Optimizing multidimensional Queries using Bitmap Indices

Helmut Schmücker, CERN

- Bitmap indices
  - Introduction
  - Coping high cardinality attributes
- Root-based Prototype
  - Design / Features
  - Performance tests
- Outlook

Motivation

- Queries in physics analysis:
  - E.g. Event tag collections, Ntuple-based analysis
  - Multidimensional (e.g. **`energy>1.5 && NMuon>4 && ...)`** typically include a small subset of a large number of attributes
  - Ad hoc, attribute combinations are not known a priori
  - High cardinality attributes, "continuously" distributed floats
  - In most cases performed by a slow data scan
- Indices ?
  - B-tree, R-tree, Grid-File, ...
    - Efficiency deteriorates at high dimensions, "curse of dimensionality"
    - Specific attribute combinations
  - Bitmap indices:
    - perfectly suited for high dimensional ad hoc queries
    - but current implementations are space efficient only for low cardinality attributes

Basic Bitmap Indices

- Each distinct attribute value is represented by a bit vector:
  Number of bit vectors = attribute cardinality
- Each bit addresses a data record:
  bit vector length = number of data records
- A bit is set if the record fulfills the property in focus

Attribute Value	B_0	B_1	B_2	B_3	B_4	B_5
3	0	0	0	1	0	0
2	0	0	1	0	0	0
4	0	0	0	0	1	0
0	1	0	0	0	0	0
5	0	0	0	0	0	1
1	0	1	0	0	0	0
4	0	0	0	0	1	0

- *Equality Encoding:*
  - The i^{th} bit of the bit vector B_x is set if the attribute takes the value x in the i^{th} data record
  - Optimal for equality checks:
    Result of *"attr = x"* given directly by B_x
  - Range queries: E.g. *"attr<2"* -> *"B_0 v B_1"*,
    worst case: Half of the index has to be scanned
  - Sparse bit vectors ⇒ good compression efficiency

Basic Bitmap Indices

- *Range Encoding*
  - A bit is set if the attribute value is equal or less than the constant x associated with the bit vector B_x.
  - Optimal for range queries:
    Result of *"attr≤x"* is given directly by B_x.
  - Equality check: *"attr = x"* → *"B_x XOR B_{x-1}"*
  - Only the bit vectors at the edges of the bit matrix can be efficiently compressed.

Attribute Value	B_0	B_1	B_2	B_3	B_4	B_5
3	0	0	0	1	1	1
2	0	0	1	1	1	1
4	0	0	0	0	1	1
0	1	1	1	1	1	1
5	0	0	0	0	0	1
1	0	1	1	1	1	1
4	0	0	0	0	1	1

- *Pros and Cons of Bitmap indices:*
  - \+ Disk I/O reduction
  - \+ CPU efficiency: Multi-dimensional queries are evaluated by fast boolean combinations of bit vectors.
  - – Limited query complexity:
    E.g. "**`attr1 < const`**" but not "**`attr1 < attr2`**"
  - – Large index size for attributes with high cardinality ...

Coping high Cardinalities

- Basic bitmap indices explode in size in case of floating point attributes:

  Cardinality C ~ Number of data records N

  $\Rightarrow$ index size $S = C * N = f(N^2)$
- Solutions:
  - Bitmap Compression
    - Only efficient on equality encoded indices
    - Shoshani, Stockinger, Wu: WAH-algorithm
      - Boolean operations w/o prior decompression
      - S=f(N), index size = 2...6 $*$ data size
  - Reduction of the number of bit vectors
    - Binning
    - Bitmap encoding -> multi component indices

Coping high Cardinalities

- **Binning**
  - Partitioning of the attribute values into bins:
    → Regions of interest, adaptive binning
  - Creation of a bitmap index that addresses the bins
  - Index does not provide an exact query result,
    primary data has to be partially scanned (costly!).
  - Efficiency heavily depends on:
    1) Disk page size of the primary data
    2) Binning granularity
    3) Query dimension
    4) Selectivity
    - For sparse and high dimensional queries,
      a broad binning is sufficient. (10-100 bins)
    - If either the number of attributes involved in the query is low or the selectivity is high, a very fine binning is required
      (up to 10000 bins: 10000 index bits per 32-bit attribute value?)

