Indexing Algorithms and Data Structures

David Boyuka, daboyuka@ncsu.edu
North Carolina State University
March 31, 2015

What Is an Index?

- A data structure to speed up certain types of access to a dataset vs. a sequential scan
  - Minimizes costly access to slow storage (disk, RAM?)

- Data consists of "records" (sometimes "documents")
  - Generally, each record has an ID, known as an "RID"

- Data has "attributes" ("columns," "variables," "fields")
  - Each record may have a value for each attribute
  - Attributes in semi-structured data sparsely cover records

Where Are Indexes* Used?

- Database Management Systems (DBMSs)
  - Relational
  - Distributed
  - Key-value store

- Web and Document Search

- File Systems

**While the plural "indices" of index is more common in general usage, "indexes" is common in certain contexts, including database literature.*

Overview

- “Traditional” DBMS indexes

- Bitmap indexes

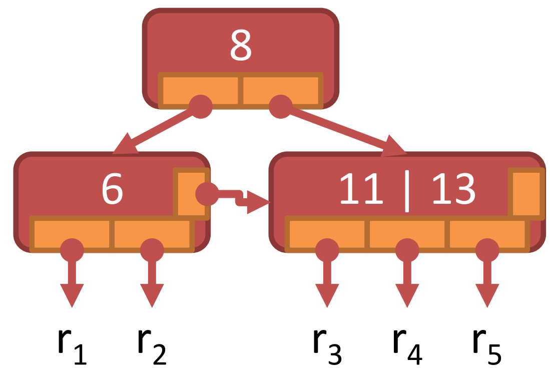
- Inverted indexes

Overview

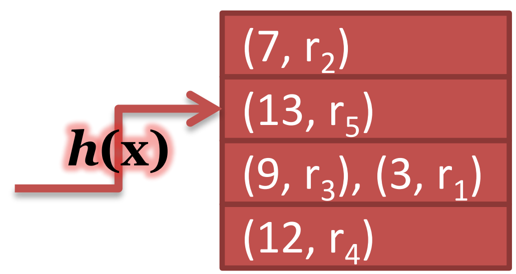
- **“Traditional” DBMS indexes**
- **Bitmap indexes**
- **Inverted indexes**

"Traditional" DBMS Indexes

- **B+ Tree Index**
  - Based on the B-tree search tree
  - Fast range and equality queries



- **Hash Index**
  - Based on the hash table
  - Very fast equality queries



- Both map search keys to matching record(s)

