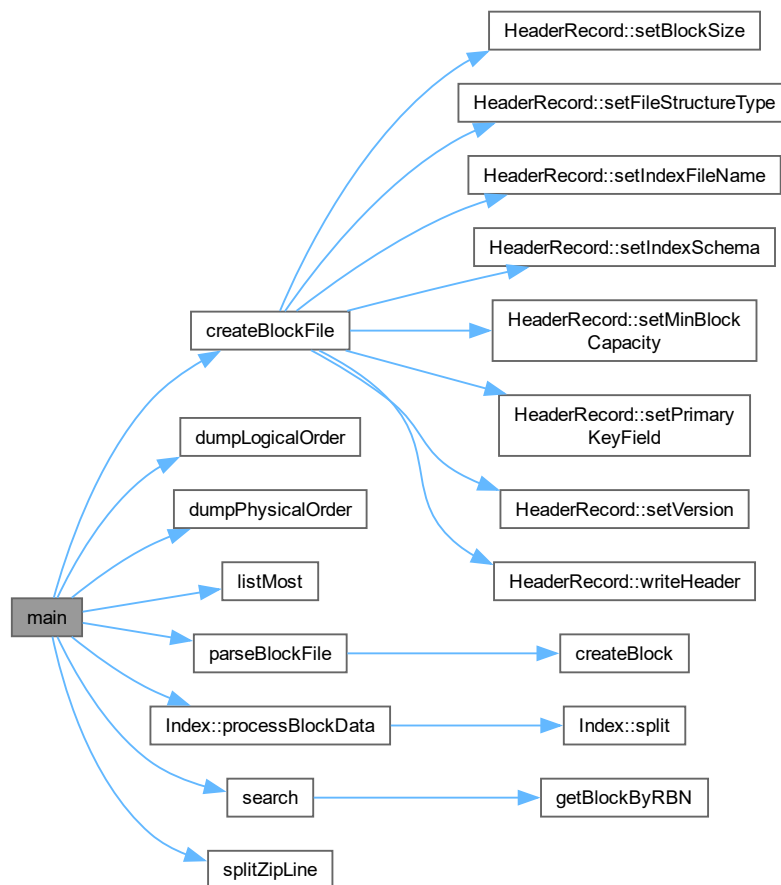
The user can query the details of a specific block by entering its RBN, including availability, records, and predecessor/successor RBNs.

Returns

int Exit code. Returns 0 if successful.

Definition at line 27 of file [main.cpp](#).

Here is the call graph for this function:



4.20 main.cpp

[Go to the documentation of this file.](#)

```

00001 #include "Block.h"
00002 #include "Index.h"

```

```

00003 #include <iostream>
00004 #include <string>
00005
00006 using namespace std;
00007
00027 int main() {
00028     string inputFile = "us_postal_codes.csv";
00029     string outputFile = "block.txt";
00030
00031     // Step 1: Create the block file from the input CSV
00032     if (createBlockFile(inputFile, outputFile)) {
00033         cout << "Block file created successfully.\n";
00034     } else {
00035         cerr << "Failed to create block file.\n";
00036         return 1;
00037     }
00038
00039     Index index;
00040     index.processBlockData( outputFile, "index.idx" );
00041     // Step 2: Parse the block file to populate the global blocks map
00042     parseBlockFile(outputFile);
00043
00044     // Step 3: Enter an infinite loop to provide a user menu
00045     while (true) {
00046         cout << "\n==== Block Management Menu =====\n";
00047         cout << "1. Dump Blocks in Physical Order\n";
00048         cout << "2. Dump Blocks in Logical Order\n";
00049         cout << "3. Query a Block by RBN\n";
00050         cout << "4. Get the most of each state.\n";
00051         cout << "5. Search for several zip codes.\n";
00052         cout << "6. Exit\n";
00053
00054         cout << "Enter your choice: ";
00055
00056         int choice;
00057         cin >> choice;
00058
00059         switch (choice) {
00060             case 1:
00061                 cout << "\n----- Physical Order Dump ----- \n";
00062                 dumpPhysicalOrder();
00063                 break;
00064
00065             case 2:
00066                 cout << "\n----- Logical Order Dump ----- \n";
00067                 dumpLogicalOrder();
00068                 break;
00069
00070             case 3: {
00071                 cout << "\nEnter the RBN of the block you want to query: ";
00072                 int RBN;
00073                 cin >> RBN;
00074
00075                 if (blocks.find(RBN) != blocks.end()) {
00076                     const Block& block = blocks[RBN];
00077                     cout << "\nDetails of Block RBN " << RBN << ":\n";
00078                     cout << "Available: " << (block.isAvailable ? "Yes" : "No") << "\n";
00079                     cout << "Records: ";
00080                     for (const string& record : block.records) {
00081                         cout << record << " ";
00082                     }
00083                     cout << "\nPredecessor RBN: " << RBN-1 << "\n";
00084                     cout << "Successor RBN: " << RBN+1 << "\n";
00085                 } else {
00086                     cout << "\nError: Block with RBN " << RBN << " not found.\n";
00087                 }
00088                 break;
00089             }
00090
00091             case 4: {
00092                 listMost();
00093                 cout << "\n----- State Most Data ----- \n";
00094                 break;
00095             }
00096
00097
00098             case 5: {
00099                 cout << "Please enter the zip codes you want!" << endl;
00100                 std::string text;
00101                 cin >> text;
00102                 auto result = splitZipLine(text);
00103                 for (const auto& str : result) {
00104                     search(str, "index.idx");
00105                 }
00106                 break;
00107             }
00108

