HaskStore-DB

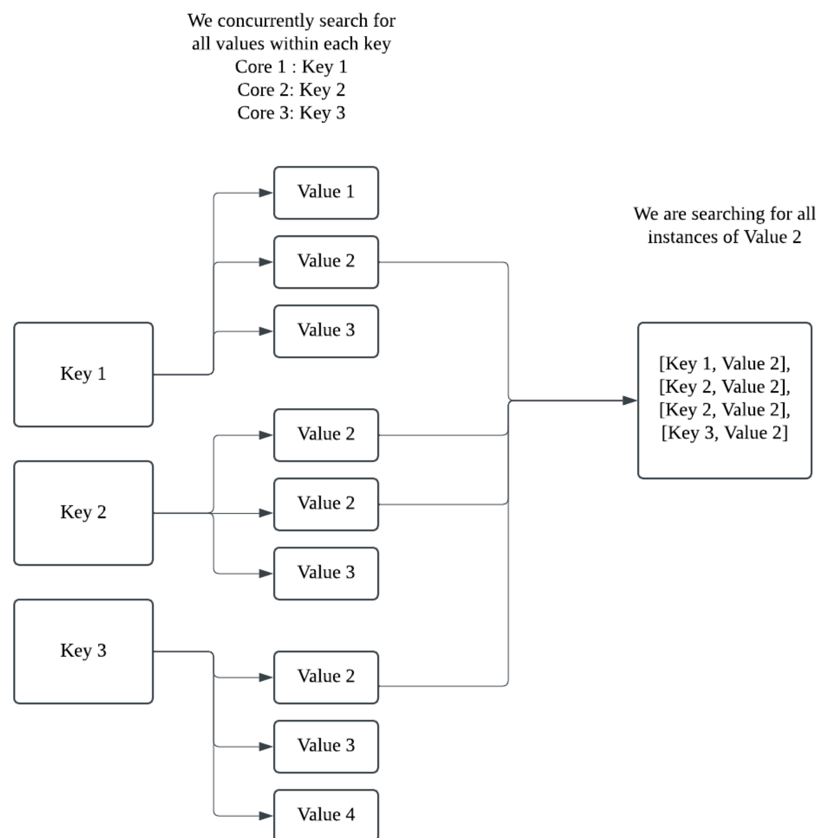
Haolin Guo (hg2691), Yuanqing Lei (yl5457), Ava Penn (ap4315)

Summary

We propose to implement a parallelized key-value (KV) database in Haskell, with parallelism leveraged to optimize search operations. Our implementation will focus on efficient concurrent querying of values associated with keys, along with basic functionality for adding and deleting entries from the database. This feature will be particularly beneficial in scenarios such as keyword retrieval and data filtering, where efficient querying of values plays a critical role.

By implementing this, we can not only search for values for any given key (which is what traditional KV stores do) but also search for all keys that contain certain values. For instance, we can search for all values associated with "Key 1", and also all keys that contain "Value 2."

In the diagram below, we show how our algorithm can be used to search for all instances of "Value 2." The user might want all the keys that contain value 2, or the number of keys that contain value 2. In either case, the user can easily process their desired information from our return value of all [key, value] pairs that have value 2.



The database will store a list of keys, where each key maps to multiple associated values. When performing a search for a specific value, we will divide the workload among multiple processes, with each process searching a subset of the data concurrently.

Parallel Search Optimization

To optimize search performance, we will incorporate workload balancing strategies:

- Keep track of the total number of keys, total number of values, and the size of the value lists for each key.
- Distribute keys across cores to balance the workload effectively.
    - If the length of a list of values exceeds the workload per core, we split the list of values between cores
    - If the length of a list of values is below the workload per core, we add multiple key, value pairs to that workload until reaching the limit
    - For example, keys with longer value lists will be evenly distributed among cores, ensuring no core becomes a bottleneck.

By dividing the search space and assigning balanced workloads, each core will process a subset of the keys, performing the search concurrently. The results from each core will be combined to produce the final search output. The size of chunks that we search sequentially will be determined by threadscope analysis.

Additional Features

In addition to parallel search, the KV database will support the following:

- Adding Entries: Insert a new key and its associated values into the database.
- Deleting Entries: Remove a key and its associated values from the database.

Implementation Plan

1. Data Retrieval & Storage:
    - We can create mock key-value data for testing. (See the diagram above as an example)
    - Use a Haskell data structure (e.g., Map, List, or Repa) to store keys and their associated values.
    - We also keep track of the number of values for each key, as well as the total number of values to help us distribute work among different cores.

2. Sequential Method

- Write a sequential method for searching for values as a starting point and comparison against our parallel approach.

3. Parallelism
    - Implement a parallel search using the *Control.Parallel.Strategies* library.
    - The strategies library is useful because we want to be able to test which chunk size is most effective at parallelizing:
        - Can use a single strategy to evaluate the performance when using different chunk sizes
        - For instance, using parList rseq to spark the computations of every chunk of the same size and evaluate the result into WHNF.
        - Then use a timing package to compile these results into a list for different chunks sizes, where the same strategy can be used for all chunk sizes.
        - Then we can determine the minimum time and its corresponding chunk size.
    - We shouldn't need to use the Par monad, since our problem doesn't require the result of any previous chunk to determine the result of the following chunk.
    - Another area for efficiency is splitting the work into smaller chunks. This will require splitting the data corresponding to one specific key, which would require copying over the entirety of at least one list, into 1 or more chunks.
        - We are interested in approaches that could avoid this bottleneck. For instance, using the Vector class so that we can randomly access the elements in a list in constant time, instead of linearly.
        - This part should be done sequentially, but quickly.
        - Repa could be useful for determining the size of a list of values quickly, which is useful in determining how lists of values should be split across chunks.

4. Testing and Optimization:
    - Test for the correctness of our code using unit tests and injecting test data.
    - Compare performance metrics (between single-core and multi-core implementations).
    - Find at which point parallel techniques can significantly and meaningfully speed up, and decide when to use parallel implementation (i.e. determine the size of chunks that we process sequentially, and how many of these chunks to assign to each core).
    - Update our code to determine when to use parallel v.s. sequential method to optimize performance.