

Amsterdam, The Netherlands

4 – 6 November 2026



Index Internals – Part 1 (of 3)

more research from the parmington foundation

Gus Björklund,
Head Groundskeeper,
The Parmington Foundation

please ask questions
if we do not explain something clearly

the only stupid question is
the one you do not ask

"In theory there is no difference between theory and practice. But in practice there is."

Jan L. A. van de Snepscheut

Abstract

This is part 1 of 3 a part series of talks about Indexes in the OpenEdge RDBMS.

Part 1 (this one) is (mostly) Theory. It covers the internals of the index structures, compression, etc. as well as on-disk representation.

Part 2 is (mostly) Practice. It covers how 4GL applications can use indexes. Mike Furgal is performing this talk.

Part 3 is (mostly) Practice. It covers index monitoring and maintenance topics. Rob Fitzpatrick is performing this talk.

Subjects

- Why We Need Indexes
- Basics: What Are Indexes
- Index Structures
- Index Types
- Index Compression
- Basic Index Usage

Why We Need Indexes – Life Without Indexes:

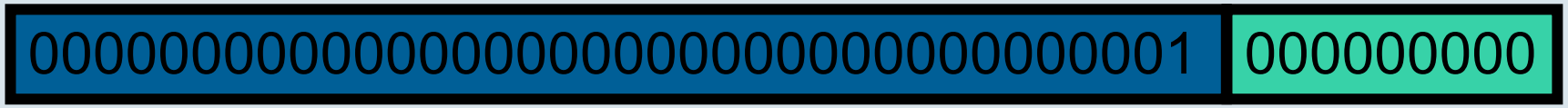
- To find a specific record of a table in a Type 1 data area
  - Start at the first block of the database. Examine the contents.
  - If the record is not there, the move on to the next block.
  - You might have to search for a long time.

- To find a specific record in a type 2 data area
  - Start at the first block of the first cluster of the table. Examine the contents.
  - If the record is not there, the move on to the next block of the table
  - You might have to search a long time

What are Indexes

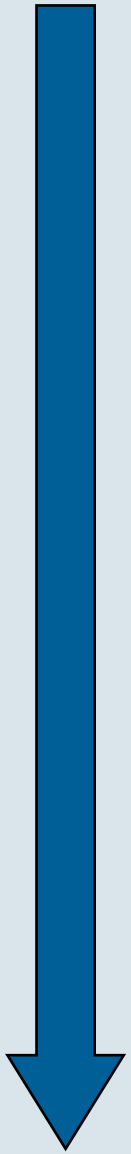
- Indexes are lists of records and their key values
- Indexes are used for
  - Quick access to a row or set of rows by key value
  - To retrieve rows in a specific order
  - To enforce uniqueness of column values in a table
  - To locate rows that contain a word or phrase

Records Have A Rowid



- 64-bit identifier for a specific row in a table
  - Contains Partition Id, Block number within an area, row number within a block
  - Unique within an area or partition
- Encodes the “physiological” storage addresss
  - Used to calculate location of a record
- Same as a RECID for OpenEdge databases
- Constant for the “life” of the record
  - Until you delete it or change it’s partition key value
- Rowid established by the storage location assigned at creation

Index – an Ordered List of Rowids



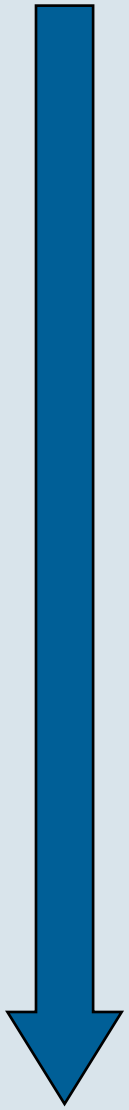
City	Rowid
BOLONIA	3331
BOLTON	5554
BOLTON	9001
BOLTON	9022
BONN	8001
BOSTON	1111
BOSTON	1118
BOSTON	7001
BOSTON	9002
BOSTON	9003
BOSTON	9006
BOSTON	9007
BOSTON	9008
BOSTON	9009
BOSTON	9999
CARDIFF	3333

What if the ordered list of records is too LONG?

1. Subdivide the long list into multiple shorter lists

2. Make an ordered list of the shorter lists

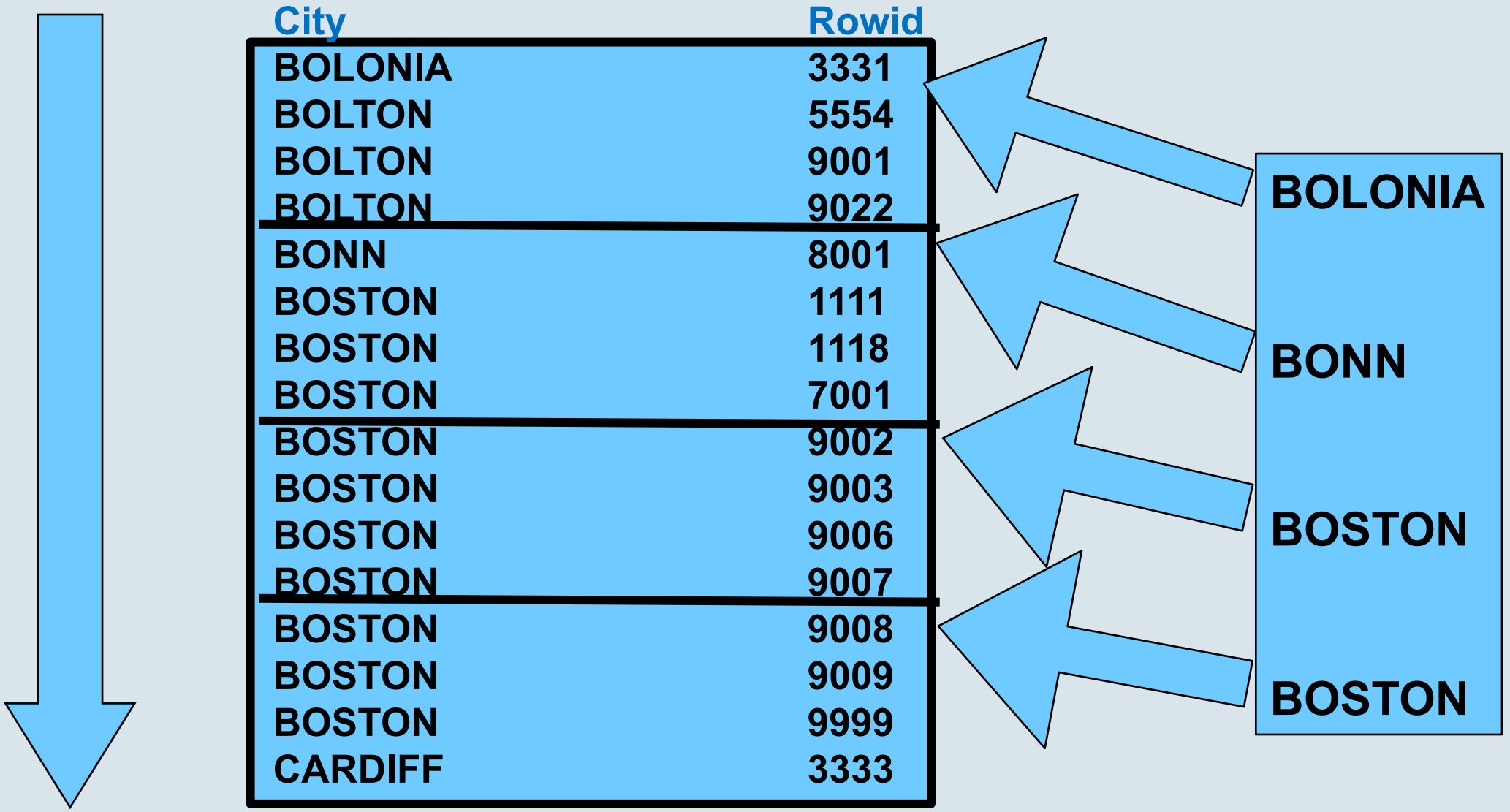
Subdivided List



City	Rowid	
BOLONIA	3331	List 1
BOLTON	5554	
BOLTON	9001	
BOLTON	9022	
BONN	8001	List 2
BOSTON	1111	
BOSTON	1118	
BOSTON	7001	
BOSTON	9002	List 3
BOSTON	9003	
BOSTON	9006	
BOSTON	9007	
BOSTON	9008	List 4
BOSTON	9009	
BOSTON	9999	
CARDIFF	3333	

4 lists

A List of Lists



What if the ordered list of shorter lists is TOO LONG?

