面向海量数据的轻量级索引

Lightweight Indexes for Massive Datasets

Indexing Massive Datasets…

- Data characteristics
  - Read-mostly, append-only, few updates
    - Data warehousing, time series

Indexing Massive Datasets…

- Data characteristics
  - Read-mostly, append-only, few updates
    - Data warehousing, time series
  - High-dimensional (many attributes)

Indexing Massive Datasets…

- Data characteristics
  - Read-mostly, append-only, few updates
    - Data warehousing, time series
  - High-dimensional (many attributes)
  - Mix of low-cardinality and high-cardinality attributes
    - $nation$ vs $item\_id$, $zone$ vs $source\_ip$

Indexing Massive Datasets…

- Data characteristics
  - Read-mostly, append-only, few updates
    - Data warehousing, time series
  - High-dimensional (many attributes)
  - Mix of low-cardinality and high-cardinality attributes
    - $nation$ vs $item\_id$, $zone$ vs $source\_ip$

- Why indexing? → To accelerate queries

Indexing Massive Datasets…

- Data characteristics
  - Read-mostly, append-only, few updates
    - Data warehousing, time series
  - High-dimensional (many attributes)
  - Mix of low-cardinality and high-cardinality attributes
    - $nation$ vs $item\_id$, $zone$ vs $source\_ip$

- Why indexing? → To accelerate queries
- What queries? → OLAP queries
  - Ad-hoc, arbitrarily multi-dimensional, relatively low selectivity

Traditional DB Indexes

- B-Tree Indexes
  - Not suited for arbitrary high-dimensional queries
    - Multi-column B-tree index: only queries on key prefixes can benefit
  - Large storage footprints
    - 5%-15% space overhead per index
  - High maintenance overhead

- Multi-Dimensional Indexes (R-Tree, KD-Tree, Range Tree)
  - Not scalable for large number of dimensions

Bitmap Indexes

- Also a traditional index scheme (1980s)
- Primarily intended for data warehousing applications
  - Not suitable for OLTP: can't be updated efficiently
- Recommended/used by commercial DBMS vendors
  - Oracle Online Documentation:
    - *"B-tree indexes should be used only for unique columns or other columns with very high cardinalities"*
    - *"**The majority of indexes in a data warehouse should be bitmap indexes**"*
    - *"In general, bitmap indexes should be more common than B-tree indexes in most data warehouse environments"*

Bitmap Indexes & Other Lightweight Indexes

- Scalable for large amounts of data
- Small storage footprints
- Can be efficiently maintained*
  - scenarios: append only, few updates

In This Talk, We Will…

- Give an overview of related research work
- Show the connections between different index designs
- Explore the design space of lightweight indexes

The Scope of This Talk

- Queries
  - Point Queries
  - Range Queries

```
SELECT *
FROM   orders
WHERE  item_id = 10 AND
       2 < amount AND amount < 10
```

- Indexes
  - Non-Unique Secondary Indexes
    - No modification to base data

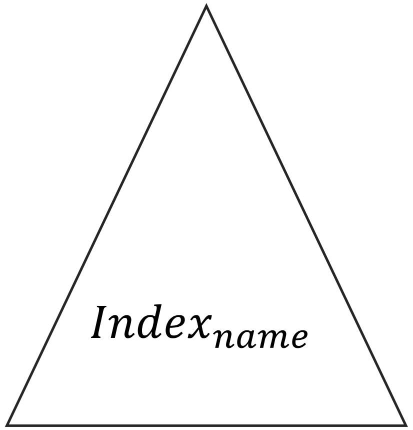
Outline

- Exact Indexes
  - Bitmap Index
- Approximate Indexes
  - Block-Level Bitmap Index
  - Zone Map
  - Bloom Filter
  - Range Filter

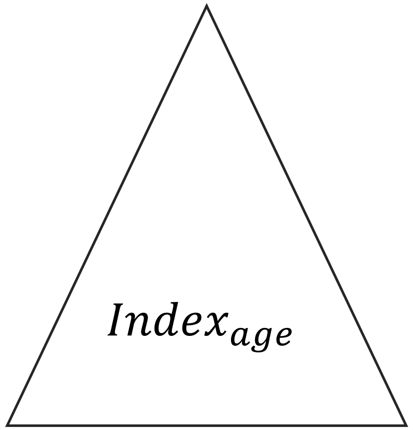
Exact Indexes

- $predicate \rightarrow RID\ list$
  - Return the identifiers of **all** records that satisfy a given predicate

RID	$Name$	Age
1	Alice	22
2	Bob	23
3	Daniel	25
4	Smith	18
5	Smith	21
6	Smith	30



$Name = \text{‘}Smith\text{’} \rightarrow [4, 5, 6]$



$20 < Age < 25 \rightarrow [1, 2, 5]$

From B-Tree to Bitmap

- None-unique B-tree index on $(name)$
  - Option 1: $key = (name,\ rid)$

Alice, 1	Bob, 2	Daniel, 3	Smith, 4	Smith, 5	Smith, 6

  - Option 2: $key = name,\ value = rid\ list$

Alice: 1	Bob: 2	Daniel: 3	Smith: [4, 5, 6]

Inverted Index

  - Option 3: $key = name,\ value = bit\ vector$

Alice: 100000	Bob: 010000	Daniel: 001000	Smith: 000111

Bitmap Index

RID	$Name$	Age
1	Alice	22
2	Bob	23
3	Daniel	25
4	Smith	18
5	Smith	21
6	Smith	30

Some Notations…

- Let the $\boldsymbol{cardinality}$ of an indexed attribute $= C$
  - The number of possible/distinct values（可能的取值数量）
    - e.g., $C_{married} = 2$, $C_{month} = 12$

- Let the number of records = $\boldsymbol{N}$

Bitmap Indexes

- Logically, each bitmap index is a **bit matrix**
  - We view the matrix as a collection of **columns**
    - number of bit vectors = $\boldsymbol{C}$, length of each bit vector = $\boldsymbol{N}$

T

RID	$Name$	Age
1	Alice	22
2	Bob	23
3	Daniel	25
4	Smith	18
5	Smith	21
6	Smith	30

	Alice	Bob	Daniel	Smith
1	1	0	0	0
2	0	1	0	0
3	0	0	1	0
4	0	0	0	1
5	0	0	0	1
6	0	0	0	1

