# 6.830/6.814 — Notes[*] for Lecture 5: Database Internals Overview (2/2)

Carlo A. Curino

September 27, 2010

## 1 Announcements

- LABS: Do not copy... we perform automatic checking of plagiarism... it is not a good gamble!
- Projects ideas and rules are posted online.

## 2 Readings

For this class the suggested readings are:

- Joseph Hellerstein, Michael Stonebraker and James Hamilton. Architecture of a Database System. Online at: `http://db.cs.berkeley.edu/papers/fntdb07-architecture.pdf`

It is a rather long paper (don't be too scared by the 119 pages, the page format makes it look much longer than it is) that is in general worth reading, however we only require you too read sections: 1, 2 (skim through it), 3, 4 (up to subsection 4.5 included), 5. You can also skim through section 6 that we will discuss later on. Probably doesn't all make sense right now – you should look at this paper again to this paper through the semester for context.

## 3 Query Optimization

### 3.1 Plan Formation (SQL → relational algebra)

First step in which we start thinking of "how" this query will be solved. It still abstracts many implementation details (e.g., various implementations of each

[*]These notes are only meant to be a guide of the topics we touch in class. Future notes are likely to be more terse and schematic, and you are required to read/study the papers and book chapters we mention in class, do homeworks and Labs, etc.. etc..

operator).

- Projection: $\Pi_{a,b}(T)$
- Selection: $\sigma_{a=7}(T)$
- Cross-Product: $R \times S$
- Join: $R \bowtie_{a1=a2} S$
- Rename: $\rho_{a/b}(R)$
- ...

Let's use the example query from the introduction lecture (it's in the notes but we didn't discuss it last time):

```
SELECT p.phone
FROM person p, involve i, operation o
WHERE p.name = i.person AND
   i.oper_name = o.name AND
   o.coverup_name = "laundromat";
```

Corresponding Algebra expression:
$\Pi_{phone}(\sigma_{o.coverup_name="laundromat"}(\sigma_{p.name=i.person}(\sigma_{i.oper_name=o.name}(person \times (involved \times operation))$

Let's show this in a more visual way (the query plan):

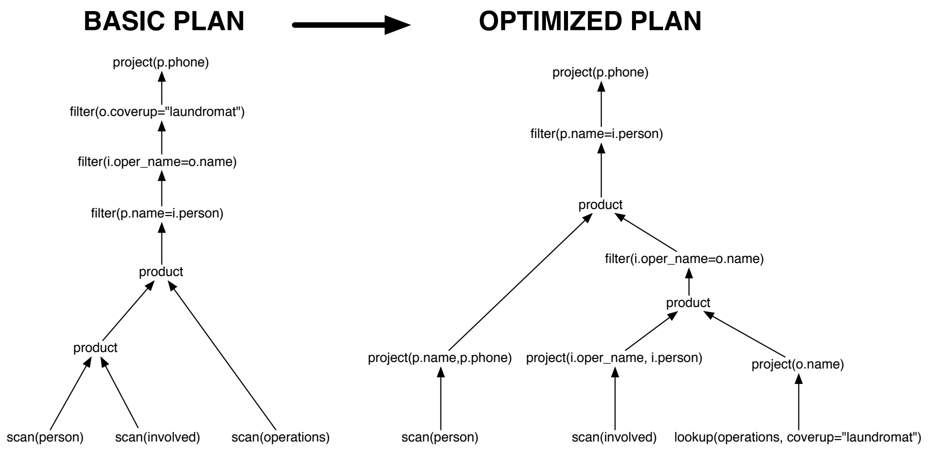

Figure 1: Two equivalent query plans for the same query.

The goal of the optimizer is two find equivalent query plans that lead to higher performance. This is done in two ways:

- logical: re-ordering operators in a semantic-preserving way

- physical: choosing among multiple implementation of the operators the one that for the specific scenario supposedly will give best performance

There are two main approaches to do this:

- heuristic: uses a set of rules or strategies that typically pay back, e.g., push all selections before joins.
- cost-based: define a cost model for the execution (e.g., number of CPU operations + amount of disk I/O) and compare various plans according to the expected cost. Pick lowest cost.

Restricting to certain plans (left deep) allows to avoid temporary buffering of results or re-computation.

Optimization is often based on statistical information about the data... e.g., data distributions... often maintained as histograms. This can allows us to predict selectivity of operators, and thus have an idea of size of intermediate results and the cost of the following operators.

