Index tuning

Viet-Trung Tran
SoICT

Credit: Pei Li – Database management and performance tuning

Outline

- Query type
- Index type
- Index data structures

Query types: Why?

- Different indexes are good for different query types.
- We identify categories of queries with different index requirements.

Query types [cont'd]

- Point query: returns at most one record
  - SELECT name
    FROM Employee
    WHERE ID = 8478
- Multipoint query: returns multiple records based on equality condition
  - SELECT name
    FROM Employee
    WHERE department = 'IT'
- Range query on *X* returns records with values in interval of *X*
  - SELECT name
    FROM Employee
    WHERE salary >= 155000

Query types

- Extremal query: returns records with max or min values on some attributes
  - SELECT name
    FROM Employee
    WHERE salary = MAX(SELECT salary FROM Employee)
- Ordering query: orders records by some attribute value
  - SELECT *
    FROM Employee
    ORDER BY salary
- Grouping query: partition records into groups; usually a function is applied on each partition
  - SELECT dept, AVG(salary)
    FROM Employee
    GROUP BY dept

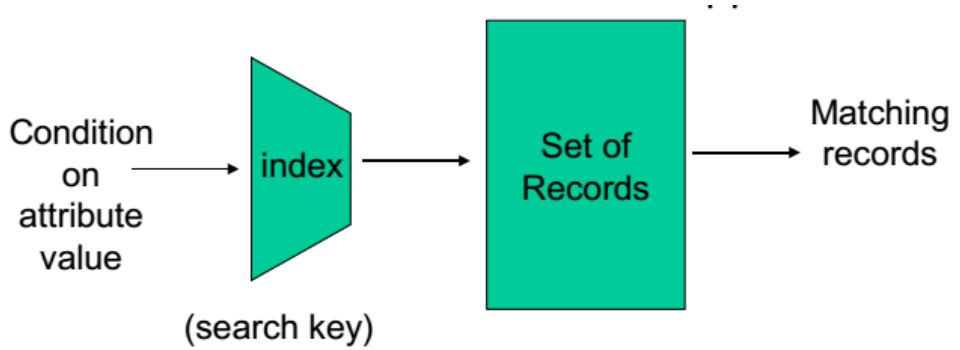
Query types

- Join queries: link two or more tables
  - Equality join:
    - SELECT Employee.ssnum
      FROM Employee, Student
      WHERE Employee.ssnum = Student.ssnum
  - Join with non-equality condition:
    - SELECT e1.ssnum
      FROM Employee e1, Employee e2
      WHERE e1.manager = e2.ssnum
      AND e1.salary > e2.salary

INDEX TYPE

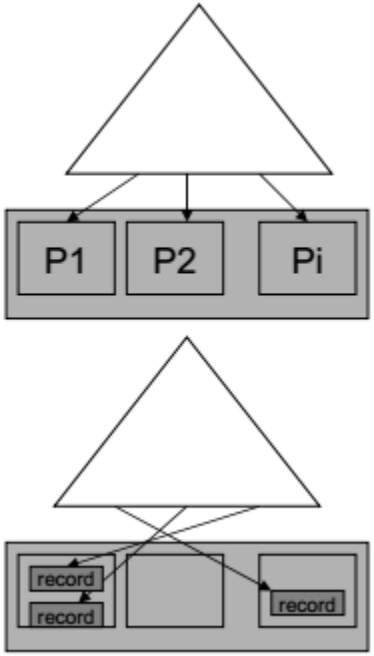
What is an index?

