6.814/6.830 Quiz 1 Review

Logistics

- Wednesday during lecture
- 75 min + 5 min to upload solution
- Covers material up to lecture 10
- Open book/notes, but no googling please.
- Email staff for special accommodation

Topics

- Schema design: Normal Forms, ER diagrams
- Query Optimization: Access methods, join algorithms, join ordering, cost analysis
- Database internals: Buffer pool, iterator model, index structures
- Column stores for analytics

Relational Algebra

Projection

π(R,c1, …, cn) = $\pi_{c1\ldots c2n}$R

select a subset c1 … cn of columns of R

Selection

σ(R, pred) = σ_{pred}R

select a subset of rows that satisfy pred

Cross Product (||R|| = #attrs in R, |R| = #rows in row)

R1 X R2 (aka Cartesian product)

*combine R1 and R2, producing a new relation with ||R1|| + ||R2|| attrs, |R1| * |R2| rows*

Join

$\bowtie$(R1, R2, pred) = R1 $\bowtie_{pred}$ R2 = σ_{pred} (R1 X R2)

IMS v CODASYL v Relational

	IMS	CODASYL	Relational
Many to many relationships without redundancy	✗	✓	✓
Declarative, non "navigational" programming	✗	✗	✓
Physical data independence	✗	✗	✓
Logical data independence	✗	✗	✓

Physical Data Independence

- Can change representation of data without needing to change code
- Relational / SQL doesn't specify how records are stored
  - No hashes, sort keys, etc.
  - Users can change these without changing code!

- Both CODASYL and IMS expose representation-dependent operations in their query API

Logical Data Independence

- What if I want to change the schema without changing the code?
- *Views* allow us to map old schema to new schema, so old programs work

Functional Dependencies

- $X \rightarrow Y$
- Attributes X uniquely determine Y
  - I.e., for every pair of instances x1, x2 in X, with y1, y2 in Y, if x1=x2, y1=y2
- For Hobbies, we have:
  1. SSN, Hobby → Name, Addr, Cost
  2. SSN → Name, Addr
  3. Hobby → Cost

  – 2 & 3 imply 1, by union under

  Armstrong’s Axioms

FDs are a property of the application, not the data!

Boyce-Codd Normal Form (BCNF)

- For a relation R, with FDs of the form X→Y, every FD is either:
  1) Trivial (e.g., Y contains X), or
  2) X is a key of the table

- If an FD violates 2), multiple rows with same X value may occur
  - Indicates redundancy, as rows with given X value all have same Y value
  - E.g., SSN → Name, Addr in non-decomposed hobbies schema
    - Name, Addr repeated for each appearance of a given SSN

- To put a schema into BCNF, create subtables of form XY
  - E.g., tables where key is left side (X) of one or more FDs
  - Repeat until all tables in BCNF

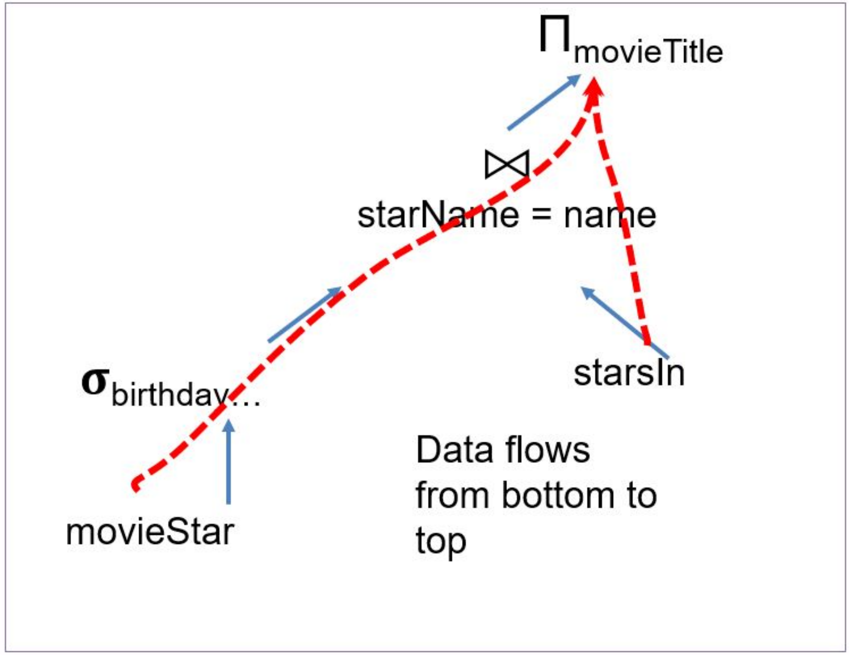
BCNFify

```
BCNFify(schema R, functional dependency set F):

D = {(R,F)}    // D is set of output relations
while there is a (schema ,FD set) pair  (S,F') in D not in BCNF, do:
    given $X \rightarrow Y$ as a BCNF-violating dependency in F’
    replace (S,F’) in D with
        S1 = (XY,F1) and
        S2 = ((S-Y) U X, F2)
    where F1 and F2 are the FDs in F’ over XY or (S-Y) U X, respectively
End

return D
```

