Optimizing Selection Performance on Scientific Data by utilizing Bitmap Indices

H. Schmücker, CERN, Geneva, Switzerland

Abstract

Bitmap indices have gained wide acceptance in data warehouse applications handling large amounts of read only data. High dimensional ad hoc queries can be efficiently performed by utilizing bitmap indices, especially if the queries cover only a subset of the attributes stored in the database. Such access patterns are common use in HEP analysis. Bitmap indices have been implemented by several commercial database management systems. However, the provided query algorithms focus on typical business applications, which are based on discrete attributes with low cardinality. HEP data, which are mostly characterized by non discrete attributes, cannot be queried efficiently by these implementations.

Support for selections on continuously distributed data can be added to the bitmap index technique by extending it with an adaptive binning mechanism. Following this approach a prototype has been implemented, which provides the infrastructure to perform index based selections on HEP analysis data stored in ROOT trees/tuples. For the indices a range encoded design with multiple components has been chosen. This design concept allows to realize a very fine binning granularity, which is crucial to selection performance, with an index of reasonable size. Systematic performance tests have shown that the query processing time and the disk-I/O can be significantly reduced compared to a conventional scan of the data. This especially applies to optimization scenarios in HEP analysis, where selections are slightly varied and performed repetitively on one and same data sample.

INTRODUCTION

Queries in physics analysis commonly exhibit the following characteristics: They are multi-dimensional and in a lot of cases they cover only a relative small subset of a much larger number of attributes stored in the queried database. They are ad hoc, i.e. the attribute subsets involved in the queries are not known a priori and the attribute combinations might change frequently. Furthermore, the queried quantities involve floating point attributes with large cardinalities. Although there are a number of indexing schemes designed to speed up multi-dimensional queries, most selections in physics analysis are evaluated by a sequential data scan without using an index, because none of the currently available index implementations meets all requirements of the above query pattern. Most tree or hash based multi-dimensional index schemes suffer from the "curse of dimensionality", i.e. either the index size or the query processing time grows super linearly with the number of indexed attributes. So these indices can only be utilized for relatively small attribute subsets that are known a priori, which is not always the case in interactive analysis sessions. Bitmap indices are perfectly suited for high dimensional ad hoc queries, but current implementations are space efficient only for attributes with a low cardinality. For attributes with high cardinality ($C \gg 100$), which are quite common in physics analysis, their size is out of scale. Possible approaches to reduce the index size are bitmap compression, binning and bitmap encoding. The technique presented in this paper uses a combination of binning and bitmap encoding to realize a bitmap index of reasonable size for attributes with high cardinality, a range encoded bitmap index with multiple components and binning.

BITMAP INDICES

The basic principle of a bitmap index is to store a bit vector for each distinct value of either an attribute or an expression involving several attributes. Each bit in these bit vectors addresses a record of the primary data sample, e.g a row in a table or a persistent object in a database. The associated bit is set if and only if the record fulfills the property in focus, e.g. the respective value of the record's attribute is equal to a certain value. The bit vectors mapped to all possible values of a given attribute form a matrix — the bitmap index. A major advantage of bitmap indices is that multi-dimensional queries are performed by hardware supported bit-wise logical combinations of the bit vectors.

A detailed discussion on the design of bitmap indices is presented by Chan and Ioannidis [1, 2]. The most fundamental encoding scheme is equality encoding. Here the i^{th} bit in the bit vector associated with the value v is set if and only if the indexed attribute is equal to v in the i^{th} record. An example is shown in table 1. This type of index is optimal for exact match queries, since the result of an equality check is given directly by the according bit vector. The evaluation of range queries requires the scan of more bit vectors. In the worst case half of the index has to be read. The optimal solution for range queries are range encoded indices. Here a bit is set if the attribute value is less or equal the constant associated with the bit vector. But also exact match queries can be efficiently performed using this index type. The result is given by a logical XOR combination ($\oplus$) of two adjacent bit vectors. In the example of figure 1 the query "$A = 5$" is evaluated by $R_5 \oplus R_4$.

