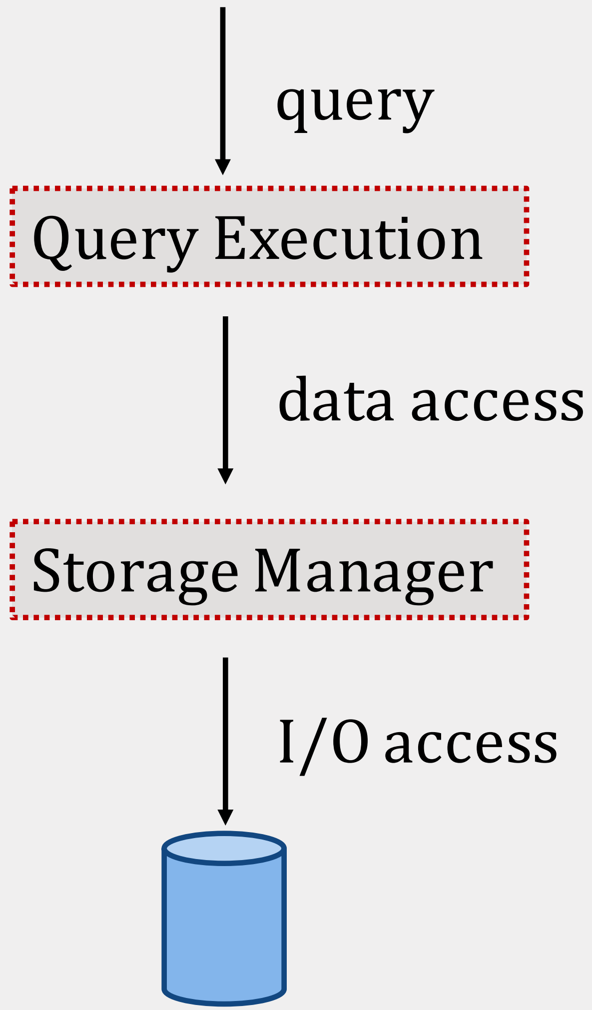
RELATIONAL OPERATORS

CS 564- Fall 2016

ACKs: *Jeff Naughton, Jignesh Patel, AnHai Doan*

ARCHITECTURE OF A DBMS



LOGICAL VS PHYSICAL OPERATORS

- Logical operators
  - *what* they do
  - e.g., union, selection, project, join, grouping

- Physical operators
  - *how* they do it
  - e.g., nested loop join, sort-merge join, hash join, index join

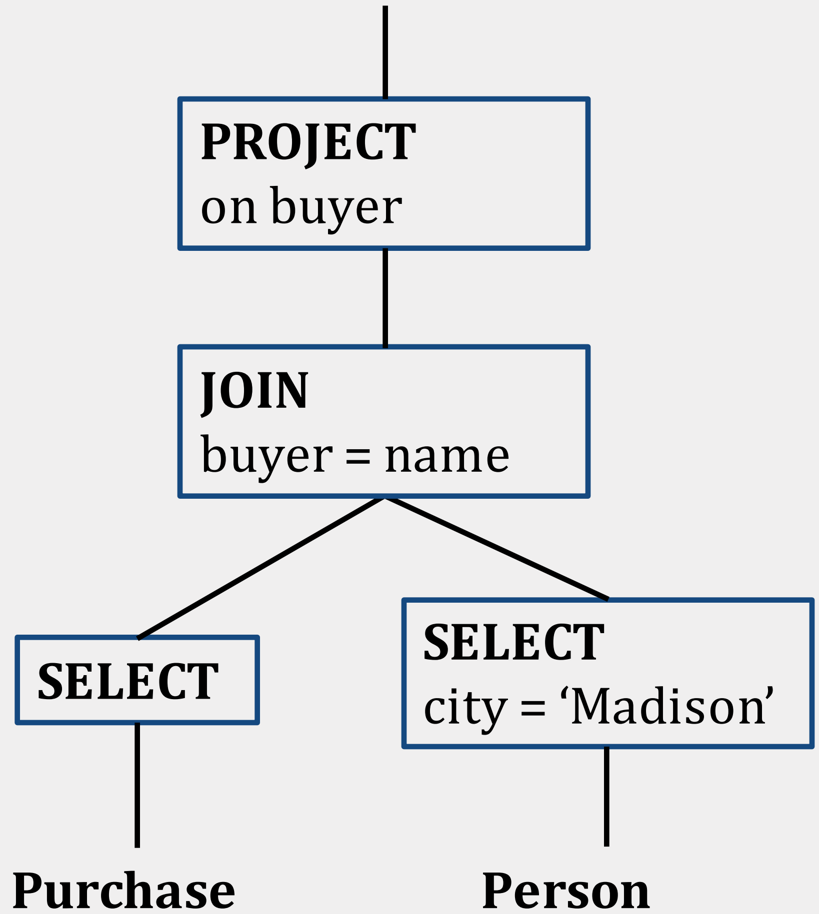
EXAMPLE QUERY

```
SELECT P.buyer
FROM   Purchase P, Person Q
WHERE  P.buyer=Q.name
AND    Q.city='Madison'
```

