INDEXES

MICHAEL LIUT (LIUTM@MCMASTER.CA)
DEPARTMENT OF COMPUTING AND SOFTWARE
MCMASTER UNIVERSITY

SE 3DB3 (Slides adapted from Dr. Fei Chiang) Fall 2016

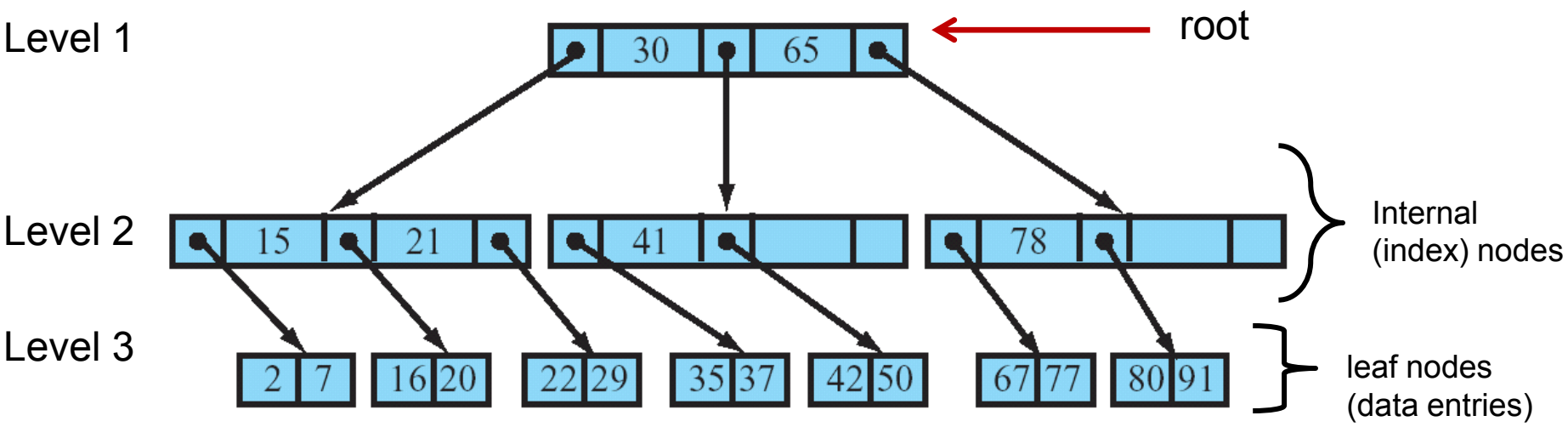
An Index

- Data structure that organizes records via trees or hashing
  - They speed up searches for a subset of records, based on values in certain ("search key") fields
- Given a value *v*, the index takes us to only those tuples that have *v* in the attribute(s) of the index.

- Example: use BeerInd (on manf) and SellInd (on bar, beer) to find the prices of beers manufactured by Pete's and sold by Joe.

