CS315: Database Systems
Indexing

Arnab Bhattacharya
arnabb@cse.iitk.ac.in

Computer Science and Engineering,
Indian Institute of Technology, Kanpur
http://web.cse.iitk.ac.in/~cs315/

2nd semester, 2022-23

Tue 10:30-11:45, Thu 12:00-13:15

Basics

- Indexing is used to speed up search
- A search key is used
- An index file consists of records or index entries which has two fields
    1. Search key: Attribute that is used for searching
    2. Pointer to the entire object or tuple
- Index files should be smaller than data files

Basics

- Indexing is used to speed up search
- A search key is used
- An index file consists of records or index entries which has two fields
    1. Search key: Attribute that is used for searching
    2. Pointer to the entire object or tuple
- Index files should be smaller than data files
- Index evaluation metrics
    - Search time
    - Modification overhead
    - Space overhead

Basics

- Indexing is used to speed up search
- A search key is used
- An index file consists of records or index entries which has two fields
  1. Search key: Attribute that is used for searching
  2. Pointer to the entire object or tuple
- Index files should be smaller than data files
- Index evaluation metrics
  - Search time
  - Modification overhead
  - Space overhead
- Two basic types of indices
  1. Ordered index: search keys are organized according to some order
  2. Hash index: search keys are organized according to a hash function

Static Hashing

- A hash function maps a key to a bucket
- A bucket is a unit of storage
- It is typically a disk block
- A key may need to be searched sequentially inside a bucket
- Results in hash file organization
- Example: mod n where n is the number of buckets

Hash Function

- Two important qualities of an ideal hash function
- Uniform: Total number of keys from the domain is spread uniformly over all the buckets
- Random: Number of keys in each bucket is same irrespective of the actual distribution of keys

Bucket Overflow

- Buckets may overflow at runtime due to
  - Skew in distribution of keys
  - Non-uniformity of hash function
- Probability can only be reduced, but not eliminated
- Handled using overflow buckets

Bucket Overflow

- Buckets may overflow at runtime due to
  - Skew in distribution of keys
  - Non-uniformity of hash function
- Probability can only be reduced, but not eliminated
- Handled using overflow buckets
- Closed addressing:

Bucket Overflow

- Buckets may overflow at runtime due to
  - Skew in distribution of keys
  - Non-uniformity of hash function
- Probability can only be reduced, but not eliminated
- Handled using overflow buckets
- Closed addressing: Overflow buckets are linked together
  - Also called separate chaining or chaining

Bucket Overflow

- Buckets may overflow at runtime due to
  - Skew in distribution of keys
  - Non-uniformity of hash function
- Probability can only be reduced, but not eliminated
- Handled using overflow buckets
- Closed addressing: Overflow buckets are linked together
  - Also called separate chaining or chaining
- Open addressing:

Bucket Overflow

- Buckets may overflow at runtime due to
  - Skew in distribution of keys
  - Non-uniformity of hash function
- Probability can only be reduced, but not eliminated
- Handled using overflow buckets
- Closed addressing: Overflow buckets are linked together
  - Also called separate chaining or chaining
- Open addressing: Excess keys assigned to some other bucket and total number of buckets is kept fixed
  - Address at i^{th} attempt for key k is $h(k, i) = (h(k) + o_i.i) \bmod m$
  - Linear probing:

Bucket Overflow

- Buckets may overflow at runtime due to
  - Skew in distribution of keys
  - Non-uniformity of hash function
- Probability can only be reduced, but not eliminated
- Handled using overflow buckets
- Closed addressing: Overflow buckets are linked together
  - Also called separate chaining or chaining
- Open addressing: Excess keys assigned to some other bucket and total number of buckets is kept fixed
  - Address at i^{th} attempt for key k is $h(k,i) = (h(k) + o_i.i) \bmod m$
  - Linear probing: Interval between buckets is fixed
    - $h(k,i) = (h(k) + c.i) \bmod m$
  - Quadratic probing:

