

Inverted Indexes vs. Bitmap Indexes in Decision Support Systems

Truls A. Bjørklund and
Nils Grimsmo
Norwegian University of
Science and Technology
{trulsamu,nilsgri}
@idi.ntnu.no

Johannes Gehrke
Cornell University
Ithaca, NY
johannes@cs.cornell.edu

Øystein Torbjørnsen
Fast Search and Transfer, a
Microsoft® subsidiary
oysteint@microsoft.com

ABSTRACT

Bitmap indexes are widely used in Decision Support Systems (DSSs) to improve query performance. In this paper, we evaluate the use of compressed inverted indexes with adapted query processing strategies from Information Retrieval as an alternative. In a thorough experimental evaluation on both synthetic data and data from the Star Schema Benchmark, we show that inverted indexes are more compact than bitmap indexes in almost all cases. This compactness combined with efficient query processing strategies results in inverted indexes outperforming bitmap indexes for most queries, often significantly.

Categories and Subject Descriptors: H.2.2 Database management: Physical design - access methods

General Terms: Experimentation, Performance.

1. INTRODUCTION

Decision Support Systems (DSSs) support queries over large amounts of structured data, and bitmap indexes are often used to improve the efficiency of important query classes involving selection predicates and joins [16, 17].

Bitmap indexes were formerly also used in Information Retrieval (IR), but are today mainly replaced by *inverted indexes*. Part of the reason why inverted indexes gained popularity in IR was that they easily support integrating new fields required to support ranked queries. The switch from bitmap indexes to inverted indexes lead to a flood of research on efficient inverted indexes [25, 30, 6, 13, 24, 3, 31, 29], and inverted indexes are now the preferred indexing method in search engines [30].

In this paper, we are asking (and answering) the question: What are the trade-offs of using inverted indexes in DSSs, and should they be considered a serious alternative to bitmap indexes? The main contributions of this paper are (1) the study of how to use and implement inverted indexes in DSSs, and (2) a thorough performance evaluation

that compares inverted indexes and bitmap indexes in DSSs. In particular, we compare inverted indexes with FastBit,¹ a state-of-the-art bitmap query processing and indexing system based on WAH-compressed bitmap indexes [27].

2. BACKGROUND

A standard bitmap index has one bitmap per distinct value for the indexed attribute, with 1's at positions for tuples with the represented value, and 0's elsewhere. Bitmaps can be combined using bitwise operators to answer complex boolean queries. For attributes with few distinct values, bitmap indexes are relatively compact, but their space usage increases linearly with the cardinality. One approach to limit the space usage of bitmap indexes for high cardinality attributes is compression. WAH [27] is one of several introduced compression schemes. Although there are schemes with more compact indexes, WAH supports efficient query processing. This combined with the fact that FastBit is openly available motivates the use of WAH-compressed bitmap indexes in the experiments in this paper.

WAH-compression is a form of word-aligned run-length encoding for bitmap indexes, where consecutive words containing only 0's or 1's are stored as fill words, and other words are stored literally [26, 27]. WAH-compressed bitmaps for high cardinality attributes are relatively compact because most words contain only 0's.

In IR, inverted indexes consist of a search structure for all searchable words called a dictionary, and lists of references to documents containing each searchable word, called inverted lists. An inverted index for an attribute in a DSS consists of a dictionary of the distinct values in the attribute, with pointers to inverted lists that reference tuples with the given value through tuple identifiers (TIDs). To reduce both space usage and the I/O requirements in query processing, the inverted lists are often compressed by storing the deltas between the sorted references [30]. This approach makes small values more likely, and several compression schemes that represent small values compactly have been suggested. According to a recent study, PForDelta [31] is currently the most efficient method [29], and is therefore used in this paper. PForDelta stores deltas in a word-aligned version of bit packing, which also includes exceptions to enable storing larger values than the chosen number of bits allows [31].

Two overall query processing approaches exist in search engines. Document-at-a-time strategies avoid materializing

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

CIKM'09, November 2–6, 2009, Hong Kong, China.

Copyright 2009 ACM 978-1-60558-512-3/09/11 ...\$10.00.

¹<http://sdm.lbl.gov/fastbit/>

intermediate results by processing all inverted lists in a query in parallel [6, 24], and are well suited for boolean query processing. They can be combined with skipping, which is used in search engines to avoid reading and decompressing parts of inverted lists that are not required to process a query [13]. We give a brief description of how we use these ideas in the query processing in this paper in the next section.