B+ Tree Index

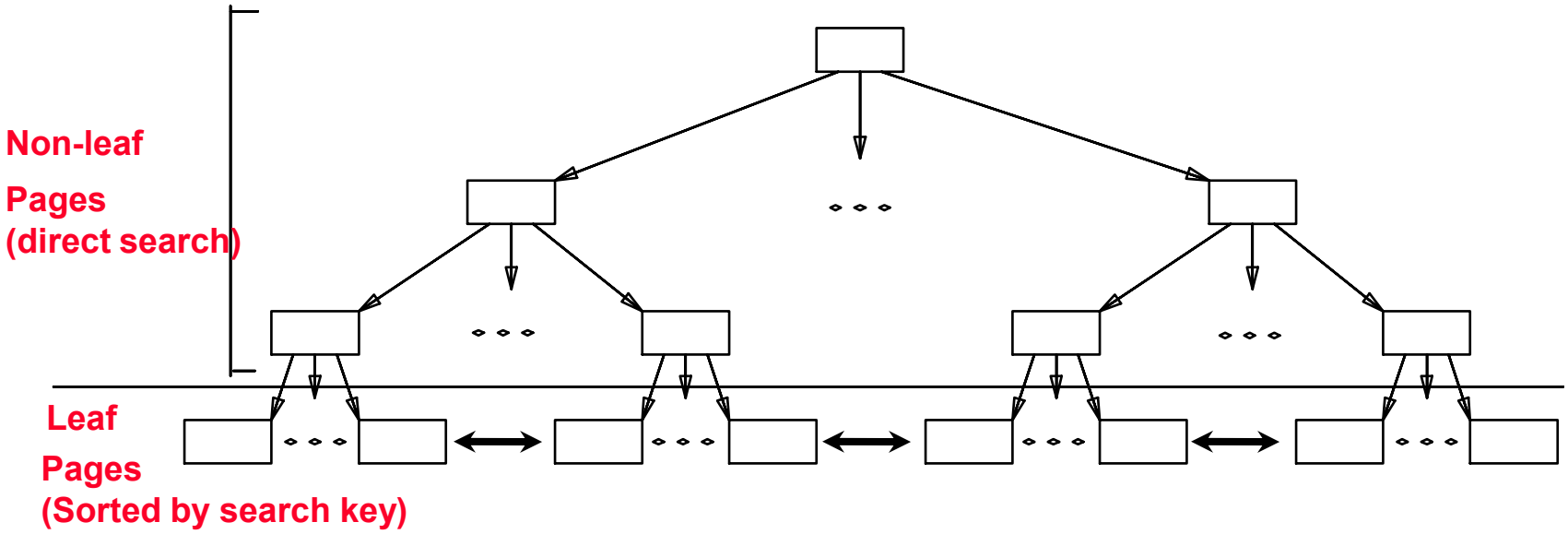
- The B+ tree structure is the most common index type in databases today.
- Index files can be quite large, often stored on disk, partially loaded into memory as needed
- Each node is at least 50% full