```

```
00109         case 6:{
00110             cout << "Exiting the program. Goodbye!\n";
00111             return 0;
00112         }
00113
00114         default:
00115             cout << "Invalid choice. Please try again.\n";
00116             break;
00117     }
00118 }
00119 }
```

Index

- activeListRBN
 - HeaderRecord, 23
- add_record
 - Buffer, 10
- addField
 - HeaderRecord, 17
- availHeadRBN
 - Block.cpp, 46
 - Block.h, 58
- availListRBN
 - HeaderRecord, 23
- Block, 5
 - isAvailable, 6
 - predecessorRBN, 6
 - RBN, 6
 - records, 6
 - successorRBN, 6
- Block.cpp
 - availHeadRBN, 46
 - blocks, 46
 - createBlock, 38
 - createBlockFile, 39
 - dumpLogicalOrder, 40
 - dumpPhysicalOrder, 41
 - getBlockByRBN, 41
 - listHeadRBN, 46
 - listMost, 42
 - parseBlockFile, 43
 - search, 44
 - splitZipLine, 45
- Block.h
 - availHeadRBN, 58
 - blocks, 58
 - createBlock, 52
 - createBlockFile, 53
 - dumpLogicalOrder, 54
 - dumpPhysicalOrder, 55
 - listHeadRBN, 58
 - listMost, 55
 - parseBlockFile, 56
 - search, 57
- block_data
 - BlockBuffer, 8
- BlockBuffer, 7
 - block_data, 8
 - BlockBuffer, 8
 - unpack_block, 8
- blockCount
 - HeaderRecord, 23
- blocks
 - Block.cpp, 46
 - Block.h, 58
 - Buffer, 13
- blockSize
 - HeaderRecord, 24
- Buffer, 9
 - add_record, 10
 - blocks, 13
 - dump_blocks, 10
 - get_blocks, 10
 - parse_csv_line, 11
 - process_blocks, 11
 - read_csv, 12
 - records, 13
 - sort_records, 13
- C:/Users/mujah/OneDrive/Desktop/project/zip-code-group-project-2/p3/Block.cpp, 37, 47
- C:/Users/mujah/OneDrive/Desktop/project/zip-code-group-project-2/p3/Block.h, 51, 59
- C:/Users/mujah/OneDrive/Desktop/project/zip-code-group-project-2/p3/Buffer.cpp, 59, 60
- C:/Users/mujah/OneDrive/Desktop/project/zip-code-group-project-2/p3/Buffer.h, 62, 63
- C:/Users/mujah/OneDrive/Desktop/project/zip-code-group-project-2/p3/HeaderRecord.cpp, 64
- C:/Users/mujah/OneDrive/Desktop/project/zip-code-group-project-2/p3/HeaderRecord.h, 66, 67
- C:/Users/mujah/OneDrive/Desktop/project/zip-code-group-project-2/p3/HeaderTest.cpp, 68
- C:/Users/mujah/OneDrive/Desktop/project/zip-code-group-project-2/p3/Index.cpp, 70
- C:/Users/mujah/OneDrive/Desktop/project/zip-code-group-project-2/p3/Index.h, 71, 72
- C:/Users/mujah/OneDrive/Desktop/project/zip-code-group-project-2/p3/main.cpp, 73, 74
- city
 - RecordBuffer, 33
 - ZipCodeRecord, 35
- county
 - mostStorage, 29
- createBlock
 - Block.cpp, 38
 - Block.h, 52
- createBlockFile
 - Block.cpp, 39
 - Block.h, 53
- dump_blocks