$bit\ vector$ $T[2].name = 'Bob'$

$T[3].age = 25$

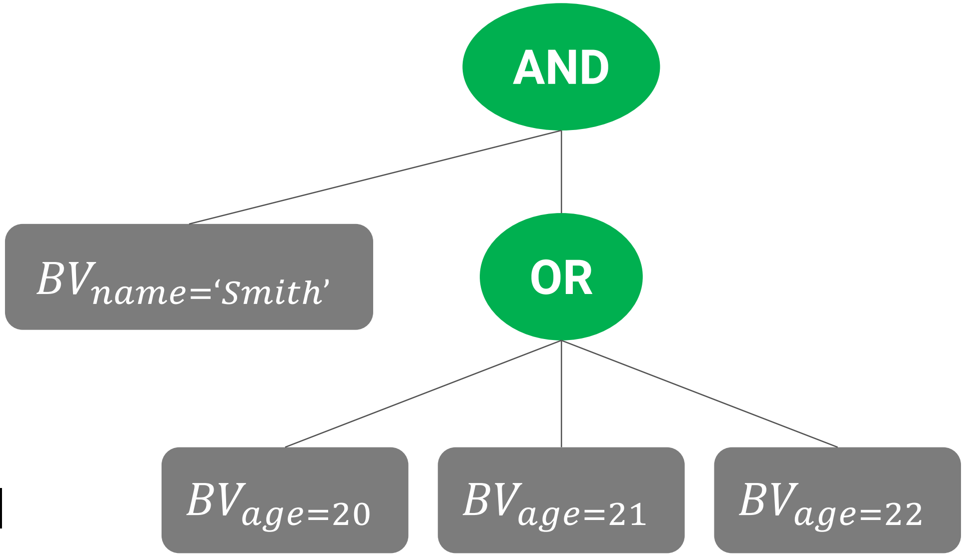
	18	21	22	23	25	30
1	0	0	1	0	0	0
2	0	0	0	1	0	0
3	0	0	0	0	1	0
4	1	0	0	0	0	0
5	0	1	0	0	0	0
6	0	0	0	0	0	1

Query Processing with Bitmap Indexes

- Query Rewrite
  - $predicates \rightarrow query\ evalution\ graph$
    - leaf node: ***bit vector***
    - internal node: ***bitwise operator***

```
WHERE name = 'Smith' AND
      age >= 20 AND
      age <= 22
```

- Query Evaluation
  - Space-saving strategy
    - Maintain only one intermediate result
  - Time-efficient strategy
    - Evaluate independent operators in parallel



So Far So Good…

- Consider a retail database
  - $orders(oid,\ date,\ customer\_id,\ item\_id,\ amount,\ price)$

So Far So Good…

- Consider a retail database
  - $orders(oid,\ date,\ customer\_id,\ item\_id,\ amount,\ price)$
  - 1 billion records
    - size of each uncompressed bit vector ≈ **125MB** !!!

$Problems$	#1: **compress the bit vectors**

So Far So Good…

- Consider a retail database
  - $orders(oid,\ date,\ customer\_id,\ item\_id,\ amount,\ price)$
  - 1 billion records
    - size of each uncompressed bit vector ≈ **125MB** !!!
  - 1 million items
    - indexing $item\_id$: number of bit vectors = $\mathbf{10^6}$!!!

$Problems$	#1: **compress the bit vectors**
	#2: **reduce the total number of bit vectors**

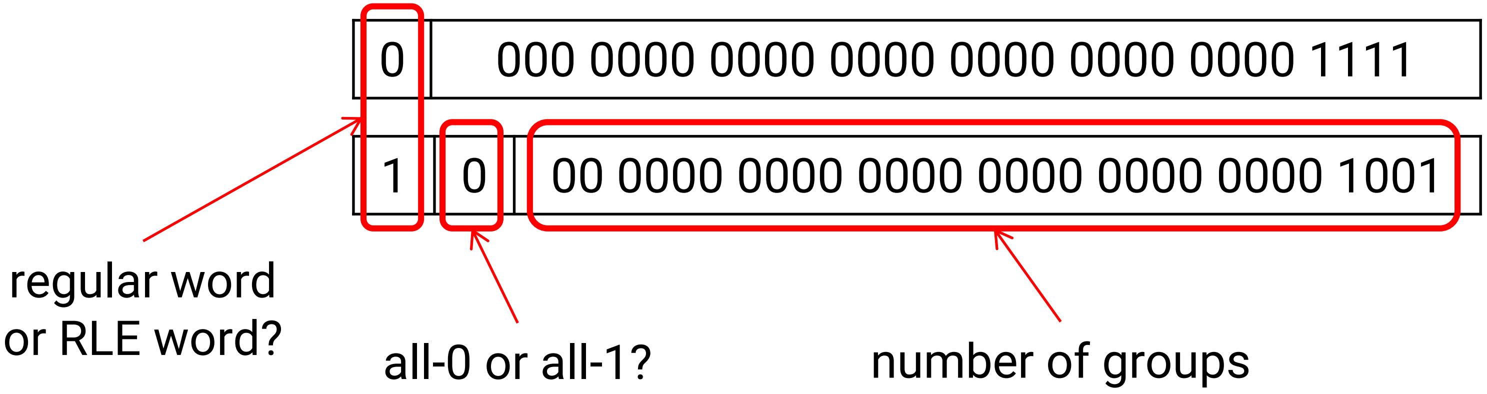
So Far So Good…

- Consider a retail database
  - $orders(oid,\ date,\ customer\_id,\ item\_id,\ amount,\ price)$
  - 1 billion records
    - size of each uncompressed bit vector ≈ **125MB** !!!
  - 1 million items
    - indexing $item\_id$: number of bit vectors = $\mathbf{10^6}$!!!
  - price ∈ $[0.01, 10^4)$, data type: $numeric(6, 2)$
    - "`price > 0 AND price < 100`" needs to read $\mathbf{10^4}$ bit vectors !!!

