CMSC724: Query Optimization; AQP

Amol Deshpande

University of Maryland, College Park

February 19, 2016

Outline

1. Query Optimization
2. Adaptive Query Processing
   - Eddies

Query Optimization

- Goal: Given a SQL query, find the best physical operator tree to execute the query
- Problems:
  - Huge plan space
    - More importantly, cheapest plan orders of magnitude cheaper than worst plans
    - Typical compromise: avoid really bad plans
  - Complex operators/semantics etc
    - (R outerjoin S) join T $\neq$ R outerjoin (S join T)

Query Compilation: Steps

- Parsing: analyze sql query, detect syntax errors, create internal query representation
- Semantic checking:
  - Validate SQL statement, view analysis, incorporate constraints/triggers etc
- Query rewrite: Modify query to improve performance
- Optimization
- Code generation

Query Rewrite

- Goal: more latitude for optimizer; more efficient processing
- Typically done using a rule-based approach
  - IBM Query Graph Model paper has details on how it is done
- Examples:
  - Original: select distinct custkey, name from TPCD.CUSTOMER
  - Rewritten: select custkey, name from TPCD.CUSTOMER
  - Why? custkey is a key

Query Rewrite

- Original:
  - SELECT ps.* FROM partsupp ps
  - WHERE ps.ps_partkey IN (SELECT p_partkey FROM tpcd.parts WHERE p_name LIKE 'forest%');
- Rewritten:
  - SELECT ps.* FROM parts, partsupp ps
  - WHERE ps.ps_partkey = p_partkey AND p_name LIKE 'forest%';
- Predicate translation:
  - WHERE NOT(COL1 = 10 OR COL2 > 3) → WHERE COL1 <> 10 AND COL2 <= 3

Query Rewrite

- Must be careful with distincts and "nulls"
- Original:
  - SELECT Dept.Name FROM Dept
  - WHERE Dept.num-of-machines >=
  - (SELECT Count(EMP.*) FROM Emp WHERE Dept.name = Emp.Dept_name)
- Rewritten:
  - SELECT Dept.Name FROM Dept Join Emp
  - GROUP BY Dept.name
  - HAVING Dept.num-of-machines < Count(EMP.*)
- Must use a left-outer-join
  - Otherwise a dept with no employees may cause problems

Query Optimization

- Heuristical approaches
  - Perform selection early (reduce number of tuples)
  - Perform projection early (reduce number of attributes)
  - Perform most restrictive selection and join operations before other similar operations.
  - Don't do Cartesian products
- INGRES:
  - Always use NL-Join (indexed inner when possible)
  - Order relations from smallest to biggest

Query Optimization

- A systematic approach
  - Define a plan space (what solutions to consider)
  - A cost estimation technique
  - An enumeration algorithm to search through the plan space

System-R Query Optimizer

- Define a plan space
  - Left-deep plans, no Cartesian products
  - Nested-loops and sort-merge joins, sequential scans or index scans
- A cost estimation technique
  - Use statistics (e.g. size of index, max, min etc) or magic numbers
  - Formulas for computing the costs
- An enumeration algorithm to search through the plan space
  - Dynamic programming

Aside...

- Cost metric
  - Typically a combination of CPU and I/O costs
    - The "w" parameter set to balance the two
  - Response time (useful in distributed and parallel scenarios)
    - Behaves different from the above *total work* metric
  - Time to first tuple (useful in interactive applications)

Aside...

- Cost metric
  - Typically a combination of CPU and I/O costs
    - The "w" parameter set to balance the two
  - Response time (useful in distributed and parallel scenarios)
    - Behaves different from the above *total work* metric
  - Time to first tuple (useful in interactive applications)
- How about a simpler metric ?
  - *Count the total number of intermediate tuples that would be generated*
  - Independent of access methods
  - Ok in some scenarios, but reasoning about indexes is key in optimization

System-R Query Optimizer

- Dynamic programming
- Uses "principle of optimality"
    - Bottom-up algorithm
    - Compute the optimal plan(s) for each k-way join, k = 1, ..., n
        - Only $O(2^n)$ instead of $O(n!)$
    - Computes plans for different "interesting orders"
        - Extended to "physical properties" later
    - Another way to look at it:
        - Plans are not comparable if they produce results in different orders
        - An instance of multi-criteria optimization

Since then...

- Search space
  - “Bushy” plans (especially useful for parallelization)
  - Cartesian products (star queries in data warehouses)
  - Algebraic transformations
    - Can “group by” and “join” commute ?
  - More physical operators
    - Hash joins, semi-joins (crucial for distributed systems)
  - Sub-query flattening, merging views
    - “Query rewrite”
  - Parallel/distributed scenarios...