Coping high Cardinalities

- **Multi component bitmap indices**
  - Bin numbers are decomposed to digits according to some base
  - For each digit a separate basic bitmap index is created
  - Significantly reduced index size:
    - e.g. a 3-component base<10,10,10> range encoded index addressing 1000 bins has a size of 9+9+9=27 bits per attribute value
  - Applicable on equality and range encoded indices
  - Query evaluation more complex:
    - maximum number of bit vectors involved: $2n_{comp}-1$
      e.g. base<10,10,10> → scan of 5 bit vectors
  - Choice of basis → decision on speed vs size

Prototype

- Multi component bitmap indices + binning
  - Range encoding
- Based on Root
  - Indexing of data stored in TTrees
  - Indices are stored in separate TTrees
- Features:
  - Basic and multi component bitmap indices with and w/o binning
    - Indices can be created for almost any expression accepted by Root's TTreeFormula query mechanism, e.g. **`sqrt(tracks[].px**2+tracks[].py**2)`**
    - Adaptive binning algorithm:
      - Each bitmap addresses similar number of records
      - Switches automatically to direct indexing w/o binning for low-cardinality attributes
    - Index creation in user definable intervals
    - Index compression by Root's zip algorithm

Prototype

- Features cont.
  - Query engine accepts TTreeFormula-like queries
    - Complex queries can be composed using all C++ comparative and logical operators
    - Indexed expressions should be compared to constant values
  - Automatic query evaluation optimizer
    - Sub-queries with low acceptances are evaluated first
      - Acceptance estimation based on information gathered at index creation
      - If primary data needs to be scanned, disk seeks are minimized by a prefetching algorithm.
    - Row-wise and column-wise evaluation of multi-dimensional queries (depends on the persistent layout of the TTree)

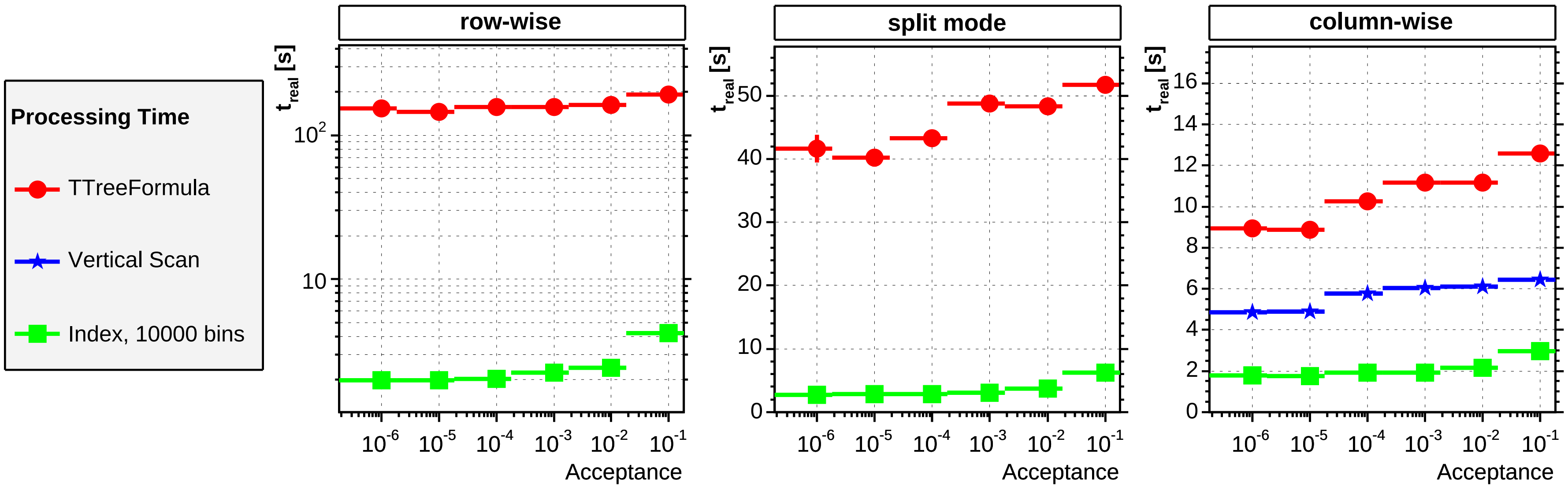
Performance Tests

Data scan efficiency vs. persistent data layout

- *Horizontal partitioning:*
  - Relational databases, streamed objects, unsplit TTrees
  - All data attributes are written to common disk pages (or Root-*TBaskets*)
  - Queries require a full database scan, even if they involve only a subset of the stored attributes
- *Vertical Partitioning*
  - Column-wise writing of attribute data to consecutive disk pages
  - Optimal for simple multi-dimensional queries: **`A1<x && A2>y && ...`**
    - Evaluation by a column-wise scan of attribute data
    - Inefficient, if query involves complex expressions including more than on attribute: e.g. **`sqrt(px**2+py**2)`**
  - Column-wise writing of a large database is in most cases not feasible, since data is produced record by record not attribute by attribute.
- *Split Partitioning*
  - Root TTree (split mode)
  - Row-wise filling, but data of each attribute is written to separate TBaskets.
  - Exclusive access to attribute data, but the according TBaskets are not organized consecutively on disk. → Affects scan efficiency.

