Lecture 13
Advanced Query Processing

New Requirements

- Top-N/Bottom-N queries
- Interactive queries
  - Decision making queries
    - Tolerant of errors – approximate answers acceptable
  - Control over what one sees
- Skyline queries, …

Fast initial response time!

Top-N/Bottom-N Queries

- STOP AFTER N clause in SQL

Find the 5 hotels closest to the Changi airport

```
SELECT h.name, h.addr, h.phone
FROM Hotels h, Airports a
WHERE a.name = 'Changi'
ORDER BY distance(h.location, a.location)
STOP AFTER 5;
```

Find the top 10% of software products in terms of gross sales revenues

```
SELECT p.name, s.gross
FROM Products p, Sales s
WHERE p.type='Software'
AND p.prod_num=s.prod_num
ORDER BY s. gross DESC
STOP AFTER (SELECT count(*)/10
            FROM Products p
            WHERE p.type='Software');
```


2 Different Strategies

- 'Middleware' approach
  - Traditional vs Rewriting
  - Can reuse existing optimizer
  - Miss opportunities for performance improvement
- Introduce new operator: STOP
  - Need to change the optimizer
  - Likely to produce better plans

Rewriting Approach

- Rewrite a query into a set of subqueries
- Evaluate each subqueries 'on-demand'

```
SELECT name, salary
FROM emp
ORDER BY salary DESC
STOP AFTER N
```

```
SELECT name, salary
FROM emp
WHERE salary > 50K
ORDER BY salary DESC
```

```
SELECT name, salary
FROM emp
WHERE salary <= 50K
ORDER BY salary DESC
```


Rewriting Approach

```
Rewrite query;
#ans = 0;
answer = ∅
i = 1;
While #ans < N AND moreSubqueries do {
        ans =  subquery(i);  // suppose there are k answers;
        answer = answer ∪ ans;
        #ans = #ans + k;
        if #ans ≥ N
            return top N answer;
        else
            i++;
}
If #ans < N return answer
// as optimization, answers can be returned immediately to reduce initial response time
```

STOP operator

- SCAN-STOP
  - Pipelined operator that requests and then passes each of the first N tuples of its input stream on to its consumer
- SORT-STOP
  - Sort the input, then return the first N tuples
  - If N is small, priority heap can be used; otherwise external sort is used
- Issue: Placement of STOP operator in a query plan
  - Pushing deep down cuts the cost of opeators higher up in plan
  - May eliminate too many tuples of intermediate results

Example

Emp(empId, name, salary, *works_in*, *teaNo*)
Dept(dno, name, budget, function, description)
TEA(accNo, expenses, comments)

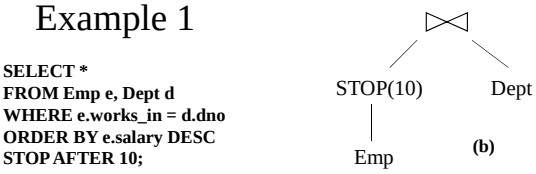
works_in is foreign key (same domain as dno)
teaNo is foreign key (for accNo)
Not every employee has a travel account. Suppose 50%.

Conservative STOP Placement

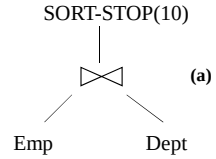
- Never place a STOP operator at a point in a plan where its presence can cause tuples to be discarded that may be required to compose the requested N tuples of the query result

Example 1

SELECT *
FROM Emp e, Dept d
WHERE e.works_in = d.dno
ORDER BY e.salary DESC
STOP AFTER 10;



(b)

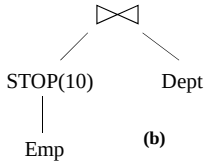


(a)

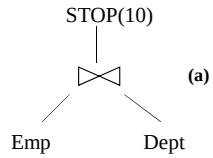
Plan (b) is better?