1. Subdivide the list of shorter lists into multiple lists of shorter lists

2. Make a another list of lists of shorter lists

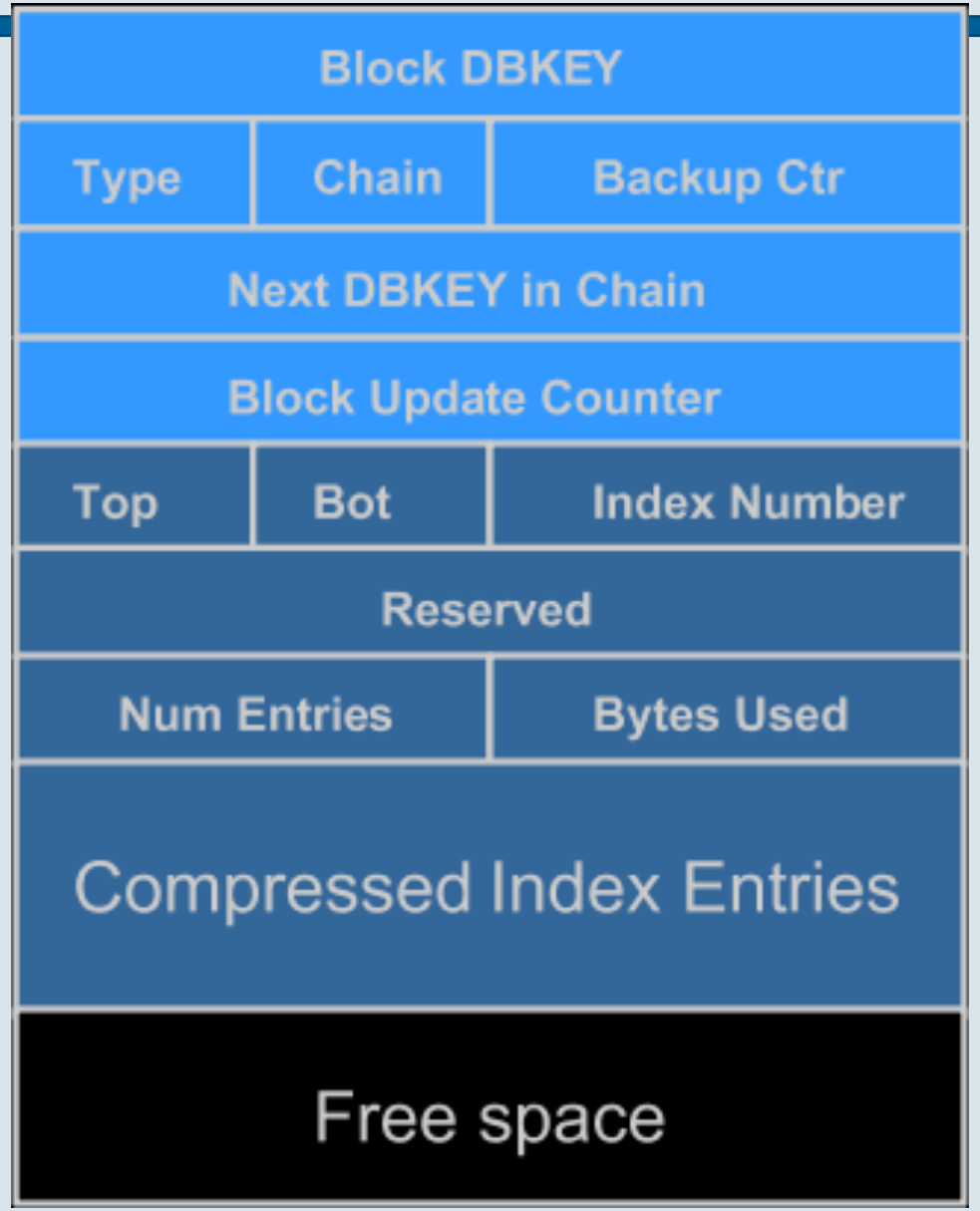
Index Structures

A look at how these smaller lists and

lists of smaller lists and

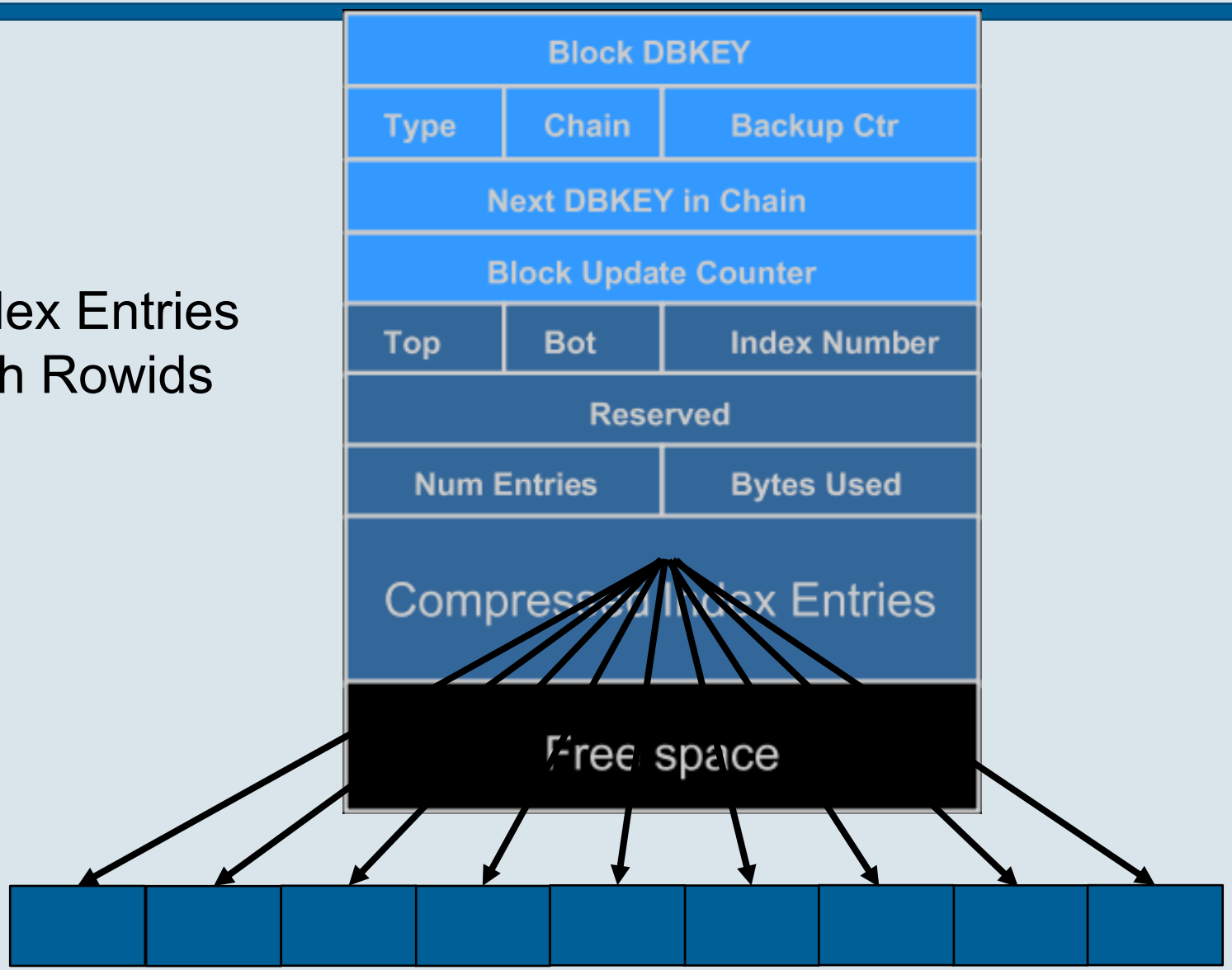
lists of lists of lists of smaller lists are stored and managed.

Each list lives in an “index block”



