Optimizing multidimensional Queries using Bitmap Indices

- 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, 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 don't cope high cardinality attributes, as size grows linearly with the cardinality.
    - read only data (no updates,inserts,...)

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
- Multidimensional queries are evaluated by fast boolean combinations of bit vectors

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: *"attr<2"* *->* *"B_0 v B_1"*,
    in the worst case half of the index has to be scanned
  - The sparse bit vectors can be efficiently compressed

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

Coping high cardinalities

- Basic bitmap indices explode in size for floating point attributes:

  Cardinality C $\sim$ Number of data records N

  $\Rightarrow$ index size $S = f(N^2)$

- Possible solutions:
  - Reduction of the number of bit vectors
    - Binning
    - Bitmap encoding -> multi component indices
  - Bitmap Compression

  or combinations

Coping high cardinalities

- **Binning**

  1) Partitioning of the attribute values into bins (adaptively)

  2) Creation of a bitmap index based on bin numbers

  – Index does not provide an exact query result, original data has to be partially scanned (expensive)

  – Efficiency heavily depends on:

  1) binning granularity

  2) query dimension

  3) selectivity

  - For sparse and high dimensional queries, a broad binning is sufficient.
  - If either the number of attributes involved in the query is low or the selectivity is high, a very fine binning is necessary.

Coping high cardinalities

- **Binning example:**
  range encoded index (5 bins)
  - Query "A<0.3"
  - Candidates: $B_{\leq 0.4}$
  - Hits: $B_{\leq 0.2}$
  - To be checked: $B_{\leq 0.2}$ XOR $B_{\leq 0.4}$

Attribute Value	$B_{\leq 0.2}$	$B_{\leq 0.4}$	$B_{\leq 0.6}$	$B_{\leq 0.8}$	$B_{\leq 1.0}$
0.73	0	0	0	1	1
0.55	0	0	1	1	1
0.24	0	1	1	1	1
0.12	1	1	1	1	1
1.13	0	0	0	0	0
0.33	0	1	1	1	1
0.05	1	1	1	1	1

  - Amount of scanned data records: $P_{scan} = P_{cand} - P_{hit} = 0.4 - 0.2 = 20\%$
    (finite disk page size ⇒ complete scan)
  - Multidimensional queries: $A_1 < x_1 \wedge A_2 < x_2 \wedge \ldots \wedge A_n < x_n$

    Global cand and hit vector: $B_{HIT} = \wedge^n B_{hit\ i}$, $B_{CAND} = \wedge^n B_{cand\ i}$

    $B_{SCAN} = B_{HIT}\ XOR\ B_{CAND} \Rightarrow P_{SCAN} = \prod^n P_{cand\ i} - \prod^n P_{hit\ i}$

    Example: 5-dim query, $A_1 < 0.3 \wedge \ldots \wedge A_5 < 0.3$ $P_{SCAN} = 0.4^5 - 0.2^5 = 0.1\%$
- Necessary number of bins for floating point attributes: 100 - 100000,
  up to 100000 index bits per 32-bit float attribute value?

Coping high cardinalities

- **Multi component bitmap indices**
  - Bin numbers (or integer attribute values) are decomposed to digits according to some base
  - For each digit a 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+2=29 bits per attribute value (2 bits for under- and overflow)
  - Query evaluation more complex:
    - maximum number of bit vectors involved: $2n_{comp}-1$

      e.g. base<10,10,10> → 5 bit vectors
  - Choice of basis → decision on speed vs size

Coping high cardinalities

- **Bitmap Compression**
  - Although the information content of the index matrix is quite small, compression is difficult, since the bit vectors have to be stored separately.
  - Only equality encoded bitmap indices can be compressed efficiently (low bit density)
  - Efficient algorithms exists that allow boolean operations directly on compressed bit vectors
    - Shoshani, Stockinger, Wu
      - basic equality encoded index, compressed, no binning
      - Index size scales linearly with the number of data records. worst case: index size = 4 $*$ data size
      - Query processing time scales linearly with acceptance
      - Test: 12 attributes, 2.2 million entries, average cardinality per attribute 220000, size $\sim$100 MB
        - index: 2.7 million bit vectors, compressed size 186 MB
        - outperforms vertical data scan by factor of 2-50 (selectivities: 10^{-6}- 0.1, dimensions: 2, 5)

Prototype

- Multi component bitmap indices + binning
- Based on Root
  - Indices are stored in TTrees
- Supports basic and multi component indices with arbitrary base definitions with and w/o binning
  - So far only range encoding
  - Binning modes:
    - Equidistant
    - Discrete
    - Adaptive (with spike search, automatic change to discrete mode)
  - index creation in user definable intervals
    (proper under- and overflow handling)
- Compression:
  - No special compression method
  - Root's gzip algorithm can be used

Prototype

- Indices can be created for almost any expression accepted by TTreeFormula:
  - e.g. **`sqrt(px**2+py**2)`**
  - limited support for complex TTrees (var size arrays, ...)
    e.g. **`sqrt(tracks[].px**2+tracks[].py**2)`**
    but no fancy matrix multiplications: **`vector[]*matrix[][]`**
- Built-in Parser for TTreeFormula-like queries
  - query format: **`EXPR OPERATOR CONST`**
    - **`EXPR`**: indexed expression or index name
    - **`OPERATOR`**: any C++ comparative operator
    - **`CONST`**: some constant
    - e.g. **`sqrt(px**2+py**2)<=0.5`** but not **`px<py`** (**`-> px-py<0`**)
  - any logical combination of subqueries accepted: **`&&,||,!,()`**
  - subqueries on expressions w/o an index
  - supports row-wise and column-wise evaluation of multi dim. queries
- Automatic query evaluation optimizer
  - sub-queries with low acceptances are evaluated first (IO, CPU-time), persistent data is scanned consecutively (disk seek time)

