# Lecture 15: Indexing

CS348 Spring 2025: Introduction to Database Management

> Guest Lecture: **Chao Zhang** Sections: 001, 002, 003

#### Announcements

- Assignment 3 released on Learn
- Midterm exam tomorrow!
  - Time: 4:30 PM 6:00 PM
  - Location: M3 1006 and STC 0040 (check your room)
  - Coverage: Lectures 1 12
  - Content: relational model and relational algebra, SQL, database design
  - See the midterm-review lecture
- No class on next Tue, July 1 (Canada day)!

### Outline

#### • Types of indexes:

- Dense v.s. sparse
- Clustering v.s. non-clustering
- Primary v.s. secondary
- Indexing structure
  - ISAM
  - B+-tree
  - Hashing
- How to use index

### Static hashing



### Performance of static hashing

- Depends on the quality of the hash function
  - Best (hopefully average) case: one I/O
  - Worst case: all keys hashed into one bucket
  - See Knuth vol. 3 for good hash functions
    - Efficiency + uniformity (low collision)
- Rule of thumb: keep utilization at 50%-80%

How do we cope with growth?

#### Strawman solution

Rehash the whole table — using a new hash function, or at least changing *n* in mod *n* to the new number of buckets

- Entries in an old bucket may show up across many different new buckets, causing lots of I/Os
- Cost of (re)building a giant hash table on external storage ≈ sorting (later in this course)

Is it possible to reduce data movement?

### Extensible hashing

 Idea 1: have hash function h output a large number of bits, but only use the lowest i bits, and dynamically increase i as needed



- E.g.,  $h_4(k) = 1011$ , then  $h_5(k)$  is either 01011 or 11011  $\Rightarrow$  contents in one old bucket can only go to two new buckets!
- Problem: ++*i* doubles the number of buckets!
- Idea 2: use a directory
  - Only split overflowing buckets
  - But double the directory size
  - Many directory entries can point to the same bucket

Showing hashed values here for ease of understanding, but in reality, we store original key values



- Insert k with h(k) = 0101
- Bucket too full? Split (next slide)
  - Allowing some overflow is also fine (and sometimes necessary)



 ++local depth, redistribute contents, and ++global depth (double the directory size) if necessary







### Extensible hashing – deletions

Just the reverse of insertions

- If the bucket becomes too empty:
  - Merge with "sibling" (differing only on the leading bit)
    - Adjust any pointer from the directory as needed
  - – local depth
  - If possible, – global depth and half the directory
    - Invariant: global depth = max of all local depths

## Summary of extensible hashing

Pros:

- Handles growing/shrinking indexes
- No full reorganization

Cons:

- One more level of indirection through the directory
- Directory size still doubles/halves
- There are cases when doubling may not be enough



## Linear hashing

- No extra indirection through a directory
  - Fix the splitting/growth order
  - Use some extra math to figure out the right bucket
- Grow only when utilization (avg. # entries per bucket / max # entries per block) exceeds a given threshold

*n*: # of primary buckets (not counting overflow blocks)  $i = [\log_2 n]$ : # of hash bits in use (global depth) threshold = 85% (a range of the buckets may use i - 1 bits)

n = 2, i = 1



### Linear hashing example – 1

n = 2, i = 1



- Split the first bucket with the lowest depth it's always the bucket  $n 2^{\lfloor \log_2 n \rfloor}$  (0-based index)
  - Often not the bucket you are inserting into!
- File grows linearly at the end (hence the name)



### Linear hashing example – 2

n = 3, i = 2



### Linear hashing example – 3

*n* = 4, *i* = 2



Look up **1110** 

• Bucket **110** doesn't exist, so go to bucket **10** 

## Summary of linear hashing

Pros:

- Handles growing/shrinking indexes
- No full reorganization
- No extra level of indirection (beats extensible hashing)

Cons:

• Still has overflow chains, and may not split them right away because buckets must be split in sequence



