NeedleTail: A System for Browsing Queries

Albert Kim
MIT
alkim@csail.mit.edu

Samuel Madden
MIT
madden@csail.mit.edu

Aditya Parameswaran
U. Illinois and MIT
adityagp@illinois.edu

ABSTRACT

Analysts performing data exploration often *browse*; i.e., pose a query and then examine the details of a small number of the resulting records (independent of the size of the query result). In a typical session, analysts will start with one browsing query, examine a few of the resulting records, and then repeatedly issue new browsing queries by adding or removing predicates from their previous queries until they eventually gain a better understanding of the dataset. Unfortunately, traditional database systems are not engineered towards browsing: instead, these systems operate in an all-or-nothing manner, taking as long as it takes to return the entire set of results, however large it may be.

To this end, we demonstrate NeedleTail, a database system tailored towards an alternative database query interaction paradigm: browsing. NeedleTail makes efficient use of memory to store special bitmap indexes (called "swift indexes") that enable rapid retrieval of a small number of query result records. A key optimization challenge is ensuring that these indexes respect memory constraints while imposing as little additional retrieval overhead as possible. As part of this demonstration, we will showcase NeedleTail's "swift index optimizer", allowing conference attendees to view the impact of NeedleTail bitmap optimizations. We will also demonstrate the use of NeedleTail versus traditional database systems in a browsing setting on a variety of real-world datasets.

1. INTRODUCTION

Data analysts exploring large volumes of data often pose queries to a database and then examine a few records from the query result. They then repeat this process by modifying or reformulating the queries, until they are satisfied with the insights gathered. We call such workloads *browsing* workloads. For instance, data analysts working in an internet advertising firm may browse the profiles of a few individuals who live in Boston and clicked on a specific ad campaign. Based on what they observe, they may then examine profiles of a few individuals who live in Cambridge and clicked on the same ad, or those who clicked on alternate ads, to get a better understanding of their target audience.

Proceedings of the VLDB Endowment, Vol. 7, No. 4
Copyright 2013 VLDB Endowment 2150-8097/13/12.

Unfortunately, on large datasets, this process can be slow or unwieldy, since traditional database systems are not engineered for interactive exploration. Instead, given a query, traditional systems often operate in an all-or-nothing manner, taking a long time to return all the records in the query result, no matter how large it may be. Analysts have to wait until the entire query result is generated before they can examine them or formulate alternate queries.

While traditional databases do provide some limited support for "early return" of query results, they are not applicable to browsing. For example, top-k query processing systems [11] are optimized to return k result records: however, since the goal is to return the *top-k* records, the index structures and algorithms are very different and are not be applicable in an interactive browsing scenario. Additionally, although many databases support a `LIMIT` clause, allowing the user to indicate that only a few result records are desired, the index and data storage structures to efficiently execute queries with such a clause have not been well explored.

To this end, we present our system, titled NeedleTail[1], tailored towards this new, browsing-based database interaction paradigm. A browsing query returns a small, predetermined number of records that satisfy the query conditions as quickly as possible, independent of the total number of records in the query result. This way, analysts can issue a browsing query, instantly examine a "screenful" of records, before refactoring or modifying the query and repeating this process until they are satisfied.

There are two challenges in building NeedleTail to support browsing queries: First, given a query, NeedleTail must return the desired number of records as quickly as possible. Performance is even more crucial in such an interactive exploration scenario. Second, since the queries issued by analysts are ad-hoc, we cannot pre-compute and store the query results for all queries in advance, nor can we use traditional indexing structures like B-trees, since combining B-trees for unpredictable queries can incur high computation overhead.

To deal with both of these challenges, NeedleTail instead leverages a specially optimized in-memory bitmap index structure, called the *swift index*. Traditional in-memory bitmap indexes allow rapid retrieval of records matching ad-hoc predicates specified on the fly by a user. However, while compression schemes for bitmap indexes have been well-studied [13, 18, 19], the size of compressed bitmap indexes can still be large, especially if the original dataset is itself enormous—a common scenario in the era of "big data".

