Physical Database Design

Ramakrishnan & Gehrke, Chapter 17 & 18

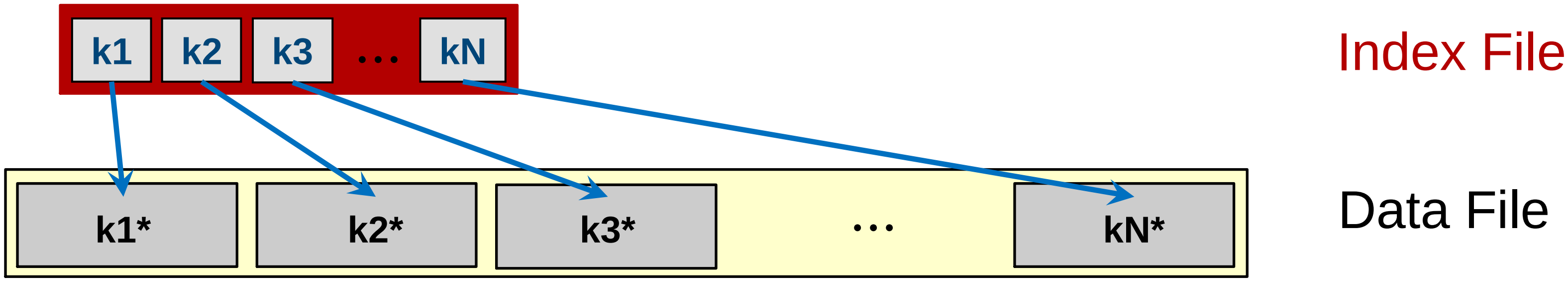
Alternative Database File Organizations

- Basic storage mapping: Table stored sequentially in a file
  - How to organise for best search performance?
- Many alternatives – each ideal for some situations, not so good in others:
- Heap (random order) files
  - Suitable when typical access is file scan retrieving all records
- Sorted Files
  - Best if records retrieved in some order, or only `range' of records needed
  - Updates expensive
- Indexes = aux data structures to quickly address records by key
  - Only index search key fields

Constructor University

Index

- Idea: Create condensed `index' (aka lookup) file
  - All non-lookup attribtues left out → file smaller → search faster
  - ...plus extra tricks
- predefined search key fields
  - Index always on one table
  - Any attribute can be search key
- speeds up retrieval of data entries k* with a given key value k



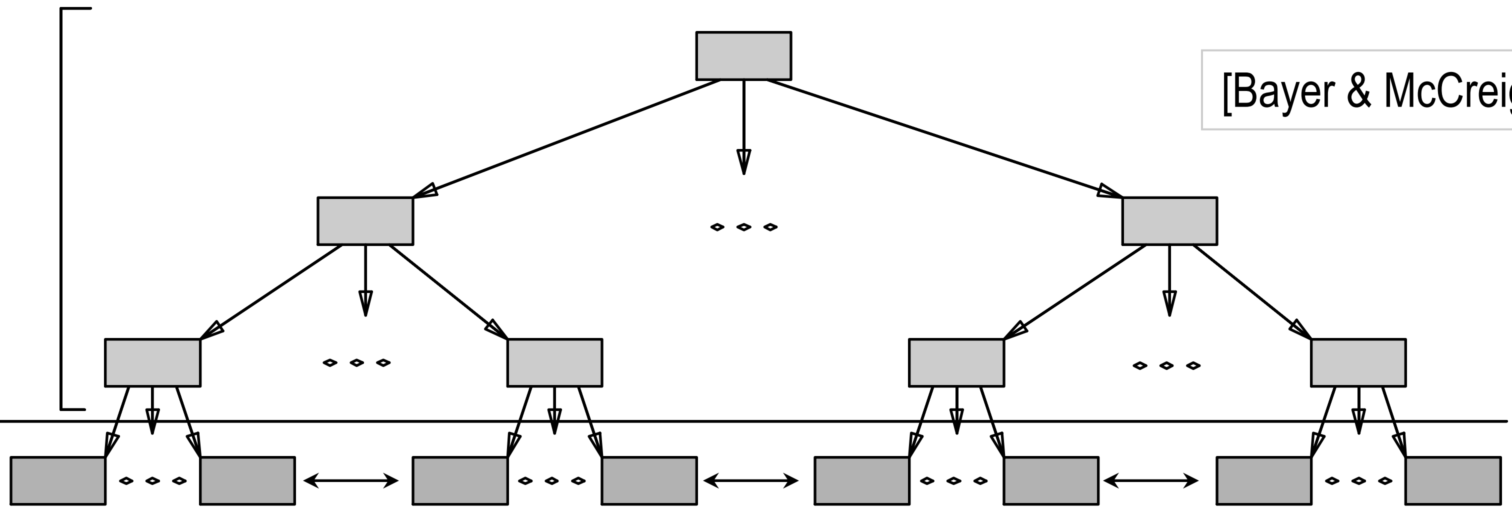
What to Search for?

