Physical Database Design

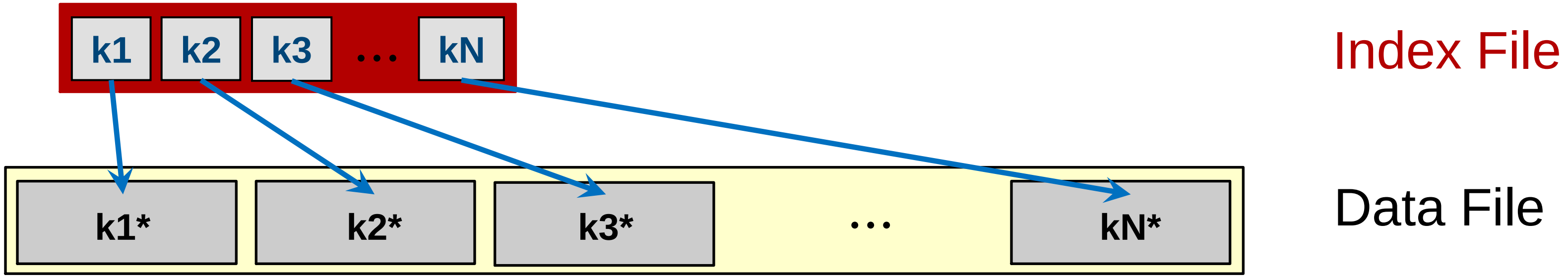
Ramakrishnan & Gehrke, Chapter 17 & 18

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

Index

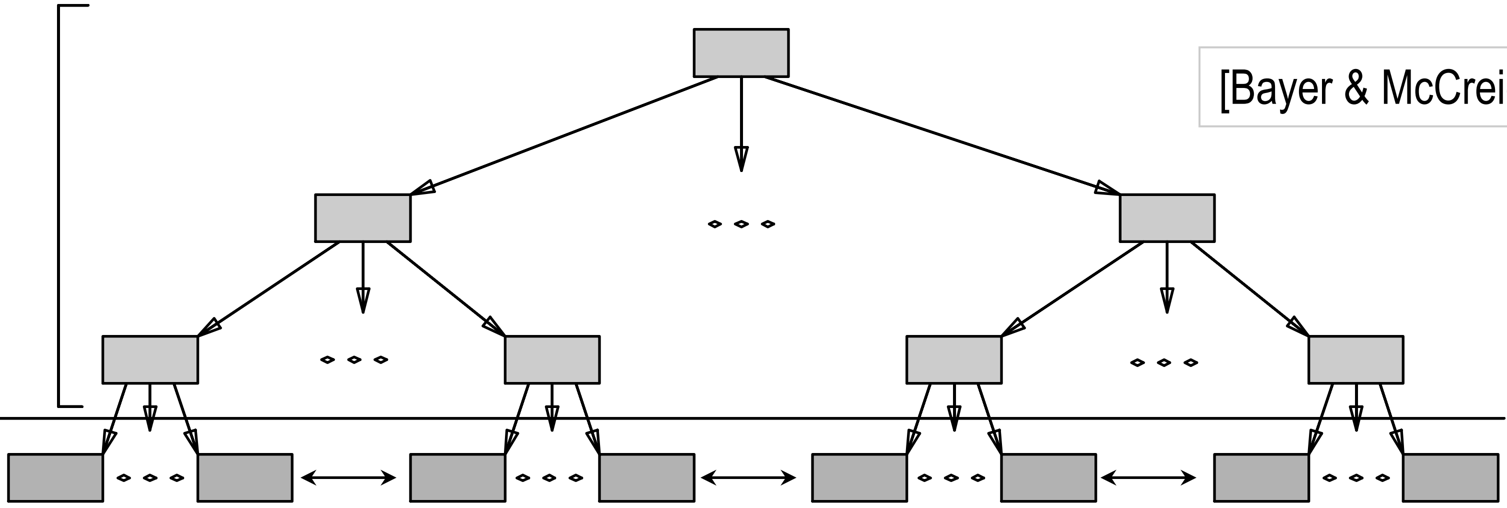
- Idea: Create condensed `index' (aka lookup) file
  - All non-lookup attribtues left out → file smaller → search faster
  - ...plus extra tricks
- predefined search key fields
  - Index always on one table
  - Any attribute can be search key
- speeds up retrieval of data entries k* with a given key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

