Database Management and Performance Tuning

Index Tuning

Pei Li

University of Zurich
Institute of Informatics

Unit 5

Acknowledgements: The slides are provided by Nikolaus Augsten
and adapted from "Database Tuning" by Dennis Shasha and Philippe Bonnet.

Outline

1. Index Tuning
   - Data Structures
   - Composite Indexes

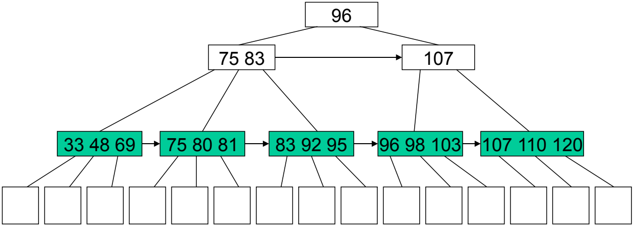
Outline

1. Index Tuning
   - Data Structures
   - Composite Indexes

Index Data Structures

- Indexes can be implemented with different data structures.
- We discuss:
  - B^+-tree index
  - hash index
  - bitmap index (briefly)
- Not discussed here:
  - 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, 2-3 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

Key Length and Fanout – Example

- 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:

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

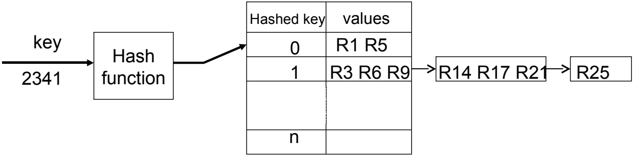
$$\text{number of levels} = \lceil \log_{\text{fanout} \times \text{utilization}}(\text{leaf key-pointer pairs}) \rceil$$

- Previous example with utilization = 69%:
  - $6B$ key: fanout $= 400$, levels $= \lceil 3.11 \rceil = 4$
  - $96B$ key: fanout $= 40$, levels $= \lceil 5.28 \rceil = 6$

Key Compression

- Key compression: produce smaller keys
  - reduces number of levels
  - adds some CPU cost (ca. 30% per access)
- Key compression is useful if
  - keys are long, for example, string keys
  - data is static (few updates)
  - CPU time is not an issue
- Prefix compression: very popular
  - non-leaf nodes only store prefix of key
  - prefix is long enough to distinguish neighbors
  - example: Cagliari, Casoria, Catanzaro → Cag, Cas, Cat

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

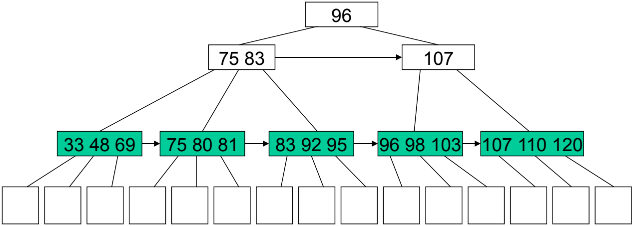
Overflow Chains

- Hash index without overflows: single disk access
- If bucket is full: overflow chain
  - each overflow page requires additional disk access
  - under-utilize hash space to avoid chains!
  - empirical utilization value: 50%
- Hash index with many overflows: reorganize
  - use special reorganize function
  - or simply drop and add index

Bitmap Index

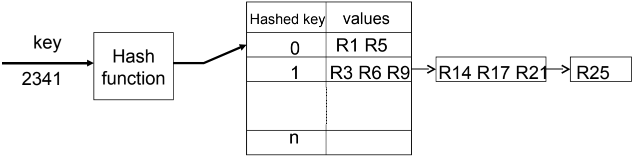
- Index for data warehouses
- One bit vector per attribute value (e.g., two for gender)
  - length of each bit vector is number of records
  - bit *i* for vector “male” is set if key value in row *i* is “male”
- Works best if
  - query predicates are on many attributes
  - the individual predicates have high selectivity (e.g., male/female)
  - all predicates together have low selectivity (i.e., return few tuples)
- Example: “Find females who have brown hair, blue eyes, wear glasses, are between 50 and 60, work in computer industry, and live in Bolzano”

Which Queries Are Supported?



- 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

Which Queries Are Supported?

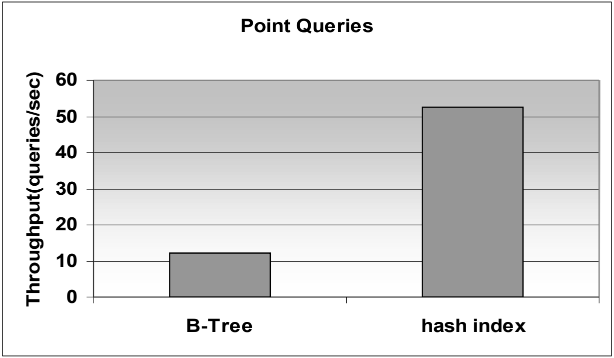


- Hash index supports
  - point: single disk access!
  - multi-point: single disk access to first record
  - grouping: grouped records have same hash value
- Hash index is useless for
  - range, prefix, extremal, ordering
  - similar key values have dissimilar hash values
  - thus similar keys are in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