Performance Tests

- **Persistent layout of TTrees**
  - *SPLIT- mode*
    - Attributes values are written to separate TBranches ("persistent columns")
    - Row-wise filling

	split mode			column wise			row-wise
Attributes	1	2	3	1	2	3	1-3
persistent TBasket-Position	1	2	3	1	4	7	1-9
	4	5	6	2	5	8	
	7	8	9	3	6	9	

    - Fragmented, TBaskets ("disk pages") of a particular attribute are not written to contiguous disk areas. Affects data scan efficiency.
  - *Vertically partitioned (column-wise)*
    - Optimal for simple queries: Column wise scan of contiguously written attribute data, e.g. $\mathbf{A_1 < x \ \&\& \ A_2 > y \ \&\& \ ...}$
    - Inefficient for complex queries involving more than one attribute, e.g. **`sqrt(px**2+py**2)`**
    - In most cases it is not feasible to write data in a column-wise manner.
      - Transform already written TTrees
      - Link TBranches to separate files
  - *Horizontally partitioned (row-wise)*
    - streamed objects, relational databases
    - very inefficient for queries involving only a subset of the stored attributes

Performance Tests

- Systematic tests
  - Data: Ntuple (1.5 GB)
    - 4 million entries
    - 100 attributes, 32-bit floats, randomly distributed (flat, [0,1]), no compression, TBasket size 16KB
    - different persistent layouts
  - Indices:
    - 10 attributes are indexed
      - basic, 10 bins, 11 bits per attr. value
      - 3-component <10,10,10>, 1000 bins, 29 bits per attr. value
      - 5-component <10,10,10,10,10>, 100000 bins, 47 bits per attr. value
  - Queries:
    - A_0`<=x && `A_{10}`<=x && ... && `A_{90}`<=x`
    - involves 2, 5, and 10 attributes
    - selectivities: 10^{-7} - 0.5 (variation of query boundary x)
    - "file cache reset" between queries

Performance Tests

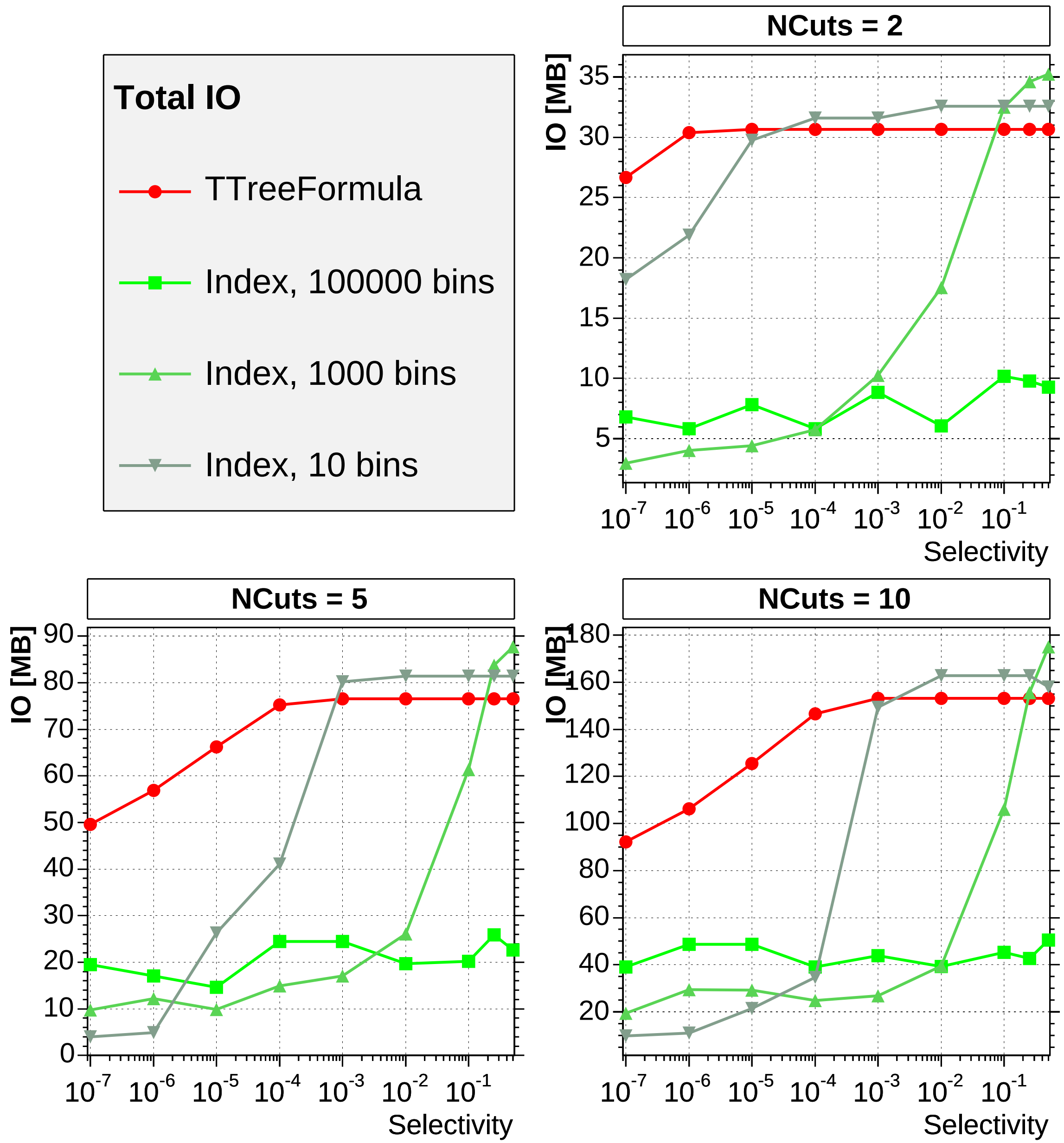
- performed on a ordinary PC
  - 1.4 GHz P4, 256 MB, 40 GB IDE disk
  - Root 4.00.06
- Index creation:

