IMPLEMENTATION OF RELATIONAL OPERATIONS
(BASED ON SLIDES FROM UC BERKELEY)
Introduction

- We’ve covered the basic underlying storage, buffering, and indexing technology.
  - Now we can move on to query processing.
- Some database operations are EXPENSIVE
- Can greatly improve performance by being “smart”
  - e.g., can speed up 1,000,000x over naïve approach
- Main weapons are:
  1. clever implementation techniques for operators
  2. exploiting “equivalencies” of relational operators
  3. using statistics and cost models to choose among these.
A Really Bad Query Optimizer

• For each Select-From-Where query block
  – Create a plan that:
    • Forms the cartesian product of the FROM clause
    • Applies the WHERE clause
    • Applies the projection of the SELECT
    • Incredibly inefficient
      – Huge intermediate results!

• Then, as needed:
  – Apply the GROUP BY clause
  – Apply the HAVING clause
  – Apply any projections and output expressions
  – Apply duplicate elimination and/or ORDER BY
Cost-based Query Sub-System

Queries

Select *
From Blah B
Where B.blah = blah

Usually there is a heuristics-based rewriting step before the cost-based steps.
The Query Optimization Game

• “Optimizer” is a bit of a misnomer...
• Goal is to pick a “good” (i.e., low expected cost) plan.
  – Involves choosing access methods, physical operators, operator orders, ...
  – Notion of cost is based on an abstract “cost model”
• Roadmap for this topic:
  – Query plan structure
  – Then: unary (single relation) operators
  – Then: joins
  – After that: optimizing multiple operators
Query Processing Overview

- The *query optimizer* translates SQL to a special internal “language”
  - Query Plans
- The *query executor* is an *interpreter* for query plans
- Think of query plans as “box-and-arrow” *dataflow* diagrams
  - Each box implements a *relational operator*
  - Edges represent a flow of tuples (columns as specified)
  - For single-table queries, these diagrams are straight-line graphs

```
SELECT DISTINCT name, gpa
FROM Students
```
Query Optimization

• A deep subject, focuses on multi-table queries
  – We will only need a cookbook version for now.
• Build the dataflow bottom up:
  – Choose an Access Method (HeapScan or IndexScan)
    • Non-trivial, we’ll learn about this later!
  – Next apply any WHERE clause filters
  – Next apply GROUP BY and aggregation
    • Can choose between sorting and hashing!
  – Next apply any HAVING clause filters
  – Next Sort to help with ORDER BY and DISTINCT
    • In absence of ORDER BY, can do DISTINCT via hashing!
Iterators

- The relational operators are all subclasses of the class `iterator`:

```cpp
class iterator {
    void init();
    tuple next();
    void close();
    iterator inputs[];
    // additional state goes here
}
```

- Note:
  - Edges in the graph are specified by inputs (max 2, usually)
  - Encapsulation: any iterator can be input to any other!
  - When subclassing, different iterators will keep different kinds of state information
class Scan extends iterator {
    void init();
    tuple next();
    void close();
    iterator inputs[1];
    bool_expr filter_expr;
    proj_attr_list proj_list;
}

Example: Scan

- **init()**:  
  - Set up internal state  
  - call init() on child – often a file open

- **next()**:  
  - call next() on child until qualifying tuple found or EOF  
  - keep only those fields in “proj_list”  
  - return tuple (or EOF -- “End of File” -- if no tuples remain)

- **close()**:  
  - call close() on child  
  - clean up internal state

Note: Scan also applies “selection” filters and “projections” (without duplicate elimination)
Example: Sort

```java
class Sort extends iterator {
    void init();
    tuple next();
    void close();
    iterator inputs[1];
    int numberOfRuns;
    DiskBlock runs[];
    RID nextRID[];
}
```

- **init():**
  - generate the sorted runs on disk
  - Allocate runs[] array and fill in with disk pointers.
  - Initialize numberOfRuns
  - Allocate nextRID array and initialize to NULLs
- **next():**
  - nextRID array tells us where we’re “up to” in each run
  - find the next tuple to return based on nextRID array
  - advance the corresponding nextRID entry
  - return tuple (or EOF -- “End of File” -- if no tuples remain)
- **close():**
  - deallocate the runs and nextRID arrays
Relational Operations

- We will consider how to implement:
  - **Selection** $\sigma$ Selects a subset of rows from relation.
  - **Projection** $\pi$ Deletes unwanted columns from relation.
  - **Join** $\times$ Allows us to combine two relations.
  - **Set-difference** $-$ Tuples in reln. 1, but not in reln. 2.
  - **Union** $\cup$ Tuples in reln. 1 and in reln. 2.
  - **Aggregation** (SUM, MIN, etc.) and **GROUP BY**

- Since each op returns a relation, ops can be *composed!*
  After we cover the operations, we will discuss how to *optimize* queries formed by composing them.
Selection (filter) Operators
Schema for Examples

Sailors (sid: integer, sname: string, rating: integer, age: real)
Reserves (sid: integer, bid: integer, day: date, rname: string)

• Similar to old schema; rname added for variations.
• Reserves:
  – Each tuple is 40 bytes long, 100 tuples per page, 1000 pages.
• Sailors:
  – Each tuple is 50 bytes long, 80 tuples per page, 500 pages.
Simple Selections

- Of the form $\sigma_{R.\text{attr} \ op \ \text{value}} (R)$
- Question: how best to perform? Depends on:
  - what indexes/access paths are available
  - what is the expected size of the result (in terms of number of tuples and/or number of pages)
- **Size of result** approximated as $\text{size of } R \ast \text{reduction factor}$
  - “reduction factor” is usually called selectivity.
  - estimate of reduction factors is based on statistics – we will discuss shortly.

```sql
SELECT *
FROM Reserves R
WHERE R.rname < 'C%'
```
Alternatives for Simple Selections

• With no index, unsorted:
  – Must essentially scan the whole relation
  – cost is $M$ (#pages in R). For “Reserves” = 1000 I/Os.

