INDEXING

CS 564- Fall 2021

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

WHAT IS THIS LECTURE ABOUT?

- Indexes
  - what is an index?
- Index classifications:
  - clustered vs unclustered
  - primary vs secondary

FILE ORGANIZATION: RECAP

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

- But we also need alternative organizations of a file to support other access patterns

MOTIVATING EXAMPLE

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

```
SELECT *
FROM Sales
WHERE Sales.date = "02-11-2019"
```

equality query

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

MOTIVATING EXAMPLE

Sales (sid, product, date, price)

```
SELECT *
FROM Sales
WHERE Sales.date > "01-01-2019"
AND Sales.date <= "01-31-2019"
```

range query

Here we also need to scan all the pages of the file!

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
    - bitmap index
    - …

INDEX BASICS

WHAT IS AN INDEX?

Index: a data structure that organizes records of a table to speed up retrieval

Search Key: attribute or combination of attributes used to retrieve the records

- any subset of the attributes can be the search key
- a search key is *not* the same as the primary key!

EXAMPLE: HASH INDEX

`Person(name,zipcode,phone)`

- *search key*: {zipcode}
- *hash function* ***h***: zipcode modulo 4

primary pages

bucket 0
(John, **53400**, 23218564)
(Alice, **54768**, 60743111)

bucket 1
(Mike, **53409**, 23200564)

bucket 2

bucket 3
(Maria, **34411**, 29010533)

To find the bucket for each record, we apply a hash function ***h*** on the search key.

INDEX ENTRIES

Index Entry: for a value **k** of the search key, it can take two forms:

#1: the record with key value **k**

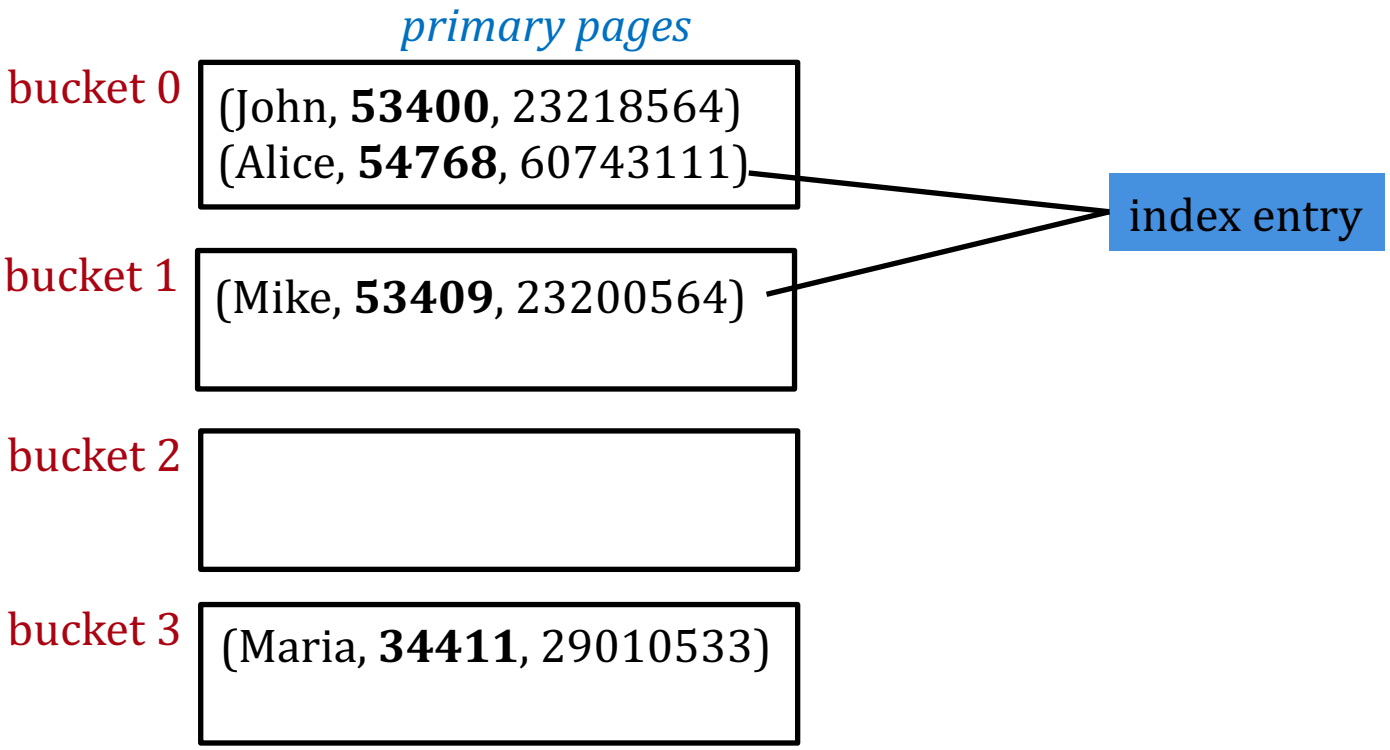
#2: <**k**, rid of record with search key value **k**>

The choice of alternative for an index entry is independent of the indexing technique!