	Creation time per attribute [s]		
persistent tree layout	split	column-wise	row-wise
10-bin index	8.2	3.7	47.1
1000-bin index	13	7.9	50.7
100000-bin index	25	15	54.4

Performance Tests

Split mode: IO

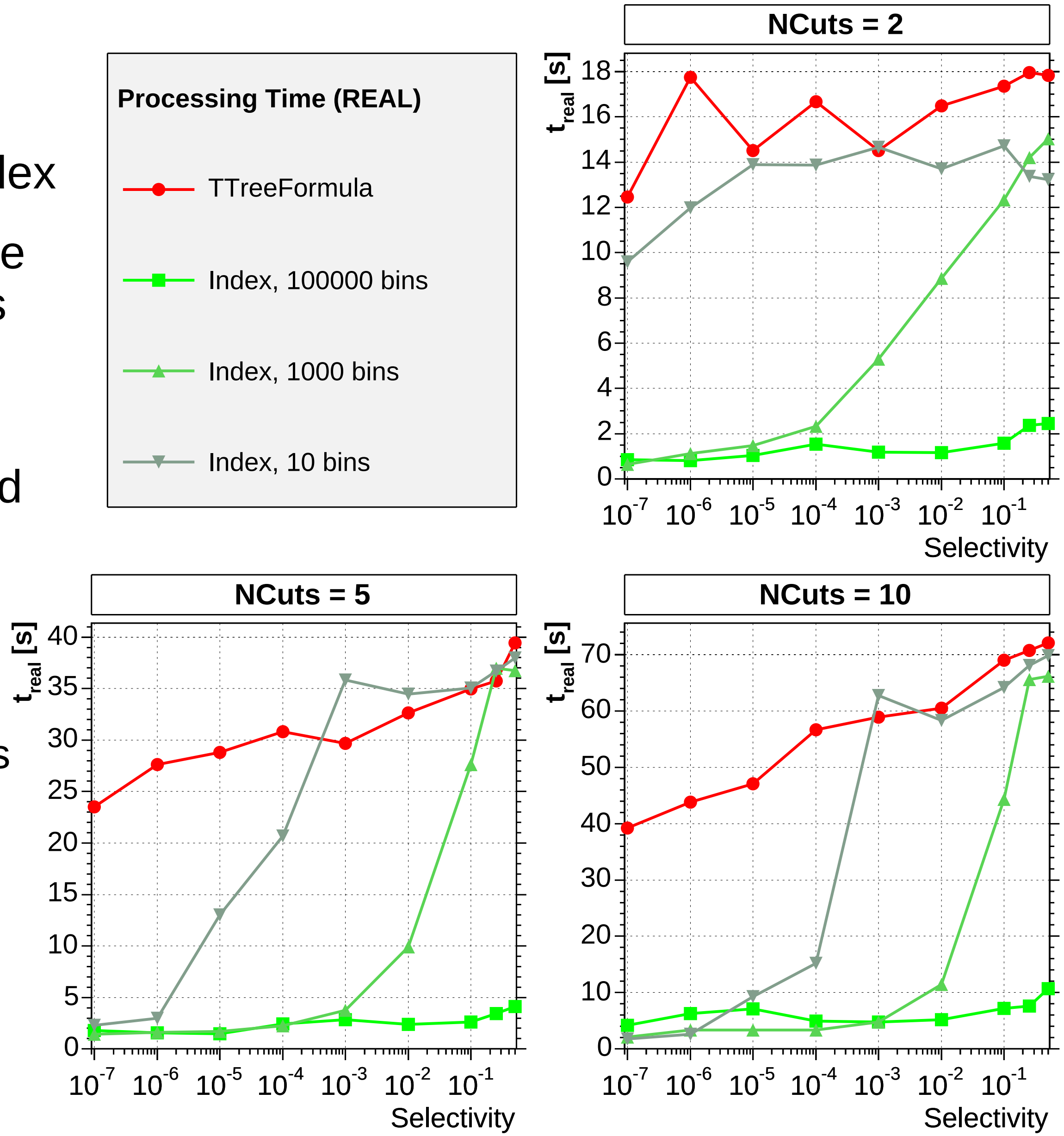
- Constant amount of index data, plateau at low selectivities
- Rise at high selectivities due to increasing number of candidates that have to be validated by scanning the original data.