Problems	#1: **compress the bit vectors**
	#2: **reduce the total number of bit vectors**
	#3: **reduce the number of bit vectors that must be read**

Problem #1: Bitmap Compression

- Compress each bit vector to reduce storage footprints
- **R**un-**L**ength **E**ncoding is sufficient
  - Perform bitwise operations directly on **compressed** bit vectors
- A **well-studied** algorithm: **WAH**(Word Aligned Hybrid)
  - Breaking a bit vector into $(w-1)$-bit groups $(w = 32 \text{ } or \text{ } 64)$
  - Using RLE to encode **successive all-0** groups and **all-1** groups

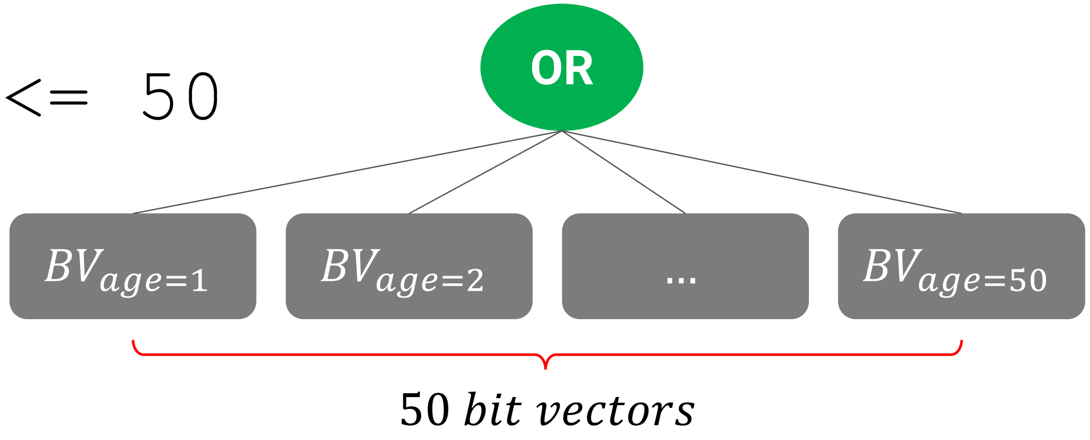


Wu K, Otoo EJ, Shoshani A. Optimizing Bitmap Indices With Efficient Compression. TODS 2006

Let's Look at Problem #3 First…

Problems	~~#1: **compress the bit vectors**~~ ✓
	#2: **reduce the total number of bit vectors**
	#3: **reduce the number of bit vectors that must be read**

- `WHERE age >= 1 AND age <= 50`



- Idea: **precomputation**
  - Aggregate some basic bit vectors in advance

Range Encoding

- Equality Encoding: $bits[rid][v] = 1 \leftrightarrow T[rid].attr = v$
- Range Encoding: $bits[rid][v] = 1 \leftrightarrow T[rid].attr \leq v$

T

RID	$Name$	Age
1	Alice	22
2	Bob	23
3	Daniel	20
4	Smith	18
5	Smith	21
6	Smith	19

$T[3].age = 20$

$\{rid \mid T_{rid}.age = 22\}$

	18	19	20	21	22	23
1	**0**	**0**	**0**	**0**	**1**	**0**
2	**0**	**0**	**0**	**0**	**0**	**1**
3	**0**	**0**	**1**	**0**	**0**	**0**
4	**1**	**0**	**0**	**0**	**0**	**0**
5	**0**	**0**	**0**	**1**	**0**	**0**
6	**0**	**1**	**0**	**0**	**0**	**0**

$T[3].age \leq 20$

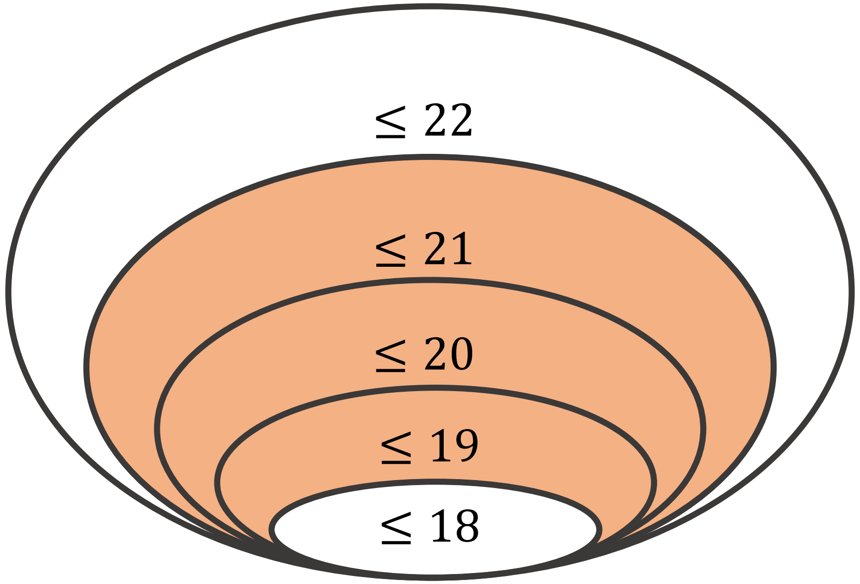
$\{rid \mid T_{rid}.age \leq 22\}$

	18	19	20	21	22	23
1	**0**	**0**	**0**	**0**	**1**	**1**
2	**0**	**0**	**0**	**0**	**0**	**1**
3	**0**	**0**	**1**	**1**	**1**	**1**
4	**1**	**1**	**1**	**1**	**1**	**1**
5	**0**	**0**	**0**	**1**	**1**	**1**
6	**0**	**1**	**1**	**1**	**1**	**1**

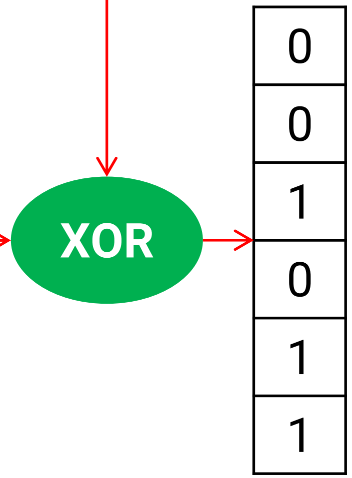
Range Encoding: Query Processing

- $[l, u] = [min, u] - [min, l)$
  - $\{rid \mid l \leq attr \leq u\} = \{rid \mid attr \leq u\} - \{rid \mid attr < l\}$
  - Any query need only read **at most 2** bit vectors

$\{19 \leq Age \leq 21\} = \{\leq 21\} - \{\leq 18\}$

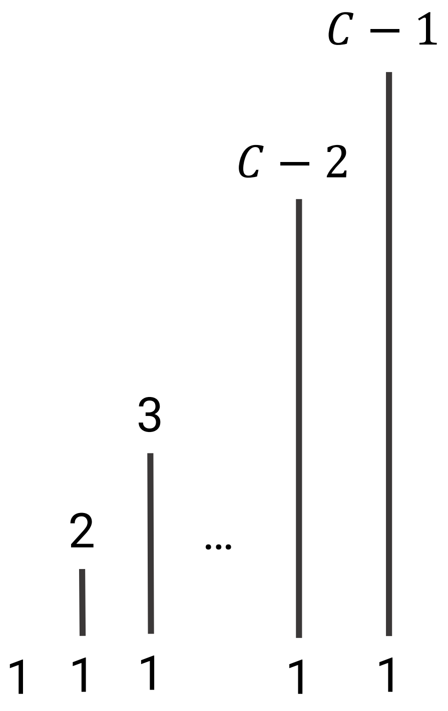


	18	19	20	21	22	23
1	0	0	0	0	1	1
2	0	0	0	0	0	1
3	0	0	1	1	1	1
4	1	1	1	1	1	1
5	0	0	0	1	1	1
6	0	1	1	1	1	1

