RELATIONAL OPERATORS

CS 564- Fall 2015

ACKs: Jeff Naughton, Jignesh Patel, AnHai Doan
ARCHITECTURE OF A DBMS

query

Query Execution

data access

Storage Manager

I/O access
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**

```sql
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'
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 $N = \#\text{pages}$

- **Index Scan:**
  - use an index available on some predicate
  - I/O cost: it varies depending on the index
I/O cost for index scan:

– **Hash**: $O(1)$
  • but we can only use it with equality predicates

– **B+-tree**: $O(\log_F N) + X$
  • $X$ depends on whether the index is clustered or not:
    – *unclustered*: $X = \# \text{ selected tuples}$
    – *clustered*: $X = (\#\text{selected tuples})/ (\#\text{tuples per page})$
EXAMPLE

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

<table>
<thead>
<tr>
<th></th>
<th>1% Selection</th>
<th>10% Selection</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clustered</td>
<td>3+100</td>
<td>3+1000</td>
</tr>
<tr>
<td>Unclustered</td>
<td>3+10,000</td>
<td>3+100,000</td>
</tr>
<tr>
<td>Unclustered + sorting</td>
<td>3+(~10,000)</td>
<td>3+(~10,000)</td>
</tr>
</tbody>
</table>
So far we studied selection on a single attribute.

How do we use indexes when we have multiple selection conditions?

- $R.a = 10 \text{ and } R.b > 10$
- $R.a = 10 \text{ or } R.b < 20$

Write the condition in Conjunctive Normal Form (CNF)

- e.g. $(p_1 \text{ and } p_2) \text{ or } p_3$ can be expressed in CNF as $(p_1 \text{ or } p_3) \text{ and } (p_2 \text{ or } p_3)$
**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. Then 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).
• A relation $R(a,b,c,d)$

• Does the index match the predicate?

<table>
<thead>
<tr>
<th>Predicate</th>
<th>B+ tree on (a,b,c)</th>
<th>Hash index on (a,b,c)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a=5$ and $b=3$</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>$a&gt;5$ and $b&lt;4$</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>$b=3$</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>$a=5$ and $c&gt;10$</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>$a=5$ and $b=3$ and $c=1$</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>$a=5$ and $b=3$ and $c=1$ and $d&gt;6$</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

$a=5$ and $b=3$ and $c=1$ are primary conjuncts here
**Index Matching**

- A predicate could match more than 1 index!
- Hash index on (a) and B+ tree index on (b, c)
  - Predicate is: \( a=7 \) and \( b=5 \) and \( c=4 \)
  - Which index should we use?
    1. use either (or a file scan)
    2. use both, and then intersect the rid sets
      - Sort rids, retrieve rids in both sets
      - Side-effect: tuples retrieved in the order on disk
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!
• How do we estimate the selectivity?
ESTIMATING SELECTIVITY

• Predicate: \( a=3 \) and \( b=4 \) and \( c=5 \)

• Search key is \((a,b,c)\):
  – selectivity is approximated by \#pages / \#keys

• Search key is \((b,c)\):
  – multiply the reduction factors for each conjunct
  – reduction factor = \#pages/\#keys. If \#keys is not known, set some default value (1/10)

• If we have a range condition, we assume that the values are uniformly distributed
Predicates with Disjunction

- Hash index on (a) and hash index on (b)
  - a=7 or b>5
  - File scan required (why?)
- Hash index on (a) and B+ tree on (b)
  - a=7 or b>5
  - Scan or use both (fetch rids and take the union)
- Hash index on (a) and B+ tree on (b)
  - (a=7 or c>5) and b > 5
  - Could use 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**

The naïve algorithm:

1. Scan R 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

The cost is $O(M \log(M))$

- $M = \#\text{pages}$
Detailed Analysis

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

- M = 1000 pages
- B = 20 buffer pages
- Each field in the tuple has the same size
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
Two phases for the 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
When does the hash table fit in memory?

• size of a partition = \( T / (B - 1) \), where \( T \) is the number of pages after projection

• size of hash table = \( f \cdot T / (B - 1) \), where \( f \) is a fudge factor

• So, it must be \( B > f \cdot T / (B - 1) \), or approximately \( B > \sqrt{f \cdot T} \)
COMPARISON

• Sort-based approach
  – better handling of skew
  – result is sorted
  – I/O costs are the same if $B^2 > T$ (why?)
**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:

```sql
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 $M_R + M_S \cdot M_R$
- $M_R = \text{number of pages in } R$
- $M_S = \text{number of pages in } S$
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 $M_R + M_S \cdot \left\lceil \frac{M_R}{B-2} \right\rceil$
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 $\mathbf{R}$
  - sort the tuples in the block
  - for each tuple $r$ in the block
    - probe the index of $\mathbf{S}$ to retrieve any matching tuples

- Why do we need to sort here?
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 $\text{sort}(R) + \text{sort}(S) + M_R + M_S$

- 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)**

- $k = B - 1$
- The hash table has fudge factor $f$
- If we construct the hash table 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 \( \langle sid, sname \rangle \)
    - 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