Oak Ridge National Laboratory

NC STATE UNIVERSITY
Department of Computer Science

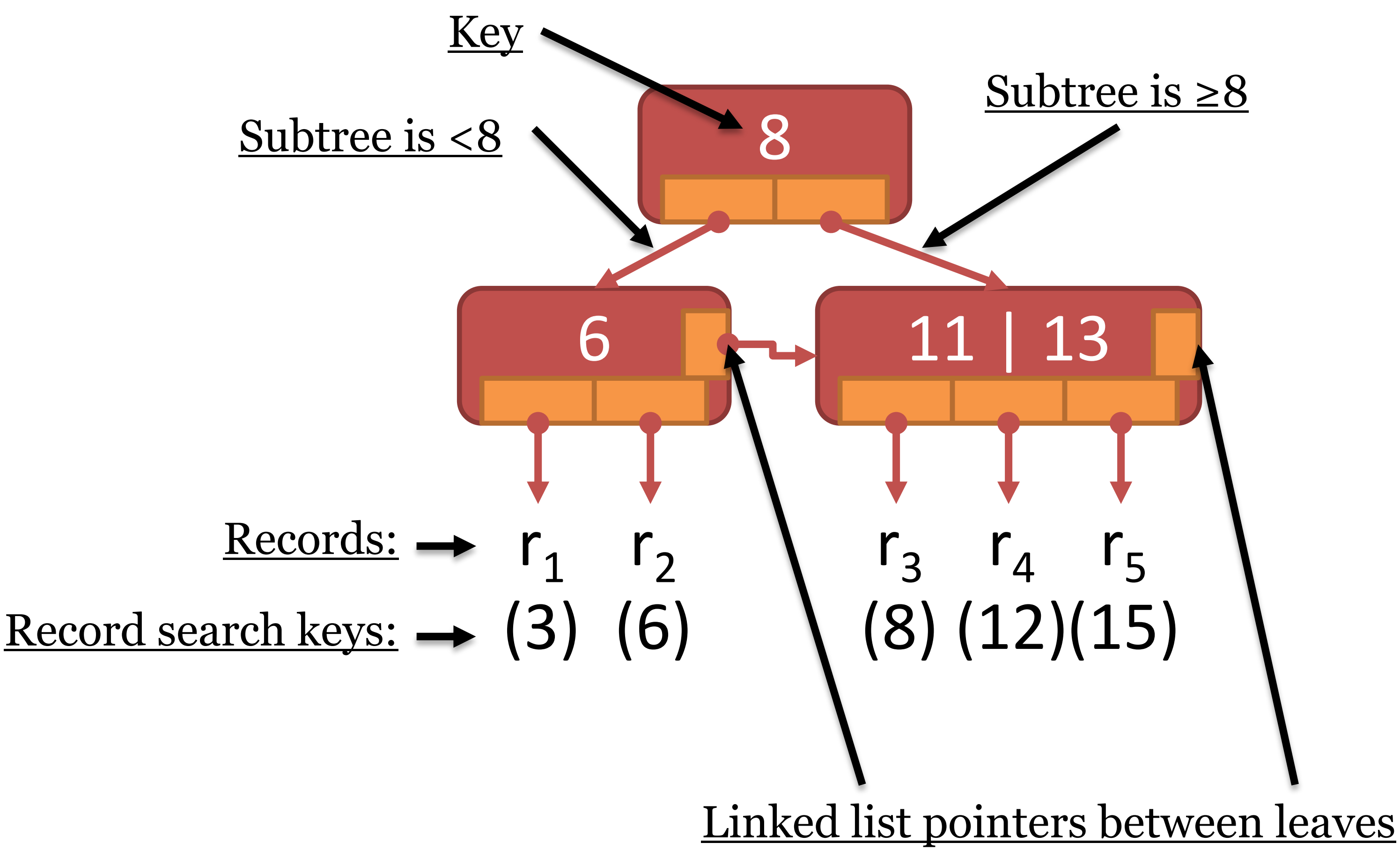
The B+ Tree

- Perfectly-balanced search tree with a wide fan-out
  - Allows search over some record attribute ("search key")
  - "Order" = b adjustable, each node has ceil(b/2) to b keys

- Leaf nodes hold k search keys, k+1 records
  - (record i) < (key i) ≤ (record i+1)
  - Leaf nodes connected via linked list

- Internal nodes have k search keys, k+1 child pointers
  - (subtree i) < (key i) ≤ (subtree i+1)

B+ Tree Example

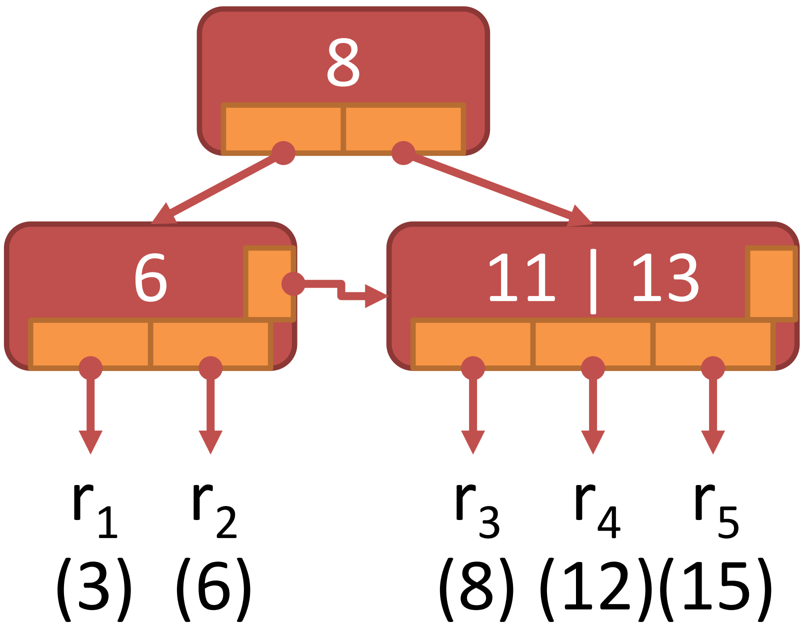


Inserting a Record into a B+ Tree

1. Find the leaf where the record belongs
2. Add the record and corresponding search key
3. If node has $\leq b$ records, done!
4. Else:
   a. Split the node, add the new half to the parent node
   b. Push the least key of the greater half to the parent
   c. If the parent has $\leq b$ records, done!
   d. Else, repeat from step 4a for the parent
   e. If the root splits, add a new root with two children