Database storage

- Heap file + Index(tree / hash)
- Index can be clustered (records sorted on indexed attribute on disk)
- Indexes can be on multiple attributes, but usually not multi-dimensional (specialized structure such as R-Trees and Quad-trees do that)
- Data organized into pages and read into a buffer pool

Query Execution



Each operator implements a simple iterator interface:

```
open(params)
getNext() → record
```

Any iterator can compose with any other iterator

```
it1 = Scan.open("movieStar", ...)
it2 = Filter.open(it1, bday=x, ...)
it3 = Scan.open("starsIn", ...)
it4 = Join.open(it2, it3,
         starName=name)
it5 = Proj.open(it4, movieTitle)
```

Database Cost models

- Typically try to account for both CPU and I/O
  - I/O = ”input / output”, i.e., data access costs from disk

- Database algorithms try to optimize for sequential access (to avoid massive random access penalties)

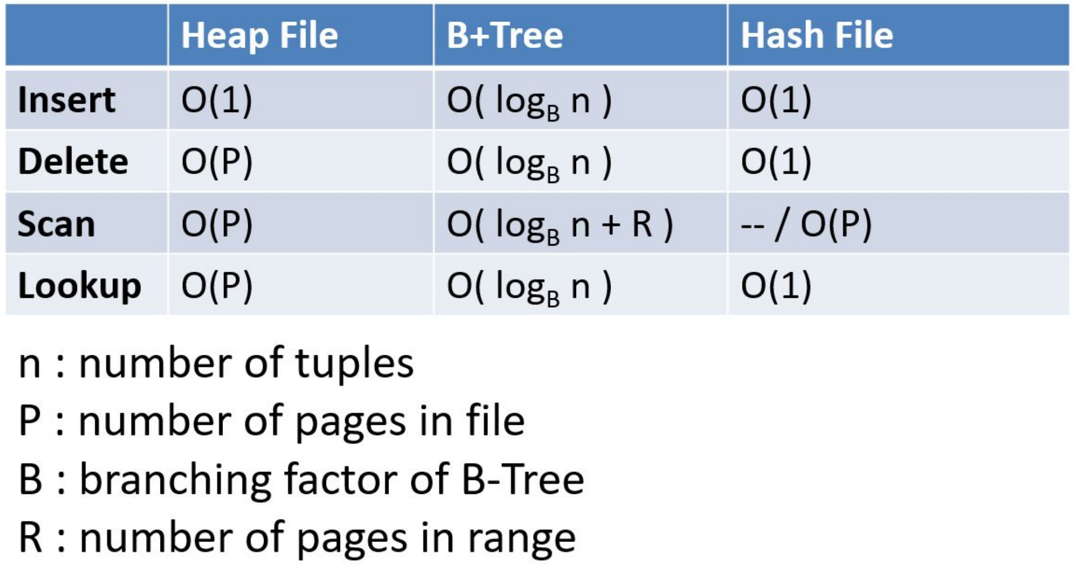
- Simplified cost model for 6.814/6.830:

  # seeks (random I/Os) x random I/O time +

  sequential bytes read x sequential B/W

Access Methods

- Accessing records in a database
- 3 main types:
  - Heap scan (sequential scan)
  - Hash index lookup/scan
  - B-Tree (or other range index) lookup/scan



	Heap File	B+Tree	Hash File
Insert	O(1)	O($\log_B$ n)	O(1)
Delete	O(P)	O($\log_B$ n)	O(1)
Scan	O(P)	O($\log_B$ n + R)	-- / O(P)
Lookup	O(P)	O($\log_B$ n)	O(1)

n : number of tuples
P : number of pages in file
B : branching factor of B-Tree
R : number of pages in range

Access Methods

Access Method	Key Features
Heap file	● Records are unsorted ● Search for records by sequentially scanning the entire file ● Use if there are no available indexes on your search key or you expect to return a large number of records
Hash index	● Typically points to an unsorted underlying heap file ● Constant time search for records ● Useful for finding a set of specific keys, *not* searching for ranges of keys ● May not be worth using if you have to perform random I/O to access a large number of records in the underlying heap file

Access Methods

Access Method	Key Features
B+ tree index	● Typically points to an unsorted underlying heap file ● Logarithmic time search for records ($\log_B n$) ● Useful for finding a set of specific keys or scanning a range of keys ● May not be worth using if you have to perform random I/O to access a large number of records in the underlying heap file
Clustered index	● Records in underlying file are sorted, eliminating need for random I/O ● Constant or logarithmic search ● Useful for finding a set of specific keys or scanning a range of keys ● Could be used as input to sort-merge join, to avoid sort step ● Can have multiple indexes per table but only one clustered index!

Column Stores

- Store columns contiguously (likely w/ compression)
- Great for analytics, somewhat slow for transactions

Compression

- Multiple types:
  - Run-length encoding <- important
  - Dictionary <- important
  - Delta Value
  - LZ
  - Block Dictionary Bitmaps
  - Null Suppression
  - Other lossless compression schemes (e.g., gzip)

- Possible to execute queries without decompressing

Join Algorithms

Join algorithm	Key Features
Nested loops	• O(nm), where n is tuples in outer, m inner • Only useful if the inner relation is very small, and therefore the overhead of building a hash table is not worth it • *Block nested loops*: Can operate on blocks of tuples of inner relation, to make more efficient; complexity is then (nB), where B is number of blocks
Index nested loops	• Only possible if you have an index on the inner relation • Efficient if the number of lookups you need to do on the index is small

Join Algorithms

Join algorithm	Key Features
In-memory hash	● If one of the tables can fit in memory, can create a hash table on it on the fly ● Pipeline lookups from other table (which may not fit in memory) ● Good choice for equality joins when there is no index
Simple hash	● Good choice if one of the tables almost fits in memory ● I/O cost is P(|R| + |S|), where P is the number of partitions you split each relation into. Each partition P must fit in memory ● |R| and |S| are the number of pages in relations R and S ● Always better to use Grace hash if P > 2
Grace hash	● Usually the best choice if neither relation can fit in memory ● I/O cost is 3(|R| + |S|)

Join Algorithms

Join algorithm	Key Features
Sort-merge	● Same I/O cost as Grace hash, but less efficient due to cost of sorting ● Could be a good choice if the relations are already sorted or you will need the output to be sorted on the join attribute later in the query plan (e.g., ORDER BY)

Join Algorithms

Algo	I/O cost	CPU cost	In Mem?
Nested loops	|R|+|S|	O({R}x{S})	R in mem
Nested loops	{S}|R| + |S|	O({R}x{S})	No
Index nested loops (R index)	|S| + {S}c (c = 1 or 2)	O({S}log{R})	No
Block nested loops	|S| + B|R| (B=|S|/M)	O({R}x{S})	No
Sort-merge	|R|+|S|	O({S}log{S})	Both
Hash (Hash R)	|R|+|S|	O({S} + {R})	R in mem
Blocked hash (Hash S)	|S| + B|R| (B=|S|/M)	O({S} + B{R})	No
External Sort-merge	3(|R| + |S|)	O(P x {S}/P log {S}/P)	No
Simple hash	P(|R|+|S|) (P=|S|/M)	O({R} + {S})	No
Grace hash	3(|R| + |S|)	O({R} + {S})	No

