Summer 2020

Replacement Policies in Semantic Database Caching

Colin Monaghan

Replacement Policies in Semantic Database Caching

Colin Monaghan
Professor David Chiu
University of Puget Sound

Abstract

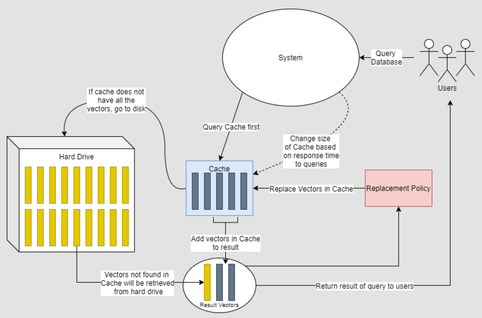
Efficient data management is vital any organization that access databases. Because computers' hard drives are slow, the more data that is stored, the longer it takes to access useful information. To improve the speed of data retrieval, caching is technique that can be used to store frequently-used results in fast levels of computer memory. By storing query results in a cache, we can narrow the search of future database queries and improve the speed of similar queries. However, most systems are not able to reserve large chunks of memory for this cache storage. A cache replacement algorithm can limit and control the amount of memory the cache consumes by determining which cache entries to remove in order to make space for new entries. This research finds the most efficient replacement policy for our cache that provides an acceleration to database queries without requiring an exorbitant amount of memory.

Introduction

Databases in modern computer systems store most data on disk, the slowest storage device attached to a computer, so retrieving or searching for each query using disk takes too long.

To speed up response times, databases using caching to store previously computed results in memory which is faster than disk. The solutions of future queries can be obtained by finding the cached results of previous queries. Cached result vectors may also partially satisfy the query. The remaining bit-vectors of the query is a smaller subset that will be processed faster.

Since memory is a limited resource, if the size of the cache is unmonitored, it will grow uncontrollably and cause thrashing. A computer thrashes when it spends more time managing memory than processing information. To control the size of the cache, we created a replacement algorithm for the cache so that when the cache is full, the replacement algorithm will determine which cache entries to remove in order to make space for new entries.



Methodology

To determine the maximum size of the cache, we ran a simulation without any replacement to determine how large the cache would grow unmonitored which provided us the maximum size of the cache. For our experiments, we incremented portions of the maximum cache size (50%, 25%, 10% and 5%) to use as the fixed cache sizes for the cache. Each simulation processed the same 1,000,000 queries. For each cache size, we ran each policy 5 times and removed the flyaway queries from our results caused by software interruptions like garbage collection, then the 5 results from were averaged together.

Replacement Policies

We applied some of the well-known policies and then created our own policies specific to this cache system.

Random
Removes a random entry in the cache
- Simple and quick
- Does not consider if the removed entry is favorable to keep
- Does not consider temporal locality

First In, First Out (FIFO)
Removes the oldest entry in the cache
- Simple and quick
- The oldest entry might an entry that is used frequently
- Victim to Belady's anomaly: Increasing cache size does not always increase performance

Least Frequently Used (LFU)
Removes the least frequently used entry in the cache
- Always removes an entry that doesn't contribute to future calculations
- To keep track of the frequency, each entry requires a counter
- Every time an entry in the cache is referenced, it's position in the cache will be updated which is a linear search

