CMSC724: Query Processing

Amol Deshpande

University of Maryland, College Park

March 6, 2011

Outline

Query Processing

- Assume single-user, single-threaded
  - Concurrency managed by lower layers
- Steps:
  - Parsing: attritube 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

Query Processing

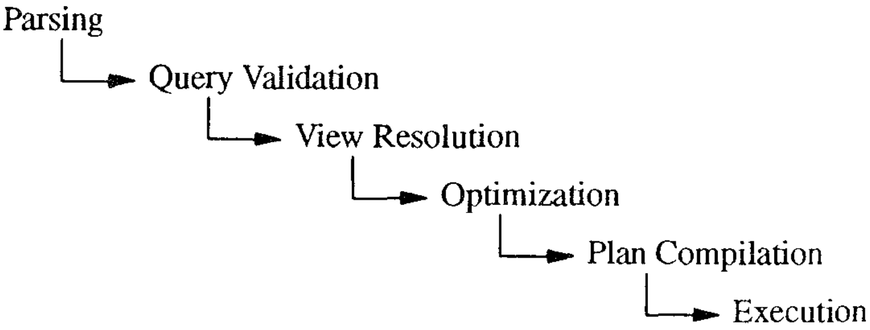
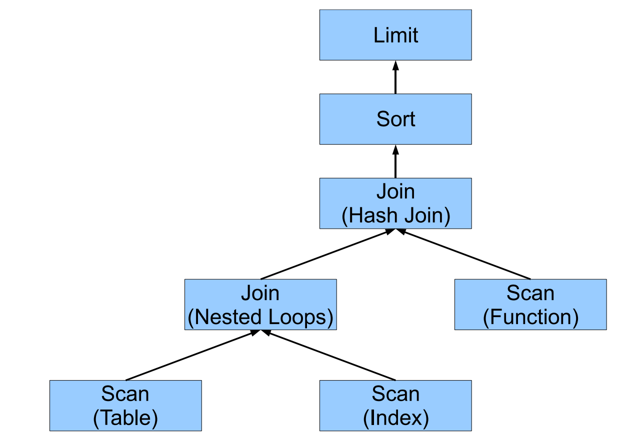


Figure 2. Query processing steps.

Example Query Plan



Query Plans

- Lot of confusion between left-deep vs right-deep
  - Careful when reading some of the early work
  - Think about hash joins
  - One of them builds hash tables on intermediate relations, one only on base tables

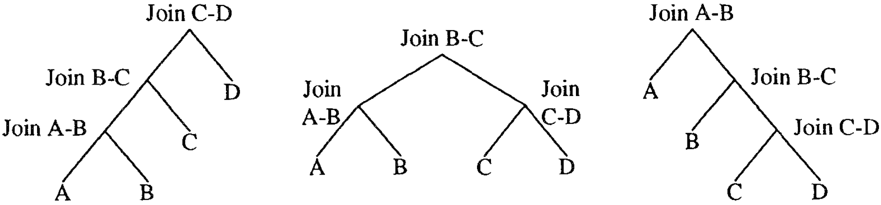


Figure 4. Left-deep, bushy, and right-deep plans.

Outline

Options for query processing

- Materialize the results after each operator
- Each operator runs in a separate process; use inter process commmunication
  - Use "queues" in between the operators to pass data
  - Too many context switches, but better parallelism
  - See the River system (Berkeley)
- Use threads?
  - Issues with blocking for I/Os
- Translation programs
  - Translate the plan into a single iterative program
  - Probably not feasible given the complexity of operators
- Iterator model
  - Single process/thread

Iterator Model

- Each operator implementation supports:
  - init/open()
    - Typically no data involved (although Graefe's examples do that)
  - get_next()
    - Return the next output tuple; may call get_next() on children
    - First call typically builds hash tables, sorted runs etc...
  - end/close()
  - rescan()
    - Often needed (e.g., for nested loops)