Bucket Overflow

- Buckets may overflow at runtime due to
  - Skew in distribution of keys
  - Non-uniformity of hash function
- Probability can only be reduced, but not eliminated
- Handled using overflow buckets
- Closed addressing: Overflow buckets are linked together
  - Also called separate chaining or chaining
- Open addressing: Excess keys assigned to some other bucket and total number of buckets is kept fixed
  - Address at i^{th} attempt for key k is $h(k, i) = (h(k) + o_i.i) \bmod m$
  - Linear probing: Interval between buckets is fixed
    - $h(k, i) = (h(k) + c.i) \bmod m$
  - Quadratic probing: Interval between buckets increase linearly
    - $h(k, i) = (h(k) + c.i.i) \bmod m$
  - Double hashing:

Bucket Overflow

- Buckets may overflow at runtime due to
  - Skew in distribution of keys
  - Non-uniformity of hash function
- Probability can only be reduced, but not eliminated
- Handled using overflow buckets
- Closed addressing: Overflow buckets are linked together
  - Also called separate chaining or chaining
- Open addressing: Excess keys assigned to some other bucket and total number of buckets is kept fixed
  - Address at i^{th} attempt for key k is $h(k, i) = (h(k) + o_i.i) \bmod m$
  - Linear probing: Interval between buckets is fixed
    - $h(k, i) = (h(k) + c.i) \bmod m$
  - Quadratic probing: Interval between buckets increase linearly
    - $h(k, i) = (h(k) + c.i.i) \bmod m$
  - Double hashing: Interval computed using another hash function h'
    - $h(k, i) = (h(k) + h'(k).i) \bmod m$

Bucket Overflow

- Buckets may overflow at runtime due to
  - Skew in distribution of keys
  - Non-uniformity of hash function
- Probability can only be reduced, but not eliminated
- Handled using overflow buckets
- Closed addressing: Overflow buckets are linked together
  - Also called separate chaining or chaining
- Open addressing: Excess keys assigned to some other bucket and total number of buckets is kept fixed
  - Address at i^{th} attempt for key k is $h(k, i) = (h(k) + o_i.i) \bmod m$
  - Linear probing: Interval between buckets is fixed
    - $h(k, i) = (h(k) + c.i) \bmod m$
  - Quadratic probing: Interval between buckets increase linearly
    - $h(k, i) = (h(k) + c.i.i) \bmod m$
  - Double hashing: Interval computed using another hash function h'
    - $h(k, i) = (h(k) + h'(k).i) \bmod m$
- Changing size of a database is a problem
- Periodic re-hashing is the only solution
- Dynamic hashing: h changes dynamically but deterministically

Dynamic Hashing

- Dynamic hashing

Dynamic Hashing

- Dynamic hashing
- Organize overflow buckets as binary trees
- m binary trees for m primary pages

Dynamic Hashing

- Dynamic hashing
- Organize overflow buckets as binary trees
- m binary trees for m primary pages
- $h_0(k)$ produces index of primary page
- Particular access structure for binary trees

Dynamic Hashing

- Dynamic hashing
- Organize overflow buckets as binary trees
- m binary trees for m primary pages
- $h_0(k)$ produces index of primary page
- Particular access structure for binary trees
- Family of functions $g(k) = \{h_1(k), \ldots, h_i(k), \ldots\}$
- Each $h_i(k)$ produces a *bit*
- At level i, if $h_i(k) = 0$, take left branch, otherwise right branch

Dynamic Hashing

- Dynamic hashing
- Organize overflow buckets as binary trees
- m binary trees for m primary pages
- $h_0(k)$ produces index of primary page
- Particular access structure for binary trees
- Family of functions $g(k) = \{h_1(k), \ldots, h_i(k), \ldots\}$
- Each $h_i(k)$ produces a *bit*
- At level i, if $h_i(k) = 0$, take left branch, otherwise right branch
- Example: bit representation

Ordered Index

- Index sequential file: ordered sequential file with an index

Ordered Index

- Index sequential file: ordered sequential file with an index
- Primary index or clustering index: index whose search key specifies the sequential order of the file
  - Generally primary key
- Secondary index or non-clustering index: any other index

