CS 133: Databases

Fall 2019
Lec 11 – 10/10
Query Evaluation
Prof. Beth Trushkowsky

Administriva

- Lab 2 -- Final version due next Wednesday
  - Insert/Delete Operators
  - BufferPool eviction policy
- Problem sets
  - PSet 5 due today
  - No PSet out this week… optional practice problems instead (Sakai)
- Midterm
  - In class next Thursday 10/17
  - Covers material through today's lecture
  - Closed book, closed notes
  - Allowed:
    - One *handwritten* cheat sheet, 8.5x11" (both sides)
    - Calculator (not actually needed)

Alternative: Hashing

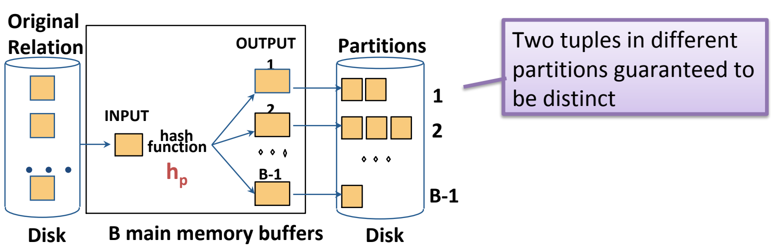
- We do not always require *order* for tuples
  - Removing duplicates
  - Forming groups
- Just need "like" things to be together
  - Hashing!
  - But how to build **hash table without staying in RAM**?

External Hashing: Divide and Conquer

- ***Divide***: Use a hash function h_p to separate records into **disk-based** partitions
- ***Conquer***: Read partitions into RAM-based hash table one at a time
  - For each partition, hash with *another* hash function h_r
- Note: Two different hash functions: h_p is coarser-grained than h_r

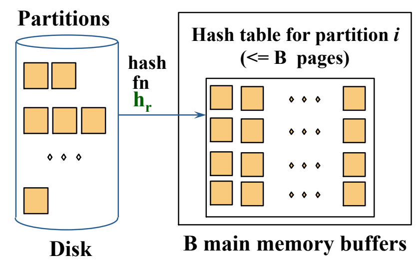
Projection: DupElim Based on Hashing

- Partition phase:
  - Read relation using one input buffer frame, retaining only necessary fields for projection
  - Hashing on h_p to yield B-1 partitions



Projection: DupElim Based on Hashing

- Duplicate Elimination phase
- For each partition:
  - Read in pages
  - Build an in-memory hash table, using second hash function h_r, and **eliminating duplicates as you go**
- If a partition does not entirely fit in buffer pool, need to *recursively* partition before this phase



Note: ignoring small overhead in memory for hash data structure

Example: Hashing DupElim

- Cost for Projection with DupElim using hashing?
  - Assume each of the partitions formed in first pass fits in buffer pool…
- For *Reserves* query:
  - Read **1000** pages
  - Write out partitions of projected tuples
    - **250** pages, because 25% of record retained
  - Read and do duplicate elimination on each partition
    - total **250** page reads
- Total : **1000** + **250** + **250** = 1500 I/Os.

Goals for Today

- Discuss algorithms for implementing query plan operators: selection, joins
- Reason about factors influencing operator cost
  - Input size (number of pages)
  - Indexes available
  - Buffer pool space
- Understand how external sorting and hashing can be used for these algorithms

Simple Selections

```
SELECT  *
FROM    Reserves R
WHERE   R.bid < 100;
```

- Of the form $\sigma_{R.attr\ op\ value}(R)$
- **Size of result** approximated as ***size of R * reduction factor***
  - "Reduction factor" also called *selectivity*
  - Statistics in Catalog can help estimate
- How best to execute a selection? Depends on:
  - What access paths are available… any indexes?
  - Expected size of the result
    (in terms of number of tuples and/or number of pages)

General Selection Conditions

```
SELECT  *
FROM    Reserves R
WHERE   R.bid = 103 AND R.sid = 42;
```

- A **B+-tree** index *matches* (a conjunction of) terms if the term(s) involve only attributes in a *prefix* of the search key.
  - E.g., Index on ***<a, b, c>*** matches predicate "***a=5 AND b= 3***", but not "***b=3***"
- For **Hash** index: index must involve all attributes in search key

Why?

General Selections: Two Approaches

- What if several indexes exist that could be used?
- Approach 1: **pick one** index to use
  - Find the *most selective access path*, retrieve tuples using it, then apply the other conditions

Most selective access path: Index estimated to require fewest page I/Os

Applying other conditions won't impact number of pages fetched

General Selections: Two Approaches

- Approach 2: **use multiple** indexes
- To use two or more matching indexes (Alt 2 or 3 for data entries):
  - Get **sets of record ids** of data records using **each** matching index.
  - Then ***intersect*** these sets of rids.
  - Retrieve the records and apply any remaining conditions
- Example: ***day > 10/10/2010*** *AND* ***bid=103*** *AND* ***sid=42***
  Suppose have **B+ tree index on *day*** and an **index on *sid***
  - Intersect: rids using index on *day* with rids using index on *sid*
  - Then check *bid=103*

Exercise 2: Selection

Exercise: Selection

- Exercise 2

  I. ~~B+tree on <bid,day>~~
  II. B+tree on <day, bid>
  III. ~~Hash index on <day, bid>~~

- Disjunction:
  - if all conditions have an index, use the *union* of rids!
  - But if even one of them does not have index, have to do sequential scan anyway

Join Operators

- Joins are a very common query operation!

- Joins can be very expensive:
  - Consider an inner join of R and S each with 1M records
    How many tuples in the answer (worst case)?