Iterator Examples

Table 1. Examples of Iterator Functions

Iterator	*Open*	*Next*	*Close*	Local State
Print	*open* input	call *next* on input; format the item on screen	*close* input	
Scan	open file	read next item	close file	open file descriptor
Select	*open* input	call *next* on input until an item qualifies	*close* input	
Hash join (without overflow resolution)	allocate hash directory; *open* left "build" input; build hash table calling *next* on build input; *close* build input; *open* right "probe" input	call *next* on probe input until a match is found	*close* probe input; deallocate hash directory	hash directory
Merge-Join (without duplicates)	*open* both inputs	get *next* item from input with smaller key until a match is found	*close* both inputs	
Sort	*open* input; build all initial run files calling *next* on input; *close* input; merge run files until only one merge step is left	determine next output item; read new item from the correct run file	destroy remaining run files	merge heap, open file descriptors for run files

Iterator Model

- DAGs: use "split" operator
  - Multiple consumers – buffer each input tuple till all consumers have seen it
  - Can use a bitmap for this purpose
  - Only need to spool to disks if consumer rates vary too much
- No parallelism – what about multi-core?
  - Also, shared-nothing parallel databases (i.e., no shared memory)?
- Use special "Exchange" operators

Iterator Model

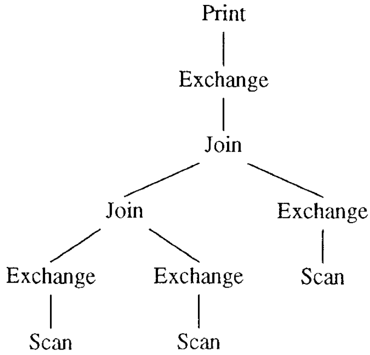


Figure 26. Operator model of parallelization.

Sorting and Hashing

- Two key strategies underlying almost all operators
- Critical differences:
  - Hashing can be pipelined vs sorting is *blocking*
  - Sorting-based operators produce output in *sorted order*
- Can be seen as duals of each other
  - Very nice observations in the paper on this topic

External Sorting

- Basic idea:
  - Create sorted "runs"
  - Merge the sorted "runs"
- How to create runs?
  - (1) Read as much as memory; use quicksort
  - (2) Use replacement selection
    - Option 2 much better – produces runs that are much larger, and hence smaller number of runs
    - If the input almost sorted, can get away with just one run

External Sorting

- Graefe has a survey on this topic alone
- Some interesting points:
  - Need to worry about random vs sequential I/Os
    - In case of sorting: when merging, random I/O is required
  - General technique: read many blocks at once
    - For sorting: that reduces the number of runs you can merge at once
    - However in some cases, that may be better since random I/O so much slower
    - Probably not a big deal now-a-days
  - Hybrid hash seems superior when the amount of data just larger than memory
  - However "reverse" writes help (can be explicitly coded)
    - Write the run in reverse order
    - The tail of the run will be in the buffer when merging so avoid that I/O

Hashing

- Very good option if the table fits in memory
  - For hash joins, only the smaller input needs to fit in memory
- If not, then need to do in multiple phases
- Hybrid hashing
  - Optimal when the (build) relation just larger than memory
  - Can keep most of the hash table in memory, and spill some to disk
- Some issues to keep in mind
  - Quality of hash functions
  - Must deal with skew

Disk Access

- Sequential Scan
  - Push down selections and apply as soon as possible
  - Also push down projections
  - Interesting issues in choosing the order in which to apply selection predicates
- Associate (Index) Access
  - B+-Tree indexes very widely used
    - Data warehouses often build them on every column
  - Other indexes not typically supported in database systems even today
    - Some spatial-oriented databases support R-Trees or variants
  - Perhaps the key reason is that the complexity is not seen as worth the effort
    - Especially complexity of dealing with concurrency and recovery issues