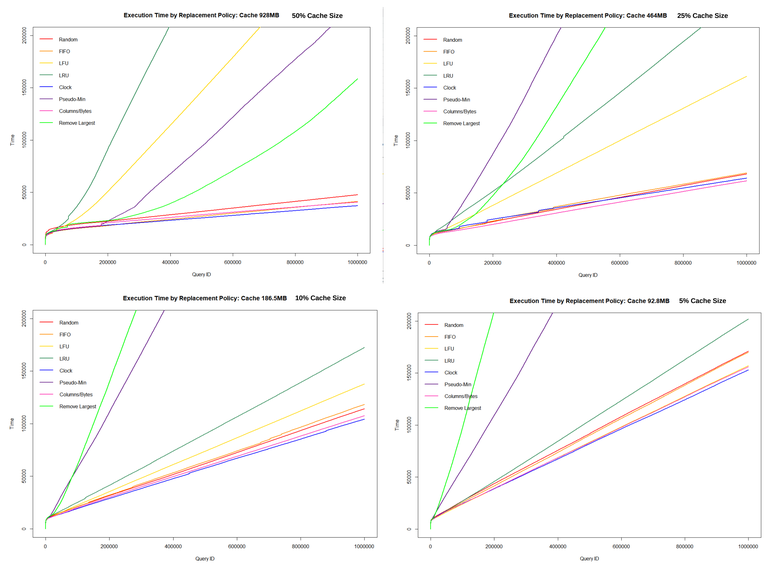
Least Recently Used (LRU)
Removes the least recently used entry
- Attempts to approximate the optimal algorithm (MIN) by using the past to predict the future
- Every time an entry in the cache is referenced, it's position in the cache will be updated which is a linear search

Clock
This policy uses a circular queue and a pointer to which passes through the queue. Each entry has a bit which is set to 1 upon entry. When replacement occurs, the pointer traverses through the queue until it reaches a 0. When it lands on a 1, it changes the 1 to a 0.
- Attempts to approximate LRU, picking an entry that is close enough to the least recently used
- Tends to be fast and tends to remove undesirable entries
- Even in the worst-case scenario, this policy will perform as well as FIFO

Remove Largest
Removes the entry that takes up the most space in the cache
- The goal of this policy is to limit the amount of time spent finding a replacement by removing the largest cache entry
- Larger entries tend have a larger coverage and took longer to compute, so this policy does not value those aspects

Columns Over Bytes
Removes the entry with the smallest Column/Bytes ratio
- Orders the entries in a priority queue based on the Column/Bytes ratio of the entries
- The Columns/Bytes ratio attempts to maximize coverage (Number of Columns the entry spans) while minimizing the space in the cache (the size of the entry in Bytes)\
- Every new entry needs to be sorted into the queue which is a logarithmic sort

Results



Analysis and Conclusion

- Clock seems to be consistently the best replacement policy, narrowly beating Columns/Bytes at every cache size except 25%. By finding the closest, less recently used entry, Clock can select a decent entry to remove quickly which is why it outperformed the other policies.
- Columns/Bytes was the next best replacement policy. The logarithmic sort for each new entry seemed to slow the algorithm down just enough to be slower than Clock.
- The next two best policies were FIFO and Random. The simplicity and speed of these four policies performed best at large cache sizes and slowed down as the cache size decreased.
- LRU and LFU performed better at smaller cache sizes than at large cache sizes. The bookkeeping required for these policies weighed down the execution time.
- Remove Largest performed better at large cache sizes than at smaller cache sizes. At smaller cache sizes, this policy would only be able to keep the smallest cache entries, requiring the database to spend more time on disk patching together the gaps between the cached solutions.