Since then...

- Statistics and cost estimation
  - Optimization only as good as cost estimates
    - Optimizers not overly sensitive ($\pm$ 50% probably okay)
    - Better to overestimate selectivities
  - Histograms, sampling commonly used
  - Correlations ?
    - Ex: where model = "accord" and make = "honda"
    - Say both have selectivities 0.0001
    - Then combined selectivity is also 0.0001, not 0.0000001
  - Learning from previous executions
    - Learning optimizer (LEO@IBM), SITS (MS SQL Server)
  - Cost metric: Response time in parallel databases, buffer utilization...

Since then...

- Enumeration techniques
  - Bottom-up more common
    - Easier to implement, low memory footprint
  - Top-down (Volcano/Cascades/SQL Server)
    - More extensible, typically larger memory footprint etc...
  - Neither work for large number of tables
    - Randomized, genetic etc...
    - More common to use heuristics instead
  - "Parametric query optimization"

Other issues

- Non-centralized environments
  - Distributed/parallel, P2P
  - Data streams, web services
  - Sensor networks??
- User-defined functions
- Materialized views

Outline

1. Query Optimization
2. Adaptive Query Processing
   - Eddies

Adaptive Query Processing

- Why? Traditional optimization is breaking
- In traditional settings:
  - Queries over many tables
  - Unreliability of traditional cost estimation
  - Success, maturity make problems more apparent, critical
- In new environments:
  - e.g. data integration, web services, streams, P2P...
  - Unknown dynamic characteristics for data and runtime
  - Increasingly aggressive sharing of resources and computation
  - Interactivity in query processing
- Note two distinct themes lead to the same conclusion:
  - *Unknowns:* even static properties often unknown in new environments and often unknowable a priori
  - *Dynamics:* environment changes can be very high
    - **Motivates intra-query adaptivity**

Some related topics

- Autonomic/self-tuning optimization
  - Chen and Roussoupolous: Adaptive selectivity estimation [SIGMOD 1994]
  - LEO (@IBM), SITS (@MSR): Learning from previous executions
- Robust/least-expected cost optimization
- Parametric optimization
  - Choose a collection of plans, each optimal for a different setting of parameters
  - Select one at the beginning of execution
- Competitive optimization
  - Start off multiple plans... kill all but one after a while
- Adaptive operators
- More details in our survey: “Adaptive Query Processing”; FnT 2007

AQP: Overview/Summary

- Low-overhead, evolutionary approaches
  - Typically apply to non-pipelined execution
  - **Late binding:** Don't instatntiate the entire plan at start
  - **Mid-query reoptimization:** At "materialization" points, review the remaining plan and possibly re-optimize
    - More recently, much work/implementation along these lines at IBM

AQP: Overview/Summary

- Low-overhead, evolutionary approaches
  - Typically apply to non-pipelined execution
  - **Late binding:** Don't instatntiate the entire plan at start
  - **Mid-query reoptimization:** At "materialization" points, review the remaining plan and possibly re-optimize
    - More recently, much work/implementation along these lines at IBM
- Pipelined execution
  - No materialization points, so the above doesn't apply
  - The operators may contain complex states, raising correctness issues
  - **Eddies**
    - Always guarantee correct execution, but allows reordering during execution
  - Much other work in recent years (see the survey)

Outline

1 Query Optimization

2 Adaptive Query Processing

- Eddies

Eddy/Tuple Router

- An operator that controls the tuple in-flow and out-flow for a collection of operators
  - Allows better control over scheduling and output
    - For interactive applications, for user feedback etc...
  - Enables adaptivity
    - Different tuples can be processed in different orders
  - Better suited for “reacting” to tuples

Eddy/Tuple Router

- An operator that controls the tuple in-flow and out-flow for a collection of operators
  - Allows better control over scheduling and output
    - For interactive applications, for user feedback etc...
  - Enables adaptivity
    - Different tuples can be processed in different orders
  - Better suited for "reacting" to tuples
- Can be implemented as an iterator
  - See details in
    "An initial study of overheads of routing", SIGMOD Record 2004