INDEX ENTRY #1

Person(name,zipcode,phone)

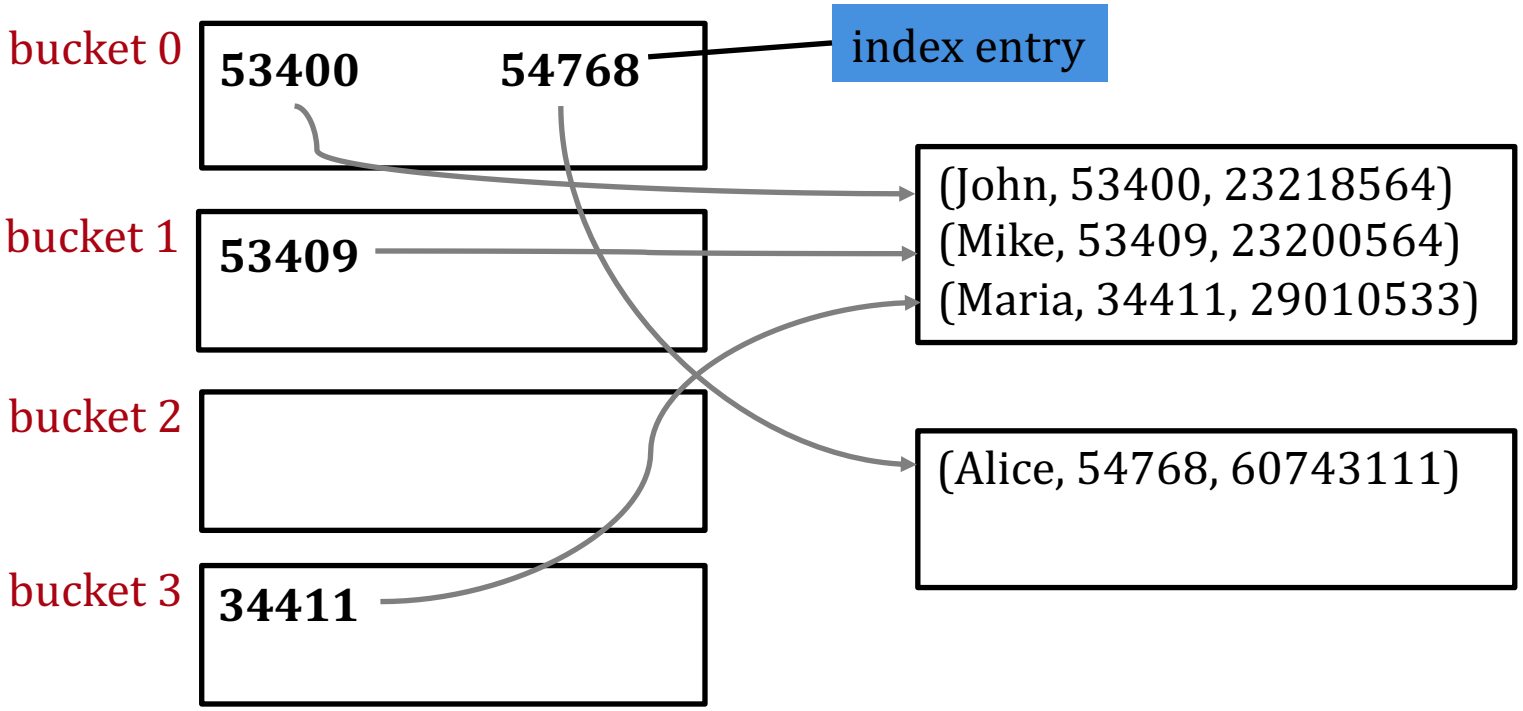
- *search key*: {zipcode}
- *hash function* ***h***: zipcode modulo 4



INDEX ENTRY #2

Person(name,zipcode,phone)

- *search key*: {zipcode}
- *hash function* ***h***: zipcode modulo 4



ALTERNATIVES FOR INDEX ENTRIES

#1: *the index entry contains the record*

- the index structure is *by itself* a file organization for records
- *at most one* index on a given table can use #1
- if records are very large, the number of pages containing index entries is high
  - this means possibly slower search!