The size of a equality encoded index matrix is given by the product of the attribute's cardinality C (number of bit vectors (range encoding: $C - 1$)) and the number of indexed records (length of the bit vectors). In case of non discrete floating point attributes the cardinality is of the same order of magnitude as the number of records. So the in-

Attribute Value	Equality Encoding									Range Encoding							
	E_8	E_7	E_6	E_5	E_4	E_3	E_2	E_1	E_0	R_7	R_6	R_5	R_4	R_3	R_2	R_1	R_0
3	0	0	0	0	0	**1**	0	0	0	**1**	**1**	**1**	**1**	**1**	0	0	0
2	0	0	0	0	0	0	**1**	0	0	**1**	**1**	**1**	**1**	**1**	**1**	0	0
0	0	0	0	0	0	0	0	0	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**
8	**1**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	0	0	**1**	0	0	0	0	0	0	**1**	**1**	0	0	0	0	0	0
5	0	0	0	**1**	0	0	0	0	0	**1**	**1**	**1**	0	0	0	0	0

Figure 1: Example of an equality encoded and range encoded bitmap index for a sample of discrete attributes with cardinality $C = 9$, integers within the interval $[0, 8]$.

Attribute Values	Intervals represented by the bins					
	[0;120]	[0;100]	[0;80]	[0;60]	[0;40]	[0;20]
	R_5	R_4	R_3	R_2	R_1	R_0
34.7	1	1	1	1	1	0
94.0	1	1	0	0	0	0
64.9	1	1	1	0	0	0
15.5	1	1	1	1	1	1
61.7	1	1	1	0	0	0
137.2	0	0	0	0	0	0

Figure 2: Example of a range encode bitmap index for floating point attributes in the interval [0;140] (equidistant binning). A bit is set if the attribute value is less or equal the upper bin edge.

dex size $S = C \cdot N$ grows quadratically with the number of records in the database. Bitmap compression can limit this growth to a linear behavior. A detailed discussion on bitmap compression is presented by Shoshani, Stockinger and Wu [3, 5]. They show that in many cases an index size of twice the raw data size is achieved and that an upper bound of a factor of 6 is never exceeded. However compression can be applied efficiently only to the sparse bit vectors of an equality encoded index. The compressibility of range encoded bit vectors is much lower, especially in the middle of the index matrix, where the bit density is close to 50%.

Binning

The index size can be significantly reduced, when the attribute values are partitioned into bins before the index is created. In this case a bit vector does not represent a distinct attribute value but a range of attribute values. Hence the selection result provided by the index is not exact and has to be validated by a partial scan of the primary data. Consider a selection "$A < 63.0$" using the example of a binned range encoded index shown in figure 2: The bit vector R_3, referred to as candidate vector, addresses all records that fulfill the query and a few more that need to be rejected. A XOR combination of R_3 with R_2 (referred to as the hit vector) removes those records from R_3 that definitely matches the query. Only the remaining entries in R_3 needs to be validated by inspecting the primary data. In case of a conjunction of queries the candidate and hit vectors of each dimension are combined by logical AND operations to a global candidate and hit vector. Then the hits are removed from the global candidate vector via a XOR combination with the global hit vector. The remaining candidates are checked separately for each dimension, first by a validation via the according hit vector and then, if this is unsuccessful, by inspecting the original data. A more detailed discussion on evaluation strategies using binned bitmap indices can be fount in [4].

The amount of primary data, that has to be scanned, depends on the page size of the database, the query dimension, the selectivity and the binning granularity. For sparse and high dimensional selections a sufficient reduction of primary data, that need to be scanned, is achieved with a relatively broad binning. If either the query dimension or the selectivity is high, a very fine binning granularity, up to 10000 bins, is required. A basic bitmap index addressing this large number of bins would be out of scale (10000 index bits per 32-bit attribute value). One solution of this storage overhead problem is bitmap compression. Another promising approach is the use of a multi component index. In contrast to compression algorithms, which are only efficient for equality encoded indices, this bitmap encoding technique significantly reduce the size of both, equality encoded and range encoded indices.