Disk Access

- Index-only Scans
  - Often we don't need to retrieve the records, the lowest level of index has sufficient information
- Index-ANDING and Index-ORING
  - Important optimizations
  - e.g., imagine two predicates on a relation, both on columns with indexes
  - Can get a list of RID (record ids) from both of them, intersect (or union), sort, and retreive the tuples in one pass

Disk Access: Different Storage Technologies

- What happens if the underlying storage device is not a standard disk?
  - RAID
    - Very commonly used in large server deployments
    - Many issues with use in databases
    - The costs of reads and writes are different
    - Failure behavior is different from standard disks (since RAID automatically recovers)
    - Using write caches can be problems
    - Writing "parity" blocks not required for temporary data (e.g., sorted runs etc)
  - Many of these hidden underneath an abstraction layer
  - Database vendors must deal with this
  - Flash?

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”)

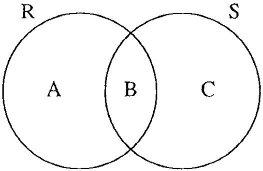
Executor: Operators

- 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
- Typically implemented using sorting, sometimes hashing
- 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



Output	Match on all Attributes	Match on some Attributes
A	Difference	Anti-semi-join
B	Intersection	Join, semi-join
C	Difference	Anti-semi-join
A, B		Left outer join
A, C	Symmetric difference	Anti-join
B, C		Right outer join
A, B, C	Union	Symmetric outer join

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 repeatadly issueing 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

Executor: Memory Allocation

- Commercial systems (also PostgreSQL) uses context-based memory allocators
- Each operator creates its own context
  - Allocates memory in that context (through special calls: "pmalloc" for PostgreSQL)
  - Entire context deallocated at once (after finished)
  - Essentially a custom garbage collector
- Data movement between operators
  - Each operator typically has a few "slots" it uses for data movement that are shared
  - When it is called by a parent operator, the next tuple is copied into a shared slot
  - Thus, avoid creation of new objects
- Any long-lived data (e.g., hash tables) copied into operator contexts

Outline

Data Warehouses

- A (usually) stand-alone system that integrates data from everywhere
  - Read-only, updated at night
  - Geared toward business analytics, data mining etc...
- Heavily used and heavily optimized
  1. Materialized views (summary tables, data cubes)
  2. New types of indexes
  3. New join techniques geared toward “star” (or “snowflake”) schemas
  4. Compressed storage techniques
- Key observation: Read-only, so updating not an issue

Data Warehouses: Overview

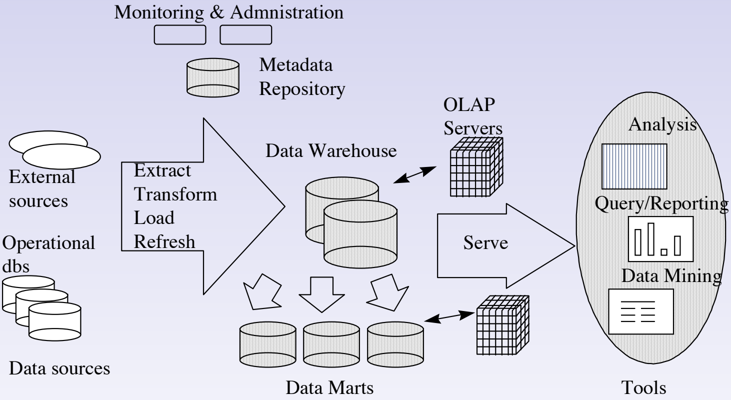


Figure 1. Data Warehousing Architecture

Figure: Overview (From Chaudhuri, Dayal; SIGMOD Rec., 1997)

Data Warehouses

- Extract-Transform-Load (ETL)
  - Data cleaning, auditing, integrity constraints
  - Semantic heterogeneity
    - Issues like entity resolution, schema mapping/matching, cleaning etc..
- Load/Refresh:
  - Typically done periodically
  - Batch loading, so can heavily optimize the indexes
    - E.g. If using a B+-tree, bulk-loading can result in much better indexes, than inserting one at a time
  - Refresh:
    - Usually done incrementally, at night or something
- Real-time analysis ? Typically not done today