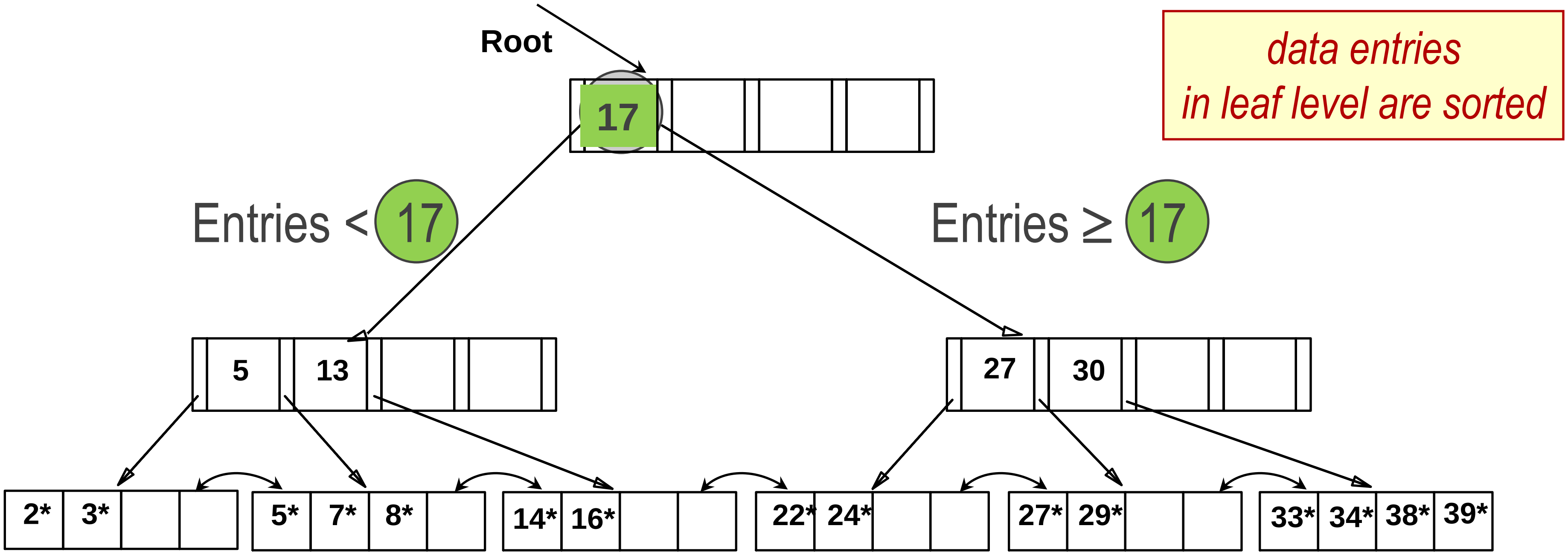
Index pages

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:

