Bitmap Index

Objectives:

• Basic introduction to a very different indexing concept.

• Place for the method in application use and culture

Reading: text 14.7

Thanks: a few slides borrowed from Ramakrishnan text slides

Review & Backfill from last time

Access-Path

Access-Path:

- A method for fetching data from disk.

or

- The sequence of blocks/pages required to located and retrieve data.

Usage: “Choose an access path”,

e.g. either a sequential scan based on the primary index, or an available secondary index.

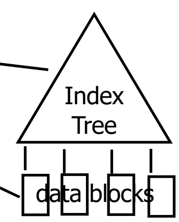
RDBMS Storage

Default: Two Access Paths

B+ tree: Primary index
- Search key aka index key

= primary key

Table Scan aka sequential access path
- sorted on primary key
- (if possible) contiguously on disk.
  - Fully contiguous = 100% clustered



Cultural Background (caveat)

Bitmap index:
- outcome of
  - research in parallel database systems
  - development and research in decision support systems (OLAP, data warehouses)
- Viewed a *very* inconsistent with set theoretic models of database - (even just a hack)
  - academics …. little textbook material
  - actual performance benefit in many circumstances leaves no choice
  - by serendipity, works well in circumstances much broader than first promulgated.

Why?

Speculation [Miranker]
- In a manner consistent enough with relational databases.
- Create a tiny, high density encoding of a database.
  - tiny often means fast.
    - I/O: definitely
    - CPU:

Effectiveness → Taking over

Big Data in the relational model (used directly)

→ Column Store Databases // next time

Bitmaps, as introduction to *Bloom Filters*

- "Bloom Filter" primary method of indexing in Big Data (Hadoop, but others as well)

Basic Idea (1)

- For a select (few) columns/attributes

  Male: row1, row3, row5, row6…
  Female : row2, row4, …

  – represent the contents of each row of the inverted index as a bitmap.

  Male: 101011…
  Female:010100…

Expandable for Any Type

Faculty Table

RowId	FacSSN	…	FacRank
1	098-55-1234		Asst
2	123-45-6789		Asst
3	456-89-1243		Assc
4	111-09-0245		Prof
5	931-99-2034		Asst
6	998-00-1245		Prof
7	287-44-3341		Assc
8	230-21-9432		Asst
9	321-44-5588		Prof
10	443-22-3356		Assc
11	559-87-3211		Prof
12	220-44-5688		Asst

Bitmap Index on FacRank

	Bitmap
Asst	110010010001
Assc	001000100100
Prof	000101001010

Notes:

- As the number of values increases the bitmaps get bigger
  - > results that suggest this method is of value for small domains (NOT, more later)
  - Even continuous data types (numbers and other infinitely sized) this works

Defining Bitmap Indexes

- Oracle syntax:

```
– CREATE BITMAP INDEX
  ON Faculty(FacRank);
```

- PostgreSQL, MS SQL Server, IBM DB2 – users can't declare
  - So effective, query engine will
    - build an index at the beginning of a query,
    - Throw it out when query is done

