Query Processing: Systems Perspective

Introduction to Databases
CompSci 316 Spring 2017
Announcements (Mon., Mar. 20)

• Homework #3
  • 3.1 and 3.2 due today
  • Remaining parts to be posted today
  • Due on Monday

• Project
  • Milestone 2 due next Monday March 27
QP so far

• Scan-based algorithms
  • Nested loop join (tuple nested, block nested)

• Sort-based algorithms
  • External merge sort
  • Sort-merge join

• Hash-based algorithms
  • Hash join

• Can be adapted to
  • Selection, projection, aggregate
Recall: Performance of SMJ

• If SMJ completes in two passes:
  • I/O’s: $3 \cdot (B(R) + B(S))$
  • Memory requirement
    • We must have enough memory to accommodate one block from each run: $M > \frac{B(R)}{M} + \frac{B(S)}{M}$
    • $M > \sqrt{B(R) + B(S)}$

• If SMJ cannot complete in two passes:
  • Repeatedly merge to reduce the number of runs as necessary before final merge and join
Recall: Performance of (two-pass) hash join

• If hash join completes in two passes:
  • I/O’s: $3 \cdot (B(R) + B(S))$
  • Memory requirement:
    • In the probing phase, we should have enough memory to fit one partition of $R$: $M - 1 > \frac{B(R)}{M-1}$
    • $M > \sqrt{B(R)} + 1$
    • We can always pick $R$ to be the smaller relation, so:
      $$M > \sqrt{\min(B(R), B(S))} + 1$$
Generalizing for larger inputs

• What if a partition is too large for memory?
  • Read it back in and partition it again!
  • See the duality in multi-pass merge sort here?
Hash join versus SMJ

Pros? Cons?

(Assuming two-pass)

• I/O’s: same

• Memory requirement: hash join is lower
  
  \[ \sqrt{\min(B(R), B(S))} + 1 < \sqrt{B(R) + B(S)} \]
  
  • Hash join wins when two relations have very different sizes

• Other factors
  
  • Hash join performance depends on the quality of the hash
    
    • Might not get evenly sized buckets
  
  • SMJ can be adapted for inequality join predicates
  
  • SMJ wins if \( R \) and/or \( S \) are already sorted
  
  • SMJ wins if the result needs to be in sorted order
What about nested-loop join?

- May be best if many tuples join
  - Example: non-equality joins that are not very selective

- Necessary for black-box predicates
  - Example: `WHERE user_defined_pred(R.A, S.B)`
Other hash-based algorithms

- Union (set), difference, intersection
  - More or less like hash join

- Duplicate elimination
  - Check for duplicates within each partition/bucket

- Grouping and aggregation
  - Apply the hash functions to the group-by columns
  - Tuples in the same group must end up in the same partition/bucket
  - Keep a running aggregate value for each group
    - May not always work
Duality of sort and hash

• Divide-and-conquer paradigm
  • Sorting: physical division, logical combination
  • Hashing: logical division, physical combination

• Handling very large inputs
  • Sorting: multi-level merge
  • Hashing: recursive partitioning

• I/O patterns
  • Sorting: sequential write, random read (merge)
  • Hashing: random write, sequential read (partition)
Index-based algorithms
Selection using index

• Equality predicate: \( \sigma_{A=v}(R) \)
  • Use an ISAM, B\(^+\)-tree, or hash index on \( R(A) \)

• Range predicate: \( \sigma_{A>\nu}(R) \)
  • Use an ordered index (e.g., ISAM or B\(^+\)-tree) on \( R(A) \)
  • Hash index is not applicable

• Indexes other than those on \( R(A) \) may be useful
  • Example: B\(^+\)-tree index on \( R(A, B) \)
  • How about B\(^+\)-tree index on \( R(B, A) \)?
Index versus table scan

Situations where index clearly wins:

• **Index-only queries** which do not require retrieving actual tuples
  • Example: $\pi_A(\sigma_{A>v}(R))$

• Primary index clustered according to search key
  • One lookup leads to all result tuples in their entirety
Index versus table scan (cont’d)

BUT(!):

• Consider $\sigma_{A>v}(R)$ and a secondary, non-clustered index on $R(A)$
  • Need to follow pointers to get the actual result tuples
  • Say that 20% of $R$ satisfies $A > v$
    • Could happen even for equality predicates
  • I/O’s for index-based selection: lookup + 20% $|R|$  
  • I/O’s for scan-based selection: $B(R)$
• Table scan wins if a block contains more than 5 tuples!
Index nested-loop join

\[ R \bowtie_{R.A=S.B} S \]

Suppose there is an index on \( S(B) \): \( S \) is outer or inner?

