Multidimensional Points Storage Project

Alex Kalfakakos

# Overview

The project was created in the C# programming language, backed by the .NET 6.0 framework, using additional preview features. Development almost entirely took place on Visual Studio 2022. Another compatible IDE is JetBrains Rider 2022.1.

The C# language was chosen for its organized, consistent and friendly syntax, while also providing enough low-level features for this project. The .NET runtime relies on the presence of a GC (**G**arbage **C**ollector), which might induce a cost in performance in the form of minor barely noticeable stutters during execution.

The project also made use of the following packages:

* **Garyon**, a package developed by me, containing useful extensions and common methods, to further speed up the development process.
* **System.Runtime.Experimental**, a package that contains additional experimental features of the language, available in the preview version that is used. The specific usage of that package is for generic math.
* **UnitsNet**, accompanied by **UnitsNet.NumberExtensions**, a package bundle that makes referring to units more convenient, including information units (byte and its multiples).
* **RoseLynn**, a package developed by me, providing convenient tools for developing code analysis components (such as source generators).
* **ManyConsole**, a framework for a more convenient development experience when setting up CLI applications.

A source generator was implemented in the process of automating the data header block’s property code, while also ensuring correctness of the offsets.

# Usage

The program is usable in two ways:

* Running commands on the SpatialAccessMethods.Main.exe file directly from the terminal, which is the guaranteed way of operating. To execute commands, open the terminal in the directory of the compiled binary of the SpatialAccessMethods.Main project (bin/Debug or bin/Release), and type any command like in the example:

.\SpatialAccessMethods.Main.exe rangeball -c "(1, 2)" -r 3

* Running the SpatialAccessMethods.MainInterface project’s executable that interfaces with the executable, allowing running multiple commands. It might fail to work in certain environments or permission contexts.

There is a help command that displays all available commands for the program. Typing .\SpatialAccessMethods.Main.exe on its own yields the same results.

The examples above assume using Windows. For other Operating Systems, there might be slight variation in the execution method. The executables might require elevated permissions for handling files.

# Disclaimers

The data formats that are used are integers, strings, Boolean flags and floating-point numbers. All floating-point numbers follow the IEEE-754 standard, and only float32 and float64 are candidates for usage. Thus, 4/8-byte floats refer to the IEEE-754-defined float32 or float64 respectively. Strings are always defined as null-terminated, to avoid variably sized records, in order to simplify indexing.

# Data Structures

For the purposes of this assignment, a binary heap and an R\*-tree were implemented. From the binary heap, the min heap version is used, as described below. All implemented data structures are designed with reliance on secondary storage.

## Min Heap

The stored values of the min heap can be easily stored in a list while preserving level and child order, and so is chosen in this implementation. They are sequentially stored in the file and can be directly indexed. During insertion and deletion, the number of blocks (actual IO operations are almost always fewer) that are accessed and written has a complexity. In most practical scenarios, due to using 32 KB blocks, a min heap instance will store less than 32,768 elements, resulting in a single block access. The multiple block writes will not be propagated to disk writes, and instead a single disk write will be performed, once the block gets swapped out of memory.

## R\*-tree

The implementation of the R\*-tree supports any number of dimensions, as the Point and Rectangle types enable. The approach for bulk loading was chosen to be based on the STR algorithm (sort-tile-recursive), with some slight variations in the structure of the final tree. Bulk loading is performed in memory, as the requested entries are accepted from in-memory values, with the loaded values being stored in the secondary storage.

# Record Storage

The database system supports storing records of marked locations of any number of dimensions, accompanied by a name. The ID is omitted for brevity, as it can be inferred by the record’s position in the file. This layout provides enough flexibility to expand on in the future, as more columns can be added on separate tables, further enabling abiding to commonly applied normal forms such as 3NF.

When a record is deleted, its (inferred) ID is freed up and ready to be reallocated by another record. Gaps formed by record deletion are stored in another file, in the form of a min heap. The IDs stored in that min heap are **not** strictly less than the current max ID. The next allocatable ID is the minimum ID in the gaps, or the next ID available if there are no gaps.

When the max allocated ID decreases, the immediately previous available IDs are not removed. This reduces the number of operations performed on the heap, as searching random values on the heap is O(n), thus reducing both CPU and IO time. The best approach to eliminating the excess unallocated IDs is eliminating entire levels of the heap, only maintaining sufficient entries for the max possible unallocated gaps, based on the max allocated ID and the entry count. It is intuitively (not actually proven) assumed that it is impossible to run into cases of eliminating sufficiently low IDs at the bottommost levels.

## Entry Record Layout

The record of the data table is laid out in 256 bytes and stores:

* A flag byte denoting whether the record is disposed (0), or alive (any non-zero value).
* *n* 4-byte floats, where *n* is the dimensionality of the records, with each float representing a coordinate of the location.
* A null-terminated UTF-16 string supporting up to *k* characters (not runes), **including** the null terminator, where:

The remaining byte of the 256-byte sector is unused, for convenience.

## R\*-tree Entry Node Layout

An entry in the R\*-tree node table has a size of 256 bytes and it contains:

* The ID of its parent, or 0 if the node is the root (4-byte integer).
* The dimensions of the rectangle the node reflects. For *n* dimensions, *2n* 4-byte floats are stored, resulting in *8n* bytes of size. They are stored in the order min0, min1, ..., minn-1, max0, max1, ..., maxn-1, where min and max are the minimum and maximum points of the rectangle, with the following identity obviously holding true:
* The type of the node, whether it’s a parent, or a leaf node (1-bit flag; 0 = parent, 1 = leaf).
* The number of children the node contains (7-bit integer, complementing the rest of the bytes from the flag, allowing up to 27 – 1 = 127 children).
* An array of child IDs using the remaining bytes, which are equal to:

Each child is an ID representing either the child node of the R\*-tree, or the record, stored as a 4-byte integer. This means that the maximum number of children that can be physically stored in the array is:

and is covered by the range of the 7-bit field storing the number of children in the array. The IDs are sorted in ascending order envisaging in optimizing IO operations, as child order doesn’t matter.

