Cheetah Whitepaper

Executive Summary

The Cheetah search engine is the fastest dynamic sorted string search algorithm available. It is protected by both US Patent 10,262,081 and copyright 2014-2021. The only comparable search engine is the tree search, which by one estimate, is nearly one percent of the Gross World Product. Cheetah consistently runs four times faster than Tree. Cheetah search engine does insert, update, delete, select, greater than, less than, like, and between and the corresponding positional calls. This is all the basic database calls. Each of the eight calls require only one very simple, straight forward line of code per call. There is one field per Cheetah class, but multiple classes can easily be used for as many fields and tables as you need. Duplicates are handled internally. Cheetah can be very easily added as a third engine option in any existing database by a database company or can very easily be used outside of a database as a standalone search algorithm.

Search Algorithm

Background

Any computer science school teaches dozens of search algorithms but almost all of them have major drawbacks. Very few are excellent at large records for large number of records. There are three main search algorithms used, Tree, Hash, and Binary. Binary is not dynamic and cannot do insert, update, or delete. The entire array has to be completely resorted if any of the data changes. Therefore, it is never used in Databases but is sometimes used in standalone situations outside of a database when one knows in advance exactly what the data is and it never changes. Hash is a very seldom used alternate search engine in any database, but it is not sorted and therefore cannot do greater than, less than, like or between.

Tree is used in most searches in the world. Both Tree and Cheetah are complete solutions having insert, update, delete, select, greater than, less than, like, and between.

Function

Cheetah search engine looks at the individual letters in a string, but only looks at the minimum number of letters to make it unique. It does this is such a way as to easily handle when the beginning letters are all the same or similar. Even when only the very last letters of very large records are different, it only looks at enough letters to give each unique string its own slot, thus using very few letters even when the beginning letters are the same.

Cheetah dynamically loads groups of branches as needed so that the memory requirement is very close to the size and number of records and grows only as needed. Old slots are reused when records are deleted so that memory does not grow with a dynamic database. This is all done, with linked lists.

When a record is inserted, it looks to see if it is a duplicate string and if so, adds itself to this linked list of identical strings, or otherwise adds itself to the branches. As each record is inserted, it firsts looks at the first character to see which branch is associated with this letter. It then looks at the second letter to determine which branch belongs to the second letter, linked to the first letter. It continues this process until it finds no next branch. It then compares itself to this last find to determine whether to go before or after it. It adds itself to the sorted linked list at this position.

When the groups of branches are filled up, a new group of branches is added to memory. The number of branches in a group can be modified to suit. Each branch allows for characters between character 32 and character 127, or 96 characters. This range can be changed to suit. The size of the range affects the memory used. All memory tests were done with a 96-character range.

Insert

The eight standard database calls are incorporated into the Cheetah class. 'Insert' accepts a string and places the string into the class. It first does a 'Find' to find the first position that it is found. If it already exists, it adds it to the duplicate linked list for this string. If it is not found, it adds the string to the Cheetah branches. A 'Load' command is also available which allows an array of strings to be passed in then calls 'Insert' for each string.

Update

The 'update' call first does a 'find' on the string, then a 'delete' on this string, then an 'insert' of the new string. It changes the duplicates as well. The sample database program provided shows how to handle complex database calls for 'update' and other calls using the Cheetah class.

Delete

The 'delete' call removes the record from the branches, removes it from the sorted linked list, then removes the duplicates from the duplicate linked list. The previous and next record in the sorted linked list are then deleted and reinserted to clean up the branches to make them as if the original string had never been entered.

Select

The 'select' call does a 'find' to determine the first match. If no match is found, an empty array of integers is returned. If 'find' determines that there is at least one match. Then the duplicate linked list is converted into an array of integers to return a list of integers to correspond to the positions of the found strings.

Greater

The 'greater' call is for greater than. It does a 'between' call with the upper limit the highest possible string. All duplicates are added for each string found.

Less

The ‘less’ call is for less than. It does a ‘between’ call with the lower limit the lowest possible string, a blank string. All duplicates are added for each string found.

Like

The ‘like’ call does a basic string% which does a ‘find’ on the string then follows the sorted linked list upward until the highest string with the same beginning is found. More complicated likes can be done with external software but are not included in the class.