Example 1

SELECT *
FROM Emp e, Dept d
WHERE e.works_in = d.dno
ORDER BY e.salary DESC
STOP AFTER 10;



(b)

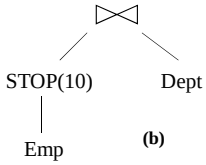


(a)

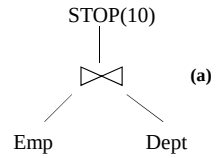
Plan (b) is correct if Emp records are retrieved in salary order! and better because of the foreign key constraint: Every employee must belong to a department. The join condition is referred to as a "non-reductive" predicate

Example 2

SELECT *
FROM Emp e, Dept d
WHERE e.works_in = d.dno
AND d.function = 'Research'
ORDER BY e.salary DESC
STOP AFTER 10;



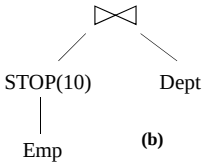
(b)



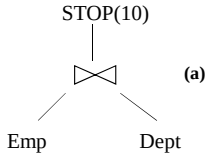
(a)

Example 2

SELECT *
FROM Emp e, Dept d
WHERE e.works_in = d.dno
AND d.function = 'Research'
ORDER BY e.salary DESC
STOP AFTER 10;



(b)

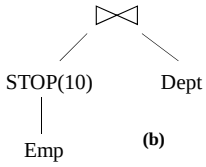


(a)

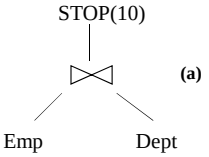
Plan (b) is incorrect.

Example 2

SELECT *
FROM Emp e, Dept d
WHERE e.works_in = d.dno
AND d.function = 'Research'
ORDER BY e.salary DESC
STOP AFTER 10;



(b)



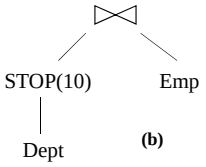
(a)

Plan (b) is incorrect.

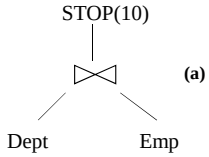
Some of the top 10 employees may not belong to the dept with function = "Research"
d.function is a reductive predicate.

Example 3

SELECT *
FROM Emp e, Dept d
WHERE e.works_in = d.dno
ORDER BY d.budget DESC
STOP AFTER 10;



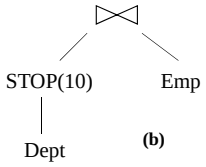
(b)



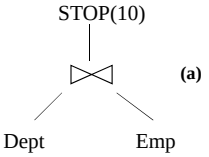
(a)

Example 3

SELECT *
FROM Emp e, Dept d
WHERE e.works_in = d.dno
ORDER BY d.budget DESC
STOP AFTER 10;



(b)



(a)

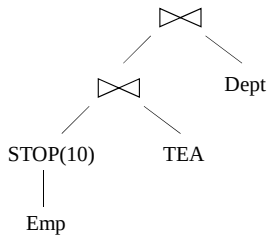
Plan (b) is incorrect unless every dept must have at least one employee (even though the join predicate is non-reductive)

Aggressive STOP Placement

- Insert STOP operators in query plans whenever they can provide a beneficial cardinality reduction
  - Need to estimate intermediate results accurately
  - Need to compute the stopping cardinality for the STOP operators (may be different from N)
  - Need a RESTART operator and placed it well

Example

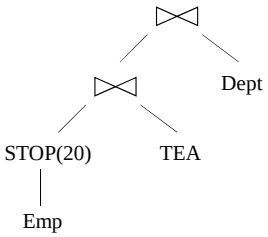
SELECT e.name, e.salary,
d.name, t.expenses
FROM Emp e, Dept d, TEA t
WHERE e.works_in = d.dno
AND e.teaNo = t.accNo
ORDER BY e.salary DESC
STOP AFTER 10



This is incorrect!