- Assume that Person has a B+ tree index on city

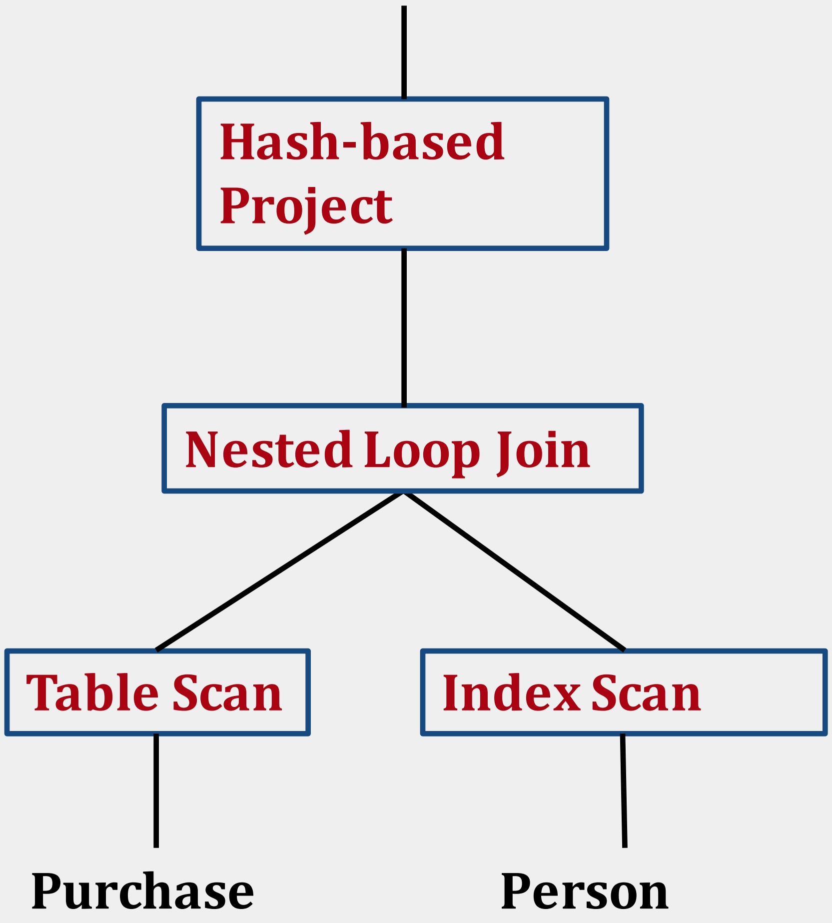
EXAMPLE: LOGICAL PLAN

```
SELECT P.buyer
FROM   Purchase P, Person Q
WHERE  P.buyer=Q.name
AND    Q.city='Madison'
```