Between

The ‘between’ call is used in ‘greater’ and ‘less’ and is almost always what is actually required in a call. It does a ‘find’ on the lower string then marches up the sorted linked list until it reaches the higher string, including all duplicates. All ‘between’, ‘greater’, and ‘less’ calls always include the both end strings accordingly. They can be easily removed with external software from the returned lists. This allows greater, greater than or equal, less, less than or equal to be a handled with only one call.

Tests

All tests were run on a standard off the shelf laptop. It should be pointed out that the timings do vary because of standard background jobs on an off the shelf laptop, but the averages are consistent. Input files were generated with two parameters, the number of words and the number of rows. Each word is zero to seven characters between ‘a’ and ‘z’ with spaces in between. cheetah was tested with a character range of 96. This range is programable. Words per record is typically between 10 and 100. Up to a million words per record has been tested with similar results.

Intense 24/7 speed and functionality tests have been run on the Cheetah class and database class for years with no errors or warnings. The ‘insert’ is self-correcting but it has never had to correct itself in all of testing. The tables below show speed and memory tests. The test machine had thirty-two gigabyte of memory.

Table 1 shows the percent of time that Cheetah takes compared to tree, which is the only other complete search algorithm. Notice that the percent is almost always around 20 to 25 percent. That means that Cheetah is running four times faster, on average, than tree for all of these words and rows.

Table 2 shows the approximate maximum number of records for each type for the number of words. Cheetah does allow a fewer number of records than the others for small record sizes, but medium and large record sizes have no difference In memory requirements. This difference is partially due to number of groups of branches that are added at a time which is programable. A small memory footprint of one gigabyte was chosen for this test but should be consistent with larger memory footprints.

Table 3 compares Cheetah to all four search algorithms, even though binary and hash are not complete algorithms and are seldom used. The entry is the number of nanoseconds per record to do the search for a record on average for the run. Words between 10 and 100 by 10s are matched with rows at multiples of 100 from 10 to 10,000,000. ‘Type’, ‘words’, and ‘rows’ are self-explanatory. ‘Load’ is the number of

nanoseconds to initially load the strings. Note that load time for Cheetah can be minimized for small number of records by changing the load branches size, but it is about the same per record as tree for a large number or records. ‘Same’ is matching each row with an exact match one at a time. ‘Find’ is used in Cheetah to make it apples to apples with the other search algorithms. The only difference between ‘find’ and ‘select’ is that select gets all the duplicates. The percentages between the two are always within a couple of percent. ‘More’ adds a letter ‘a’ to each string so that it is not found. ‘Less’ takes off the last character so that it is not found. The summary of table 3 is found in table 4. The only drawback of Cheetah is that load always takes longer than the other search algorithms, but load is only done once and ‘find’ or ‘select’ and the other calls are done thousands of times. Table 4 shows that ‘same’ is always faster for Cheetah compared to tree but slower than binary or hash. When the string is not found in ‘more’ or ‘less’, Cheetah is almost always faster than tree and comparable to binary and hash. Cheetah does its best with large records (words) and large number of records (rows).

Table 1 (percent speed between Tree and Cheetah)

	words									
rows	**10**	**20**	**30**	**40**	**50**	**60**	**70**	**80**	**90**	**100**
200000	37	28	26	24	27	31	32	29	32	28
400000	26	24	24	21	24	27	26	24	26	25
600000	26	27	23	22	24	25	24	27	24	22
800000	25	22	21	22	23	22	23	23	23	23
1000000	24	21	25	21	24	24	22	22	24	22
1200000	23	26	21	21	23	22	22	22	22	21
1400000	23	21	21	23	24	22	22	22	24	22
1600000	22	21	21	29	24	23	22	22	22	23
1800000	23	24	24	21	29	24	24	23	21	20
2000000	19	22	21	21	21	21	21	21	20	24
2200000	25	21	21	21	21	23	22	23	21	20
2400000	31	25	30	24	21	24	23	23	24	21
2600000	22	21	24	21	21	23	24	23	23	23
2800000	22	21	24	21	23	20	24	23	22	22
3000000	22	21	21	21	23	24	23	20	23	20
3200000	24	21	21	23	23	23	20	23	20	
3400000	21	21	23	23	23	20	23	22	20	
3600000	22	20	23	23	23	20	22	23		
3800000	29	22	21	23	20	22	23	20		
4000000	25	21	23	23	21	21	22			
4200000	25	24	23	22	20	22	21			