Experimental Setup

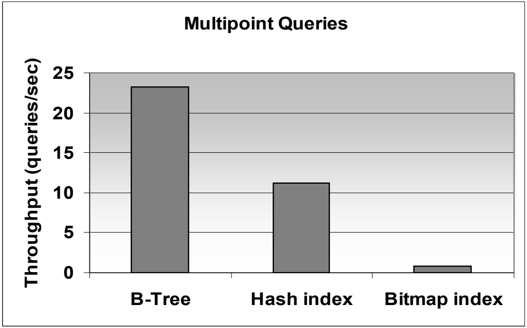
- `Employee(ssnum, name, hundreds ...)`
- 1,000,000 records
- `ssnum` is a key (point query)
- `hundreds` has the same value for 100 employees (multipoint query)
- point query: index on `ssnum`
- multipoint and range query: index on `hundreds`
- B^+-tree and hash indexes are clustered
- bitmap index is never clustered

Experiment: Point Query



Oracle 8i Enterprise Edition on Windows 2000.

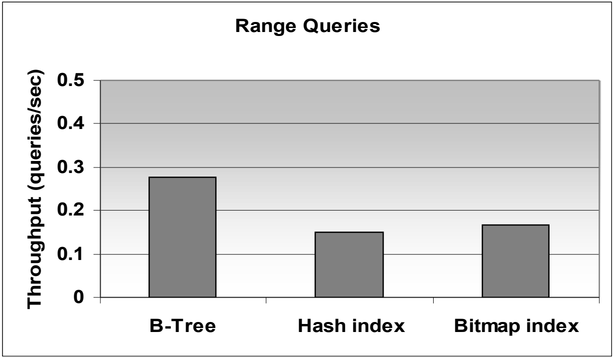
Experiment: Multi-point Query



- Setup: 100 records returned by each query
- B^+-tree: efficient since records are on consecutive pages
- Hash index: key maps to single page and produces an overflow chain
- Bitmap index: traverses entire bitmap to fetch a few records

Oracle 8i Enterprise Edition on Windows 2000.

Experiment: Range Query



- B^+-tree: efficient since records are on consecutive pages
- Hash index, bitmap index: do not help

Oracle 8i Enterprise Edition on Windows 2000.

Outline

Composite Indexes

- Index on more than one attribute (also "concatenated index")
- Example: `Person(ssnum,lastname,firstname,age,address,...)`
  - composite index on `(lastname,firstname)`
  - phone books are organized like that!
- Index can be dense or sparse.
- Dense index on (A, B, C)
  - one pointer is stored per record
  - all pointers to records with the same A value are stored together
  - within one A value, pointers to same B value stored together
  - within one A and B value, pointers to same C value stored together

Composite Indexes – Efficient for Prefix Queries

- Example: composite index on (lastname,firstname)

  ```
  SELECT * FROM Person
  WHERE lastname='Gates' and firstname LIKE 'Ge%'
  ```

- Composite index more efficient than two single-attribute indexes:
  - many records may satisfy `firstname LIKE 'Ge%'`
  - condition on `lastname` and `firstname` together has lower selectivity
  - two-index solution: results for indexes on `lastname` and `firstname` must be intersected
- Dense composite indexes can cover prefix query.

Composite Indexes – Skip Scan in Oracle

- Typically composite index on `(lastname,firstname)` not useful for

```
SELECT lastname FROM Person
WHERE firstname='George'
```

- Problem: Index covers query, but condition is not a prefix.
- Solution: Index skip scan (implemented in Oracle)
  - composite index on (A, B)
  - scan each A value until you find required B values
  - then jump to start of next A value
  - partial index scan instead of full table scan!
  - especially useful if A can take few values (e.g., male/female)

Composite Indexes – Multicolumn Uniqueness

- Example: `Order(supplier, part, quantity)`
  - supplier is not unique
  - part is not unique
  - but `(supplier,part)` is unique
- Efficient way to ensure uniqueness:
  - create unique, composite index on `(supplier,part)`
  - `CREATE UNIQUE INDEX s_p ON Order(supplier,part)`

Composite Indexes – Attribute Order Matters

- Put attribute with more constraints first.
- Example: Geographical Queries
  - table: `City(name,longitude,latitude,population)`

```
SELECT name FROM city
WHERE population >= 10000 AND latitude = 22
      AND longitude >= 5  AND longitude <= 15
```

- Efficient: clustered composite index on `(latitude,longitude)`
  - pointers to all result records are packed together
- Inefficient: clustered composite index on `(longitude, latitude)`
  - each longitude 5 to 15 has some pointers to latitude 22 records
- General geographical queries should use a multi-dimensional index (for example, an R-tree)

Disadvantages of Composite Indexes

- Large key size:
  - B^+ tree will have many layers
  - key compression can help
  - hash index: large keys no problem, but no range and prefix queries supported
- Expensive updates:
  - in general, index must be updated when key attribute is updated
  - composite index has many key attributes
  - update required if any of the attributes is updated