CompSci 516
Data Intensive Computing Systems

Lecture 11
Query Optimization

Instructor: Sudeepa Roy

Announcements

• HW2 has been posted on sakai
  – Due on Oct 18, 11 pm
  – Start early!
• Midterm next week
  – Wednesday, Oct 11, in class
  – Closed book, closed electronic devices
  – Everything until and including Lecture 12

Reading Material

• [RG]
  – Query optimization: Chapter 15 (overview only)
• [GUW]
  – Chapter 16.2 - 16.7
• Original paper by Selinger et al.:
  – No need to understand the whole paper, but take a look at the example (link on the course webpage)

Acknowledgement:
• The following slides have been created adapting the instructor material of the [RG] book provided by the authors Dr. Ramakrishnan and Dr. Gehrke.
• Some of the following slides have been created by adapting slides by Profs. Shivnath Babu and Magda Balazinska.

Query Evaluation and Operator Algorithm

Continued from Lecture 10

Hash-Join

• Partition both relations using hash function \( h \)
• \( R \) tuples in partition \( i \) will only match \( S \) tuples in partition \( i \)
• Read in a partition of \( R \), hash it using \( h_2 (\neq h) \).
• Scan matching partition of \( S \), search for matches.

Cost of Hash-Join

• In partitioning phase
  – \( \text{read+write both relns; } 2(M+N) \)
  – In matching phase, read both relns; \( M+N \) I/Os
  – remember – we are not counting final write
• In our running example, this is a total of 4500 I/Os
  – \( 3 \ast (1000 + 500) \)
  – Compare with the previous joins
Sort-Merge Join vs. Hash Join

- Both can have a cost of $3(M+N)$ I/Os
  - if sort-merge gets enough buffer (see 14.4.2)
- Hash join holds smaller relation in buffer—better if limited buffer
- Hash Join shown to be highly parallelizable
- Sort-Merge less sensitive to data skew
  - also result is sorted

General Join Conditions

- Equalities over several attributes
  - e.g., $R$.sid=$S$.sid AND $R$.name=$S$.name
  - For Index Nested Loop, build index on $<$sid, sname$>$ (if S is inner); or use existing indexes on sid or sname
  - For Sort-Merge and Hash Join, sort/partition on combination of the two join columns.

- Inequality conditions
  - e.g., $R$.name < $S$.name
  - For Index NL, need (clustered) B+ tree index.
  - Hash Join, Sort Merge Join not applicable

Review: Join Algorithms

- Nested loop join:
  - for all tuples in $R$; for all tuples in $S$...
  - variations: block-nested, index-nested
- Sort-merge join
  - like external merge sort
- Hash join

  Make sure you understand how the I/O varies

- No one join algorithm is uniformly superior to others
  - depends on relation size, buffer pool size, access methods, skew

Algorithms for Selection

- No index, unsorted data
  - Scan entire relation
  - May be expensive if not many 'Joe's
- No index, sorted data (on 'name')
  - locate the first tuple, scan all matching tuples
  - first binary search, then scan depends on matches
- B+ tree index, Hash index
  - Discussed earlier
  - Cost of accessing data entries + matching data records
  - Depends on clustered/unordered

More complex condition like $day<8/9/94$ AND $bid=5$ AND $sid=3$
  - Either use one index, then filter
  - Or use two indexes, then take intersection, then apply third condition
  - etc.

Algorithms for Projection

- Two parts
  - Remove fields: easy
  - Remove duplicates (if distinct is specified): expensive
- Sorting-based
  - Sort, then scan adjacent tuples to remove duplicates
  - Can eliminate unwanted attributes in the first pass of merge sort
- Hash-based
  - Exactly like hash join
  - Partition only one relation in the first pass
  - Remove duplicates in the second pass
- Sort vs Hash
  - Sorting handles skew better, returns results sorted
  - Hash table may not fit in memory — sorting is more standard
- Index-only scan may work too
  - If all required attributes are part of index

Algorithms for Set Operations

- Intersection, cross product are special cases of joins
- Union, Except
  - Sort-based
  - Hash-based
  - Very similar to joins and projection
Algorithms for Aggregate Operations