- Two main classes of JOIN algorithms:
  - Algorithms that ***enumerate cross product***
  - Algorithms that ***avoid cross product*** by getting "like" partitions together

Equality Joins on One Join Column

```
SELECT *
FROM   Reserves R, Sailors S
WHERE  R.sid=S.sid
```

- Relation info:
  - M = 1000 pages in R, t_R =100 tuples per page.
  - N = 500 pages in S, t_S = 80 tuples per page.
  - In examples, R is Reserves and S is Sailors.

- ***Cost metric*** : # of I/Os
  (We will ignore cost of final output from query)

Simple Nested Loops Join

```
foreach tuple r in R do
    foreach tuple s in S do
        if r_i == s_j then add <r, s> to result
```

- For each tuple in the ***outer*** relation R, we scan the entire ***inner*** relation S.
  - Cost: $M + (t_R * M) * N$ = 100,000*500 + 1000 I/Os.
- What if smaller relation (S) was outer?
  - $N + (t_S * N) * M$ = 40,000*1000 + 500 I/Os.

Page-Oriented Nested Loops Join

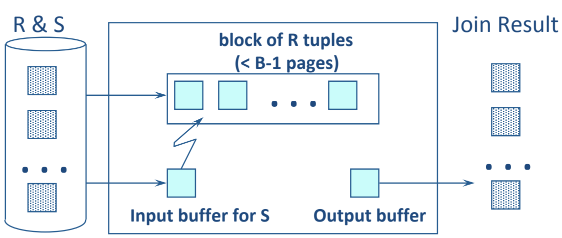
```
foreach page p_R in R do
  foreach page p_S in S do
    foreach tuple r in p_R do
      foreach tuple s in p_S do
        if r_i == s_j then add <r, s> to output page
```

Minimum buffer pool frames needed?

- For each *page* of R, get each *page* of S, and write out matching pairs of tuples <r, s>, where r is in R-page and S is in S-page.
- What is the cost of this approach? (Try Exercise 3)
- With R as outer, cost = M*N + M= 1000*500 + 1000
  - If smaller relation (S) is outer, cost = 500*1000 + 500

Block Nested Loops Join

- Page-oriented NL doesn't use all available buffer frames!
- Alternative approach:
  - Use one page as an **input buffer** for scanning **the inner S**,
  - one page as the output buffer
  - and use all remaining pages to **hold block of outer R**
  - For each ***block*** of R, scan through each page of S for matches



Block Nested Loop Join: Examples

- *Cost:* Scan of outer + # outer blocks * scan of inner

  # outer blocks = ceiling(# pages of outer/blocksize)

- With Reserves (R) as outer, and **100 pages/block**:
  - Scanning R is **1000** I/Os; a total of 10 *blocks*.
  - Per block of R, scan Sailors (S); **10*500** I/Os.

  How many times would we scan S if the **block size was B** instead of 100?

- With 100-page block of Sailors as outer:
  - Cost of scanning S is 500 I/Os; a total of 5 blocks.
  - Per block of S, scan Reserves: 5*1000 I/Os.

Avoiding Cross-product

- Simple, Page-oriented, and Block Nested-loop join algorithms **effectively enumerate the cross-product**
  - every pair of tuples is compared

- Next: algorithms that avoid cross-product (for equality joins)
  - tuples in the two relations can be thought of as belonging to ***partitions***

Index Nested Loops Join

foreach tuple r in R do
  foreach tuple s in S where s == r do
    add <r, s> to result

Index probe

- If there is an index on the join column of one relation (say S), can make that relation the inner and use the index
  - Cost: M + ((**$M*t_R$**) * **cost of finding matching S tuples**)

- Typical "**probe**" costs:
  - 1.2 I/Os for hash index
  - 2-4 I/Os for B+ tree

Probe to find matching ***data entries***

- The **cost of finding S tuples** (assuming Alt. (2) or (3) for data entries) depends on if index is clustered
  - Clustered: 1 I/O per page of matching S tuples.
  - Unclustered: up to 1 I/O per matching S tuple.

Exercise 4: Index Nested Loops

- Have Hash-index (Alt. 2) on *sid* of Sailors (as inner) Scan Reserves: **1000 page I/Os**, 100*1000 tuples.
  - For each Reserves tuple:
    - **1.2 I/Os** to get data entry in index,
    - plus **1 I/O** to get [***the exactly one***] matching Sailors tuple
      Total cost: 1000 + 2.2* 100,000 = **221,000 I/Os**

Sort-Merge Join $(R \bowtie_{i=j} S)$

- Sort R and S on the join column, then scan them to do a "merge" (on join field), and output result tuples.

- Particularly useful if
  - one or both inputs are already sorted on join field(s)
  - output is required to be sorted on join field(s)

Example of Sort-Merge Join

sid	sname	rating	age
22	dustin	7	45.0
28	yuppy	9	35.0
31	lubber	8	55.5
44	guppy	5	35.0
58	rusty	10	35.0

Instance of Sailors (outer)

sid	bid	day	rname
28	103	12/4/96	guppy
28	103	11/3/96	yuppy
31	101	10/10/96	dustin
31	102	10/12/96	lubber
31	101	10/11/96	lubber
58	103	11/12/96	dustin

Instance of Reserves (inner)

- Suppose joining on sid = sid
- Cost for this JOIN: Sort S +Sort R + (M+N)
  - The cost of merging: typically M+N

Why?

Hash-Join

(this variant: "Grace Hash Join")

- Partition both relations on the join attributes using hash function **h**
- R tuples in partition R_i will only match S tuples in partition S_i.
- For each partition *i*
  - Read in all of R_i,
  - Hash R_i on h2
  - Scan through pages of S_i, probing hash table for matches