Ordered Index

- Index sequential file: ordered sequential file with an index
- Primary index or clustering index: index whose search key specifies the sequential order of the file
  - Generally primary key
- Secondary index or non-clustering index: any other index
- Dense index: index record appears for every key
  - Secondary index must be dense
- Sparse index: index record appears for only some keys
  - Records must be sequentially ordered by key
  - To locate k, find largest key $< k$, and then sequential search from there

Ordered Index

- Index sequential file: ordered sequential file with an index
- Primary index or clustering index: index whose search key specifies the sequential order of the file
  - Generally primary key
- Secondary index or non-clustering index: any other index
- Dense index: index record appears for every key
  - Secondary index must be dense
- Sparse index: index record appears for only some keys
  - Records must be sequentially ordered by key
  - To locate k, find largest key $< k$, and then sequential search from there
  - Deletion: Replaced by next key
  - Insertion: Inserted only if entries exceed a block
- Sparse index requires less space and less maintenance but more searching time

Ordered Index

- Index sequential file: ordered sequential file with an index
- Primary index or clustering index: index whose search key specifies the sequential order of the file
  - Generally primary key
- Secondary index or non-clustering index: any other index
- Dense index: index record appears for every key
  - Secondary index must be dense
- Sparse index: index record appears for only some keys
  - Records must be sequentially ordered by key
  - To locate k, find largest key $< k$, and then sequential search from there
  - Deletion: Replaced by next key
  - Insertion: Inserted only if entries exceed a block
- Sparse index requires less space and less maintenance but more searching time
- Multilevel index: primary index does not fit in memory
  - Outer index: Sparse primary index
  - Inner index: Dense primary index file

B-Tree

- Balanced hierarchical data structure

B-Tree

- Balanced hierarchical data structure
- Keys (and associated objects) are in secondary storage, i.e., disk

B-Tree

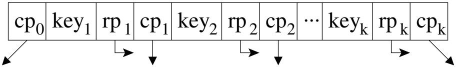
- Balanced hierarchical data structure
- Keys (and associated objects) are in secondary storage, i.e., disk
- A B-tree of order Θ has the following properties:
  1. Leaf nodes are in same level, i.e., the tree is balanced
  2. Root has at least 1 key
  3. Other internal nodes have between Θ and 2Θ keys
  4. An internal node with k keys have $k + 1$ children
  5. Child pointers in leaf nodes are null
- Branching factor is between $\Theta + 1$ and $2\Theta + 1$
- Pointer to the object corresponding to a key is stored alongside

B+-Tree

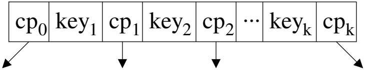
- B+-tree is the most important variety of B-tree
- Internal nodes do *not* contain pointers to objects
- Data is stored *only* at the leaves

B+-Tree

- B+-tree is the most important variety of B-tree
- Internal nodes do *not* contain pointers to objects
- Data is stored *only* at the leaves
- Often siblings are connected by pointers to avoid parent traversal



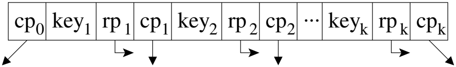
B-tree node



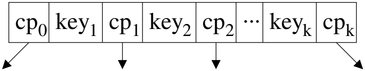
B+-tree node

B+-Tree

- B+-tree is the most important variety of B-tree
- Internal nodes do *not* contain pointers to objects
- Data is stored *only* at the leaves
- Often siblings are connected by pointers to avoid parent traversal



B-tree node



B+-tree node

- More keys can fit in a B+-tree
- Height may be less

Order

- Order mainly determined by disk page size

Order

- Order mainly determined by disk page size
- Disk page capacity is C bytes
- Key consumes γ bytes
- Pointer to key (or object) requires η bytes

Order

- Order mainly determined by disk page size
- Disk page capacity is C bytes
- Key consumes γ bytes
- Pointer to key (or object) requires η bytes
- For B+-tree