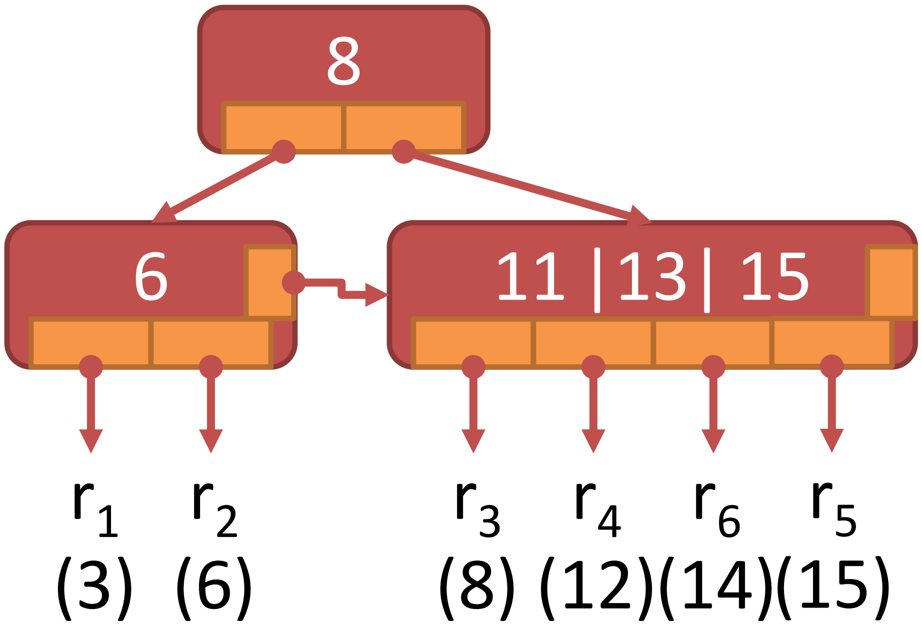
B+ Tree Insert Example

Insert r_6 with search key 14



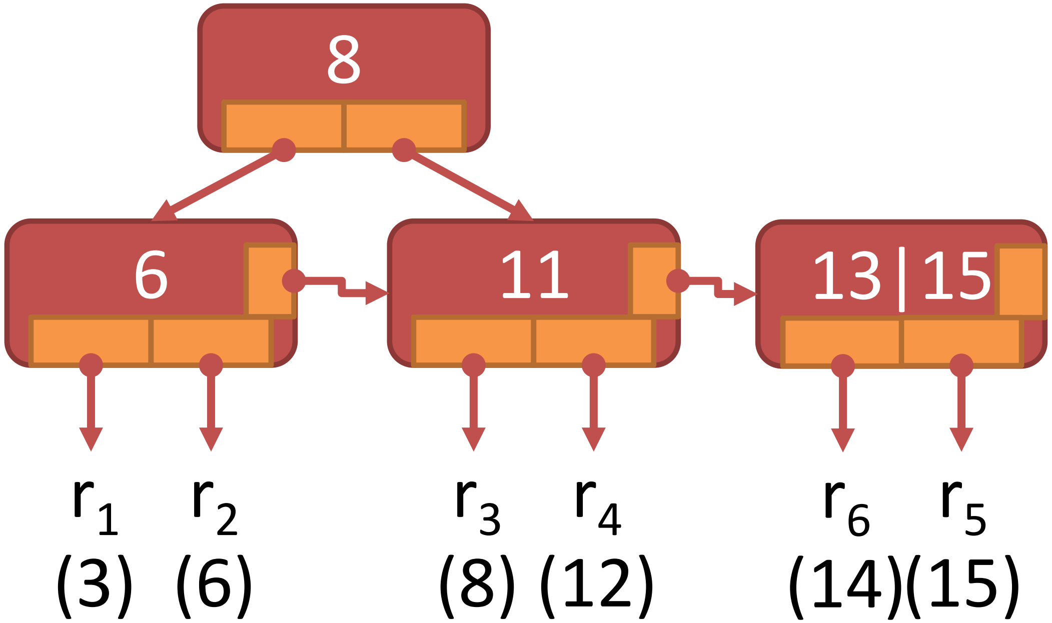
B+ Tree Insert Example

Insert r_6 with search key 14



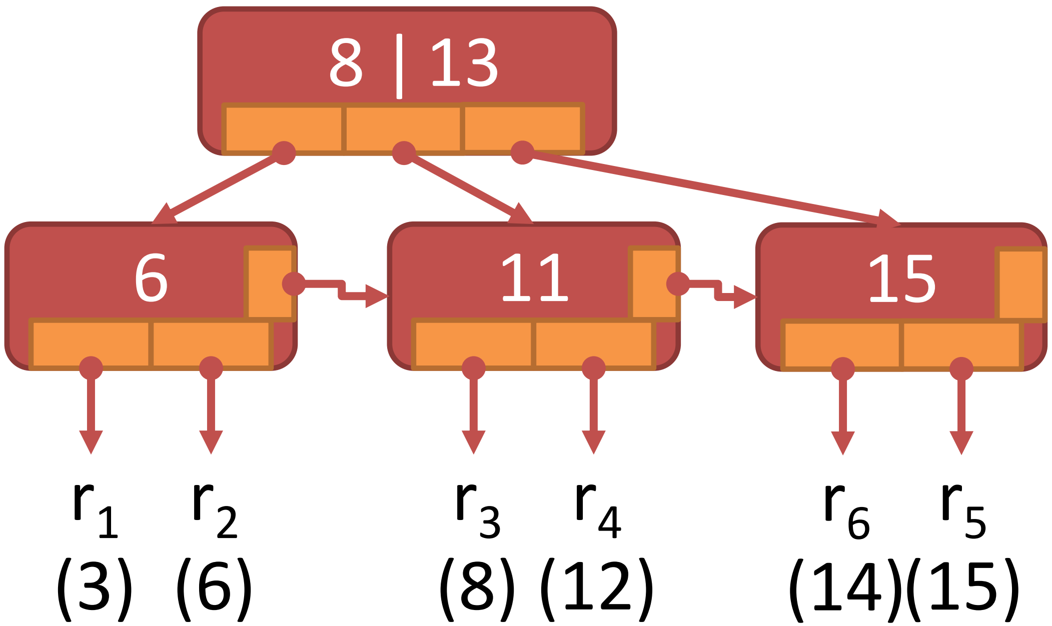
B+ Tree Insert Example

Insert r_6 with search key 14



B+ Tree Insert Example

Insert r_6 with search key 14



Deleting a Record from a B+ Tree

1. Find the leaf containing the record
2. Delete the record and the corresponding key
3. If node has ≥ ceil(b/2) records, done!
4. Else:
   a. Try stealing a key/record from an adjacent sibling
   b. If too few keys in all siblings, merge with a sibling
   c. If merged, repeat from 3 for parent node