Query Optimization

- Cost estimation
  - Selectivity estimation - Selinger stats, Histograms etc.
  - Cost model
- Plan enumeration
  - Push down selections
  - No cross products
  - Left deep plans
  - DP with entry for every sub plan
  - DP table is filled from the smallest sub-plan to the largest
  - Interesting orders
    - Scan over primary index, sort merge join
    - DP table has one entry per interesting order

Main Memory Databases

- Key differences with disk based systems
  - Prefetching data and instructions
  - Branches
  - Function call overhead
- Vectorized (batched) processing
  - Promote sequential access
  - Amortize function calls
- Column stores
  - Expensive field offsets

You are given the following schema and SQL query:

```
dept (did int primary key, bldg int, campus int)  // 12 bytes per record
hobby (hid int primary key, hname char(17), cost int) //25 bytes per record
emp (eid int primary key, ename char(17), d int references dept.did) //25 bytes per record
hobbies (e int references emp.eid, h int references hobby.hid) //8 bytes per record
```

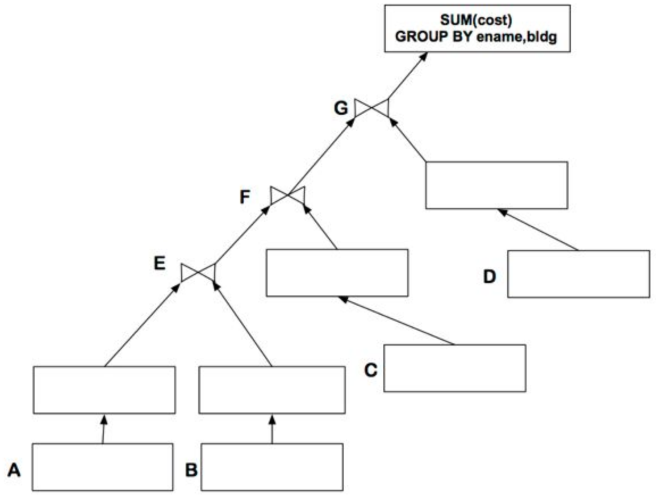
```
SELECT ename,bldg,SUM(cost)
FROM emp,dept,hobbies,hobby
WHERE emp.d = did
AND hobbies.e = eid
AND hobbies.h = hid
AND ename LIKE '%Sam%'
GROUP BY ename,bldg
```

You are given the following statistics (here, $|A|$ denotes the number of tuples in relation A, and $S(e)$ denotes the selectivity of expression e).

Statistic	Value
$\lvert emp \rvert$	10^5
$\lvert dept \rvert$	10^3
$\lvert hobby \rvert$	10^3
$\lvert hobbies \rvert$	2×10^5
S(ename LIKE '%Sam%')	.1

Assume an integer is 4 bytes and a character is 1 byte, and that a disk page is 1000 bytes. Suppose you are running in a system with 100 pages of memory. Assume that each employee has about the same number of hobbies, and that hobbies and departments are assigned to employees uniformly and at random.

