Lecture 02.07

Some more indexes

By Marina Barsky
Winter 2017, University of Toronto

Multi-dimensional indexes

Multi-dimensional data

- Databases often store data in more than 1 dimension

- Examples:
  - Relation – a collection of k-dimensional points. Each attribute is a separate dimension.

    Customer (age, salary, pcode, maritalstatus, etc.)

    Sale (store, day, item, color, size, etc.).

    Each sale = point in 5dim space.

  - GIS – 2-dimensional representation of objects on the map.
  - Image databases – medical imaging, photographs

Multi-dimensional queries

- **Range queries**:

  Relation: Customer (age, salary, pcode, maritalstatus).

  Query: "How many customers for gold jewelry have age between 45 and 55, and salary less than 100K?"

- **Nearest neighbor:**

  GIS – 2-dimensional points representing objects on the map.

  Query: "If I am at coordinates (a,b), what is the nearest McDonalds?"

- **Content-based queries:**

  Image databases – medical imaging, photographs

  Query: find images similar to a given image

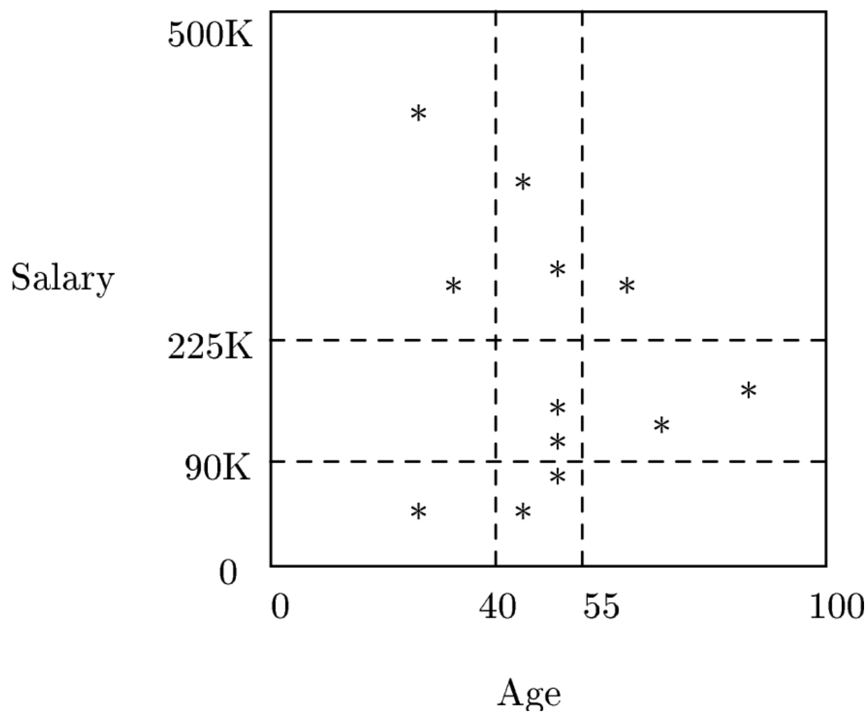
MD range query

Example:

Customers (id, age, salary, spent)

for people *who buy gold jewelry.*

"How many customers for gold jewelry have age between 45 and 55, and salary less than 100K?"