- Idea: use a value of \( R. A \) to probe the index on \( S(B) \)
- For each block of \( R \), and for each \( r \) in the block:
  - Use the index on \( S(B) \) to retrieve \( s \) with \( s.B = r.A \)
  - Output \( rs \)

- I/O’s: \( B(R) + |R| \cdot (\text{index lookup}) \)
  - Typically, the cost of an index lookup is 2-4 I/O’s
  - Beats other join methods if \( |R| \) is not too big
  - Better pick \( R \) to be the smaller relation

- Memory requirement: 3
Zig-zag join using ordered indexes

\[ R \bowtie_{R.A=S.B} S \]

- Idea: use the ordering provided by the indexes on \( R(A) \) and \( S(B) \) to eliminate the sorting step of sort-merge join
- Use the larger key to probe the other index
  - Possibly skipping many keys that don’t match
Summary of techniques

• Scan
  • Selection, duplicate-preserving projection, nested-loop join

• Sort
  • External merge sort, sort-merge join, union (set), difference, intersection, duplicate elimination, grouping and aggregation

• Hash
  • Hash join, union (set), difference, intersection, duplicate elimination, grouping and aggregation

• Index
  • Selection, index nested-loop join
Query Processing: Systems aspects
A query’s trip through the DBMS

SQL query

Parse tree

Logical plan

Physical plan

Result

SELECT name, uid
FROM Member, Group
WHERE Member.gid = Group.gid;

π name, uid
σ Member.gid = Group.gid

Member Group
Parsing and validation

• **Parser**: SQL → parse tree
  • Detect and reject **syntax** errors

• **Validator**: parse tree → logical plan
  • Detect and reject **semantic** errors
    • Nonexistent tables/views/columns?
    • Insufficient access privileges?
    • Type mismatches?
      • Examples: AVG(name), name + pop, User UNION Member

• Also
  • Expand *
  • Expand view definitions

• Information required for semantic checking is found in **system catalog** (which contains all schema information)
Logical plan

- Nodes are **logical** operators (often relational algebra operators)
- There are many equivalent logical plans

An equivalent plan:

```
π_{Group.name} ∘ (π_{Group.name} ⋈ (π_{User.name="Bart" \land User.uid = Member.uid \land Member.gid = Group.gid} ⋈ User \times Member))
```
Physical (execution) plan

• A complex query may involve multiple tables and various query processing algorithms
  • E.g., table scan, index nested-loop join, sort-merge join, hash-based duplicate elimination...

• A physical plan for a query tells the DBMS query processor how to execute the query
  • A tree of physical plan operators
  • Each operator implements a query processing algorithm
  • Each operator accepts a number of input tables/streams and produces a single output table/stream
Examples of physical plans

SELECT Group.name
FROM User, Member, Group
WHERE User.name = 'Bart'
AND User.uid = Member.uid AND Member.gid = Group.gid;

- Many physical plans for a single query
  - Equivalent results, but different costs and assumptions!
  - DBMS query optimizer picks the “best” possible physical plan
Physical plan execution

• How are intermediate results passed from child operators to parent operators?
  • Temporary files
    • Compute the tree bottom-up
    • Children write intermediate results to temporary files
    • Parents read temporary files
  • Iterators
    • Do not materialize intermediate results
    • Children pipeline their results to parents
Iterator interface

• Every physical operator maintains its own execution state and implements the following methods:
  • open(): Initialize state and get ready for processing
  • getNext(): Return the next tuple in the result (or a null pointer if there are no more tuples); adjust state to allow subsequent tuples to be obtained
  • close(): Clean up
An iterator for table scan

• State: a block of memory for buffering input $R$; a pointer to a tuple within the block

• **open()**: allocate a block of memory

• **getNext()**
  • If no block of $R$ has been read yet, read the first block from the disk and return the first tuple in the block
    • Or null if $R$ is empty
  • If there is no more tuple left in the current block, read the next block of $R$ from the disk and return the first tuple in the block
    • Or null if there are no more blocks in $R$
  • Otherwise, return the next tuple in the memory block

• **close()**: deallocate the block of memory
An iterator for nested-loop join

R: An iterator for the left subtree
S: An iterator for the right subtree

• open()
  R.open()
  S.open()
  r = R.getNext()

• getNext()
  • To output just the next (r, s) tuple from R Join S
    while True:
      s = S.getNext()
      if s is null: # no more tuple from S to join with the current R-tuple
        S.close()
      r = R.getNext() # move on to next r
      if r is null: # no more tuple from R
        return null
      # otherwise we are at current r
      S.open() # reopen S
      s = S.getNext()
      if s is null: # S is empty!
        return null
      if joins(r, s):
        return concat(r, s)

• close()
  R.close()
  S.close()
An iterator for 2-pass merge sort

• open()
  • Allocate a number of memory blocks for sorting
  • Call open() on child iterator

• getNext()
  • If called for the first time
    • Call getNext() on child to fill all blocks, sort the tuples, and output a run
    • Repeat until getNext() on child returns null
    • Read one block from each run into memory, and initialize pointers to point to the beginning tuple of each block
  • Return the smallest tuple and advance the corresponding pointer; if a block is exhausted bring in the next block in the same run

• close()
  • Call close() on child
  • Deallocate sorting memory and delete temporary runs
Blocking vs. non-blocking iterators

• A **blocking** iterator must call `getNext()` exhaustively (or nearly exhaustively) on its children before returning its first output tuple
  • Examples: sort, aggregation

• A **non-blocking** iterator expects to make only a few `getNext()` calls on its children before returning its first (or next) output tuple
  • Examples: dup-preserving projection, filter, merge join with sorted inputs
Execution of an iterator tree

• Call `root.open()`
• Call `root.getNext()` repeatedly until it returns null
• Call `root.close()`

✓ Requests go down the tree
✓ Intermediate result tuples go up the tree
✓ No intermediate files are needed
  • But maybe useful if an iterator is opened many times
    • Example: complex inner iterator tree in a nested-loop join; “cache” its result in an intermediate file