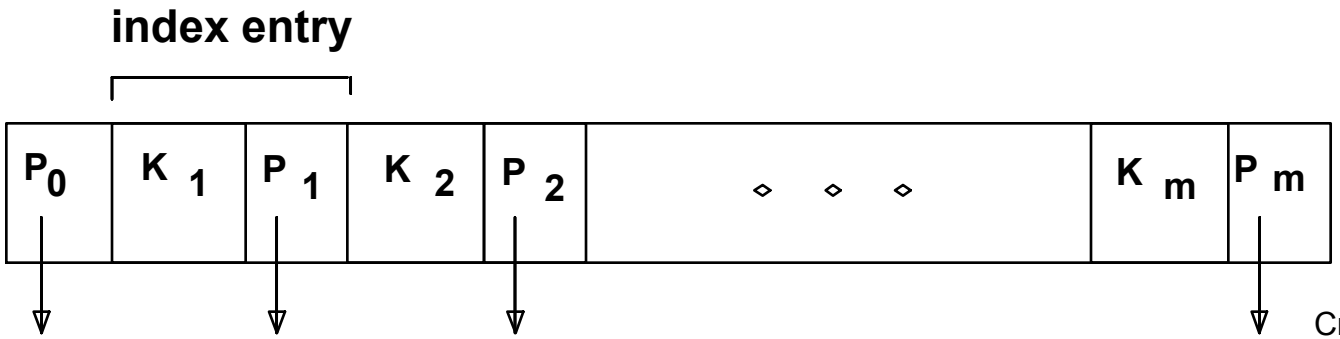


Credit: S. Lee

B+ Tree Index

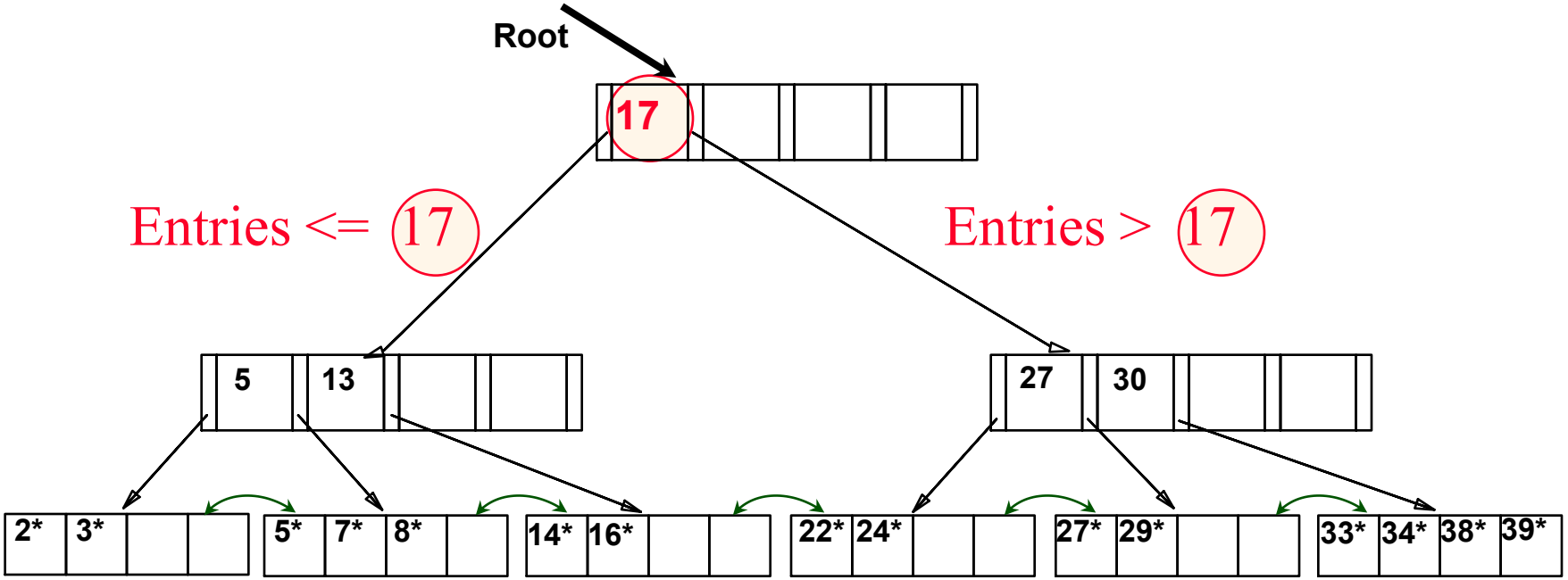
Supports equality and range-searches efficiently





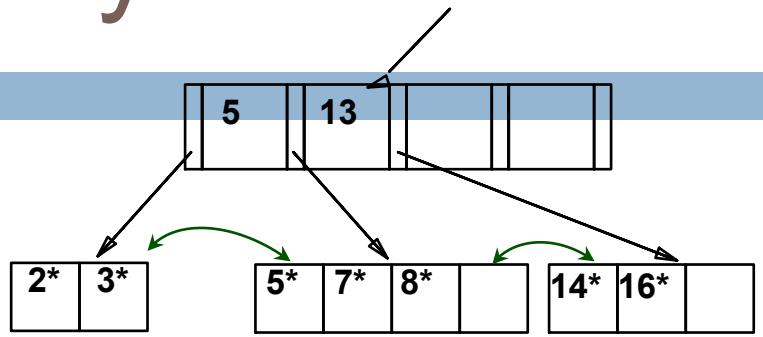
Credit: Renee Miller

Example



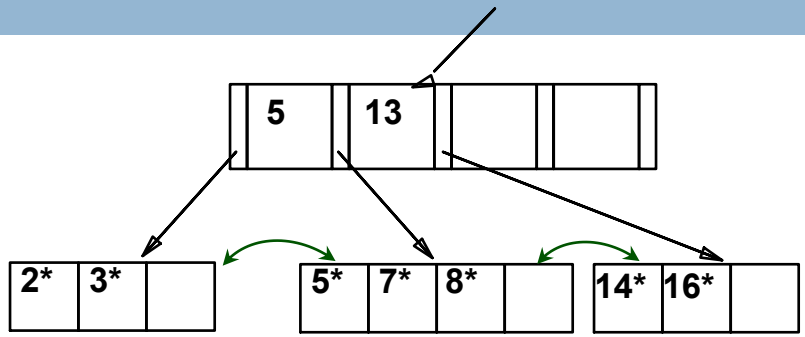
- Find 28*? 29*? All > 15* and < 30*
- Insert/delete: Find data entry in leaf, then change it. Need to adjust parent sometimes.
  - And change sometimes bubbles up the tree