Example

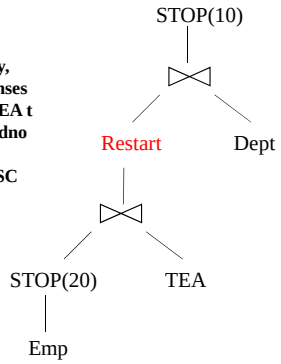
SELECT e.name, e.salary,
d.name, t.expenses
FROM Emp e, Dept d, TEA t
WHERE e.works_in = d.dno
AND e.teaNo = t.accNo
ORDER BY e.salary DESC
STOP AFTER 10



This is incorrect!

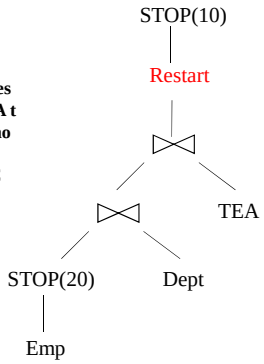
Example

SELECT e.name, e.salary,
d.name, t.expenses
FROM Emp e, Dept d, TEA t
WHERE e.works_in = d.dno
AND e.teaNo = t.accNo
ORDER BY e.salary DESC
STOP AFTER 10



Example

SELECT e.name, e.salary,
d.name, t.expenses
FROM Emp e, Dept d, TEA t
WHERE e.works_in = d.dno
AND e.teaNo = t.accNo
ORDER BY e.salary DESC
STOP AFTER 10

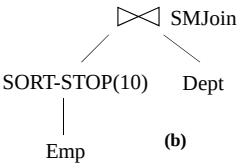


Query Optimization

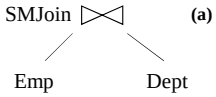
- Simply treat STOP operator as one possible access path
- Need to be careful when pruning plans

Example

SELECT *
FROM Emp e, Dept d
WHERE e.works_in = d.dno
ORDER BY e.salary DESC
STOP AFTER 10;



(b)

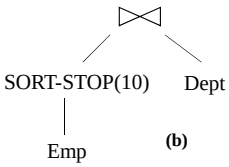


(a)

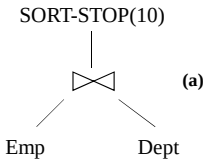
Suppose plan (a) is cheaper. But, cannot prune (b) since (b) may be the cheaper plan eventually

Example

SELECT *
FROM Emp e, Dept d
WHERE e.works_in = d.dno
ORDER BY e.salary DESC
STOP AFTER 10;



(b)



(a)

Plan (a) without SORT-STOP (10) is cheaper but is more expensive with SORT-STOP(10)

Example

```
SELECT *
FROM Emp
WHERE age > 50
ORDER BY salary DESC
STOP AFTER 10;
```

SCAN-STOP(10)
|
SORT(salary)
|
TBL-SCAN(Emp.age > 50)

SORT-STOP(10)
|
TBL-SCAN(Emp.age > 50)

SCAN-STOP(10)
|
RID-SCAN(age>50)
|
IDX-SCAN(Emp.salary)

Exploiting Range Partitioning

- New operators:
  - Part-mat: takes a partitioning vector, scan the input and write out the partitions to disk
  - Part-scan: scan the partitions one at a time
  - Part-reread: takes a set of (range predicates) and materializes a partition's tuples by (re)reading its input stream from the beginning
  - Part-hybrid: materializes a specified number of its highest (or lowest) ranked partitions and computes the rest only on demand


```
SELECT *
FROM Emp
WHERE age > 50
ORDER BY salary DESC
STOP AFTER 10;
```

Scan-stop(10)
|
Restart(N)
|
Sort(salary)
|
Part-scan
|
Part-mat
|
Tblscan(Emp.age>50)

Scan-stop(10)
|
Restart(N)
|
Sort(salary)
|
Part-reread
|
Tblscan(Emp.age>50)

Scan-stop(10)
|
Restart(N)
|
Sort(salary)
|
Part-scan
|
Part-hybrid
|
Tblscan(Emp.age>50)