Order

- Order mainly determined by disk page size
- Disk page capacity is C bytes
- Key consumes γ bytes
- Pointer to key (or object) requires η bytes
- For B+-tree

$$C \geq 2.\Theta.\gamma + (2.\Theta + 1).\eta$$
$$\Rightarrow \Theta_{\text{B+-tree}} = \left\lfloor \frac{C - \eta}{2(\gamma + \eta)} \right\rfloor$$

Order

- Order mainly determined by disk page size
- Disk page capacity is C bytes
- Key consumes γ bytes
- Pointer to key (or object) requires η bytes
- For B+-tree

$$C \geq 2.\Theta.\gamma + (2.\Theta + 1).\eta$$
$$\Rightarrow \Theta_{\text{B+-tree}} = \left\lfloor \frac{C - \eta}{2(\gamma + \eta)} \right\rfloor$$

- For B-tree

Order

- Order mainly determined by disk page size
- Disk page capacity is C bytes
- Key consumes γ bytes
- Pointer to key (or object) requires η bytes
- For B+-tree

$$C \geq 2.\Theta.\gamma + (2.\Theta + 1).\eta$$
$$\Rightarrow \Theta_{\text{B+-tree}} = \left\lfloor \frac{C - \eta}{2(\gamma + \eta)} \right\rfloor$$

- For B-tree

$$C \geq 2.\Theta.\gamma + 2.\Theta.\eta + (2.\Theta + 1).\eta$$
$$\Rightarrow \Theta_{\text{B-tree}} = \left\lfloor \frac{C - \eta}{2(\gamma + 2\eta)} \right\rfloor$$

Example

- What is the order of a B+-tree and a B-tree with a page size of 4 KB indexing keys of 8 bytes each, and having pointers of size 4 bytes?

Example

- What is the order of a B+-tree and a B-tree with a page size of 4 KB indexing keys of 8 bytes each, and having pointers of size 4 bytes?
    - $\Theta_{\text{B+-tree}} = \left\lfloor \frac{4\times1024-4}{2(8+4)} \right\rfloor = 170$

Example

- What is the order of a B+-tree and a B-tree with a page size of 4 KB indexing keys of 8 bytes each, and having pointers of size 4 bytes?
  - $\Theta_{\text{B+-tree}} = \left\lfloor \frac{4 \times 1024 - 4}{2(8+4)} \right\rfloor = 170$
  - $\Theta_{\text{B-tree}} = \left\lfloor \frac{4 \times 1024 - 4}{2(8+2 \times 4)} \right\rfloor = 127$

Example

- What is the order of a B+-tree and a B-tree with a page size of 4 KB indexing keys of 8 bytes each, and having pointers of size 4 bytes?
  - $\Theta_{\text{B+-tree}} = \left\lfloor \frac{4 \times 1024 - 4}{2(8+4)} \right\rfloor = 170$
  - $\Theta_{\text{B-tree}} = \left\lfloor \frac{4 \times 1024 - 4}{2(8+2 \times 4)} \right\rfloor = 127$
- What is the height of the B+-tree and the B-tree for 3×10^7 keys?

Example

- What is the order of a B+-tree and a B-tree with a page size of 4 KB indexing keys of 8 bytes each, and having pointers of size 4 bytes?
  - $\Theta_{\text{B+-tree}} = \left\lfloor \frac{4\times1024-4}{2(8+4)} \right\rfloor = 170$
  - $\Theta_{\text{B-tree}} = \left\lfloor \frac{4\times1024-4}{2(8+2\times4)} \right\rfloor = 127$
- What is the height of the B+-tree and the B-tree for 3×10^7 keys?
  - Height of B+-tree is $\lceil \log_{2*170}(3 \times 10^7) \rceil = 3$

Example

- What is the order of a B+-tree and a B-tree with a page size of 4 KB indexing keys of 8 bytes each, and having pointers of size 4 bytes?
  - $\Theta_{\text{B+-tree}} = \left\lfloor \frac{4 \times 1024 - 4}{2(8+4)} \right\rfloor = 170$
  - $\Theta_{\text{B-tree}} = \left\lfloor \frac{4 \times 1024 - 4}{2(8+2 \times 4)} \right\rfloor = 127$
- What is the height of the B+-tree and the B-tree for 3×10^7 keys?
  - Height of B+-tree is $\lceil \log_{2*170}(3 \times 10^7) \rceil = 3$
  - Height of B-tree is $\lceil \log_{2*127}(3 \times 10^7) \rceil = 4$