3. QUERY PROCESSING

Recall that we use document-at-a-time strategies that avoid materializing intermediate results to process inverted index queries in this paper. We support three operators which can be combined to answer complex queries. They all support skipping to the next result with a given minimum TID value, in addition to standard Volcano-style iteration [9].

The **SCAN** operator can iterate through an inverted list. To support skipping, each k th TID in each inverted list is stored in an external list. The external list is kept in memory during scans, and supports binary searches to find the correct part of the inverted list to process when skipping.

The **OR** operator provides an iterator interface over the sorted merge of its multiple input iterators. The iterators are organized in a priority-queue based on a heap, which is maintained to make sure that the input with the smallest next TID is at the top. Skipping in the **OR** operator is based on a breadth-first search in the heap. A skip may not result in an actual skip for a given input iterator. If so, we know that neither of its children in the heap can do any skipping either, and we therefore avoid testing. After the search, we make sure that only the part of the heap involving iterators that actually skipped is maintained. This approach is reasonably efficient when actually performing skips in both large and small fractions of the set of input iterators.

The **AND** operator expects that the input iterators are sorted in ascending order according to the expected number of returned results. To find the next result, we start with a candidate from the iterator with the fewest number of expected results. We then try to skip to the candidate in the other input iterators, re-starting with a new candidate if the current candidate is absent in one iterator. A candidate found in all inputs is returned as a result. To support skipping, we start with the value to skip to as the candidate and proceed as in normal iteration.

4. EXPERIMENTS

To investigate the trade-offs between inverted indexes and bitmaps, we experiment with FastBit and our inverted index solutions with and without support for skipping. We present results from experiments with synthetic data and data from the Star Schema Benchmark (SSB) [14].

All experiments are run on a quad-core Intel Xeon CPU at 3.2 GHz with 16 GB main memory. All indexes are stored on disk, but queries are run warm by performing 10 runs during one invocation of the system, and reporting the average from the last 8, thus measuring the in-memory query processing performance. We run FastBit version 1.0.5 (implemented in C++), with extra stack space to enable processing queries with many operands. Our approaches are implemented in Java (version 1.6). We use additional warm-up for our system to enable run-time optimizations in the Java virtual machine that reduce variance between runs. Additional warm-up did not change the performance for FastBit.

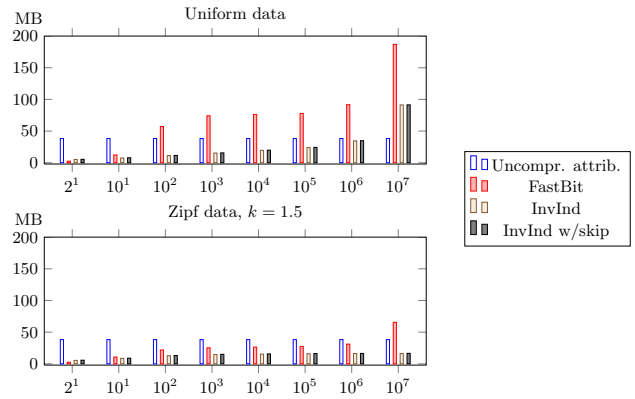


Figure 1: Index sizes. X-axis gives attribute cardinality.

4.1 Synthetic Data

To experiment with synthetic data we generate two tables. Both tables have 10 million tuples and 8 indexed attributes with maximum cardinalities ranging from 2 via all powers of 10 up to 10 million. The attributes in the first table follow a uniform distribution, while a Zipf distribution (with $k = 1.5$) is used in the other.

4.1.1 Index Size

The sizes of the uncompressed attributes in the synthetic tables and their indexes are shown in Figure 1. When using standard PForDelta compression on the attribute with cardinality 2 in the table with uniform data, each value is represented with 4 bits in the most compact index. The reason why a lower number of bits results in a larger index is that the implementation of PForDelta might result in artificial extra exceptions when using a small number of bits per value [31]. Bitmap indexes are known to be compact when the cardinality is 2, and FastBit outperforms our approaches in this case. PForDelta results in compact indexes for higher cardinality attributes, and most of the space usage for the highest cardinality attribute comes from the dictionary (62 out of 91MB). The WAH-compressed bitmaps for the same attribute can in worst case contain nearly 3 computer words per tuple, resulting in a space usage of over 228MB on a 64-bit architecture. The actual results are significantly better, but compressed inverted indexes are clearly more compact.