Inserting a Data Entry



- Find correct leaf L.
- Put data entry onto L.
  - If L has enough space, *done*!
  - Else, must *split* L *(into L and a new node L2)*
    - Redistribute entries evenly, **copy up** middle key.
    - Insert index entry pointing to $L2$ into parent of L.
- This can happen recursively
  - To split index node, redistribute entries evenly, but **push up** middle key.
- Splits "grow" tree; root split increases height.

Deleting a Data Entry

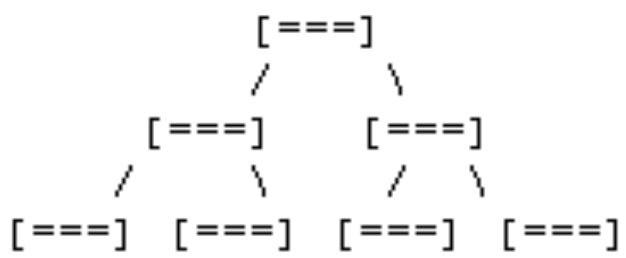


- Start at root, find leaf L where entry belongs.
- Remove the entry.
  - If L is at least half-full, *done!*
  - If not,
    - Try to re-distribute, borrowing from *sibling* (*adjacent node with same parent as L*).
    - If re-distribution fails, *merge* L and sibling.
- If merge occurred, must delete entry (pointing to L or sibling) from parent of L.
- Merge could propagate to root, decreasing height.

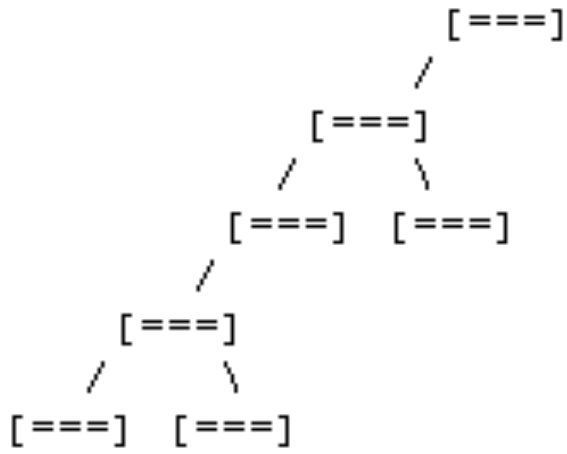
Balanced vs. Unbalanced Trees

- In a balanced tree, every path from the route to a leaf node is the same length.

- Balanced



- Unbalanced



Credit: S. Lee

Prefix Key Compression

- Key values in index entries only `direct traffic'; can often compress them.
  - E.g., If we have adjacent index entries with search key values *Dannon Yogurt*, *David Smith* and *Devarakonda Murthy*, we can abbreviate *David Smith* to *Dav*. (The other keys can be compressed too ...)