The key idea in NeedleTail is that we can store partial bitmaps, i.e., bitmap indexes where only the first $x\%$ (say 10%) of the records are indexed. These partial bitmaps allow us to retrieve a small number of the matching records needed to allow a user to browse results for any query. Specifically, the number of records that need to be

[1] NeedleTail is named after the fastest bird species in existence (a type of swift) [4].

indexed for browsing in NEEDLETAIL is proportional to the number of records the analyst wishes to view at one time rather than the size of the original dataset. This means we can always be sure that the size of the index will remain relatively small. We can also apply standard bitmap compression schemes developed for regular bitmap indexes to ensure that our partial bitmap structure can fit in memory. In addition, we introduce three novel bitmap compression schemes that are also employed by the swift index-optimizer.

Since record retrieval speed remains the top priority for browsing, NEEDLETAIL's swift index-optimizer selects the optimal combination of compressed and partial bitmap structures while respecting the memory budget specified by the user. In fact, a guarantee our optimized index structures provide is that NEEDLETAIL will not do more than a small constant multiple of the amount of work it would have done if it knew exactly where to find the desired output records. Our techniques are of independent interest for optimizing performance given constraints on memory, wherever bitmap indexes are used.

As part of this demonstration, we will showcase NEEDLETAIL's swift index optimizer, which will allow the conference attendees to view the impact of NEEDLETAIL bitmap optimizations, given constraints on memory and other system parameters. We will also demonstrate the use of NEEDLETAIL versus traditional database systems for browsing queries on a variety of real-world datasets.

2. PROBLEM AND SOLUTION OUTLINE

Setting: Our goal is to build a database system that accepts as input a regular SQL query Q operating on a database D, and returns as output k (a predetermined, fixed, small number, typically in the few tens or hundreds) records from $Q(D)$, as long as the number of records in the result for the original query $Q(D)$ is greater than or equal to k. If the number of records in the query result $Q(D)$ is less than k, then the entire query result $Q(D)$ is displayed. Any such database system is called a *k-browsing database system.*

We focus our discussion in this section on simple selection queries on a star schema; this includes the example query discussed in the introduction on providing records corresponding to individuals who lived in Boston and clicked on a specific ad. However, our query processing techniques and indexing structures are not limited to this subset of queries.

The kind of query we focus on for this discussion is a selection query, abstractly expressed using the following DNF (Disjunctive Normal Form):

$$\begin{aligned} Q = \quad & (A_{i_{11}} = a_{i_{11}} \wedge A_{i_{12}} = a_{i_{12}} \wedge \ldots \wedge A_{i_{1r_1}} = a_{i_{1r_1}}) \\ \ldots \\ \vee \quad & (A_{i_{l1}} = a_{i_{l1}} \wedge A_{i_{l2}} = a_{i_{l2}} \wedge \ldots \wedge A_{i_{lr_l}} = a_{i_{lr_l}}) \end{aligned}$$

For clarity, we focus on simple equality-based predicates instead of range predicates, even though our queries can involve range predicates, and our index structures can handle range predicates as well.

Query Processing Alternatives: We next consider various options for processing queries like Q above. Naturally, coupled with query processing strategies, we do need to decide up-front on the indexing strategies, without which such a system will simply not be able to answer queries at interactive browsing speeds.

Since the queries Q that can be applied to a $k-$browsing database system can be arbitrary, ad-hoc, and complex, this rules out several candidate methods for processing such queries:

- Sequentially scanning the entire relation until we find k records $\in Q(D)$ can be incredibly slow, especially when Q is complex.
- Conventional indexes, such as B-tree indexes, are also problematic. Since queries Q are ad-hoc and unpredictable, we must either index all attributes and intersect the B-trees at query time or create joint B-tree indexes across multiple attributes. The first scheme leads to high memory consumption and high computation overhead, while the second scheme leads to exponentially many indexes (every subset of attributes must be indexed).
- Approaches based on sampling or precomputation are not feasible either because we need to account for every possible ad-hoc query Q (where the number of attributes can be in the hundreds, and the number of values per attribute in the thousands), and generate appropriate samples beforehand — the set of samples is likely to take up too much space.

