1 Query Processing

Query Processing

- Assume single-user, single-threaded
  - Concurrency managed by lower layers

- Steps:
  - Parsing: attribute references, syntax etc...
    * Catalog stored as “denormalized” tables
  - Rewriting:
    * Views, constants, logical rewrites (transitive predicates, true/false predicates), semantic (using constraints), subquery flattening
  - Optimizer – Later
  - Executor: Next

1.1 Traditional Operators

Executor: Operators

- Selections: Usually pushed down if possible
  - SARGABLE predicates
  - Advantages in not doing so (for expensive predicates)

- Project
  - If no duplicate elimination, then trivial
  - If duplicate elimination, can use sorting (preferred) or hashing
  - Note that: this suggests that sort-merge joins may be preferable as the child operator
  - Decision made by the optimizer (“interesting orders”)
Executor: Operators

- Aggregates and Group by (usually together)
  - Distributive (MAX, MIN, COUNT, SUM): Constant state
  - Algebraic (AVERAGE): Can use COUNT and SUM
  - Holistic (MEDIAN, QUANTILE): May need to gather the whole input

- PostgreSQL allows defining user-defined aggregates:
  - User-defined Aggregates in PostgreSQL
    - Basically need to define an “accumulator” function..
      * Take in one tuple at a time (get_next())
      * Eventually produce the aggregate (one by one)

Executor: Operators

- Joins
  - Equijoin (natural join): Nested loops, Index nested loops, hash join (classic, GRACE, hybrid), merge join
  - Non-equijoins?
    * Sort-merge joins in some cases (e.g. ABS(R.a - S.b) < 5)
    * Index nested loops in some cases (e.g. index on R.a, may use for R.a < S.a)
    * Nested loops otherwise (always works)
  - Join variants: Outerjoins, semijoins, Anti-joins etc...
    * Usually same algorithms as above, with minor modifications (may even be an "if" in the code)

Executor: Operators

- Set operators: Intersection, Union, Difference etc..
  - Variants of join operators (different logic based on duplicate eliminate or not)
  - Note that: SQL is bag algebra

- Others?
  - Top-K, CUBE etc...
  - List goes on
Executor: Operators

- Much commonality between operators
- Usually a smaller set of Physical Operators
  - e.g. TEMP is a materialization operator: Reads all tuples from the child operator and stores them somewhere
    * by repeatedly issuing get_next()
  - Similarly, HASH, SORT etc..
  - See [An overview of DB2 Optimizer](#) for more details

Executor: Operators

- Blocking operators vs Pipelining operators
  - Important: dictates memory use, time to first tuple
    * TEMP, SORT are blocking
  - All operators in a pipeline must be in memory, so higher memory requirements
  - Some operators are naturally blocking
    * DISTINCT (duplicate elimination)
    * AGGREGATES (can’t really produce a COUNT without seeing all input)
  - Increasingly prefer pipelining operators (larger memories)

Executor

- “get_next()” iterator model
  - Narrow interface between iterators
  - Can be implemented independently
  - Assumes non-blocking-I/O

- Memory
  - Usually managed carefully: swapping not good
  - Sorting can exploit the memory naturally to the fullest
  - Hashing needs careful partitioning

- Some low-level details
  - Tuple-descriptors
  - Very carefully allocated memory slots
  - “avoid in-memory copies”
  - Pin and unpin
Query Processing

- SQL Update/Delete
  - “Halloween” problem
- Access Methods
  - B+-Tree and heap files
    - Multi-dimensional indexes not common
  - init(SARG)
    - “avoid too many back-and-forth function calls”
- Allow access by RID

1.2 New Operators

Query Processing

- Three new operators...
  - (Binary) Symmetric Hash Join
  - n-Ary Symmetric Hash Join (mJoin)
  - Eddy
- Developed in parallel databases or streams contexts
  - But useful in deterministic context as well
- Key difference between streams and disk-based
  - Push vs Pull
    - Iterators pull data (eventually from disk)
    - Streams push data into the query processor
    - Similarly, wide area data sources push data
  - Parallel query processing has a combination
    - push (across processor) and pull (within a processor)
    - Volcano paper (later)

Query Processing: Symmetric Hash Join

- Produces results immediately → Better time to first tuple
- Can implement as an iterator
  - Alternate pulling data from the two children
- Problems:
  - Larger memory requirement
  - Not as easy to extend to disk (XJoin)
n-Ary Symmetric Hash Join Operator (MJoin)

- For each relation: build a hash-table on each join attr.
- For each new tuple:
  - *insert* it into appropriate hash table(s)
  - *probe* into hash-tables on other relations

Example Query

```sql
SELECT *
FROM R, S, T, U
WHERE R.a = S.a
     AND S.b = T.b
     AND S.c = U.c
```

MJoin Operator

n-Ary Symmetric Hash Join Operator (MJoin)

- Intermediate tuples are never stored anywhere
- Need a policy for choosing the *probing sequences*
  - Similarities to *selection ordering*
Example Query

```sql
SELECT *
FROM R, S, T, U
WHERE R.a = S.a
AND S.b = T.b
AND S.c = U.c
```

MJoin Operator

Fig. 3.2 Executing a 4-way join query using the MJoin operator. The triangles denote the in-memory hash indexes built on the relations.

- **Rank ordering**: sort ascending by $c/(1 - p)$
  - where $c =$ cost of probing, $p =$ selectivity
- Can change the probing sequence anytime w/o problems (adaptivity)
- Many more details in Survey on Adaptive QP

- Issues:
  - Typically less efficient than a tree of binary joins

- Iterator ?
  - Can alternate pulling from different children

### 1.3 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
* Can be implemented as an iterator
  - See details in “An initial study of overheads of routing”, SIGMOD Record 2004

**Eddy/Tuple Router**

![Diagram](image)

Figure 2: Using traditional operators along with an eddy

**Eddy/Tuple Router**

![Diagram](image)

Figure 3: Eddy instantiated for the example query
Example Query

```
SELECT *
FROM R, S, T, U
WHERE R.a = S.a
AND S.b = T.b
AND T.c = U.c
AND P(T)
```

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

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
- 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

2 Query Optimization

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 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)

- 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