- **SUM, AVG, MIN etc.**
  - again similar to previous approaches

- **Without grouping:**
  - In general, requires scanning the relation.
  - Given index whose search key includes all attributes in the `SELECT` or `WHERE` clauses, can do index-only scan

- **With grouping:**
  - Sort on group-by attributes
  - or, hash on group-by attributes
  - can combine sort/hash and aggregate
  - can do index-only scan here as well

Query Optimization

Cost Estimation

- For each plan considered, must estimate cost:
  - Must estimate cost of each operation in plan tree.
    - Depends on input cardinalities
    - We’ve discussed how to estimate the cost of operations (sequential scan, index scan, joins, etc.)
  - Must also estimate size of result for each operation in tree
    - gives input cardinality of next operators
  - Also consider
    - whether the output is sorted
    - intermediate results written to disk

Query Blocks: Units of Optimization

- Query Block
  - No nesting
  - One `SELECT`, one `FROM`
  - At most one `WHERE`, `GROUP BY`, `HAVING`

- SQL query
- => parsed into a collection of query blocks
- => the blocks are optimized one block at a time

- Express single-block it as a relational algebra (RA) expression

Relational Algebra Equivalences

- Allow us to choose different join orders and to ‘push’ selections and projections ahead of joins.

  - **Selections:**
    \[ \sigma_{\text{A}_1 \ldots \text{A}_n}(R) = \sigma_{\text{A}_1}( \ldots \sigma_{\text{A}_n}(R) ) \] (Cascade)
    \[ \sigma_{\text{A}_1}(\sigma_{\text{A}_2}(R)) = \sigma_{\text{A}_2}(\sigma_{\text{A}_1}(R)) \] (Commute)

  - **Projections:**
    \[ \pi_{\text{A}_1}(R) = \pi_{\text{A}_1}( \ldots \pi_{\text{A}_n}(R) ) \] (Cascade)

  - **Joins:**
    \[ R \bowtie (S \bowtie T) \equiv (R \bowtie S) \bowtie T \] (Associative)
    \[ (R \bowtie S) \equiv (S \bowtie R) \] (Commute)

Notation

- **T(R):** Number of tuples in R
- **B(R):** Number of blocks (pages) in R
- **V(R, A):** Number of distinct values of attribute A in R

There are many more intuitive equivalences, see 15.3.4 for details
Query Optimization Problem

Pick the best plan from the space of physical plans

Cost-based Query Optimization

Pick the plan with least cost

Challenge:

• Do not want to execute more than one plans
• Need to estimate the cost without executing the plan

Cost-based Query Optimization

Pick the plan with least cost

Tasks:
1. Estimate the cost of individual operators 
2. Estimate the size of output of individual operators
3. Combine costs of different operators in a plan
4. Efficiently search the space of plans

Cost of Table Scan

```
R
Table Scan
R
```

Cost: $B(R)$
Size: $T(R)$

Desired Properties of Estimating Sizes of Intermediate Relations

Ideally,

• should give accurate estimates (as much as possible)
• should be easy to compute
• should be logically consistent
  – size estimate should be independent of how the relation is computed (e.g. which join algorithm/join order is used)
• But, no “universally agreed upon” ways to meet these goals

Task 1 and 2

Estimating cost and size of different operators

• Size = #tuples, NOT #pages
• Cost = #page I/O
  • but, need to consider whether the intermediate relation fits in memory, is written back/to/read from disk (or on-the-fly goes to the next operator), etc.
Cost of Index Scan

\[ R \]

\[ \text{Index Scan} \]

Cost: \( B(R) \) – if clustered \( T(R) \) – if unclustered

Size: \( T(R) \)

\[ R \]

Note:
1. Size is independent of the implementation of the scan/index
2. Index scan is bad if unclustered

\[ R \]

\[ \text{Index Scan} \]

Cost of Index Scan with Selection

\[ R \]

\[ \sigma_{R.A > 50} R \]

\[ \text{Index Scan} \]

