FILE ORGANIZATIONS

• So far we have seen heap files
  – unordered data
  – fast for scanning all records in a file
  – fast for retrieving by record id (rid)

• Do we need alternative organizations of a file?
 MOTIVATION 

- Consider the following SQL query:
  
  ```sql
  SELECT * 
  FROM Sales 
  WHERE Sales.date = "02-10-2015"
  ```

- What is the execution like for a heap file?
We can speed up query execution by better organizing the data in a file!

There are many alternatives:

- Sorted files
- Indexes
  - B+ tree
  - Hash index
INDEXES
INDEXES

• **Indexes**: data structures that organize records via trees or hashing
  – they speed 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
  – the search key is not the same as 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 $h$
  - $h(r) = \text{bucket in which (data entry for) record } r \text{ belongs}$
- Good for equality search
- Not so good for range search (use tree indexes instead)
**Hash Index Example**

Sales (sid, product, date, price)

On the blackboard!
**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 can store:
  1. the record with key value \( k \)
  2. \( <k, \text{rid of record with search key value } k> \)
  3. \( <k, \text{list of rids of records with search key } k> \)
- The choice of alternative for data entries is **orthogonal** to 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:
  – clustered vs unclustered
  – primary vs secondary
**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**
  - returns no duplicates
EXAMPLE

Sales (sid, product, date, price)

• An index on (sid) is a primary and unique index
• 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 index varies greatly based on whether index is clustered or not (why?)
EXAMPLE

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

On blackboard!
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 indexes?

• For each index, what kind of an index should it be?
  – clustered?
  – hash/tree?
CHOOSING INDEXES

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

• 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
INDEXES IN SQL

CREATE INDEX index_name
ON table_name (column_name);

• Example:
  
  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
• Can be used to check integrity constraints!
**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`
CREATE INDEX index3
ON Sales (date, price);
**Composite Keys**

- Composite indexes are larger and more expensive to update
- Can be used if we have multiple selection conditions