Indexing Multiple Attributes

- Search keys having more than one attribute are called composite search keys
- Separate indices may be used
  - Union, intersection, etc. of individual results
- Multi-dimensional indexing
  - Indexing is specified by hyper-rectangles

Indexing Multiple Attributes

- Search keys having more than one attribute are called composite search keys
- Separate indices may be used
  - Union, intersection, etc. of individual results
- Multi-dimensional indexing
  - Indexing is specified by hyper-rectangles
  - Space-partitioning
  - Quadtree, KD-tree, K-d-B-tree: Extension of BST

Indexing Multiple Attributes

- Search keys having more than one attribute are called composite search keys
- Separate indices may be used
  - Union, intersection, etc. of individual results
- Multi-dimensional indexing
  - Indexing is specified by hyper-rectangles
  - Space-partitioning
  - Quadtree, KD-tree, K-d-B-tree: Extension of BST
  - Data-partitioning
  - R-tree: Extension of B+-tree
  - Uses minimum bounding rectangles (MBRs)

Bitmap Index

- Attribute domain consists of a small number of distinct values
- A bitmap or a bit vector is an array of bits
- Each distinct value has an array of the size of the number of tuples
  - If the *i*-th bit is 1, tuple *i* has that value

Gender	Grade
Male	C
Female	A
Female	C
Male	D
Male	A

- Two sets of bit vectors
  - Male = (10011), Female = (01100)
  - A = (01001), B = (00000), C = (10100), D = (00010)

Bitmap Operations

- Queries are answered using bitmap operations
- Example: Find the male student who got ‘D’
  - Bitmap(Male) AND Bitmap(D)
- Null values require a special bitmap for null

Bitmap Operations

- Queries are answered using bitmap operations
- Example: Find the male student who got ‘D’
  - Bitmap(Male) AND Bitmap(D)
- Null values require a special bitmap for null
- O/S allows efficient bitmap operations when they are packed in word sizes

Indices in SQL

- create [unique] index name on rel (att) creates an index *name* on attribute *att* of relation *rel*

```
create index idx_ctype on course (ctype);
```

Indices in SQL

- create [unique] index name on rel (att) creates an index *name* on attribute *att* of relation *rel*

```
create index idx_ctype on course (ctype);
```

- Index will be used whenever attribute *att* of relation *rel* is used

Indices in SQL

- create [unique] index name on rel (att) creates an index *name* on attribute *att* of relation *rel*

```
create index idx_ctype on course (ctype);
```

- Index will be used whenever attribute *att* of relation *rel* is used
- *unique* does not allow duplicates on the attributes

Indices in SQL

- create [unique] index name on rel (att) creates an index *name* on attribute *att* of relation *rel*

```
create index idx_ctype on course (ctype);
```

- Index will be used whenever attribute *att* of relation *rel* is used
- *unique* does not allow duplicates on the attributes
- Often, primary keys are *implicity* (by default) indexed

Indices in SQL

- create [unique] index name on rel (att) creates an index *name* on attribute *att* of relation *rel*

```
create index idx_ctype on course (ctype);
```

- Index will be used whenever attribute *att* of relation *rel* is used
- *unique* does not allow duplicates on the attributes
- Often, primary keys are *implicity* (by default) indexed
- Slows down modification operations as index is also modified

Indices in SQL

- create [unique] index name on rel (att) creates an index *name* on attribute *att* of relation *rel*

```
create index idx_ctype on course (ctype);
```

- Index will be used whenever attribute *att* of relation *rel* is used
- *unique* does not allow duplicates on the attributes
- Often, primary keys are *implicity* (by default) indexed
- Slows down modification operations as index is also modified
- drop index *i* deletes the index

```
drop index idx_ctype;
```