#### Hash indexes vs. B<sup>+</sup>-trees

- Hashing is faster on average, but the worst case can be bad
- B<sup>+</sup>-tree's worst-case performance guarantees rely on fewer assumptions, and in practice these trees are not very tall
- Hashing destroys order, but B<sup>+</sup>-trees provide ordering and support range scans

We will come back to sorting vs. hashing again in query processing

### Outline

#### • Types of indexes:

- Dense v.s. sparse
- Clustering v.s. non-clustering
- Primary v.s. secondary
- Indexing structure
  - ISAM
  - B+-tree
  - Hashing
- How to use index

### Multi-attribute indices

- Index on several attributes of the same relation.
  - CREATE INDEX NameIndex ON User(LastName, FirstName);

tuples (or tuple pointers) are organized first by Lastname. Tuples with a common lastname are then organized by Firstname.

- This index would be *useful* for these queries:
  - select \* from User where Lastname = 'Smith'
  - select \* from User where Lastname = 'Smith' and Firstname='John'
- This index would be not *useful* at all for this query:
  - select \* from User where Firstname='John'

### Index-only plan

- For example:
  - SELECT firstname, pop FROM User WHERE pop > '0.8' AND firstname = 'Bob';
  - non-clustering index on (firstname, pop)
- A (non-clustered) index contains all the columns needed to answer the query without having to access the tuples in the base relation.
  - Avoid one disk I/O per tuple
  - The index is much smaller than the base relation

#### Physical design guidelines for indices

- Don't index unless the performance increase outweighs the update overhead
- Attributes mentioned in WHERE clauses are candidates for index search keys
- Multi-attribute search keys should be considered when a WHERE clause contains several conditions; or it enables index-only plans

#### Physical design guidelines for indices

- Choose indexes that benefit as many queries as possible
- Each relation can have at most one clustering scheme; therefore choose it wisely
  - Target important queries that would benefit the most
    - Range queries benefit the most from clustering
  - A multi-attribute index that enables an index-only plan does not benefit from being clustered

User (<u>uid</u> int, name string, age int, pop float) Group (<u>gid</u> string, name string, date DATE) Member (<u>uid</u> int, <u>gid</u> string)

- Common queries
  - 1. List the name, pop of users in a particular age range
  - 2. List the uid, age, pop of users with a particular name
  - 3. List the average pop of each age
  - 4. List all the group info, ordered by their starting date
  - 5. List the average pop of a particular group given the group name
- Pick a set of clustering/non-clustering indexes for these set of queries (without worrying too much about storage and update cost)

User (<u>uid</u> int, name string, age int, pop float) Group (<u>gid</u> string, name string, date DATE) Member (<u>uid</u> int, <u>gid</u> string)

A clustering index

on User(age)

- Common queries
  - 1. List the name, pop of users in a particular age range
  - 2. List the uid, age, pop of users with a particular name
  - 3. List the average pop of each age
  - 4. List all the group info, ordered by their starting date
  - 5. List the average pop of a particular group given the group name

A non-clustering index

on User(name)

Common q

User (<u>uid</u> int, name string, age int, pop float) Group (<u>gid</u> string, name string, date DATE) Member (<u>uid</u> int, <u>gid</u> string)

A non-clustering index on User(age, pop) → index-only plan

A clustering index A non-clustering index on User(age) on User(name)

- 1. List the name, populaters in a particular age range
- 2. List the uid, age, pop f users with a particular name
- 3. List the average pop of each age
- 4. List all the group info, ordered by their starting date
- 5. List the average pop of a particular group given the group name

Common q

User (<u>uid</u> int, name string, age int, pop float) Group (<u>gid</u> string, name string, date DATE) Member (<u>uid</u> int, <u>gid</u> string)

A non-clustering index on User(age, pop) → index-only plan

A clustering index A non-clustering index on User(age) on User(name)