- An index is a data structure that supports efficient access to data:



- Index tuning essential to performance! (2nd to Query tuning)
- Improper index selection can lead to:
  - indexes that are maintained but never used
  - files that are scanned in order to return a single record
  - multi-table joins that run for hours or days

- Indexes can often be viewed as trees (*B*+-tree, hash)
  some nodes are in main memory (e.g., root)
  nodes deeper down in tree are less likely to be in main memory
- Sparse index:
  - pointers to disk pages
    at most one pointer per disk page
    usually much less pointers than records
- Dense index:
  - pointers to individual records
    one key per record
    usually more keys than sparse index
    optimization: store repeating keys only once, followed by pointers



Spare vs. Dense

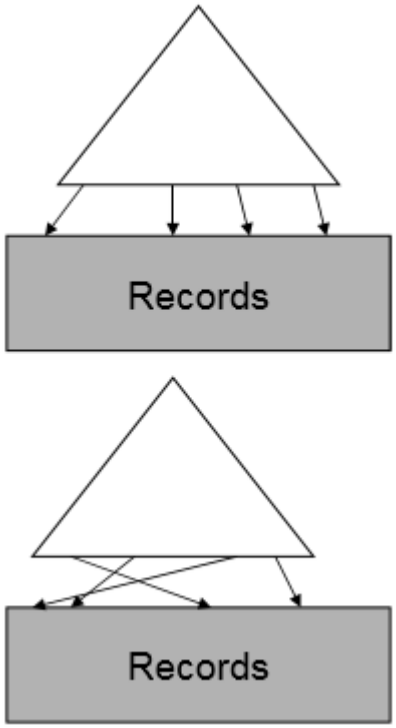
- Number of pointers:
  - ptrs in dense index = records per page × ptrs in sparse index
- Pro sparse: less pointers
  - typically record size is smaller than page size
  - less pointers result in less levels (and disk accesses)
  - uses less space
- Pro dense: index may “cover” query

Covering index

- answers read query within index structure
  - fast, since data is not accessed
- Example:
  - dense index on lastname
  - SELECT COUNT(lastname) WHERE lastname='Smith'

Clustering vs. Non-Clustering

- Clustering index on attribute X
  - (also primary index)
  - records are grouped by attribute X on disk
  - B+-tree: records sorted by attribute X
  - only one clustering index per table
  - dense or sparse
- Non-clustering index on attribute X
  - (also secondary index)
  - no constraint on table organization more than one index per table always dense



Clustering index

- Can be sparse:
  - fewer pointers than non-clustering index (always dense!)
    if record is small, save one disk access per record access
- Good for multi-point queries:
  - equality access on non-unique attribute
  - all result records are on consecutive pages
  - example: look up last name in phone book
- Good for range, prefix, ordering queries:
  - works if clustering index is implemented as *B*+-tree
  - prefix example: look up all last names starting with 'St' in phone book
  - result records are on consecutive pages

Clustering index [cont'd]

- Good for equality join:
  - fast also for join on non-key attributes
  - index on one table: indexed nested-loop
  - index on both tables: merge-join
- Overflow pages reduce efficiency:
  - if disk page is full, overflowing records go to overflow pages
  - overflow pages require additional disk accesses

Equality Join with Clustering Index

- Example query:
  - SELECT Employee.ssnum, Student.course
    FROM Employee, Student
    WHERE Employee.firstname = Student.firstname
- Index on Emplyee.firstname:
  - **use index nested loop join**
  - for each student look up employees with same first name
  - all matching employees are on consecutive pages
- Index on both firstname attributes:
  - **use merge join**
  - read both tables in sorted order and merge (*B*+-tree)
  - each page read exactly once
  - works also for hash indexes with same hash function

Clustering Index and Overflow Pages

- Why overflow pages?
  - clustering index stores records on consecutive disk pages
  - insertion between two consecutive pages not possible
  - if disk page is full, overflowing records go to overflow pages
- Additional disk access for overflow page: reduced speed
- Overflow pages can result from:
  - Inserts
  - updates that change key value
  - updates that increase record size (e.g., replace NULL by string)
- Reorganize index:
  - invoke special tool
  - or simply drop and re-create index