(We'll skip the example and move on to other topics)

Clustered vs. Non-clustered B+ Tree

- **Non-clustered:**
  - Only pointers to records are stored, with full data stored elsewhere on disk
- **Clustered:**
  - Full records are stored directly in the leaves
- Maximum of **one** clustered index per table/dataset

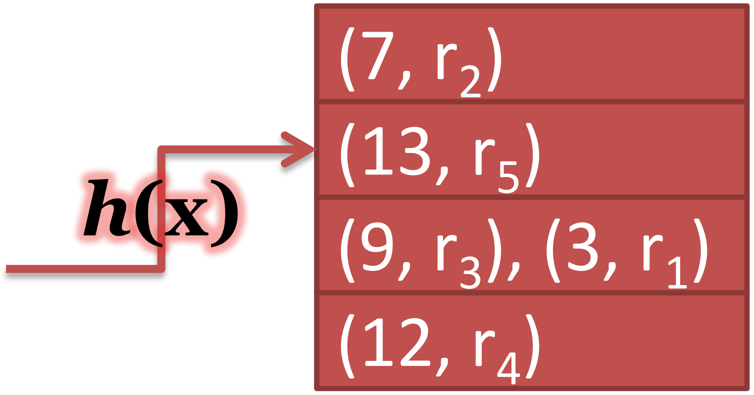
B+ Trees: Pros and Cons

+ Supports both range and equality queries
+ Inserts/deletes inexpensive (usually just modify 1 leaf)
+ Allows search key-sorted scan of data via linked leaves
– May use more seeks vs. other options
– May use more storage vs. other options
– Expensive to combine queries over multiple B+ trees (e.g. X>10 AND Y<20), as record IDs are unsorted

- Good for: high-cardinality attributes, value-sorted record listing, frequent updates

The Hash Index

- Essentially a hash table on disk (or other slow medium)
- Key/record pairs stored in buckets (one per disk block)
  - Overflow buckets via linked list as needed



Inserting a Record into a Hash Index

- Hash record to find the bucket, insert key/record pair
  - Add new overflow bucket if necessary
  - Overflow hurts performance

- Use "dynamic hashing" to grow hash table as needed

- Example: "extensible hashing"
  - In-memory bucket pointer array, double in size as needed
- Example: "linear hashing"
  - Add single new bucket when avg. records/bucket is high

Hash Index: Pros and Cons

- + ~1 seek per query (if overflow is limited), better than B+ tree
- + Fewer key compares than B+ tree, usually
- + Inserts/deletes inexpensive (usually just modify 1 bucket)
- – Only equality queries supported
- – May use more storage vs. other options

- Good for: faster equality queries, expensive key compares (e.g., long strings), frequent updates

Oak Ridge National Laboratory

NC STATE UNIVERSITY Department of Computer Science

Summary: Traditional DBMS Indexes

- **B+ Trees**
  - Range/equality queries
  - Inexpensive updates
  - Value-sorted record access
  - Storage space and seeks/query not the best

- **Hash Indexes**
  - Fast equality queries, but no range queries
  - Inexpensive updates
  - Better on seeks, still not best storage space

Overview

- "Traditional" DBMS indexes
- **Bitmap indexes**
- Inverted indexes

Bitmap Indexes

- Less common than B+ trees in DBMSs, but popular in certain applications, e.g., indexing scientific data
- Idea: assume r records and k attribute values $a_1, \ldots, a_k$
  - Build k bitmaps $b_1, \ldots, b_k$ of length r bits each
  - Bit i in b_j is set iff record i has attribute value a_j

(RID, value)

(1, X)
(2, Y)
(3, X)
(4, Z)

→

Attrib. value	Bitmap
X	1010
Y	0100
Z	0001

Oak Ridge National Laboratory

NC STATE UNIVERSITY Department of Computer Science

Benefits of Bitmap Indexing

- B+ trees/hash indexes allow query on single attributes*
- Multi-dimensional indexes exist (R tree, etc.), but quickly succumb to the "curse of dimensionality"
  - Combinatorial explosion…
- Bitmap indexes are built on single attributes, but can be harnessed to answer multi-attribute queries
- Also, bitmap indexes may use less storage than B+ trees

**B+ trees can be built on a list of attributes (just as one can sort with secondary and tertiary keys), but can only aid queries on a prefix of these attributes*

Querying a Bitmap Index (One Variable)

