CS315 Assignment 1

B.L.O.B.

2 November 2024

1 Disk Layout

The records in the disk are laid out in the following manner.

- First half consists of all records sorted by year then name (This is to avoid seek time in the case of year only queries.)
- The second half consists of all records sorted by name then year (This is to avoid seek time in the case of name only queries.)

2 Indexing and statistics

2.1 Experimentation

- **Binary Search**
  - **Disk Layout:** Data sorted by `id`, data sorted by `name`, and data sorted by `year`.
  - **Data Structures:** Utilizes sorted lists for efficient binary search operations.
  - **Year-only Queries:** Binary search on tuples sorted by years handles specific and range-based year queries.
  - **Name-only Queries:** Binary search on tuples sorted by name for exact and prefix searches.
  - **Conjunctive Queries:** Uses intersection of ID sets from multiple query conditions for conjuctive searches.
  - **Metrics Optimization:**
    - **More Optimized:** `t_build`
    - **Lack of Optimized:** `t_idx`
- **Array**
  - **Disk Layout:** Data is sorted by `year` then name and data is sorted by `name`.

  - **Data Structures:** Arrays for indexing and range queries.
  - **Year-only Queries:** Uses binary search in segmented arrays by year.
  - **Name-only Queries:** Implements binary search for exact and prefix matches.
  - **Conjunctive Queries:** Applies binary search on segmented lists for combined name and year queries.
  - **Metrics Optimization:**
    * **More Optimized:** `idx_size`
    * **Lack of Optimized:** `t_idx`

- **Numeric Buckets** (for year-only queries only)
  - **Disk Layout:** Data sorted by `year`.
  - **Data Structures:** Buckets index each year to its range in the sorted array, enhancing query speed for years.
  - **Year-only Queries:** Efficiently handles year-specific queries:
    * `get_eq(year)` returns indices for exact year matches.
    * `get_le(year)` returns indices for years up to a specified year.
    * `get_ge(year)` returns indices for years starting from a specified year.

- **Single Trie With Year Buckets**
  - **Disk Layout:** Data is sorted by year then name and data is sorted by name.
  - **Data Structure Used:** The Trie is employed where each node features buckets for specific years. These buckets store ranges of disk locations and counts of records for those years, enhancing year query performance.
  - **Year Only Queries:** Yearly data retrieval is implemented by a binary search on a pre-sorted list of years, each linked to a range of disk locations.
  - **Name Only Queries:** The Trie supports fast name searches, capable of resolving exact and prefix ('LIKE') queries by linking each character node to specific disk locations.
  - **Conjunctive Queries:** Queries involving both names and years are efficiently handled by first resolving the name through the Trie and then applying year filters via the buckets.
  - **Metrics Optimization Analysis:**
    * **More Optimized Metric:** `t_idx`
    * **Lack of Optimized Metric:** `idx_size`

- **Global Trie and Year Tries**
  - **Disk Layout:** Data is sorted by year then name, and data is sorted by name.
  - **Data Structure Used:** The indexing system utilizes separate Tries for each unique year in the dataset. These year-specific Tries optimize queries within their respective years.
  - **Year Only Queries:** Year-only queries use a sorted list of years to rapidly access specific records via binary search, linking directly to disk locations.
  - **Name Only Queries:** Name queries are managed within each year-specific Trie, providing rapid search capabilities by traversing nodes that correspond to each character and providing direct access to disk locations.
  - **Conjunctive Queries:** For conjunctive queries, the appropriate year-specific Tries are selected based on the year predicates of the query. Disk locations from each relevant year-specific Trie are then merged to produce the final set of results, effectively narrowing down the search space based on both name and year.
  - **Metrics Optimization Analysis:**
    - **More Optimized Metric :** `t_idx`
    - **Lack of Optimized Metric:** `t_build`
- **Filtering By Names, Then Years**
  - **Disk Layout:** Records are organized into two segments: one sorted by year then name, and another sorted by name only.
  - **Data Structure Used:** A single Trie is used for all names, indexing the entire dataset to manage name searches across all years. Each node maintains ranges of disk locations.
  - **Year Only Queries:** Year-only queries utilize a sorted list of years to quickly access records by year using binary search, linking directly to corresponding disk locations in the indexed data.
  - **Name Only Queries:** The Trie facilitates rapid searches by name, with each node representing a character and storing disk location ranges that span across all years.
  - **Conjunctive Queries:** For queries involving both name and year, the system first filters results by name using the Trie. It then applies year criteria to further refine the disk locations returned, and selecting the disk locations that meet both the name and year conditions.
  - **Metrics Optimization Analysis:**
    - **More Optimized Metric :** `idx_size`
    - **Lack of Optimized Metric :** `t_idx`