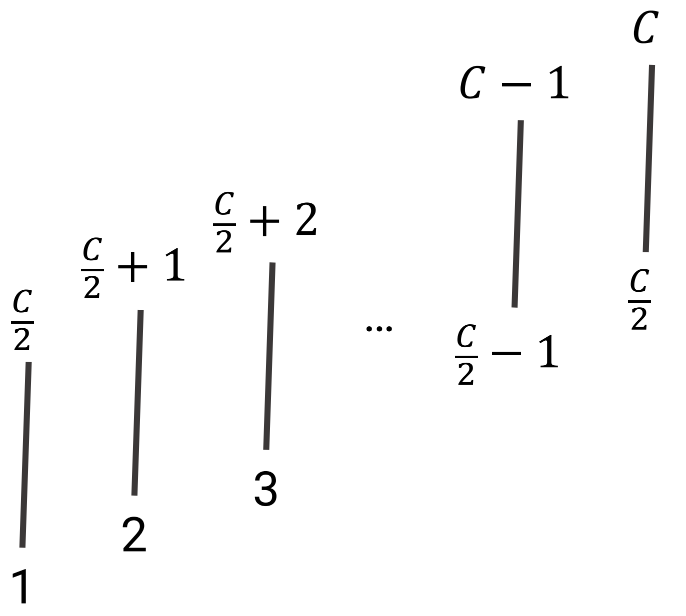


Further Improvement: Interval Encoding

- Require only $\boldsymbol{C/2}$ bit vectors to be stored
  - Similar to range encoding: need only read at most 2 bit vectors



Range Encoding



Interval Encoding

$$[1,3] = \left[1, \frac{C}{2}\right] \wedge \overline{\left[3, \frac{C}{2}+2\right]}$$

$$[1, C-1] = \left[1, \frac{C}{2}\right] \vee \left[\frac{C}{2}-1, C-1\right]$$

Chan CY, Ioannidis YE. An Efficient Bitmap Encoding Scheme for Selection Queries. SIGMOD 1999

Range/Interval Encoding: *Bad News*

- Interval encoding and range encoding are **hard to compress**
  - Each bit vector contains more information → higher entropy
- Information theoretic analysis
  - Under ideal compression:

$$\frac{\mathit{worst\text{-}case\text{-}size}_{\text{interval-encoding}}}{\mathit{worst\text{-}case\text{-}size}_{\text{equality-encoding}}} \approx \frac{C}{\log C}$$

  - interval/range encoded bitmap indexes are not suited for **high-cardinality** attributes!

Multi-Resolution Bitmap Indexes

- finest level: basic bitmap index (highest resolution)
- coarser levels: <u>**binning**</u> values to reduce the # of bit vectors

T

RID	*Name*	*Age*
1	Alice	22
2	Bob	23
3	Daniel	20
4	Smith	18
5	Smith	21
6	Smith	19

OR

18	19	20	21	22	23
0	0	0	0	1	0
0	0	0	0	0	1
0	0	1	0	0	0
1	0	0	0	0	0
0	0	0	1	0	0
0	1	0	0	0	0

6 bit vectors

18-19	20-21	22-23
0	0	1
0	0	1
0	1	0
1	0	0
0	1	0
1	0	0

3 bit vectors

Sinha RR, Winslett M. Multi-Resolution Bitmap Indexes for Scientific Data. TODS 2007

Querying Multi-Resolution Bitmap Indexes

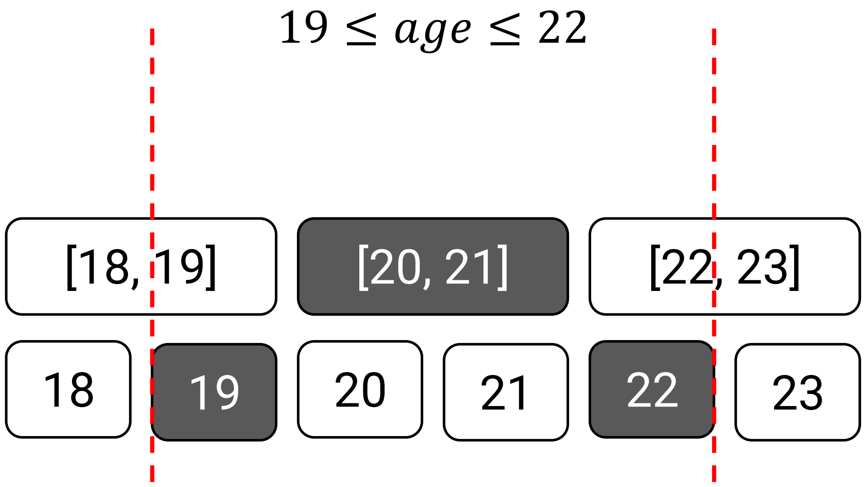
- Prefer coarse-level bit vectors

higher resolution

18	19	20	21	22	23
0	0	0	0	1	0
0	0	0	0	0	1
0	0	1	0	0	0
1	0	0	0	0	0
0	0	0	1	0	0
0	1	0	0	0	0

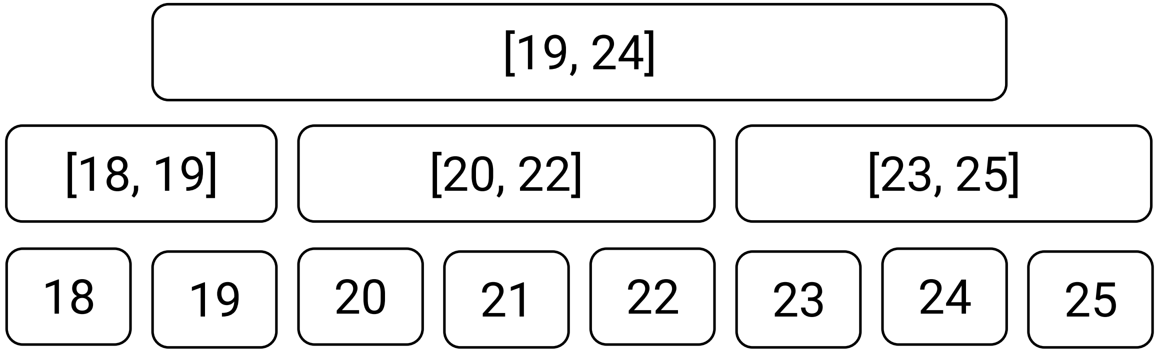
lower resolution

18-19	20-21	22-23
0	0	1
0	0	1
0	1	0
1	0	0
0	1	0
1	0	0



Time-Space Tradeoff

- Multi-resolution design can be very flexible
  - Fixed/variable bin width
  - Number of levels
  - Different encodings at each level
    - equality/range/interval