Example: Physical Plan

```
SELECT P.buyer
FROM   Purchase P, Person Q
WHERE  P.buyer=Q.name
AND    Q.city='Madison'
```



RELATIONAL OPERATORS

We will see implementations for the following relational operators:

- select
- project
- join
- aggregation
- set operators

SELECT

SELECT OPERATOR

access path = way to retrieve tuples from a table

- **File Scan**
  - scan the entire file
  - I/O cost: O(N), where N = #pages
- **Index Scan:**
  - use an index available on some predicate
  - I/O cost: it varies depending on the index

INDEX SCAN COST

I/O cost for index scan

- Hash index: O(1)
  - but we can only use it with equality predicates
- B+ tree index: $O(\log_F N) + X$
  - X depends on whether the index is clustered or not:
    - *unclustered*: X = # selected tuples
    - *clustered*: X = (#selected tuples)/ (#tuples per page)

B+ Tree Scan Example

Example

- A relation with 1M records
- 100 records on a page
- 500 (key, rid) pairs on a page

	1% Selectivity	**10% Selectivity**
clustered	3+100	3+1000
unclustered	3+10,000	3+100,000
unclustered + sorting	3+(~10,000)	3+(~10,000)

General Selection Condition

- So far we studied selection on a single attribute
- How do we use indexes when we have multiple selection conditions?
  - `R.a = 10` **`AND`** `R.b > 10`
  - `R.a = 10` **`OR`** `R.b < 20`

INDEX MATCHING

- We say that an index *matches* a selection predicate if the index can be used to evaluate it
- Consider a conjunction-only selection. An index matches (part of) a predicate if
  - Hash: only equality operation & the predicate includes *all* index attributes
  - B+ Tree: the attributes are a prefix of the search key (any ops are possible)

EXAMPLE

- A relation `R(a,b,c,d)`
- Does the index match the predicate?

Predicate	**B+ tree on (a,b,c)**	**Hash index on (a,b,c)**
a=5 **AND** b=3	yes	no
a>5 **AND** b<4	yes	no
b=3	no	no
a=5 **AND** c>10	yes	no
a=5 **AND** b=3 **AND** c=1	yes	yes
a=5 **AND** b=3 **AND** c=1 **AND** d >6	yes	yes

a=5 and b=3 and c=1 are primary conjuncts here

Index Matching

- A predicate can match more than one index
- Example:
  - hash index on (a) and B+ tree index on (b, c)
  - predicate: `a=7 AND b=5 AND c=4`
  - which index should we use?
    1. use either index
    2. use both indexes, then intersect the rid sets, and then fetch the tuples

Choosing The Right Index

- Selectivity of an access path = *fraction* of data pages that need to be retrieved
- We want to choose the *most selective* path!
- Estimating the selectivity of an access path is a hard problem

ESTIMATING SELECTIVITY

- Predicate: `a=3 AND b=4 AND c=5`
- hash index on (a,b,c)
  - selectivity is approximated by *#pages / #keys*
  - #keys is known from the index
- hash index on (b)
  - multiply the *reduction factors* for each primary conjunct
  - *reduction factor = #pages/#keys*
  - if #keys is unknown, use 1/10 as default value
  - this assumes independence of the attributes!

ESTIMATING SELECTIVITY

- Predicate: `a > 10 AND a < 60`
- If we have a range condition, we assume that the values are uniformly distributed
- The selectivity will be approximated by $\frac{interval}{High - Low}$

Predicates with Disjunction

- hash index on (a) **+** hash index on (b)
  - `a=7 or b>5`
  - a file scan is required
- hash index on (a) **+** B+ tree on (b)
  - `a=7 or b>5`
  - scan or use both indexes (fetch rids and take the union)
- hash index on (a) **+** B+ tree on (b)
  - `(a=7 or c>5) and b > 5`
  - we can use the B+ tree

PROJECT

PROJECT OPERATOR

Simple case: **SELECT** `R.a, R.d`

- scan the file and for each tuple output R.a, R.d

Hard case: **SELECT DISTINCT** `R.a, R.d`

- project out the attributes
- eliminate *duplicate tuples* (this is the difficult part!)

