INDEXING

CS 564- Fall 2016

ACKs: *Dan Suciu, Jignesh Patel, AnHai Doan*

File Organization

- So far we have see heap files
  - store unordered data
  - fast for scanning all records in a file
  - fast for retrieving by record id (rid)

- But we need alternative organizations of a file!

Motivation

- Consider the following SQL query:

```
SELECT *
FROM Sales
WHERE Sales.date = “02-11-2016”
```

- For a heap file, we have to scan all the pages of the file to return the correct result

Alternative File Organizations

- We can speed up the query execution by better organizing the data in a file
- There are many alternatives:
  - sorted files
  - indexes
    - B+ tree
    - hash index

INDEX BASICS

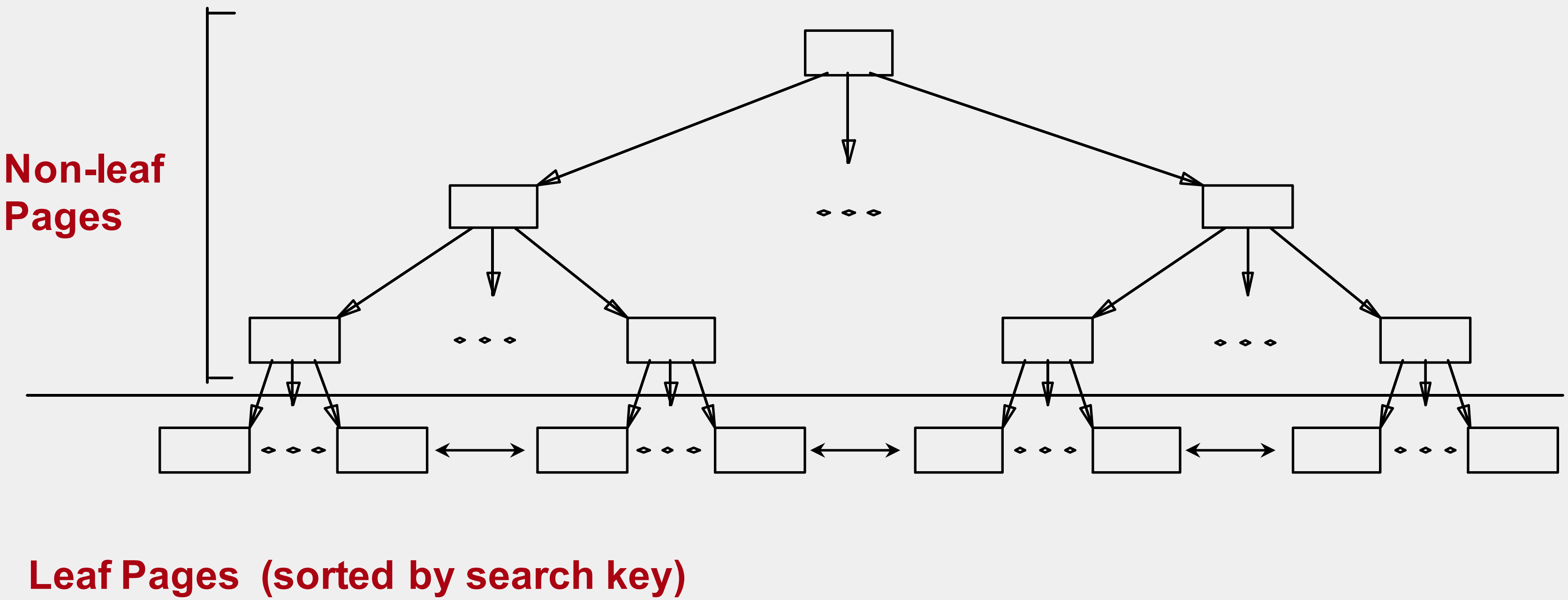
INDEXES

- **Index**: data structure that organizes records to optimize retrieval
  - speeds up searches for a subset of records, based on values in certain (*search key*) fields
  - any subset of the fields of a relation can be the search key
  - a search key is *not* the same as the primary key
- An index contains a collection of *data entries* (each entry with enough info to locate the records)