- In general, maintaining more high-level bit vectors may **benefit more queries** at the cost of **larger storage footprint**

Problem #2: Reduce the Total # of Bit Vectors

- Number of bit vectors that need to be stored:
  - Equality encoding: $\boldsymbol{C}$
  - Range encoding: $\boldsymbol{C-1}$
  - Interval encoding: $\boldsymbol{C/2}$
  - Multi-resolution: $\boldsymbol{> C}$
- However, $\boldsymbol{C}$ can be arbitrarily large
- Can we reduce the total number of bit vectors **greatly**?

Multi-Component Design

- Indexing the age attribute: integers $\in [0, 199]$

$200\ bit\ vectors$

0	1	…	99	…	199
1	0	…	0	…	0
0	0	…	1	…	0
0	0	…	1	…	0
0	0	…	1	…	0
0	1	…	0	…	0
0	0	…	0	…	1

$22\ bit\ vectors$

$2\ bit\ vectors$

0	1
1	0
1	0
1	0
1	0
1	0
0	1

百位

$10\ bit\ vectors$

0	1	…	9
0	0	…	0
0	0	…	1
0	0	…	1
0	0	…	1
1	0	…	0
0	0	…	1

十位

$10\ bit\ vectors$

0	1	…	9
1	0	…	0
0	0	…	1
0	0	…	1
0	0	…	1
0	1	…	0
0	0	…	1

个位

Multi-Component Design: Query Processing

- `WHERE 50 <= age AND age <= 100`
  - $(A_3 = 1 \wedge A_2 = 0 \wedge A_1 = 0) \vee (A_3 = 0 \wedge A_2 \geq 5)$

V_3^0	V_3^1
1	0
1	0
1	0
1	0
1	0
0	1

百位(A_3)

V_2^0	V_2^1	…	V_2^9
0	0	…	0
0	0	…	1
0	0	…	1
0	0	…	1
1	0	…	0
0	0	…	1

十位(A_2)

V_1^0	V_1^1	…	V_1^9
1	0	…	0
0	0	…	1
0	0	…	1
0	0	…	1
0	1	…	0
0	0	…	1

个位(A_1)

$$(V_3^1 \wedge V_2^0 \wedge V_1^0) \vee \left(V_3^0 \wedge \bigvee_{i=5}^{9} V_2^i\right)$$

Max # of Components: Binary Encoding

- Each **bit** as a component
  - also called $bit - sliced\ index$
- Each component can take 2 possible values (0 or 1)
  - **1 bit vector per component**
- Number of bit vectors $= \lceil \mathbf{log_2} \boldsymbol{C} \rceil$

0	0		0	0
1	1		1	0
1	0	…	1	0
1	1		0	0
0	0		0	0
1	1		1	1

$\underbrace{\qquad\qquad\qquad\qquad}_{7\ bit\ vectors}$

0	1	…	64	…	127
1	0	…	0	…	0
0	0	…	1	…	0
0	0	…	1	…	0
0	0	…	1	…	0
0	1	…	0	…	0
0	0	…	0	…	1

Basic Bitmap Index vs Bit-Sliced Index

one-component equality encoding (basic bitmap index)	min ⟵ #components ⟶ max	binary encoding (bit-sliced index)

✓ optimal for point queries
✗ higher storage overhead

✓ minimum number of bit vectors
✗ most of the bit vectors have to be accessed

- Analysis shows **both** indexes achieve the <u>**minimal**</u> index sizes and query processing costs
- A counter-intuitive conclusion: binary-encoding is more efficient for lower cardinality attributes

Wu K, Shoshani A, Stockinger K. Analyses of Multi-Level and Multi-Component Compressed Bitmap Indexes. TODS 2010

Outline

- Exact Indexes
  - Bitmap Index

- Approximate Indexes
  - Block-Level Bitmap Index
  - Zone Map
  - Bloom Filter
  - Range Filter

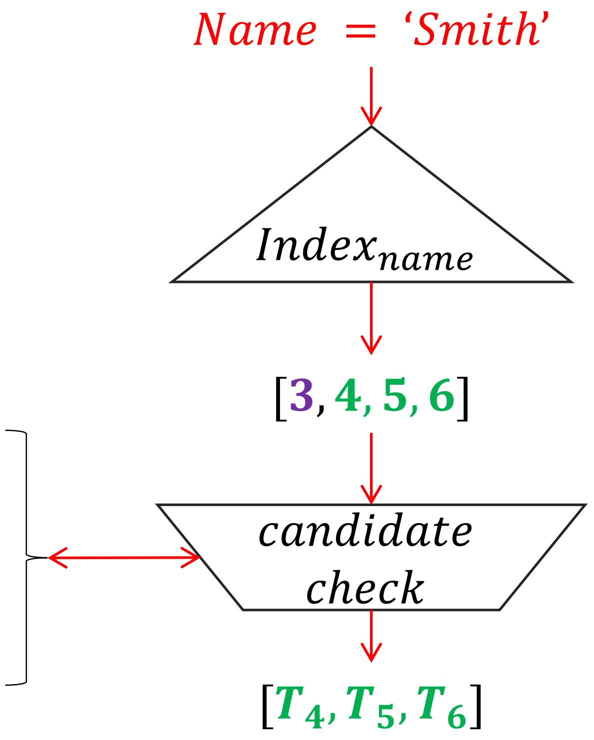
Approximate Indexes

- *predicate* → *RID list*
  - Return the IDs of all **candidates** that **may** satisfy a given predicate

- Requirements
  - No false negative
  - Small **false positive** rate
  - Small storage footprints
    - fit into memory