4400000	25	25	23	23	21	23	22			
4600000	24	24	24	23	23	22				
4800000	24	20	24	25	23	23				
5000000	26	24	24	22	23	23				
5200000	24	24	24	21	22					
5400000	24	23	24	21	23					
5600000	25	22	31	23	21					
5800000	24	23	23	23	21					
6000000	25	25	24	21	22					
6200000	23	23	24	21	24					
6400000	24	23	31	23						
6600000	24	23	22	23						
6800000	35	24	22	24						
7000000	24	25	21	24						
7200000	24	25	21	22						
7400000	27	24	23	23						
7600000	24	24	23	23						
7800000	25	22	23							
8000000	25	24	23							
8200000	25	24	23							
8400000	25	24	23							
8600000	37	24	23							
8800000	24	32	23							
9000000	24	31	23							
9200000	25	24	23							
9400000	24	22	23							
9600000	24	22	23							
9800000	24	23	23							
10000000	24	23	23							

Table 2 (records)

thousands of records at one gigabyte memory										
words	**10**	**20**	**30**	**40**	**50**	**60**	**70**	**80**	**90**	**100**
memory										
file	4330	1800	970	900	800	700	600	500	400	300
type										
1 cheetah	1200	1100	930	870	800	700	600	500	400	300
2 tree	3290	1800	970	900	800	700	600	500	400	300
3 binary	4160	1800	970	900	800	700	600	500	400	300
4 hash	3980	1800	970	900	800	700	600	500	400	300

Table 3 (nanoseconds)

type	**words**	**rows**	**load**	**same**	**more**	**less**
Cheetah:	10	10	702406	976	1645	1233
Tree:	10	10	58705	1439	1799	2210
Binary:	10	10	24880	719	3238	668
Hash:	10	10	3290	359	925	925
type						
Cheetah:	10	1000	13471	225	759	636
Tree:	10	1000	721	957	1288	1015
Binary:	10	1000	1238	460	667	454
Hash:	10	1000	819	199	551	199
type						
Cheetah:	10	100000	2531	263	425	302
Tree:	10	100000	478	478	792	651
Binary:	10	100000	715	225	427	280
Hash:	10	100000	274	74	348	204

type						
Cheetah:	10	10000000	3534	666	851	725
Tree:	10	10000000	2599	2648	3035	2981
Binary:	10	10000000	761	216	341	287
Hash:	10	10000000	353	78	296	245
type						
Cheetah:	40	10	108928	257	462	257
Tree:	40	10	1439	308	514	308
Binary:	40	10	1387	51	308	51
Hash:	40	10	976	0	308	205
type						
Cheetah:	40	1000	1676	61	181	86
Tree:	40	1000	209	145	388	215
Binary:	40	1000	270	69	203	93
Hash:	40	1000	251	6	245	153
type						
Cheetah:	40	100000	1450	191	337	229
Tree:	40	100000	626	508	800	686
Binary:	40	100000	382	239	377	272
Hash:	40	100000	341	23	415	317
type						
Cheetah:	40	10000000	10052	924	1878	748
Tree:	40	10000000	2877	2932	10471	2245
Binary:	40	10000000	900	286	394	296
Hash:	40	10000000	532	93	757	334
type						
Cheetah:	70	10	185265	102	565	154
Tree:	70	10	616	205	411	154
Binary:	70	10	2210	102	462	205
Hash:	70	10	2107	51	514	308
type						
Cheetah:	70	1000	1700	68	285	110
Tree:	70	1000	131	113	399	192
Binary:	70	1000	202	66	305	119
Hash:	70	1000	346	10	431	271

type						
Cheetah:	70	100000	1459	211	415	252
Tree:	70	100000	515	554	989	752
Binary:	70	100000	375	229	443	301
Hash:	70	100000	419	28	565	409
type						
Cheetah:	100	10	108568	51	411	51
Tree:	100	10	102	1696	462	154
Binary:	100	10	308	0	411	102
Hash:	100	10	462	0	616	411
type						
Cheetah:	100	1000	1751	78	469	139
Tree:	100	1000	180	111	486	205
Binary:	100	1000	176	77	343	126
Hash:	100	1000	423	8	725	385
type						
Cheetah:	100	100000	1586	243	515	280
Tree:	100	100000	559	559	1061	831
Binary:	100	100000	409	231	530	338
Hash:	100	100000	509	27	725	532
type						

Table 4 (percent)

%compared to cheetah			
type	**same**	**more**	**less**
tree	100	92	92
binary	28	50	50
hash	0	50	57