MLDS 413, Nikos Hardavellas
Fall 2023

Indexing

Lingzhi Wang
TA section 8

1 Database Indices

Indices: Employ auxiliary data structures to speed up searches for a subset of records

- Based on values in certain ('search key') fields
  - Search key is not the same as key–minimal set of fields that uniquely identify a record in a relation
  - Any subset of the fields of a relation can be the search key for an index on the relation.
- In general, an index supports efficient retrieval of data entries that satisfy a given selection condition

> If we have a B+ tree index on *age*, we can use it to retrieve only tuples that satisfy the selection $E.age > 40$. However this enhancement may be worthless. Consider the fraction of employee older than 40. If basically everyone is older than 40, we gain little by using an index on *age*. However, if say 10% of employees are older than 40 then an index would be more useful.

```
SELECT E.dno
FROM Employees E
WHERE E.age > 40
```

1.1 Binary Trees and B+ Trees

Binary Tree Visualizer: https://www.cs.usfca.edu/~galles/visualization/BST.html
B+ Tree Visualizer: https://www.cs.csub.edu/~msarr/visualizations/BPlusTree.html

1.2 Selection of Indices

- The existence of an index on an attribute may speed up the execution of those queries in which a value, or range of values, is specified for that attribute, and may speed up joins involving that attribute as well
- On the other hand, every index built for one or more attributes of some relation makes insertions, deletions, and updates to that relation more complex and time-consuming

1.3 Composite Indices

The search key for an index can contain several fields; such keys are called **composite search keys**. Consider a collection of employee records, with fields *name*, *age*, and *sal* stored in sorted order by *name*. The figure below illustrates the difference between a composite index with key $< age, sal >$, a composite index with key $< sal, age >$, an index with key *age*, and an index with key *sal*

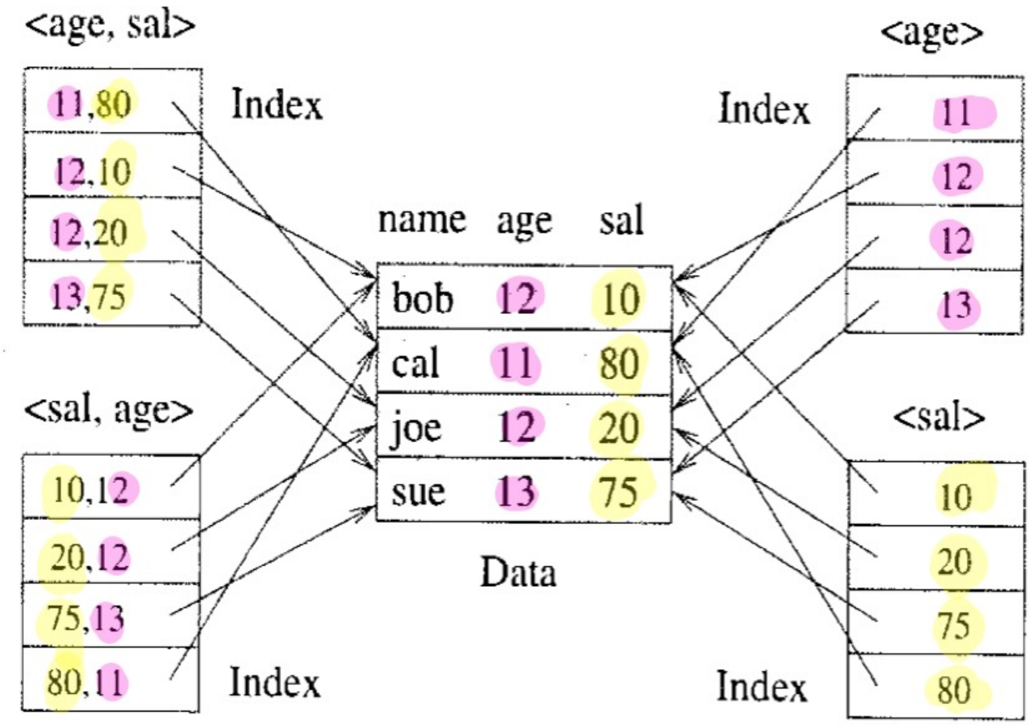

Figure 8.5 Composite Key Indexes

- we can use $< age, sal >$ to answer equality queries such as $age = 20$ and $sal = 10$.
- we can use $< age, sal >$ to retrieve all data entries with $age = 20$; implicitly this says that any value is acceptable for the *sal* field.
- we can use $< age, sal >$ to answer a range query such as $age < 30$
- $< age, sal >$ cannot help on query $sal > 40$, because, intuitively, *the index organizes records by age first and then sal*. If *age* is left unspecified, qualifying records could spread across the entire index.