T

RID	$Name$	Age
1	Alice	22
2	Bob	23
3	Daniel	25
4	Smith	18
5	Smith	21
6	Smith	30



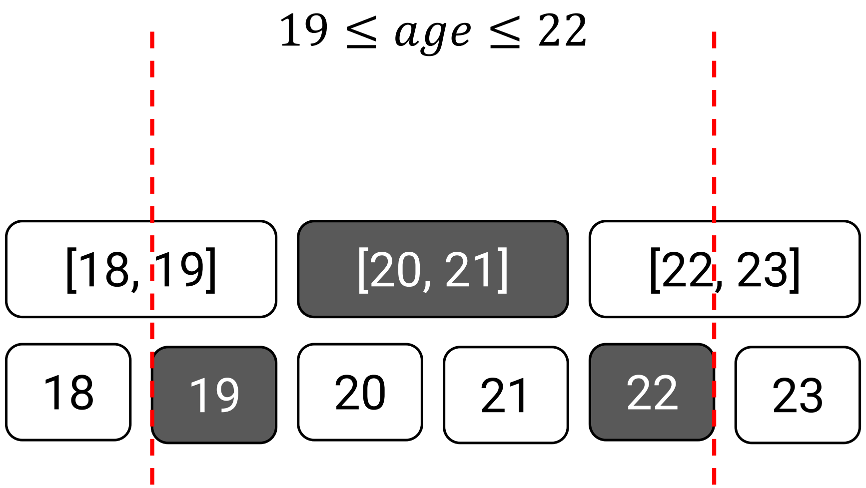
Recall the Multi-Resolution Bitmap Index

higher resolution

18	19	20	21	22	23
0	**0**	**0**	**0**	**1**	**0**
0	**0**	**0**	**0**	**0**	**1**
0	**0**	**1**	**0**	**0**	**0**
1	**0**	**0**	**0**	**0**	**0**
0	**0**	**0**	**1**	**0**	**0**
0	**1**	**0**	**0**	**0**	**0**

lower resolution

18-19	20-21	22-23
0	**0**	**1**
0	**0**	**1**
0	**1**	**0**
1	**0**	**0**
0	**1**	**0**
1	**0**	**0**

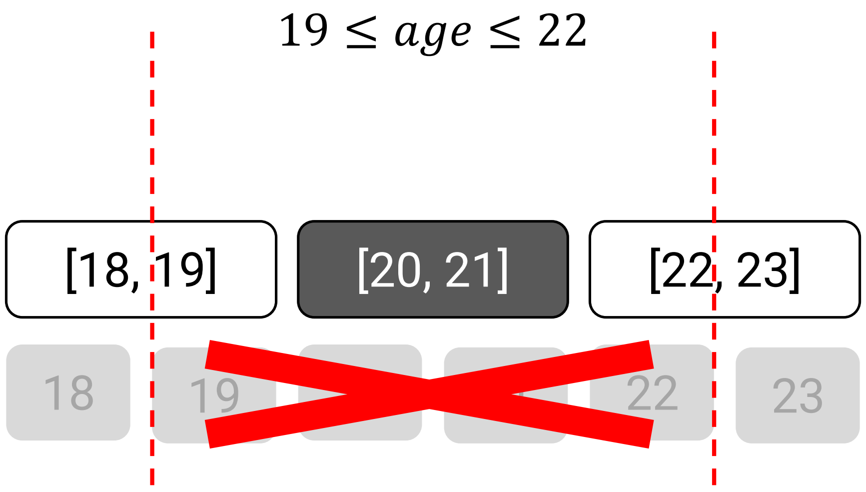


Towards Approximate Indexes

- Now assume only the **low-resolution** bit vectors can fit into the storage budget
- All RIDs in [18, 19] and [22, 23] should be reported as candidates
  - Introduce **false positives**

18	19	20	21	22	23
0	0	0	0	1	0
0	[illegible]	0	0	[illegible]	1
0	0	[illegible]	[illegible]	0	0
1	[illegible]	0	0	[illegible]	0
0	0	0	1	0	0
0	1	0	0	0	0

18-19	20-21	22-23
0	0	1
0	0	1
0	1	0
1	0	0
0	1	0
1	0	0



Let's Adjust Our Viewpoints…

- In above discussion:
  - View the bit matrix as **a collection of columns**
  - Bin and pre-aggregate these columns

RID	*Name*	*Age*
1	Alice	22
2	Bob	23
3	Daniel	20
4	Smith	18
5	Smith	21
6	Smith	19

	18	19	20	21	22	23
1	0	0	0	0	1	0
2	0	0	0	0	0	1
3	0	0	1	0	0	0
4	1	0	0	0	0	0
5	0	0	0	1	0	0
6	0	1	0	0	0	0

18-19	20-21	22-23
0	0	1
0	0	1
0	1	0
1	0	0
0	1	0
1	0	0

Let's Adjust Our Viewpoints…

- Now, let us:
  - View the bit matrix as <u>**a collection of rows**</u>

Because IO operations are performed at the block level,
we are interested in which blocks, rather than which records, are qualified

BID	*RID*	*Name*	*Age*
1	1	Alice	22
	2	Bob	23
2	3	Daniel	20
	4	Smith	18
3	5	Smith	21
	6	Smith	19

18	19	20	21	22	23
0	0	0	0	1	0
0	0	0	0	0	1
0	0	1	0	0	0
1	0	0	0	0	0
0	0	0	1	0	0
0	1	0	0	0	0

Bit Vectors At the Block Level

- Now, let us:
  - View the bit matrix as a collection of rows
  - Bin these rows: aligning the **bin width** with the **I/O unit**, i.e., data block
- This view is useful because we manage records in blocks
  - Moreover, block-oriented processing can be heavily paralleled

BID	*RID*	*Name*	*Age*
1	1	Alice	22
	2	Bob	23
2	3	Daniel	20
	4	Smith	18
3	5	Smith	21
	6	Smith	19

18	19	20	21	22	23
0	0	0	0	1	0
0	0	0	0	0	1
0	0	1	0	0	0
1	0	0	0	0	0
0	0	0	1	0	0
0	1	0	0	0	0

18	19	20	21	22	23
0	0	0	0	1	1
1	0	1	0	0	0
0	1	0	1	0	0

bit vector

Sidirourgos L, Kersten M. Column Imprints: A Secondary Index Structure. SIGMOD 2013

Query Processing

- Construct a bit vector V_q for the query
- If $Vectors_{BID} \wedge V_q \neq 0$, report all RIDs in $Block_{BID}$ as candidates

T

BID	RID	$Name$	Age
1	1	Alice	22
	2	Bob	23
2	3	Daniel	20
	4	Smith	18
3	5	Smith	21
	6	Smith	19