- Point search: find exactly 1 record
  - „Find student with sid=4711“
- Range search: find tuples where attribute values match range (interval)
  - *“Find all students with gpa > 3.0”*

B+ Tree Indexes

[Bayer & McCreight, 1972]

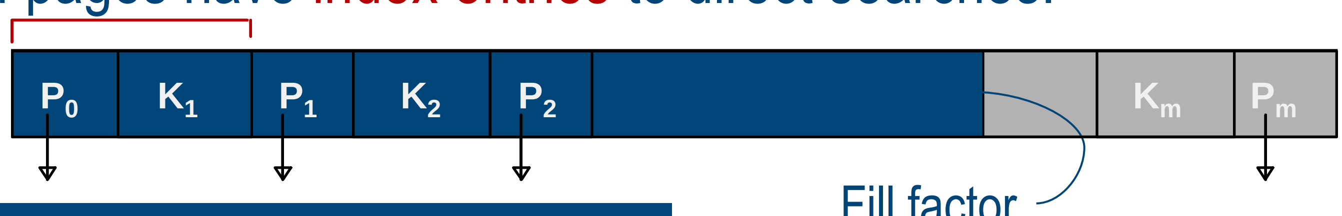
Index pages



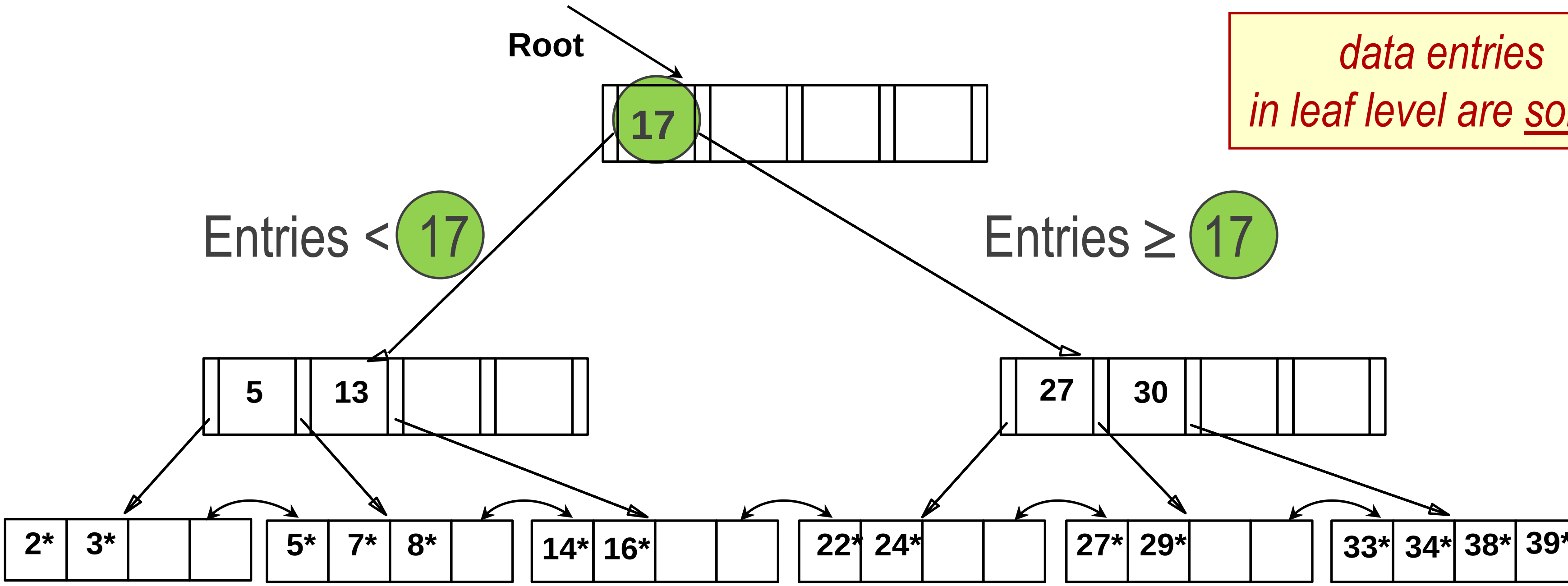
Leaf pages

("sequence set"; sorted by search key)