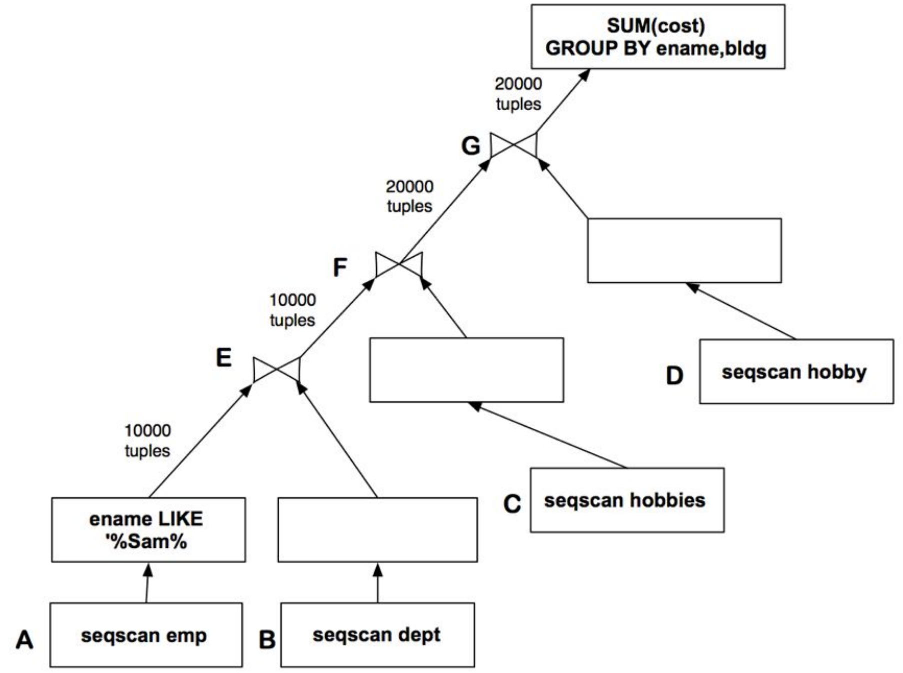
5. [10 points]: For now, assume that each join is a nested loops join and that there are no indices and no projection operations. In the diagram below, in the boxes labeled A, B, C, and D, write the names of the relations in the optimal left-deep join order for this plan. You should place the relation in the outer loop of the join in the leftmost box. Also indicate where the filters from the query should be placed. Some boxes may be empty. You should assume that the 100 pages of memory are optimally allocated between the joins to minimize the amount of I/O required by the plan.



6. [6 points]: Estimate the total number of disk pages read by the plan you drew above (do not worry about seeks for this problem).

7. [6 points]: Estimate the number of tuples produced by the plan you drew above.

5. [10 points]: For now, assume that each join is a nested loops join and that there are no indices and no projection operations. In the diagram below, in the boxes labeled A, B, C, and D, write the names of the relations in the optimal left-deep join order for this plan. You should place the relation in the outer loop of the join in the leftmost box. Also indicate where the filters from the query should be placed. Some boxes may be empty. You should assume that the 100 pages of memory are optimally allocated between the joins to minimize the amount of I/O required by the plan.



Answer:

The hobby and dept tables can both fit into RAM. The emp and hobbies do not. By doing the emp-dept join first, with emp as the outer, we are able to scan emp just once. We only scan dept once because it fits into memory. We have to do the join with hobbies next (because the join with hobby would be a cross product). Due to the constraints of left-deep plans, we must put the hobbies table on the inner, requiring us to scan it once for each tuple output by the emp/dept join (10^4 times). Finally, we can do the join with hobby; since hobby fits into memory, we only scan it once as well.

The following table summarizes the pages used by each of the tables:

Table	Formula	No. pages
emp	$10^5/(1000/25)$	2500
dept	$10^3/(1000/12)$	12
hobby	$10^3/(1000/25)$	25
hobbies	$2 \times 10^5/(1000/8)$	1600

6. [6 points]: Estimate the total number of disk pages read by the plan you drew above (do not worry about seeks for this problem).

(Write your answer in the space below.)

Answer: The total I/O cost is one scan of dept + one scan of hobby + one scan of emp + 10^4 scans of hobbies. Emp is 2500 pages, dept is 12 pages, and hobbies is 25 pages. Summing these numbers, we get: $1.6x10^7 + 2500 + 12 + 25 \approx 1.6x10^7$ pages

7. [6 points]: Estimate the number of tuples produced by the plan you drew above.

(Write your answer in the space below.)

The top-most join produces 20,000 records, but this represents only 10,000 distinct employees, so the GROUP BY will output 10,000 values (assuming employee name are unique.)