Performance Tests

Split mode: Real time

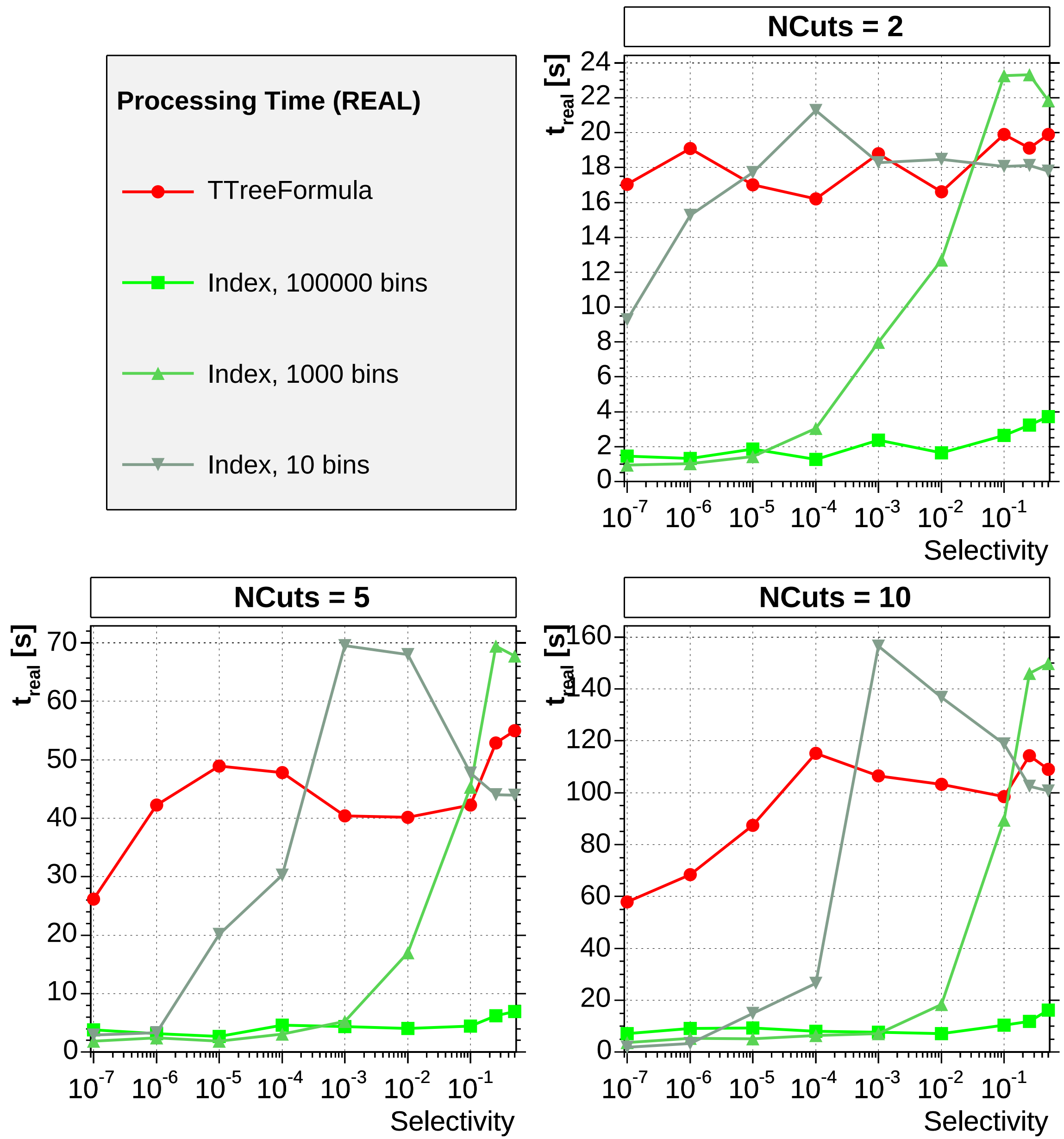
- Performance gain by a factor 6...15 with the 100000 bin index
- 10 bin index: Only very sparse and high-dimensional queries can be efficiently performed
- Indices with broad binning superior in case of sparse and high dimensional queries
  - Deactivation of index components would yield the same performance with the finely binned indices (range encoding only)