SELECT *

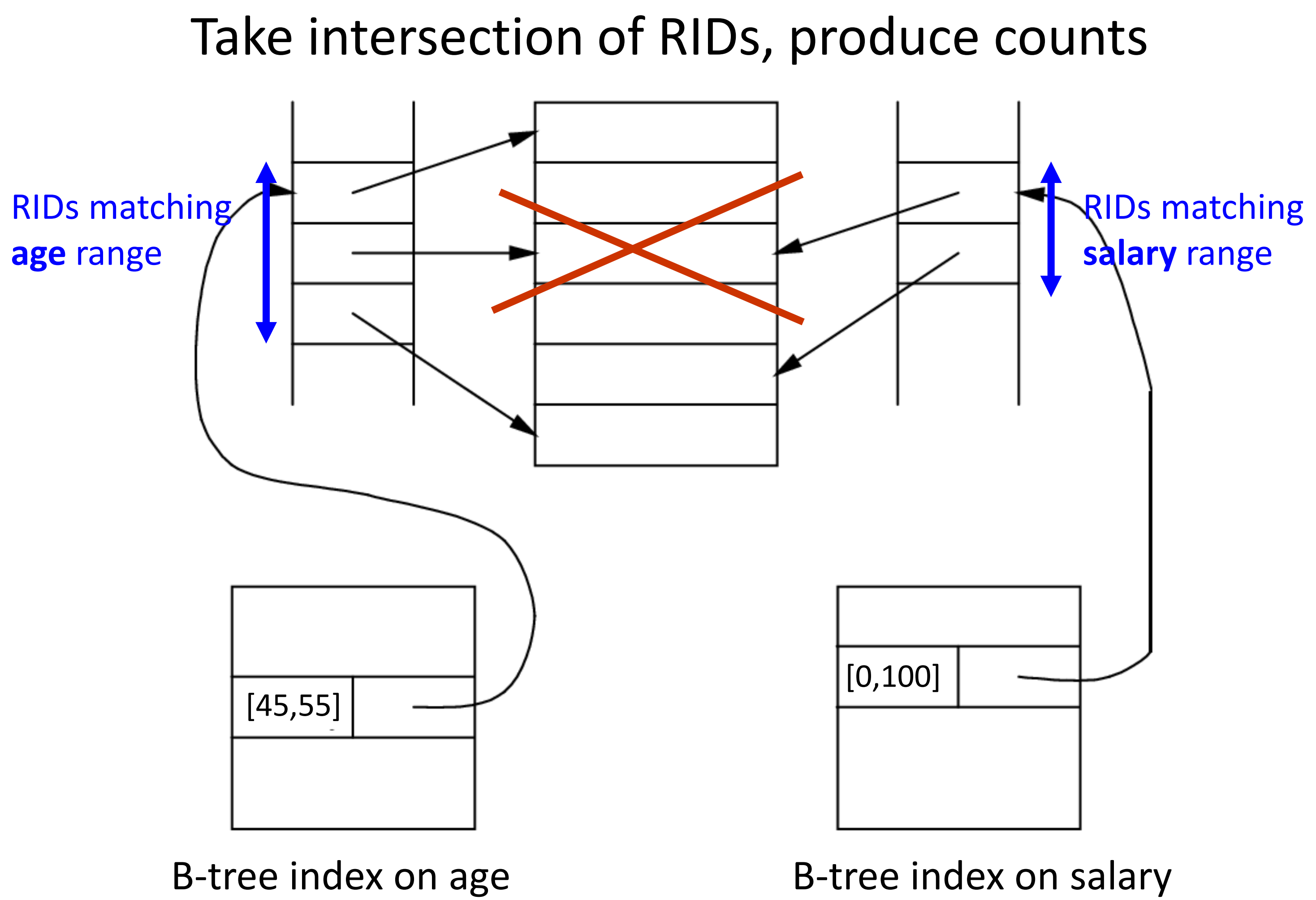
FROM Customers

WHERE age>=45 AND age<=55

AND sal<100;

Data:
(25,60) (45,60) (50,75) (50,100)
(50,120) (70,110) (85,140) (30,260)
(25,400) (45,350) (50,275) (60,260)

Sometimes can use secondary B-trees on Age and Salary



Multi-dimensional indexes

- Hash inspired:
  - Grid files
  - Partitioned hash functions
- Tree-inspired:
  - KD-trees
  - Quad-trees
  - R-trees (Region trees)

Multi-dimensional indexes in external memory

- Adaptations of all these indexes for disk give up at least one of the following:
  - Correspondence between tree nodes and blocks
  - Balance of the tree
  - Complexity of dynamic operations (insertions, deletions)

- We are not going to study them in-depth in this course

Bitmap Indexes

example of multi-dimensional indexing

Bitmap Indexes

- Suppose we have *n* tuples (rows, records)
- A ***bitmap index*** for a field *F* is a collection of bit vectors of length *n*, one for each possible value that may appear in the field *F*
- The vector for value *v* has 1 in position *i* if the *i*-th record has *v* in field *F*, and it has 0 there if not

	Bitmap index for second column
(30, foo)	**foo: 100100**
(30, bar)	**bar: 010010**
(40, baz)	**baz: 001001**
(50, foo)	
(40, bar)	
(30, baz)	

Example

Customer table.
We will index Gender and Rating. Note that this is just a partial list of all the records in the table