- Ordered Tree
- Leaf pages contain data entries, are chained (prev & next)
- Non-leaf pages have index entries to direct searches:



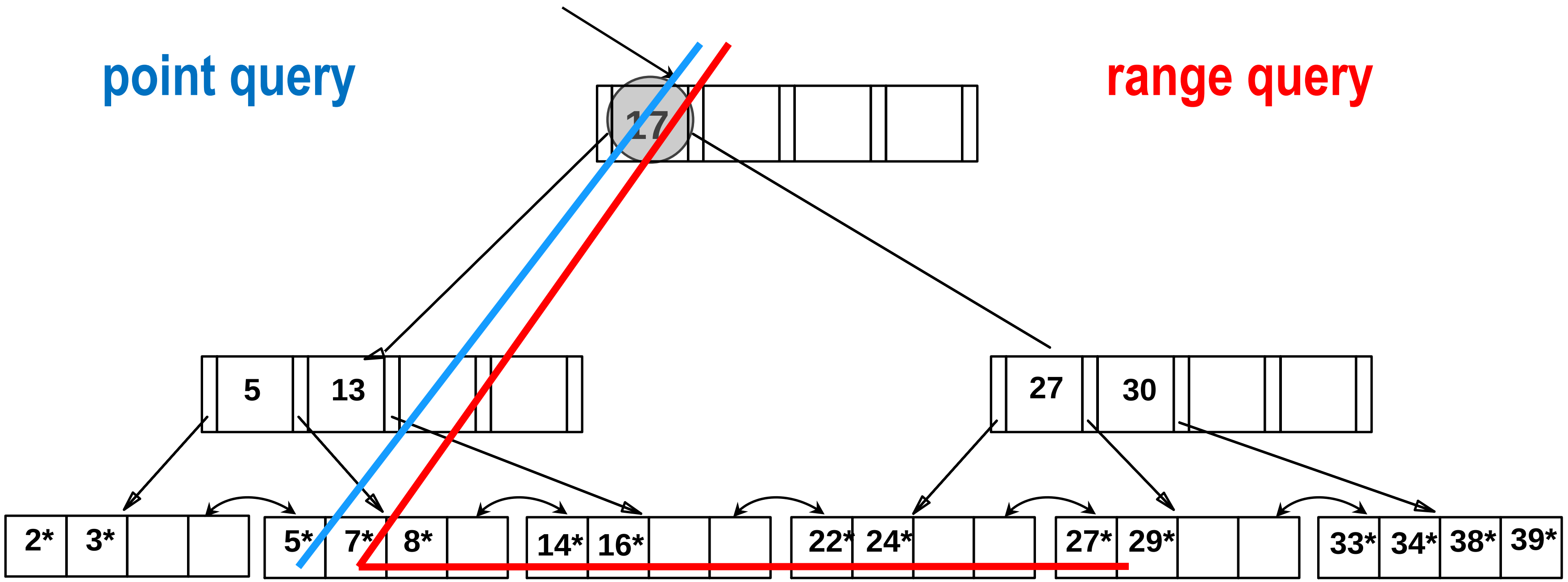
Example B+ Tree



data entries
in leaf level are sorted

- Find 28*? 29*? All > 15* and < 30*?
- Insert/delete: Find data entry in leaf, change it; adjust parent if needed
  - change sometimes bubbles up the tree
- Complexity: O($\log_F N$) where F = fan-out, N = # leaf pages

Example B+ Tree: Traversal Pattern



B+ Trees in Practice

- Typical fill-factor: 67%
- Average fanout: 133
- Typical capacities:
  - Height 3: 133^3 = 2,352,637 records
  - Height 4: 133^4 = 312,900,700 records
- Can often hold top levels in buffer pool:
  - Level 1 = 1 page = 8 Kbytes
  - Level 2 = 133 pages = 1 Mbyte
  - Level 3 = 17,689 pages = 133 MBytes

Hash-Based Indexes

- Goal: *compute* address without disk access, i.e., in O(1)
- Idea: distribute data evenly into fixed number of "buckets"
  - Compute location from key via Hashing function h: key → bucket
  - Example hashing function: h(int r) = r**a* mod *b*   with *b* prime relative to *a*
  - If keys match same address: overflow pages
- Hash index = collection of buckets + hashing function
  - Bucket = primary page plus zero or more overflow pages
  - Buckets contain data entries
- Good for equality, no support for range queries

Summary

- Many alternative file organizations, each appropriate in some situation
- Index = collection of data entries
  plus a way to quickly find entries with given key values
- If selection queries are frequent, sort file or build an index
  - Hash indexes only good for equality search
  - Sorted files and tree indexes best for range search; also good for equality search
  - Files rarely kept sorted in practice; B+ tree index is better
- Understand workload and DBMS query plans

Index-Only Plans

Index-Only Plans

- With suitable index + suitable query:
  can answer without retrieving tuples from relations involved

<E.did>

```
SELECT E.did, COUNT(*)
FROM Emp E
GROUP BY E.did
```

<E.did,E.sal>

Tree index!

```
SELECT E.did, MIN(E.sal)
FROM Emp E
GROUP BY E.did
```

<E. age,E.sal>
or
<E.sal, E.age>

Tree index!

