Database System Design and Architecture

Introduction

The following describes the design and architecture of a client-server application. The storage of the data is in columnar form, therefore representing a Column Database. This database also retrieves and manipulates the data. The system uses customized Domain Specific Language to emulate the main features of a modern day column store. The different commands will take different paths depending on our algorithms. The different paths will access the different data structures and logical controls built into the system. The different data access patterns make use of scanning, indexing, relational joins, inserts, deletes, and updates to help us learn how to control Data Access Patterns, a skill that can be applied to any Key Value store.

Client and Server instantiation – Variable Pooling

As with any design, we start with an empty space, basically no physical entities, so our first task is to create the client and server executables. The Client and Server are the two main entry points for this system, which are represented by the executables named 'client.c', and 'server.c'. The 'client.c' represents the user, or the application, passing commands into our server and ideally this user should just be able to use our system. We emulate this abstraction with the console screen, which will send user/application commands to our system for parsing to execute the different operations depending on the command entered. We are given a LINUX environment, so our system will pass data over UNIX sockets. The data sent to our system is limited to fixed-point numerical data.

While the client is connected, it can send commands to the server in text format which are parsed, executed, and then a response is sent back to the client. The 'load' operation sends a comma-separated (CSV) file of values to the server for insertion, uses a binary format to optimize the data transfer over the socket. When the client disconnects, the server resumes waiting for a new client connection. The client can order the server to shut down. The server will then persist its databases to disk in binary format and load them back up during the next start. The client accepts a filename passed in from the command-line which launches and attaches to the process immediately. The client application exploits the stdio.h library by reading input from 'stdin' to read input specified from the command line arguments entered from the console to transmit it to the server. It then retrieves a response and displays it. In the case of the load command, it will also parse a CSV file and transmit it to the server in binary. The client and the server both send and receive data over the socket in small chunks of 4096 bytes to keep the system in balance.

Basic Functionality:

- Create (database, table, column, index)
- Insert a tuple
- Load multiple tuples from file
- Return tuples
- Filter tuples by value
- Shutdown and persist data
- Select (including batch selects)
- Select from pre-selected positions
- Fetch
- Print results
- Math (sum, average, min, max, add, subtract)
- Join
- Update
- Delete
- Indexing (sorted and btrees, clustered or unclustered)

Module organization

The server is composed of the following 13 modules:

- hashtable.c and list.c which implement a hashtable and linked list respectively. The hashtable is only used in joins. Linked lists are used for a collection that is not iterated over often and whose size is not predetermined (e.g. list of databases).
- memory.c implements a memory allocation and a memory re-allocation function that are used to allocate integer arrays. If the preprocessor variable ALIGNMENT is defined, the arrays are aligned to 64-byte boundaries (default).
- utils.c contains utility methods for logging information, errors and trace, reading/writing arrays from/to files in chunks, measuring time intervals and qsort functions that are used throughout the application.
- server_parse.c implements the parsing of all of the commands into the structures used internally to represent each operation and its arguments. The parse_* functions do not only separate the query string into names but also look up the relevant objects. Thus, it is where validation is performed; e.g. if a variable is not found, an “object not found” status is returned. This separates the concerns and keeps the processing functions and connecting logic simpler.
- server.c contains the main method accepts client connections, and handles each client connection where queries are received, processed and responses sent back to the client.
- db_result.c contains all the functions related to the Result type which is designed to hold results of any type (integer, long integer, and float) and size.
- db_manager.c contains all the functions related to the management of the database, such as initialization/shutdown, creation and retrieval of objects (databases, tables, columns, indexes).

- db_ops.c contains all the functions that implement the operations supported by the database (select, fetch, print, insert, delete, etc.).
- db_internal.c contains functions that are used internally by the database and are not exposed in the main header file (cs165_api.h), such as allocating/freeing memory for database objects and loading from/saving to disk.
- db_indexes.c contains the main entry points for index-related functions (creation and select) which then branch out to variants specific to the index type. There are four variants for each operation, one for each index and clustering type combination (sorted/clustered, sorted/unclustered, btree/clustered, btree/unclustered). db_idx_sorted.c and db_idx_btree.c contain all the relevant functions for each index type.

The client is composed of a single module:

- client.c provides the basic UNIX socket implementation. Uses stdin to receive input directly from the console to send to the server.

File organization

The server application will create a 'data' directory if it does not exist. Each database will then be stored inside its own directory and each table within that database in its own file. Example layout

- data
  - db1
    - tbl1.bin
    - tbl2.bin
    - tbl3.bin
    - ...
  - db2
  - ...

The tables are serialized in column order; column1 values first followed by column2 values, etc. Values are preceded by a small header which contains the name of the column and the index type. If the column is indexed, the values are followed by the index data which varies depending on the index.

The active database's name is saved as text in a file named 'active' in the folder named 'data' and loaded after all the database files have been loaded.

Generic design principles

The column store has been designed with the following logical pattern:

- Structures are always allocated with calloc to avoid problems caused by uninitialized values.

- Read-only strings such as database and column names are copied using strdup and freed during clean up.
- Fixed-size buffers for strings are only used to make serialization easier when saving to file or transmitting to/from a socket.
- Memory for integer arrays which are the essence of the store is managed using custom malloc/realloc replacements which enables us to use memory alignment or not, using a preprocessor definition.
- Range limits are initialized to the min/max of their respective data type if the value is optional, such that it is not required to further keep their state separately. For example, the left and right operands for the select operation, only one of which is required (but both can be defined) are initialized to INT_MIN and INT_MAX respectively.
- Binary search variants are implemented for all necessary specializations. For selects, the binary search is "smart" enough to with a greater than or equal to (>=) searches.

Batching implementation

To execute selects in parallel, a global variable is maintained, plus a linked list of operations, which is cleared at the end of each batch execution.

When a select command is received, if batching is disabled, the command is executed. If batching is enabled, it is added to the list of batch operations.

When the batch execute command is received, the list of batched selects is iterated and a thread is started for each select execution. The parameters passed to the thread function specify a pointer to the segment to process and the segment size. The item array is segmented in multiples of PAGESIZE.

Indexing implementation

Indexing in this project is simplified by not allowing updates to the indexes but only their creation. The store supports both sorted and b-tree indexes, as clustered or unclustered.

Sorted indexing

Sorted indexes maintain a sorted version of the column data such that binary search can be used to look for a range of data. In the unclustered version, the index maintains only an array of positions that refer to the original data, sorted by their respective data value. To sort this array, qsort_r is used, which allows to pass an extra argument (in this case the data array pointer) to the sort function. During select, a binary search variant is used which can refer to a different array for the data value is used to locate the range of selected elements, which are copied in bulk to the result.

If the index is clustered, the positions array from the unclustered version is followed by a sorted copy of the data for all columns, sequentially: [sorted positions] [sorted col1]....[sorted col n].

This allows us to keep all the sorted data tightly packed together which could optimize many use cases of the column store.

B-tree indexing

The b-tree design is bottom-up. First the sorted data is split into leaf nodes, each holding a maximum of PAGESIZE items. If the index is unclustered, the items are the positions of the respective data values. If it is clustered, much like the sorted version, each leaf holds a set of positions, followed by the sorted values for each column.

Once the leafs have been created, their min and max values are calculated, and they are linked to a predetermined amount of internal nodes specified by a preprocessor definition (BTREE_FANOUT, default=10). The internal nodes' min/max is calculated from the leafs' min/max and finally the internal nodes are connected to the root.

During select, the internal nodes are scanned for the appropriate node based on the left operand. If one exists, the appropriate node is search one level down, in the leafs. The leaf is then searched for qualifying items and the search continues with the next leaf in sequence. If the operands are no longer satisfied, the search is aborted. The type of search is controlled by a preprocessor directive (BTREE_SELECT_USE_LEAF_SCAN, scan if 1, binary search otherwise).