Multi Component Indices

The basic idea of multi component indices [1] is to decompose the attribute values into digits according to some base and to create a basic bitmap index for each of these digits separately. The result of this encoding is a very compact index. E.g. a 3-component base–$< 10, 10, 10 >$ range encoded index, that addresses 1000 bins of a 32-bit floating point attribute, is smaller than the original data set: For each 32-bit attribute value $3 * 9 + 2 = 29$ index bits have to be stored (2 bits for over- and underflow bins). On the other hand the selection process using a multi component index is more complex compared to a basic bitmap index. In principle the same selection algorithms apply as for basic indices, but they have to be executed on each index component separately. E.g., a range query $A \leq 57$ using a base–$< 10, 10 >$ range encoded index decomposes to $A_2A_1 \leq 5_{10}7_{10}$ and can be rewritten as $(A_2 \leq 4) \vee [(A_2 \leq 5) \wedge (A_1 \leq 7)]$. In general, for a n-component range encoded index with binning at most $2n$ bit slices are involved in the query evaluation.

PERFORMANCE TESTS

Prototype

The prototype used for the performance tests has been implemented based on the popular Root framework [6]. Data stored in Root-*TTrees* can be indexed using either basic or multi component range encoded indices with and w/o binning. The indices are stored in separate *TTrees* and can be compressed using Root's zip algorithm. An adaptive binning method ensures that each bit vector of a binned index addresses a similar numbers of data records. The algorithm includes a spike search and for attributes with low cardinality it automatically switches to a discrete mode without binning. The indices can be created in user definable intervals for

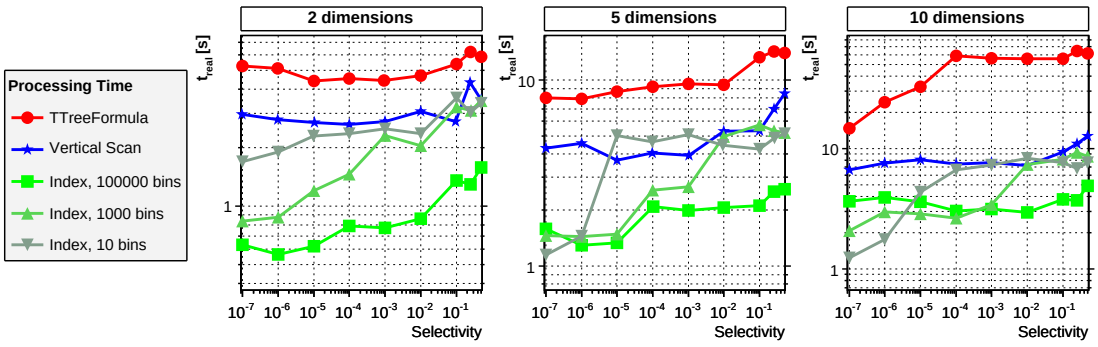

Figure 3: Response time for multi-dimensional queries against uniformly distributed random data

almost any expression accepted by *TTreeFormula*, e.g. `sqrt(tracks[].px**2+tracks[].py**2)`. Also the query engine accepts *TTreeFormula*-like selection strings. Queries can be composed using all C++ comparative and logical operators. One limitation is of course that indexed expression should be compared to constant values, in order to utilize the according index. The query engine minimizes the processing time of multi-dimensional selections by rearranging them in that way, that sub queries with low acceptance are evaluated first. Depending on the persistent layout of the Root-*TTree* either a row-wise or column-wise query evaluation strategy can be chosen.