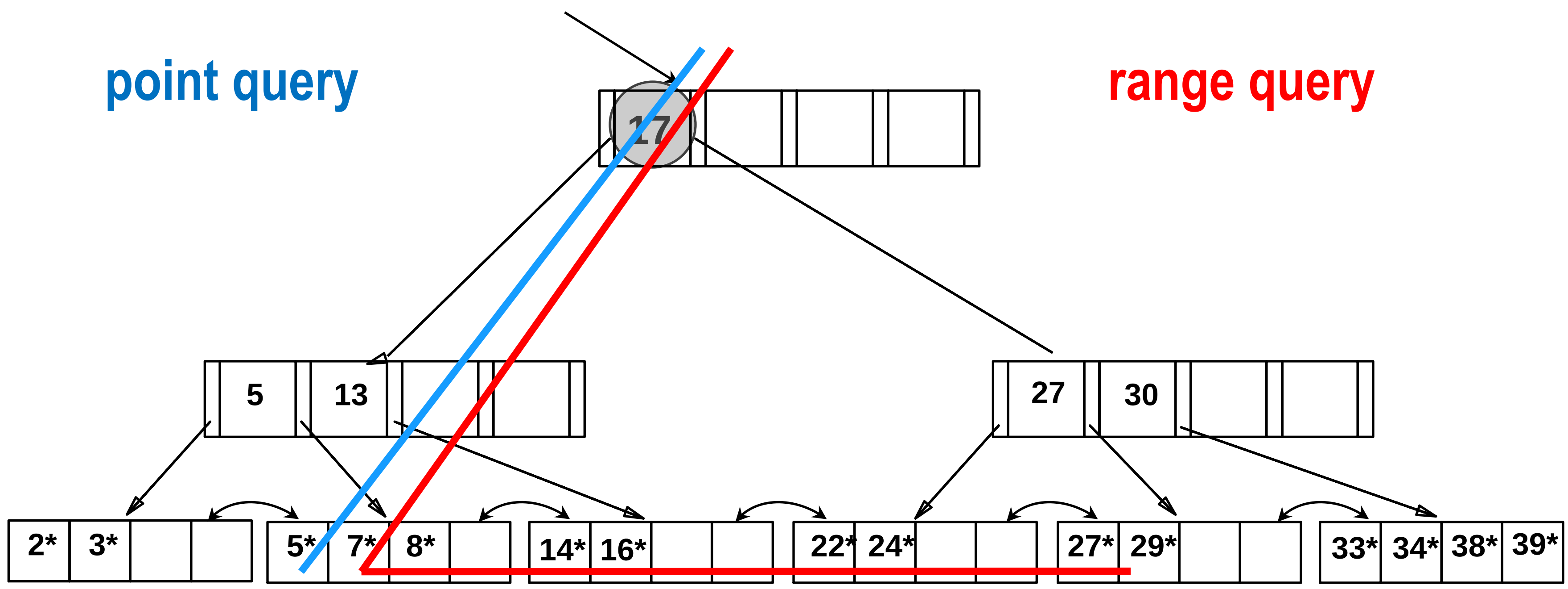
Example B+ Tree



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

Indexing Spatial Data

Outlook: Spatial Data Management

Points(X number, Y number, ptType: integer)

- Spatial data
  = multi-dimensional data
  - Objects regions have location
  - [+ spatial extent, ie, boundary]
- 2 fundamentally distinct categories:
  - Vectorial: point, line, region data in n-dimensional space
  - Raster: n-D “images” = arrays
- Not only spatio-temporal data:
  Also feature vectors extracted from text/images = non-spatial data!
  - Usually *very* high-dimensional, 1000s

Types of Multidimensional Queries

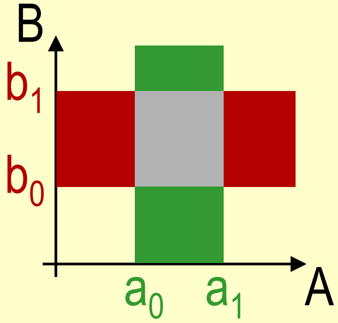
- Point Queries
  - *"Show Bremen"*
- Spatial Range Queries
  - *"Find all cities within 50 km of Bremen"*
  - Query has associated region (location, boundary)
- Nearest-Neighbor Queries
  - *"Find the 10 cities nearest to Bremen"*
  - Results must be ordered by proximity
- Spatial Join Queries
  - *"Find all cities near a lake"*
  - Expensive; join condition involves regions and proximity!
- Similarity queries
  - content-based retrieval
  - "Given a face, find the five most similar faces"
- *…plus aggregation, and several more*

Multiple B+ Trees?

- Query example:

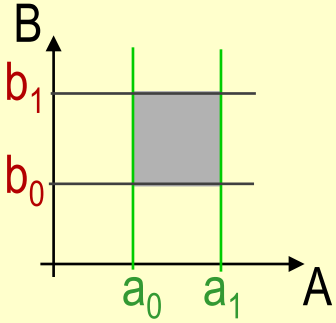
select * from R where $a_0 < A < a_1$ and $b_0 < B < b_1$

Several conventional indexes:



- read tuple with $a_0<A<a_1$
- read tuple with $b_0<B<b_1$
- intersect

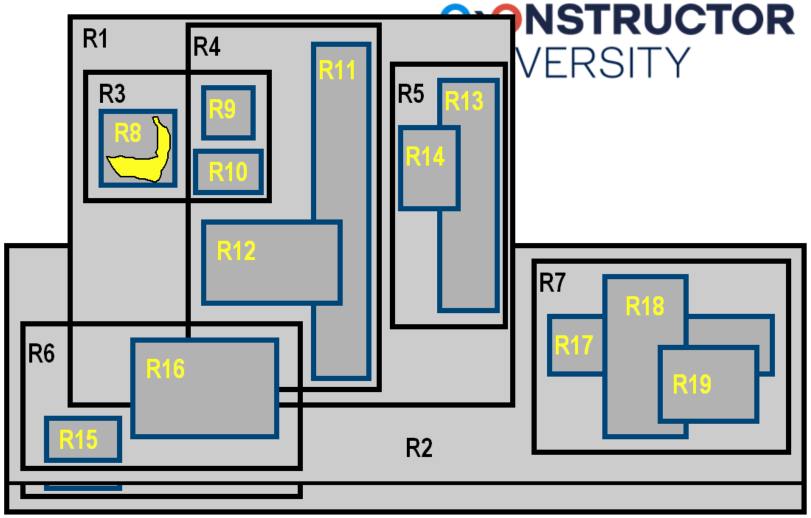
wanted:



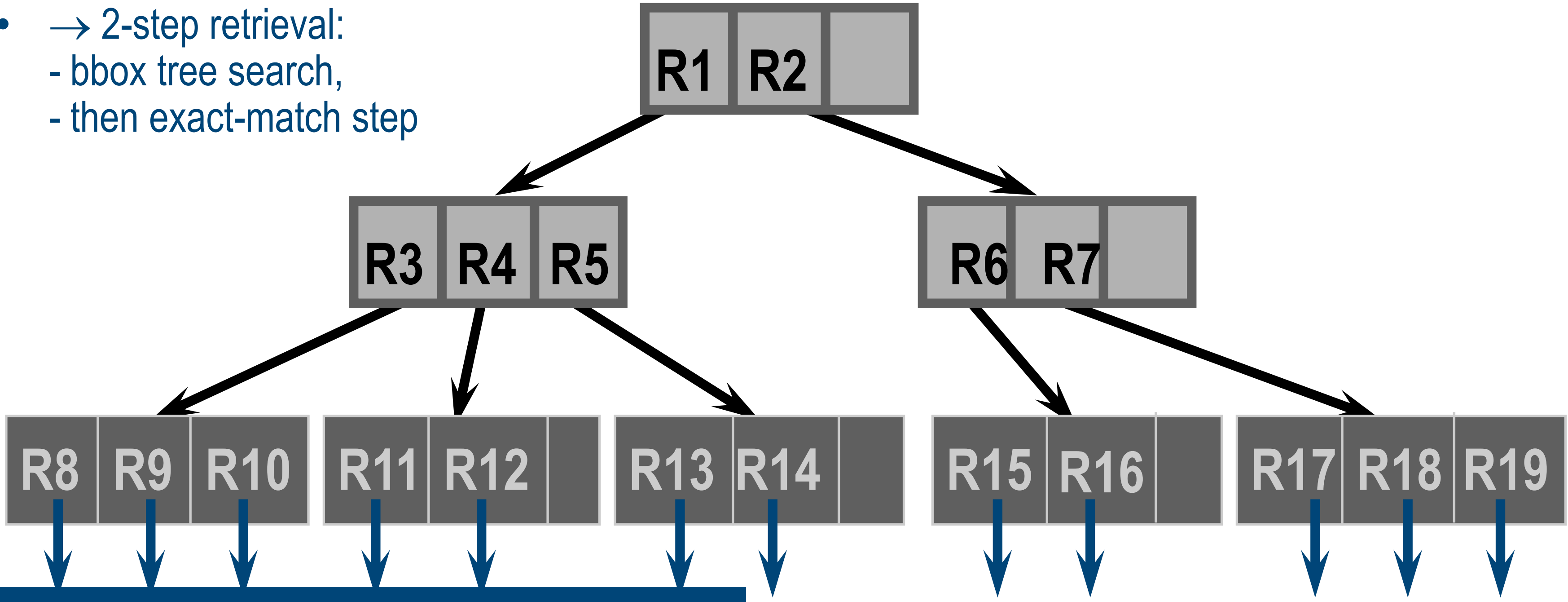
read only tuples with $a_0<A<a_1$ <u>and</u> $b_0<B<b_1$

- Specific family of n-D ("spatial") indexing techniques
  - R-tree = balanced tree; widely used in GIS
  - Grid Files, Quad trees, “space-filling” curves, …

R-Tree



- tree-structured n-D index [Guttman 1984]
- Index value = bounding box
  - Node's box covers its subtree
  - we do not search exact object boundaries, but their bounding boxes
  - → 2-step retrieval:
    - bbox tree search,
    - then exact-match step