Joins implementation

The join operator can be executed as a nested loop join or a hash join. Hash joins are implemented both as simple hash joins and grace hash joins, as controlled by the JOIN_HASH_GRACE preprocessor definition. For the simple hash join, a simple hashtable implementation is used, whose hash function is simply the modulus of its size.

For the grace hash join, a slightly more complex hash function is used that is reported to have in a good statistical distribution.

Updates implementation

If inserts are performed before indexes are created, the values are appended to the end of the table. After index creation, any writes are written to a write-log which is kept as a key-value pair for inserts and an integer array for deletes (deleted positions).

Select was modified to scan deleted positions and remove them from the select result. The inserts array is also binary searched and any matching values are added to the result. Fetches were also updated to use the write log; if a position is beyond the end of the table/column in question, the position is looked for at the inserts log.

Deletes will first binary search the inserts log and if a value is found it is removed (result of an insert/update). If it is not found, it is appended to the deletes log.

Updates will scan the inserts log and update any matching values. If not found, a delete and an insert are inserted in the respective logs.

To be able to use binary search for the deletes/inserts logs, the logs are sorted after an item is appended. To delete values from these arrays, memmove is used.

Limitations

- Currently, only one client can be connected to the server.
- Only select operations are parallelized.
- Print results sends the output of the variable(s) to the client in text format. This is a remnant from the initial implementation that there was no time to change.
- Argument validation is basic. For example, for the select operation, there is no checking that at least one operand is defined, though if no operator is defined the whole set will be selected.
- Select on non-indexed columns for tables with a clustered index are not moved to the clustered index. E.g. if tbl1 contains col1, col2 and col3 and col1 has a clustered index (while col2 and col3 are not indexed), if a select on col2 is performed, it can take advantage of the col1 index.
- The inserts/deletes are not persisted to disk currently because the tests do not require it. Should this implementation be extended, it would re-create the index to include the updates from the log before persisting the data and index to disk.

Experimental design and results

To set up each of the experiments, there were four different sample sets of the data. The four sample sets were taken from data4.csv. The data was taken from data4.csv and trimmed down into four distinct amounts of tuples. The first data set of tuples were 250K, the second amount of data was 500K, the third sample was 750K, and finally the fourth sample set from data4.csv was 1 Million Tuples. Each of these samples were loaded into four separate .csv files named bench1.csv, bench2.csv, bench3.csv, and bench4.csv containing 250K, 500K, 750K, and 1 Million tuples respectively.

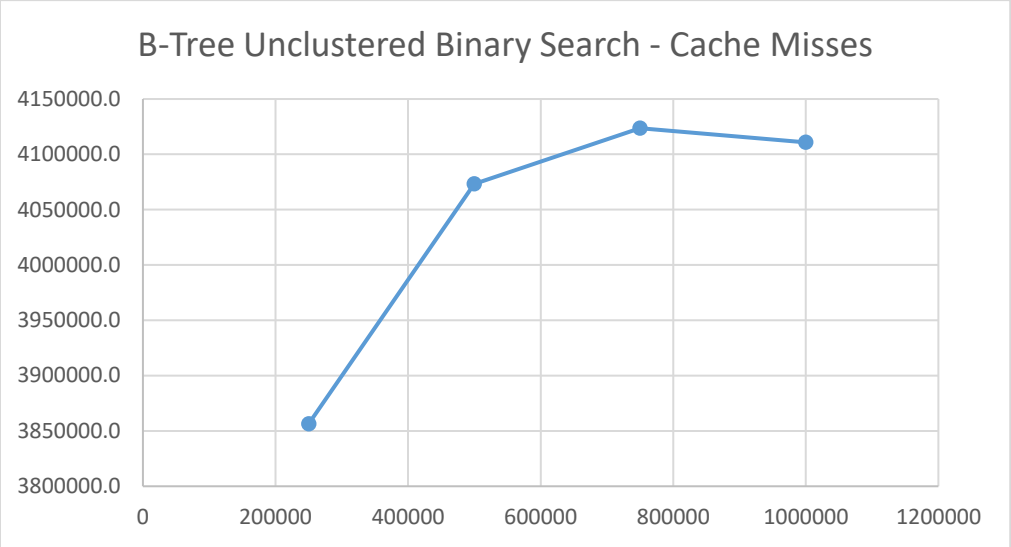
The shell file named bench.sh is now called from the terminal and connects the above perf attributes to our server executable, 'server.c'. The perf command was used to measure the performance of the system. The perf commands used were, 'perf stat –e branches, branch-misses, cache-misses, cache-references. A .sh file was used to take advantage of the LINUX operating system. Next a new terminal is called by the user, and then the customized Domain Specific Language DSL files are run to load our bench1.csv and the other three .csv files mentioned above. For one test a btree clustered index is created, namely, bench_btree_01.dsl, bench_btree02.dsl, bench_btree_03.dsl. The next tests are for the Hash and Grace Joins, bench_join_01, 02, 03 and 04. These tests differ only by the number of tuples entered for each.

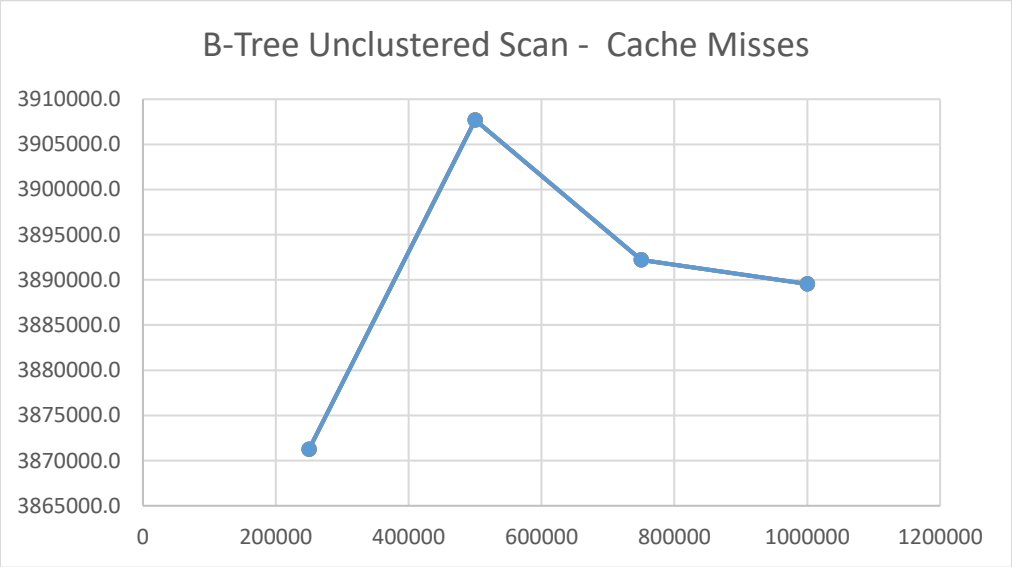
After the data is loaded a benchmark_start is called before the operation and then a benchmark_finish is called to stop the clock and then the time is recorded as well as the perf attributes input into our system through the bench.sh shell command discussed above. There are also two #Define variables to toggle between the Simple and Grace Hash Joins and the between the Btree Scan and Binary search Indexes.

Now that we have designed and built this main-memory column store, it is time to fine tune it. The goal of experimenting with tools such as perf is to help fine tune and minimize costs of data retrieval and management of that data retrieval. CPUs are changing, Memory costs are shifting, and todays methods, may simply not be good enough in the future, so an understanding of the demands on our system and an examination of the trade-offs that future technology may or not bring is an important goal of design and research. Understanding the nuances of this main memory column store helps to develop the proper skills to help manage and foresee these changes. The goal of these experiments and the building of these models is not to design them to pass today's tests, but to continually build on the skills of controlling memory accesses, increasing skills to move data up from the lowest level of memory up to the processors, and to efficiently use processors and technology of today and into the future.