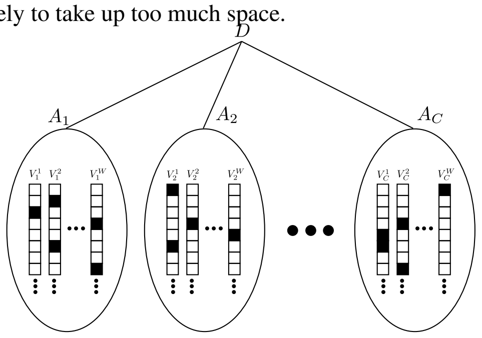

Figure 1: Bitmap Indexes

Instead, NEEDLETAIL employs a bitmap-based index. We first describe traditional bitmap-based indexes before we describe NEEDLETAIL's index structure. In traditional bitmap-based indexes (also displayed in Figure 1), we store a bitmap array V_i^j for each value a_i^j taken by each attribute $A_i, i \in \{1 \ldots C\}$. The rth entry of these arrays record whether the rth record in the relation has value a_i^j for attribute A_i (1 if yes, 0 if no). Thus, in order to evaluate a query like Q above, assuming these bitmaps for all A_i fit in memory, one simply needs to "AND" all the bitmap arrays for all the conditions joined together by conjunctions and "OR" all the conditions joined together by disjunctions. In fact, we do not need to perform this operation for the entire bitmap array; we can stop once we have isolated k records satisfying the query. At that point, these records can be retrieved from disk and returned to the analyst.

As illustrated above, bitmap-based indexes can result in efficient retrieval for arbitrary, complex selection queries. However, for very large datasets, vanilla bitmap-based indexes do not suffice, since they take up too much memory. Therefore, our focus is on designing index structures to support browsing, ensuring that the space utilization is within the memory available, such that the time taken to retrieve k records $\in Q(D)$ is minimized.

Formal Question: Thus, the formal problem we aim to solve is the following: *Given constraints on memory, database size, cardinalities of attributes, build in-memory indexes so that the time taken to retrieve k records is minimized under three settings:*

- for all selection queries in the worst case
- for all selection queries on average
- for queries on average given a workload

We next describe the index structure we use to address the problem.

Swift Indexes: NEEDLETAIL uses a specially optimized bitmap index structure called the *swift index*. Swift indexes benefit from two kinds of *optimization strategies* that enable it to take up significantly less memory than bitmap indexes.

- Partial Bitmaps: Since k-browsing database systems only need to return k results per query, we can get away with storing less information in memory. For each array V_i^j, we store only a part of it, i.e., we can retain only the first $x\%$ of the records (where x depends on both j and i).
- Compressed Bitmaps: Since bitmaps save a lot of redundant information, we can reduce the amount of information necessary

by using various compression schemes. We consider six compression schemes in NEEDLETAIL: the first three compression schemes have been used for bitmaps in the past, while the remaining are novel schemes that we introduce. We discuss compression schemes next.

Compression Schemes: Compression schemes for bitmaps allow us to store more information in a compact form. However, for many compression schemes, this leads to *loss of information*; that is, we may end up having to retrieve more records than we would if we didn't perform compression. (All the schemes we study produce only false positives, no false negatives; thus, at most, we will retrieve more records from disk than necessary.) One of our key contributions is to be able to reason about how we can leverage the benefits of all the optimization strategies—compression schemes and partial bitmaps—while not inducing too many false positive retrievals. We consider the following compression schemes.