1. Read the bitmaps for values matching the query
2. Compute the bitwise OR of these bitmaps
3. (Optionally) convert final bitmap to RIDs

Range query:
var ∈ [X, Y]

Attrib. value	Bitmap
X	1010
Y	0100
Z	0001

1010 | 0100
= 1110

Dataset (RID, value): **(1, X), (2, Y), (3, X), (4, Z)**

Querying a Bitmap Index (Multivariate)

Example: (temperature > 90) AND (pressure < 1000)

1. Evaluate single-variable constraints to bitmaps, as last slide
2. Combine bitmaps via bitwise operations via expression tree
3. (Optionally) convert final bitmap to RIDs

Dataset (RID, temp., pres.):
(1, 85, 994), (2, 95, 995), (3, 87, 991), (4, 83, 1002) , (5, 92, 997) , (6, 93, 1004) , (7, 81, 1010) , (8, 97, 1003)

(temperature > 90) AND (pressure < 1000) =
01001000

↑

Bitwise AND (&)

↗ ↖

(temperature > 90) ⟶ **01001101**

(pressure < 1000) ⟶ **11101000**

Updating a Bitmap Index

- Naïve inserts/deletes in bitmap indexes would be costly
  - Inserting/deleting a bit in the middle shifts many bits
  - Bitmaps are fixed length, would need to expand/shrink
  - Bitmap compression exacerbates the situation

- To mitigate, manage where how record RIDs
  - New records have increasing RIDs, always append a bit
  - Deleted record RIDs are not reused; bit position remains

- Still, bitmaps best with read-mostly/bulk-append data

3 Key Techniques in Bitmap Indexing

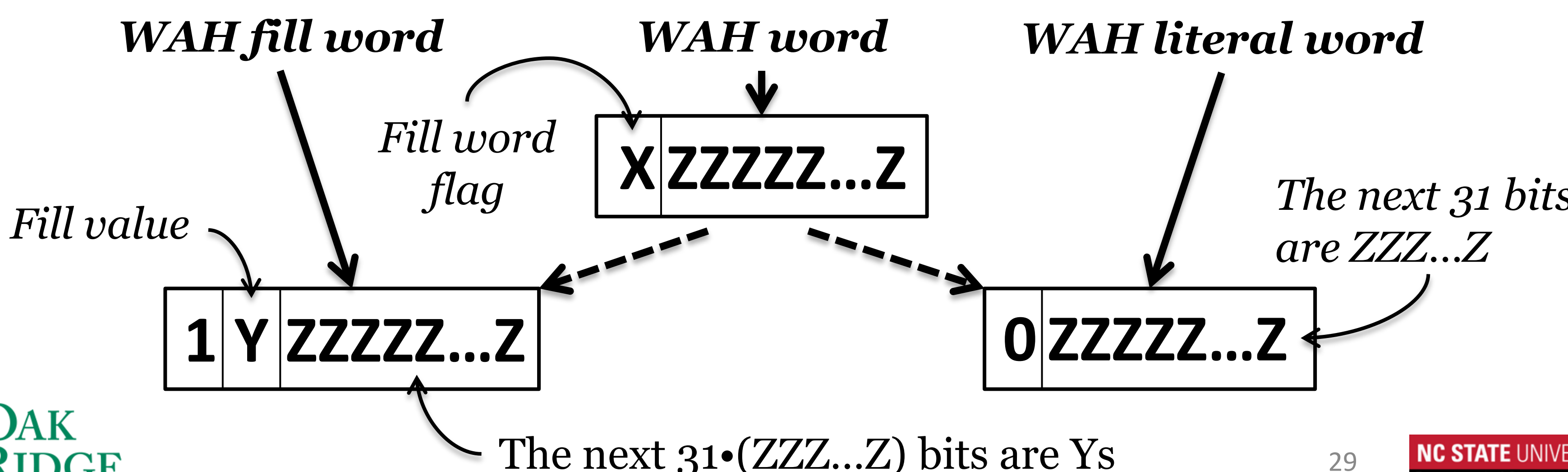
- **Compression**
  - Reduces index size, can speed up query
- **Encoding**
  - Trades index size, query I/O, query CPU time, etc.
- **Binning**
  - Enables indexing of high-cardinality attributes
- These techniques are orthogonal, can be combined

Compressing Bitmap Indexes

- When an attribute's distinct value count is high (i.e., high cardinality), bitmaps become storage-heavy
  - However, they also become sparse (i.e., mostly 0-bits)
  - Compression can give sublinear growth with cardinality
- Popular compression: Word-Aligned Hybrid (WAH)
  - Run-length encoding of 0-bit/1-bit runs
  - All operations aligned to machine words, for speed
  - AND/OR/NOT possible on WAH without decompression

WAH Bitmap Compression

- Bitmap compressed as a series of 32-bit "words"
  - First bit is a "fill word flag"; 0→literal word, 1→fill word
  - **Literal word**: the remaining 31 bits are used verbatim
  - **Fill word**: let f = second bit and the c = the last 30 bits. The fill word represents a $(31 \cdot c)$-long run of f-bits