In each of the experiments below, the Timed Performance in milliseconds, Branch Cache Misses, and Cache misses are examined to try and get an insight of how the current algorithms are working with the CPU to make correct predictions with the logic we give them and our goals as designers that we wish to achieve. The goal is to fine tune these trade-offs of expectations and changes in logic to achieve an optimal experience for the application or user interfacing with our system. The following output and graphs begin my adventure of this task of optimization for this 'Black Box' of abstractions.

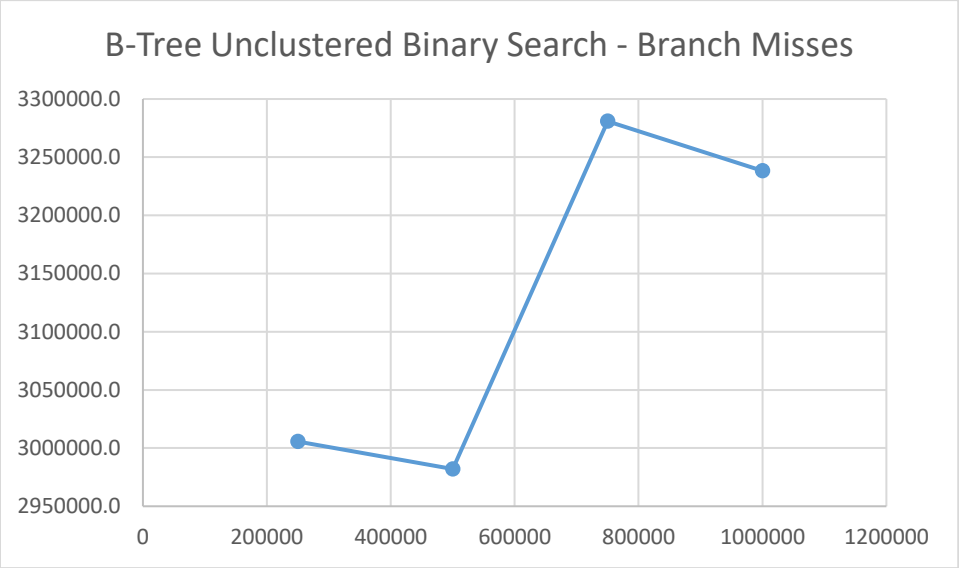
B-Tree Unclustered Scan vs B-Tree Unclustered Binary – X-axis is number of Tuples and Y-axis is number of Cache Misses.

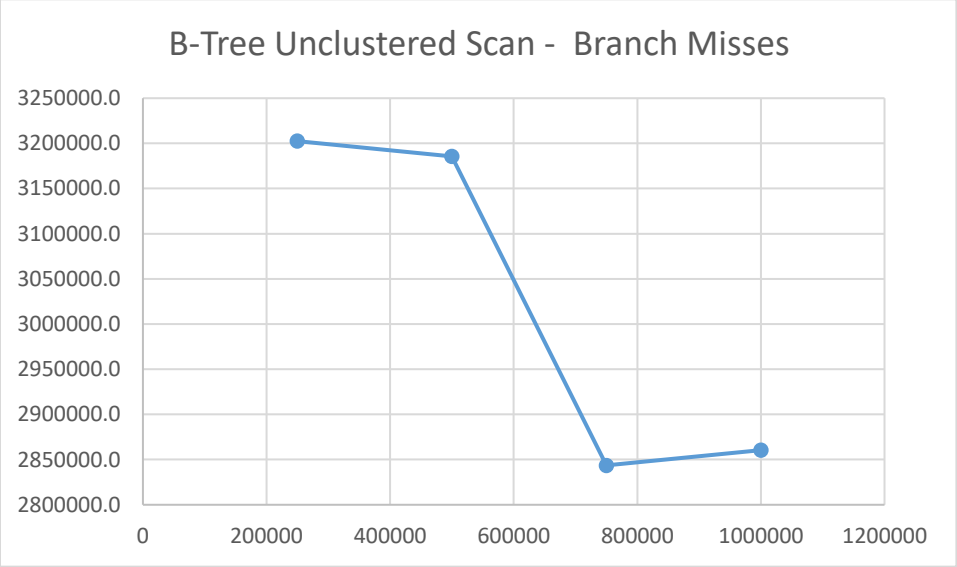




The above charts show that Cache misses scale fairly well, but the B-Tree Scan appear much more efficient.

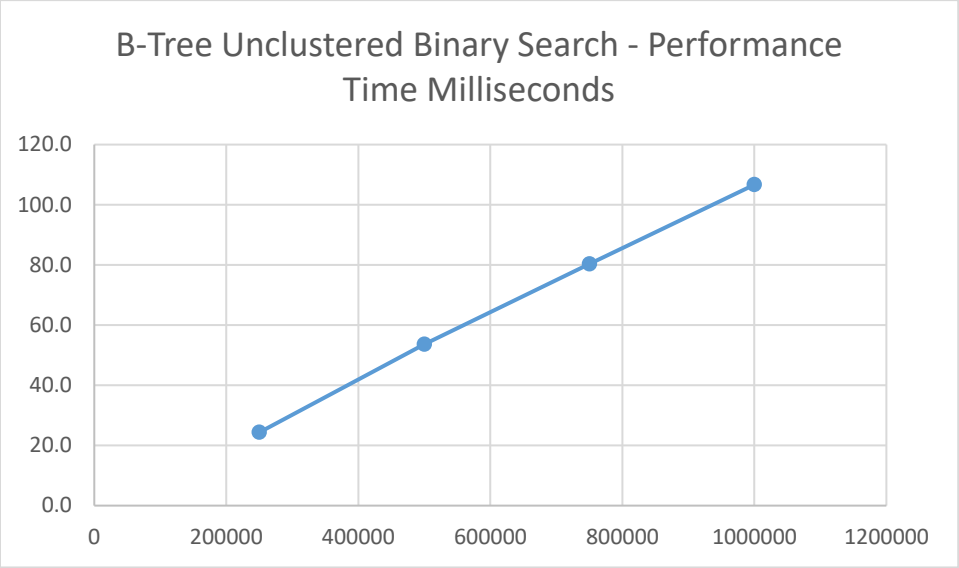
B-Tree Unclustered Scan vs B-Tree Unclustered Binary – X-axis is number of Tuples and Y-axis is number of Branch Misses.

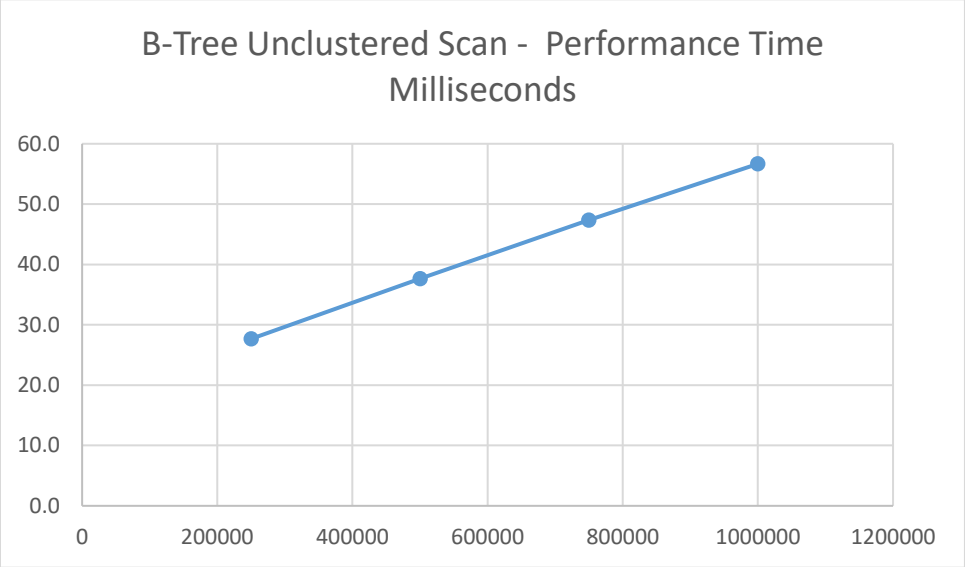




The B-Tree Unclustered Binary Search vs the Unclustered Scan appear to be inversely related to Branch misses as the tuples increase in size. The Scan works better in the case of Branch Misses, while there is an inflection point for the Binary Search at around 500K tuples. This is most likely the result of the implied fanout of a binary search of a log2 distribution, where the scan can take advantage of having a larger fanout.