Performance Tests

Split mode

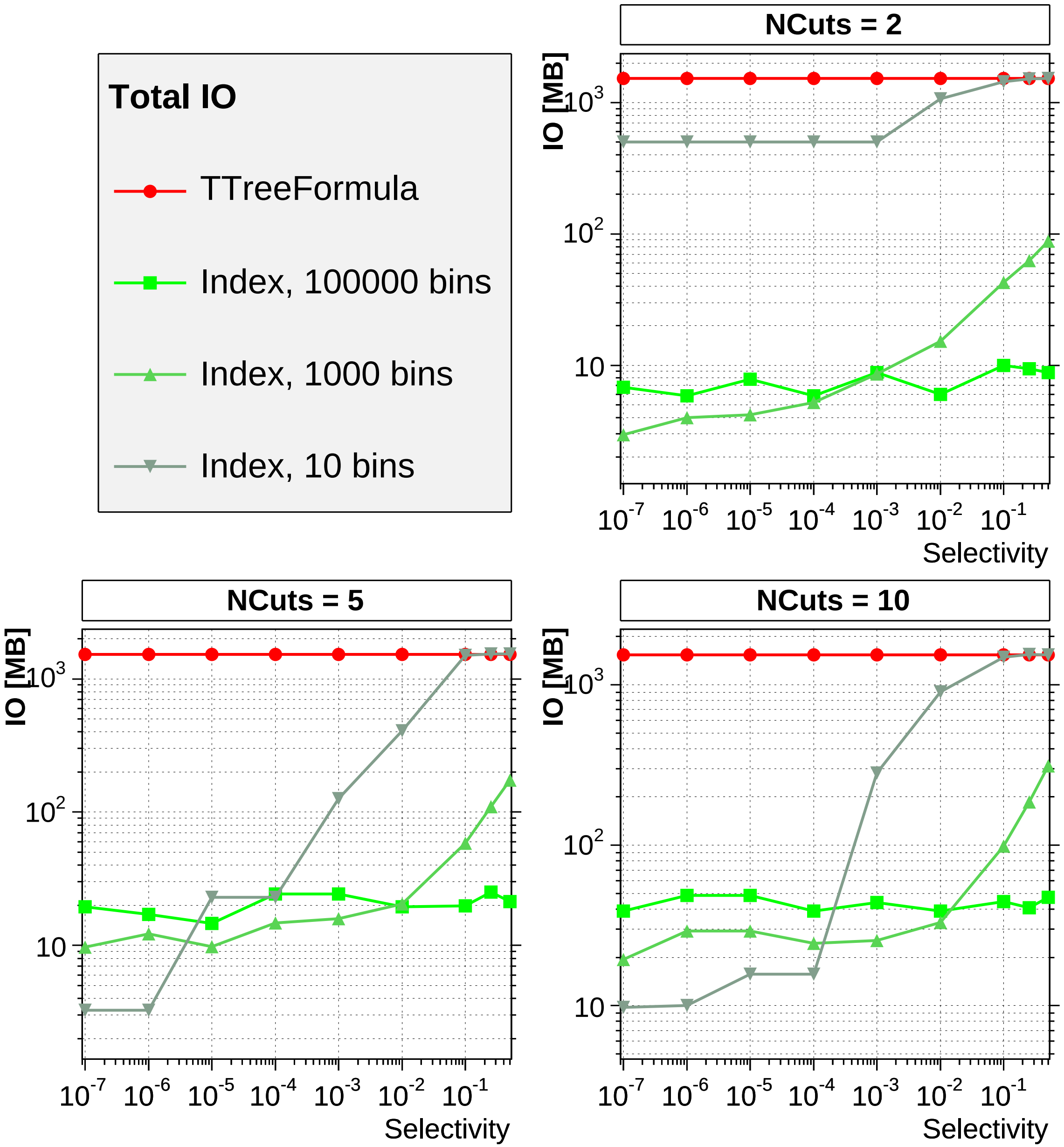
- remote access via rootd



Performance Tests

Row-wise TTree: IO

- Rough estimate of performance gain with relational databases