Applications of Multidimensional Data

- Geographic Information Systems (GIS)
  - Geospatial information; service standards by Open GeoSpatial Consortium (OGC)
  - Vendors: ESRI, Intergraph, SmallWorld, …, Oracle, …; open-source: Grass, PostGIS, …
  - All classes of spatial queries and data are common
- Computer-Aided Design / Manufacturing
  - spatial objects, ex: surface of airplane fuselage
  - Range queries and spatial join queries are common
- Multimedia Databases
  - Images, video, text, etc. stored and retrieved by content
  - First converted to *feature vector* form; high dimensionality
  - Nearest-neighbor queries are the most common

Database Tuning

Tuning Queries and Views

- If a query runs slower than expected,
  check if index needs to be re-built or statistics too old
- Sometimes, DBMS may not be executing the plan you had in mind.
  Common areas of weakness:
  - Selections involving null values; arithmetic or string expressions; OR conditions; …
  - Missing features (ex: index-only strategies), join methods, poor size estimation, …
- Check plan used, adjust choice of indexes or rewrite query/view
  - Avoid nested queries,
    temporary relations,
    complex conditions,
    operations like DISTINCT and GROUP BY

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*

More Decisions to Make

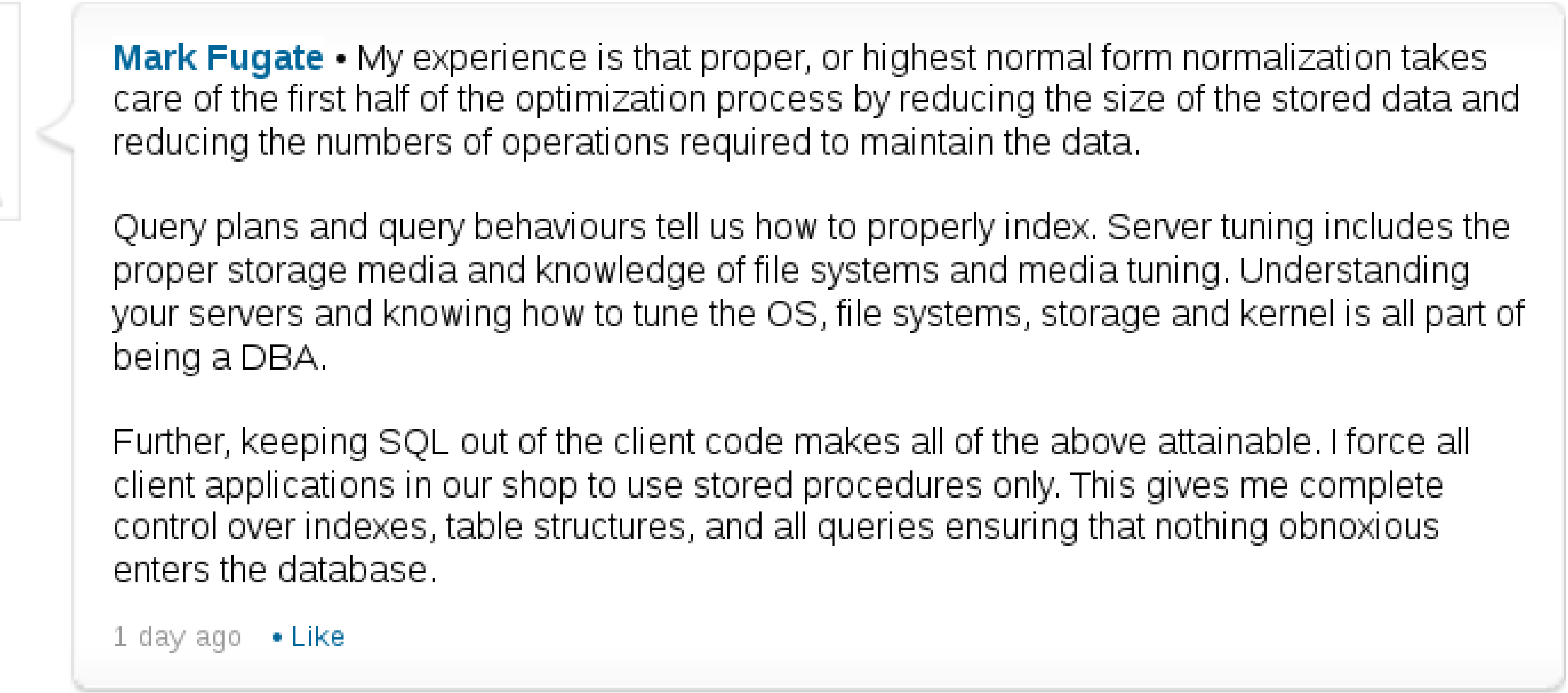
- Change conceptual schema = ER diagram?
  guided by workload, in addition to redundancy issues
  - Consider alternative normalized schemas? (many choices!)
  - “undo’’ some decompositions, settle for a lower normal form, such as 3NF? (denormalization)
  - Horizontal partitioning, replication, views ...see manuals
- Change logical schema = table definitions?
- If made after a database is in use, called schema evolution