Indexes for Zipf distributed attributes are more compact than for uniformly distributed attributes with the same maximum cardinality, because skewed distributions make it less likely for the actual cardinality to be equal to the maximum.

4.1.2 Query processing

To experiment with query processing performance, we test four different query types which all vary the attribute on which there is a single value predicate:

1. **Query type SCAN:** Finds all tuples with value 0 for a varied attribute.
2. **Query type skewed AND:** Finds all tuples having value 0 for the attribute with cardinality 10, in addition to 0 in one other varied attribute.
3. **Query type OR:** Finds all tuples having values in the lower half range for a varied attribute.
4. **Query type AND-OR:** Finds all tuples with value in

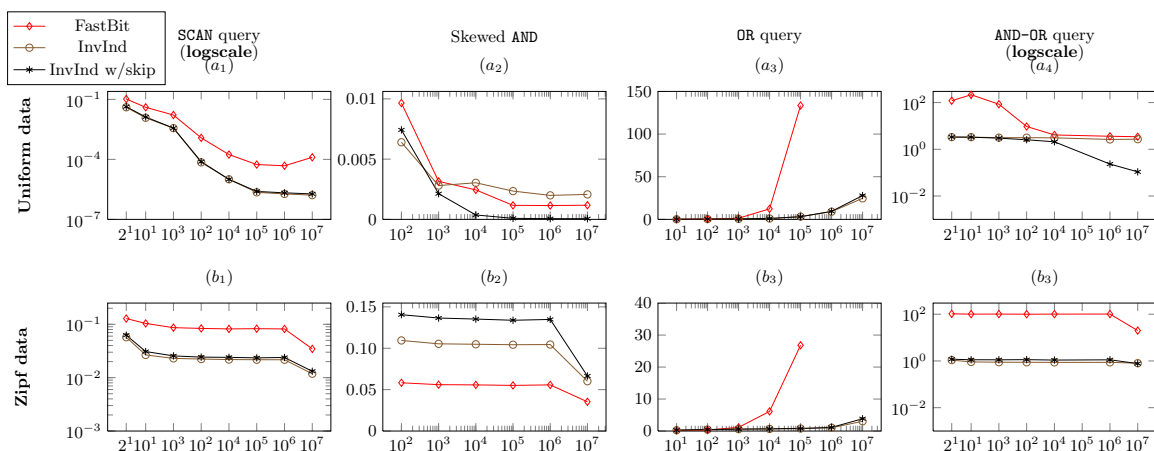


Figure 2: Results from running queries on generated tables, showing query time in seconds for varying cardinalities.

the lower half range for the attribute with cardinality 100000, and value 0 for another varied attribute.

All queries compute the sum of the primary keys of the matching tuples, to ensure that the output from the index is used to perform table look-ups. In the table with uniform distributions, there were no tuples with value 0 for the highest cardinality attribute, so all single valued predicates on this attribute was changed to require the value to be 2. The results are shown in Figure 2.

Compared to bitmaps, decompressed inverted lists are well suited for looking up other attributes for the qualifying tuples, a factor contributing to faster scans for uniform data. The difference in index size also seems to have an impact. All scans are relatively slow for Zipfian data because we always search for the most common attribute value in a skewed distribution, except for in the highest cardinality attribute as noted above.

Skewed AND favors methods capable of taking advantage of the different density in the operands. Inverted indexes with skipping are therefore efficient for uniform data, but introduce overhead for Zipfian data because both operands are dense when the most common values in skewed distributions are accessed. FastBit performs well on dense operands, both because it can combine multiple logical TIDs using one CPU instruction, and because it performs the operator before extracting the tuple references. Because neither input is smaller than the output for AND operators, FastBit decodes fewer references compared to the inverted indexes.

The multi-way OR operators in our solution demonstrate better scalability than FastBit with respect to the number of inputs for both tables.

The idea of skipping in OR operators is ideally suited for query type AND-OR, but it is only useful when the other operands to the AND return data that enables reasonable skip lengths, which occurs for high cardinality attributes with uniform distributions.

4.2 Star Schema Benchmark

Star schemas represent a best practice for how to organize data in decision support systems, and are characterized by a central fact table that references several smaller dimension tables. Typical queries on such schemas involve joins of the fact table with relevant dimension tables called star joins.