Eddy/Tuple Router

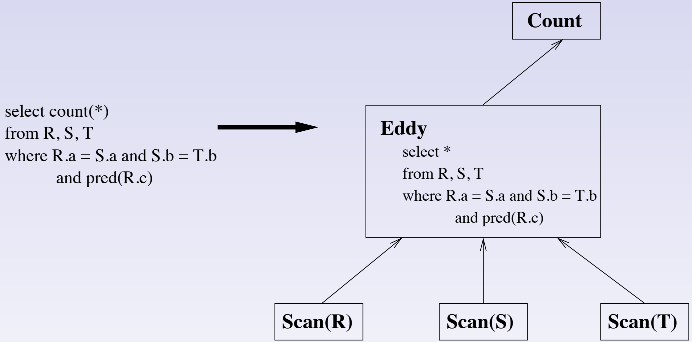


Figure 2: Using traditional operators along with an eddy

Eddy/Tuple Router

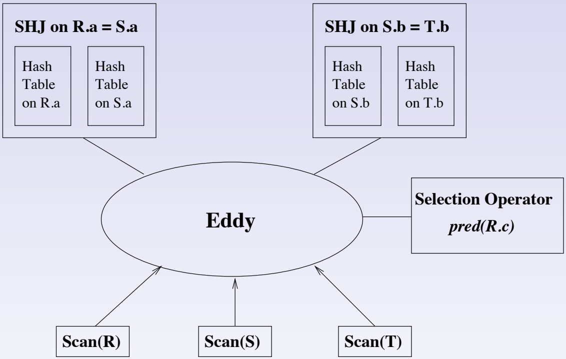
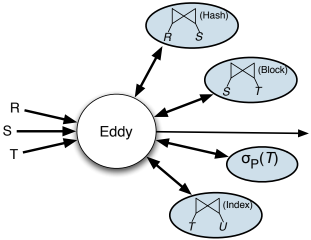


Figure 3: Eddy instantiated for the example query

Eddy/Tuple Router

Example Query

SELECT *
FROM R, S, T, U
WHERE R.a = S. a
AND S.b = T.b
AND T.c = U. c
AND $\sigma_P(T)$



Tuple Signature

Base Tables	Routed Through	Valid Destinations
{R}	{}	$(R \bowtie S, 1.0)$
{S, T}	$\{S \bowtie T, \sigma_P(T)\}$	$(R \bowtie S, 0.3), (T \bowtie U, 0.7)$
..	..	..

Routing Table

Fig. 3.1 Example of an eddy instantiated for a 4-way join query (taken from Avnur and Hellerstein [AH00]). A routing table can be used to record the valid routing destinations, and possibly current probabilities for choosing each destination, for different tuple signatures.

Eddy/Tuple Router: Mechanism vs Policy

- Tricky to reason about: Encapsulates too much logic
- Break into two pieces (discussion from AQP Survey)

Eddy/Tuple Router: Mechanism vs Policy

- Tricky to reason about: Encapsulates too much logic
- Break into two pieces (discussion from AQP Survey)
- Mechanism: Enables the adaptivity
  - By allowing eddy choice at any point
  - As long as the eddy obeys some rules, the execution will be correct
    - Not always easy... arbitrary routings can be nonsensical
  - For any tuple, the mechanism tells the eddy the valid set of operators to route to
  - Mechanism can be implemented efficiently (see SIGMOD Record paper)

Eddy/Tuple Router: Mechanism vs Policy

- Tricky to reason about: Encapsulates too much logic
- Break into two pieces (discussion from AQP Survey)
- Mechanism: Enables the adaptivity
    - By allowing eddy choice at any point
    - As long as the eddy obeys some rules, the execution will be correct
        - Not always easy... arbitrary routings can be nonsensical
    - For any tuple, the mechanism tells the eddy the valid set of operators to route to
    - Mechanism can be implemented efficiently (see SIGMOD Record paper)
- Policy: Exploit the adaptivity
    - For each tuple, choose the operator to route too
    - This can be as complex as you want

Eddy/Tuple Router: Steps

- Instatiate operators based on the query
  - Fully pipelined operators (SHJ, MJoins) preferred, otherwise not as much feedback
  - Sort-merge join will not provide any output tuples till all input tuples are consumed

Eddy/Tuple Router: Steps

- Instatiate operators based on the query
  - Fully pipelined operators (SHJ, MJoins) preferred, otherwise not as much feedback
  - Sort-merge join will not provide any output tuples till all input tuples are consumed
- At each instance:
  - Choose next tuple to process
    - Either a new source tuple or an intermediate tuple produced by an operator
  - Decide which operator to route to (using the policy)
  - Add result tuples from the operator (if any) to a queue
    - If a result tuple is fully processed, send to output

Eddy/Tuple Router: Steps

- Instatiate operators based on the query
  - Fully pipelined operators (SHJ, MJoins) preferred, otherwise not as much feedback
  - Sort-merge join will not provide any output tuples till all input tuples are consumed
- At each instance:
  - Choose next tuple to process
    - Either a new source tuple or an intermediate tuple produced by an operator
  - Decide which operator to route to (using the policy)
  - Add result tuples from the operator (if any) to a queue
    - If a result tuple is fully processed, send to output
- We will revisit *policy issues* when discussing AQP