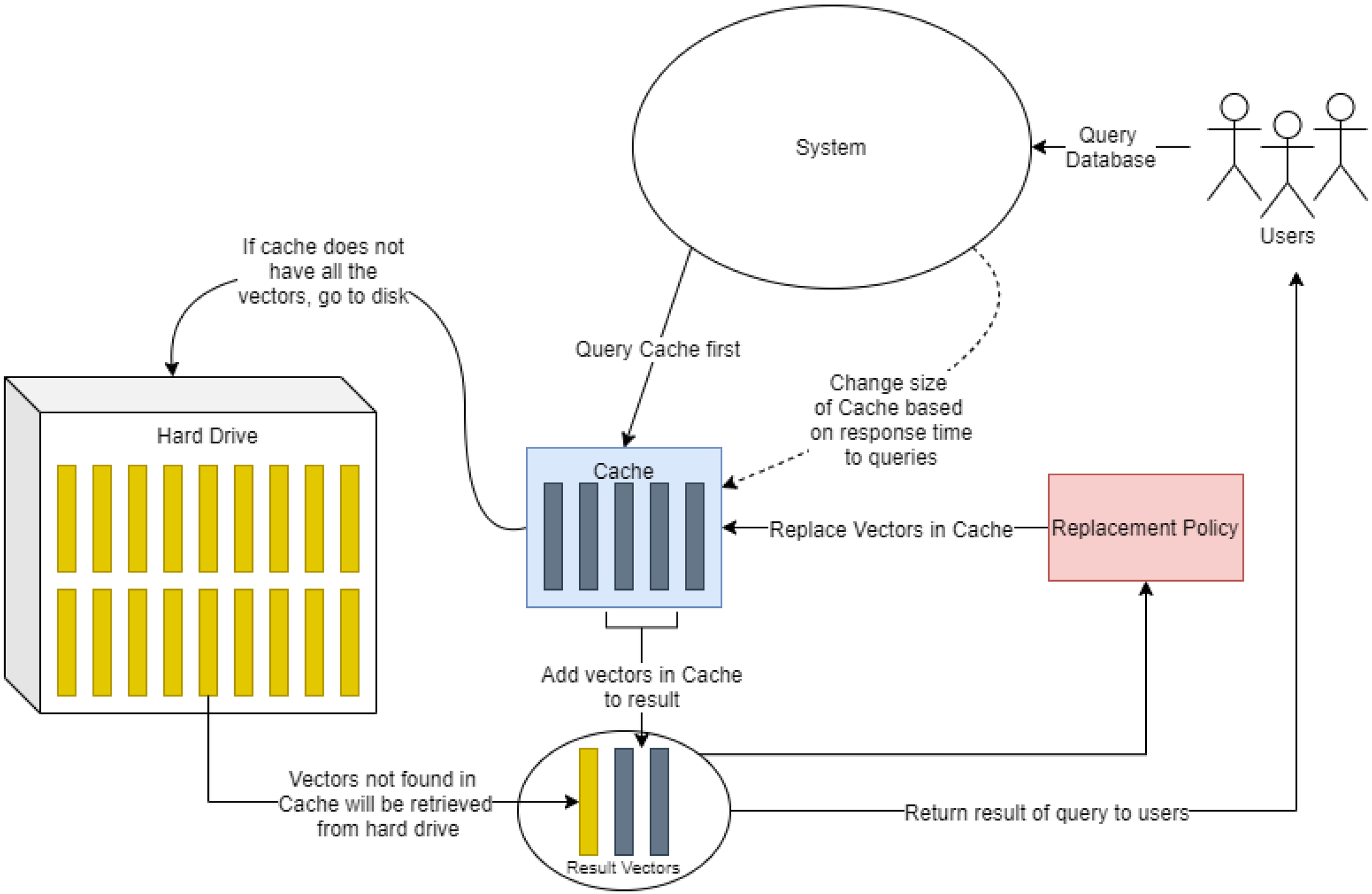
Future Work

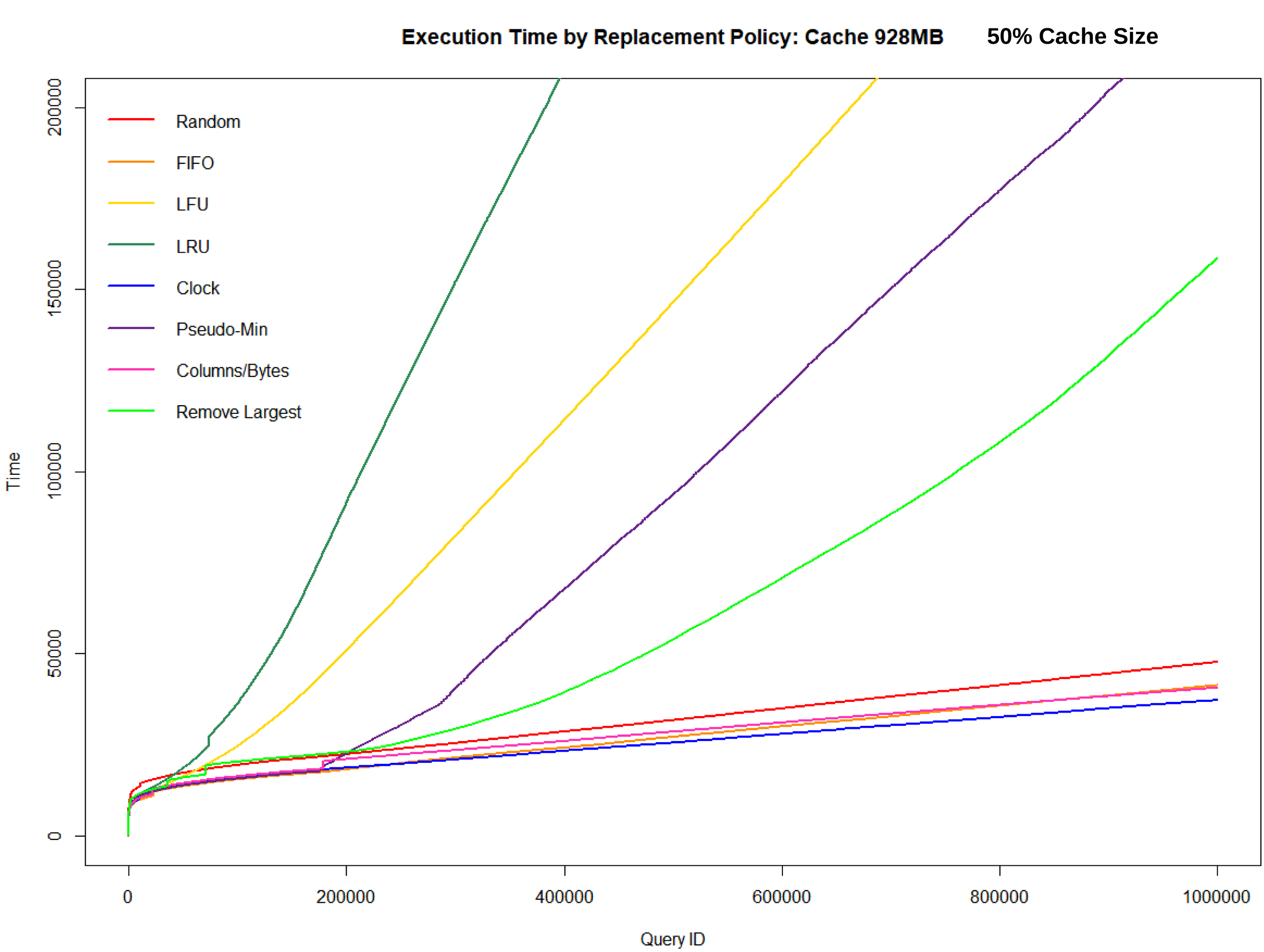
In future work, we could use our findings to determine the minimal size of cache that can deliver the specified database performance given certain quality-of-service constraints (such as response time or throughput) by creating an algorithm that alters the size of the cache to meet the quality of service constraints, (i.e. increasing the limit when the constraints are not met and decreasing the limit when the constraints are easily met).

We could also investigate possible placement policies for the cache system. In this research, every new query adds its result to the cache, but it may be more efficient to only add results that fulfill certain properties (such as if they bring in X amount of new columns to the cache).