HASH INDEX

- A hash index is a collection of buckets
  - bucket = primary page plus overflow pages
  - buckets contain data entries
- uses a hash function **h**
  - ***h****(r)* = bucket in which (data entry for) record *r* belongs
- good for equality search
- not so good for range search (use tree indexes instead)

B+ Tree Index



- Leaf pages contain data entries, and are chained (prev & next)
- Non-leaf pages have data entries

DATA ENTRIES

- The actual data may not be in the same file as the index
- In a data entry with search key **k** we have 3 alternatives of what to store:
  1. the record with key value **k**
  2. <**k**, rid of record with search key value **k**>
  3. <**k**, list of rids of records with search key **k**>
- The choice of alternative for data entries is independent of the indexing technique

ALTERNATIVES FOR DATA ENTRIES

Alternative #1:

- index structure is a file organization for records
- at most one index on a given collection of data records should use #1 (why?)
- if data records are very large, the number of pages containing data entries is high (slower search)

ALTERNATIVES FOR DATA ENTRIES

Alternatives #2 and #3:

- Data entries are typically much smaller than data records. So, better than #1 with large data records, especially if search keys are small
- #3 is more compact than #2, but leads to variable sized data entries even if search keys are of fixed length

More on Indexes

- A file can have several indexes

- Index classification:
  - *primary* **vs** *secondary*
  - *clustered* **vs** *unclustered*

Primary vs Secondary

- If the search key contains the primary key, it is called a primary index
- Any other index is called a secondary index
- If the search key contains a candidate key, it is called a unique index
  - a unique index can return no duplicates

EXAMPLE

`Sales (sid, product, date, price)`

1. An index on (sid) is a primary and unique index
2. An index on (date) is a secondary, but not unique, index

CLUSTERED INDEXES

- If the order of records is the same as, or `close to', the order of data entries, it is a **clustered** index
  - alternative #1 implies clustered
  - in practice, clustered also implies #1
  - a file can be clustered on at most one search key
  - the cost of retrieving data records through the index varies greatly based on whether index is clustered or not

Indexes in Practice

CHOOSING INDEXES

- What indexes should we create?
  - which relations should have indexes?
  - what field(s) should be the search key?
  - should we build several or one index?
- For each index, what kind of an index should it be?
  - clustered
  - hash or tree

CHOOSING INDEXES

- Consider the best plan using the current indexes, and see if a better plan is possible with an additional index
- One must understand how a DBMS evaluates queries and creates query evaluation plans
- Important trade-offs:
  - queries go faster, updates are slower
  - more disk space is required

CHOOSING INDEXES

- Attributes in **`WHERE`** clause are candidates for index keys
  - exact match condition suggests hash index
  - indexes also speed up joins (later in class)
  - range query suggests tree index (B+ tree)

- Multi-attribute search keys should be considered when a **`WHERE`** clause contains several conditions
  - order of attributes is important for range queries
  - such indexes can enable index-only strategies for queries

COMPOSITE INDEXES

Composite search keys: search on a combination of fields (e.g. <date, price>)

- equality query: every field value is equal to a constant value
  – date="02-20-2015" and price =75
- range query: some field value is not a constant
  – date="02-20-2015"
  – date="02-20-2015" and price > 40

INDEXES IN SQL

```
CREATE INDEX index_name
ON table_name (column_name);
```

- Example of simple search key:

```
CREATE INDEX index1
ON Sales (price);
```

INDEXES IN SQL

```
CREATE UNIQUE INDEX index2
ON Sales (sid);
```

- A unique index does not allow any duplicate values to be inserted into the table
- It can be used to check integrity constraints (a duplicate value will not be allowed to be inserted)

Indexes in SQL

```
CREATE INDEX index3
ON Sales (date, price);
```

- Indexes with composite search keys are larger and more expensive to update
- They can be used if we have multiple selection conditions in our queries

RECAP

- Indexes
  - alternative file organization

- Index classifications:
  - hash vs tree
  - clustered vs unclustered
  - primary vs secondary