Data Warehouses: Star Schema

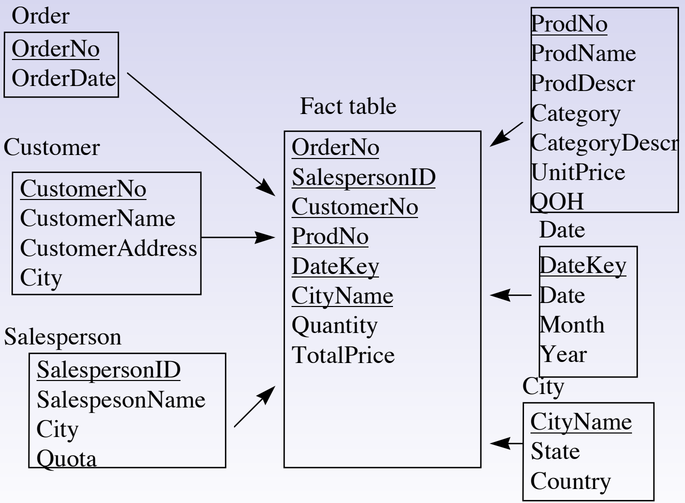


Figure 3. A Star Schema.

Data Warehouses: Snowflake Schema

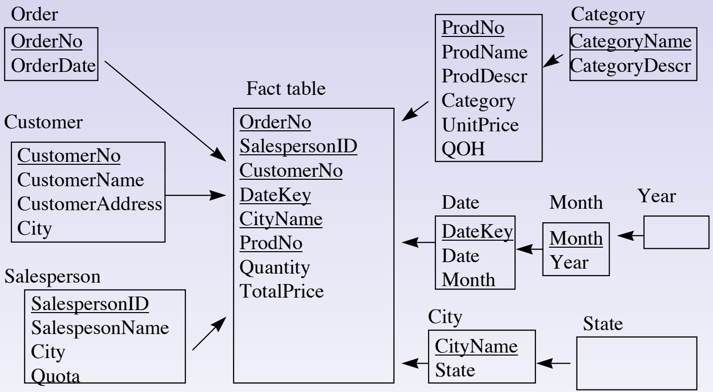


Figure 4. A Snowflake Schema.

Figure: A Snowflake Schema (From Chaudhuri, Dayal; SIGMOD Record, 1997)

Star and Snowflake Schemas

- The Facts table is HUGE
  - Dimension tables relatively small
- Strong key-foreign key dependencies
  - Each fact table tuple joins with exactly one tuple from each dimension table
  - Critical in optimizations

Star and Snowflake Schemas

- The Facts table is HUGE
  - Dimension tables relatively small
- Strong key-foreign key dependencies
  - Each fact table tuple joins with exactly one tuple from each dimension table
  - Critical in optimizations
- Many queries are of the form:
  - Join the Facts table with some of the dimension tables
  - Selections on the dimension table attributes (e.g. state = 'MD')
  - Possibly selection on the fact table
  - Group by on some of the dimension table attributes (e.g. ProdName)
  - Aggregate on a main Facts table attribute (e.g. quantity)

Disk Access: Bit-map indexes

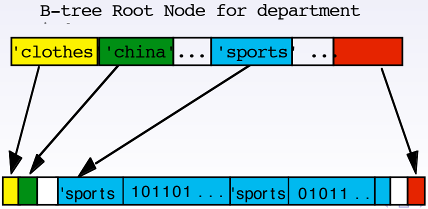
- Variant Indexes; O'Neil, Quass; SIGMOD'97
- Specialized indexes for supporting summary aggregate queries
- Different types of indexes:
  - Traditional Value-List Indexes
  - Bitmap Indexes
  - Projection Indexes
    - Very similar to Column-based storage (much research last few years)
  - Bit-sliced Indexes
  - Join Indexes
- Key observation: Read-only database, so can build as many indexes as you want