Non-Clustering Index

- Always useful for point queries.
- Particularly good if index covers query.
- Multi-point query (but not covered): only if not too selective
  - *nR*: number of records returned by query
  - *nP*: number of disk pages in table
  - the *nR* records are uniformly distributed over all pages
  - thus query will read min(*nR*, *nP*) disk pages
- Index may slow down highly selective multi-point query:
  - scan is by factor 2–10 faster than accessing all pages with index
  - thus *nR* should be significantly smaller than *nP*

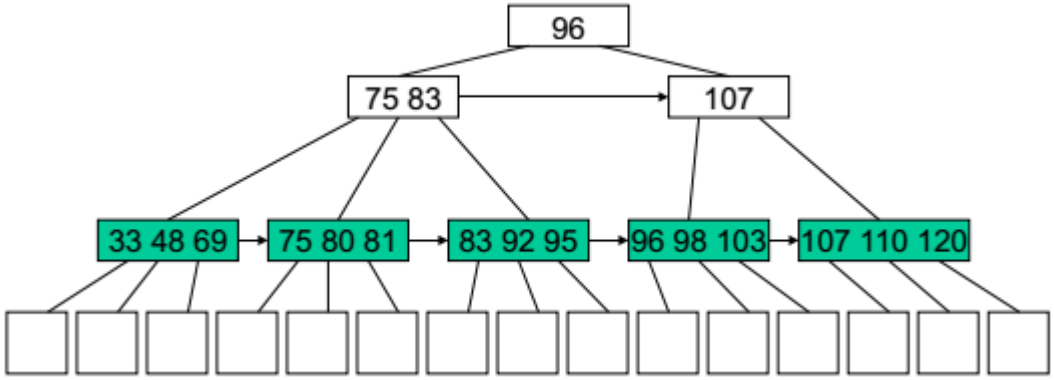
INDEX DATA STRUCTURES

Index data structure

- Indexes can be implemented with different data structures.
  - ***B*+-tree index**
  - **hash index**
  - **bitmap index (briefly)**
  - dynamic hash indexes: number of buckets modified dynamically
  - R-tree: index for spacial data (points, lines, shapes)
  - quadtree: recursively partition a 2D plane into four quadrants
  - octree: quadtree version for three dimensional data
  - main memory indexes: T-tree, binary search tree

B+-Tree

- balanced tree of key-pointer pairs
- keys are sorted by value
- nodes are at least half full
- access records for key: traverse tree from root to leaf



Key Length and Fanout

- Key length is relevant in *B*+-trees: short keys are good!
  - fanout is maximum number of key-pointer pairs that fit in node
  - long keys result in small fanout
  - small fanout results in more levels
    - Store $40M$ key-pointer pairs in leaf pages (page: $4kB$, pointer: $4B$)
      - $6B$ key: fanout 400 $\Rightarrow$ 3 block reads per accesses

level	nodes	key-pointer pairs
1	1	400
2	400	160,000
3	160,000	64,000,000

      - $96B$ key: fanout 40 $\Rightarrow$ 5 block reads per accesses

level	nodes	key-pointer pairs
1	1	40
2	40	1,600
3	1,600	64,000
4	64,000	2,560,000
5	2,560,000	102,400,000

      - $6B$ key almost twice as fast as $96B$ key!

Estimate Number of Levels

- Page utilization:
  - examples assumes 100% utilization
  - typical utilization is 69% (if half-full nodes are merged)
- Number of levels:

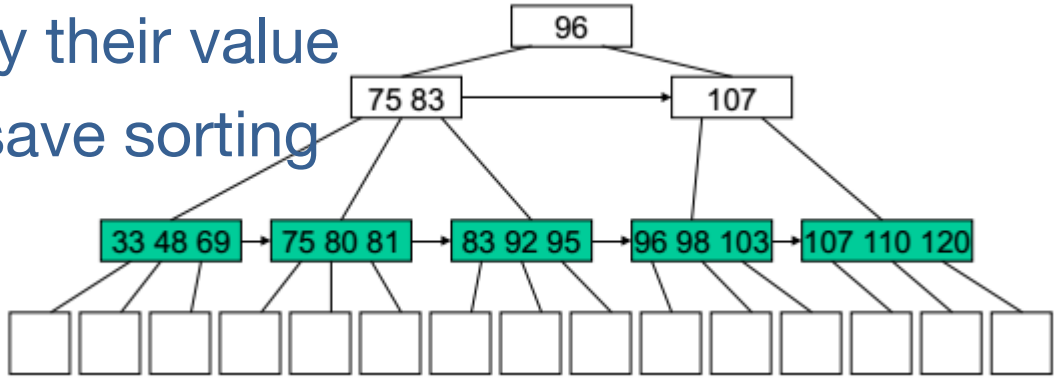
$$\text{fanout} = \left\lfloor \frac{\text{node size}}{\text{key-pointer size}} \right\rfloor$$

$$\text{number of levels} = \left\lceil \log_{\text{fanout} \times \text{utilization}} (\text{leaf key-pointer pairs}) \right\rceil$$

- Previous example with utilization = 69%:
  - 6*B* key: fanout = 400, levels = $d3.11e = 4$
  - 96*B* key: fanout = 40, levels = $d5.28e = 6$

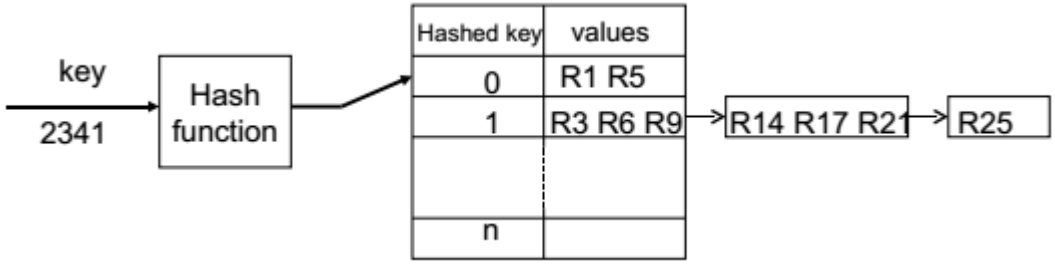
B+-tree index supports

- **point**: traverse tree once to find page
- **multi-point:** traverse tree once to find page(s)
- **range**: traverse tree once to find one interval endpoint and follow pointers between index nodes
- **prefix**: traverse tree once to find prefix and follow pointers between index nodes
- **extremal**: traverse tree always to left/right (MIN/MAX)
- **ordering**: keys ordered by their value
- **grouping**: ordered keys save sorting



Hash index

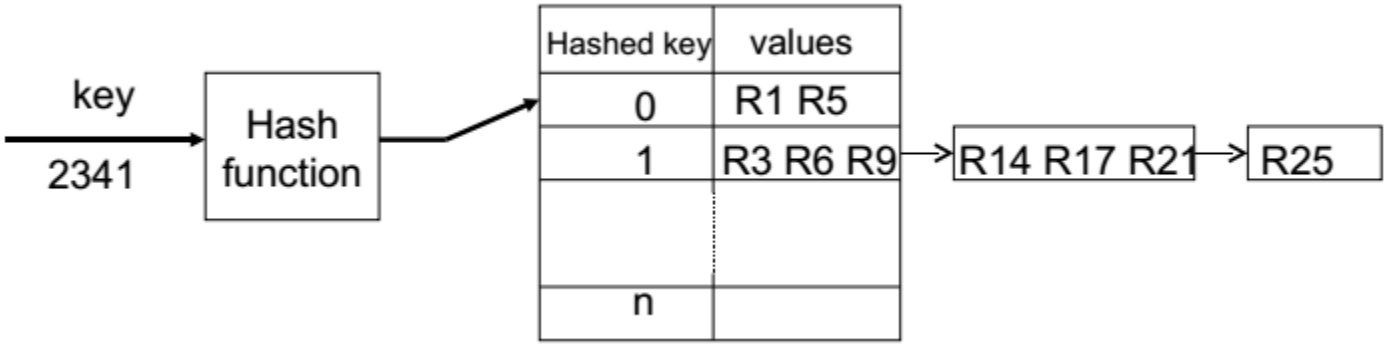
- Hash function:
  - maps keys to integers in range [0..*n*] (hash values)
  - pseudo-randomizing: most keys are uniformly distributed over range
  - similar keys usually have very different hash values!
  - database chooses good hash function for you
- Hash index:
  - hash function is “root node” of index tree
  - hash value is a bucket number
  - bucket either contains records for search key or pointer to overflow chain with records