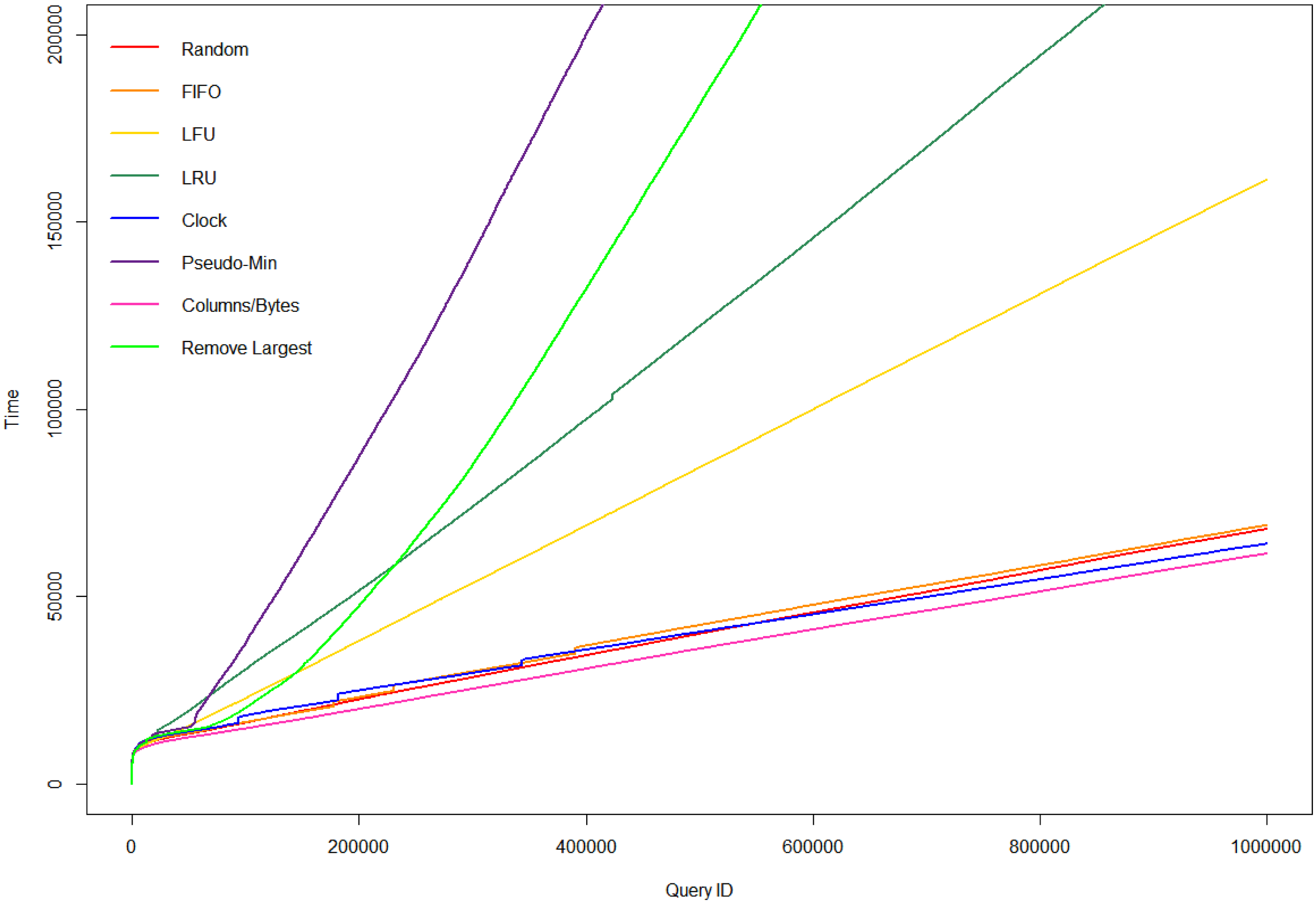
Acknowledgements

Thanks to Professor David Chiu from the University of Puget Sound and Professor Jason Sawin who helped advise and mentor me on this research, Manya Mutschler-Aldine who collaborated with me on this research, and the Agricola Donors for funding this research.