Value-List/Bitmap Index

- Key idea: Given a property over a domain, the following two are interchangeable and complementary
  - a list of values
  - a bitmap over the domain

Value-List/Bitmap Index

- Key idea: Given a property over a domain, the following two are interchangeable and complementary
  - a list of values
  - a bitmap over the domain
- In our case:
  - Domain: The set of all RIDs
  - Property: A predicate $R.a =' Sports'$
- If the number of RID that satisfy the property is:
  - small: store as a list of RIDs
  - large: store as a bitmap over the RIDs



Value-List/Bitmap Index: Segmentation

- Each disk page can store 48K bits, so must partition the Facts table into 48K row partitions
- So
  - Each B+-Tree page contains a portion of the bitmap over the RIDs

Value-List/Bitmap Index: Segmentation

- Each disk page can store 48K bits, so must partition the Facts table into 48K row partitions
- So
  - Each B+-Tree page contains a portion of the bitmap over the RIDs
- If the number of 1's is small, convert to an RID-list
  - The tipping point is when the number of 1's is < 1/32 of the size.
  - At that point, the RID-list exactly fits in the disk page
  - $(48000/32 = 1500, 1500 * 4 = 6K)$
  - This is always true regardless of the page size
- Segmentation also helps with space storage... if an entire segment is all 0's, don't store it

Value-List/Bitmap Index: Queries

- Selections on the table return bitmaps
  - AND, OR, NOT very fast on bitmaps
  - Result called a Foundset: B_f (the domain is the Facts Table)
- Next step: Aggregate (recall almost all queries compute aggregates)
  - Can perform directly on the bitmap in some cases (COUNT)
  - Otherwise use projection indexes
  - OR use a bit-sliced index

Value-List/Bitmap Index: COUNT

- shcount: count the number of ones in the binary representation
  - shcount[3] = 2, shcount[10] = 2, shcount[15] = 4 etc...

Algorithm 2.1. Performing COUNT with a Bitmap

```
/* Assume B1[ ] is a short int array
   overlaying a Foundset Bitmap             */
count = 0;
for (i = 0; i < SHNUM; i++)
   count += shcount[B1[i]];
/* add count of bits for next short int     */
```

Figure: Bitmap Index

Outline

Overview

- Traditional databases row-oriented
  - Fast for writes, but not for reads
  - Redundant columns accessed
  - Index-only scans help but are not sufficient
- Column-stores
  - Store data in columnar fashion
  - Better I/O and CPU efficiency (fewer cache misses)
  - Tuple reconstructions costs quite high
  - Better for scan queries (i.e., queries that don't focus on just a few tuples)
  - Big push toward this in recent years with increasing trend toward data warehousing and analytics
  - Many commercial systems support some mix of columnar- and row-oriented storage
- Very nice overview article by Dan Abadi

C-Store

- Commercialized as Vertica (recently acquired by HP)
- Key features (from VLDB 2005 paper – may have changed since):
  - Hybrid architecture: A Write Store (WS) optimized for inserts, and a Read Store (RS) optimized for querying
    - Data moved from WS to RS in a periodic fashion
  - Columns stored in possibly different sort orders
    - A single column may be stored multiple times in different sort orders
    - For read efficiency
  - Heavy use of compression
  - Designed for a shared-nothing environment
    - High availability through use of overlapping projections
  - Use of snapshot isolation to avoid 2PC and locking

C-Store

- "Projection" defined by:
  - An anchor table
  - A list of attributes from anchor table
  - A list of attributes from other tables s.t. the attribute values are uniquely defined
    - Through a sequence of key-foreign key joins
  - A sort order
- No. of tuples in a projection = No. of tuples in the anchor table
- Projections may be horizontally partitioned into segments based on the sort key