Cost: \( B(R) \times f \) – if clustered \( T(R) \times f \) – if unclustered

Size: \( T(R) \times f \)

Reduction factor
\[ f = \frac{\text{Max}(R.A) - 50}{\text{Max}(R.A) - \text{Min}(R.A)} \]

assumes uniform distribution

\[ R \]

Cost of Projection

\[ \pi_{R.A} R \]

Cost: depends on the method of scanning \( R \)

Size: \( T(R) \)

But tuples are smaller
If you have more information on the size of the smaller tuples, can estimate #I/O better

\[ R \]

\[ \Join R.A = S.B \]

Size of Join

Quite tricky
- If disjoint A and B values
  - If A is key of R and B is foreign key of S
    - then \( T(S) \)
  - If all tuples have the same value of \( R.A = S.B = s \)
    - then \( T(R) \times T(S) \)

\[ R \]

\[ S \]

\[ \Join R.A = S.B \]

Two standard assumptions
1. Containment of value sets:
   - if \( V(R, A) \subseteq V(S, B) \), then all A-values of R are included in B-values of S
   - e.g. satisfied when A is foreign key, B is key

2. Preservation of value sets:
   - If \( R.A = S.A = V(R, A) \)
     - If all tuples have the same value of \( V(R, A) \)
   - No value is lost in join

\[ R \]

\[ S \]
Size of Join

Reduction factor
\[ f = \frac{1}{\max(V(R, A), V(S, B))} \]

Size
\[ \text{Size} = T(R) \times T(S) \times f \]

T(R): Number of tuples in R
B(R): Number of blocks in R
V(R, A): Number of distinct values of attribute A in R

\[ R.A = S.B \]

Example Query

Task 3: Combine cost of different operators in a plan

With Examples “Given” the physical plan

• Size = #tuples, NOT #pages
• Cost = #page I/O
• but, need to consider whether the intermediate relation fits in memory, is written back to disk (or on-the-fly goes to the next operator) etc.

Assumptions

• Student: S, Book: B, Checkout: C
• Sid, bid foreign key in C referencing S and B resp.
• There are 10,000 Student records stored on 1,000 pages.
• There are 50,000 Book records stored on 5,000 pages.
• There are 300,000 Checkout records stored on 15,000 pages.
• There are 500 different authors.
• Student ages range from 7 to 24.

Warning: a few dense slides next 😊

Physical Query Plan – 1

Q. Compute
1. the cost and cardinality in steps (a) to (d)
2. the total cost

Assumptions (given):
• Data is not sorted on any attributes.
• For both in (a) and (b), outer relations fit in memory
Index
Block nested loop
S inner

Indexed, B outer, C inner

Checkout C
(Index scan)

Book B
(Index scan)

On the fly
Cardinality = 100

Cost = 0 (on the fly)

(Cardinality = 100 * 6 = 600)

Cost = B(S) = 1000

(Cardinality = 600 (one student per checkout))

Cost = 0 (on the fly)

(Cardinality = 600 * 7/18 = 234 (approx))

Student S
(File scan)

Book B
(Index scan)

On the fly
Cardinality = 500

Cost = 0 (on the fly)

(Cardinality = 100 * MAX(100, V(C, bid)) assuming V(C, bid) = |V(B, bid)| = T(B) = 50,000)

Cost = 1000

(Cardinality = 600 (one student per checkout))

Cost = 0 (on the fly)

(Cardinality = 600 * 7/18 = 234 (approx))
Task 4: Efficiently searching the plan space

Use dynamic-programming based Selinger’s algorithm

Heuristics for pruning plan space

- Apply predicates as early as possible
- Avoid plans with cross products
- Only left-deep join trees

Join Trees

Query: R1 \bowtie R2 \bowtie R3 \bowtie R4 \bowtie R5

left-deep join tree

bushy join tree

(physical plan space)
- Several possible structure of the trees
- Each tree can have n! permutations of relations on leaves
- Different implementation and scanning of intermediate operators for each logical plan

Selinger Algorithm

- Dynamic Programming based
- Dynamic Programming:
  - General algorithmic paradigm
  - Exploits “principle of optimality”
    - Useful reading: Chapter 16, Introduction to Algorithms, Cormen, Leiserson, Rivest
- Considers the search space of left-deep join trees
  - reduces search space (only one structure)
  - but still n! permutations
  - interacts well with join algs (esp. NLJ)
  - e.g. might not need to write tuples to disk if enough memory

Principle of Optimality

Optimal for “whole” made up from optimal for “parts”
Suppose, this is an Optimal Plan for joining R1…R5:

Then, what can you say about this sub-plan?

