WIKIPEDIA

Bitmap index

A **bitmap index** is a special kind of database index that uses bitmaps.

Bitmap indexes have traditionally been considered to work well for *low-cardinality columns*, which have a modest number of distinct values, either absolutely, or relative to the number of records that contain the data. The extreme case of low cardinality is Boolean data (e.g., does a resident in a city have internet access?), which has two values, True and False. Bitmap indexes use bit arrays (commonly called bitmaps) and answer queries by performing bitwise logical operations on these bitmaps. Bitmap indexes have a significant space and performance advantage over other structures for query of such data. Their drawback is they are less efficient than the traditional B-tree indexes for columns whose data is frequently updated: consequently, they are more often employed in read-only systems that are specialized for fast query - e.g., data warehouses, and generally unsuitable for online transaction processing applications.

Some researchers argue that bitmap indexes are also useful for moderate or even high-cardinality data (e.g., unique-valued data) which is accessed in a read-only manner, and queries access multiple bitmap-indexed columns using the AND, OR or XOR operators extensively.[1]

Bitmap indexes are also useful in data warehousing applications for joining a large fact table to smaller dimension tables such as those arranged in a star schema.

Bitmap based representation can also be used for representing a data structure which is labeled and directed attributed multigraph, used for queries in graph databases.`Efficient graph management based on bitmap indices (https://www.researchgate.net/publication/236593640_Efficient_graph_management_based_on_bitmap_indices)` article shows how bitmap index representation can be used to manage large dataset (billions of data points) and answer queries related to graph efficiently.

Contents

Example

Continuing the internet access example, a bitmap index may be logically viewed as follows:

On the left, Identifier refers to the unique number assigned to each resident, HasInternet is the data to be indexed, the content of the bitmap index is shown as two columns under the heading *bitmaps*. Each column in the left illustration is a *bitmap* in the bitmap index. In this case, there are two such bitmaps, one for "has internet" *Yes* and one for "has internet"

Identifier	HasInternet	Bitmaps	
		Y	N
1	Yes	1	0
2	No	0	1
3	No	0	1
4	Unspecified	0	0
5	Yes	1	0

No. It is easy to see that each bit in bitmap *Y* shows whether a particular row refers to a person who has internet access. This is the simplest form of bitmap index. Most columns will have more distinct values. For example, the sales amount is likely to have a much larger number of distinct values. Variations on the bitmap index can effectively index this data as well. We briefly review three such variations.

Note: Many of the references cited here are reviewed at (John Wu (2007)).[2] For those who might be interested in experimenting with some of the ideas mentioned here, many of them are implemented in open source software such as FastBit,[3] the Lemur Bitmap Index C++ Library,[4] the Roaring Bitmap Java library[5] and the Apache Hive Data Warehouse system.

Compression

Software can compress each bitmap in a bitmap index to save space. There has been considerable amount of work on this subject.[6][7] Though there are exceptions such as Roaring bitmaps,[8] Bitmap compression algorithms typically employ run-length encoding, such as the Byte-aligned Bitmap Code,[9] the Word-Aligned Hybrid code,[10] the Partitioned Word-Aligned Hybrid (PWAH) compression,[11] the Position List Word Aligned Hybrid,[12] the Compressed Adaptive Index (COMPAX),[13] Enhanced Word-Aligned Hybrid (EWAH) [14] and the COmpressed 'N' Composable Integer SEt.[15][16] These compression methods require very little effort to compress and decompress. More importantly, bitmaps compressed with BBC, WAH, COMPAX, PLWAH, EWAH and CONCISE can directly participate in bitwise operations without decompression. This gives them considerable advantages over generic compression techniques such as LZ77. BBC compression and its derivatives are used in a commercial database management system. BBC is effective in both reducing index sizes and maintaining query performance. BBC encodes the bitmaps in bytes, while WAH encodes in words, better matching current CPUs. "On both synthetic data and real application data, the new word aligned schemes use only 50% more space, but perform logical operations on compressed data 12 times faster than BBC."[17] PLWAH bitmaps were reported to take 50% of the storage space consumed by WAH bitmaps and offer up to 20% faster performance on logical operations.[12] Similar considerations can be done for CONCISE [16] and Enhanced Word-Aligned Hybrid.[14]