C-Store

- Example:
  - *EMP(name, age, salary, dept)*, *DEPT(dname, floor)*
  - Possible list of projections:
    - EMP1(name, age | age) – *age is the sort key*
    - EMP2(dept, age, DEPT.floor | DEPT.floor)
    - DEPT.floor uniquely associated with a tuple from the anchor table
    - EMP3(name, salary | salary)
    - DEPT1(dname, floor | floor)

C-Store

- Need mappings between different projections to be able to construct original rows
  - Called "join indexes"
- Late materialization
  - At some point, you must stitch the columns of a single table together
  - Try to postpone because that operation is expensive
  - e.g., apply selection predicates first, and then only constructs tuples that match

C-Store

- Need mappings between different projections to be able to construct original rows
  - Called "join indexes"
- Late materialization
  - At some point, you must stitch the columns of a single table together
  - Try to postpone because that operation is expensive
  - e.g., apply selection predicates first, and then only constructs tuples that match
- Block iteration
  - As opposed to get_next() interface, pass entire block of tuples between operators
  - Avoids per-tuple overheads
  - Can be done in row-stores as well, but easier in column-stores

Comparing column-stores to row-stores

- Ways we can try to get column-store benefits in a row-store
  - Vertically partition each table into a collection of two-column tables: (key, attribute)
    - Using a synthetic position attribute instead of key may be better
  - Build indexes on every attribute and use index-only scans
    - Used aggressively in commercial systems, although PostgreSQL doesn't support them
  - Aggressive use of materialized views
    - Need to know the query workload
- However, results indicate the overheads of all these approaches are too high

Outline

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

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

Outline

Data Streams

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

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

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

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

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

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

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”

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...

Outline

Scalable Trigger Processing (Hansen et al.)

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

```
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.hno, h.address)
```

Figure: Trigger Example (Hansen et al.)

Scalable Trigger Processing (Hansen et al.)

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

```
CREATE TABLE empauditlog (
    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 empauditlog VALUES (SYSDATE, USER,
        v_action);
END;
```

Figure: Trigger Example (Hansen et al.)

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

Triggerman

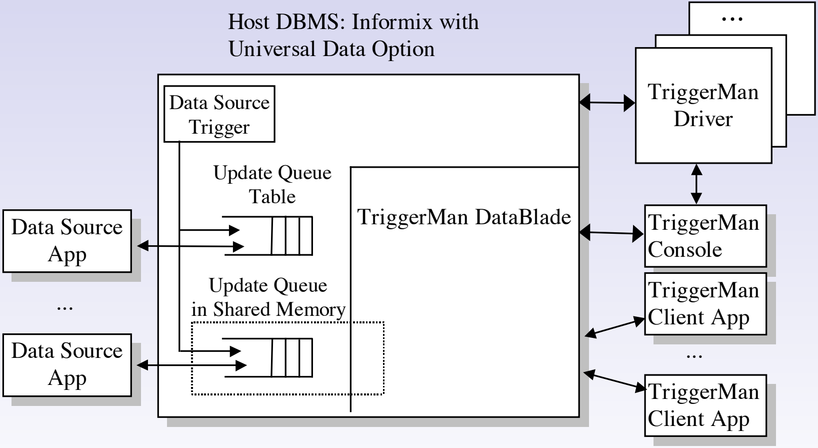


Figure: Triggerman (Hansen et al.)

Triggerman

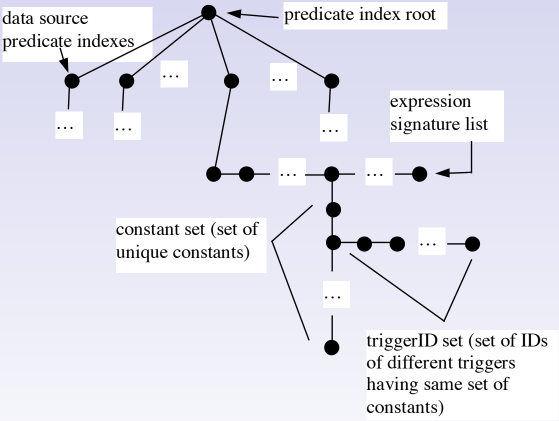


Figure: Triggerman (Hansen et al.)