1.3.1 Design Examples of Composite Keys

Consider the following query, which returns all employees with $20 < age < 30$ and $3000 < sal < 5000$

```
SELECT E.eid
FROM Employees E
WHERE E.age BETWEEN 20 and 30
      AND E.sal BETWEEN 3000 and 5000
```

A composite key on *sal* and *age* would be effective if the conditions in the `WHERE` clause are fairly selective. For this query a composite key of $< age, sal >$ or $< sal, age >$ are equally effective since the conditions on *age* and *sal* are equally selective. However, in the next example the order of search key attributes make a big difference

Consider the following query, which returns all employees with $age = 25$ and $3000 < sal < 5000$

```
SELECT E.eid
FROM Employees E
WHERE E.age = 25
        AND E.sal BETWEEN 3000 and 5000
```

In this query a composite key on $< age, sal >$ will give good performance because records are sorted by *age* first and then *sal* (if two records have the same *age* by *sal*). Thus, all records with $age = 25$ are clustered together. On the other hand, a index on $< sal, age >$ will not perform as well. In this case, record are sorted by *sal* first, and therefore two records with the same *age* value (in particular, $age = 25$) may be far apart. In effect, this erroneous index allows use of the range selection on *sal*, but not the equality selection on *age*

MLDS 413, Nikos Hardavellas
Fall 2023

Indexing

Lingzhi Wang
TA section 8

Consider the following query, which returns the average salary of all employees with $age = 25$ and $3000 < sal < 5000$

```
SELECT AVG (E.sal)
FROM Employees E
WHERE E.age = 25
    AND E.sal BETWEEN 3000 AND 5000
```

A index on $< age, sal >$ would allow us to answer the query efficiently. A index on $< sal, age >$ also allows us to answer this query, however it would not be as efficient as an index on $< age, sal >$. Notice that this query can be answered with an index-only scan. Index-only scans can be a really effective way to speed up table reads that hit an index. Of course, they're not a silver bullet to all your performance problems, but they are a very welcome and useful part of the toolbox. [a]

[a]

- **Index scan** reads through the index and uses it to quickly look up the rows that match your filter.
- **Index-only scans** start off like index scans, but they get all their column information from the index, obviating the need to go back to the table to fetch the row data — the second step in the index scan process.

MLDS 413, Nikos Hardavellas
Fall 2023

Indexing

Lingzhi Wang
TA section 8

Consider the following query, which returns for each department the number of employees with salaries equal to 10,000.

```
SELECT     E.dno, COUNT(*)
FROM       Employees E
WHERE      E.sal=10,000
GROUP BY E.dno
```

- An index on *dno* alone does not allow us to evaluate this query with an index-only scan, because we need to look at the *sal* field of each tuple to verify that $sal = 10,000$. However, we can use an index-only scan if we have a composite index on $< sal, dno >$ or $< dno, sal >$. In $< sal, dno >$, all data entries with $sal = 10,000$ are arranged contiguously, further, these entries are sorted by *dno*, making it easy to obtain a count for each *dno* group. Note that we need to retrieve only data entries with $sal = 10,000$

- This strategy does not work if the `WHERE` clause is modified to use $sal > 10,000$. An index with $< dno, sal >$ would be better for this query, since data entries with a given *dno* value are stored together, and each such group of entries is itself sorted by *sal*. For each *dno* group, we can eliminate the entries with *sal* not greater than 10000, and count the rest.

- Notice that this index is less efficient than the previous one,$< sal, dno >$, for the query with $sal = 10000$ because we must read all data entries, and so the question can not be answered with index only scan. *Thus, the choice between Indices is influenced by which query is more common*