Index Leaf Block

Index Entries
with Rowids



records

Index Entry

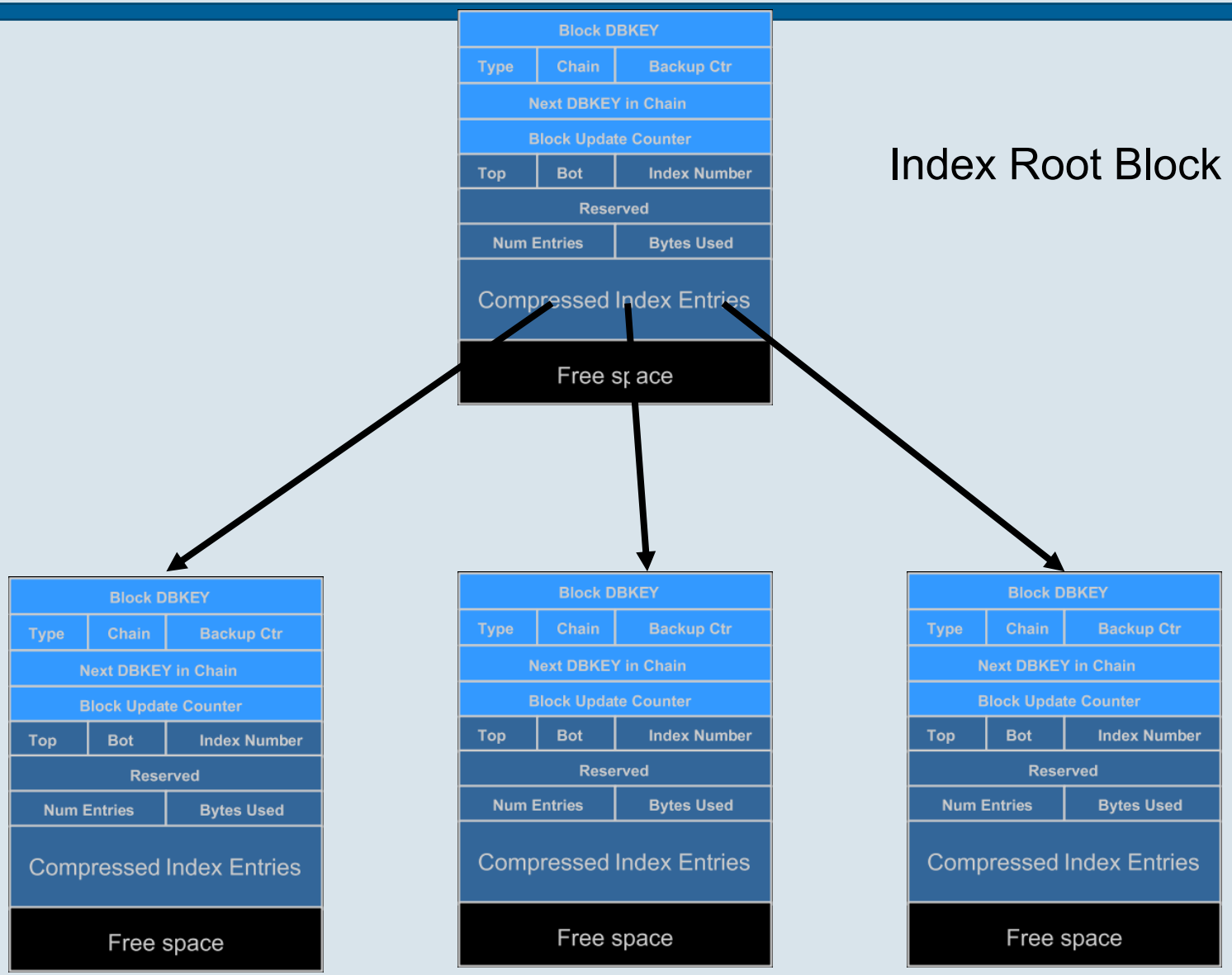


- Two parts: Key Value and Info
  - Key Value: byte array, up to ~3000 bytes
  - Info: byte array, up to 255 bytes
- Entries ordered by key value (one or more column values)
  - Numeric Value (integer, int64, decimal) or
  - Date or
  - Datetime or
  - Collating order of characters NOT the assigned characters
- Compared left to right, byte by byte

B-Trees

- All Indexes are organised as “compressed B-Trees”
- Designed for high concurrency
  - Multi-threaded access
  - Locking minimum blocks, instead of the entire B-Tree
- The index is compressed
  - Four different ways

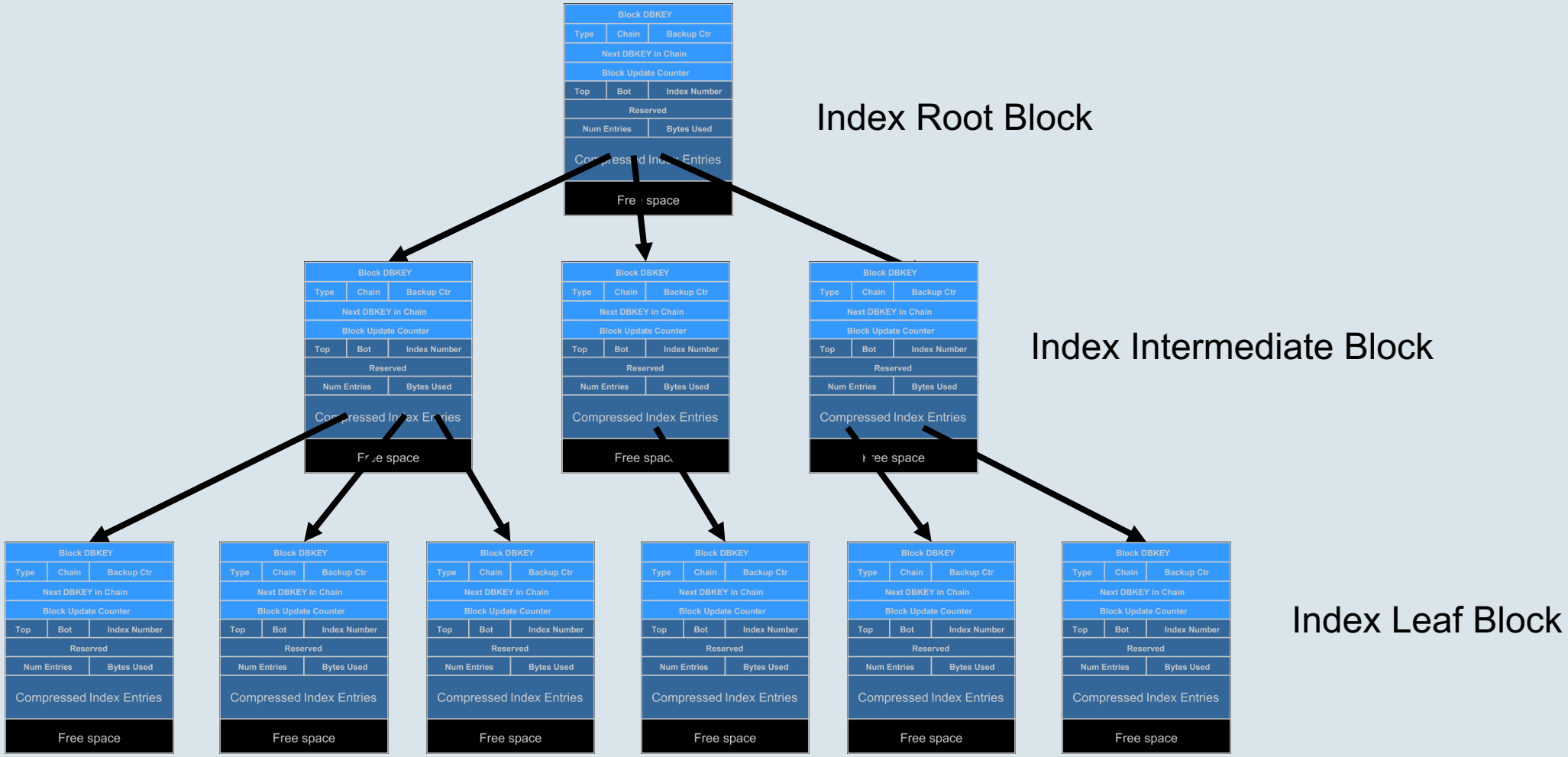
Index B-Tree (2 levels)



Index Root Block

Index Leaf Blocks

Index B-Tree (3 levels)



Multiple Indexes on A Table

- Each index (except word indexes) contains
  - All the rows of one table
  - Same number of entries
  - Same number of rows at the leaf level
- Each index
  - Is a different B-Tree
  - Gives different ordering
  - Is a different size
  - Requires a different number of disk accesses to traverse from root to leaf

Index Types

- The Default Index
- Unique or Non-Unique
- Single or Multi Component Index
- Abbreviated Index
- Word Index
- “no index” queries
- Primary Index

The Default Index

- THIS IS AN OBSOLTE FEATURE. Do not use.
- If you don’t specify at least one actual index you get a default
- The entry consists of just a ROWID, no key value
- Used with FOR EACH, FIND, etc.
- NOT useful except when reading entire table and order does not matter
- Performance likely to be poor

Unique vs. Non-Unique

- Not much difference
  - Unique only allows one entry per key value
    - Key value includes all key components
  - FIND is a bit more efficient than other query types
  - In unique indexes, deleted entries must be reserved (i.e locked) until the deleting transaction ends.
    (Rob will talk more about this in part 3)

Multi-Component Index

State	City	Department	Rowid
AZ,	Flagstaff,	Marketing,	3341
AZ,	Flagstaff,	Marketing,	9800
AZ,	Flagstaff,	Marketing,	9806
AZ,	Flagstaff,	Marketing,	9876
AZ,	Flagstaff,	Sales,	7776
AZ,	Phoenix,	Marketing,	7777
AZ,	Phoenix,	Sales,	0076
AZ,	Phoenix,	Sales,	0888
AZ,	Phoenix,	Sales,	3876
AZ,	Phoenix,	Shipping,	0006
AZ,	Tucson,	Marketing,	0345
AZ,	Tucson,	Shipping,	0347
AZ,	Tucson,	Shipping,	8877
AZ,	Tucson,	Shipping,	8888
CA,	San Diego,	Admin,	0066
CA,	San Diego,	Marketing,	1198

Multi-Component Keys

- Key value composed from more than 1 field (column)
  - State, City, Department
  - Last Name, First Name
  - Country, State, City, Zipcode
- Field values are “logically concatenated”
  - All component values needed to find a specific entry
- Ordered by concatenated value

Abbreviated Indexes

- A little known, poorly documented OBSOLETE feature
- Lookups made by using only the first few characters of the last component of key value, which must be of type character.