The node will be considered invalid if any of the following are true:

* The children number is greater than the maximum allowed for the node (62 – 2n).
* The node is a leaf node with no children (type/children control byte = 1 000 0000).

The root is always stored with an ID of 1. This specification is for data locality, to quickly refer to the first block of the file to retrieve the root node, and only have a single entry point. Otherwise, a

## Header Block

The header block in the record file contains the following information about the record file:

* Block size (4-byte integer)
* Block count (4-byte integer)
* Record count (4-byte integer)
* Max ID (4-byte integer), represents the highest allocated ID in the record table
* Max tree node ID (4-byte integer), represents the highest allocated ID in the R\*-tree
* Tree node count (4-byte integer)
* Dimensionality (1-byte integer)

The remaining 32,743 bytes remain unreserved. We could fit a whole snake game in there.

For the R\*-tree entry file, there is no header block.

For the ID gap min heap files, there is no header block. The entry count is stored in the beginning of the file, represented by a 4-byte integer. This also shifts the contained values by 4 bytes, resulting in the final node of the level to be on the very last position of the N-byte-sized block, assuming 4-byte integers are stored in the heap, to represent the IDs.

## Storage Communication

All data structures that load and handle their information follow the principles below:

* Load information from a block through the buffer controller (see analysis below).
* Any changes to the structure or the data will immediately apply the changes to the respective block, which changes will be preserved in memory and only get written to the disk during swapping or disposal.
* Block disposal from the main memory will only cause the block to be written to the disk if changed (dirty).

## Buffer Controller

The buffer controller handles the loaded and swapped buffers. Every file gets its own buffer controller, and all buffer controllers respond to the master buffer controller that guarantees the application does not consume too much memory by having multiple buffers. Buffers contain blocks of data that are mapped to specific positions in their respective files. Loaded blocks bear their respective ID and the RawData instance that reflects the array of bytes allocated for containing the disk data. The allocated arrays are not zero-initialized, as they will be initialized with disk data. Additionally, they are not pinned, as pinning is advised as avoidable.

The master buffer controller keeps all the blocks alive, and their allocated arrays. When blocks get swapped, from any child buffer controller, their allocated arrays that contain the data are swapped. This technique reduces allocation and freeing overheads, reducing strain on the GC.

1. An additional step could be taken and use an ever-expanding collection of such blocks, aggregating them all in a List<byte>, and rely on its built-in mechanism to expand and shrink, but it would impose the following adverse effects:

* The GC doesn’t like big object allocations, especially for persisting objects, risking a noticeable performance hit on loaded database query executions during a GC pass.
* The List<byte> doesn’t shrink itself unless explicitly asked, and more than necessary memory could be allocated, expecting to grow again in the future.
* Expanding the List<byte> requires adding a range of values, and a block size of 32 KB, as used in this specific assignment, would mean that every time a new memory block is allocated, 32,768 distinct bytes will be “added” to the list. Whatever optimization could take place, raw non-zeroing memory allocation is undoubtedly quicker.

When it comes to handling which blocks to keep alive and which to dispose, the policy is slightly advanced, there are two queues that preserve the order of loaded blocks, discriminated by their Dirty property. Non-dirty blocks take priority above dirty blocks to avoid writing to the storage upon block swapping. Heavy read-only operations have the potential to cause additional latency, however any more sophisticated improvements would require more extensive research that falls out of the scope of this assignment.

Each block stored in the buffers comes with its block ID, which is its sequential position in the file, and a RawData that provides access to the raw data. Due to its mutability, only a per-byte comparison can nearly guarantee whether the block has been affected since its last mutation on the storage, which in this implementation is since it was loaded. Imposing a memory usage penalty for this kind of purpose far outweighs the real benefit in this closed environment. Instead, all the places where the block of data is written come with the mutation notification. While this hurts maintainability, it’s the most balanced solution to ensure data integrity combined with IO operation minimization.

# Unit Testing

File operations are expensive in all ways, therefore MemoryStream was preferred for improved stability and speed during unit testing. The unit tests cover a good range of usage scenarios, and the randomly generated input data is probabilistically expected to catch all edge cases.

# Performance Overview

As asked, additional methods were implemented using naive techniques to execute requested queries. The performance was benchmarked using the Benchmark.NET package, and the benchmarking state environments were carefully crafted.

Below are some graphs comparing the performance of the R\*-tree-specialized operations, and their slower counterparts:

# References

1. **R-trees - A dynamic index structure for spatial searching**, by Antonin Guttman in California, US
2. **The R\*-tree: An Efficient and Robust Access Method for Points and Rectangles**, by Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider and Bernhard Seeger in Bremen, Germany
3. **A comparison of different R-tree construction techniques for range queries on neuromorphological data**, by Johan Grundberg and Axel Elmarsson in Stockholm, Sweden  
   <https://www.diva-portal.org/smash/get/diva2:1463267/FULLTEXT01.pdf>
4. **STR: A simple and efficient algorithm for R-tree packing**, by Scott T. Leutenegger, Jeffrey M. Edgington and Mario A. Lopez in NASA Langley Research Center, Hampton  
   <https://apps.dtic.mil/sti/pdfs/ADA324493.pdf>

The additional references were included to show the influence during the research for the algorithms to be applied.