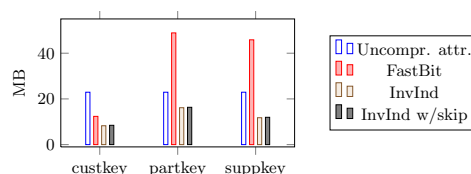


Figure 3: Size of indexes for foreign key columns in SSB

Bitmap indexes can be constructed over the foreign keys in the fact table to speed up such joins, and are then called join indexes [16, 17]. We experiment with using inverted indexes as an alternative to bitmap indexes for this purpose in the Star Schema Benchmark (SSB) [14]. We use Scale Factor 1, and pre-calculate the foreign keys that match the queries in SSB. They are submitted as part of the query to the tested systems. We also avoid calculating the exact answer, but rather let all queries return the sum of an attribute of the fact table. This isolates the effects of the indexes while making sure the returned results are suitable for further look-ups in the fact table. There are four dimension tables in SSB, but we avoid constructing join indexes for the **Date** table because FastBit is unable to process the queries involving all tables without a very significant stack size.

4.2.1 Index Size

Figure 3 shows the join index sizes in both systems. FastBit has significantly larger indexes because the foreign keys have relatively high cardinalities. The attribute **custkey** is partly sorted, resulting in longer runs in WAH-compressed indexes, and the relative difference between FastBit and inverted indexes is therefore smaller in that case.

4.2.2 Join Processing

The query processing results for SSB are given in Figure 4. Within each set of queries, the predicates on the dimension tables become increasingly selective, making FastBit perform better relative to inverted indexes because the OR operators that combine the tuples representing each qualifying foreign key have fewer inputs. Query 2.3 has skewed input to an AND operator making skipping important for performance. The OR operator providing dense input to the AND

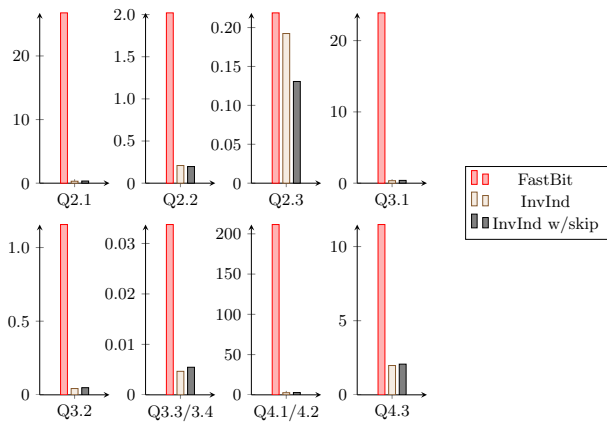


Figure 4: Query processing time in seconds for SSB queries.

is over the attribute with the lowest cardinality, contributing to the smaller performance difference between FastBit and inverted indexes for this query.

5. RELATED WORK

Several alternatives to the compression schemes discussed in this paper have been suggested both for bitmaps [4, 10, 5, 15, 21] and inverted indexes [25, 20, 1]. Experiments have shown that the query processing efficiency of WAH remains attractive, even though there are approaches resulting in smaller indexes. WAH is known to result in smaller indexes when the table is sorted on the indexed attribute [19, 12]. Due to space restrictions, we do not experiment with sorted tables in this paper. Experiments with compression in inverted indexes in IR have shown that PForDelta currently is the most efficient technique [29], and further improvements to the technique have also been suggested recently [28].

As an alternative to compression, there are several approaches that reduce the number of bitmaps in the index [17, 7, 8, 11, 22]. Strategies for operating on many bitmaps by processing two at a time have been explored for WAH-compressed bitmap indexes [26], and a recent paper suggests using multi-way operators for bitmaps, but the idea is not tested [12]. Query processing approaches in inverted indexes in IR have focused on term-at-a-time strategies in addition to the document-at-a-time approach used in this paper [6, 24, 18, 2, 3, 23].

6. CONCLUSIONS

In this paper, we have evaluated the applicability of compressed inverted indexes as an alternative to bitmap indexes in DSSs. Inverted indexes are generally significantly more space efficient. The only case where WAH-compressed bitmaps are clearly more compact is when the cardinality of the indexed attribute is very low. FastBit performs well on simple queries with dense operands, but inverted indexes are better in other cases, often significantly.

Acknowledgments: This material is based upon work supported by New York State Science Technology and Academic Research under agreement number C050061, by Grant NFR 162349, by the National Science Foundation under Grants 0534404 and 0627680, and by the iAd Project funded by the Research Council of Norway. Any opinions, findings

and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NYSTAR, the National Science Foundation or the Research Council of Norway.

