Eliminate Staged Search Investigation
Contents

1 Purpose .................................................. 1

2 Background ............................................. 1
   2.1 Overview of current search logic ......................... 1
   2.2 Overview of investigated mechanism ....................... 2

3 Prototyping process .................................. 2

4 Outcome ................................................. 3
   4.1 Minimum Postgres version requirement ..................... 3
   4.2 Observations on timing .................................. 3
   4.3 Tuning sensitivity ...................................... 3
   4.4 Future directions ...................................... 4
   4.5 Conclusion ........................................... 4
1 Purpose

Evergreen stores all data, including that useful for patron and staff search, in a normalized schema that is time and space efficient for transactional use cases, and provides guarantees on data integrity. In addition, development is made simpler than would be the case otherwise and arbitrary reporting is made possible.

However, this structure is not effective for direct, SQL-only search functionality in a hierarchical, consortial dataset. This is a problem that is relatively unique to Evergreen, as it is most often employed to host and serve large consortia with overlapping bibliographic datasets and non-overlapping item and location datasets. Other search engines, including those built into other ILSs, do not generally have to account for hierarchically organized location visibility concerns as a primary use case. In other words, because it provides functionality that requires a hierarchical view of non-bibliographic data, a problem space for Evergreen is essentially nonexistent in competing products.

Evergreen’s search infrastructure has evolved over the years. In its current form, the software first performs a full text search against extracted bibliographic data and limits this initial internal result set to a configurable size. It then investigates the visibility of each result on several non-bibliographic axes. These visibility tests take up the preponderance of CPU time spent in search, with full text search of the bibliographic data generally completing within milliseconds. The main reason this multi-stage mechanism is used is that there are many visibility axes and attempting to join all required data sources together in a single query will cause the search use case to perform very poorly. A previous attempt to create a pure SQL search mechanism failed for this reason.

A significant drawback of the current approach is that the costs imposed by visibility filtering search results using normalized non-bibliographic data, either in-query or separated from the main full-text query as it is today, make it necessary to place limits on the number of database rows matched by full-text query constructs. This in turn can cause searches to omit results in certain situations, such as a large consortium consisting of a few large libraries and many small libraries.

However, it may be possible to overcome this performance issue by providing an extensible way to collect all visibility related information together into a small number of novel data structures with a compact in-memory representation and very fast comparison functions. In this way, it is hoped that Evergreen may be able to use pure SQL search strategies and therefore avoid result visibility problems while also benefiting from improvements to the core PostgreSQL database engine. Further, it is hoped that this will also open the door to indexing improvements, such as removal of the need for duplicate data storage, which could reduce resource requirements and have a direct, positive effect on patron and staff search experience.

2 Background

2.1 Overview of current search logic

1. Construct core bibliographic search query
2. Collect non-bibliographic filtering criteria
3. Pass query and filters to a database function
4. Calculate hierarchical location information for visibility testing
5. Open cursor over core query, limited to superpage_size * max_superpages records
6. Core query implements bib-level sorting
7. For each result
   a. NEXT if not on requested superpage
b. Check deleted flag, based on search type

c. Check transcendence
   i. Return result if true

d. Check for direct Located URI in scope
   i. Return result if exists

e. Check copy status + (circ lib | owning lib) based on modifier

f. Check peer bib copy status + (circ lib | owning lib) based on modifier

g. Check copy location based on filter

h. Check peer bib copy location based on filter

i. General copy visibility checks
   i. If NOT staff
      A. Check for OPAC visible copies (trigger-maintained materialization)
      B. Check for peer bib OPAC visible copies
   ii. If staff
      A. Confirm no copies here
      B. Confirm no peer bib map
      C. Confirm no copies anywhere
      D. Confirm no Located URIs elsewhere

j. Return result if not excluded

8. Calculate summary row

2.2 Overview of investigated mechanism

Record and copy information (everything checked in (7) above) is collected into a novel data structure that allows all visibility-indicating criteria to be flattened to integer arrays. This is facilitated by a database trigger in much the same way that basic OPAC copy visibility is collected for copies today.

At search time, required and user-requested visibility restrictions are converted to \texttt{query\_int} values. Results are directly filtered based on these calculated \texttt{query\_int} values. This works in a way analogous to record attribute filtering, avoiding the need to test statuses, libraries, copy locations and location groups, copy OPAC visibility, peer bibliographic record, Located URIs, or bibliographic record sources directly.

