LECTURE 7

INDEXING AND ACCESS METHODS

PHYSICAL QUERY PLANS

Summary of Execution Models:

- Iterator Model (Volcano, pipeline model)
    - ↳ void open()
    - ↳ Tuple next()
    - ↳ void close()
    - ↳ "Pulls" tuples from the top
    - ↳ Easy to control intermediates. Easy stop (i.e. LIMIT)
- Batch-at-a-time
    - ↳ Reduce function calls (important for in-mem DBs)
- Bottom-up
    - ↳ Better locality. More memory required.

Logical and Physical (Execution) Query Plans:

- Logical QP uses relational algebra operators and extensions, and assume a way of reading data.
    - ↳ Describe the order in which operators are applied to execute a query.
    - ↳ They don't describe how each op. is implemented, or how data is precisely accessed.
- Both op. implementation and how data is accessed (access methods) are crucial for achieving good execution time / performance.
    - ↳ The role of the query optimizer is to find a good physical plan, op. impl. ~~and~~ given access methods.

Outline of this lecture:

- Overview of op. impl. and storage
- Example. Building query cost intuition
- Access Methods

Overview Op. Implementation and Storage:

- Nested loop join, $R \bowtie S$

```
for r in R:
    for s in S:
        if predicate(r,s):
            output join(r,s)
```

- Hash Join
    - ↳ Build hash table (in-mem) for the smallest relation.
    - ↳ Probe tuples from the second (by scanning it).

→ Many more implementations. One lecture on 'Join Algos'.

Storage: (assume magnetic disk or SSD)

- Records (or tuples) are stored in "pages"
- Pages are part of HeapFiles.
- All records are unordered. Pages must keep track of which "slots" are free and occupied
- Pages are sized so it's efficient to read them from disk and write them to it.
- Pages are cached in the BufferPool, which works as a 'cache' using the main memory of the server.
- Why not storing tuples ordered in disk? What happens when we write a new one?
- Access Methods are strategies to read tuples from disk, knowing how storage is organized.

Performance Engineering. Building Intuition on Query Plan cost:

- CPU cost (# instructions / unit time) — (1 Ghz ≡ 1 B instr/s), 1 ns/instr.
- IO cost (# pages read / unit time) — 100 MB/s ≡ 10 ns/byte
- Random IO (page read + seek) — 10 ms/seek ≡ 100 seeks/sec

  * It's possible to execute around 10 M instr/seek. Beware random IO — triggers seek.

- What cost dominates in database execution?
  - ↳ Depends on query workload (queries and data).
  - ↳ Depends on hardware characteristics.
  - ↳ Depends on database design (on-disk, in-memory).
  - ↳ Depends on system implementation.
- When designing a system / trying to understand one, we must be able to make sense of the performance we observe.

Example:

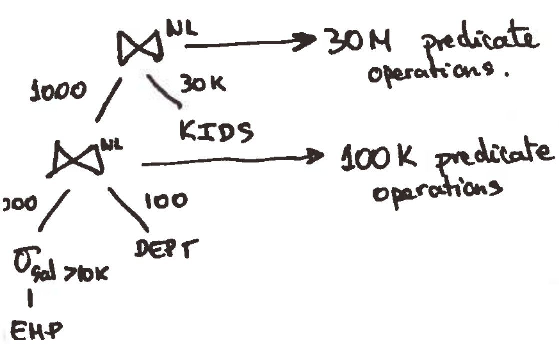
- 100 tuples/page
- 10 KB/page
- 10 pages RAM
- 10 ms seek time
- 100 MB/s IO

|DEPT| ≡ 100 ≡ 1 page ≡ 10KB

|EMP| ≡ 10K ≡ 100 pages ≡ 1 MB

|KIDS| ≡ 30K ≡ 300 pages ≡ 3 MB

QUERY: select * from EMP, DEPT, KIDS
where e.sal > 10K AND
EMP.dno = DEPT.dno AND
EMP.eid = KIDS.eid;



} CPU cost in terms of predicate evaluations.

Note we need to know the physical op. implementation

What about IO cost?

- Assume DEPT is outer in the NL join.
  - ↳ 1 scan of DEPT (1 page) : 10ms seek
  - ↳ 100 sequential scans of EMP (100 × 100 pages) [Only 10 pages fit in cache, so we need to read data from disk]
    - ↳ 1 scan of EMP : 1 seek + read in 1MB
      - 100 pages × 10KB/page ≈ 1MB
      - ↳ 10ms (seek) + $\frac{1MB}{100MB/s}$ = 20ms.
    - ↳ In total 20ms × 100 scans = 2s.
  - ↳ $\boxed{2.01s}$ (2s EMP + 0'01s DEPT)
- Assume DEPT is inner in the NL join
  - ↳ read page of EMP (seek) : 10ms
  - ↳ read DEPT into RAM : 10ms
  - ↳ seek back to EMP : 10ms (pointer was at DEPT)
  - ↳ scan EMP : 10ms ($^{1MB}/_{100MB/s}$)
  - ↳ Total cost of $\boxed{40ms}$

• How to choose the appropriate op. implementation. How to avoid IO costs?

Access Methods:

↳ Strategies to access data with minimum IO cost.

↳ Indexes. Built to be external, i.e., to operate on disk.
  ↳ Different indexes have different properties (hash index not good for range queries).

↳ Indexes: Additional structure that avoids scanning all tuples. We want to use indexes when they're available.

↳ General idea of indexes:
  - ↳ insert (key, record_id) // points from a key to a record id.
  - ↳ lookup (key) // returns record id.
  - ↳ lookup (low key ... high key) // return records.

↳ It'd be simple to design these in-memory, but for full generality we must support them on disk.

→ Heap Scan: we've seen this. Iterate over tuples over pages.

→ Hash File

→ B-Tree

Hash File:

- map (Key) → rid ; Hash table that for EMP hashes on 'name' attribute.
- ↳ h(name) : [1, K]
  ↳ h(x) = x mod K
- Suppose K buckets, and one page per bucket.
  ↳ When inserting a tuple, we hash on the attribute to determine in which page to store it, i.e., we append the record/tuple to that page.
- When we receive a query that asks for "Tim", we hash the name and know in which page to find him.
  ↳ As opposed to scanning all pages.
- The key challenge is in how to select the # buckets.
  ↳ we could choose a high number → this may be wasteful.
  ↳ if we choose too few, then buckets overflow
    ↳ we can create chains of pages, but then we progressively lose the index benefit

+ Extensible hashing:

- Create family $H_k(x)$ of hash functions parametrized by 'k'.
  ↳ $k = 1$; $h_1(x) = \{0, 1\}$ $h_k(x) = x \bmod 2^k$
  ↳ $k = 2$; $h_2(x) = \{0, 1, 2, 3\}$
- Start with the $h_k(x)$ hash function. When bucket overflows, redistribute that bucket into the new buckets given by the next hash function.

B+Tree:

- A balanced tree in which internal nodes direct the search for some point/range query and the leaf nodes contain/point to the pages with the data.
- Internal nodes designed so they fit (each one) in one page. That way root of the tree fits in memory.
  ↳ Underlying records are sorted and those pages are linked to each other
    ↳ Link pointers / Doubly-linked list. Useful to answer range queries efficiently.
  ↳ Cost of traversing tree from root to leaf and then reading the necessary pages.
- Why do we want the tree to be balanced?
- High fan-out (children per node) so height remains low. Why do we want low height?