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”)

CMSC724: Query Processing

Amol Deshpande

February 24, 2009
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

**Diagram:**

```
R Hash Table
  Build R
  Probe with S

S Hash Table
  Build S
  Probe with R

RS Tuples
```

- 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

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

Example of Probing Sequences

```
R
R.a
S
S.a S.c S.b
T
T.b
U
U.c
```

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
  - 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
Example Query

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

**Examples of Probing Sequences**

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.

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
Figure 2: Using traditional operators along with an eddy

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

• Instantiate 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 ≠ R outerjoin (S join T)

Query Optimization

• Heuristic 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
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.

- 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 (± 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
3 Adaptive Query Processing

Why?

- Traditional optimization is breaking
- In traditional settings:
  - Queries over many tables
  - Unreliability of traditional cost estimation
  - Success and maturity make problems more apparent, critical
- In new environments:
  - e.g. data integration, web services, streams, P2P, sensor nets, hosting
  - Unknown and 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
AQP: Overview/Summary

• Low-overhead, evolutionary approaches
  – Typically apply to non-pipelined execution
  – **Late binding**: Don’t instantiate 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, or slides on the webpage)

4 Data Streams

4.1 Motivation

Data Streams

• Why?

• Much data generated continuously (growing every day)
  – Financial data
  – Sensors, RFID
  – Network/systems monitoring
  – Video/Audio data
  – etc...

• Need to support:
  – High data rates
  – Real-time processing with low latencies
  – Support for temporal reasoning (time-series operations)
  – Data dissemination
  – Distributed? (at least data generation)
  – etc...
Examples of Tasks

- **Continuous** (SQL) queries
  - E.g. moving average over last hour every 10 mins
  - SQL extended to support “windows” over streams
  - Proposed extensions: SEQUENCE, CQL, StreamSQL

- Pattern recognition
  - Alert me when: A, then B within 10 mins
  - How to specify? StreamSQL has some support

- Probabilistic modeling; Applying financial models
  - Infer hidden variables
  - Remove noise (from measured readings)
  - Do complex analysis to decide whether to buy
  - We don’t even know how to specify these

- Multimedia data?
  - Online object detection, activity detection
  - Correlating events from different streams

Data Streams

- Use traditional DBMS?

- Consider simplest case:
  - Report moving average over last hour every 10 minutes
  - 1. Insert all new items into database
  - 2. Execute the query every 10 minutes

- Not easily generalizable to other tasks
  - E.g. “alert me the moment moving average > 100”?

- Typically 1000’s of such continuous queries

- Even for one query, too slow and inefficient
  - Doesn’t reuse work from previous execution

- Application-level modules typically used for complex tasks
create trigger IrisHouseAlert
on insert to house
from salesperson s, house h, represents r
when s.name = ‘Iris’ and s.spno=r.spno and
r.nno=h.nno
do raise event
NewHouseInIrisNeighborhood(h.nno, h.address)

Figure 1: Trigger Example (Hansen et al.)

Data Streams

- Triggers?
  - Similar, but current trigger systems not designed for the required scale
- Publish-Subscribe Systems
  - Similar concepts: Push-based, reactive execution
  - Typically no complex queries
  - Much focus on “dissemination”
- Major research systems (late 90’s-early 00’s):
  - NiagaraCQ (Wisc), Telegraph, TelegraphCQ (Berkeley)
  - STREAM (Stanford), Autora, Borealis, Medusa (Brown/Brandeis/MIT)
- Commercial
  - Oracle*Streams, Strembase etc...

4.2 Triggerman

Scalable Trigger Processing (Hansen et al.)

- Goal: Handle millions of triggers
- Triggers: Commonly used for integrity constraint checking, alerts etc...

Scalable Trigger Processing (Hansen et al.)

- Goal: Handle millions of triggers
- Triggers: Commonly used for integrity constraint checking, alerts etc...

Scalable Trigger Processing (Hansen et al.)

- Approach:
  - Identify unique “expression signatures” (based on data sources and attributes involved)
  - Group the triggers into “equivalence” classes based on their signatures
  - Use efficient main memory data structures to quickly find triggers that match
- Many similarities to AI Rule systems
```sql
CREATE TABLE emp_auditlog (
    audit_date      DATE,
    audit_user      VARCHAR2(20),
    audit_desc      VARCHAR2(20))
;
CREATE OR REPLACE TRIGGER emp_audit_trig
AFTER INSERT OR UPDATE OR DELETE ON emp
DECLARE v_action VARCHAR2(20);
BEGIN
    IF INSERTING THEN
        v_action := 'Added employee(s)';
    ELSIF UPDATING THEN
        v_action := 'Updated employee(s)';
    ELSIF DELETING THEN
        v_action := 'Deleted employee(s)';
    END IF;
    INSERT INTO emp_auditlog VALUES (SYSDATE, USER, v_action);
END;
```

Figure 2: Trigger Example (Hansen et al.)

Figure 3: Triggerman (Hansen et al.)