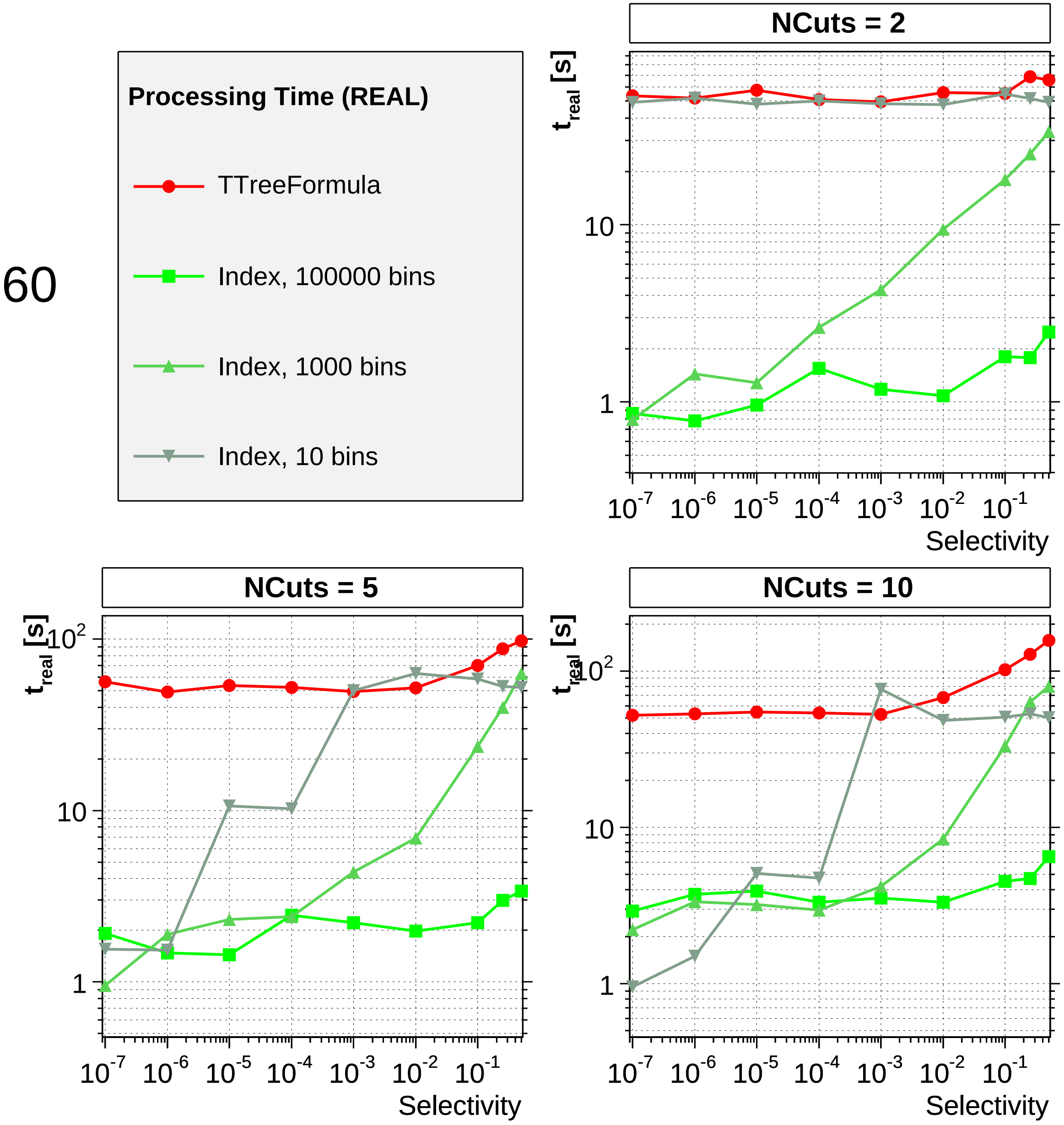


Performance Tests

Row-wise TTree

Real time

- performance gain: 15...60

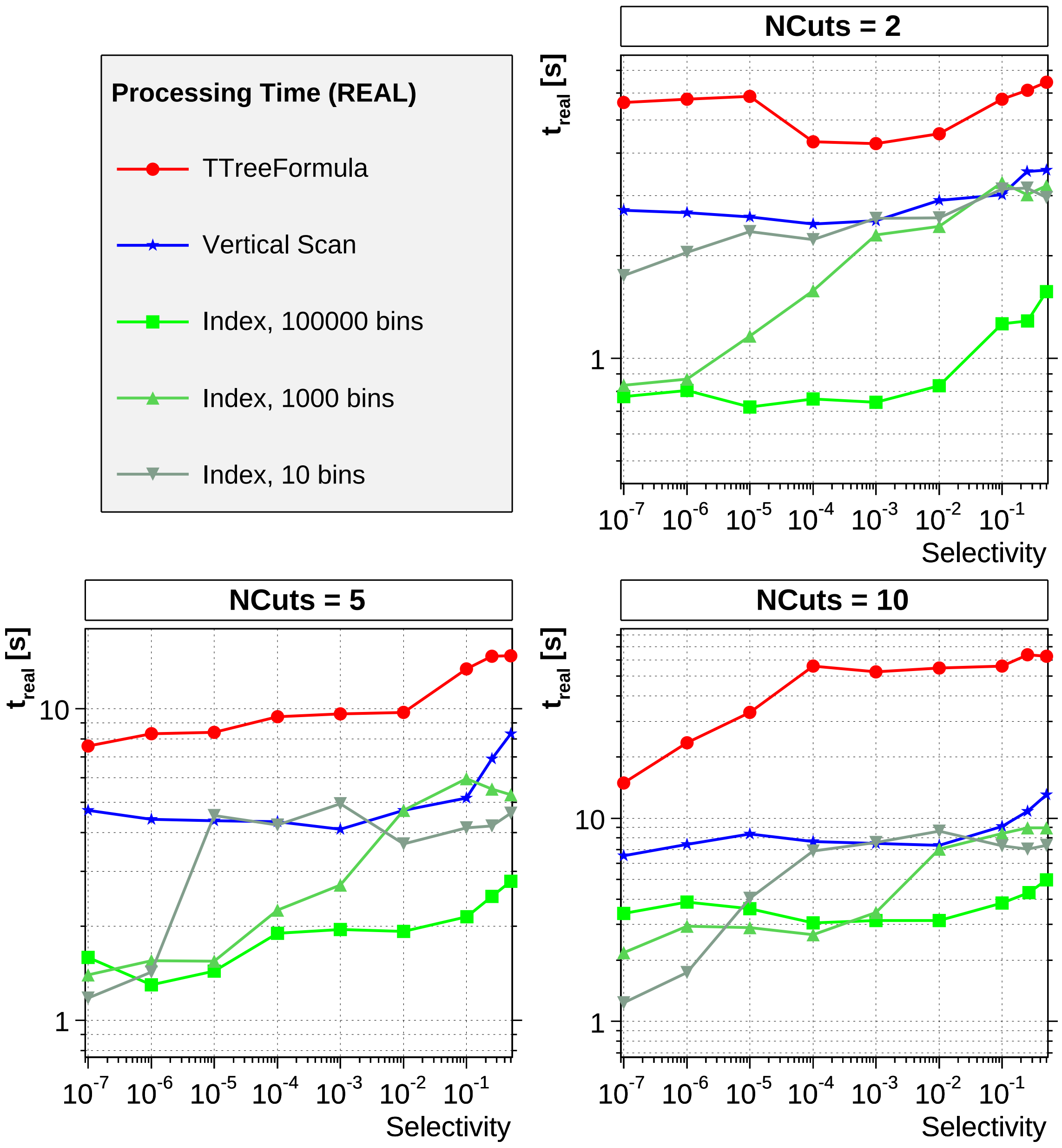


Performance Tests

Column-wise TTree:

Real time

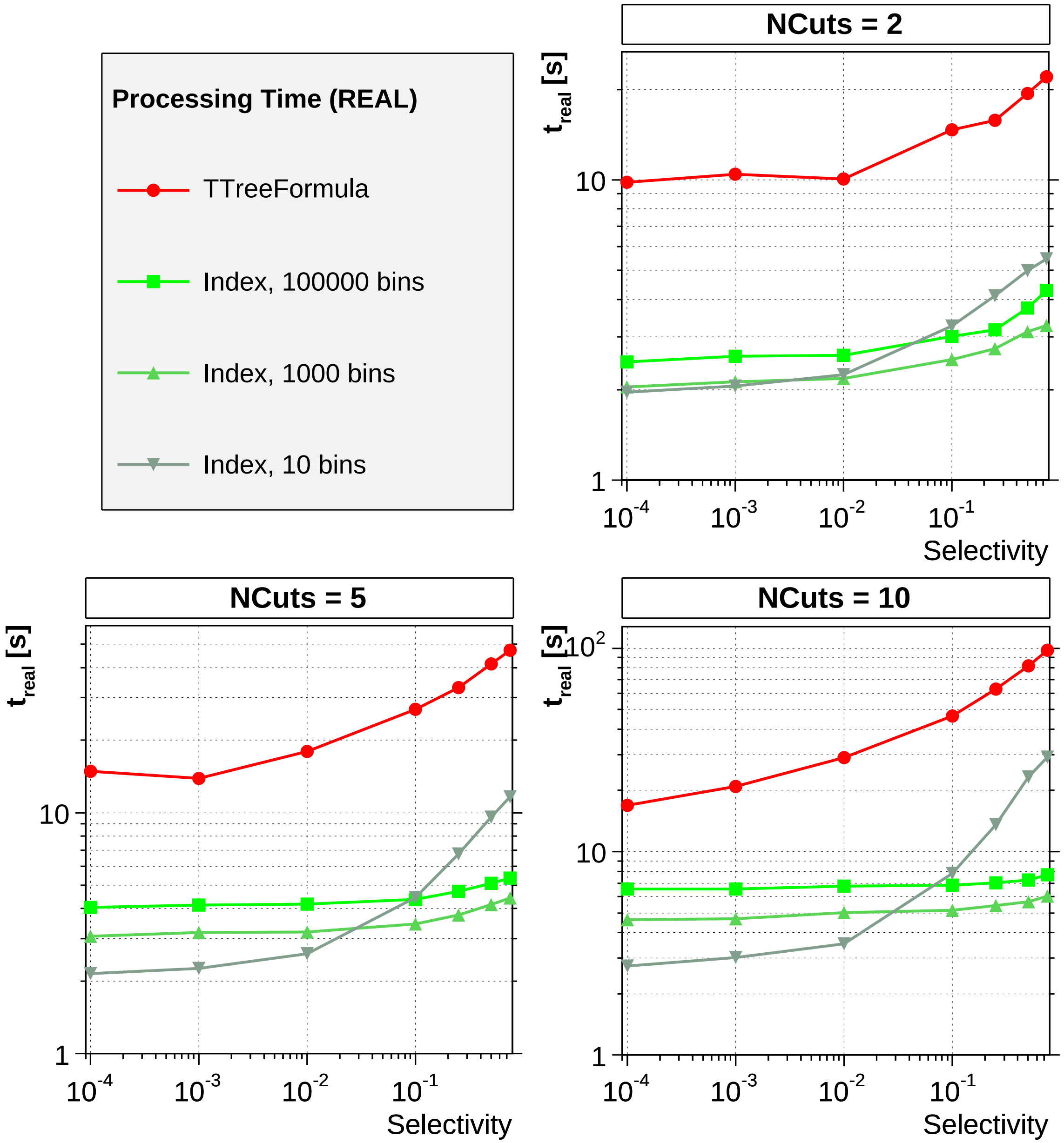
- "unfair" comparison to TTreeFormula, which evaluates the query in a row-wise manner
  - However, if only a few attributes are involved in the query or the selectivity is low, also TTreeFormula benefits from column-wise layout.
- performance gain achieved with the 10000 bin index:
  - 4...15 compared to TTreeFormula
  - 2...4 compared to vertical scan



Performance Tests

Repetitive queries on a small TTree resident in memory

- Optimization scenario (e.g. genetic algorithm)
- 500000 entries
- 50 repetitive queries with randomly varied query boundaries:
- selectivities: 10^{-4} - 0.75
- performance gain: 5 - 16

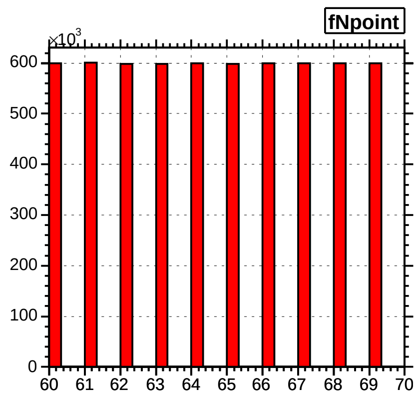


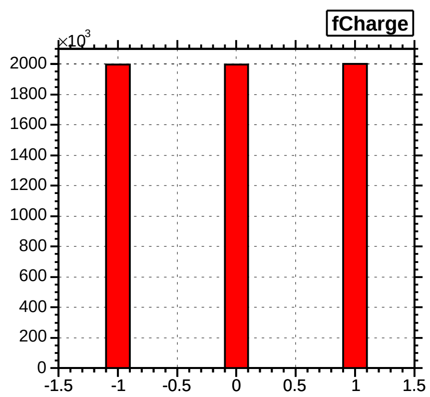
Performance Tests

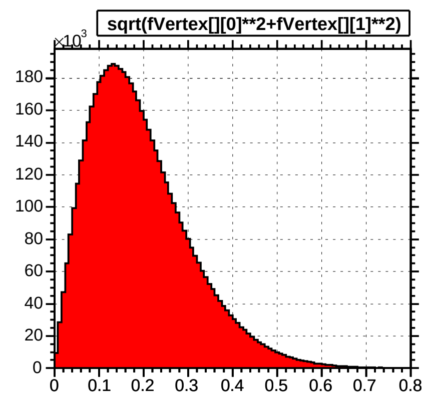
- Interactive selections on Root's toy Event demo
  - 30000 events with 18 million tracks (TObjArray)
  - 1.1 GB, compressed, split
- 3 indexed track members:
  - **`"fCharge":`** discrete
  - **`"fNpoint":`** discrete
  - **`"sqrt(fPx**2+fPy**2+fPz**2)":`** adaptive, 100000 bins
  - Creation time: 312 s / Size: 109 MB
- 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
sel. + histogram fill	140	14.1	10