The performance of schemes such as BBC, WAH, PLWAH, EWAH, COMPAX and CONCISE is dependent on the order of the rows. A simple lexicographical sort can divide the index size by 9 and make indexes several times faster.[18] The larger the table, the more important it is to sort the rows. Reshuffling techniques have also been proposed to achieve the same results of sorting when indexing streaming data.[13]

Encoding

Basic bitmap indexes use one bitmap for each distinct value. It is possible to reduce the number of bitmaps used by using a different encoding method.[19][20] For example, it is possible to encode C distinct values using log(C) bitmaps with binary encoding.[21]

This reduces the number of bitmaps, further saving space, but to answer any query, most of the bitmaps have to be accessed. This makes it potentially not as effective as scanning a vertical projection of the base data, also known as a materialized view or projection index. Finding the optimal encoding method that balances (arbitrary) query performance, index size and index maintenance remains a challenge.

Without considering compression, Chan and Ioannidis analyzed a class of multi-component encoding methods and came to the conclusion that two-component encoding sits at the kink of the performance vs. index size curve and therefore represents the best trade-off between index size and query performance.[19]

Binning

For high-cardinality columns, it is useful to bin the values, where each bin covers multiple values and build the bitmaps to represent the values in each bin. This approach reduces the number of bitmaps used regardless of encoding method.[22] However, binned indexes can only answer some queries without examining the base data. For example, if a bin covers the range from 0.1 to 0.2, then when the user asks for all values less than 0.15, all rows that fall in the bin are possible hits and have to be checked to verify whether they are actually less than 0.15. The process of checking the base data is known as the candidate check. In most cases, the time used by the candidate check is significantly longer than the time needed to work with the bitmap index. Therefore, binned indexes exhibit irregular performance. They can be very fast for some queries, but much slower if the query does not exactly match a bin.

History

The concept of bitmap index was first introduced by Professor Israel Spiegler and Rafi Maayan in their research "Storage and Retrieval Considerations of Binary Data Bases", published in 1985.[23] The first commercial database product to implement a bitmap index was Computer Corporation of America's Model 204. Patrick O'Neil published a paper about this implementation in 1987.[24] This implementation is a hybrid between the basic bitmap index (without compression) and the list of Row Identifiers (RID-list). Overall, the index is organized as a B+tree. When the column cardinality is low, each leaf node of the B-tree would contain long list of RIDs. In this case, it requires less space to represent the RID-lists as bitmaps. Since each bitmap represents one distinct value, this is the basic bitmap index. As the column cardinality increases, each bitmap becomes sparse and it may take more disk space to store the bitmaps than to store the same content as RID-lists. In this case, it switches to use the RID-lists, which makes it a B+tree index.[25][26]

In-memory bitmaps

One of the strongest reasons for using bitmap indexes is that the intermediate results produced from them are also bitmaps and can be efficiently reused in further operations to answer more complex queries. Many programming languages support this as a bit array data structure. For example, Java has the `BitSet (http://download.oracle.com/javase/6/docs/api/java/util/BitSet.html)` class.

Some database systems that do not offer persistent bitmap indexes use bitmaps internally to speed up query processing. For example, PostgreSQL versions 8.1 and later implement a "bitmap index scan" optimization to speed up arbitrarily complex logical operations between available indexes on a single table.

For tables with many columns, the total number of distinct indexes to satisfy all possible queries (with equality filtering conditions on either of the fields) grows very fast, being defined by this formula:

$$\mathbf{C}_n^{\left[\frac{n}{2}\right]} \equiv \frac{n!}{\left(n - \left[\frac{n}{2}\right]\right)! \left[\frac{n}{2}\right]!}.$$ [27][28]

A bitmap index scan combines expressions on different indexes, thus requiring only one index per column to support all possible queries on a table.