  FIND customer where name = ”abc”.
- Using the BEGINS operator with indexes that are not abbreviated has the same effect.

  FIND customer where name BEGINS “abc”.

Word Indexes

- Word index
  - Character columns are broken into ”words”.
  - A text field with many words has many index entries, one entry for each word
  - One word per key value rather than the entire field
- Word index is structured as a regular index
  - B-Tree like other indexes
  - Same structure of index entries and keys
- A word index with a field with one word in it looks exactly like a regular index on that field

Word Indexes: Queries

- … field CONTAINS “eye exam* | (visual test)”
- This is a multi-bracket query
  - 1) all entries where key = “eye” AND
  - 2) all entries where key BEGINS “exam” OR
  - 3) all entries where key = “visual” AND
  - 4) all entries where key = “test”.
- Result is all records that satisfy
  (1 AND 2) OR (3 AND 4)

“no index” queries

- Not an index at all
- When reading a whole table in a type 2 area,
  we read all the records in the first block, then the second, etc.
- This happens sequentially through the chain of data clusters
- You write code thusly:

```
FOR EACH <table> TABLE-SCAN :
        . . .
END,
```

The Primary Index

- When defining indexes, you specify one index as primary.
- This index is used when
  - There is no WHERE clause in a query, or
  - When none of the available indexes can be used to resolve a query.

Mike will go over the rules for index selection and usage in part 2

Index Compression

Compression Improves

- Performance
  - Reduces the number of key comparisons
  - Reduces the number of B-Tree levels
  - Reduces the number of index blocks
  - Reduces the number of DB Accesses
- Storage Efficiency
  - More entries per block
  - Fewer blocks

4 Types of Compression

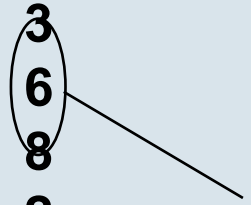
- Delete common prefix in all levels of the B-Tree
- Delete non contributing trailing bytes in the lowest non-leaf level
- Delete first key in all non-leaf blocks
- Use tag lists or bit maps for non-unique keys

Prefix Compression Example

City	Rowid
BOLONIA	3331
BOLTON	5554
BOLTON	9001
BOLTON	9022
BONN	8001
BOSTON	1111
BOSTON	1118
BOSTON	7001
BOSTON	9002
BOSTON	9003
BOSTON	9006
BOSTON	9007
BOSTON	9008
BOSTON	9009
BOSTON	9999
CARDIFF	3333

Prefix Compression Example

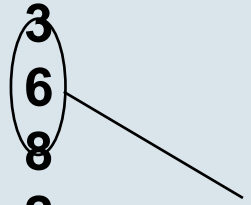
City	Rowid	CS
BOLONIA	3331	0
BOLTON	5554	3
BOLTON	9001	6
BOLTON	9022	8
BONN	8001	2
BOSTON	1111	2
BOSTON	1118	9
BOSTON	7001	6
BOSTON	9002	6
BOSTON	9003	9
BOSTON	9006	9
BOSTON	9007	9
BOSTON	9008	9
BOSTON	9009	9
BOSTON	9999	7
CARDIFF	3333	0



Number of duplicate bytes (discarded)

Prefix Compression Example

City	Rowid	CS
BOLONIA	3331	0
BOLTON	5554	3
BOLTON	9001	6
BOLTON	9022	8
BONN	8001	2
BOSTON	1111	2
BOSTON	1118	9
BOSTON	7001	6
BOSTON	9002	6
BOSTON	9003	9
BOSTON	9006	9
BOSTON	9007	9
BOSTON	9008	9
BOSTON	9009	9
BOSTON	9999	7
CARDIFF	3333	0



Number of duplicate bytes (discarded)

93 of 160 bytes removed !!!
(58 percent)