## 4 Query (Pre)Compilation

Most DBMS support query pre-compilation. I.e., the application can declare that is going to use (frequently?) a certain type of query. The structure of the query is provided in advance and later on the application only passes to the DBMS the "parameters" to specialize this template into a specific query. This allows to capitalize query parsing, some authorization work, and optimization. However, predicate selectivity becomes a static choice. This can lead to significant performance improvement for very regular workloads (e.g., Web / OLTP).

## 5 Physical Storage

All records are stored in a region on disk; probably easiest to just think of each table being in a file in the file system.

Tuples are arranged in pages in some order. Simple append to the file or "heap file" access path is a way to access these tuples on disk.

Now how to access the tuple (the *access path*)?

- heap scan
- index scan provide an efficient way to find particular tuples

What is an index? What does it do?

Index is an auxiliary data structure that aims at improving a specific type of access to the data. In particular an index is defined on one of the attributes, and for each value points at the corresponding positions on disk. Operations:

- **insert (key, recordid)** points from a key to a record on disk records

- **lookup (key)** given a single key return the location of records matching the key.
- **lookup(lowkey,highkey)** returns the location of records in the range of keys given as input.

Hierarchical indices are the most common type used—e.g., B+-Trees indices typically point from key values to records in the heap file

Shall we always have indexes? What are the pros and cons? (keep the index up-to-date, extra space required)

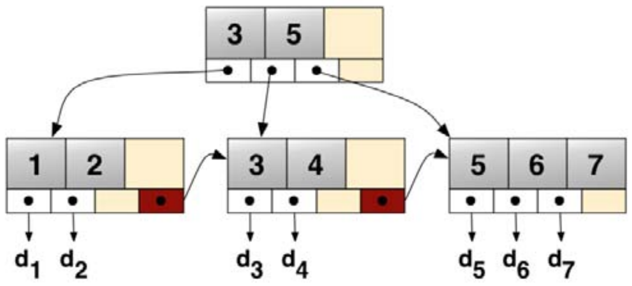

Figure 2: B+Tree graphical representation.
Courtesy of Grundprinzip on Wikipedia.

Special case, is a *"clustered"* index, i.e., when the order of the tuples on disk correspond to the order in which they are stored in the index. What is it good for? (range selections, scans).

So now we have 2 options:

- a heap scan (sequential, but load entire table),
- index scan (load less data, but might lead to a lot of Random I/O).

In query plan we can mix multiple of this choosing the best for each basic table we need to read. Why this might be useful? Consider a case in which almost all tuples are needed from one table, but only very few from another table?

Later on we will get more into optimization details, for now let's keep in mind that for many basic relational algebra operators we might have multiple implementations, and that depending on data distribution and query characteristics (selectivity), different plans might be better than others.

## 6 Query Execution

It is very common for DBMS to implement the *Iterator model*. The key idea is that each operator implements the same interface:

- open

- (hasNext)
- next
- close

This allows chaining of operators without any special logic to be implemented. Also it is a natural way to propagate data and control through the stack of calls. There are many variations around this principal to allow for more batch processing of data.

Iterator code for a simple Select:

```
class Select extends Iterator {

  Iterator child;
  Predicate pred;

 Select (Iterator child, Predicate pred) {
   this.child = child;
   this.pred = pred;
 }

 Tuple next() {
   Tuple t;
   while ((t = child.next()) != null ) {
        if (pred.matches(t)) {
          return t;
        }
   }
   return null;
 }

 void open() {
   child.open();
 }
 void close() {
   child.close();
 }
}
```

## 7 Plan Types (only mentioned in class)

Different type of plans.

Bushy:

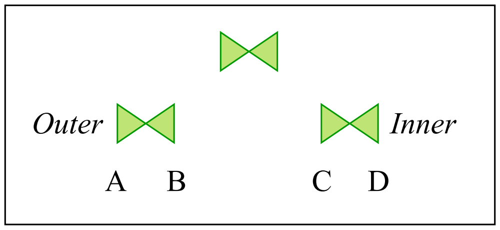

Image by MIT OpenCourseWare.

```
for t1 in outer
    for t2 in inner
       if p(t1,t2) emit join(t1,t2)
```

Bushy requires temporarily memorizing results or recompute.

Left Deep:

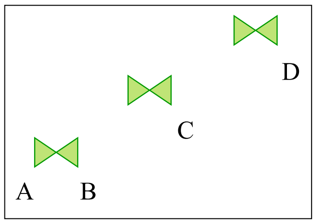

Image by MIT OpenCourseWare.