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



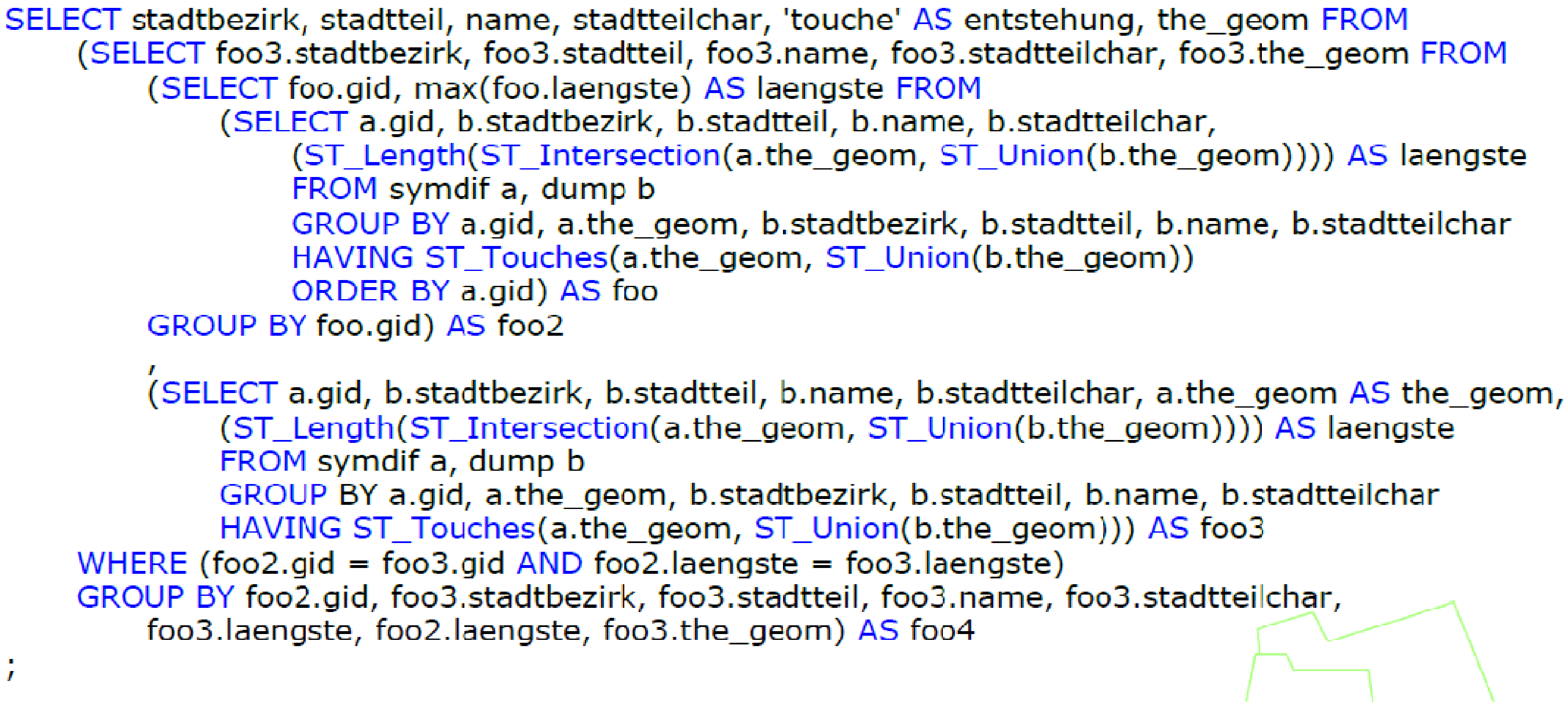
Mark Fugate • My experience is that proper, or highest normal form normalization takes care of the first half of the optimization process by reducing the size of the stored data and reducing the numbers of operations required to maintain the data.

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.

[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
;
```