A total of sixteen copy-related visibility filters are possible, while six (copy visibility flag, owning library, circulating library, copy status, copy location, and copy location group) are used in this prototype. Likewise, a total of 16 bibliographic record visibility filters are possible, with two (bibliographic source and Located URI library) in use for prototyping purposes.

3 Prototyping process

Initial prototyping was performed on the 9.4 version of Postgres. There are several set, aggregate, and window functions available in 9.4 and newer that improve performance and reduce code maintenance cost of the prototyped mechanism.

First a set of representative search queries were identified from production database logs. These queries used a mix of various features, and included single and multiple class searches, searches limiting by various non-bibliographic filters, and searches at various hierarchical levels.

Specific filter axes were identified based on both the existing code and the representative queries, and a rough data structure was designed to facilitate pre-computed storage of the relevant filter values. This was used to test the efficacy of the general concept described above, which proved to be worth pursuing further based on initial experiments.

At this point it became critical to determine the potential effort required to support this new mechanism within the existing code base. In order to do so, the QueryParser module was augmented to produce a modified core query that made use of the initial,
experimental data structures. It was determined through this process that, with appropriate reorganization, the new mechanism is supportable within the overall structure of the existing search subsystem.

A systematic design was created to derive and maintain persistent non-bibliographic filtering values for relevant database objects. While initial testing was performed on a small, controlled dataset in order to speed iterative testing and development, all efficacy and performance testing was performed again on a copy of a full production Evergreen database containing 1.7 million bibliographic records, 3.4 million copies, 140,000 Located URIs, and 125 organizational units. Identical tests were performed on Postgres versions 9.4, 9.5, and 9.6.

4 Outcome

4.1 Minimum Postgres version requirement

Due to features, particularly functions, available only in 9.4 and newer that are key to the performance of the prototyped method, Postgres 9.4 will need to be the new lowest supported version for use with Evergreen. While some of the new features and functions could be implemented as user-defined functions in PL/PGSQL, they would not be fast enough to make this pure-SQL search viable.

Among the important improvements that Postgres 9.4 and newer versions bring to Evergreen are:

- Version 9.4 improved GIN indexes in ways that directly benefit Evergreen, as well as how anti-joins are planned which matters for some Evergreen searches.
- Version 9.5 introduced many general performance improvements, especially for joins and sorting, and brought planner improvements that impact complex queries such as those prototyped here.
- Version 9.6 delivered more general performance improvements, particularly for large servers such as those that Evergreen databases tend to live on, as well as more improvements to GIN indexes, executor changes that can avoid unnecessary work in search queries, new built-in full-text phrase searching, and initial parallel query execution.

4.2 Observations on timing

The cost of the non-bibliographic filter value caching maintenance process is 10-40% faster than existing partial caching logic which it would replace.

The prototyped replacement achieves approximately 10% faster search times than the old, suboptimal mechanism time for broad searches. The new method is faster for more selective searches, often by up to 90% faster. In both broad and narrow search cases the new mechanism performs with complete accuracy and does not miss small-collection hits in large consortia as the existing code does.

Unsurprisingly, and in addition to the above improvements, performance improved marginally as each successive Postgres version was tested.

4.3 Tuning sensitivity

User-level timeouts are still possible with both, though the `gin_fuzzy_search_limit` GUC can be used to set a time cap on the prototyped mechanism.

Because it uses a more complex query structure, the prototyped mechanism is somewhat more sensitive to Postgres tuning in general. In particular, lowering `random_page_cost` from the default of 4.0 to a more reasonable 2.0 is important for proper query planning. For Evergreen use cases where the search indexes and relevant tables are kept in RAM or SSDs are used for storage, this value is acceptable and useful in general.
4.4 Future directions

While there was not time enough to implement any direct replacements for existing relevance adjusting mechanisms (the “search adjustments” of old), some initial testing was done to investigate the efficacy of the similarity() and word_similarity() functions from pg_tgrm, and they look very promising. This won’t provide a direct replacement to what is possible today, but they could help with the development of new ranking mechanisms. Additionally, the pg_tgrm extension can be used to implement dataset-specific spelling suggestions among other improvements. Several of the features most interesting for Evergreen development are, however, available only as of Postgres 9.6.

4.5 Conclusion

While further improvement to the prototyped mechanism is almost certainly possible, it is currently as fast or faster than the existing code base with regard to performance. Because of the benefits seen in this experiment, particularly the removal of search hit limits which cause problems for common configurations, and the apparent lack of performance regressions, it is recommended that we proceed with this line of development in the elimination of Evergreen’s current staged-search procedure.