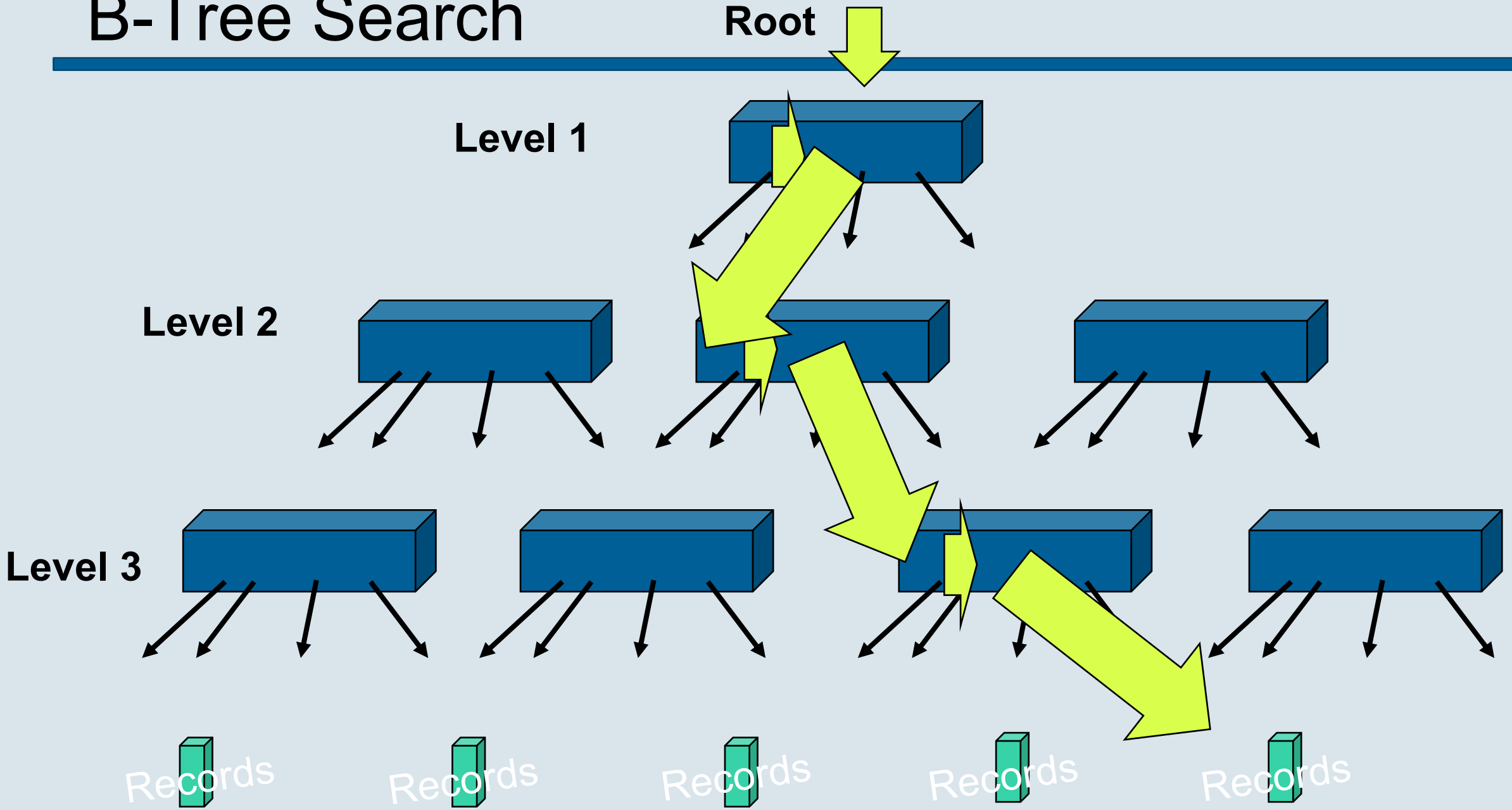
Basic Index Usage

Simple Index Usage:
FIND, Update

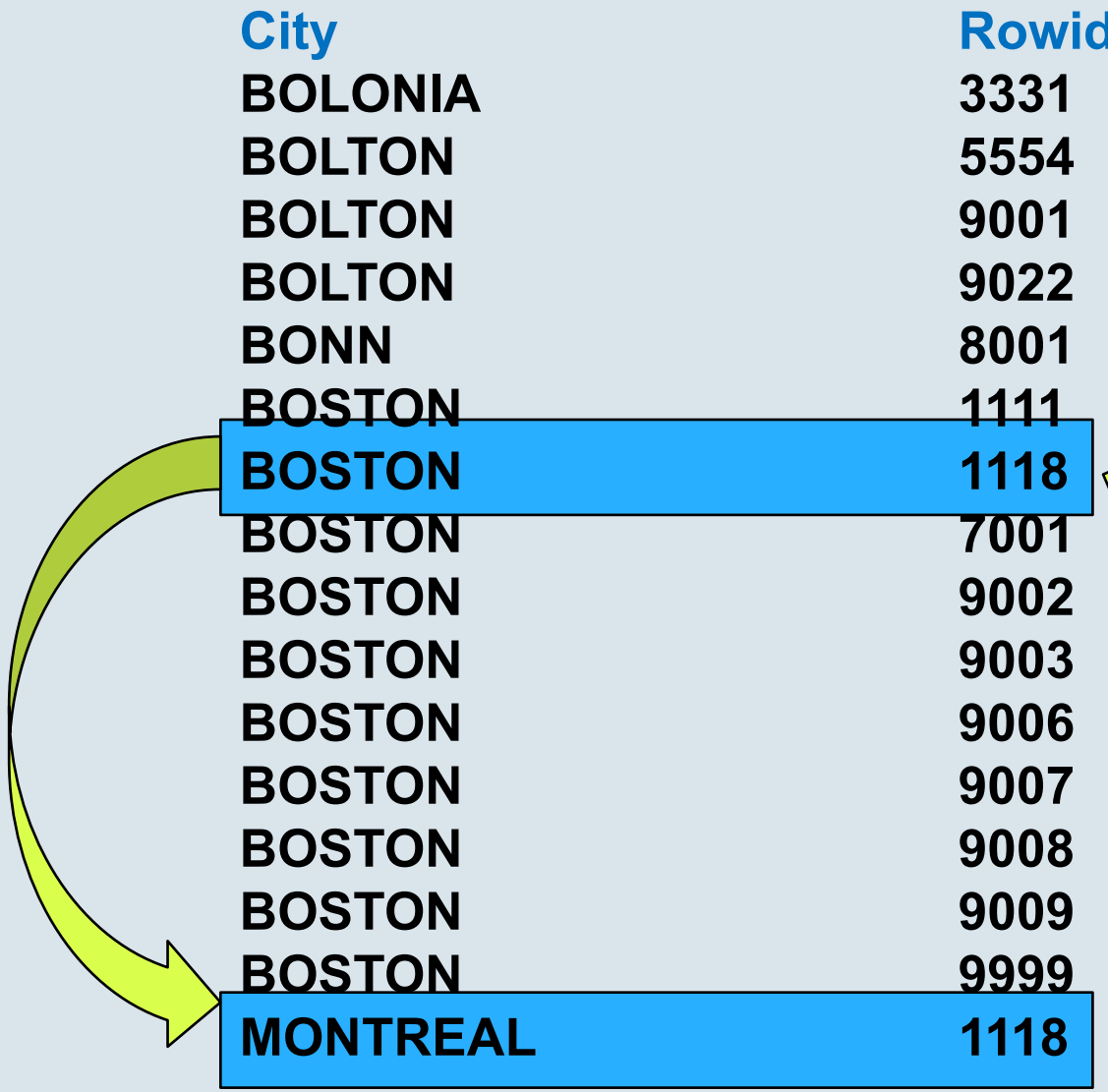
Find a Record

- FIND customer WHERE cust-num = 20.
  - Index “cust-num” is used for the find
    - Compiler determines it is index #113
  - Index 113 will be searched for key value = 20
  - Starting at the “_StorageObject” entry for index 113
    - Find the area (or partition) and root block of the index 113
    - Search the B-Tree of index 113 for key value 20
    - Extract the records ROWID
    - Use the ROWID to fetch the record for cust-num = 20

B-Tree Search



Update a key field



Original Key entry is deleted
New Key entry is created in correct location

If Index is Unique replace Rowid with an “Index Lock”

FIND must return 0 or 1 records

- If index is Unique
  - At most one row with matching can exist
  - If found, then return matching record
- If index is not Unique
  - Check the next index entry for more than one match
    - If yes, then error
    - If no, then return matching record

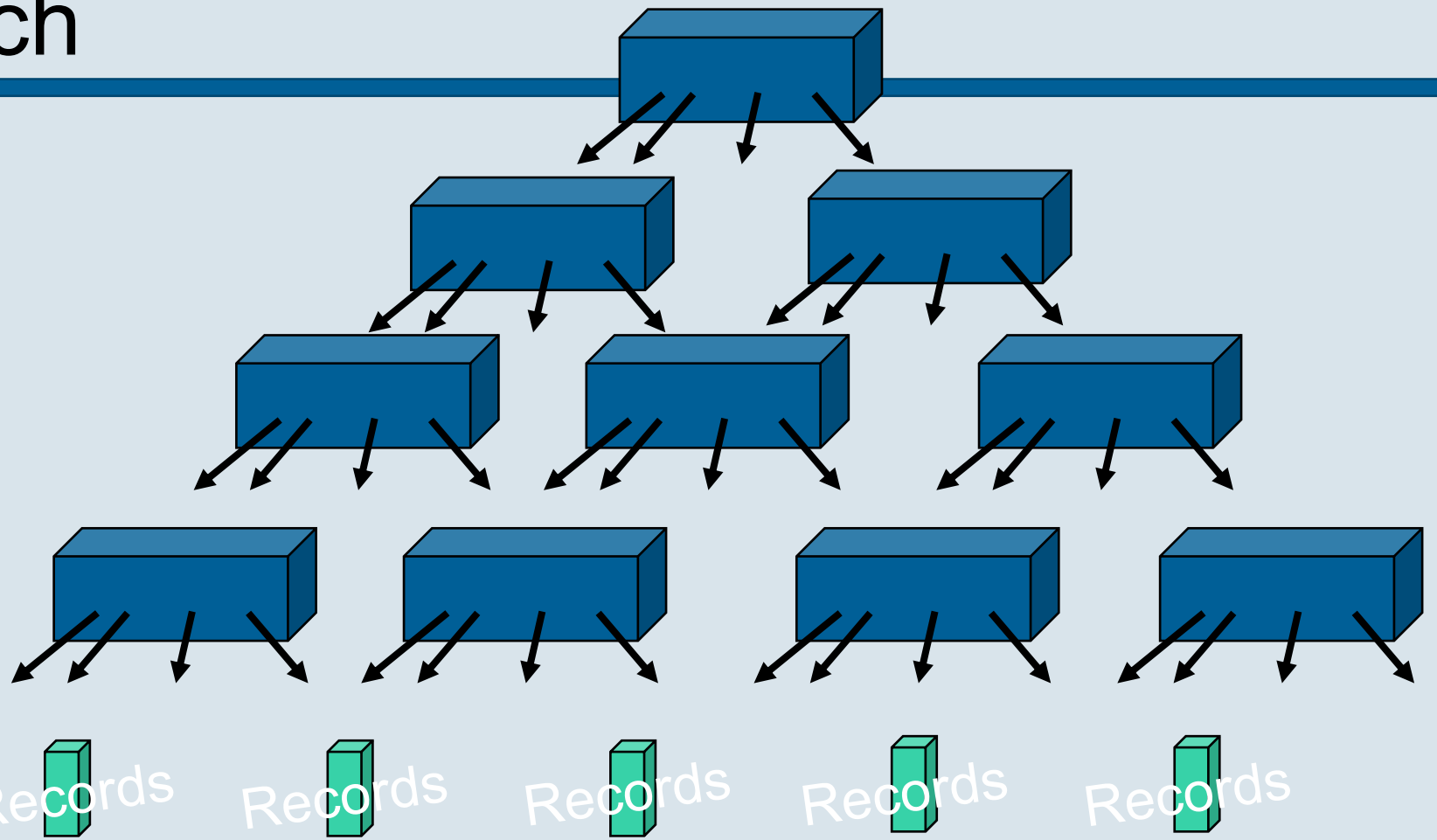
Database Block Size

- The size of the index block is significant
- Larger blocks lead to fewer B-Tree levels
  - Saves Database I/O
  - Better Compression
- Larger blocks add (some) search time per block
  - CPU time is used to search
  - Search costs less than Database I/O