Applying this access strategy to B-tree indexes can also combine range queries on multiple columns. In this approach, a temporary in-memory bitmap is created with one bit for each row in the table (1 MiB can thus store over 8 million entries). Next, the results from each index are combined into the bitmap using bitwise operations. After all conditions are evaluated, the bitmap contains a "1" for rows that matched the expression. Finally, the bitmap is traversed and matching rows are retrieved. In addition to efficiently combining indexes, this also improves locality of reference of table accesses, because all rows are fetched sequentially from the main table.[29] The internal bitmap is discarded after the query. If there are too many rows in the table to use 1 bit per row, a "lossy" bitmap is created instead, with a single bit per disk page. In this case, the bitmap is just used to determine which pages to fetch; the filter criteria are then applied to all rows in matching pages.

References

Notes

1. Bitmap Index vs. B-tree Index: Which and When? (http://www.oracle.com/technetwork/articles/sharma-indexes-093638.html), Vivek Sharma, Oracle Technical Network.
2. John Wu (2007). "Annotated References on Bitmap Index" (http://www.cs.umn.edu/~kewu/annotated.html).
3. FastBit (http://codeforge.lbl.gov/projects/fastbit/)
4. Lemur Bitmap Index C++ Library (https://code.google.com/p/lemurbitmapindex/)
5. Roaring bitmaps (http://roaringbitmap.org/)
6. T. Johnson (1999). "Performance Measurements of Compressed Bitmap Indices". In Malcolm P. Atkinson; Maria E. Orlowska; Patrick Valduriez; Stanley B. Zdonik; Michael L. Brodie. *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7–10, 1999, Edinburgh, Scotland, UK* (http://www.vldb.org/conf/1999/P29.pdf) (PDF). Morgan Kaufmann. pp. 278–89. ISBN 1-55860-615-7.
7. Wu K, Otoo E, Shoshani A (March 5, 2004). "On the performance of bitmap indices for high cardinality attributes" (http://www.osti.gov/energycitations/servlets/purl/822860-LOzkmz/native/822860.pdf) (PDF).
8. Chambi, S.; Lemire, D.; Kaser, O.; Godin, R. (2016). "Better bitmap performance with Roaring bitmaps". *Software: Practice & Experience*. **46**: 5. doi:10.1002/spe.2325 (https://doi.org/10.1002%2Fspe.2325).
9. Byte aligned data compression (https://www.google.com/patents/US5363098)
10. Word aligned bitmap compression method, data structure, and apparatus (https://www.google.com/patents/US6831575)
11. van Schaik, Sebastiaan; de Moor, Oege (2011). "A memory efficient reachability data structure through bit vector compression" (http://dl.acm.org/citation.cfm?doid=1989323.1989419). *Proceedings of the 2011 international conference on Management of data*. SIGMOD '11. Athens, Greece: ACM. pp. 913–924. doi:10.1145/1989323.1989419 (https://doi.org/10.1145%2F1989323.1989419). ISBN 978-1-4503-0661-4.
12. Deliège F, Pedersen TB (2010). "Position list word aligned hybrid: optimizing space and performance for compressed bitmaps". In Ioana Manolescu, Stefano Spaccapietra, Jens Teubner, Masaru Kitsuregawa, Alain Leger, Felix Naumann, Anastasia Ailamaki, Fatma Ozcan. *EDBT '10, Proceedings of the 13th International Conference on Extending Database Technology* (http://alpha.uhasselt.be/icdt/edbticdt2010proc/edbt/papers/p0228-Deliege.pdf) (PDF). New York, NY, USA: ACM. pp. 228–39. doi:10.1145/1739041.1739071 (https://doi.org/10.1145%2F1739041.1739071). ISBN 978-1-60558-945-9.
13. F. Fusco; M. Stoecklin; M. Vlachos (September 2010). "NET-FLi: on-the-fly compression, archiving and indexing of streaming network traffic" (http://www.comp.nus.edu.sg/~vldb2010/proceedings/files/papers/I01.pdf) (PDF). *Proc. VLDB Endow.* **3** (1–2): 1382–93.
14. Lemire, D.; Kaser, O.; Aouiche, K. (2010). "Sorting improves word-aligned bitmap indexes". *Data & Knowledge Engineering*. **69**: 3. doi:10.1016/j.datak.2009.08.006 (https://doi.org/10.1016%2Fj.datak.2009.08.006).