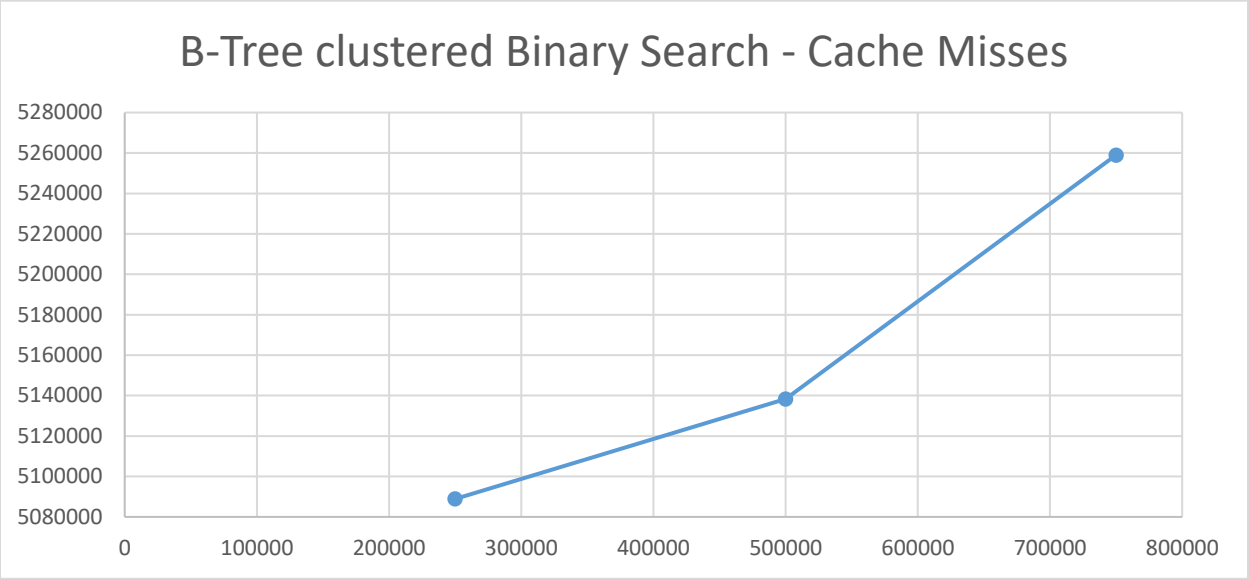
B-Tree Unclustered Scan vs B-Tree Unclustered Binary – X-axis is number of Tuples and Y-axis is number of Time in milliseconds.

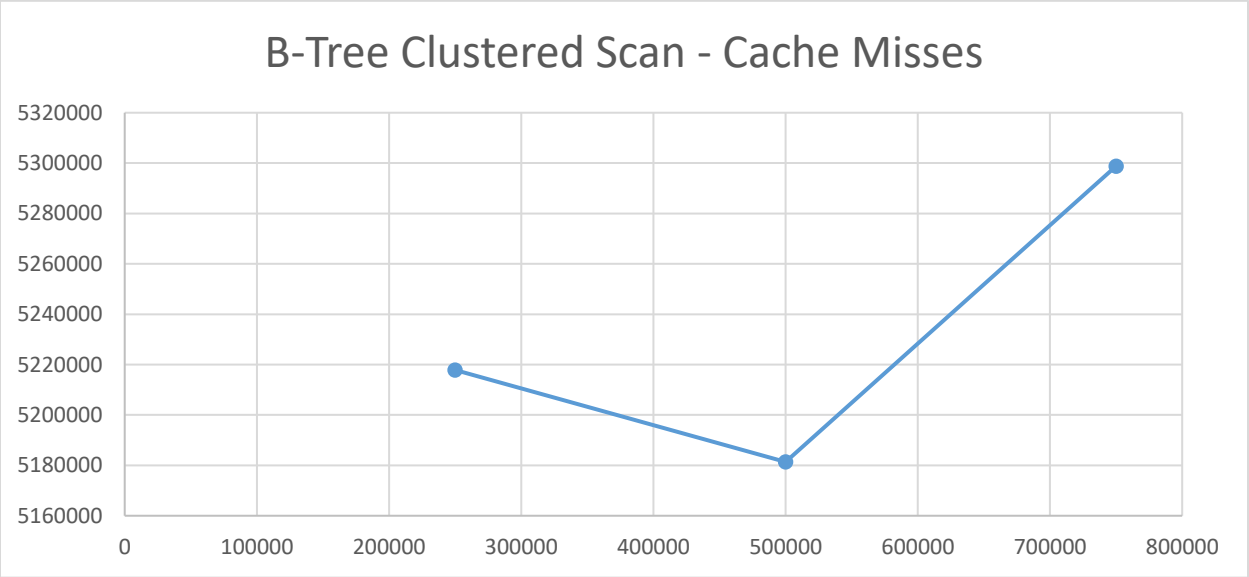




Performance time with the Unclustered Scan scales much better with a greater amount of tuples, whereas the Binary search is still fairly efficient with tuples of less than 350K.

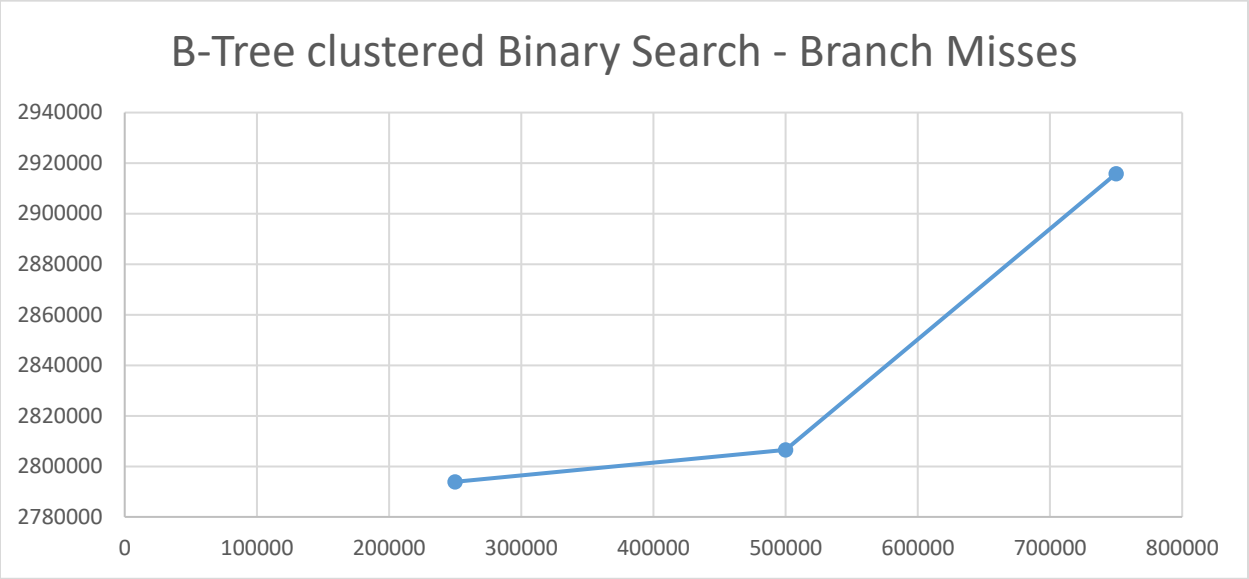
B-Tree Clustered Binary vs B-Tree Clustered Scan – X-axis is number of Tuples and Y-axis is number of Cache Misses.