B-Tree Search

1,000,000 Records
50 names per block

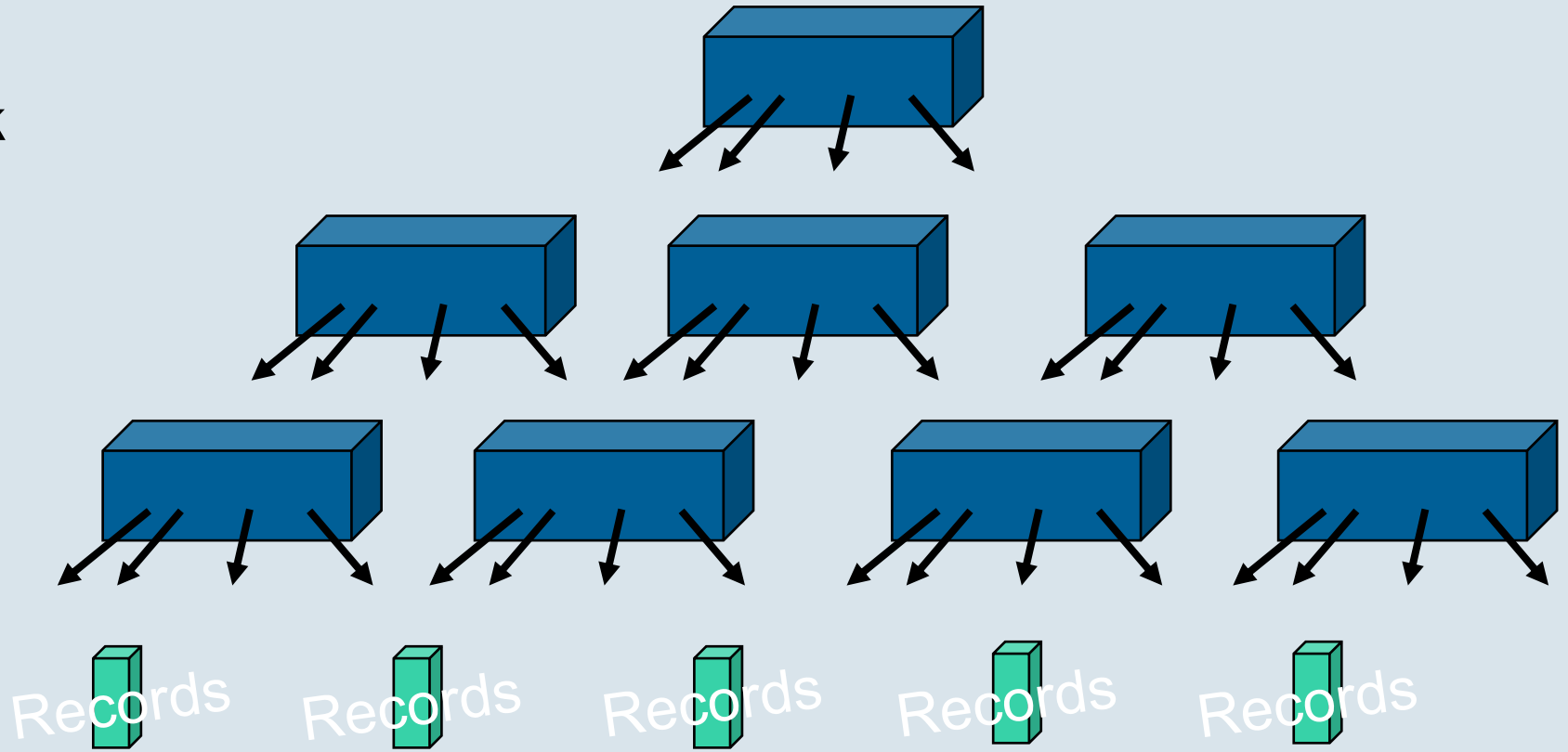
20,000 Leaf Blocks



1,000,000 Records needs 4 levels
3 levels: 50 x 50 x 50 = 125,000 records
4 levels: 50 x 50 x 50 x 50 = 6,250,000 records

B-Tree Search

1,000,000 Records
100 names per block



1,000,000 Records needs 3 levels
3 levels: 100 x 100 x 100 = 1,000,000 records
4 levels: 100 x 100 x 100 x 100 = 100,000,000 records

Index Utilization

- Index Analysis Report

```
INDEX BLOCK SUMMARY FOR AREA "SysActivityLog_Index_64" : 168
-------------------------------------------------------------
Table                          Index  Fields Levels  Blocks    Size   % Util Factor
PUB.SysActivityLog
  ByCompType                   12189       4      3   80122  359.8M     57.7    1.8
  ByDateTimeUserType           12190       4      3  128763  947.1M     94.5    1.1
  ByTableCompType              12191       5      3   81843  360.6M     56.6    1.9
  ByTableDateTime              12192       4      3   86420  380.7M     56.6    1.9
  ByTypeComp                   12193       4      3   66790  359.4M     69.1    1.6
  ByUserCompDateTime           12194       5      3  127966  705.6M     70.8    1.6
  SysActivityLogPrim           12188       1      3   58011  447.8M     99.2    1.0
  SysIndex                     12195       1      4  146218  787.4M     69.2    1.6
```

3 Level index about the same size with fewer blocks

4 Level Index

Average % full

Index Compaction

- Index Analysis Report

```
INDEX BLOCK SUMMARY FOR AREA "SysActivityLog_Index_64" : 168
-------------------------------------------------------------
Table                          Index  Fields Levels  Blocks    Size   % Util Factor
PUB.SysActivityLog
  ByCompType                   12189       4      3   80122  359.8M     57.7    1.8
  ByDateTimeUserType           12190       4      3  128763  947.1M     94.5    1.1
  ByTableCompType              12191       5      3   81843  360.6M     56.6    1.9
  ByTableDateTime              12192       4      3   86420  380.7M     56.6    1.9
  ByTypeComp                   12193       4      3   66790  359.4M     69.1    1.6
  ByUserCompDateTime           12194       5      3  127966  705.6M     70.8    1.6
  SysActivityLogPrim           12188       1      3   58011  447.8M     99.2    1.0
  SysIndex                     12195       1      4  146218  787.4M     69.2    1.6
```

Index Compaction can be run online and will likely reduce the number of levels from 4 to 3
Rob will discuss index analysis in part 3

Benchmarks show that a FIND on a
4 level index is 20% slower than a
FIND on a 3 level index

Database Block Size Matters!!

Index Utilization Matters!!

Digresssion:
Disk order versus index order.
A Hypothetical Example.

(Hypothetical means “made up”)

Assumptions

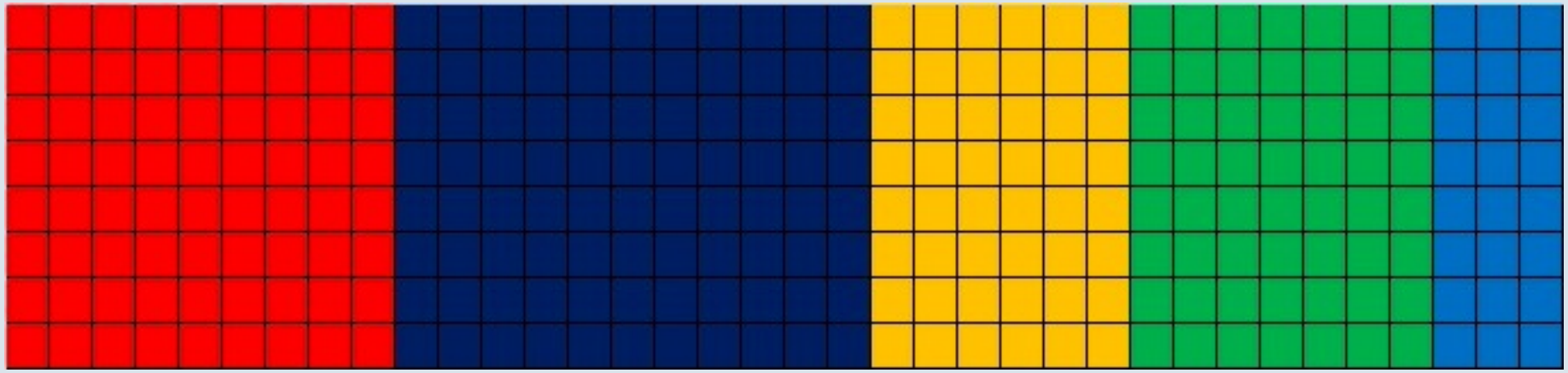
- A table has 100,000,000 rows, one index
- Row size is 250 bytes
- Index entries are 35 bytes
- Database block size is 8192 bytes
- We can fit about 32 rows per data block
- We can fit about 200 entries per index block assuming no compression.

More Assumptions

- 100,000,000 rows, 32 per block
  requires 3,125,000 data blocks
- 100,000,000 index entries, 200 per index block
  requires 500,000 index leaf blocks
- 500,000 leaf blocks might fit 500 entries in second index level, requires 1,000 second level index blocks
- 1,000 second level blocks might require 2 entries in third index level at 500 entries per index block.
- We have 501,002 index blocks for 3,125,000 data blocks