7. REFERENCES

- [1] V. N. Anh and A. Moffat. Index compression using fixed binary codewords. In *Proc. ADC*, 2004.
- [2] V. N. Anh and A. Moffat. Simplified similarity scoring using term ranks. In *Proc. SIGIR*, 2005.
- [3] V. N. Anh and A. Moffat. Pruned query evaluation using pre-computed impacts. In *Proc. SIGIR*, 2006.
- [4] G. Antoshenkov. Byte-aligned bitmap compression. In *Proc. DCC*, 1995.
- [5] T. Apaydin, G. Canahuate, H. Ferhatosmanoglu, and A. S. Tosun. Approximate encoding for direct access and query processing over compressed bitmaps. In *Proc. VLDB*, 2006.
- [6] E. W. Brown. Fast evaluation of structured queries for information retrieval. In *Proc. SIGIR*, 1995.
- [7] C.-Y. Chan and Y. E. Ioannidis. Bitmap index design and evaluation. *SIGMOD Rec.*, 1998.
- [8] C.-Y. Chan and Y. E. Ioannidis. An efficient bitmap encoding scheme for selection queries. In *Proc. SIGMOD*, 1999.
- [9] G. Graefe. Volcano – an extensible and parallel query evaluation system. *IEEE Trans. on Knowl. and Data Eng.*, 1994.
- [10] T. Johnson. Performance measurements of compressed bitmap indices. In *Proc. VLDB*, 1999.
- [11] N. Koudas. Space efficient bitmap indexing. In *Proc. CIKM*, 2000.
- [12] D. Lemire, O. Kaser, and K. Aouiche. Sorting improves word-aligned bitmap indexes. *CoRR*, abs/0901.3751, 2009.
- [13] A. Moffat and J. Zobel. Self-indexing inverted files for fast text retrieval. *ACM Trans. Inf. Syst.*, 1996.
- [14] E. O’Neil, P. O’Neil, and X. Chen. <http://www.cs.umb.edu/~poneil/StarSchemaB.PDF>.
- [15] E. O’Neil, P. O’Neil, and K. Wu. Bitmap index design choices and their performance implications. In *Proc. IDEAS*, 2007.
- [16] P. O’Neil and G. Graefe. Multi-table joins through bitmapped join indices. *SIGMOD Rec.*, 1995.
- [17] P. O’Neil and D. Quass. Improved query performance with variant indexes. In *Proc. SIGMOD*, 1997.
- [18] M. Persin, J. Zobel, and R. Sacks-Davis. Filtered document retrieval with frequency-sorted indexes. *J. Am. Soc. Inf. Sci.*, 1996.
- [19] A. Pinar, T. Tao, and H. Ferhatosmanoglu. Compressing bitmap indices by data reorganization. In *Proc. ICDE*, 2005.
- [20] F. Scholer, H. E. Williams, J. Yiannis, and J. Zobel. Compression of inverted indexes for fast query evaluation. In *Proc. SIGIR*, 2002.
- [21] M. Stabno and R. Wrembel. RLH: Bitmap compression technique based on run-length and Huffman encoding. *Information Systems*, 2008.
- [22] K. Stockinger, K. Wu, and A. Shoshani. Evaluation strategies for bitmap indices with binning. In *Proc. Database and Expert Systems Applications*, 2004.
- [23] T. Strohmman and W. B. Croft. Efficient document retrieval in main memory. In *Proc. SIGIR*, 2007.
- [24] T. Strohmman, H. Turtle, and W. B. Croft. Optimization strategies for complex queries. In *Proc. SIGIR*, 2005.
- [25] I. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes*. Academic Press, 1999.
- [26] K. Wu, E. Otoo, and A. Shoshani. On the performance of bitmap indices for high cardinality attributes. In *Proc. VLDB*, 2004.
- [27] K. Wu, E. J. Otoo, and A. Shoshani. Optimizing bitmap indices with efficient compression. *ACM Trans. Database Syst.*, 2006.
- [28] H. Yan, S. Ding, and T. Suel. Inverted index compression and query processing with optimized document ordering. In *Proc. WWW*, 2009.
- [29] J. Zhang, X. Long, and T. Suel. Performance of compressed inverted list caching in search engines. In *Proc. WWW*, 2008.
- [30] J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Comput. Surv.*, 2006.
- [31] M. Zukowski, S. Heman, N. Nes, and P. Boncz. Super-scalar RAM-CPU cache compression. In *Proc. ICDE*, 2006.