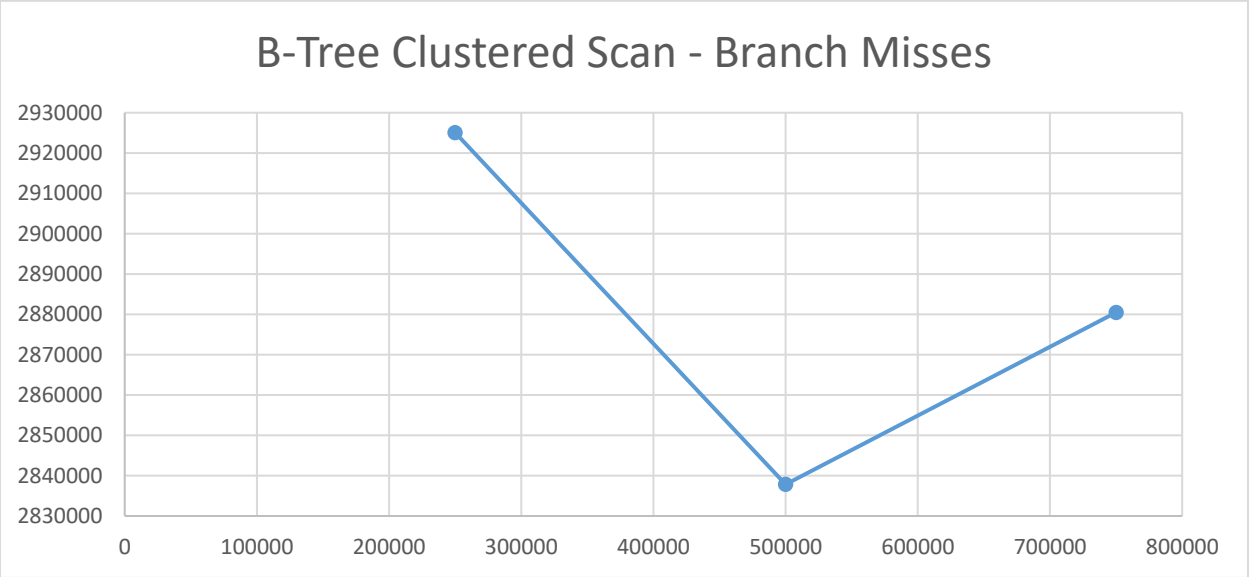




The B-Tree Clustered Scan appears similar, even though there is an inflection point the misses do not seem to be significantly different between the two. This indicates that some better logical controls or logic flows that can take advantage of CPU logic is needed.

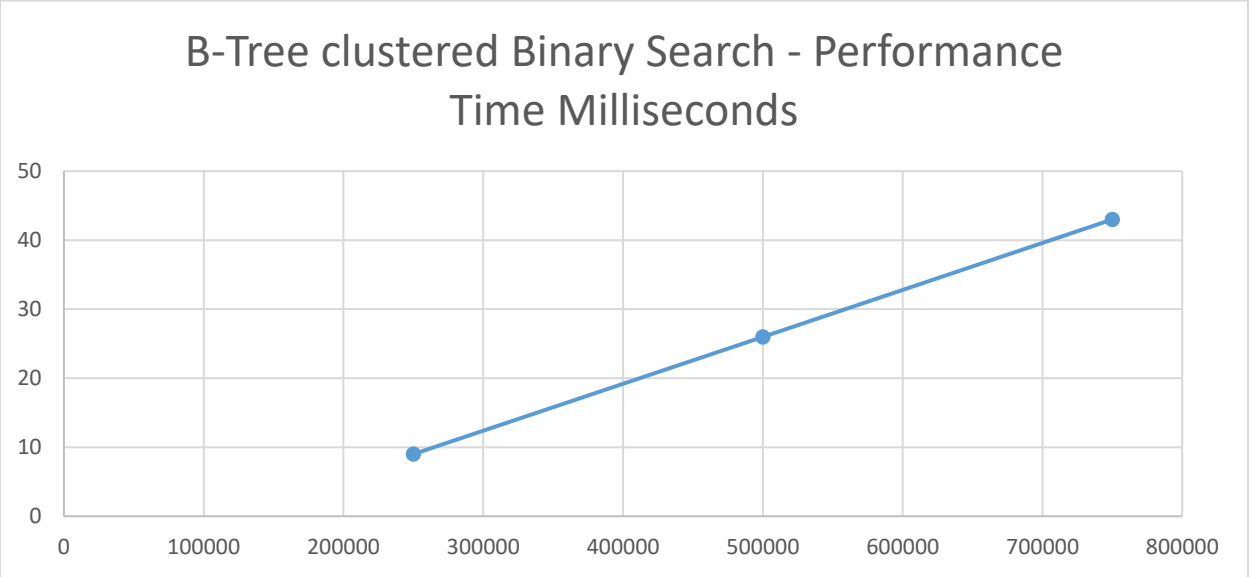
B-Tree Clustered Binary vs B-Tree Clustered Scan – X-axis is number of Tuples and Y-axis is number of Branch Misses.