Exploiting Principle of Optimality

Suppose you chose the sub-optimal one

This has to be the optimal plan for joining R3, R2, R4, R1

Neither is an optimal plan for joining R1, R2, R3

This has to be the optimal plan for joining R3, R2, R4

Then, what can you say about this sub-plan?

We are using the associativity and commutativity of joins:

\[(R \bowtie S) \bowtie T = R \bowtie (S \bowtie T)\]
\[R \bowtie S = S \bowtie R\]
Simple Cost Model

Cost (R \bowtie S) = T(R) + T(S)
All other operators have 0 cost

Note: The simple cost model used for illustration only, it is not used in practice

Cost Model Example

Cost Model Example

Selinger Algorithm:

OPT ( { R1, R2, R3 } ):

Min

OPT ( ( R1, R2 ) ) + T ( ( R1, R2 ) ) + T(R3)

OPT ( ( R2, R3 ) ) + T ( ( R2, R3 ) ) + T(R1)

OPT ( ( R1, R3 ) ) + T ( ( R1, R3 ) ) + T(R2)

Note: Valid only for the simple cost model

Selinger Algorithm:

Query: R1 \bowtie R2 \bowtie R3 \bowtie R4

e.g. All possible permutations of R2, R3, R4 have been considered after OPT([R1, R3, R4]) has been computed

Progress of algorithm

Selinger Algorithm:

Query: R1 \bowtie R2 \bowtie R3 \bowtie R4

Progress of algorithm
Selinger Algorithm:

Query: R1 ⊞ R2 ⊞ R3 ⊞ R4

Progress of algorithm

Q. How to optimally compute join of {R1, R2, R3, R4}?
Ans: First optimally join {R1, R3, R4} then join with R2 as inner.

(R1, R2, R3, R4)

(R1, R2, R3)

(R1, R2, R4)

(R1, R3, R4)

(R2, R3, R4)

(R1, R2)

(R1, R3)

(R1, R4)

(R2, R3)

(R2, R4)

(R3, R4)

(R1)

(R2)

(R3)

(R4)

NOTE: There is a one-one correspondence between the permutation (R3, R1, R4, R2) and the above left deep plan

Selinger Algorithm:

Query: R1 ⊞ R2 ⊞ R3 ⊞ R4

Progress of algorithm

Q. How to optimally compute join of {R1, R3}?
Ans: Single relation – so optimally scan R3.

(R1, R2, R3, R4)

(R1, R2, R3)

(R1, R2, R4)

(R1, R3, R4)

(R2, R3, R4)

(R1, R2)

(R1, R3)

(R1, R4)

(R2, R3)

(R2, R4)

(R3, R4)

(R1)

(R2)

(R3)

(R4)

NOTE: (*VERY IMPORTANT*)
• This is "NOT" done by top-down recursive calls.
• This is done BOTTOM-UP computing the optimal cost of *all* nodes in this lattice only once (dynamic programming).

Selinger Algorithm:

NOTE: There is a one-one correspondence between the permutation (R3, R1, R4, R2) and the above left deep plan
More on Query Optimizations

- See the survey (on course website):
  "An Overview of Query Optimization in Relational Systems" by Surajit Chaudhuri

- Covers other aspects like
  - Pushing group by before joins
  - Merging views and nested queries
  - "Semi-join"-like techniques for multi-block queries
    - covered later in distributed databases
  - Statistics and optimizations
  - Starburst and Volcano/Cascade architecture, etc