Results can be *pipelined*. No materialization of intermediate steps necessary. Many database systems restrict themselves to left or right deep plans for this reason

## 7.1 Cost of an execution plan

In class we went over another example (the one above of basic and advanced plan). That example was mainly about CPU costs, in the example below, we explore more the importance of Disk I/O. This is also good to prepare our next class, which will be all about access methods.

What's the "cost" of a particular plan?

| | | |
|---|---|---|
| CPU cost (# of instructions) | 1Ghz == 1 billions instrs/sec | 1 nsec / instr |
| I/Ocost(#ofpagesread,#ofseeks) | 100MB/sec | 10nsec/byte |
| (RandomI/O=pageread+seek) | 10msec/seek | 100seeks/sec |

Random I/O can be a real killer (10 million instrs/seek). (You probably experience it yourself on your laptop sometimes).

**When does a disk need to seek?**

**Which do you think dominates in most database systems?**

Depends. Not always disk I/O. Typically vendors try to configure machines so they are 'balanced'. Can vary from workload to workload.

For example, fastest TPC-H benchmark result (data warehousing benchmark), on 10 TB of data, uses 1296 × 74 GB disks, which is 100 TB of storage. Additional storage is partly for indices, but partly just because they needed additional disk arms. 72 processors, 144 cores – 10 disks / processor!

But, if we do a bad job, random I/O can kill us!

## 7.2 An example

Schema:

```
DEPT(dno,name);
EMPL(eid,dno,sal);
KIDS (kidname,eid);
```

Some characteristics of this hypothetical scenario:

- 100 tuples/page
- 10 pages in RAM 10 KB/page
- 10 ms seek time
- 100 MB/sec I/O
- cardinality DEPT = 100 tuples = 1 page
- cardinality EMPL = 10K tuples = 100 pages
- cardinality KIDS = 30K tuples = 300 pages

```
SELECT dept.name,kidname
FROM emp, dept, kids
WHERE e.sal > 10k AND
  emp.dno = dept.dno AND
  e.eid = kids.eid;
```

$\Pi_{name,kidname}(dept \bowtie_{dno=dno} (\sigma_{sal>10k}(emp)) \bowtie_{eno=eno} kids)$

More graphically:

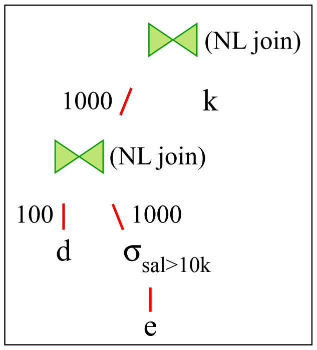

Image by MIT OpenCourseWare.

CPU operations:

- selection – 10,000 predicate ops
- 1st Nested loops join – 100,000 predicate ops
- 2nd nested loops join – 3,000,000 predicate ops

Let's look at number of disk I/Os assuming LRU and no indices

**if d is outer:**

- 1 scan of DEPT
- 100 consecutive scans of EMP (100 x 100 pg. reads) – cache doesn't benefit since e doesn't fit
  - 1 scan of EMP: 1 seek + read in 1MB =10 ms + 1 MB / 100 MB/sec = 20 msec
  - 20 ms x 100 depts = 2 sec

TOTAL: 10 msec seek to start of d and read into memory **2.1 secs**

**if d is inner:**

- read page of e – 10 msec
- read all of d into RAM – 10 msec
- seek back to e – 10 msec
- scan rest of e – 10 msec,

joining with d in memory... Because d fits into memory
TOTAL: total cost is just **40 msec**

No options **if plan is pipelined, k must be inner:**

- 1000 scans of 300 pages 3 / 100 = 30 msec + 10 msec seek = 40 x 1000 = **40 sec**

So how do we know what will be cached? That's the job of the buffer pool.

What about indexes? The DBMS uses indexes when it is possible (i.e., when an index exists and it support the required operation, e.g., hash-indexes do not support range search)

## 8 Buffer Management and Storage Subsystem

Buffer Manager or Buffer pool caches memory accesses... Why is it better if the DBMS does this instead of relying on OS-level caching? DBMS knows more about the query workload, and can predict which pages will be accessed next. Moreover, it can avoid cases in which LRU fails.

Also explicit management of pages is helpful to the locking / logging necessary to guarantee ACID properties.