**B+ Tree as “Sorted Access Path”**

- **Scenario:** Table to be retrieved in some order has a B+ tree index on the ordering column(s).
- **Idea:** Retrieve records in order by traversing the B+ tree’s leaf pages.
- **Q: Is this a good idea?**
- **Cases to consider:**
  1. B+ tree is clustered → **Good idea!**
  2. B+ tree is not clustered → **Can be a very bad idea!**
Case 1: Clustered B+ Tree

- Cost: Descend to the left-most leaf, then just scan leaves, as they contain the records. (Alternative 1)
- If rids (Alternative 2) are used? Additional cost to fetch data records, but each page fetched just once thanks to clustering.

(always better than external sorting)
Case 2: Unclustered B+ Tree

- Implies Alternative (2) for data entries; each entry contains RID of a data record. In general, will be doing one I/O per data record!

Trick: Scanning for the RIDs 1st and sorting before using them can help...!
Index Selection: Workload Analysis

- For each query in the workload:
  - Which relations does it access?
  - Which attributes are retrieved?
  - Which attributes appear in selection/join conditions, and how selective are these conditions (few vs. many results)?

- For each update in the workload:
  - Which attributes are involved in selection/join conditions? Again, how selective are these conditions likely to be?
  - What is the type of update (INSERT/DELETE/UPDATE), and which attributes are affected?
Choice of 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 a given file/table?

- For each index, what kind of index should it be?
  - Clustered? Hashed? B+ tree? ...?
Choice of Indexes (cont.)

- **One approach:** Consider the most important queries in turn. Consider the best query evaluation plan given the current indexes, and see if a better plan is possible with an additional index. If so, create it.
  - Obviously, this implies that we must understand how a DBMS evaluates queries and creates its query plans!
  - For today, we’ll focus mostly on simple 1-table queries.

- Before creating an index, must also consider its potential impact on updates in the workload!
  - **Trade-off:** Indexes can make queries go faster, but make updates go slower. They require disk space, too.
Index Selection Guidelines

- Attributes in WHERE clause are candidates for index keys.
  - Exact match condition suggests a hashed index.
  - Range query suggests a tree index.
    • Clustering especially useful for range queries; might also help on equality queries if there are many duplicates. (Q: See why?)

- 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 actually enable index-only query plans for important queries.
    • Note: For index-only strategies, clustering is not important!!

- Try to choose indexes that will benefit as many queries as possible. Since only one index per relation can be clustered, choose it based on important queries that can benefit the most from clustering.
Examples of Clustered Indexes

- B+ tree index on E.age can be used to get qualifying tuples.
  - How selective is the condition?
  - And is the index clustered?
- Consider the GROUP BY query.
  - If many tuples have E.age > 10, using E.age index and sorting the retrieved tuples (for grouping) may be costly.
  - Clustered E.dno index may be better!
- Equality with duplicates:
  - Clustering on E.hobby helps!

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

```
SELECT E.dno, COUNT (*)
FROM Emp E
WHERE E.age>10
GROUP BY E.dno
```

```
SELECT E.dno
FROM Emp E
WHERE E.hobby='Stamps'
```
Indexes with Composite Search Keys

- **Composite Search Keys**: Search on a combination of fields.
  - **Equality query**: Every field value is equal to a constant value. E.g. wrt <sal,age> index:
    - age=20 and sal=75
  - **Range query**: Some field value is a range, not a constant. E.g. again wrt <sal,age> index:
    - age=20; or age=20 and sal > 10

- Data entries in index sorted by search key(s) to support such range queries.
  - Lexicographic order

Various possible composite key indexes using lexicographic order.
Composite Search Keys

- To retrieve Emp records with $age=30$ AND $sal=4000$, an index on $<age,sal>$ would be better than an index on $age$ or $sal$ alone.
  - Choice of index key orthogonal to clustering, etc.
- If condition is: $20<age<30$ AND $3000<sal<5000$:
  - Clustered tree index on $<age,sal>$ or $<sal,age>$ is best.
- If condition is: $age=30$ AND $3000<sal<5000$:
  - Clustered $<age,sal>$ index much better than $<sal,age>$ index! (See why?)
- Composite indexes generally larger, and may be updated more often.
Index-Only Query Plans

- A number of queries can be answered w/o retrieving any (!) tuples from one or more of the relations involved, if a suitable index is available.

- \[
\begin{align*}
\text{SELECT } & \text{E.dno, COUNT(\ast)} \\
\text{FROM } & \text{Emp E} \\
\text{GROUP BY } & \text{E.dno}
\end{align*}
\]

- \[
\begin{align*}
\text{SELECT } & \text{E.dno, MIN(E.sal)} \\
\text{FROM } & \text{Emp E} \\
\text{GROUP BY } & \text{E.dno}
\end{align*}
\]

- \[
\begin{align*}
\text{SELECT } & \text{E.dno, AVG(E.sal)} \\
\text{FROM } & \text{Emp E} \\
\text{WHERE } & \text{E.age=25 AND E.sal BETWEEN 3000 AND 5000}
\end{align*}
\]
Here, index-only plans possible if the key is <dno, age> or we have a tree index with key <age, dno>:
- What is in each index entry?
- Which is better?
- What if we consider the second query?

```
SELECT E.dno, COUNT (*)
FROM Emp E
WHERE E.age=30
GROUP BY E.dno
```

```
SELECT E.dno, COUNT (*)
FROM Emp E
WHERE E.age>30
GROUP BY E.dno
```
Index-Only Plans (cont.)

- Index-only plans can also be found for some queries involving more than one table; more about that later on.

```sql
SELECT D.mgr FROM Dept D, Emp E WHERE D.dno=E.dno
```

(Q: English translation?)

```sql
SELECT D.mgr, E.eid FROM Dept D, Emp E WHERE D.dno=E.dno
```

```sql
<E.dno> index
```

```sql
<E.dno,E.eid> index
```
Files and Indexes: Summary

- Many alternative file organizations exist, with each being appropriate in some situations.
- If selection queries are frequent, building an index is important to avoid file scans.
  - Hash-based indexes (only) good for equality search. (Though could still scan index leaves...)
  - Sorted files and tree-based indexes best for range search; also good for equality search. (Files rarely kept sorted in practice; B+ tree index is better.)
- Index is a collection of data entries plus a way to quickly find entries with given key values.
Data entries can be actual data records, \(<\text{key, rid}>\) pairs, or \(<\text{key, rid-list}>\) pairs.

- Choice orthogonal to indexing technique used to locate data entries with a given key value.

- Can have several indexes on a given file of data records, each with a different search key.

- Indexes can be classified as clustered vs. unclustered, primary vs. secondary, and dense vs. sparse. Differences have important consequences for index utility/performance.
Summary (cont.)

- Understanding the nature of the *workload* for the application, and the performance goals, is essential to good physical DB design (*a.k.a.* index selection).
  - What are the important queries and updates? What attributes/relations are involved?
- Indexes must be chosen to speed up important queries (and perhaps some updates too!).
  - Maintenance overhead on updates to indexed fields.
  - Choose indexes that help many queries, if possible.
  - Build indexes in support of index-only strategies.
  - Clustering is an important decision; only one index on any given relation can be clustered!
  - Order of fields in composite keys can be very important.