Performance Tests

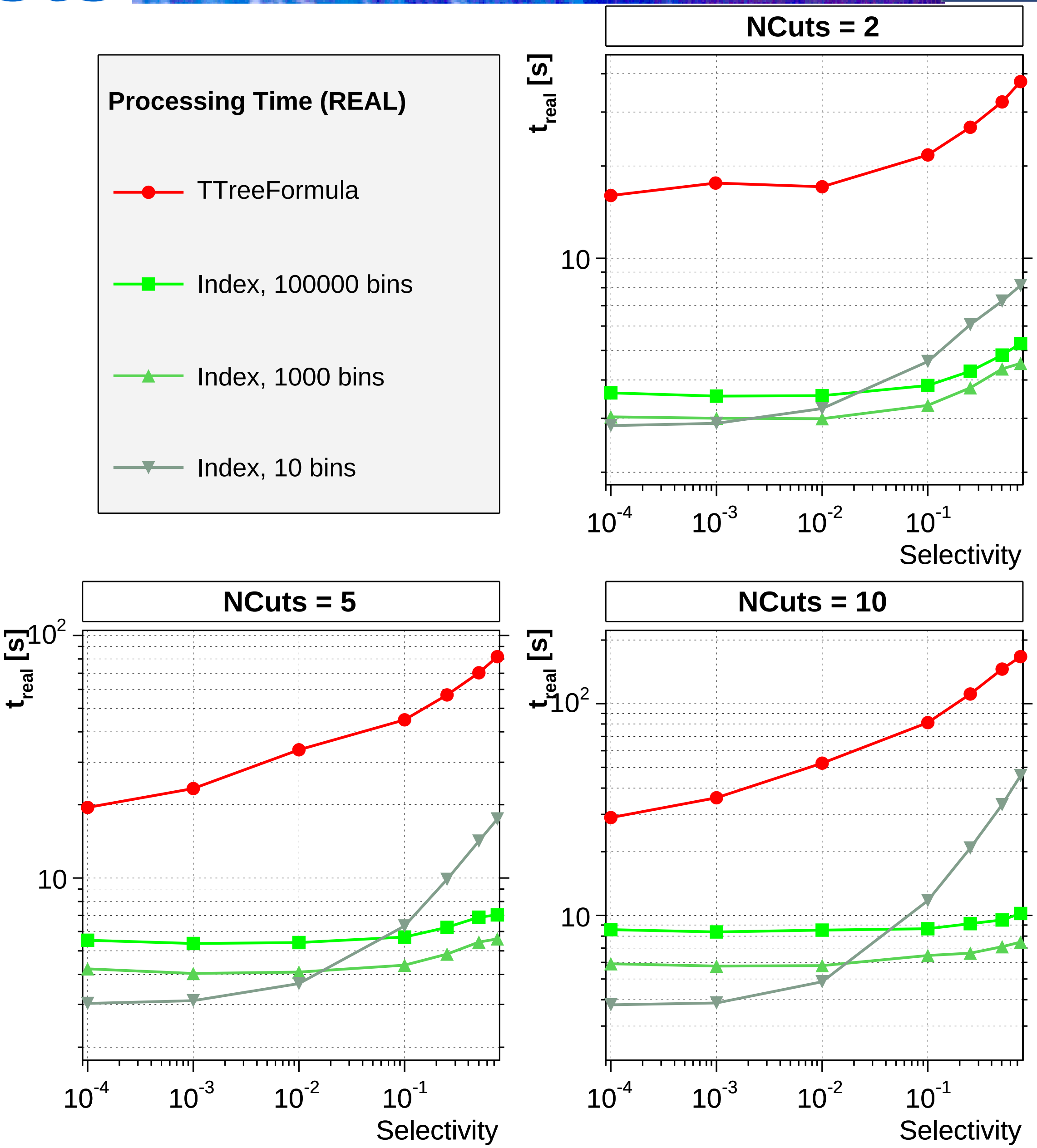
- Event TAG data extracted to TTrees of different persistent layouts:
  7.6 million entries with 40 integers and 63 floats
  1.5 GB (3 GB uncompressed), TBasket-size 16 KB
- P4 2.4 GHz, 768 MB RAM, 40 GB IDE disk
- Index: 4 components, 10000 bins, 2.0 GB (compressed),
  creation time: 35 min.
- Queries: Conjunctions of range queries on 5 attributes
- Performance gain:
  - row-wise TTree: 50 - 80
  - split TTree: 8 - 16
  - column-wise TTree: 4 - 5 (2 - 3 compared to vertical scan)