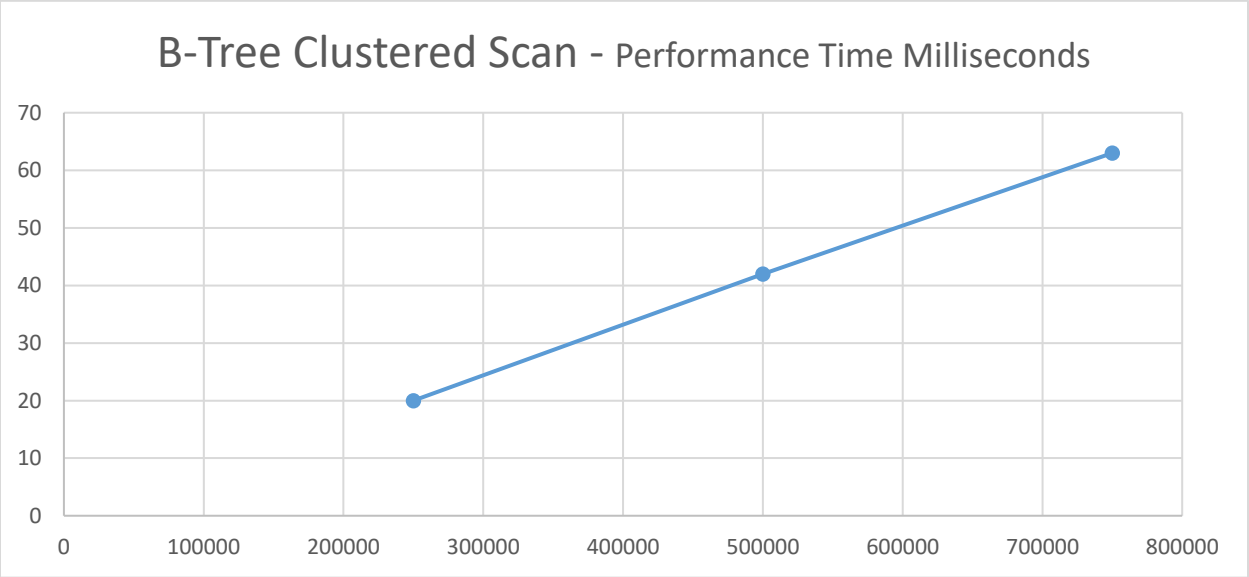




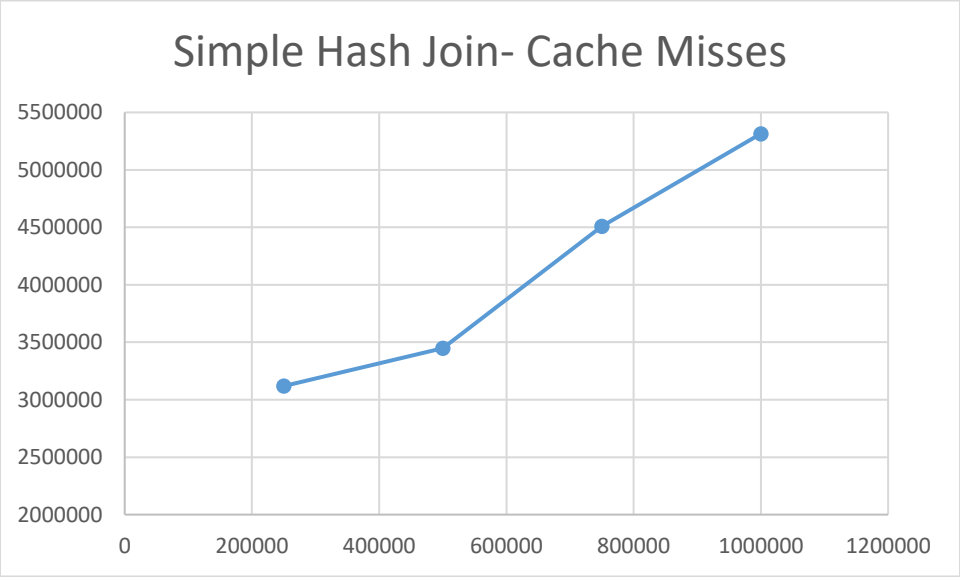
The B-Tree Clustered index seems to be the choice given the search is on the correct clustered index. The Binary version above ramps up much quicker than Clustered Scan, this again makes sense due to the implicit log2 criteria of the Binary Search.

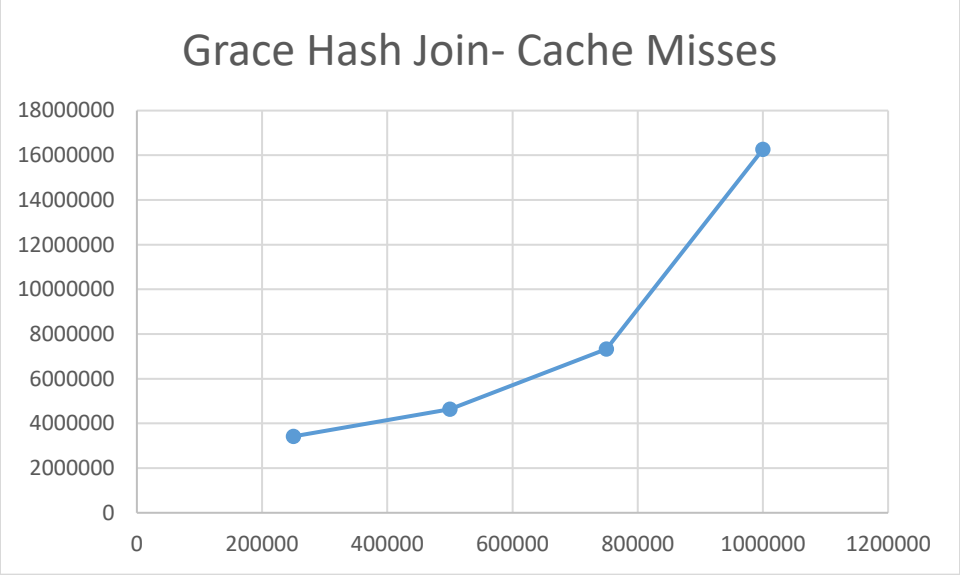
B-Tree Clustered Binary vs B-Tree Clustered Scan – X-axis is number of Tuples and Y-axis Time in Milliseconds.





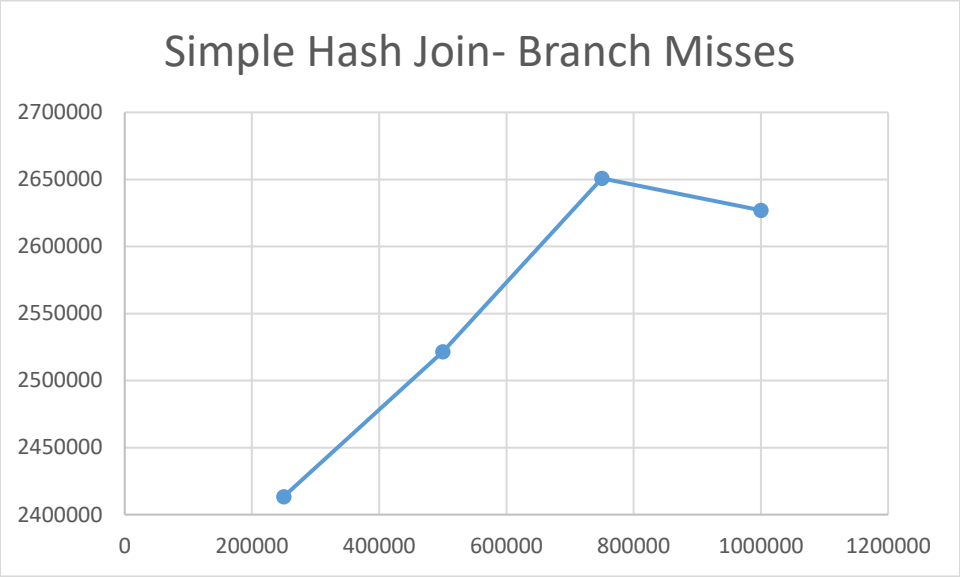
Simple Hash Join vs Grace Hash Join – X-axis is number of Tuples and Y-axis Cache Misses.

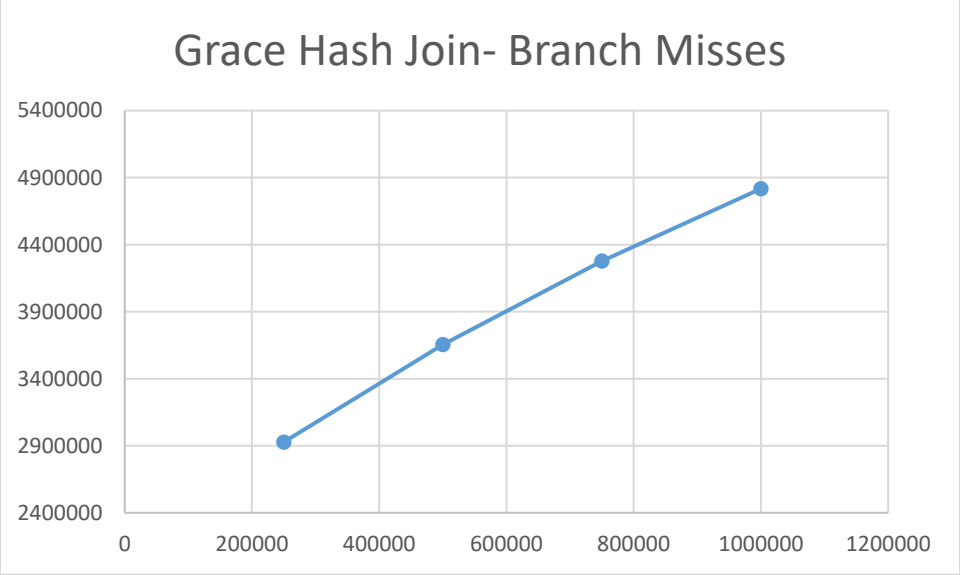




The above two graphs are indicating there is an issue with the Grace Hash Join Cache Misses. It appears to not be scaling at all with the increased size of Tuples, while the simple Hash Join appears to be more efficient, beginning to increase in slope at around 600K tuples.