Choosing a partitioning vector

- Histogram
- Sampling

Example: Join Query

```
SELECT *
FROM Emp e, Dept d
WHERE age > 50 AND d.budget > 1000 AND e.works_in = d.dno
ORDER BY salary DESC
STOP AFTER 10;
```

Scan-sort(10)
|
INLJ
|
Sort(salary) — Ridscan(b>1000)
|
Tblscan(Emp.age > 50) — idxscan(Dept.dno)

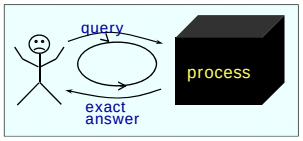
Scan-sort(10)
|
Sort(salary)
|
Restart(10)
|
INLJ
|
Part-scan — Ridscan(b>1000)
|
Part-hybrid — idxscan(Dept.dno)
|
Tblscan(Emp.age > 50)

Commercial Products

- Informix – FIRST N
- Microsoft SQL Server – FAST N
- IBM's DB2 UDB system
  - OPTIMIZE FOR n ROWS
  - FETCH FIRST n ROWS ONLY
- Oracle Rdb – LIMIT TO N ROWS
- Redbrick
  - SET ROWCOUNT N
  - WHEN RANK(col) < n

Interactive Query Processing

- Problems with traditional solutions (for aggregate queries)
  - mismatch between system functionality and mode of HCI



- black boxes
  - do batch processing
  - frustrating delays in iterative process

Interactive processing

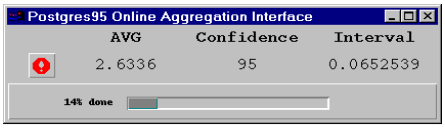
- HCI requirements
  - users must get continual feedback on results of processing
  - allow users to control processing based on prior feedback
- Performance goals
  - *not* to minimize time to give complete results
  - give continually improving partial results
  - adapt to dynamically specified performance goals

Online Aggregation

- Three-fold requirement
  - answers are or derived from summary data
  - imprecise answers tolerated
  - answers must be obtained quickly
- Traditional aggregation takes a long time to return a very small final result from a large amount of data
- Online aggregation allows users to observe the progress of their queries and control execution on the fly

New interface for aggregation

- Observe the progress of their queries
- aggregates have running output and confidence interval
- Control execution on-the-fly

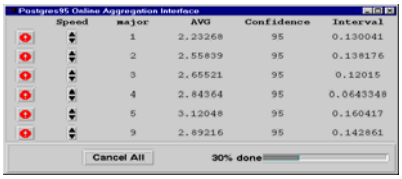


Statistical estimation

- Users do not need to set *a priori* specification of stopping condition
- The interface is easier for users with no statistical background
- It requires more powerful statistical estimation techniques (Hoeffding's inequality versus Chebyshev's inequality)

Usability goals

- Continuous observation
- Control of time/precision
- Control of fairness/partiality



Performance goals

- Minimum time to accuracy: produce a *useful* estimate of the final answer ASAP
- Minimum time to completion: *secondary* goal, assume user will terminate processing long before the final answer is produced
- Pacing: guarantee a smooth and continuous improving display

Random access to data

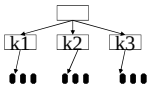
We need to retrieve data in random order to produce meaningful statistical estimation. Three ways to get records in random order:

- Heap scans
  - Assumes that records are not stored in any specific order otherwise …
- Index scans
  - Indexed attributes are different from (and not correlated to) aggregated attributes
- Sampling from indices (less efficient)

Non-blocking GROUP BY and DISTINCT

- Sorting is a blocking algorithm and only one group is computed at a time after sorting
- Hashing is non-blocking, but hash table need to fit in memory to have good performance
- Hybrid Cache (an extension of hybrid hashing) might be good

Index striding

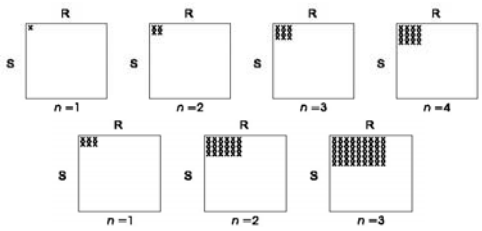


- Hash-based grouping can be unfair
- Solution: probe the index to find all the groups and then process tuples from each group in a “round robin” fashion
- Can control speed by weighting the schedule
- Fair for groups with different cardinality