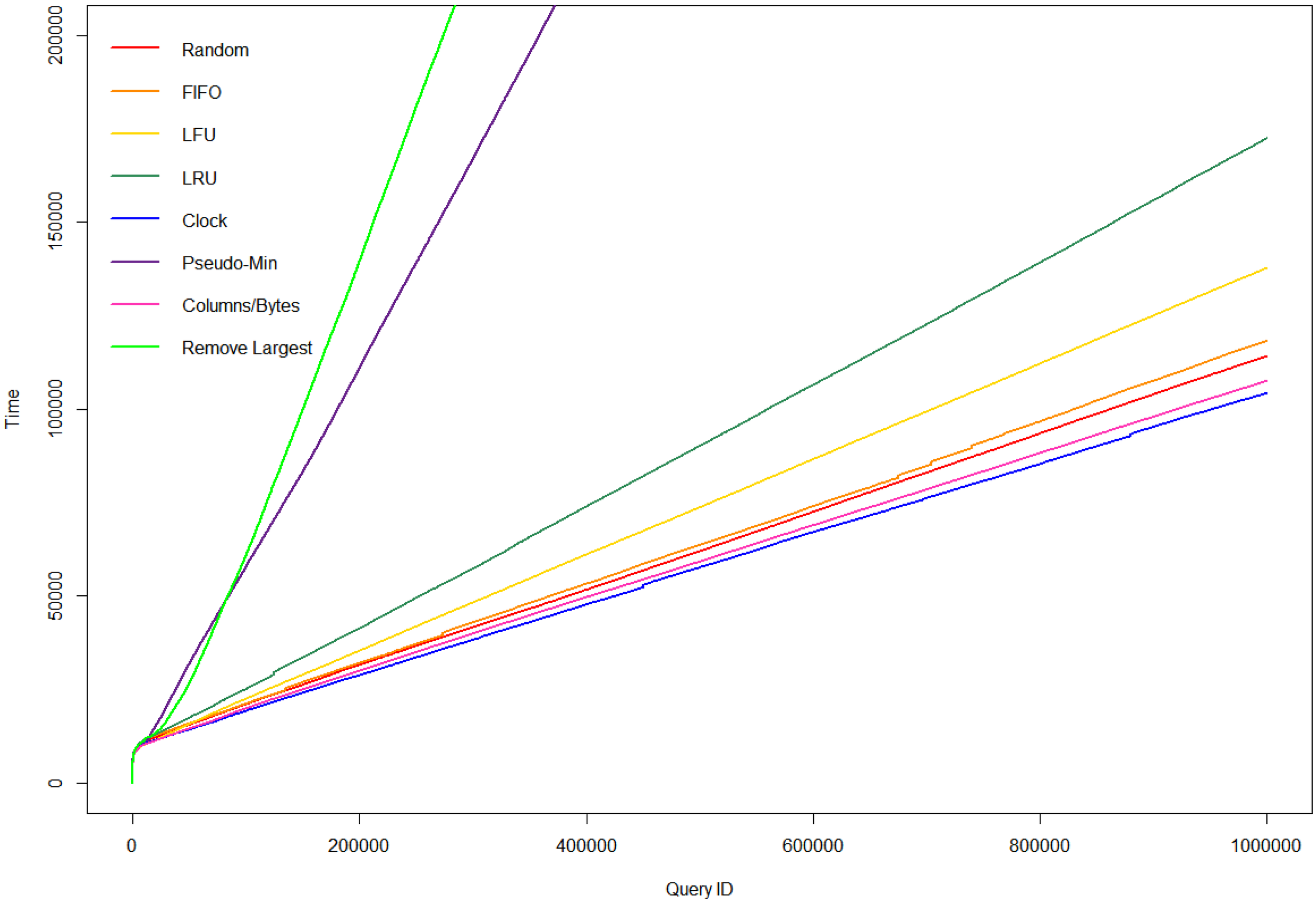

System
Query Database
Users
If cache does not have all the vectors, go to disk
Query Cache first
Change size of Cache based on response time to queries
Hard Drive
Cache
Replace Vectors in Cache
Replacement Policy
Add vectors in Cache to result
Vectors not found in Cache will be retrieved from hard drive
Result Vectors
Return result of query to users


Execution Time by Replacement Policy: Cache 928MB 50% Cache Size
Random
FIFO
LFU
LRU
Clock
Pseudo-Min
Columns/Bytes
Remove Largest
Time
0
50000
100000
150000
200000
0
200000
400000
600000
800000
1000000
Query ID

Execution Time by Replacement Policy: Cache 464MB 25% Cache Size



Execution Time by Replacement Policy: Cache 186.5MB **10% Cache Size**



Execution Time by Replacement Policy: Cache 92.8MB **5% Cache Size**