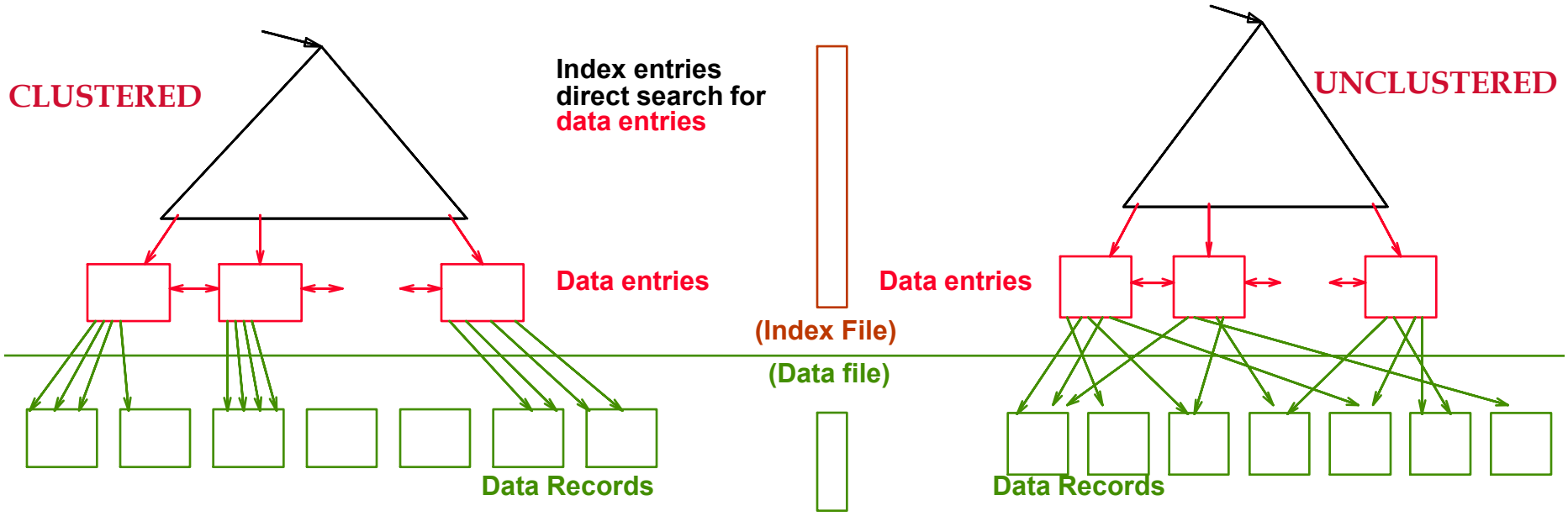
Hash Based Indexes

- Good for equality selections.
- Index is a collection of *buckets*.
  - Bucket = *primary* page plus zero or more *overflow* pages.
  - Buckets contain data entries.
- *Hashing function* **h**: **h**(r) = bucket in which (data entry for) record r belongs. **h** looks at the *search key* fields of r.
  - *No need for "index entries" in this scheme.*

Index Classification

- *Primary* vs. *secondary*: If search key contains primary key, then called primary index.
  - *Unique* index: Search key contains a candidate key.
- *Clustered* vs. *unclustered*: If order of index data entries is the same as order of data records, then called clustered index.
  - A file can be clustered on at most one search key.

Clustered vs. Unclustered Index



Understanding the Workload

- For each query in the workload:
  - Which relations does it access?
  - Which attributes are retrieved?
  - Which attributes are involved in selection/join conditions? How selective are these conditions likely to be?
- For each update in the workload:
  - The type of update (INSERT/DELETE/UPDATE), and the attributes that 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 each index, what kind of an index should it be?
  - Clustered? Hash/tree?

Choice of Indexes (cont'd)

- One approach: Consider the most important queries in turn. Consider the best plan using the current indexes, and see if a better plan is possible with an additional index.
  - Implies an understanding of how a DBMS evaluates queries and creates query evaluation plans.
- Before creating an index, must also consider the impact on updates in the workload!
  - Trade-off: Indexes can make queries go faster, updates slower. Require disk space, too.

Guidelines

- Attributes in WHERE clause are candidates for index keys.
  - Exact match condition suggests hash index.
  - Range query suggests tree index.
- Multi-attribute search keys should be considered when a WHERE clause contains several conditions.
- Try to choose indexes that benefit as many queries as possible.
  - Since only one index can be clustered per relation, choose it based on important queries that would benefit the most from clustering.

Examples

- B+ tree index on E.age can be used to get qualifying tuples.
  - Is the index clustered?