Two bit strings for the Gender bitmap

M	F
1	0
1	0
0	1
1	0

Custid	Name	Gender	Rating
112	Joe	M	3
115	Sam	M	5
119	Sue	F	5
112	Wu	M	4

Five bit strings for the Rating bitmap

1	2	3	4	5
0	0	1	0	0
0	0	0	0	1
0	0	0	0	1
0	0	0	1	0

Bitmap operations

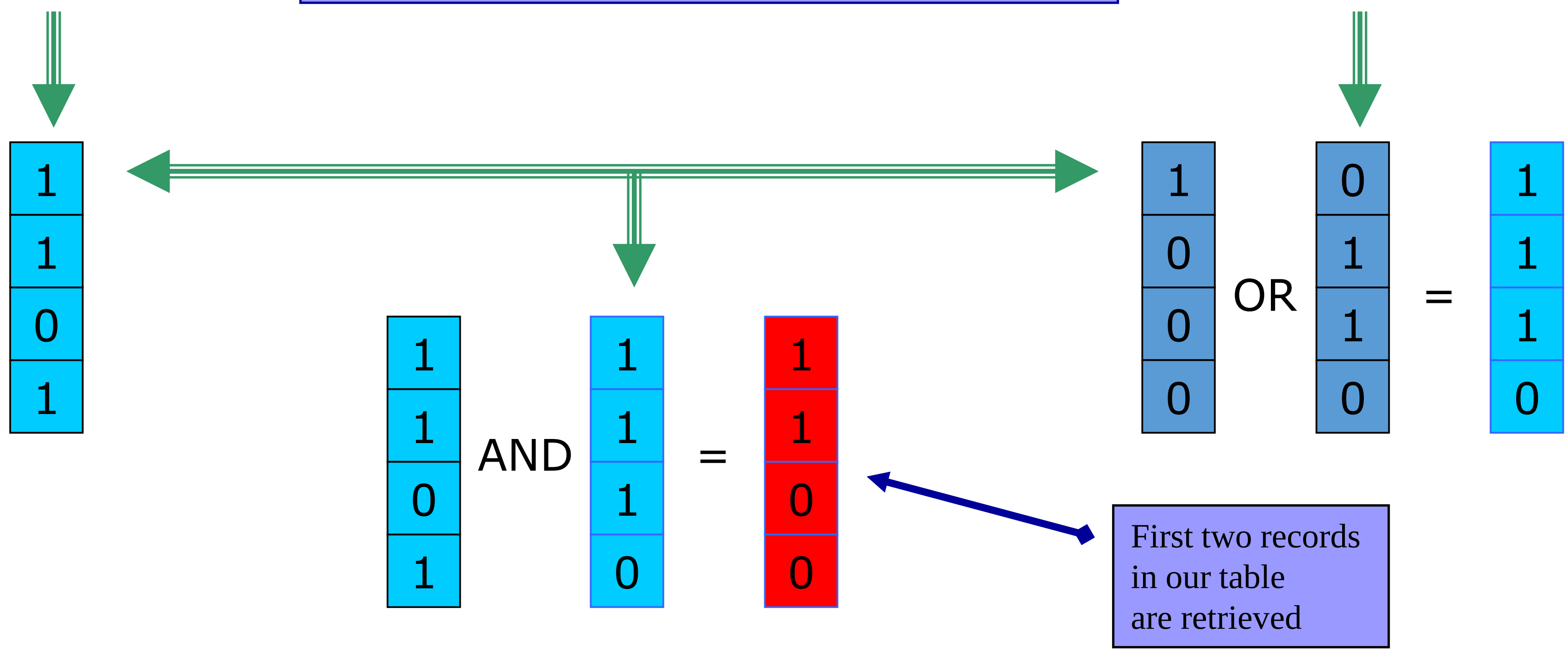
- Bit maps are designed to support partial match and range queries
- To identify the records holding a subset of the values from a given dimension, we can do a binary OR on the bitmaps from that dimension.
  - Example: all customers with high ratings: the ORing of bit strings for *Rating = (3, 4, 5)*
- To identify the partial matches on a group of dimensions, we can simply perform a binary AND on the OR-ed maps from each dimension
- These operations can be done very efficiently since binary operations are natively supported by the CPU

Query example

M	F
1	0
1	0
0	1
1	0

```
SELECT *
 FROM Customer
  WHERE gender = M AND
     (rating = 3 OR rating = 5)
```