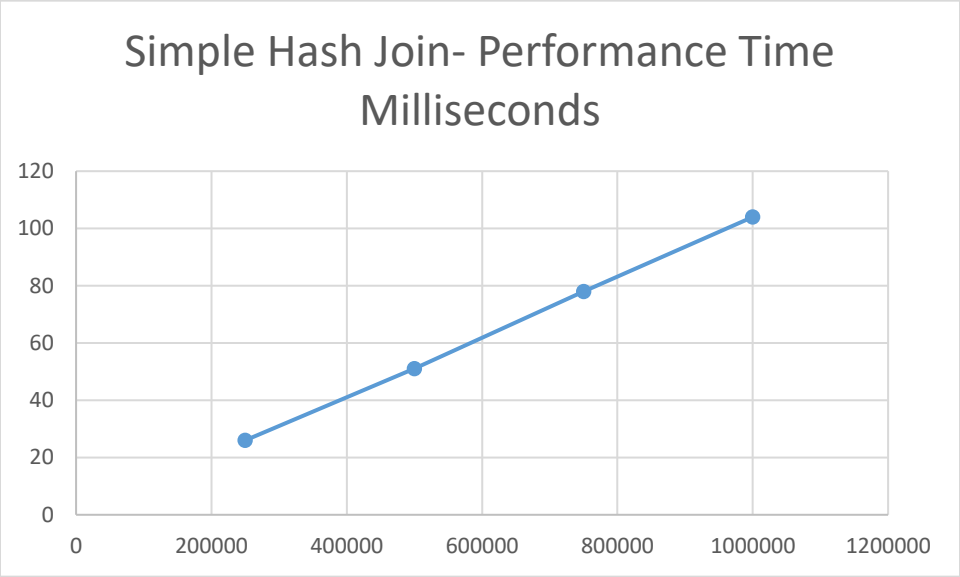
Simple Hash Join vs Grace Hash Join – X-axis is number of Tuples and Y-axis Branch Misses.

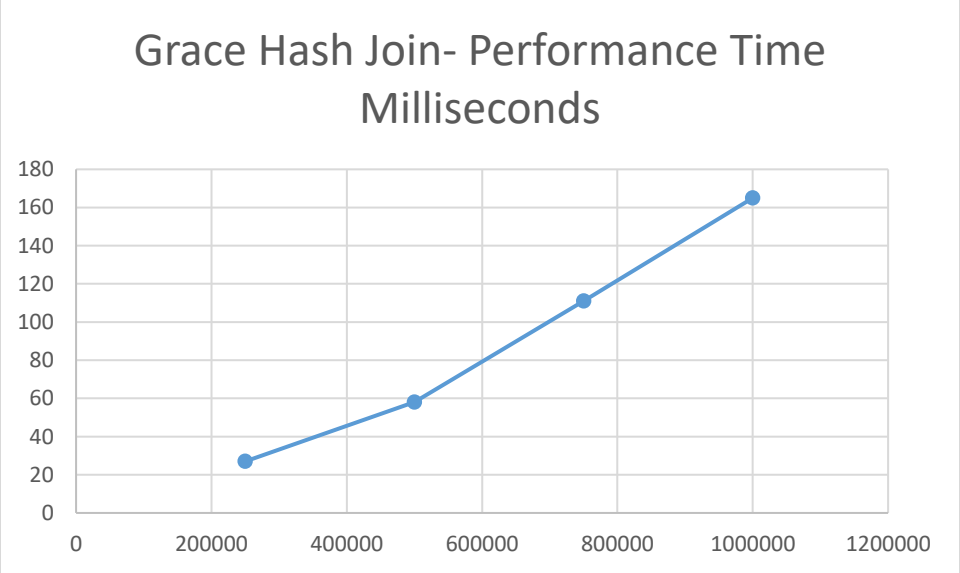




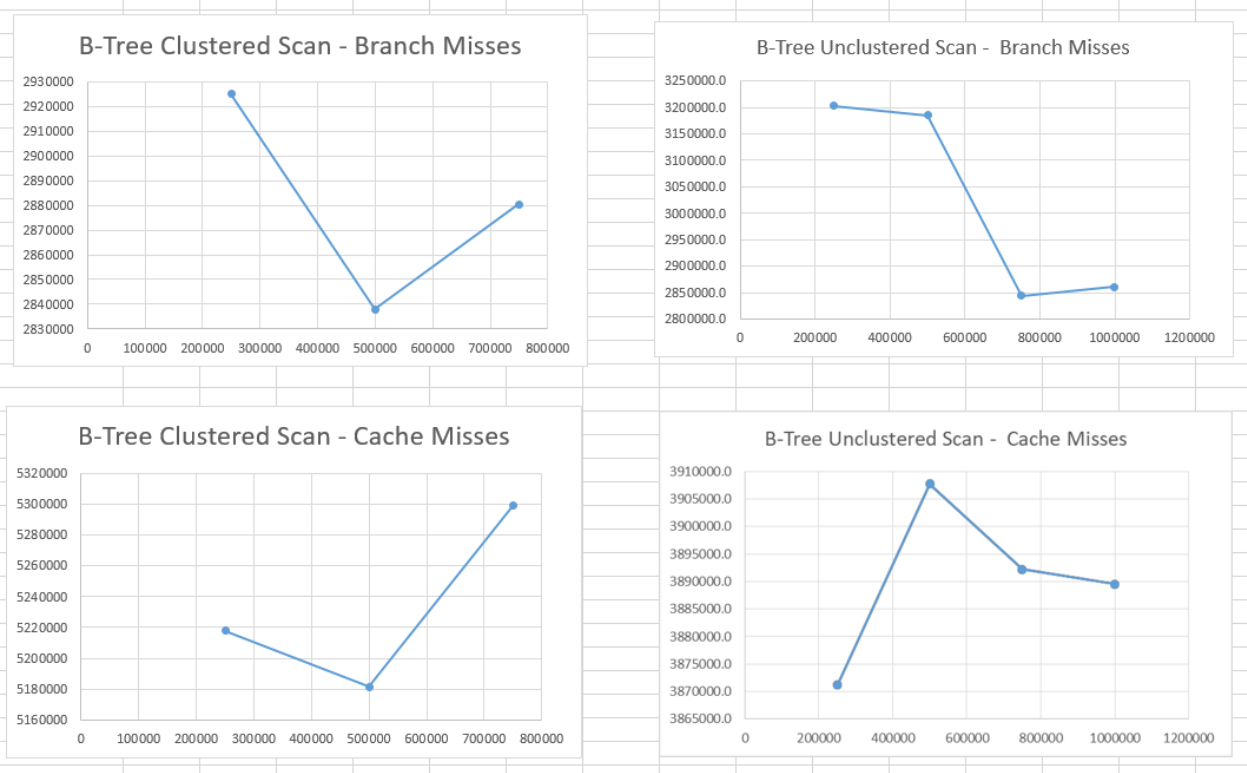
Similar to the Cache misses, the Simple Hash Join appears to perform much better with larger tuple size. This is probably an indicator that the Grace Hash Join would probably benefit and be faster if we executed the hashing of each partition in parallel.

Simple Hash Join vs Grace Hash Join – X-axis is number of Tuples and Y-axis Performance Time in Milliseconds.