Triggers

- Precursor to data streams work
- Event-driven as opposed to query driven
- Can handle pub-sub applications well

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

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

Outline

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

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*

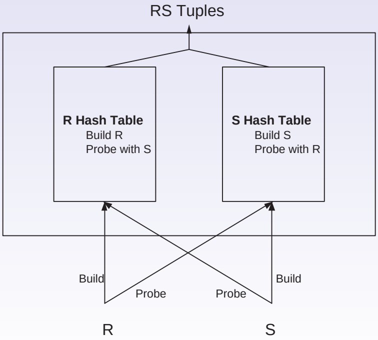
Outline

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)



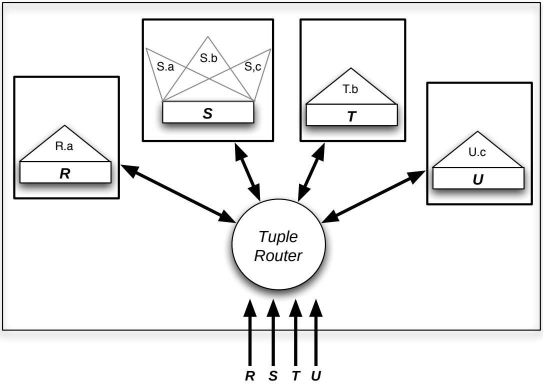
n-Ary Symmetric Hash Join Operator (MJoin)