- Buffer, 10
- dumpLogicalOrder
 - Block.cpp, 40
 - Block.h, 54
- dumpPhysicalOrder
 - Block.cpp, 41
 - Block.h, 55
- fieldCount
 - HeaderRecord, 24
- FieldMetadata, 14
 - name, 14
 - typeSchema, 14
- fields
 - HeaderRecord, 24
- fileStructureType
 - HeaderRecord, 24
- get_blocks
 - Buffer, 10
- getActiveListRBN
 - HeaderRecord, 18
- getAvailListRBN
 - HeaderRecord, 18
- getBlockByRBN
 - Block.cpp, 41
- getBlockSize
 - HeaderRecord, 18
- getFields
 - HeaderRecord, 18
- getFileStructureType
 - HeaderRecord, 18
- getIndexFileName
 - HeaderRecord, 18
- getIndexSchema
 - HeaderRecord, 18
- getMinBlockCapacity
 - HeaderRecord, 19
- getPrimaryKeyField
 - HeaderRecord, 19
- getStaleFlag
 - HeaderRecord, 19
- getVersion
 - HeaderRecord, 19
- HeaderRecord, 15
 - activeListRBN, 23
 - addField, 17
 - availListRBN, 23
 - blockCount, 23
 - blockSize, 24
 - fieldCount, 24
 - fields, 24
 - fileStructureType, 24
 - getActiveListRBN, 18
 - getAvailListRBN, 18
 - getBlockSize, 18
 - getFields, 18
 - getFileStructureType, 18
 - getIndexFileName, 18
 - getIndexSchema, 18
 - getMinBlockCapacity, 19
 - getPrimaryKeyField, 19
 - getStaleFlag, 19
 - getVersion, 19
 - headerSize, 24
 - headerSize
 - HeaderRecord, 24
 - Index, 26
 - processBlockData, 27
 - split, 28
 - indexFileName
 - HeaderRecord, 24
 - indexFileSchema
 - HeaderRecord, 25
 - isAvailable
 - Block, 6
 - isStale
 - HeaderRecord, 25
 - latitude
 - mostStorage, 29
 - RecordBuffer, 33
 - ZipCodeRecord, 35
 - listHeadRBN
 - Block.cpp, 46
 - Block.h, 58
 - listMost
 - Block.cpp, 42
 - Block.h, 55
 - longitude
 - mostStorage, 30
 - RecordBuffer, 33
 - ZipCodeRecord, 35

- main
 - main.cpp, 73
- main.cpp
 - main, 73
- minBlockCapacity
 - HeaderRecord, 25
- mostStorage, 29
 - county, 29
 - latitude, 29
 - longitude, 30
 - other, 30
 - state, 30
 - zip_code, 30
- name
 - FieldMetadata, 14
- other
 - mostStorage, 30
- parse_csv_line
 - Buffer, 11
- parseBlockFile
 - Block.cpp, 43
 - Block.h, 56
- predecessorRBN
 - Block, 6
- primaryKeyField
 - HeaderRecord, 25
- print_record
 - RecordBuffer, 32
- process_blocks
 - Buffer, 11
- processBlockData
 - Index, 27
- RBN
 - Block, 6
- read_csv
 - Buffer, 12
- readHeader
 - HeaderRecord, 19
- record_data
 - RecordBuffer, 33
- RecordBuffer, 31
 - city, 33
 - latitude, 33
 - longitude, 33
 - print_record, 32
 - record_data, 33
 - RecordBuffer, 32
 - state_id, 33
 - unpack_record, 32
 - zip_code, 34
- recordCount
 - HeaderRecord, 25
- records
 - Block, 6
 - Buffer, 13
- recordSizeBytes
 - HeaderRecord, 25
- search
 - Block.cpp, 44
 - Block.h, 57
- setActiveListRBN
 - HeaderRecord, 19
- setAvailListRBN
 - HeaderRecord, 20
- setBlockSize
 - HeaderRecord, 20
- setFileStructureType
 - HeaderRecord, 20
- setIndexFileName
 - HeaderRecord, 20
- setIndexSchema
 - HeaderRecord, 21
- setMinBlockCapacity
 - HeaderRecord, 21
- setPrimaryKeyField
 - HeaderRecord, 21
- setStaleFlag
 - HeaderRecord, 22
- setVersion
 - HeaderRecord, 22
- sizeFormatType
 - HeaderRecord, 26
- sort_records
 - Buffer, 13
- split
 - Index, 28
- splitZipLine
 - Block.cpp, 45
- state
 - mostStorage, 30
- state_id
 - RecordBuffer, 33
 - ZipCodeRecord, 35
- successorRBN
 - Block, 6
- typeSchema
 - FieldMetadata, 14
- unpack_block
 - BlockBuffer, 8
- unpack_record
 - RecordBuffer, 32
- version
 - HeaderRecord, 26
- writeHeader
 - HeaderRecord, 22
- zip_code
 - mostStorage, 30
 - RecordBuffer, 34
 - ZipCodeRecord, 35

ZipCodeRecord, [34](#)
 city, [35](#)
 latitude, [35](#)
 longitude, [35](#)
 state_id, [35](#)
 zip_code, [35](#)