18	19	20	21	22	23
0	**0**	**0**	**0**	**1**	**1**
1	**0**	**1**	**0**	**0**	**0**
0	**1**	**0**	**1**	**0**	**0**

$19 \leq age \leq 21$

AND
011100

$Result$

000000
001000
010100

$[\mathbf{3}, \mathbf{4}, \mathbf{5}, \mathbf{6}]$

High-Cardinality Attributes, Again…

- Must compress these bit vectors
  - Convert them into a more **compact** format

T

BID	RID	$Name$	$PostCode$
1	1	Alice	000012
	2	Bob	100100
2	3	Daniel	123456
	4	Smith	233334
3	5	Smith	000012
	6	Smith	111111

$bit\ vectors$
000….1….0010….0
000….1….0010….0
000….1….0010….0

$10^6\ bits$

Generalized Problem

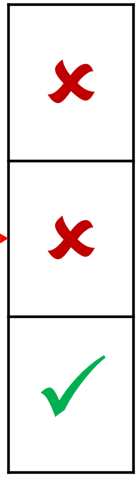
- Maintain a **compact data structure** for each block
  - Can be quickly checked to filter out **unqualified blocks**
    - Avoid loading these blocks into memory

T

BID	RID	$Name$	$PostCode$
1	1	Alice	000012
	2	Bob	100100
2	3	Daniel	123456
	4	Smith	233334
3	5	Smith	000012
	6	Smith	111111

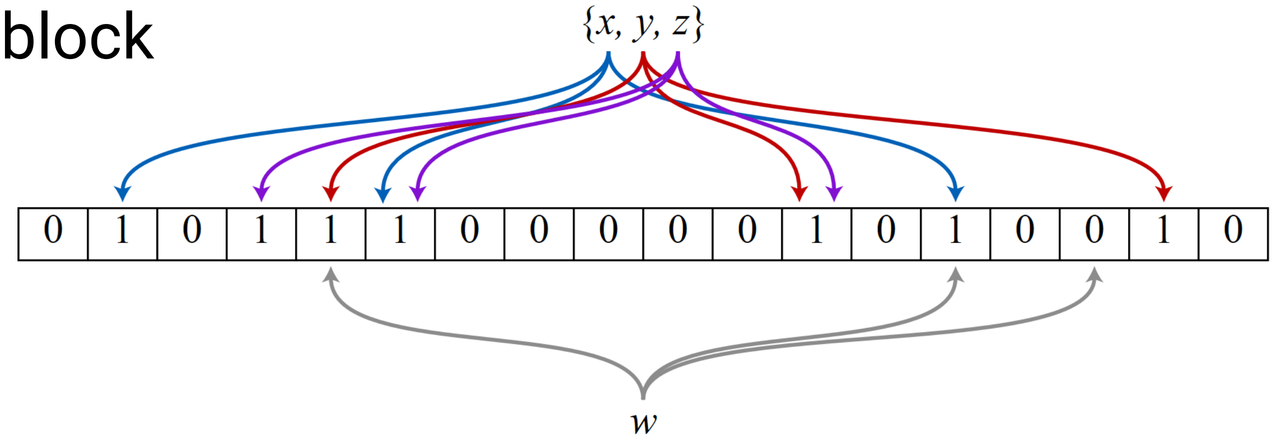
BID	$compact\ data\ structure$
1	???
2	???
3	???

$\rightarrow 19 \le age \le 21 \rightarrow$



A Standard Solution: Bloom Filters

- A well-studied probabilistic data structure
- Used to test whether an element is a member of a set
  - Answers: “possibly in set” or “definitely not in set”
    - False positive: “possibly in set” → actually not in set
    - ~~False negative: “definitely not in set” → actually in set~~
- In our case: test whether the block contains qualified records
  - YES: must load this block into memory and recheck it
  - NO: can safely discard this block



Processing Range Queries with Bloom Filters?

- Unfortunately, bloom filters can only handle point queries
  - e.g., `WHERE name = 'Smith'`
- Range queries: $[lower,\ upper]$
- Idea: test the bloom filter $(upper - lower + 1)$ times
  - `WHERE age >= 19 AND age <= 22`: test 19, 20, 21 and 22
  - **Impractical** for large range queries ☹
  - Moreover, false positive rate will be $(upper - lower + 1) \times p$

Range Queries: Zone Map

- Maintain the **min/max** values of each zone
  - Each zone can contain more than 1 block

ZoneID	ZM_{year}	ZM_{nation}	ZM_{price}
1	[2010, 2011]	[CA, KR]	[10.5, 40.0]
2	[2012, 2013]	[KR, US]	[20.0, 35.0]

BID
1
2
3
4

Year	Nation	Price
2010	CN	40.0
2010	CA	10.5
2011	JP	30.5
2011	KR	35.0
2012	US	20.0
2012	UK	35.0
2013	KR	23.0
2013	RU	33.0

Zone Map: Query Processing

- $\color{red}{Interval_{query} \cap ZM_{ZID} = \emptyset}$ $\rightarrow$ discard all blocks in $Zones_{ZID}$
  - E.g., `SELECT * FROM t WHERE year < 2011`
    - Block #3 and #4 can be skipped

- Useful for **<u>clustered/ordered</u>** attributes
- For arbitrary attributes, its efficiency depends on the distribution of data
- Worst case: each block contains the minimum and maximum value

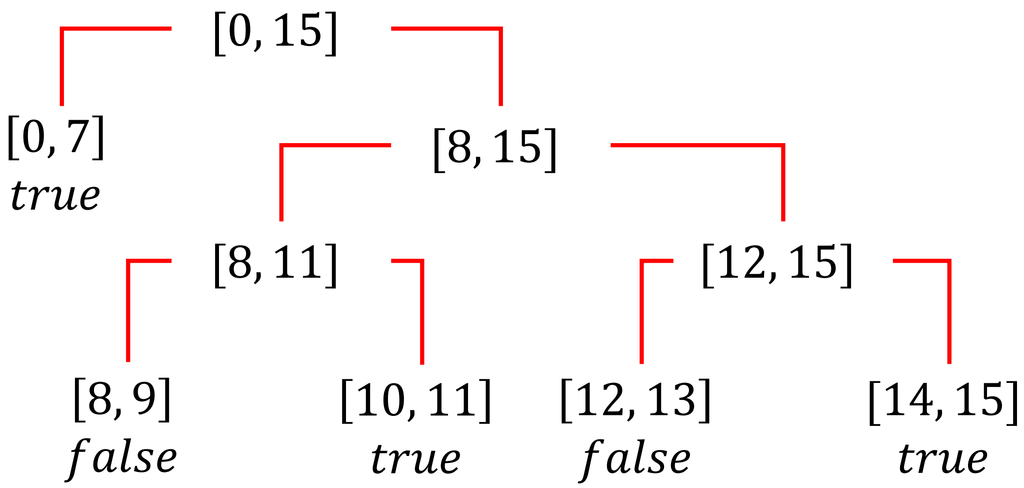
ZoneID	ZM_{year}
1	**[2010, 2011]** ✓
2	[2012, 2013] ✗

