Online, Adaptive Query Processing

CPS 216

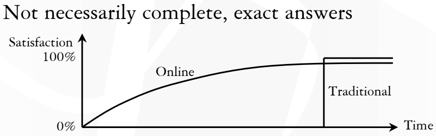
Advanced Database Systems

Announcements

- ❖ Homework #3 due today (Wednesday, April 9)
- ❖ Homework #4 due in 14 days (Wednesday, April 23)
- ❖ Project milestone #2 due in 5 days (Monday, April 14)

Online query processing

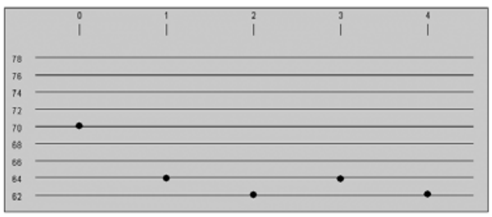
- ❖ Traditional query processing offers:
  - Complete, exact results
  - Long processing times for large inputs
- ❖ Users wants:
  - Real-time interaction
  - Iterative querying with progressive refinement
  - Not necessarily complete, exact answers



Online aggregation example

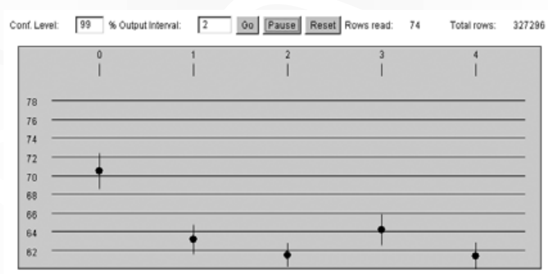
`SELECT AVG(temp) FROM R GROUP BY site;`

- ❖ 330K rows in table
- ❖ Exact answer:



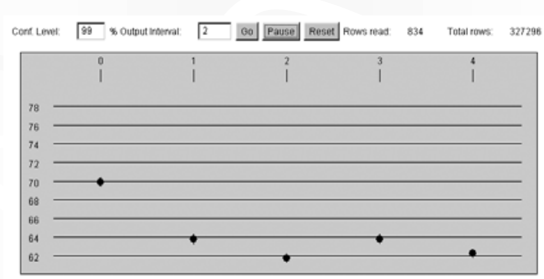
Online aggregation example (cont'd)

- ❖ Online aggregation, after 74 rows

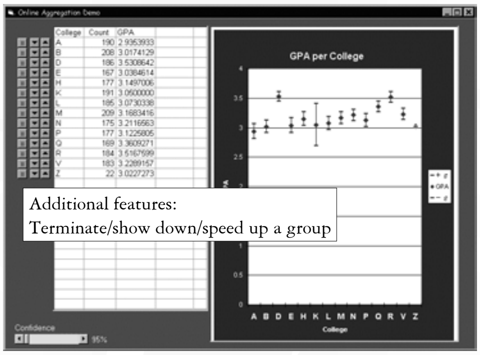


Online aggregation example (cont'd)

- ❖ Online aggregation, after 834 rows



Online aggregation example (cont'd)



Additional features:
Terminate/show down/speed up a group

Goals

- Usability
  - Continuous observation
  - Control of time/precision
  - Control of fairness/partiality
- Performance
  - Minimum time to accuracy: produce an acceptable estimate A.S.A.P.
  - Minimum time to completion: only a secondary goal, assuming user will terminate processing long before the final answer is produced
  - Pacing: provide a smooth and continuously improving display

Random access to data

Needed to produce statistically meaningful estimates
Example: random *R.temp* values wanted

- Heap scan
  - Make sure *R* is not sorted by *temp*!
  - Ideally, sort *R* by `rand()`
- Index scan
  - Use an index on an uncorrelated column, say *R.A*
- Sampling from indices
  - Probe random index blocks; less efficient
- Extent-map sampling
  - Pick a random block, then a random row within a block
  - Use acceptance/rejection sampling if block has variable number of rows

GROUP BY

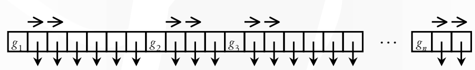
Choice: sorting or hashing

- ❖ Sorting is bad
- ❖ Hashing is non-blocking
  - Good performance if hash table fits in memory
  - Hybrid Cache is a good alternative otherwise