- 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

```
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)

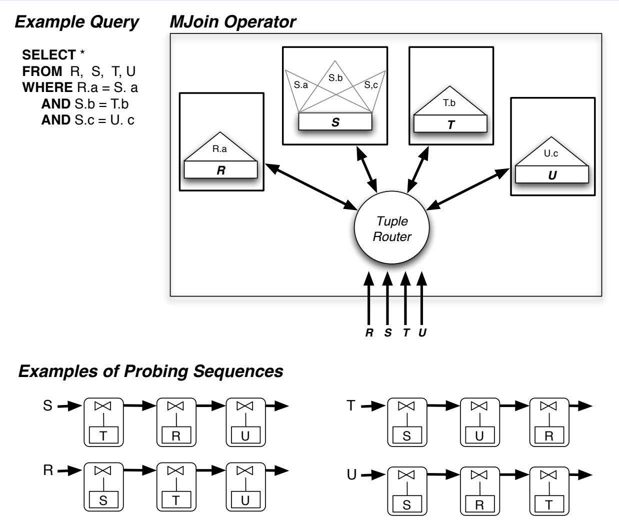


Fig. 3.2 Executing a 4-way join query using the MJoin operator. The triangles denote the in-memory hash indexes built on the relations.

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 $\boxed{\text{Survey on Adaptive QP}}$

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 $\boxed{\text{Survey on Adaptive QP}}$
- Issues:
  - Typically less efficient than a tree of binary joins
- Iterator ?
  - Can alternate pulling from different children

Outline

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 200

Eddy/Tuple Router

select count(*)
from R, S, T
where R.a = S.a and S.b = T.b
and pred(R.c)

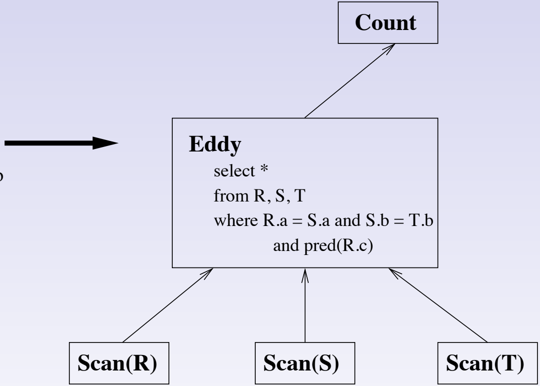


Figure 2: Using traditional operators along with an eddy

Eddy/Tuple Router

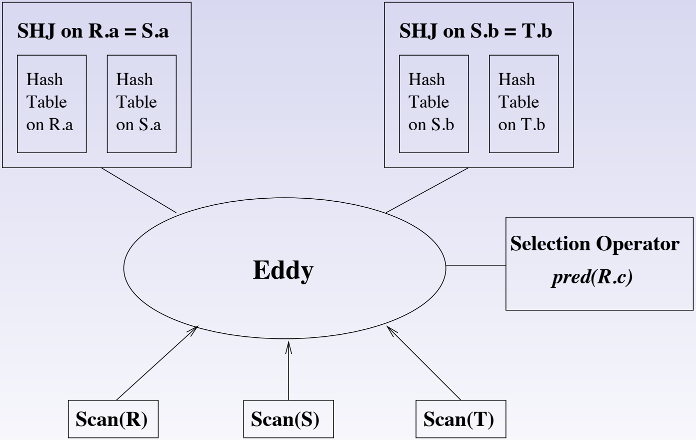
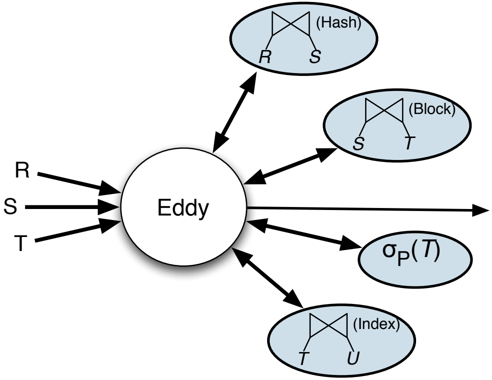


Figure 3: Eddy instantiated for the example query

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 σ_P(T)



Tuple Signature

Base Tables	Routed Through	Valid Destinations
{R}	{}	(R ⋈ S, 1.0)
{S, T}	{S⋈T, σ_P(T)}	(R⋈S, 0.3), (T⋈U, 0.7)
..	..	..

Routing Table

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

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..

Outline

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

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: $O(\frac{1}{\epsilon}\log^2 \epsilon N)$, with ϵ 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 is $\approx \theta(log_2(n))$
    - 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.

FM-Sketches

- Algorithm:
  - Use a bitmap, B, of size k, where k is $\approx \theta(log_2(n))$
    - 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 $\{0, \cdots, 2^k - 1\}$.
  - For each value, v in the sequence, find $h(v)$.

FM-Sketches

- Algorithm:
  - Use a bitmap, B, of size k, where k is $\approx \theta(log_2(n))$
    - 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 $\{0, \cdots, 2^k - 1\}$.
  - 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 $l(v) = 1$.

FM-Sketches

- Algorithm:
  - Use a bitmap, B, of size k, where k is $\approx \theta(log_2(n))$
    - 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 $\{0, \cdots, 2^k - 1\}$.
  - 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
  - 1.2928×2^c is an estimator for the number of distinct values

FM-Sketches

- Algorithm (Cntd):
  - At the end, let c be the least-significant (right-most) 0 in B
  - 1.2928×2^c is an estimator for the number of distinct values
  - Why ?
    - Choose a number, x, uniformly between 0 to $2^k - 1$.
    - $Prob(l(x) = c) = 1/2^{c+1}$
    - Hash function is assumed to map values in the sequence uniformly onto the above range as well

FM-Sketches

- Algorithm (Cntd):
  - At the end, let c be the least-significant (right-most) 0 in B
  - 1.2928×2^c is an estimator for the number of distinct values
  - Why ?
    - Choose a number, x, uniformly between 0 to $2^k - 1$.
    - $Prob(l(x) = c) = 1/2^{c+1}$
    - 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 $F_k = \Sigma_{i=1}^{n} m_i^k$

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)

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 $F_k = \Sigma_{i=1}^{n} m_i^k$
  - $F_0 = 5$ = 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