Non-blocking join algorithms (1)

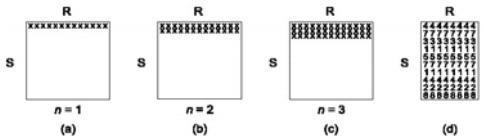
- Sort-merge join is not acceptable for online aggregation because sorting is blocking
- Hash join blocks for the time required to partition the relations
- Pipeline hash join techniques may be appropriate for online aggregations when both relations are small
- Merge join (without sort) and hash join provide output with orders – not good for statistic estimation
- The “safest” traditional join algorithm is nested loop, particularly if there is an index on the inner relation

Overview of ripple join (1)



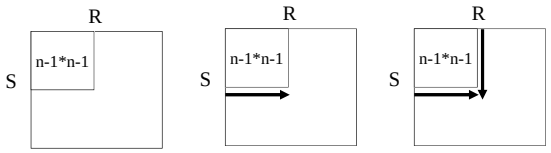
Overview of ripple join (2)

- online nested-loops join is a special case of ripple join



Ripple join algorithms (1)

- It can be viewed as a generalization of nested-loops join in which the traditional roles of "inner" and "outer" relation are continually interchanged during processing



Skyline queries

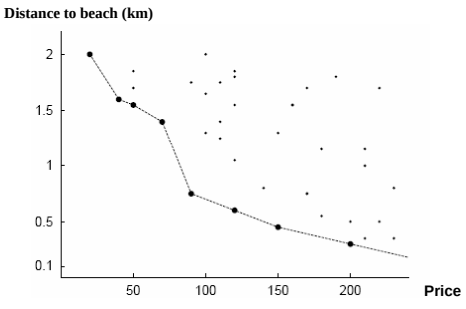
- Decision making queries
  - Based on multiple criteria
  - No single optimal answer
  - *Satisficing* answer
  - Determined by user preferences
    - E.g., budget hotel with reasonable rating and is close to the city

```
SELECT name, rating
FROM hotel
WHERE rating = '3*'
AND cost BETWEEN 100 AND 150
AND  distanceToCity < 1km
```



```
SELECT name, rating, cost
FROM hotel
WHERE rating > '2*'
AND cost BETWEEN 100 AND 150
AND  distanceToCity < 1km
SKYLINE rating MAX, cost MIN, distanceToCity MIN, roomType DIFF
```

A tuple t1 dominates another tuple t2 if
rating1 >= rating2 AND
cost1 <= cost2 AND
distanceToCity1 <= distanceToCity2 AND
roomType1 = roomType2



Block-Nested-Loops Algorithm

- Scan some records of R several times
- Keep a window of *incomparable* tuples in main memory
- When a tuple p is read, p is compared to all tuples of the window:
  - p is dominated by a tuple within the window; throw p away
  - p dominates one or more tuples in the window; remove those tuples; insert p into window
  - P is incomparable with all tuples in window; insert into window if there is room; otherwise, p is written to a temporary file on disk.
- At the end of the iteration, output tuples of window which have been compared to all tuples that have been written to temporary file
- Repeat the process on the temporary file and the remaining content of memory

tuples	x	y
p_1	10	9
p_2	6	8
p_3	1	7
p_4	3	6
p_5	7	6
p_6	4	5
p_7	8	4
p_8	2	3
p_9	5	2
p_{10}	9	1

Window

p_8	p_9	p_{10}

Temporary file on disk

Skyline

p_3, p_8, p_9, p_{10}

Window when temporary was created

p_3	p_4	p_6

Summary

- Many new applications call for novel query processing methods
- In particular, fast initial response time is desirable.
- New operators may need to be introduced for optimal performance