1	2	3	4	5
0	0	1	0	0
0	0	0	0	1
0	0	0	0	1
0	0	0	1	0



Bitmap indexes in Oracle: example

```
CREATE TABLE property
(
    property_code NUMBER,
    bedrooms NUMBER,
    receptions NUMBER,
    garages NUMBER
);

CREATE BITMAP INDEX index1 ON property (bedrooms);
CREATE BITMAP INDEX index2 ON property (receptions);
CREATE BITMAP INDEX index3 ON property (garages);

SELECT property_code FROM property
WHERE bedrooms = 4
AND receptions = 3
AND garages = 2
```

Bitmaps can be combined using the logical operations AND, OR , NOT.
Oracle also implements a MINUS operation internally

A MINUS B is equivalent to A AND NOT B

Gold-Jewelry Data: think about

(25; 60) (45; 60) (50; 75) (50; 100)

(50; 120) (70; 110) (85; 140) (30; 260)

(25; 400) (45; 350) (50; 275) (60; 260)

- How would you create

  the bitmap index for age, and

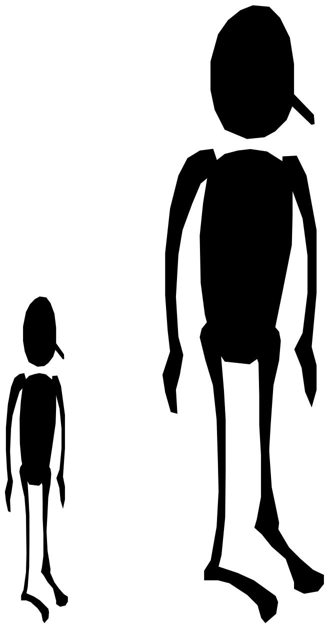
  the bitmap index for salary?

- Suppose we want to find the jewelry buyers with an age in the range 45-55 and a salary in the range 100-200. What do we do?

How big do these things get?

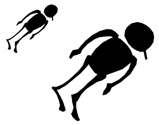
- Assuming each attribute value fits in a 32-bit machine word, the bitmap index for an attribute with value cardinality 32 takes as much space as the base data column

- Since a B-tree index for a 32-bit attribute often uses 3 or 4 times more space than the base data column, many users consider attributes with cardinalities less than 100 to be suitable for using bitmap indices



How big do these things get?

- However, some other users believe: bit map indexes are good for attributes with cardinalities more than 100
- The compression of binary strings is used for sparse bit vectors



Basic Compression

- ***Run length encoding*** is used to encode sequences or ***runs*** of zeros.
- Say that we have 20 zeros, then a 1, then 30 more zeros, then another 1.
- Naively, we could encode this as the integer pair <20, 30>
  - This would work. But what's the problem?
  - On a typical 32-bit machine, an integer uses 32 bits of storage. So our <20, 30> pair uses 64 bits. The original string only had 52!

Basic Compression (Cont'd)

- So we must use a technique that stores our run-lengths as compactly as possible
- Let’s say we have the string **000101**
    - This is made up of runs with 3 zeros and 1 zero.
    - In binary, 3 = 11, while 1 is, of course, just 1
    - This gives us a compressed representation of 111.
- The problem?
    - How do we decompress this?
    - We could interpret this as 1-11 or 11-1 or even 1-1-1.
    - This would give us three different strings after the decompression.

Proper RLE encoding

- We want to uniquely encode run of *i* 0's followed by a 1.
- Let *j* be the number of bits required to represent *i*.

- To define a run, we will use two values:
  1. The "unary" representation of *j*

     A sequence of *j* – 1 "1" bits followed by a zero (the zero signifies the end of the unary string)

     The special cases of *j* = 0 and *j* = 1 use 00 and 01 respectively.
  2. The binary value of *i* (using next *j* bits)

Proper RLE encoding: example

Example: 000000000000010000001

- Here we have two “0” runs of length 13 and 6
- 13 can be represented by 4 bits, 6 requires 3 bits
- Run 1: j – 1 “1” bits + 0 + i → 111 0 1101
- Run 2: j – 1 “1” bits + 0 + i → 11 0 110
- Final compressed string: 11101101110110
- Compression rate: (21-14)/21 = 33%

Decoding

- Let’s decode 11101101001011

1110**1101**001011 → 13

11101101**00**1011 → 0

1110110100**10****11** → 3

Our sequence of run lengths is: 13, 0, 3. What’s the bitmap?