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

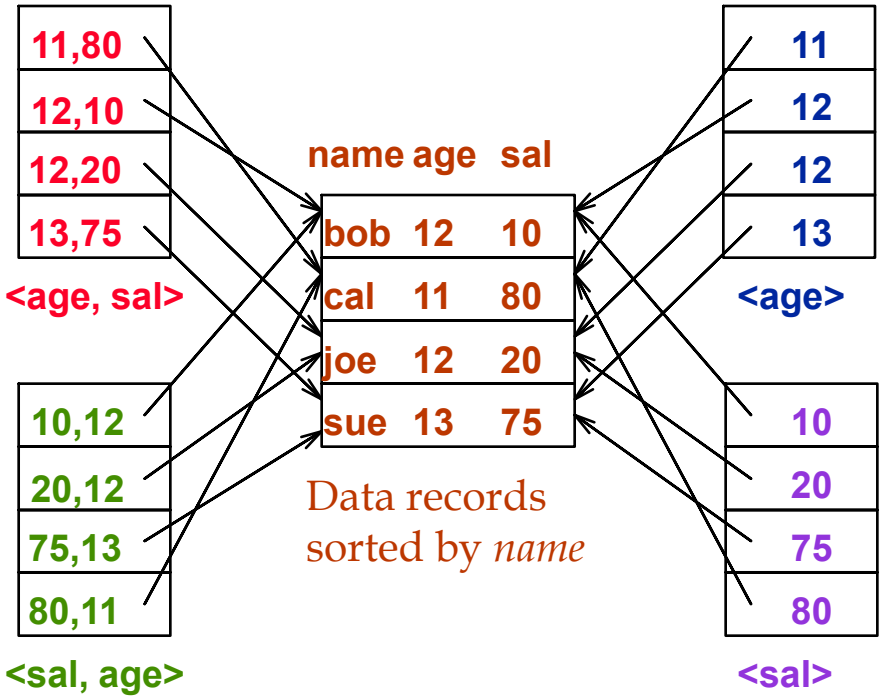
- Equality queries and duplicates:
  - Indexing on *E.hobby* helps

```
SELECT  E.dno
FROM  Emp E
WHERE  E.hobby=Stamps
```

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:
    - age=20 and sal > 10
- Data entries in index sorted by search key to support range queries.
  - Lexicographic order

Examples of composite key indexes using lexicographic order.



Database Tuning

- A major problem in making a database run fast is deciding which indexes to create.
- Pro: An index speeds up queries that can use it.
- Con: An index slows down all modifications on its relation because the index must be modified too.

Example: Tuning

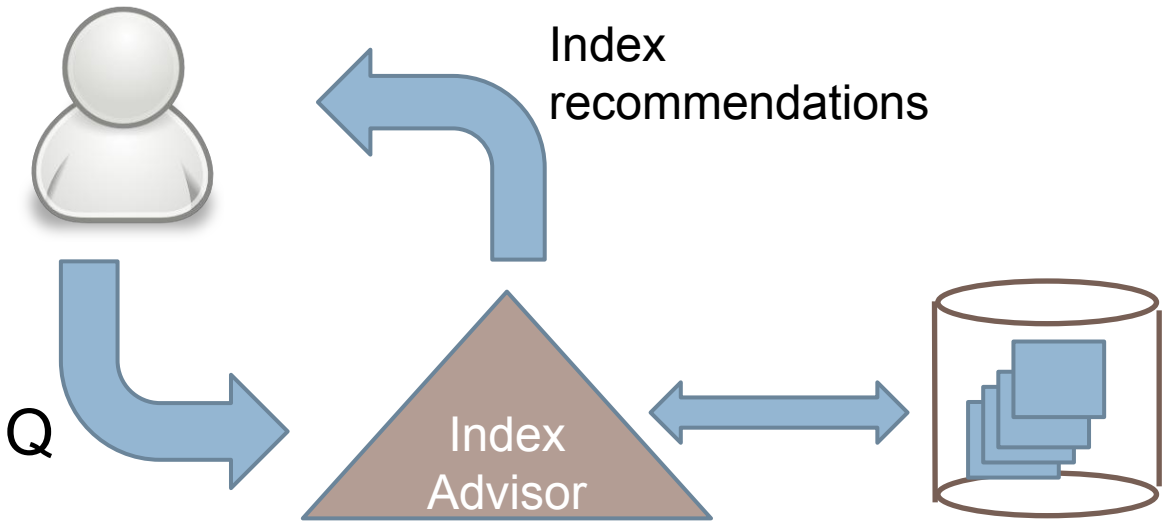
- Suppose the only things we did with our beers database was:
  1. Insert new facts into a relation (10%).
  2. Find the price of a given beer at a given bar (90%).
- Then SellInd on Sells(bar, beer) would be wonderful, but BeerInd on Beers(manf) would be harmful.