Triggerman

Triggerman

Triggers

- Precursor to data streams work
- Event-driven as opposed to query driven
- Can handle pub-sub applications well
- Can identify quickly queries that should be executed
- But, no discussion on how to execute those queries efficiently
  - E.g., “moving average” query
Every new tuple will satisfy the query
Trigger action (compute moving avg) will be invoked per new tuple
- No sharing of work from previous execution
- No sharing of work between multiple triggers
  * E.g. If one person wants moving average over last hour, other person over last two hours

4.3 Major Concepts

Data Streams: Some Major Concepts

- New non-blocking operators
  - Symmetric hash join, MJoin, XJoin, Eddy etc...
- Adaptivity
  - Dealing with unpredictability
- Sharing/Multi-query optimization
  - 1000’s of queries; must share execution
- Load shedding
  - Bursty data: Too much to handle at some times
- Declarative languages
  - Especially for pattern recognition, modeling etc
- Theoretical developments
  - “One-pass” algorithms
Query execution

- Duality between queries and data
  - Traditional: Apply queries to data
  - Streams: Apply data to queries
- New operators
  - Symmetric hash join, XJoins
  - MJoin
- Predicate indexes
- Push vs Pull Execution
- Execution using a router
  - E.g. using an eddy

Stream Systems

- NiagaraCQ (Wisconsin)
  - Early work on data streams
- TelegraphCQ (Berkeley)
  - Based on eddies; implemented in PostgreSQL
  - Focus on adaptivity and sharing issues
  - Declarative querying interface: SQL-type
- Aurora (Brown/Brandeis/MIT)
  - Boxes-and-arrows paradigm for setting up dataflows
  - Much focus on Quality of Service
- STREAM (Stanford)
  - Addressed many issues including optimization, language design, approximate query answering, memory constraints etc.
- Much other work...

5 Sketches

Brief Aside: Sketches

- One-pass algorithms: You can only look at each data item once
- Goal: Compute some aggregate of interest
- Question: What is the amount of space needed if the data size is $N$ ?
– For exact or approximate computation

• Examples:
  – 1. Average: $O(1)$ (number of entries, total sum)
  – 2. Median:
    * Exact: Space complexity = $N$
    * Approximate: $O\left(\frac{1}{\epsilon} \log^2 \epsilon N\right)$, with $\epsilon$ error

FM-Sketches

• Flajolet-Martin Sketch: Count distinct number of values in a sequence in one pass with minimum memory
• $N =$ Length of the sequence
• $n =$ Number of distinct values

• Naive Approach:
  – Keep a list of all distinct values, and update incrementally
  – $O(n)$

• FM-Sketches: Approximate counting in $O(\log(n))$ space

FM-Sketches

• Algorithm:
  – Use a bitmap, $B$, of size $k$, where $k \approx \theta(\log_2(n))$
    * Aren’t we trying to estimate $n$ ?
    * Use a rough upper bound. Even if you overestimate by a factor of 4, you only use 2 more bits.
  – Need a uniform hash function: $h(x)$ maps values in the sequence to $\{0, \ldots, 2^k - 1\}$.
  – For each value, $v$ in the sequence, find $h(v)$.
  – Let $l(h(v))$ denote the least-significant 1 bit in $h(v)$.
    * $k = 6$, $h(v) = 000100$, then $l(v) = 3$.
    * $k = 6$, $h(v) = 000101$, then $l(v) = 1$.
  – Set $B(l(v)) = 1$.
  – Note: Duplicate values will just set the same bit again: “duplicate-insensitive”
FM-Sketches

- Algorithm (Cntd):
  - At the end, let \( c \) be the least-significant (right-most) 0 in \( B \)
  - \( 1.2928 \times 2^c \) is an estimator for the number of distinct values
  - Why ?
    * Choose a number, \( x \), uniformly between \( 0 \) to \( 2^k - 1 \).
    * \( \text{Prob}(l(x) = c) = 1/2^{c+1} \)
    * Hash function is assumed to map values in the sequence uniformly onto the above range as well
  - Use multiple hash functions for more confidence
  - Space: \( O(\log(n)) \)
  - Choosing hash functions ?
    * Tricky: uniform hash functions take a lot of space
    * Much work on relaxing the requirement

AMS Sketches

- Alon, Matias, Szegedy: Space Complexity of Approximating the Frequency Moments; STOC 1996

- Consider a stream: \((1, 2, 3, 1, 5, 2, 1, 3, 4)\)

- Let \( m_i \) be the frequency of \( i \) in the stream
  - \( m_1 = 3, m_2 = m_3 = 2, m_4 = m_5 = 1 \).

- Frequency moment \( F_k = \sum_{i=1}^{n} m_i^k \)
  - \( F_0 = 5 \) = number of distinct elements in the stream
  - \( F_1 = 9 \) = total number of elements in the stream
  - \( F_2 = 19 \) = comes in up many places (e.g. self-join size of a relation)

- How to compute ?
  - Exact computation: \( O(n) \), where \( n \) is the number of distinct elements, not the size of stream
  - Approximate: AMS Result: Can approximate \( F_0, F_1, F_2 \) in logarithmic space, requires \( O(n^{\Omega(1)}) \) space for others