Performance Tests

Repetitive queries on a small database resident in memory

- Simulation of a selection optimization scenario
- TTree: 500000 entries with 10 randomly distributed float attributes (uncompressed)
- Indices:
  - basic, 10 bins
  - 3 components, 1000 bins
  - 5 components, 100000 bins
- Test queries:
  - Conjunctions of range queries on 2, 5, and 10 attributes
  - Applied repetitively 100 times with randomly varied query boundaries
  - Selectivity: 10^{-4} - 0.75
- Performance gain: 5 - 20, especially for selectivities greater than 10%

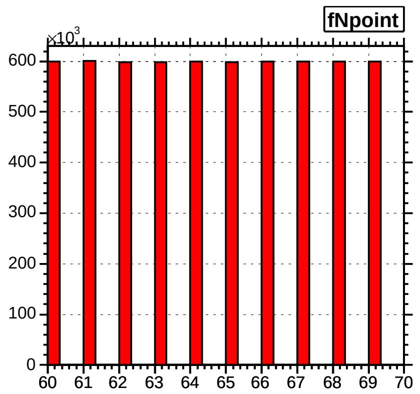


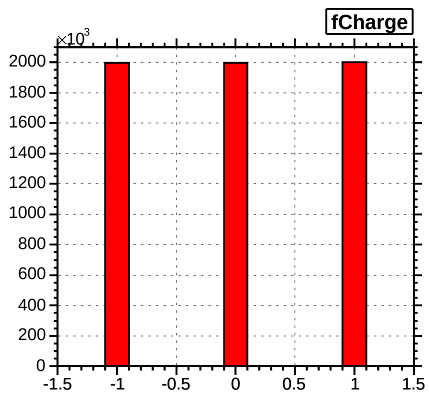
Performance Tests

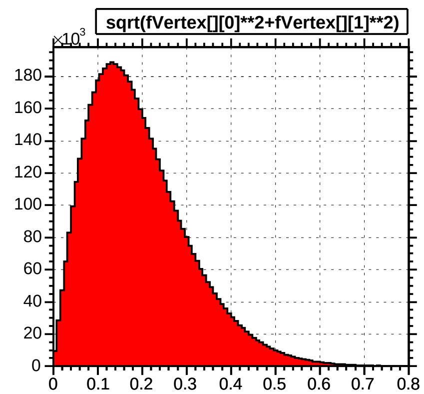
- Interactive selections on Root's toy Event demo
  - 30000 events with 18 million tracks (TObjArray)
  - 1.1 GB, compressed, split
- 3 indexed track members:
  - **`"fCharge"`**: discrete
  - **`"fNpoint"`**: discrete
  - **`"sqrt(fPx**2+fPy**2+fPz**2)"`**: adaptive, 100000 bins
  - Creation time: 312 s / Size: 109 MB (uncompressed)
- Selections:

```
"fCarge==X && fNpoint>=Y && sqrt(fPx**2+fPy**2+fPz**2)>Z"
```

mean query time [s]	TTreeFormula	index	gain
pure selection	139	7.5	18
selection + histogram filling	140	14.1	10







Performance Tests

- Selections on HEP analysis data
  - Taken from a currently performed analysis
  - TChain:
    - 360 TTrees in separate Files (17 GB)
    - 23 million entries with 430 attributes (split, TBasket size 8K, compressed)
  - Selections involve 11 attributes
    - 3 mass windows
    - cuts on 3 vertex probabilities, momenta, lifetime and 2 selector bits
  - Indices:
    - adaptive binning, 4 components, 10000-bins
    - cover only the region of interest

Performance Tests

- Selections applied on the whole TChain:
  - average acceptance $1.2 * 10^{-4}$
  - TTreeFormula: 558 s
    (Entries outside the region of interest are masked out by TEventList)
  - Index: 14.5 s *(gain: 38)*
- Selections applied on skimmed subsets
  TChain merged to a single TTree
  - 9 million entries with 40 attributes, Basket size 32 KB, compressed, 1.1 GB
    - TTreeFormula: 170 s
    - Index: 8.0 s *(gain: 21)*
  - 12 attributes, 61000 entries, Basket size 32 KB, uncompressed, 3.5 MB
    - 200 repetitive queries: (average acceptance: 6 %)
      - TTreeFormula: 29.1 s
      - Index: 4.1 s *(gain: 7)*

Summary

- Binned multi component bitmap indices can significantly improve the performance of multidimensional ad hoc queries
  - efficient in a wide range of selectivities
  - efficient on both, large data samples on disk and small memory resident samples
  - reasonable index size: $< 1.5 * $ data size
- Outlook
  - Collaboration with John Wu and Kurt Stockinger (LBL)
    - Experts on bitmap compression
    - Workshop at CERN in December 2004
      - Participation of Root and POOL team
      - Integration of bitmap indices to Root / Pool
      - Use of indices in a parallel environment ?