- **Additional Approaches**
  - **Trie Compression:** We explored trie compression to reduce the `idx_size`. While it successfully decreased the index size, it led to increases in both `t_build` (the time required to build the index) and `t_idx` (the time required to index data). Due to these trade-offs, particularly the increased times, we opted not to implement trie compression in our final solution.
  - **Translation of Disk Locations:** To optimize `t_seek`, we evaluated whether year-sorted IDs (forming the first half of the disk) or name-sorted IDs (forming the second half) resulted in less seek time for each range of disk locations. Seek time was calculated as the maximum index minus the minimum index plus one minus `t_read`. This analysis revealed that the optimal arrangement reduced `t_seek` by approximately 24 times, with some increase in `idx_size` and some increase in `t_idx`. Given these significant improvements in performance, we implemented this translation of disk locations in our final solution.

c

Approach	t_build	disk_size	idx_size	t_idx	t_seek	t_read	score
Binary Search	0.2281367667	300000	286100	3.787527567	6400303	1400713	1
Array	0.736580000	200000	1662898	0.930382966	2752003	1400713	1
Global Trie and Year Tries	3.812614667	200000	19770894	0.035879767	2227760	1400713	1
Single Trie With Year Buckets	2.3308658666	200000	13788002	0.028946199	2227760	1400713	1
Global Trie and Year Tries Compression	3.2062207666	200000	7502228	0.037612599	2227760	1400713	1
Single Trie With Year Buckets Compression	3.0678317333	200000	1004106	0.044787533	2227760	1400713	1
Filtering By Names, Then Years	1.4122694333	200000	6384134	0.1231322999	93206	1400713	1
Single Trie With Year Buckets Translated	1.872402300	200000	14289759	0.04361066666	93206	1400713	1

Table 1: Comparison of different data indexing approaches

2.2 Final Approach: Single Trie With Year Buckets + Numeric Buckets + Disk Location Translation

- **Disk Layout:** Records are organized into two segments on the disk: one sorted by year then name, and another by name then year.
- **Data Structure Used:** The Trie is employed where each node features buckets for specific years. These buckets store ranges of disk locations and counts of records for those years, enhancing year query performance.
- **Year Only Queries:** Utilizing numeric buckets that index each year to its range in the sorted array enhances query speed:
  - `get_eq(year)` returns indices for exact year matches.
  - `get_le(year)` returns indices for years up to a specified year.
  - `get_ge(year)` returns indices for years starting from a specified year.

- **Name Only Queries:** The Trie supports fast name searches, capable of resolving exact and prefix ('LIKE') queries by linking each character node to specific disk locations.
- **Conjunctive Queries:** Queries involving both names and years are efficiently handled by first resolving the name through the Trie and then applying year filters via the buckets.
- **Metrics Optimization Analysis:**
  - **More Optimized Metric:** `t_idx`
  - **Lack of Optimized Metric:** `idx_size`
- **Translation of Disk Locations:** To optimize `t_seek`, we evaluated whether year-sorted IDs (forming the first half of the disk) or name-sorted IDs (forming the second half) resulted in less seek time for each range of disk locations. This analysis showed that the optimal arrangement reduced `t_seek` by approximately 24 times, with only some MB increase in `idx_size` and some increase in `t_idx`. Given these significant improvements in performance, we implemented this translation of disk locations in our final solution.

3 Query optimization techniques

- Year only querries
  - If the query is such that there can be no possible matching records in the specified domain then we directly return 'empty'. E.g. year = 10, year >= 10,000, year <= 12.
  - We have made use of pre-computation of the start and end index for the querries. This helps us answer the queries in O(1) time.
- Name only querries
  - To optimise the query processing time, instead of building a secondary index over the names, we decide to trade-off space and build a sparse primary index over the names. This helps us to optimise for these types of querries.
- Optimising conjunctive querries
  - For the conjunctive query, if the 'year' is invalid, then we directly return 'empty'. This helps the time we would have otherwise spent in traversing the tree for the name.
  - If the year in conjunctive query is such that it covers all the records, then we basically treat that query as an only 'name' query. This results in time saving as only 'name' query implementation is faster than the conjunctive query due to no need of iteration over year buckets in a node.

– Two types of data are present on the disk namely 'sorted by name then year' and 'sorted by year then name'. We compare the seek time for returning indexes on both of the layouts and finally return the most optimal helping to optimise seek time.

4 References

https://medium.com/@nirmalya.ghosh/13-ways-to-speedup-python-loops-e3ee56cd6b73
https://stackoverflow.com/questions/40961684/custom-sorting-tuples