PROJECT: SORT-BASED

Naïve algorithm:

1. scan the relation and project out the attributes
2. sort the resulting set of tuples using all attributes
3. scan the sorted set by comparing only adjacent tuples and discard duplicates

RUNNING EXAMPLE

R(a, b, c, d, e)

- M = 1000 pages
- B = 20 buffer pages
- Each field in the tuple has the same size
- Suppose we want to project on attribute **a**

Sort-Based Cost Analysis

- initial scan = *1000 I/Os*
- after projection T =(1/5)*1000 = 200 pages
- cost of writing T = *200 I/Os*
- sorting in 2 passes = 2 * 2 * 200 = *800 I/Os*
- final scan = *200 I/Os*

total cost = 2200 I/Os

Project: Sort-Based

We can improve upon the naïve algorithm by modifying the sorting algorithm:

1. In Pass **0** of sorting, project out the attributes
2. In subsequent passes, eliminate the duplicates while merging the runs

Sort-Based Cost Analysis

- we can sort in 2 passes
- first pass costs *1000 + 200 = 1200 I/Os*
- the second pass costs *200 I/Os* (not counting writing the result to disk)

total cost = 1400 I/Os

Project: Hash-Based

2-phase algorithm:

- **partitioning**
  - project out attributes and split the input into B-1 partitions using a hash function *h*
- **duplicate elimination**
  - read each partition into memory and use an in-memory hash table (with a *different* hash function) to remove duplicates

Project: Hash-Based

When does the hash table fit in memory?

- size of a partition = $T\ /\ (B - 1)$, where T is #pages after projection
- size of hash table = $f \cdot T\ /\ (B - 1)$, where is a fudge factor (typically ~ 1.2)
- So, it must be $B\ >\ f \cdot T\ /\ (B - 1)$, or approximately $B > \sqrt{f \cdot T}$

Hash-Based Cost Analysis

- T = 400 so the hash table fits in memory!
- partitioning cost = 1000 + 200 = 1200 I/Os
- duplicate elimination cost = 200 I/Os

total cost = 1400 I/Os

COMPARISON

- Benefits of sort-based approach
  - better handling of skew
  - the result is sorted

- The I/O costs are the same if $B^2 > T$
  - 2 passes are needed by both algorithms

PROJECT: INDEX-BASED

- Index-only scan
  - Projection attributes subset of index attributes
  - apply projection algorithm only to data entries
- If an *ordered index* contains all projection attributes as prefix of search key:
  1. retrieve index data entries in order
  2. discard unwanted fields
  3. compare adjacent entries to eliminate duplicates

JOIN

JOIN OPERATOR

Algorithms for equijoin:

```
SELECT *
FROM   R, S
WHERE  R.a = S.a
```

Why can't we compute it as cartesian product?

JOIN ALGORITHMS

Algorithms for equijoin:

- nested loop join
- block nested loop join
- index nested loop join
- block index nested loop join
- sort merge join
- hash join

NESTED LOOP JOIN (1)

- for each page P_R in **R**
  - for each page P_S in **S**
    - join the tuples on P_R with the tuples in P_S

The I/O cost is $M_R + M_S \cdot M_R$

- M_R = number of pages in **R**
- M_S = number of pages in **S**

NESTED LOOP JOIN (2)

- Which relation should be the outer relation in the loop?
  - The smaller of the two relations
- How many buffer pages do we need?
  - only 3 pages suffice

BLOCK NESTED LOOP JOIN (1)

- for each block of *B-2* pages from **R**
  - for each page P_S in **S**
    - join the tuples from the block with the tuples in P_S

The I/O cost is $M_R + M_S \cdot \left\lceil \frac{M_R}{B-2} \right\rceil$

Block Nested Loop Join (2)

- To increase CPU efficiency, create an in-memory hash table for each block
  - what will be the key of the hash table?

- What happens if **R** fits in memory?