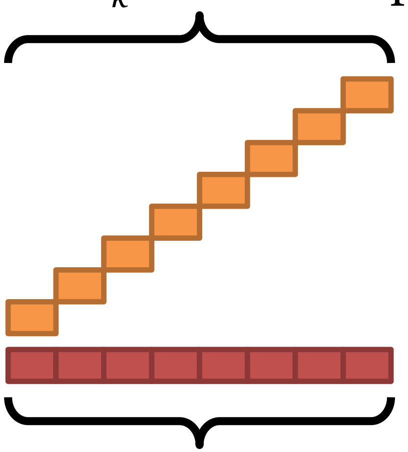
Bitmap Index Encodings

- A bitmap index with one bitmap per attribute value is considered "equality encoded"
- Other encoding options exist:
  - **Range encoding**: bit i in b_j set iff record i value $\le a_j$
  - **Interval encoding**: similar to range, more compact
  - **Binary encoding**: only ceil(log k) bitmaps, next slide
  - No longer a 1-1 association between bitmaps and values
- Recover per-attribute bitmaps via bitwise operations

Bitmap Index Encoding Examples

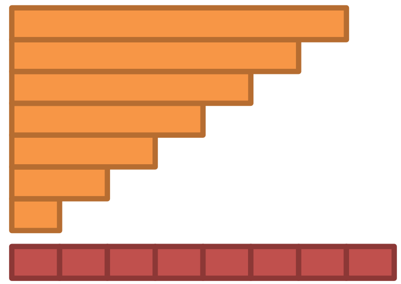
Equality Encoding

Bitmaps (b_1 to b_8)
(a_k covered → RIDs with value a_k set in bitmap)

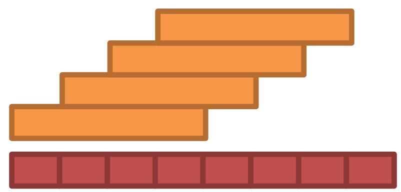


Attribute values (a_1 to a_8)

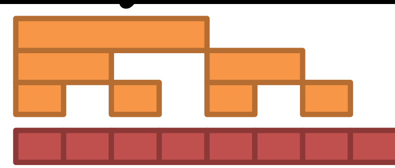
Range Encoding



Interval Encoding



Binary Encoding



Equality Encoding

(1, X), (2, Y)
(3, X), (4, Z) →
(5,W)

00001 (W)
10100 (X)
01000 (Y)
00010 (Z)

Range Encoding

(1, X), (2, Y)
(3, X), (4, Z) →
(5,W)

00001
10101
11101

Interval Encoding

(1, X), (2, Y)
(3, X), (4, Z) →
(5,W)

10101
11100

Binary Encoding

(1, X), (2, Y)
(3, X), (4, Z) →
(5,W)

01001
10101

Why Use Other Bitmap Encodings?

- Each bitmap encoding optimizes for some property

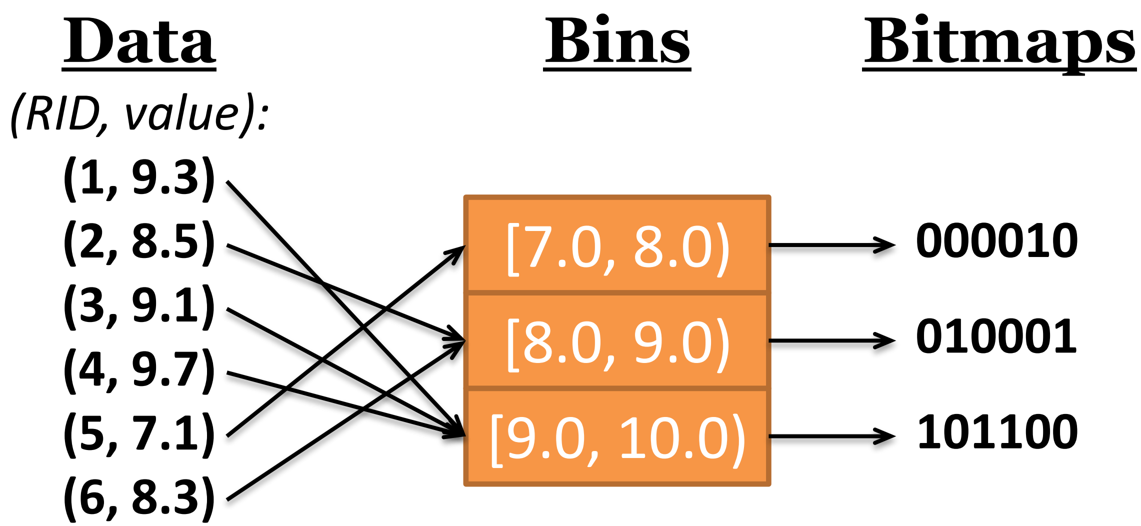
- **Range encoding**: fast range queries
  - Range query $[a_i, a_j) = b_j - b_i$ (at most 2 bitmaps needed)
  - However, index compression becomes less effective

- **Interval encoding**: similar to range, fewer bitmaps

- **Binary encoding**: fewer bitmaps = reduced storage
  - However, every query requires reading most bitmaps