- *Base-Encoding* [13]: This scheme maps multiple bitmap arrays into a smaller number of arrays by encoding them using a small number as a base. This scheme produces no false-positives.
- *Data Compression* [18]: This scheme involves the use of a lossless data compression technique, such as Run Length Encoding, or Word-Aligned Hybrid (WAH) code [17] to compress each bitmap array into a smaller number of bits. This scheme produces no false-positives.
- *Range-Binning* [13]: This scheme maps bitmap arrays for a consecutive range of values of an attribute ($\{V_i^j \dots V_i^{j+k}\}$ in our notation) into one bitmap array. The value at index r for the resulting array is set to 1 if the rth record has the given attribute equal to any value in the range.
- *Value-Binning:* This scheme is a generalization of range binning, in which we allow bitmap arrays for any arbitrary set of values for a given attribute to be mapped into a single bitmap array.
- *Attribute-Binning:* This scheme maps bitmap arrays for different values of different attributes into a single bitmap array.
- *Tuple-Binning:* In this scheme, a specific index in the bitmap array refers to a collection of records instead of a single record. For instance, the value at index r for a bitmap array is set to 1 if any of the $[\alpha r, \alpha(r+1)]$, $\alpha{>}1$, records has that specific value.

Optimization: Given these optimization strategies (compression and partial bitmap schemes), the time taken to retrieve k records depends on two quantities:

- The time taken to retrieve records from disk. Often, the total number of records retrieved in order to give us k records is larger than k (due to the false positives induced by the compression schemes).
- The time taken to lookup bitmap arrays in memory and the time taken to perform AND and OR operations on bitmap arrays in memory until we find k records that satisfy $Q(D)$; often, if there are false positives, we may need to do a lot more operations on bitmap arrays to find the desired k records.

In practice, the first quantity dominates the second, so we only optimize the first quantity.

As it turns out, designing optimized swift indexes (by identifying the best combination of compression and partial bitmap schemes) given constraints is NP-Hard; however, we can use an ILP solver to identify the best combination of optimization strategies the swift indexes should employ to minimize retrieval time while adhering to the memory budget.

3. SYSTEM DESCRIPTION

When preprocessing a dataset, NEEDLETAIL allows the database administrator to specify a memory budget. NEEDLETAIL then runs the *swift index optimizer* to design the swift indexes to minimize retrieval time while adhering to the provided budget. In addition to building the index structures during preprocessing, NEEDLETAIL also stores the original dataset on disk in column-oriented fashion so that at query time, matching records can be verified quickly (in case of false positives) before being returned to the user.

The current implementation of NEEDLETAIL is written in C++ and uses the Boost library [1] for its bitmap implementation. Parallelization is currently employed in NEEDLETAIL, but it is only used for concurrent reading and preprocessing of the original data. However, we plan to have support for parallel query processing in the future; because we assume a read-only workload[2], we can safely have multiple threads operating on the index structure concurrently. As of the moment, NEEDLETAIL is constrained to running on a single machine, but we plan to extend NEEDLETAIL to operate in a distributed environment. This allows NEEDLETAIL to spread its index structure across multiple machines if the memory space on a single machine is not sufficient. Once again, assuming a read-only workload makes this distribution trivial. By choosing to distribute NEEDLETAIL across multiple machines, we may also gain load balancing benefits for free.

4. RELATED WORK

The work related to NEEDLETAIL can be placed in three categories:

Approximate Query Processing: Over the past two decades, there has been a lot of work on approximate query processing, as examples, see [7,9,10,12]. Garofalakis et al. [8] provides a good survey of the area. There are multiple systems that support approximate query processing, including BlinkDB [6] and Aqua [5]. All of these systems focus on approximating aggregate queries, focusing on operators such as `SUM`, `AVG`, `VAR`. Our work is complementary to the work on approximating aggregation queries, since we focus primarily on selection queries, returning a subset of the query result.

Bitmap Optimization: There has been a lot of work on designing efficient bitmap schemes [13,14,15,18,19]. However, none of these papers consider *partial bitmaps*, which are relevant primarily in the browsing context. Furthermore, none of these papers consider the variety of other optimizations that we consider for swift indexes.