- 1. List the name, populaters in a particular age range
- 2. List the uid, age, pop f users with a particular name
- 3. List the average pop of each age
- 4. List all the group info, ordered by their starting date
- 5. List the average pop of a particular group given the group name A clustering

A clustering index on Group(date)

Common q

User (<u>uid</u> int, name string, age int, pop float) Group (gid string, name string, date DATE) Member (uid int, gid string)

A non-clustering index on User(age, pop)  $\rightarrow$  index-only plan

A non-clustering index A clustering index on User(name) on User(age)

- List the name, populaters in a particular age range 1.
- List the uid, age, pop f users with a particular name 2.
- List the average pop of each age 3.
- List all the group info, ordered by their starting date 4.
- 5. List the average pop of a particular group given the group name A join between User(uid, ..., pop), Member(uid,gid), Group(gid, name)

A clustering index on Group(date)

(i) Search gid by a particular name  $\rightarrow$  Clustering/non-clustering index on Group(name)?

(ii) Search uid by a particular gid  $\rightarrow$  Clustering/non-clustering index on Member(gid)?

(iii) Search pop by a particular uid  $\rightarrow$  Clustering/non-clustering index on User(uid)?

Non-clustering, as we already have a clustered index on Group(date)

If many other queries need a clustering index on Group(name), we may reconsider!

Common q

User (<u>uid</u> int, name string, age int, pop float) Group (gid string, name string, date DATE) Member (uid int, gid string)

A non-clustering index on User(age, pop)  $\rightarrow$  index-only plan

A clustering index A non-clustering index on User(age) on User(name)

- users in a particular age range List the name, pop 1.
- List the uid, age, pop f users with a particular name 2.
- List the average pop of each age 3.
- List all the group info, ordered by their starting date 4.
- List the average pop of a particular group given the 5. group name A join between User(uid, ..., pop), Member(uid,gid), Group(gid, name)

A clustering index on Group(date)

(i) Search gid by a particular name → Non-clustering index on Group(name)	
<ul> <li>(ii) Search uid by a particular gid</li> <li>→ Clustering/non-clustering index on Member(gid)?</li> </ul>	Clustering -> all records of the same gid are clustered
<ul> <li>(iii) Search pop by a particular uid</li> <li>→ Clustering/non-clustering index on User(uid)?</li> </ul>	Or clustering index on Member(gid,uid)

Common q

User (<u>uid</u> int, name string, age int, pop float) Group (<u>gid</u> string, name string, date DATE) Member (<u>uid</u> int, <u>gid</u> string)

A non-clustering index on User(age, pop) → index-only plan

A clustering index A non-clustering index on User(age) on User(name)

- 1. List the name, populaters in a particular age range
- 2. List the uid, age, pop f users with a particular name
- 3. List the average pop of each age
- 4. List all the group info, ordered by their starting date
- 5. List the average pop of a particular group given the group name A join between User(uid, ..., pop), A clustering index OMERCIAL Member(uid, gid), Group(gid, name) A clustering index OMERCIAL MEMBERCIAL ACTION OF A CLUSTER O

(i) Search gid by a particular name
 → Non-clustering index on Group(name)

- (ii) Search uid by a particular gid
- → Clustering index on Member(gid)

(iii) Search pop by a particular uid
→ Clustering/non-clustering index on User(uid)?

Or non-clustering index on User(uid, pop) → index-only plan, if without worrying about storage/update cost

Non-clustering, as we already have a clustering index on User(age)

### Summary

- Types of indexes:
  - Dense v.s. sparse
  - Clustering v.s. non-clustering
  - Primary v.s. secondary
- Indexing structure
  - ISAM
  - B+-tree
  - Hashing
- How to use index
  - Use multi-attribute indices
  - Index-only plan
  - General guideline

#### What is next?

#### DBMS Internals:

- Storage: records, blocks, and files
- Indexing
- Query processing & optimization