Scatter

Data blocks stored all together near each other

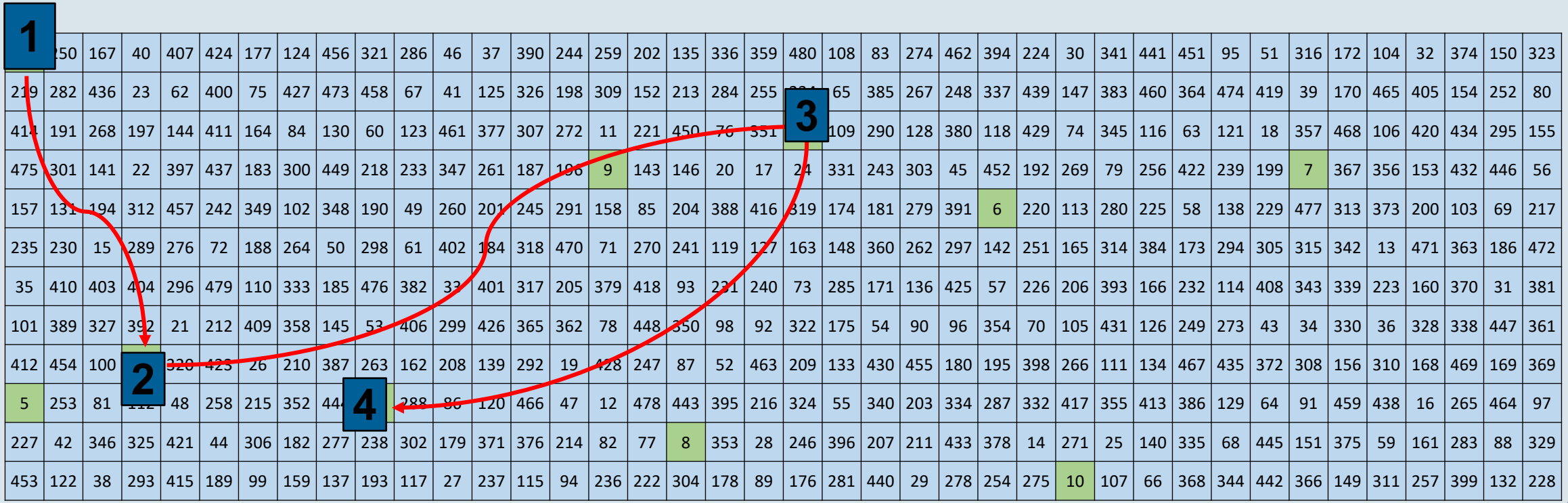


good

each color represent a different table

Internal Scatter

- How ordered are the records by some index?
- Each index gives a different logical ordering
- Are rowids in the same order as the keys?
  (there can be only one physical ordering)



Summary

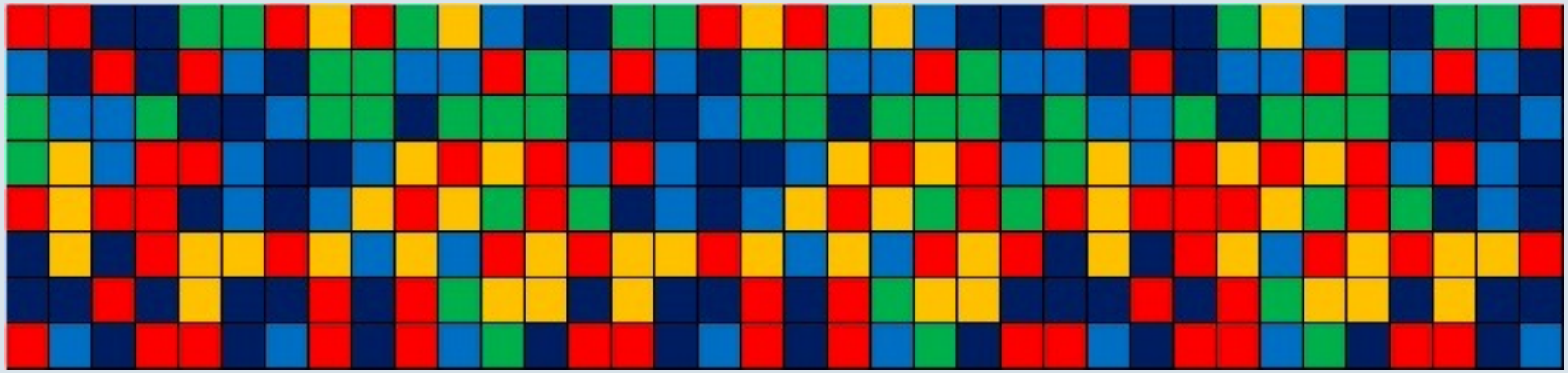
- Table has 3,125,000 data blocks
- Index has 501,002 index blocks

Now let's do a FOR EACH

- Assume index blocks are in same order as data blocks on disk, index blocks all together, data blocks all together.
- We have to read all 501,002 index blocks once each.
- We have to read all 3,125,000 data blocks once each.
- If sequential disk reads take 1 millisecond
  FOR EACH will take 3,626,002 milliseconds – a little over 1 hour.

External Scatter: Table rows scattered all over database

Data blocks stored all over an area in different places



not good

each color represent a different table

Now let's assume rows are NOT in index order

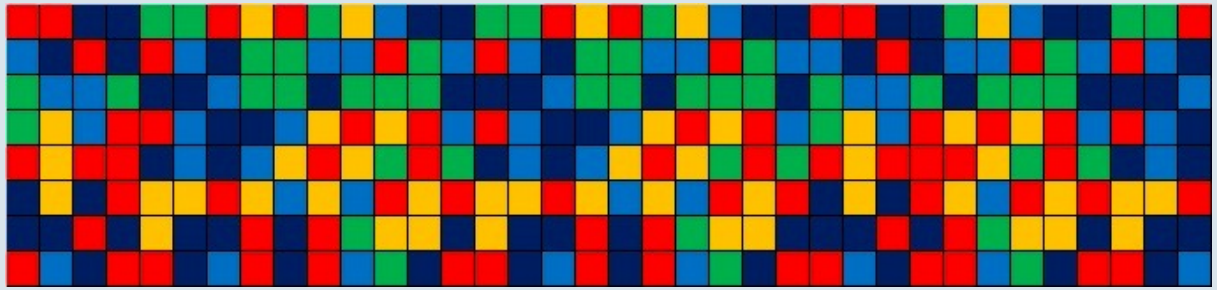
- Random disk reads might take 3 milliseconds.
- Again we read each index block once with 3 needed in memory at once.
- We have space for 10,000 data blocks in memory. That means there is a 1 in 10,000 chance that a block we need to access next is in memory.

Continued . . .

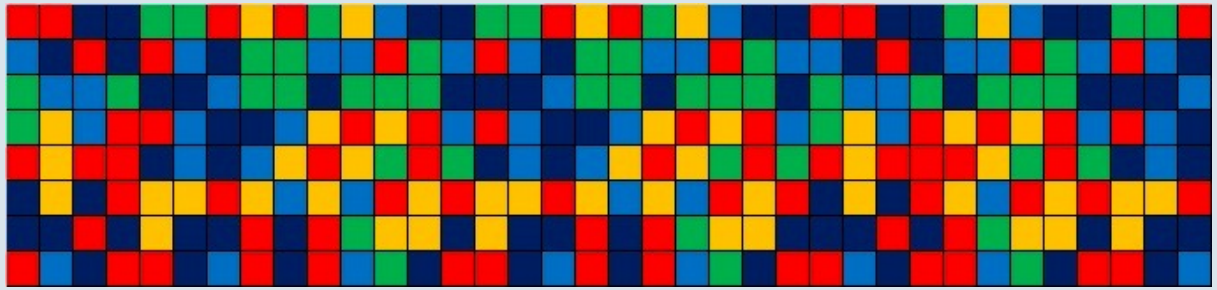
rows are NOT in index order part 2

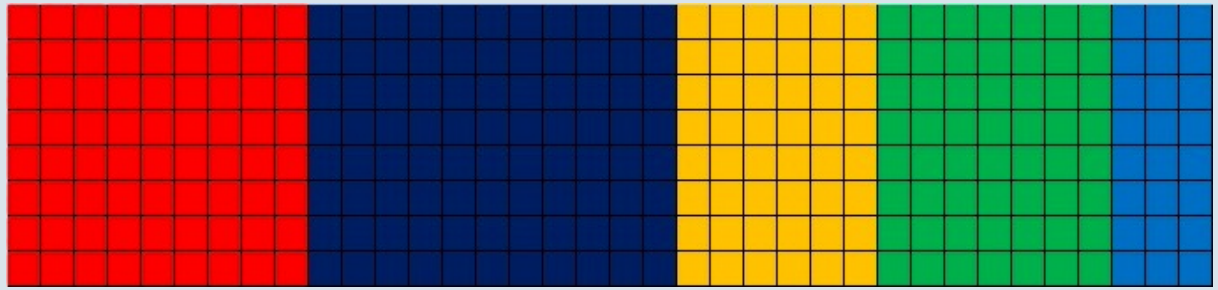
- Row are not in order
- Reading first row takes 3 milliseconds
- Second row is not in same block and not in memory
- Reading second row takes 3 milliseconds
- Etc.
- Worst case: each row requires a read of a data block
- Total time 100,501,002 x 3 milliseconds
  301,503,006 milliseconds – about 84 hours