INDEX NESTED LOOP JOIN

S has an index on the join attribute

- for each page P_R in **R**
  - for each tuple r in **R**
    - probe the index of **S** to retrieve any matching tuples

The I/O cost is $M_R + |R| \cdot I^*$

- I^* depends on the type of index and whether it is clustered or not

Block Index Nested Loop Join

- for each block of *B-2* pages in **R**
  - sort the tuples in the block
  - for each tuple *r* in the block
    - probe the index of **S** to retrieve any matching tuples

- Why do we need to sort here?

Sort Merge Join (1)

The simple version:

- sort R and S on the join attribute
- read the sorted relations in the buffer and merge

The I/O cost is $sort(R) + sort(S) + M_R + M_S$

- careful when a join value appears many times!

Sort Merge Join (2)

- Generate sorted runs of size B for **R** and **S**
- Merge the sorted runs for **R** and **S**
  - while merging check for the join condition

The I/O cost is $3(M_R + M_S)$

- the algorithm works only if $B > \sqrt{L}$, where L is the number of pages of the largest relation!

HASH JOIN (1)

Start with a hash function h on the join attribute

- partition **R** and **S** into k partitions using h
- join each partition of **R** with the corresponding partition of **S** (using an in-memory hash table)

The I/O cost is $3(M_R + M_S)$

- but only if it fits in memory

Hash Join (2)

- $k = B\text{-}1$
- The hash table has fudge factor f
- If we construct the hash tuble for the smaller relation of size M:
  - $B > \frac{fM}{B-1} + 2$
  - so approximately $B > \sqrt{fM}$

Comparison of Join Algorithms

Hash Join **vs** Block Nested Loop Join

- the same if smaller table fits into memory
- otherwise, hash join is much better

Comparison of Join Algorithms

Hash Join **vs** Sort Merge Join

- Suppose $M_R > M_S$
- To do a two-pass join, SMJ needs $B > \sqrt{M_R}$
  - the IO cost is: $3(M_R + M_S)$
- To do a two-pass join, HJ needs $B > \sqrt{M_S}$
  - the IO cost is: $3(M_R + M_S)$

General Join Conditions

- Equalities over multiple attributes
  - e.g., *R.sid=S.sid* **and** *R.rname=S.sname*
  - for Index NL
    - index on <*sid, sname*>
    - index on *sid* or *sname*
  - for SMJ and HJ, we can sort/hash on combination of join attributes

GENERAL JOIN CONDITIONS

- Inequality conditions
  - e.g., *R.rname < S.sname*
  - For Index NL, need (clustered) B+ tree index
  - SMJ and HJ not applicable
  - Block NL likely to be the winner (why?)

SET OPERATIONS & AGGREGATION

SET OPERATIONS

- Intersection is a special case of a join
- Union and difference are similar
- Sorting:
  - sort both relations (on *all attributes*)
  - merge sorted relations eliminating duplicates
- Hashing:
  - partition R and S
  - build in-memory hash table for partition R_i
  - probe with tuples in S_i, add to table if not a duplicate

AGGREGATION: SORTING

- sort on group by attributes (if any)
- scan sorted tuples, computing running aggregate
  - max/min: max/min
  - average: sum, count
- when the group by attribute changes, output aggregate result
- **cost** = sorting cost

AGGREGATION: HASHING

- Hash on group by attributes (if any)
  - Hash entry = group attributes + running aggregate
- Scan tuples, probe hash table, update hash entry
- Scan hash table, and output each hash entry
- **cost** = scan relation
- What happens if we have many groups?

AGGREGATION: INDEX

- Without grouping
  - Can use B+ tree on aggregate attribute(s)
- With grouping
  - B+ tree on all attributes in SELECT, WHERE and GROUP BY clauses
    - Index-only scan
    - If group-by attributes prefix of search key, the data entries/tuples are retrieved in group-by order

RECAP

Implementation of relational operators:

- select, project, join, set operators, aggregation

Key ideas:

- sort-based methods
- hash-based methods
- indexes can help in certain cases