Maybe a whole lecture later in the semester

Bit vectors can also be used to represent join indexes

- Join indexes:
  - like a “join table” for a many-to-many association
- Oracle syntax:
  - ```
    CREATE BITMAP INDEX
    ON Sales(Customer.State)
    FROM Sales, Customer
    WHERE Sales.Customer_key = Customer.Customer_key
    ```
- Creates an index with <State, Sales RID> entries
- (revisit this at the end of the semester as part of parallel DBMS)

How are these useful?

Select *
From Faculty
Where Faculty.gender = “F” and
Faculty.FacRank = “asst”

- Output records determined by ANDing the two bitmaps together.
  - indexes for all columns of a table are the same length

Example

Select *
From Faculty
Where Faculty.gender = “F” and
Faculty.FacRank = “asst”

	Bitmap
Asst	110010010001
F	01010…
result	01000…

Sizing:

- Consider 8 million rows of faculty:
- five megabytes of index
- two megabytes of I/O to process the query based on the index.
- ________ megabytes of I/O without the index.
- Will another index work?

- What is the impact on size of the index?
  - as a function of number of
    - distinct values, m
    - rows, n

--> (initially thought:) good for small enumerated types

Idea (2a) Compression

- Suppose we have a large domain,
  - Example, domain size = 20
    - probability that a bit is 1?
    - how many zero’s in a row?
- (draw example on board)
    ```

Idea (2a) Compression

- The bitmaps are compressed
- Run length encoding methods,
  - (run length of zero's = number of zero's, terminated by a one)

  e.g. 001, 000, 001, 0000
  - Run length of, 2, followed by run length of 5

  What about zeros at the end of the vector?

  Answer: Know the total number of rows….

How to represent run lengths?

simple sequence of binary numbers won't work

Consider,

e.g. 001, 000, 001, 000, 0

Run length of, 2, followed by run length of 5

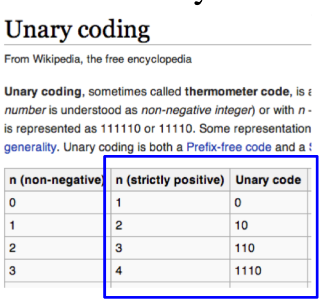
10, 101 = 10101

But 10101, could be run length, 1010, 1

i.e. run length 10 followed by 1

→ 000,000,000,010,1

One Method: Exploits Unary Encoding



Unary coding

From Wikipedia, the free encyclopedia

Unary coding, sometimes called **thermometer code**, is a
number is understood as *non-negative integer*) or with *n* -
is represented as 111110 or 11110. Some representation
generality. Unary coding is both a Prefix-free code and a

n (non-negative)	n (strictly positive)	Unary code
0	1	0
1	2	10
2	3	110
3	4	1110

One Method: Exploits Unary Encoding

- Given, e.g. 001, 000, 001, 000, 0

  Run length of, 2, followed by run length of 5

For each run, represent pair (x : y)

x, number of bits in unary to represent length of the run in binary

y, length of the run in binary

For run length 2

x = 10

y = 10, x:y 1010

One Method: Unary Encoding

- Given, e.g. 001, 000, 001, 000, 0

  Run length of, 2, followed by run length of 5

For each run, represent pair (x : y)

x, number of bits in unary to represent length of the run

y, length of the run in binary

For run length 2

x = 10

y = 10, x:y 1010

For run length 5

x = 110

y = 101 x:y 110101

One Method: Unary Encoding

- Given, e.g. 001, 000, 001, 000, 0

  Run length of, 2, followed by run length of 5

For run length 2

x = 10

y = 10, x:y 1010

For run length 5

x = 110

y = 101 x:y 110101

001, 000, 001, 000, 0 → 1010 110101

Decoding

Decode 1101001011

locate length of first run by finding “0”

1101001011 → run length takes 3 bits to represent

110, 100 (x/3, y/4) → 00001

1011 remains → run length takes 2 bits to represent

10, 11 (x/2, y/3) → 0001

so

1101001011 → 00001 0001

An odd case

- Map = 100, 000, 100

Run length 0, then 5,

To encode 1000…… (i.e. run length 0)

x = 0 //still takes one bit to represent “0”

y = 0 // run length is 0 long

00 110101

Idea (2b) Operations on Compressed bitmaps

Select *
From Faculty
Where Faculty.gender = “F” and
Faculty.FacRank = “asst”

	Bitmap
Asst	110010010001
F	01010…
result	01000…

CounterAsst ↓

0, 0, 2, 2, 3

1, 1, ….

↑ CounterF

Use counters and arithmetic in a merge-like operation

Given Compression, consider:

- attributes with *many* values

→lot’s of zero’s in the maps
→long runs of zeros, occasional 1’s

suppose 100’s of zero’s in a run
- how fast is the merge compared to logical ANDs?
- how fast does index storage grow with added values?

Consider case where: value is a key

- n, records, n = m different values

→each and every map, just one “1”.

worst case, run length of n-1,

→space required for index,

$n * 2 \log_2 n$

What about real numbers?

- Worst-case
  - Every number in a column is unique
  - > but that’s the same as if it were a key

Internal Implementation

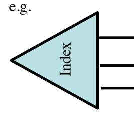
- Map position assignment
  - rows numbered, stored in a special place/table
    - (position, row-id) // supported with secondary index

e.g. (1, row-id_1)
(2, row-id_2)
…

Internal Implementation

- Storing/locating bitmaps
  - (key, value pair),
    - key = index key value
    - value = bitmaps

e.g.



key	bitmap
asst	10010………
assoc	00101……..
full	01101……

Internal Implementation

- Update an indexed value in a row,
  - look up bitmaps for old value and new value
  - flip respective bits (may change vector length)
- Insert new row:
  - assign next row number
  - what happens to existing bitmaps?
- Delete row:
  - set previously set bit to 0
  - "retire" row number (until compaction)

How powerful?

MS SQL Server, PostgreSQL
and
IBM DB2

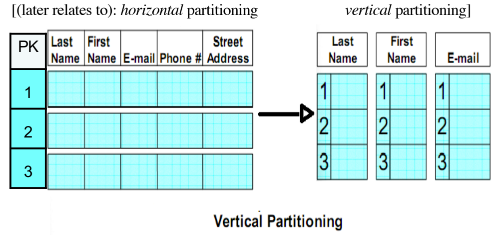
- will build bitmap indexes, dynamically, to execute *a* query
- then thrown the index out

Near Future

- Columns stores

Row Storage *Column* Storage

[(later relates to): *horizontal* partitioning *vertical* partitioning]



Part of databases called "Column-Stores"

Column Stores

- From research
  - MonetDB (open source) [2002]
  - H-store and C-store ~[2005]
- To commercial practice
  - Sybase IQ [1995],
    - bought by SAP…
  - SAP HANA [2008],
    - main-memory, cluster, derived from Sybase IQ
  - HP Vertica
    - [Vertica founded 2005, forked open source C-store fork]
  - Sisense
    - a Tableau competitor, offers similar function but on multiple terabytes on a desktop