• With no index, sorted on $rname$:
  – cost of binary search + number of pages containing results.
  – For reserves = 10 I/Os + \([selectivity]*\#pages\]

• With an index on selection attribute:
  – Use index to find qualifying data entries,
  – then retrieve corresponding data records.
  – (Hash index useful only for equality selections.)
Using an Index for Selections

• Cost depends on #qualifying tuples, and clustering.
  – Cost:
    • finding qualifying data entries (typically small)
    • plus cost of retrieving records (could be large w/o clustering).
  – In example “reserves” relation, if 10% of tuples qualify (result size estimate: 100 pages, 10000 tuples).
    • With a clustered index, cost is little more than 100 I/Os;
    • if unclustered, could be more than 10000 I/Os! unless...
Selections using Index (cont)

• *Important refinement for unclustered indexes:*
  1. Find qualifying data entries.
  2. Sort the rid’s of the data records to be retrieved.
  3. Fetch rids in order. This ensures that each data page is looked at just once (though # of such pages likely to be higher than with clustering).
General Selection Conditions

\[(day < 8/9/94 \text{ AND } rname = 'Paul') \text{ OR bid}=5 \text{ OR sid}=3\]

- Such selection conditions are first converted to **conjunctive normal form (CNF)**:
  \[\text{AND} \quad (day < 8/9/94 \text{ OR bid}=5 \text{ OR sid}=3) \text{ AND} \quad (rname = 'Paul' \text{ OR bid}=5 \text{ OR sid}=3)\]

- We only discuss the case with no **ORs** (a conjunction of **terms** of the form *attr op value*).

- A **B-tree index** **matches** (a conjunction of) terms that involve only attributes in a **prefix** of the search key.
  - Index on \(<a, b, c>\) matches \(a=5 \text{ AND } b=3\), but not \(b=3\).

- For **Hash index**, must have all attributes in search key
Composite Search Keys

• Search on a combination of fields.
  – Equality query: Every field value is equal to a constant value. E.g. wrt \(<age, sal>\) index:
    • age=20 and sal =75
  – Range query: Some field value is not a constant. E.g.:
    • age > 20; age=12 and sal > 10
• Data entries in index sorted by search key to support range queries.
  – Lexicographic order
  – Like the dictionary, but on fields, not letters!
Two Approaches to General Selections

- **First approach:** Find the *most selective access path*, retrieve tuples using it, and apply any remaining terms that don’t match the index:
  - *Most selective access path:* An index or file scan that we estimate will require the fewest page I/Os.
  - *Terms that match* this index reduce the number of tuples retrieved; other terms are used to discard some retrieved tuples, but do not affect number of tuples/pages fetched.
Most Selective Index - Example

- Consider $\text{day} < 8/9/94$ AND $\text{bid}=5$ AND $\text{sid}=3$.
- A B+ tree index on $\text{day}$ can be used;
  - then, $\text{bid}=5$ and $\text{sid}=3$ must be checked for each retrieved tuple.
- Similarly, a hash index on $<\text{bid}, \text{sid}>$ could be used;
  - Then, $\text{day}<8/9/94$ must be checked.
- How about a B+tree on $<\text{rname},\text{day}>$?
- How about a B+tree on $<\text{day}, \text{rname}>$?
- How about a Hash index on $<\text{day}, \text{rname}>$?
Intersection of Rids

- **Second approach**: if we have 2 or more matching indexes (w/Alternatives (2) or (3) for data entries):
  - Get *sets of rids* of data records using *each* matching index.
  - Then *intersect* these *sets of rids*.
  - Retrieve the records and apply any remaining terms.

- Consider *day<8/9/94 AND bid=5 AND sid=3*. With a

- **B+ tree index on** *day* and an index on *sid*, we can retrieve rids of records satisfying *day<8/9/94* using the first, rids of recs satisfying *sid=3* using the second, intersect, retrieve records and check *bid=5*.
  - Note: commercial systems use various tricks to do this:
    - bit maps, bloom filters, index joins
Join Operators
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. Q: How many tuples in the answer? (cross product in worst case, 0 in the best(?))

• Many join algorithms have been developed
• Can have very different join costs.
Equality Joins With One Join Column

```
SELECT  *
FROM    Reserves R1, Sailors S1
WHERE   R1.sid=S1.sid
```

• Assume:
  – M = 1000 pages in R, \( p_R = 100 \) tuples per page.
  – N = 500 pages in S, \( p_S = 80 \) tuples per page.
  – In our examples, R is Reserves and S is Sailors.

• **Cost metric** : # of I/Os. We will ignore output costs.

• We will consider more complex join conditions later.
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.
- How much does this Cost?
- \((p_R * M) * N + M = 100,000*500 + 1000\) I/Os.
  - At 10ms/IO, Total: ???
- What if smaller relation (S) was outer?
- \((p_S * N) * M + N = 40,000*1000 + 500\) I/Os.
- What assumptions are being made here?
Page-Oriented Nested Loops Join

foreach page b_{R} in R do
  foreach page b_{S} in S do
    foreach tuple r in b_{R} do
      foreach tuple s in b_{S} do
        if r_{i} == s_{j} then add <r, s> to result

• 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?

• M*N + M = 1000*500 + 1000
  – If smaller relation (S) is outer, cost = 500*1000 + 500