Limit Clause, Rate-Based Optimization, and Top-K: A number of database systems support a `LIMIT` clause: the way queries with such a clause are handled traditionally is by selecting a pipelined execution plan with no blocking operators (e.g., opting for nested-loops join instead of sort-merge join). Another related area of research is selecting query plans that maximize the output rate (i.e., the rate at which records are generated) [16]. However, there has been no work designing index structures to enable performant execution of ad-hoc browsing queries.

Also related is the work on top-k queries, which return the top k records of dataset ordered by a specified attribute. Although the work done in this field is extensive as demonstrated by Ilyas et al.'s survey [11], the browsing paradigm is markedly different from the top-k paradigm because we do not need to consider the ordering of the resulting records. Instead, NEEDLETAIL focuses on building index structures which return *any* k records as fast as possible.

5. DEMONSTRATION DESCRIPTION

We will demonstrate two aspects of NEEDLETAIL: First, through simulation, we demonstrate how NEEDLETAIL's swift index optimizer takes into account the available memory, as well as other

[2]Although we assume a read-only workload to support parallel query processing without locks, all the techniques discussed thus far can handle mutations in the dataset.

database parameters, to build optimized swift indexes that retrieve as few additional (false positive) records as possible to return k result records for any query, while respecting memory constraints. Second, we demonstrate on three real datasets, how NEEDLETAIL returns results orders of magnitude faster than existing database systems for browsing. We describe both these aspects in detail next.

5.1 Index Optimization Comparison

In our first demonstration, our goal is to illustrate to conference attendees how NEEDLETAIL's index optimization algorithms take as input, system and dataset parameters, and output a set of swift indexes, optimized for the scenario at hand.

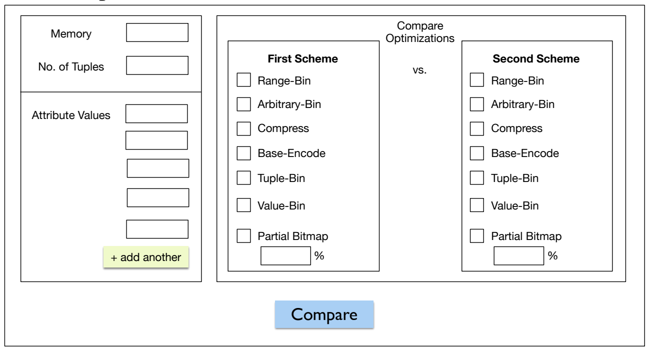

Figure 2: Swift Index Optimizer Input Parameters

We will provide to conference attendees an interface like Figure 2, where they can specify input parameters, including:

- the number of distinct values for each attribute in the dataset
- the amount of memory available
- the number of records in the dataset

For the given set of input parameters, the attendees may also choose to compare two sets of optimizations: NEEDLETAIL's swift indexes, by default, use all these optimizations. Once the attendees have chosen two sets of optimizations to compare, and have clicked on the compare button, a results pane like Figure 3 (one for each of the sets of optimizations) is displayed. The results pane provides

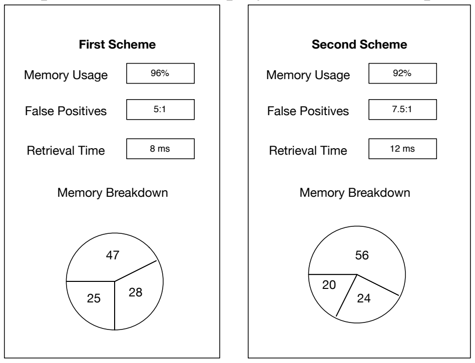

Figure 3: Index Result Display

the following information:

- the percentage of memory used for indexes
- the worst-case false positive rate, i.e., the ratio of the number of records that can be retrieved using the indexes, to the number of records that actually satisfy the query (Note that NEEDLETAIL can also optimize for the expected false positive rate, or the false positive rate given a workload, but we do not display that here.).
- the time taken to retrieve k records, assuming default system parameters
- the breakdown of how memory is used across different attributes

By comparing the results pane for two sets of optimizations, attendees can gain intuition for which swift index optimizations are most crucial and which ones only provide marginal improvement.