BID
1
2
3
4

Year	Nation	Price
2010	**CN**	**40.0**
2010	**CA**	**10.5**
2011	JP	30.5
2011	KR	35.0
2012	US	20.0
2012	UK	35.0
2013	KR	23.0
2013	RU	33.0

Range Filter

- Motivation: the **granularity** of a $[min, max]$ pair is **too coarse**
  - Can result in high false positive rate
- Idea: use a tree structure to record <u>**fine-grained**</u> information



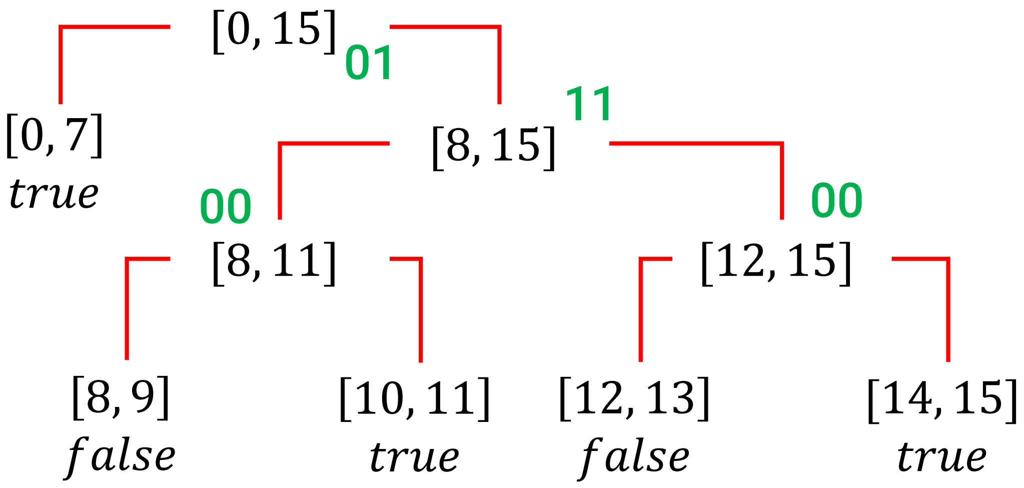
- A internal node is always divided into <u>**two equal-width sub-intervals**</u>
- Each leaf node records whether the block contains values in this interval
- <u>**False positive is still allowed**</u> – the tree need not grow into the finest level

Age	0	1	2	2	3	5	6	11	11	15

Alexiou K, Kossmann D, Larson PÅ. Adaptive Range Filters for Cold Data: Avoiding Trips to Siberia. VLDB 2013

Encoding the Range Filter

- Necessary information
  - root node, the **structure** of the tree, $true/false$ on leaves
- Internal nodes are encoded according to their children
  - Represented using two bits: $00, 01, 10, 11$ (0: leaf, 1: internal node)
  - Breadth-first



$Root$	$[0, 15]$
$Structure$	$01\ 11\ 00\ 00$
$Leaves$	10101

Range Filter: Learning and Adaptation

- Tree grows by splitting leaves and shrinks by merging leaves
- Splitting leaves: reduce the false positive rate
  - Learning from $false\ positives$
    - don't make the same mistake twice
  - Learning from $true\ positives$
    - $Query([5,15]) = \{7,9\}$ $\rightarrow [5,6], [8,8]$ and $[10,15]$ are empty
- Merging leaves: reduce the storage footprint
  - Similar to the $buffer\ replacement$ problem
  - Known $replacement\ policy$ can be used: **LRU**, **LFU**, **LRU-K**
  - Merges cascade if two leaves have the same value

Recall the Block-Level Bit Vectors

- Bloom Filter: lossy compression of these bit vectors
- Zone Map: storing the leftmost and rightmost 1-bits
- Range Filter: multi-resolution design, again ☺

New Opportunities?

T

BID	RID	$Name$	$PostCode$
1	1	Alice	000012
	2	Bob	100100
2	3	Daniel	123456
	4	Smith	233334
3	5	Smith	000012
	6	Smith	111111

$bit\ vectors$
000….1….0010….0
000….1….0010….0
000….1….0010….0

$10^6\ bits$

One Final Problem

- Too many blocks
  - scanning the `(bit vector | zone map | bloom filter | range filter)` of each block linearly can be slow

- Bit vector: multi-resolution design
- Zone map/range filter: interval tree
- Bloom filter: tree-structured bloom filter

Summary

- Both basic bitmap index and bit-sliced index are OK
  - Must be compressed
  - Space-time tradeoff: multi-resolution design
- Bloom filter and zone map are standard approximate indexes
  - Bloom filter can not handle range queries efficiently
  - Zone map is ill-suited to non-clustered attributes
- Range filter makes a great improvement
  - However, it is relatively new and is not well-studied

Backup Slides

Properties of Bloom Filters

- False Positive Rate

$$\left(1-\left[\left(1-\frac{1}{m}\right)\right]^{kn}\right)^{k} \approx \left(1-e^{kn/m}\right)^{k}$$

- Optimal number of hash functions

$$k = \frac{m}{n}\ln 2$$

- Given p and n

$$m = -\frac{n \ln p}{(\ln 2)^2}, \quad k = -\frac{\ln p}{\ln 2}$$

Basic Bitmap Index vs Bit-Sliced Index

	Basic Bitmap Index	Bit-Sliced Index
$s = Index\ Size$	$2N$ $(if\ 1 \ll C \ll N)$	$\frac{N\log_2 C}{w-1}$
$Equality\ Query$	s/C	s
$One\text{-}Sided\ Range\ Query$	$s/4$	$\frac{N(\log_2 C - 1)}{w-1}$
$Two\text{-}Sided\ Range\ Query$	$s/4$	$\frac{N(\log_2 C - 0.5)}{w-1}$

Wu K, Shoshani A, Stockinger K. Analyses of Multi-Level and Multi-Component Compressed Bitmap Indexes. TODS 2010