External Scatter: To go from this mess

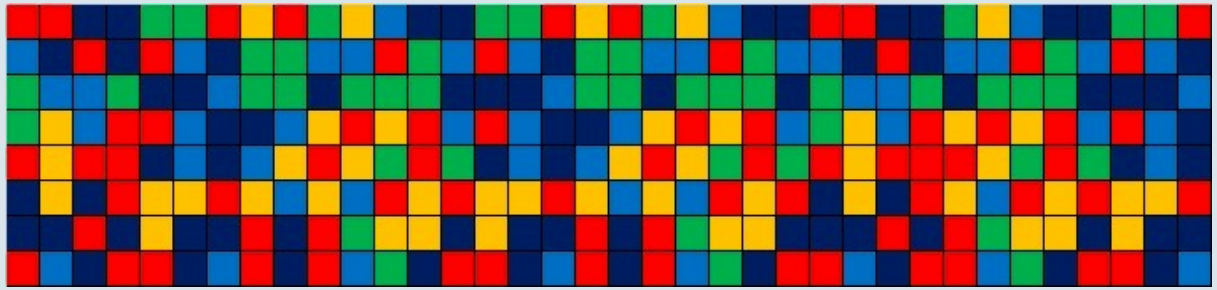


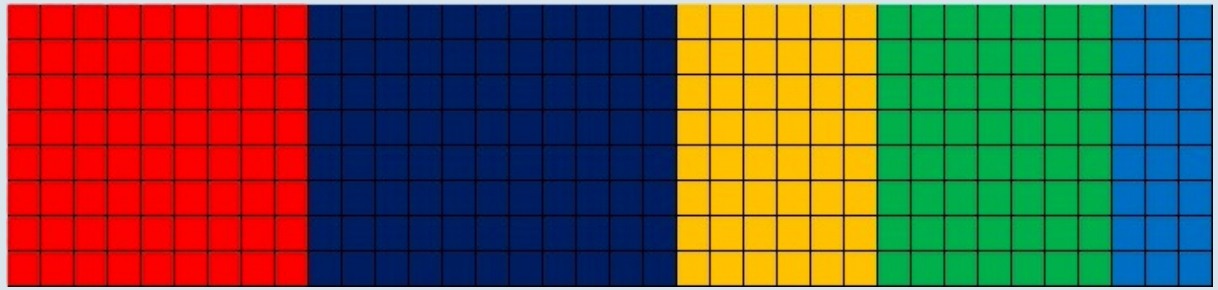
External Scatter: To go from this mess to that





External Scatter: To go from this mess to that





You can use the “tablereorg” tool. It can be run online or offline.

Choose one index for one table and tablereorg makes the index order and storage order match and makes storage order contiguous for that table. ROWIDS will change.

End of digresssion

More complex Index Usage
Index Brackets

Index Brackets

- A set of consecutive entries in an index
- Defined by a range of key values
  - Cust-num = 20
  - Age >= 21
  - Age >= 21 AND age <= 50
  - Name BEGINS “John”
- Determined by
  - Index number
  - Low Key value
  - High Key value

2 Bracket Types

- Equality Bracket
  - All key values are equal (including all the components)
  - Index entries are consecutive
- Range Bracket
  - All key values within a specific range
    - Between low key and high key
  - Index entries are consecutive

Equality Brackets

State	City	Department	Rowid
AZ,	Flagstaff,	Marketing,	3341
AZ,	Flagstaff,	Marketing,	9800
AZ,	Flagstaff,	Marketing,	9806
AZ,	Flagstaff,	Marketing,	9876
AZ,	Flagstaff,	Sales,	7776
AZ,	Phoenix,	Marketing,	7777
AZ,	Phoenix,	Sales,	0076
AZ,	Phoenix,	Sales,	0888
AZ,	Phoenix,	Sales,	3876
AZ,	Phoenix,	Shipping,	0006
AZ,	Tucson,	Marketing,	0345
AZ,	Tucson,	Shipping,	0347
AZ,	Tucson,	Shipping,	8877
AZ,	Tucson,	Shipping,	8888
CA,	San Diego,	Admin,	0066
CA,	San Diego,	Marketing,	1198

Entire key same

Entire key same

Equality Brackets

State	City	Department	Rowid
AZ,	Flagstaff,	Marketing,	3341
AZ,	Flagstaff,	Marketing,	9800
AZ,	Flagstaff,	Marketing,	9806
AZ,	Flagstaff,	Marketing,	9876
AZ,	Flagstaff,	Sales,	7776
AZ,	Phoenix,	Marketing,	7777
AZ,	Phoenix,	Sales,	0076
AZ,	Phoenix,	Sales,	0888
AZ,	Phoenix,	Sales,	3876
AZ,	Phoenix,	Shipping,	0006
AZ,	Tucson,	Marketing,	0345
AZ,	Tucson,	Shipping,	0347
AZ,	Tucson,	Shipping,	8877
AZ,	Tucson,	Shipping,	8888
CA,	San Diego,	Admin,	0066
CA,	San Diego,	Marketing,	1198

State = “AZ”

First key component same

Equality Brackets

State	City	Department	Rowid
AZ,	Flagstaff,	Marketing,	3341
AZ,	Flagstaff,	Marketing,	9800
AZ,	Flagstaff,	Marketing,	9806
AZ,	Flagstaff,	Marketing,	9876
AZ,	Flagstaff,	Sales,	7776
AZ,	Phoenix,	Marketing,	7777
AZ,	Phoenix,	Sales,	0076
AZ,	Phoenix,	Sales,	0888
AZ,	Phoenix,	Sales,	3876
AZ,	Phoenix,	Shipping,	0006
AZ,	Tucson,	Marketing,	0345
AZ,	Tucson,	Shipping,	0347
AZ,	Tucson,	Shipping,	8877
AZ,	Tucson,	Shipping,	8888
CA,	San Diego,	Admin,	0066
CA,	San Diego,	Marketing,	1198

State = “AZ”

First key component same

(State = “AZ”)
AND
(City = “Phoenix”)

First two key components same

RangeBrackets

State	City	Department	Rowid
AZ,	Flagstaff,	Marketing,	3341
AZ,	Flagstaff,	Marketing,	9800
AZ,	Flagstaff,	Marketing,	9806
AZ,	Flagstaff,	Marketing,	9876
AZ,	Flagstaff,	Sales,	7776
AZ,	Phoenix,	Marketing,	7777
AZ,	Phoenix,	Sales,	0076
AZ,	Phoenix,	Sales,	0888
AZ,	Phoenix,	Sales,	3876
AZ,	Phoenix,	Shipping,	0006
AZ,	Tucson,	Marketing,	0345
AZ,	Tucson,	Shipping,	0347
AZ,	Tucson,	Shipping,	8877
AZ,	Tucson,	Shipping,	8888
CA,	San Diego,	Admin,	0066
CA,	San Diego,	Marketing,	1198

State = “AZ”

(State = “AZ”)
AND
(City >= “Phoenix”)

Bracket Quiz

State	City	Department	Rowid
AZ,	Flagstaff,	Marketing,	3341
AZ,	Flagstaff,	Marketing,	9800
AZ,	Flagstaff,	Marketing,	9806
AZ,	Flagstaff,	Marketing,	9876
AZ,	Flagstaff,	Sales,	7776
AZ,	Phoenix,	Marketing,	7777
AZ,	Phoenix,	Sales,	0076
AZ,	Phoenix,	Sales,	0888
AZ,	Phoenix,	Sales,	3876
AZ,	Phoenix,	Shipping,	0006
AZ,	Tucson,	Marketing,	0345
AZ,	Tucson,	Shipping,	0347
AZ,	Tucson,	Shipping,	8877
AZ,	Tucson,	Shipping,	8888
CA,	San Diego,	Admin,	0066
CA,	San Diego,	Marketing,	1198

Is there a bracket for

(State = “AZ”) AND
(Department = “Marketing”)

?

Bracket Quiz

State	City	Department	Rowid
AZ,	Flagstaff,	Marketing,	3341
AZ,	Flagstaff,	Marketing,	9800
AZ,	Flagstaff,	Marketing,	9806
AZ,	Flagstaff,	Marketing,	9876
AZ,	Flagstaff,	Sales,	7776
AZ,	Phoenix,	Marketing,	7777
AZ,	Phoenix,	Sales,	0076
AZ,	Phoenix,	Sales,	0888
AZ,	Phoenix,	Sales,	3876
AZ,	Phoenix,	Shipping,	0006
AZ,	Tucson,	Marketing,	0345
AZ,	Tucson,	Shipping,	0347
AZ,	Tucson,	Shipping,	8877
AZ,	Tucson,	Shipping,	8888
CA,	San Diego,	Admin,	0066
CA,	San Diego,	Marketing,	1198

Is there a bracket for

(State = “AZ”) AND
(Department = “Marketing”)

NO!!
Entries are not consecutive

Index Cursors

Index Cursors

- An internal data structure
  - Allocated as needed on behalf of the connection
- Used for a sequential scan of a bracket
- Used to maintain location in an index bracket
  - Says where we are in current block of each level in index
- One or more are used in a query resolution

Index Cursors

- Has index number, area, etc
- 1 entry per level in the index
  - Block Number
  - Block Version
  - Position in the block

Thus endeth Index Internals Part 1

Part 2: 4GL Index Selection
(mostly) Mike Furgal
10:15 am Thursday in New York 2.

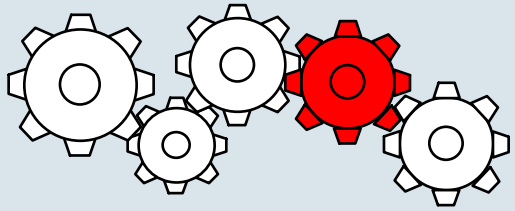
Part 3: Index Monitoring and Maintenance
(mostly) Rob Fitzpatrick
9:00 am Friday in New York 2.

Vragen



email:

gus@parmington.com



research from the parmington foundation