5.2 Performance Experiments

In this demonstration, we will allow conference attendees to actually see NEEDLETAIL in action on two separate datasets:

- Flight Quest Data [2]: This dataset is 50GB and contains various features (e.g., airport, delay time, aircraft type) about flights that occurred over the span of 12 weeks in 2013.
- Sloan Digital Sky Survey [3]: This dataset is tens of TBs and contains a wide variety of scientific measurements made using optical and infrared spectroscopy. The dataset covers close to a million galaxies.

These datasets represent both a variety of scenarios where NEEDLETAIL could be used (data journalism vs. scientific data analysis), and also represent datasets of varying sizes, showing that NEEDLETAIL gives us significant speedups even on very large datasets.

For any of these datasets, the users will be able to input a SQL query of their choice, or choose one of our preselected queries from a drop-down menu. The users can also select the number of desired records k they want to look at.

We will run the query side-by-side on NEEDLETAIL and on PostgresSQL, allowing the users to compare how quickly results are generated. This comparison will illustrate to the attendees how traditional databases are unsuited for browsing, and how systems like NEEDLETAIL can provide immense benefits on practical datasets.

6. REFERENCES

[1] Boost libraries. `http://www.boost.org/`.
[2] Flight quest data. `www.gequest.com/c/flight2-main/data`.
[3] Sloan digital sky survey. `www.sdss.org`.
[4] Swift. `https://en.wikipedia.org/wiki/Needletail`.
[5] S. Acharya, P. B. Gibbons, V. Poosala, and S. Ramaswamy. The aqua approximate query answering system. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, SIGMOD '99, pages 574–576, New York, NY, USA, 1999. ACM.
[6] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica. Blinkdb: queries with bounded errors and bounded response times on very large data. In *EuroSys*, pages 29–42, 2013.
[7] K. Chakrabarti et al. Approximate query processing using wavelets. In *VLDB*, pages 111–122, 2000.
[8] M. N. Garofalakis and P. B. Gibbon. Approximate query processing: Taming the terabytes. In *Proceedings of the 27th International Conference on Very Large Data Bases*, VLDB '01, pages 725–, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.
[9] P. B. Gibbons. Distinct sampling for highly-accurate answers to distinct values queries and event reports. In *VLDB*, pages 541–550, 2001.
[10] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. In J. Peckham, editor, *SIGMOD 1997*, pages 171–182. ACM Press, 1997.
[11] I. F. Ilyas, G. Beskales, and M. A. Soliman. A survey of top-k query processing techniques in relational database systems. *ACM Computing Surveys (CSUR)*, 40(4):11, 2008.
[12] C. Jermaine et al. Scalable approximate query processing with the dbo engine. *ACM Trans. Database Syst.*, 33(4), 2008.
[13] N. Koudas. Space efficient bitmap indexing. In *CIKM*, pages 194–201, 2000.
[14] D. Rotem, K. Stockinger, and K. Wu. Minimizing i/o costs of multi-dimensional queries with bitmap indices. In *SSDBM*, pages 33–44, 2006.
[15] R. R. Sinha, M. Winslett, K. Wu, K. Stockinger, and A. Shoshani. Adaptive bitmap indexes for space-constrained systems. In *ICDE*, pages 1418–1420, 2008.
[16] S. D. Viglas and J. F. Naughton. Rate-based query optimization for streaming information sources. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, SIGMOD '02, pages 37–48, New York, NY, USA, 2002. ACM.
[17] K. Wu, E. J. Otoo, and A. Shoshani. A performance comparison of bitmap indexes. In *Proceedings of the tenth international conference on Information and knowledge management*, pages 559–561. ACM, 2001.
[18] K. Wu, E. J. Otoo, and A. Shoshani. Optimizing bitmap indices with efficient compression. *ACM Trans. Database Syst.*, 31(1):1–38, 2006.
[19] K. Wu, A. Shoshani, and K. Stockinger. Analyses of multi-level and multi-component compressed bitmap indexes. *ACM Trans. Database Syst.*, 35(1), 2010.