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

Index-Only Plans (contd.)

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

- Index-only plans possible
  if key is <dno,age> or <age,dno>
  - Which is better?

Index-Only Plans (contd.)

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

- Index-only plans possible if key is <dno,age> or <age,dno>
  - Which is better?

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

  - What if we consider the second query?

Online Index Demos

- R-tree Demo
- R-Tree Visualisation Demo
- Quadtree Index Demos

Database Tuning

Index Selection Guidelines

- Understand workload:
  - Queries vs. update
  - What relations (sizes!), attributes, conditions, joins (selectivity!), …?
- Attributes in WHERE clause are candidates for index keys
  - Exact match condition suggests hash index, range query suggests tree index
  - Consider multi-attribute search keys for several WHERE clause conditions
    - *Order of attributes important for range queries*
- Choose indexes that benefit as many queries as possible
  - impact on updates: Indexes make queries faster, updates slower
  - require disk space
- *understand how DBMS evaluates queries & creates query evaluation plans*

Masking Conceptual Schema Changes

```
CREATE VIEW Contracts(cid, sid, jid, did, pid, qty, val)
        AS
            SELECT * FROM LargeContracts
          UNION
            SELECT * FROM SmallContracts
```

- Assumption: few large (high-budget) contracts → important to be fast
- Split Contracts → LargeContracts + SmallContracts, masked by view
  - Regular users simply access Contracts
  - high-profile users (boss) access LargeContracts for efficient execution

Key Performance Factors

Mark Fugate • My experience is that proper, or highest normal form normalization takes care of the first half of the optimization process by reducing the size of the stored data and reducing the numbers of operations required to maintain the data.

Query plans and query behaviours tell us how to properly index. Server tuning includes the proper storage media and knowledge of file systems and media tuning. Understanding your servers and knowing how to tune the OS, file systems, storage and kernel is all part of being a DBA.

Further, keeping SQL out of the client code makes all of the above attainable. I force all client applications in our shop to use stored procedures only. This gives me complete control over indexes, table structures, and all queries ensuring that nothing obnoxious enters the database.

1 day ago • Like

[LinkedIn Database list]

PS: A Moderately Complex Query

```
SELECT stadtbezirk, stadtteil, name, stadtteilchar, 'touche' AS entstehung, the_geom FROM
    (SELECT foo3.stadtbezirk, foo3.stadtteil, foo3.name, foo3.stadtteilchar, foo3.the_geom FROM
        (SELECT foo.gid, max(foo.laengste) AS laengste FROM
            (SELECT a.gid, b.stadtbezirk, b.stadtteil, b.name, b.stadtteilchar,
                (ST_Length(ST_Intersection(a.the_geom, ST_Union(b.the_geom)))) AS laengste
                FROM symdif a, dump b
                GROUP BY a.gid, a.the_geom, b.stadtbezirk, b.stadtteil, b.name, b.stadtteilchar
                HAVING ST_Touches(a.the_geom, ST_Union(b.the_geom))
                ORDER BY a.gid) AS foo
        GROUP BY foo.gid) AS foo2
        ,
        (SELECT a.gid, b.stadtbezirk, b.stadtteil, b.name, b.stadtteilchar, a.the_geom AS the_geom,
            (ST_Length(ST_Intersection(a.the_geom, ST_Union(b.the_geom)))) AS laengste
            FROM symdif a, dump b
            GROUP BY a.gid, a.the_geom, b.stadtbezirk, b.stadtteil, b.name, b.stadtteilchar
            HAVING ST_Touches(a.the_geom, ST_Union(b.the_geom))) AS foo3
    WHERE (foo2.gid = foo3.gid AND foo2.laengste = foo3.laengste)
    GROUP BY foo2.gid, foo3.stadtbezirk, foo3.stadtteil, foo3.name, foo3.stadtteilchar,
        foo3.laengste, foo2.laengste, foo3.the_geom) AS foo4
;
```