A comparison of B-Tree Clustered Scan vs. B-Tree Unclustered Scan - X-axis is number of Tuples and Y-axis are Branch and Cache Misses.



Observation: As the Tuples grow, it appears that the Unclustered Scan has less Cache and Branch Misses, and this make logical sense since the more data pulled, then the higher frequency of matches should occur. The above also shows that the Clustered index has an inflection point for both the Cache and the Branch Misses, so we can conclude, from this graph that, more research could be done to build more intelligence into our algorithms that would optimize this fact, for instance given the tuple size within a certain range the logical path to follow would be a Clustered Scan to a certain point, and then after that inflection point it would be more optimal to choose the Unclustered Index path.

We found your repository at git@code.seas.harvard.edu:~edzielinski/cs165-2017-base/modernsystemexploration.git.

The tag for milestone 5 was on commit b6be37a76b1c46448ce0ef91aba9e9583e579373 on 2017-12-14 15:41:52 -0500.

The project compiled without any warnings.

Verifying the output of tests:
Test 01 output matches the expected output. Success! Runtime: 1102072 us
Test 02 output matches the expected output. Success! Runtime: 1897023 us
Test 03 output matches the expected output. Success! Runtime: 256845 us
Test 04 output matches the expected output. Success! Runtime: 5673 us
Test 05 output matches the expected output. Success! Runtime: 4247 us
Test 06 output matches the expected output. Success! Runtime: 5227 us
Test 07 output matches the expected output. Success! Runtime: 4688 us
Test 08 output matches the expected output. Success! Runtime: 5828 us
Test 09 output matches the expected output. Success! Runtime: 5907 us
Test 10 output matches the expected output. Success! Runtime: 1870063 us
Test 11 output matches the expected output. Success! Runtime: 6047 us
Test 12 output matches the expected output. Success! Runtime: 5911 us
Test 13 output matches the expected output. Success! Runtime: 6074 us
Test 14 output matches the expected output. Success! Runtime: 9149 us
Test 15 output matches the expected output. Success! Runtime: 9203 us
Test 16 output matches the expected output. Success! Runtime: 175528 us
Test 17 did not satisfy performance expectations (Runtime: 185012 us / Expected time: <122869 us) Error.

Disabling benchmarking because at least one test was not succesful.

Test 18 output matches the expected output. Success! Runtime: 1857530 us
Test 19 output matches the expected output. Success! Runtime: 2014415 us
Test 20 output matches the expected output. Success! Runtime: 5158 us
Test 21 output matches the expected output. Success! Runtime: 3491 us
Test 22 output matches the expected output. Success! Runtime: 5171 us
Test 23 output matches the expected output. Success! Runtime: 3545 us
Test 24 output matches the expected output. Success! Runtime: 1900919 us
Test 25 output matches the expected output. Success! Runtime: 2060522 us
Test 26 output matches the expected output. Success! Runtime: 5221 us
Test 27 output matches the expected output. Success! Runtime: 3461 us
Test 28 output matches the expected output. Success! Runtime: 4635 us
Test 29 output matches the expected output. Success! Runtime: 3619 us
Test 30 output matches the expected output. Success! Runtime: 1993460 us
Test 31 output matches the expected output. Success! Runtime: 4982733 us
Test 32 did not satisfy performance expectations (Runtime: 160439 us / Expected time: <49827 us) Error.
Test 33 output matches the expected output. Success! Runtime: 152345 us
Test 34 output matches the expected output. Success! Runtime: 221706 us
Test 35 output matches the expected output. Success! Runtime: 161330 us
Test 36 output matches the expected output. Success! Runtime: 3994 us
Test 37 output matches the expected output. Success! Runtime: 11257 us
Test 38 output matches the expected output. Success! Runtime: 7206 us
Test 39 output matches the expected output. Success! Runtime: 5066 us
Test 40 output matches the expected output. Success! Runtime: 5991 us
Test 41 output matches the expected output. Success! Runtime: 3739 us

We found your repository at git@code.seas.harvard.edu:~edzielinski/cs165-2017-base/modernsystemexploration.git.

The tag for milestone 5 was on commit 6c2e5492875568097ba22066060dac7637591a7a on 2017-12-09 09:02:06 -0

The project compiled without any warnings.

Verifying the output of tests:
Test 01 output matches the expected output. Success! Runtime: 1086914 us
Test 02 output matches the expected output. Success! Runtime: 1945050 us
Test 03 output matches the expected output. Success! Runtime: 306863 us
Test 04 output matches the expected output. Success! Runtime: 11310 us
Test 05 output matches the expected output. Success! Runtime: 10935 us
Test 06 output matches the expected output. Success! Runtime: 7866 us
Test 07 output matches the expected output. Success! Runtime: 7939 us
Test 08 output matches the expected output. Success! Runtime: 10956 us
Test 09 output matches the expected output. Success! Runtime: 8494 us
Test 10 output matches the expected output. Success! Runtime: 2017469 us
Test 11 output matches the expected output. Success! Runtime: 8932 us
Test 12 output matches the expected output. Success! Runtime: 9438 us
Test 13 output matches the expected output. Success! Runtime: 9059 us
Test 14 output matches the expected output. Success! Runtime: 10997 us
Test 15 output matches the expected output. Success! Runtime: 12172 us
Test 16 output matches the expected output. Success! Runtime: 252074 us
Test 17 did not satisfy performance expectations (Runtime: 192015 us / Expected time: <176451 us) Error.
Test 18 output matches the expected output. Success! Runtime: 2155451 us
Test 19 output matches the expected output. Success! Runtime: 2212977 us
Test 20 output matches the expected output. Success! Runtime: 8171 us
Test 21 output matches the expected output. Success! Runtime: 3530 us
Test 22 output matches the expected output. Success! Runtime: 7935 us
Test 23 output matches the expected output. Success! Runtime: 3620 us
Test 24 output matches the expected output. Success! Runtime: 1954426 us
Test 25 output matches the expected output. Success! Runtime: 2182659 us
Test 26 output matches the expected output. Success! Runtime: 8082 us
Test 27 output matches the expected output. Success! Runtime: 3739 us
Test 28 output matches the expected output. Success! Runtime: 7304 us
Test 29 output matches the expected output. Success! Runtime: 3470 us
Test 30 output matches the expected output. Success! Runtime: 2211913 us
Test 31 output matches the expected output. Success! Runtime: 16464066 us
Test 32 output matches the expected output. Success! Runtime: 162793 us
Test 33 output matches the expected output. Success! Runtime: 143843 us
Test 34 output matches the expected output. Success! Runtime: 150641 us
Test 35 output matches the expected output. Success! Runtime: 132601 us
Test 36 output matches the expected output. Success! Runtime: 5968 us
Test 37 output matches the expected output. Success! Runtime: 8331 us
Test 38 output matches the expected output. Success! Runtime: 9086 us
Test 39 output matches the expected output. Success! Runtime: 3640 us
Test 40 output matches the expected output. Success! Runtime: 10082 us
Test 41 output matches the expected output. Success! Runtime: 3601 us

Conclusions:

The above output seems to support our experimental results. If better parallelism and proper threading are implemented more efficiently, then the results would be more efficient. Also, the test seems to pass when the simple Hash Join is manually chosen with the #Define in the config.h. This is probably an indicator that the Grace Hash Join would probably benefit and be faster if we executed the hashing of each partition in parallel. This experiment is the beginning of a process, which perhaps, is never ending and will bring lots of research and design opportunities to these type of data access problems. As technology changes, data grows, and users/applications change, trade-offs will occur and the goal of these experiments and the building of these models is to not only design them to pass today's tests, but to continually build on the skills of controlling memory accesses, increasing skills to move data up from the lowest level of memory up to the processors, and to efficiently use processors and technology of today and into the future. The research and design opportunities appear to be a great challenge as technology evolves around us.