- Key length:
  - size of hash structure independent of key length
  - key length slightly increases CPU time for hash function

Hash index usage

- Hash index supports
  - **point**: single disk access!
  - **multi-point**: single disk access to first record
  - **grouping**: grouped records have same hash value
- Hash index is useless for
  - range, prefix, extremal, ordering
  - similar key values have dissimilar hash values
  - thus similar keys are in different pages



Bitmap index

- Every value has it's own column == bitmap.
- One bit vector per attribute value (e.g., two for gender)
  - length of each bit vector is number of records
  - bit *i* for vector "male" is set if key value in row *i* is "male"

Value List

	$\pi_A(R)$	B^8	B^7	B^6	B^5	B^4	B^3	B^2	B^1	B^0
1	3	0	0	0	0	0	1	0	0	0
2	2	0	0	0	0	0	0	1	0	0
3	1	0	0	0	0	0	0	0	1	0
4	2	0	0	0	0	0	0	1	0	0
5	8	1	0	0	0	0	0	0	0	0
6	2	0	0	0	0	0	0	1	0	0
7	2	0	0	0	0	0	0	1	0	0
8	0	0	0	0	0	0	0	0	0	1
9	7	0	1	0	0	0	0	0	0	0
10	5	0	0	0	1	0	0	0	0	0
11	6	0	0	1	0	0	0	0	0	0
12	4	0	0	0	0	1	0	0	0	0

(a) (b)

Bitmap index

- Works best if
  - query predicates are on many attributes
  - the individual predicates have high selectivity (e.g., male/female)
  - all predicates together have low selectivity (i.e., return few tuples)
- Advantages
  - Compact size.
  - Efficient hardware support for bitmap operations (AND, OR, XOR, NOT).
  - Fast search.
  - Multiple differentiate bitmap indexes for different kind of queries.

Bitmap index example

- “Find females who have brown hair, blue eyes, wear glasses, are between 50 and 60, work in computer industry, and live in Bolzano”

INDEX TUNING EXERCISES

- The examples use the following tables:
  - Employee(ssnum, name, dept, manager, salary)
  - Student(ssnum, name, course, grade, stipend, evaluation)

Exercise 1 – Query for Student by Name

- Student was created with non-clustering index on name.
- Query:
  - SELECT *
    FROM Student
    WHERE name='Bayer'
- **Problem**: Query does not use index on name.
- **Solution**: Try updating the catalog statistics.
  - Oracle, Postgres: ANALYZE
  - SQL Server: sp createstats
  - DB2: RUNSTATS

Exercise 2 – Query for Salary I

- Non-clustering index on salary.
- Catalog statistics are up-to-date.
- Query:
  SELECT *
  FROM Emplyee
  WHERE salary/12 = 4000
- **Problem**: Query is too slow.
- Solution: Index not used because of the arithmetic expression.
  - Rewrite query:
    - SELECT *
      FROM Emplyee
      WHERE salary = 48000

Exercise 3 – Query for Salary II

- Non-clustering index on salary.
- Catalog statistics are up-to-date.
- Query:
  SELECT *
  FROM Emplyee
  WHERE salary = 48000
- **Problem**: Query still does not use index. What could be the reason?
- **How**: The index is non-clustering. Many employees have a salary of 48000, thus the index may not help. It may still help for other, less frequent, salaries!