ALTERNATIVES FOR INDEX ENTRIES

#2: *the index entry contains the rid*

- Index entries are typically much smaller than data records
- better than #1 with large data records, especially when the search key is small

MORE ON INDEXES

A file can have several indexes, on different search keys!

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
  - in a primary index, there are no duplicates for a value of the search key
  - there can only be one 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 also return 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

COMPOSITE INDEXES

Composite search key: search on a combination of attributes

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

- An index on {date, price} is a composite index
- Order matters: {price, date} is a different index!

CLUSTERED INDEXES

Clustered index: the order of records matches the order of index entries

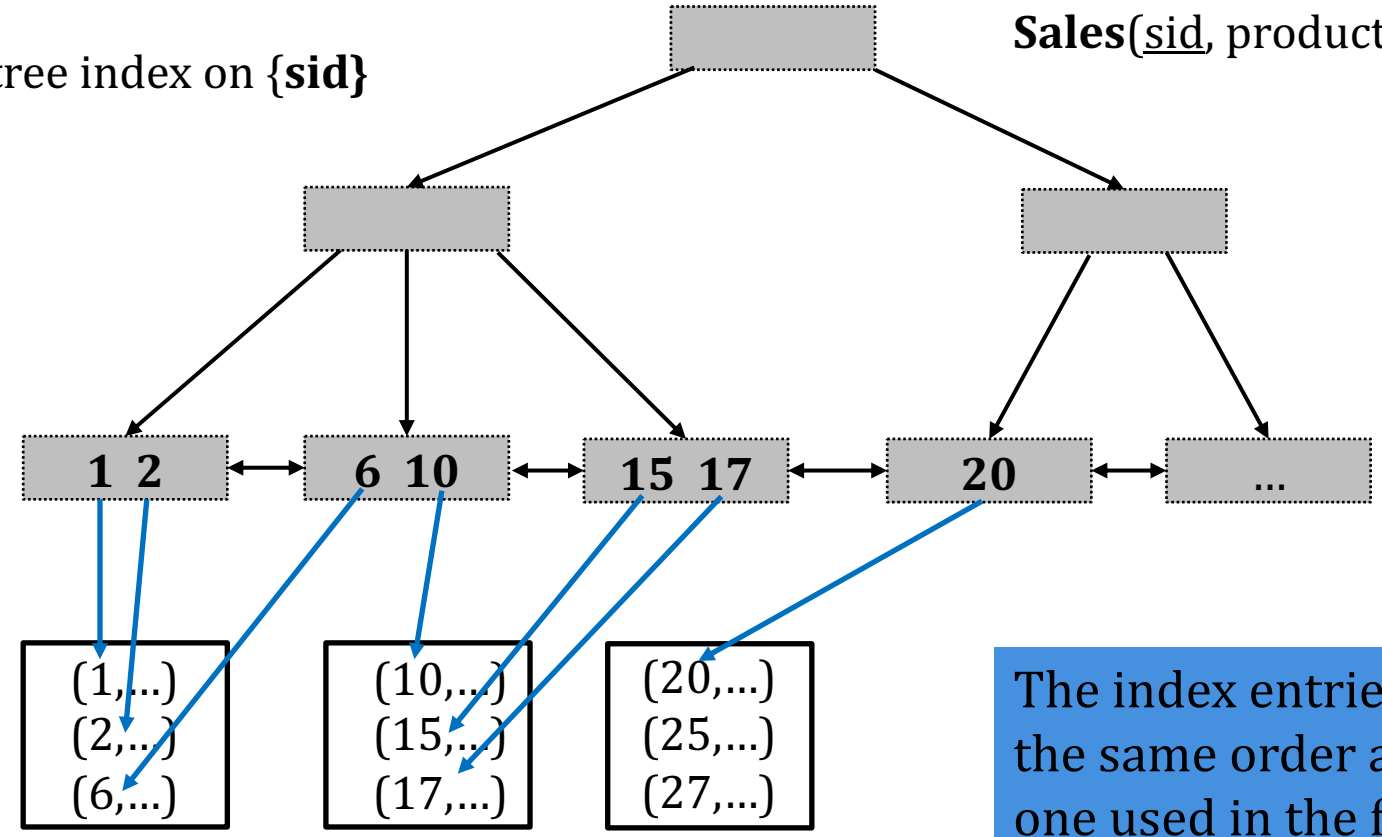
- alternative #1 implies that the index is clustered
- a table can have at most one clustered index
- the cost of retrieving data records through the index varies greatly based on whether index is clustered or not

logical order of index ~ physical order of records

EXAMPLE: CLUSTERED INDEX

B+ tree index on {**sid**}

Sales(<u>sid</u>, product, date, price)