Binning for Bitmap Indexes

- Suppose we want to index a floating-point attribute
  - Need one per unique value, which is intractable
- Idea: collect values into "bins," use one bitmap per bin
  - Each "bin" groups together a range of values



Types of Binning

- **Equal-width:** each bin has a fixed width *k*
  - Easy to compute, sometimes too rigid
- **Equal-frequency:** adjust *k* bins to each match approx. equal number of records
  - Guarantees balanced binning
  - Requires sampling and/or knowledge of data distribution
- **Precision:** round values to *k* significant digits
  - Easy to compute, bin widths adaptive to scale of data
  - May allocate too much precision near 0

The Cost of Binning

- Binning greatly reduces effective data cardinality
  - Improves bitmap compression, query performance

- However: exact query results no longer possible
  - If a query partially covers a bin, which records match?

- **Option 1:** Accept error in results
  - Fast, but (bounded) error in results
- **Option 2:** Do "candidate checks" on uncertain records
  - Exact results, but costly random access

Summary: Bitmap Indexing

- Bitmap indexes offer an alternative to traditional DBMS indexes for efficient multi-attribute querying
  - Bitmaps may use less storage, thanks to compression
  - Bitmap encoding can reduce storage, speed up query
  - Binning allows bitmap indexing on high-cardinality data

+ Fast equality and range queries + multivariate queries
+ Low storage space
– Frequent inserts/deletes are difficult
– High cardinality data needs binning, with its trade-offs

Overview

- “Traditional” DBMS indexes
- Bitmap indexes
- **Inverted indexes**

Oak Ridge National Laboratory

NC STATE UNIVERSITY Department of Computer Science

Inverted Indexes

- Most commonly used in text/document search engines

- Maps words to containing documents via a list of RIDs
  - This model differs a bit from attributes/records

- Supports "full text" search queries: which documents contain certain words?

Inverted Index

be	: 1, 2
not	: 1
or	: 1
own	: 2
self	: 2
thine	: 2
true	: 2
to	: 1, 2

Document 1: "to be or not to be"
Document 2: "to thine own self be true"

Querying an Inverted Index

1. Read the RID lists for each word involved in the query
2. Use list intersections/merges for AND/OR between words

Query:
document contains
“be” AND “true”:

be	: 1, 2
not	: 1
or	: 1
own	: 2
self	: 2
thine	: 2
true	: 2
to	: 1, 2

AND: intersect
(1, 2) with (2)

→

Document 2
(“to thine own
self be true”)

Updating an Inverted Index

- Has some similar issues to bitmap indexes
  - RID insert/delete in the middle of a list would shift RIDs
- However, inverted lists needn't be (completely) sorted
  - Inserts can be simply be appended
  - Deleted RIDs may overwritten by moving the last RID
  - Maintaining a partial sort can speed up intersect/merge
- Still, inverted indexes work best for read-mostly data

Inverted Index Compression

- Compression greatly reduces storage for long RID lists

- Most compression methods are "chunk-based"
  - RIDs grouped into chunks, compressed independently

- **Frame-of-Reference (FOR)**
  - Keep lowest RID, subtract it from all RIDs in chunk
  - Can then use fewer bits per RID

103, 105, 106, 109, 111, 137, 139 →
lowest: **103**, relative RIDs: **0, 2, 3, 6, 7, 34, 36** (**6 bits each**)

Inverted Index Compression (cont.)

- **Patched Frame-of-Reference (PFOR)**
  - As FOR, but keep large values separate as "exceptions"
  - If few large exceptions, allows even fewer bits per RID

103, 105, 106, 109, 110, 137, 139 →
lowest: **103**, relative RIDs: **0, 2, 3, 6, 7, *, *** (**3 bits each**), exceptions: **137, 139**

- **Patched Frame-of-Reference Delta (PFOR-Delta)**
  - As PFOR, but sort RIDs and store pairwise deltas
  - Stores large jumps in the list with only one exception each

103, 105, 106, 109, 110, 137, 139 →
lowest: **103**, RID deltas: **2, 1, 3, 1, *, 2** (**2 bits each**), exceptions: **27**

Use Case: ElasticSearch*

- Documents are JSON objects with various fields
  - Running example:

```
{
 content: “The quick brown fox jumped over the lazy dog”
},
{
 content: “Quick brown foxes leap over lazy dogs in summer”
}
```

- Want to allow keyword search within each JSON field
  - Approach: build an inverted index on each field

- However, some challenges exist in practice…

**This example draws from the ElasticSearch Guide, especially Mapping and Analysis ([here]) and Full Text Search ([here])*

Challenge #1: Search Term "Fuzziness"

- Suppose we search for "Quick"
  - Should this match document 1 (below)?
  - Capitalization probably doesn't matter to the user...