Performance Tests

- Real Data
  - Taken from a currently performed analysis
  - TChain:
    - 360 TTrees in separate Files (17 GB)
    - 430 attributes (split, TBasket size 8K, compressed)
    - 23 million entries (lots of background)
  - Selections involve 11 attributes
    - 3 mass windows
    - cuts on 3 vertex probabilities, momenta, lifetime and 2 selector bits
  - Indices
    - adaptive binning, 10000-bins
    - cover only the region of interest
  - TTreeFormula
    - entries outside the region of interest are masked out (TEventList)

Performance Tests

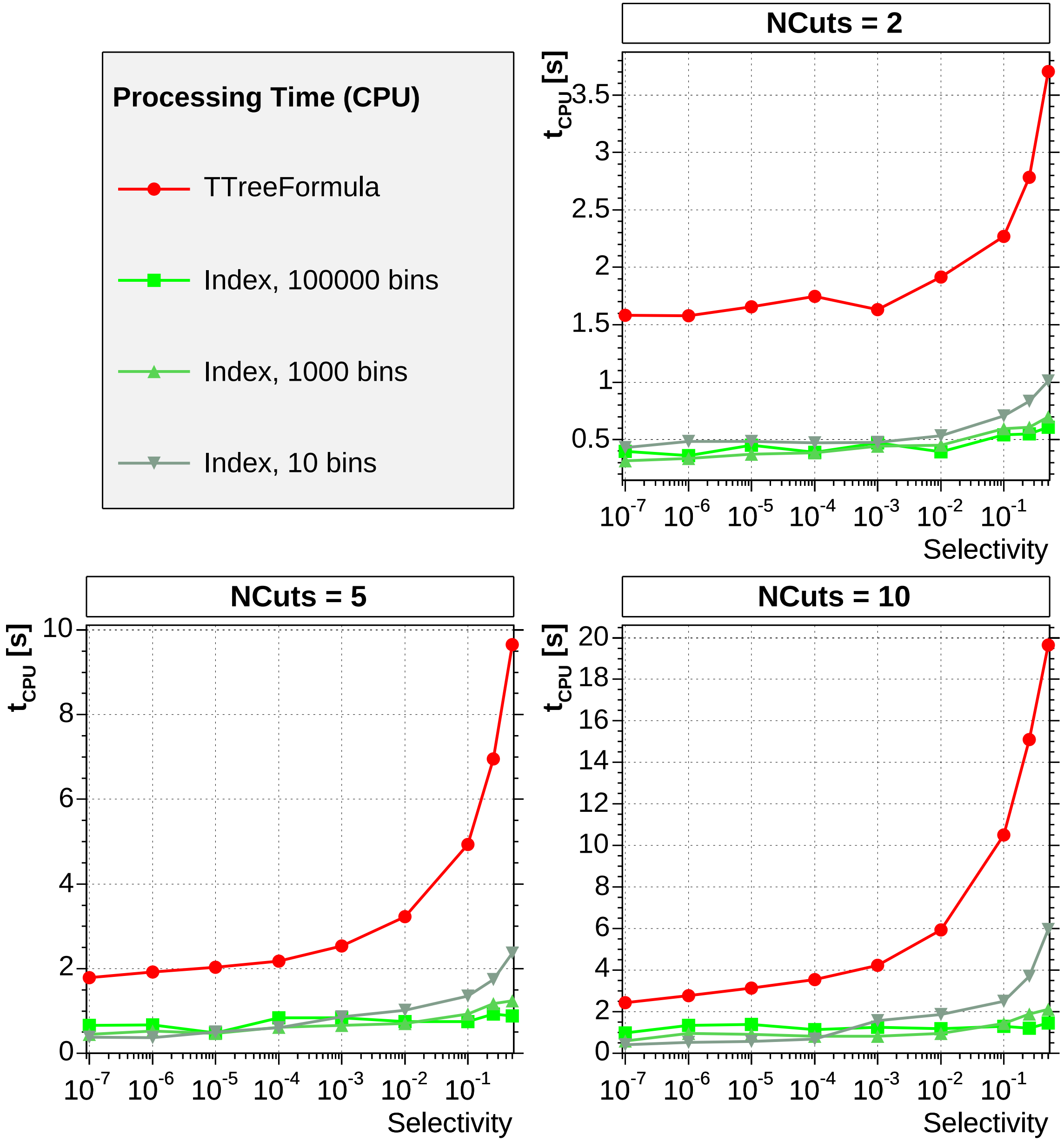
- Selections applied on the whole TChain:
  - average acceptance $1.2 * 10^{-4}$
  - TTreeFormula: 558 s
  - Index: 14.5 s (gain: 38)
- Selections applied on preselected subsets
  TChain merged to a single TTree
  - 40 attributes, 9 million entries, Basket size 32 KB, compressed, 1.1 GB
    - average acceptance: $3 * 10^{-4}$
    - 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
    - Experts on compressed bitmap indices, but also investigating binning methods
    - Comparison of the two approaches
  - Integration of bitmap indices to Root
    - Will come along with a new abstract index interface (TEventList, B-tree-like TTree index, bitmaps)
  - Pool event collections

Performance Tests

Split mode: CPU time



Performance Tests

Row-wise TTree

Real time

- 4 million entries
- only 10 attributes

- rough estimate of performance gain for selections on streamed objects
- Even selections involving all attributes are evaluated faster by the index (CPU efficiency)