Persistent data layout

The data organization strategy used by most DBMS groups the attributes of each record, a row of a table or an object, next to each other. While this row-wise layout, also referred to as horizontal partitioning, is optimal for transactional operations, it is inefficient for performing selections that involve only a subset of the stored attributes, because the values of a distinct attribute cannot be accessed exclusively and therefore the full database has to be scanned. An obvious alternative is to fill the database in a column-wise way. In such a vertically partitioned database the values of each attribute are stored in consecutive disk pages (or Root-*TBaskets*) and simple multi-dimensional queries, can be performed efficiently by a column-wise scan. However, due to additional disk seeks, this does not apply, when the query involves complex expressions that include more than one attribute, e.g. `sqrt(px**2+py**2)`. Furthermore in most cases column-wise writing of one single large databases is not feasible, because the data is produced in a row-wise manner. A compromise of these two approaches is the strategy used for Root-*TTrees* written in split mode. The *TTree* is filled record by record, but the attribute values are stored in separate *TBaskets*. As a result the values of each attribute can be accessed exclusively, but the according *TBaskets* are not organized consecutively on disk. This fragmentation has indeed an impact on the data scan efficiency.

Systematic Performance Tests

First the prototype has been systematically tested on uniform random data stored uncompressed in a vertically partitioned *TTree* with a *TBasket*-size of 16K. 4 million entries with 100 32-bit floating point attributes have been written. The resulting *TTree*-size is 1.5 GB. Selections on databases of this dimension are common practice in physics analysis. The tested indices with 1, 3 and 5 components of base 10 address 10, 1000 and 100000 bins respectively and have a size of 11, 29 and 47 index bits per 32-bit attribute value. The test have been performed on a 1.4 GHz P4 CPU equipped with 256 MB RAM and a 40 GB IDE disk. The mean index creation times per attribute are 3.7 s, 7.9 s and 15.0 s for the 10-, 1000- and 100000-bin indices. The test queries are conjunctions of 2, 5 and 10 one sided range selections: $A_i \leq x_i \;\&\&\; A_j \leq x_j \ldots$. By randomly varying the query boundary x the selectivities have been adjusted to values ranging from 10^{-7} to 0.5. The file cache has been reset before each query. Figure 3 shows a comparison of the selection processing times of the three indices, Root's *TTreeFormula* query engine and a vertical data scan. As expected the 10-bin index shows good results only for very sparse and high dimensional queries. In all selection scenarios the 100000-bin index outperforms *TTreeFormula* and the vertical data scan. The achieved performance gains ranges from 4 to 18 compared to *TTreeFormula* and from 2 to 4 compared to the vertical data scan.

An important use case in HEP analysis is the optimization of selection cuts, which is characterized by repetitive queries with slightly varied constraints on one and the same subset of data attributes. The performance of bitmap indices in such an optimization scenario has been investigated using a flat TTree with 500000 rows of 10 uniform random 32-bit floating point attributes. The *TTree* has a size of 19 MB and fits completely into memory. The index setup is the same as described in the previous section. The selectivities of the test queries, conjunctions of one sided range selections on all 10 attributes, ranges from 10^{-4} to 0.75. The results of this test are shown in figure 5. Since both, the primary data tree and the index fits completely into memory, the processing times are solely bound by the processor speed. In all cases the index aided queries per-

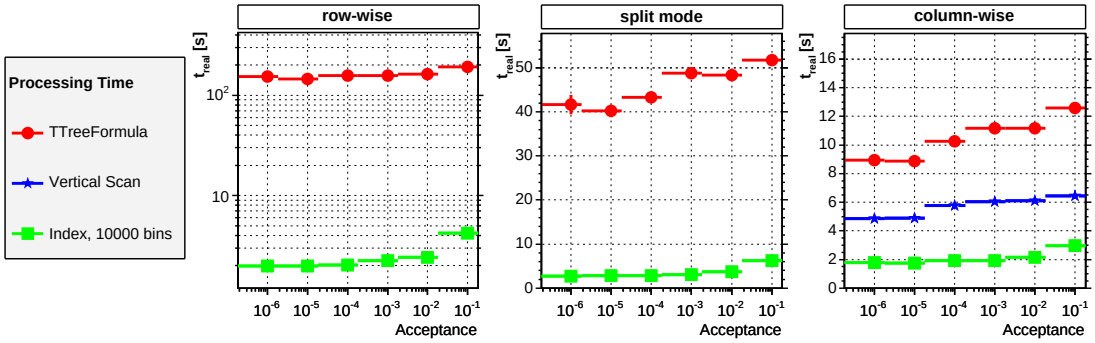

Figure 4: Response time for 5-dimensional queries against event-tag data of different persistent layouts