- Other (usually) unimportant differences exist
  - Punctuation
  - Misspellings
  - Synonyms

Document 1: `The quick brown fox jumped over the lazy dog`
Document 2: `Quick brown foxes leap over lazy dogs in summer`

Solution: Analysis/Normalization

- ElasticSearch uses "analysis" to normalize text
  - Applied to both documents and search text equally

1. **Character filters**: cleans up raw text
   - Remove uninteresting punctuation, etc.
2. **Tokenizer**: splits text into "terms"
   - Split on whitespace, handle hyphenated words, etc.
3. **Token filters**: normalize each term
   - Example: convert to lowercase
   - Example: remove suffixes (e.g., -ing, -ed)
   - Example: remove stop word terms entirely
   - Example: insert synonym terms

Challenge #2: Relevance vs. Exact Match

- Suppose we search for "quick brown fox in summer"
  - Should this match document 1 (below)?
  - Maybe, though it is less relevant than document 2

- Strict "expression tree" evaluation won't "almost match"

Document 1: `The quick brown fox jumped over the lazy dog`
Document 2: `Quick brown foxes leap over lazy dogs in summer`

Solution: Results Ranking

- ElasticSearch "match" queries return documents containing any query words, sorted by relevance score
  - We can already get the document list: use an OR query
  - Score from matching terms via weighted heuristics:

1. **Term frequency**: how many appearances?
   - More appearances in a field = more weight for matching
2. **Document frequency**: how common is the term?
   - More common across all documents = less weight
3. **Field-length norm**: how long is the matched field?
   - Shorter fields (e.g., "title") = more weight

Other Uses for Inverted Indexes

- Though designed for word-to-document mapping, inverted indexes can still be useful on structured data

- Inverted lists have been substituted for bitmap indexes
  – "Words" = attribute values, "documents" = record IDs

- Inverted indexes can use less storage than bitmaps
  – Especially high-cardinality attributes; similar to "words"

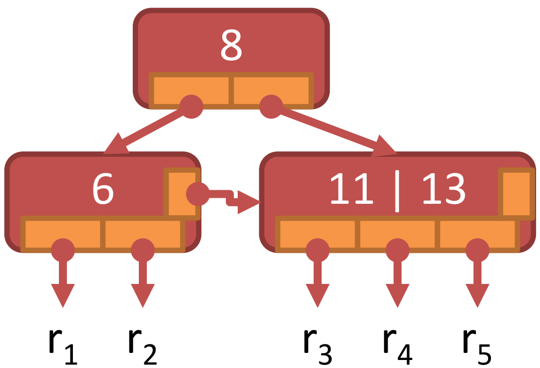
- Bitmaps are often faster at ANDs/ORs, though

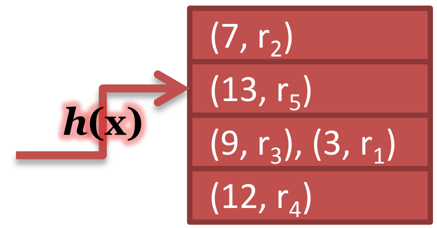
Summary: Inverted Indexes

- Most commonly used to support full text search
- Compression is often used to reduce storage footprint
- Search needs practical considerations beyond indexing

+ Fast equality queries
+ Very low storage space for text/high-cardinality data
– Frequent inserts/deletes are difficult
– Multivariate query possible, costlier than with bitmaps

Thank You!





Questions?

Attrib. value	Bitmap
X	1010
Y	0100
Z	0001

be	: 1, 2
not	: 1
or	: 1
own	: 2
self	: 2
thine	: 2
true	: 2
to	: 1, 2

References

- Much of this presentation's content is covered in:

  "Database Systems: The Complete Book" by Garcia-Molina, Ullman, and Widom

- Classic B-tree paper:

  Douglas Comer. "Ubiquitous B-Tree." *ACM Comput. Surv.* Vol. 11. No. 2, 1979.

- Bitmap indexing:

  Wu, Kesheng. "FastBit: an efficient indexing technology for accelerating data-intensive science." *Journal of Physics: Conference Series*. Vol. 16. No. 1, 2005.

  Wu, Kesheng, Ekow J. Otoo, and Arie Shoshani. "Optimizing bitmap indices with efficient compression" *ACM Trans. Database Syst*. Vol. 31, No. 1, 2006.

  Chee-Yong Chan and Yannis E. Ioannidis. "An efficient bitmap encoding scheme for selection queries." In *Proc. ACM SIGMOD,* 1999.

Oak Ridge National Laboratory

NC STATE UNIVERSITY Department of Computer Science

References (cont.)

- Inverted indexing:

  Zobel, Justin, and Alistair Moffat. “Inverted files for text search engines.” *ACM Computing Surveys (CSUR)*. Vol. 38, No. 2, 2006.

  Zukowski, Marcin, et al. “Super-scalar RAM-CPU cache compression.” *Data Engineering, 2006. ICDE'06. Proceedings of the 22nd International Conference on*. IEEE, 2006.

  Jenkins, John et al. “ALACRITY: Analytics-driven lossless data compression for rapid in-situ indexing, storing, and querying.” *Transactions on Large-Scale Data-and Knowledge-Centered Systems X*, 2013.

- ElasticSearch-related content can be found at:

  http://www.elastic.co/guide/en/elasticsearch/guide