Tuning Advisors

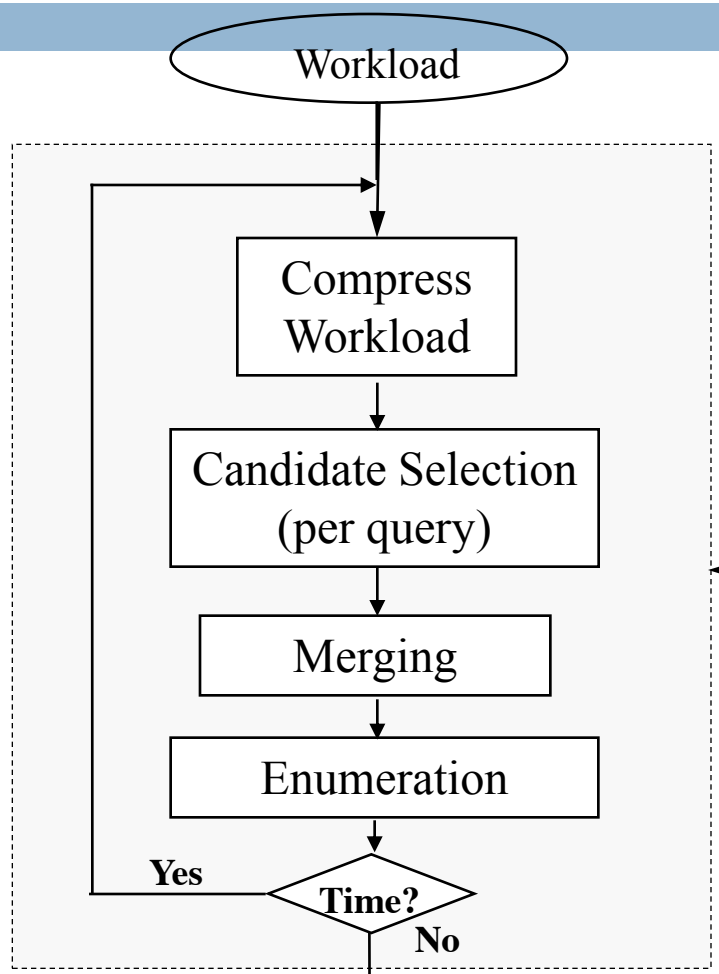
- A major research area
  - Because hand tuning is so hard.
- An advisor gets a *query load*, e.g.:
  1. Choose random queries from the history of queries run on the database, or
  2. Designer provides a sample workload.

Tuning Advisors --- (2)

- The advisor generates candidate indexes and evaluates each on the workload.
  - Measure the improvement/degradation in the average running time of the queries.

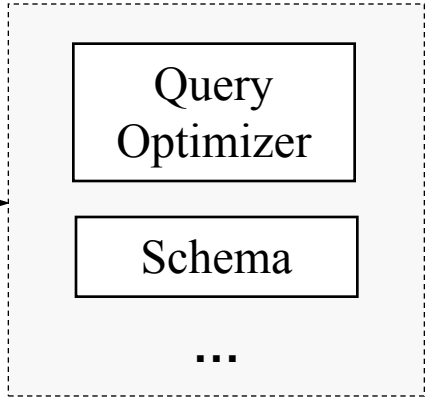


Example: Database Tuning Architecture



Tuning Client

What-if API



Database Server

- Create Hypothetical Index/View.
- Optimize Query with respect to hypothetical configurations.

Credit: Microsoft

Summary

- Many alternative file organizations exist, each appropriate in some situation.
- If selection queries are frequent, sorting the file or building an *index* is important.
  - Hash-based indexes only good for equality search.
  - Tree-based indexes best for range search; also good for equality search.

Summary (cont'd)

- Can have several indexes on a given file of data records, each with a different search key.
- Understanding the nature of the *workload* for the application
  - 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!).
  - Index maintenance overhead on updates to key fields.
  - Choose indexes that can help many queries, if possible.