000000000000001100001

Bitmap indexes: summary

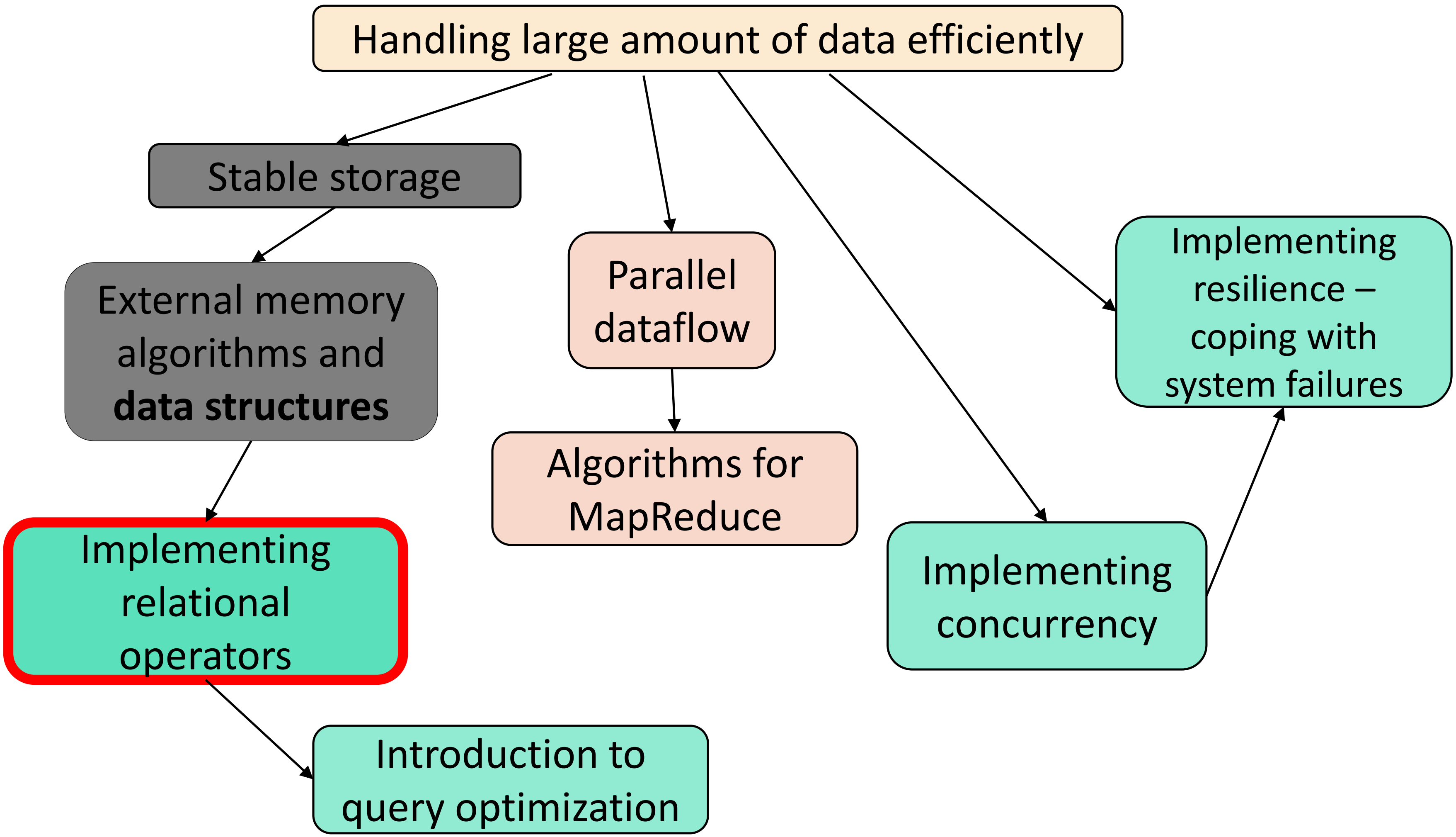
Pros:

1. Bitmaps provide efficient indexing for **low cardinality dimensions**
2. On **sparse, high cardinality dimensions**, compression can be effective
3. Bit operations support **multi-dimensional partial match and range queries**

Cons:

1. **De-compression** requires run-time overhead
2. Bit operations on **large maps** and with large dimension counts can be expensive.
3. Maintaining in **heavy-update scenarios** is **expensive**, thus they are widely used on more static databases (data warehouses)

Roadmap


Handling large amount of data efficiently
Stable storage
External memory algorithms and **data structures**
Parallel dataflow
Algorithms for MapReduce
Implementing resilience – coping with system failures
Implementing concurrency
Implementing relational operators
Introduction to query optimization