Exercise 4 – Clustering Index and Overflows

- Clustering index on Student.ssnum
- Page size: *2kB*
- Record size in Student table: 1*KB* (evaluation is a long text)
- **Problem**: Overflow when new evaluations are added.
- **Solution**: Clustering index does not help much due to large record size. A non-clustering index avoids overflows.

Exercise 5 – Non-clustering Index I

- Employee table:
  - 30 employee records per page
  - each employee belongs to one of 50 departments (dept)
  - the departments are of similar size
- Query:
  - SELECT ssnum
FROM Emplyee
WHERE dept = 'IT'
- **Problem**: Does a non-clustering index on Employee.dept help?
- **Solution**:
  - Only if the index covers the query.
  - 30/50=60% of the pages will have a record with dept = 'IT'
  - table scan is faster than accessing 3/5 of the pages in random order

Exercise 6 – Non-clustering Index II

- Employee table:
  - 30 employee records per page
  - each employee belongs to one of 5000 departments (dept)
  - the departments are of similar size
- Query:
  - SELECT ssnum
    FROM Emplyee
    WHERE dept = 'IT'
- **Problem**: Does a non-clustering index on Employee.dept help?
- **Solution:**
  - Only if the index covers the query.
  - only 30/5000=0.06% of the pages will have a record with dept='IT'
  - table scan is slower

Exercise 7 – Statistical Analysis

- Auditors run a statistical analysis on a copy of Emplyee.
- Queries:
  - count employees with a certain salary (frequent)
  - find employees with maximum or minimum salary within a particular department (frequent)
  - find an employee by its social security number (rare)
- **Problem**: Which indexes to create?
- Solution:
  - non-clustering index on salary (covers the query)
  - clustering composite index on (dept, salary) using a *B*+-tree (all employees with the maximum salary are on consecutive pages)
  - non-clustering hash index on ssnum

Exercise 8 – Algebraic Expressions

- Student stipends are monthly, employee salaries are yearly.
- Query: Which employee is paid as much as which student?
- There are two options to write the query:
  - SELECT * SELECT * FROM Employee, Student WHERE salary = 12*stipend
  - SELECT * FROM Employee, Student WHERE salary/12 = stipend
- Index on a table with an algebraic expression not used.
- **Problem**: Which query is better?

Exercise 8 – Solution

- If index on only one table, it should be used.
- Index on both tables, clustering on larger table: use it.
- Index on both tables, non-clustering on larger table:
  - small table has (much) less records than large table has pages:
    use index on large table
  - otherwise: use index on small table

Exercise 9 – Purchasing Department

- Purchasing department maintains table
  - Onorder(supplier,part,quantity,price).
- The table is heavily used during the opening hours, but not over night.
- **Queries**:
  - Q1: add a record, all fields specified (very frequent)
  - Q2: delete a record, supplier and part specified (very frequent)
  - Q3: find total quantity of a given part on order (frequent)
  - Q4: find the total value on order to a given supplier (rare)
- **Problem**: Which indexes should be used?

Exercise 9 – Solution

- **Queries**:
  - Q1: add a record, all fields specified (very frequent)
  - Q2: delete a record, supplier and part specified (very frequent)
  - Q3: find total quantity of a given part on order (frequent)
  - Q4: find the total value on order to a given supplier (rare)
- **Solution**: Clustering composite *B*+-tree index on (part,supplier).
  - eliminate overflows over night
  - attribute order important to support query Q3
  - hash index will not work for query Q3 (prefix match query)
- **Discussion**: Non-clustering index on supplier to answer query Q4?
  - index must be maintained and will hurt the performance of much more frequent queries Q1 and Q2
  - index does not help much if there are only few different suppliers

Exercise 10 – Point Query Too Slow

- Employee has a clustering *B*+-tree index on ssnum.
- Queries:
  - retrieve employee by social security number (ssnum)
  - update employee with a specific social security number
- **Problem**: Throughput is still not enough.
- Solution: Use hash index instead of *B*+-tree (faster for point queries).