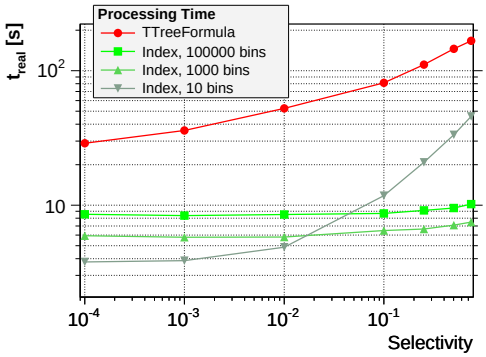

Figure 5: Processing times of repetitive queries on a *TTree* that completely fits into memory.

form better than the conventional scan of the data. Especially queries with a selectivity greater than 10%, which are most important in selection optimization scenarios, are speed up by a factor up to 20 by using the indices.

Querying Physics Data

The prototype has also been tested on an extract of a HEP event tag database, that is used in analysis for fast pre selection based on event summary data. The data sample contains 7.6 million records with 40 4-byte integer and 63 4-byte floating point attributes. The data has been extracted to compressed *TTrees* of different persistent layouts (row-wise, column-wise and split mode) with a *TBasket*-size of 16KB. The resulting *TTree*-size is 1.5 GB (3.0 GB uncompressed). For attributes with high cardinality a 4-component index addressing 10000 bins has been created. For attributes with a cardinality less than 10000 the prototype has automatically set up a direct index of smaller size without binning. The indices have been compressed. The resulting size of the whole index is 2 GB (2.9 GB uncompressed). The index creation took 35 minutes on a 2.4 GHz P4 CPU equipped with 768 MB RAM and a 40 GB IDE disk. The test queries consist of a conjunction of one sided range queries on 5 randomly chosen attributes. In figure 4 the average query processing times are plotted against the selection acceptance. In case of the row-wise *TTree* the index outperforms *TTreeFormula*, which evaluates the query via a sequential data scan, by a factor greater than 50. This result can be interpreted as rough estimate of the performance gain that could be achieved with a relational database. For the *TTree* written in split mode the gain ranges from 8 to 16. Also in the case of the column-wise *TTree* the index is always the fastest technique. It is more than twice as fast than the vertical scan. An interesting result is, that even *TTreeFormula*, which evaluates the query in a row-wise manner, profits from the column-wise persistent data layout. Compared to the results measured with a split *TTree* it performs 4 times faster.

CONCLUSION

Motivated by the need to efficiently process multi-dimensional ad hoc queries on non-discrete data attributes, a prototype, that combines the concept of range encoded multi component bitmap indices with an adaptive binning algorithm, has been implemented. This approach allows to realize efficient bitmap indices for high cardinality attributes, that are not greater than 1.5 times the original data size. Queries against large databases can be speed up by factors ranging from 2 up to 50 depending on the persistent layout of the primary database. Also the processing time of repetitive selections on small data samples, which is a common use case in HEP analysis, is significantly reduced by factors up to 20.

REFERENCES

[1] C.Y. Chan and Y.E. Ioannidis, Bitmap Index Design and Evaluation, In Proceedings of SIGMOD 1998.

[2] C.Y. Chan and Y.E. Ioannidis, An efficient Bitmap Encoding scheme for Selection Queries. In proceedings of SIGMOD 1999.

[3] K. Stockinger, K. Wu, A. Shoshani: Strategies for processing ad hoc queries on large data warehouses. In Proceedings of DOLAP 2002

[4] K. Stockinger, K. Wu, A. Shoshani, Evaluation Strategies for Bitmap Indices with Binning, In Proceedings of DEXA 2004.

[5] K. Wu, E. Otoo, A. Shoshani, An Efficient Compression Scheme For Bitmap Indices, Tech. Report LBNL-49626, 2004

[6] http://root.cern.ch