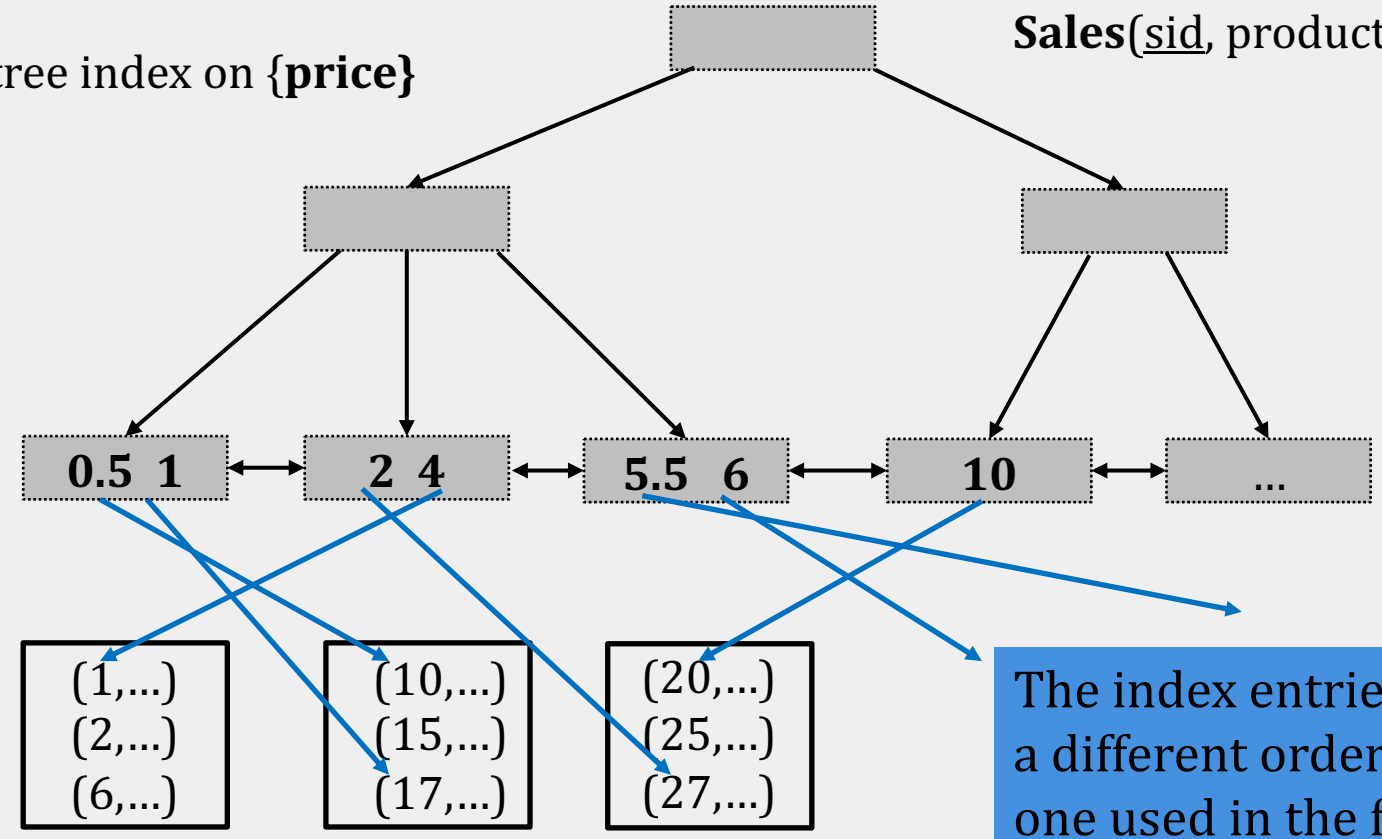
The index entries follow the same order as the one used in the file

The file is sorted by {**sid**}

EXAMPLE: NON-CLUSTERED INDEX

B+ tree index on {**price**}

Sales(<u>sid</u>, product, date, price)



The index entries follow a different order to the one used in the file

The file is sorted by {**sid**}

HOW DO WE EVALUATE AN INDEX?

- What access types does it support?
  - *e.g.* equalitiy search, range search, etc.
- Time to access a record
- Time to insert a record
- Time to delete a record
- How much space does it use?

INDEXES IN PRACTICE

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 efficiently 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

CHOOSING INDEXES

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

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
- Trade-off as we add more indexes:
  - 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
  - the order of attributes is important for range queries
  - such indexes enable index-only strategies for queries