Materialized View Updates

19

- Update on a single view without aggregate operations: update may map to an update on the underlying base table (most SQL implementations)
- Views involving joins: an update *may map to an* update on the underlying base relations not always possible

Example: A Data Warehouse

20

- Wal-Mart stores every sale at every store in a database.
- Overnight, the sales for the day are used to update a *data warehouse* = materialized views of the sales.
- The warehouse is used by analysts to predict trends and move goods to where they are selling best.

Indexes

21

- *Index* = data structure used to speed access to tuples of a relation, given values of one or more attributes.
- Could be a hash table, but in a DBMS it is always a balanced search tree with giant nodes called a *B-tree*.

Declaring Indexes

22

- No standard!
- Typical syntax:

```
CREATE INDEX BeerInd ON Beers(manf);
CREATE INDEX SellInd ON Sells(bar,
  beer);
```

Using Indexes

- 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 and SellInd to find the prices of beers manufactured by Pete's and sold by Joe. (next slide)

Using Indexes --- (2)


```
SELECT price FROM Beers, Sells
WHERE manf = 'Pete''s' AND
    Beers.name = Sells.beer AND
    bar = 'Joe''s Bar';
```

1. Use BeerInd to get all the beers made by Pete's.
2. Then use SellInd to get prices of those beers, with bar = 'Joe''s Bar'

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.

INDEXES

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

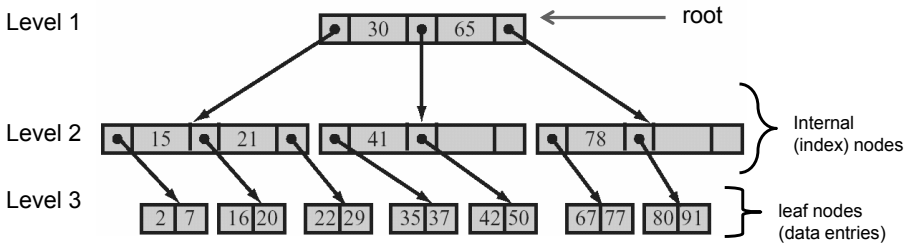
SE 3DB3 (Slides adapted from Dr. Fei Chiang) Fall 2016

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

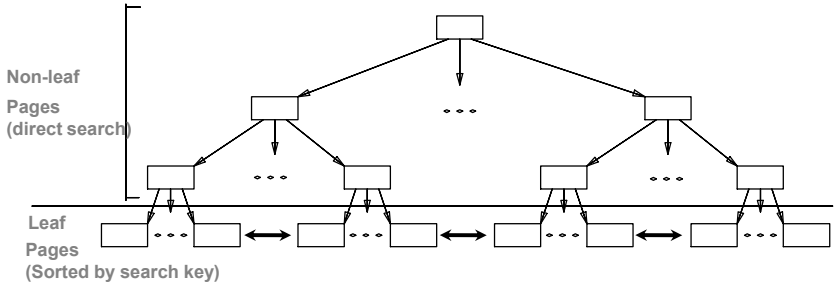
- The B+ tree structure is the most common index type in databases today.
- Index files can be quite large, often stored on disk, partially loaded into memory as needed
- Each node is at least 50% full

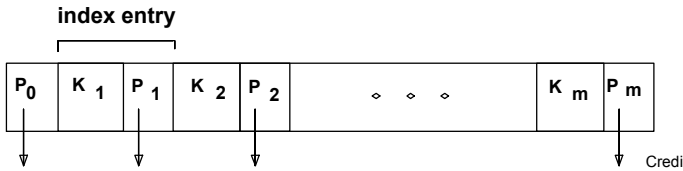


Credit: S. Lee

B+ Tree Index

Supports equality and range-searches efficiently

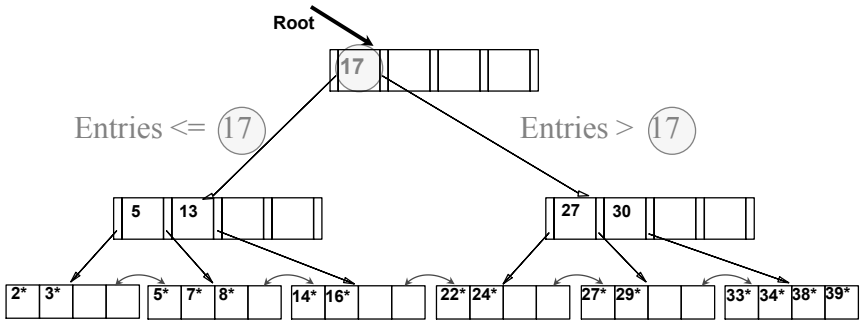




Credit: Renee Miller

Example

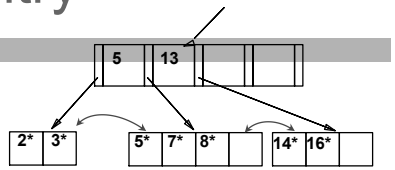
5



- 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

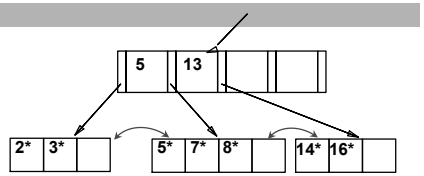
6



- 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

7



- 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.