`DISTINCT` processing is similar

Index striding

- ❖ Hashing alone still may be unfair
  - Random stream of input rows → updates to small groups will be infrequent
  - Round-robin to support predictable progress across all groups, while still providing randomness within each group
    - Round-robin can be weighted to support equal-width confidence intervals or partiality
- ❖ Use a B^+-tree index on the grouping column

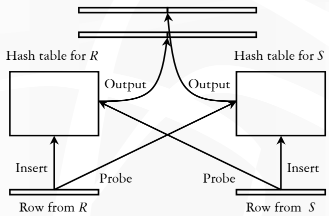


Join algorithms

- ❖ Sort-merge join is unacceptable
  - Sorting is blocking
  - Output is sorted by join column; problematic if it happens to be (or correlated with) grouping or aggregated column
- ❖ Traditional hash join also blocks (until a hash table has been constructed)
- ❖ Pipelined hash join should be used instead
- ❖ Nested-loop join is safest but slow; index nested-loop join fares better
- ☞ There are newer algorithms specifically designed for online aggregation: ripple join (*SIGMOD* 1999)

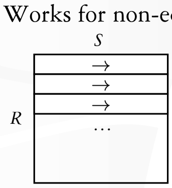
Pipelined hash join

- Non-blocking and symmetric
- Only works for equijoins

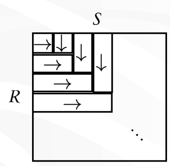


Ripple join (block-based)

- Non-blocking and symmetric
- Works for non-equality joins



Block-based nested-loop join



Block-based ripple join

Optimization issues

- Avoid sorting completely
- Interesting orders → interestingly bad orders (desirable for grouping and aggregated columns)
- Divide cost metric in two parts
  - Time t_d spent in blocking operations
  - Time t_o spend in producing output
  - Use combined cost function $f(t_o) + g(t_d)$, where f is linear and g is super-linear, to "tax" operators with too much dead time
- Prefer plans with more user control (e.g., index striding)
  - But how to quantify the degree of user control?
- Output rate vs. time to completion trade-off
  - Fast, bursty plan or slow, steady plan?

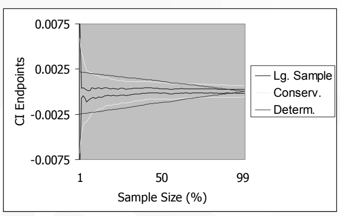
16

Running confidence intervals

- Suppose we have processed n random input rows and computed a running aggregate value v
- Given a confidence parameter $p \in (0, 1)$
- Display a precision parameter ε such that
  - Current v is within ε away from the final answer v_{exact} with probability approximately equal to p

17

Types of confidence intervals



P = probability that v is within ε from v_{exact}

- Conservative: $P \geq p$
  - Better guarantee, but too conservative
  - Requires knowledge of bounds on input values
- Large-sample: $P \approx p$
  - Looks better, but no guarantee!
  - Assumes that n is small enough to behave like a random sample with replacement
  - Assumes that n is large enough so Central Limit Theorem applies
- Deterministic: $P = 1$
  - Only useful when n is very large
  - Requires knowledge of bounds on input values

18

Adaptive query processing

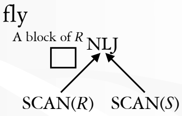
- Resources exhibit fluctuating characteristics
  - Examples: networks with fluctuating performance, sensors with fluctuating output rates, etc.
- Traditional DBMS
  - Recompute statistics daily/weekly and use them in query optimization in next day/week
  - ☞Adaptivity at daily/weekly basis
  - What if characteristics change during execution?
- Eddies: continuous adaptivity at execution time
  - ☞Adaptivity at a per-tuple basis!

Adaptable joins, issue 1

- Synchronization barriers
  - One input frozen, waiting for the other
  - Example: merging two sorted streams *slow-low* and *fast-high*; *fast-high* waits for *slow-low* to catch up
  - Cannot adapt while waiting for barrier
  - Favor joins that have fewer barriers

Adaptable joins: issue 2

- Would like to reorder joins on the fly
- Base case: swap inputs to a join
  - What about per-input state?

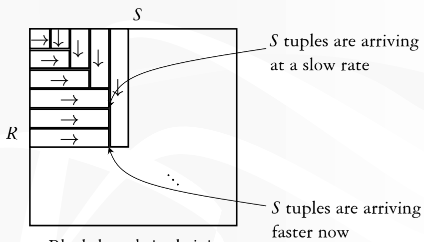


- Moment of symmetry
  - Inputs can be swapped without state management
  - Nested-loop join
  - Merge join
  - Traditional hash join
- More moments of symmetry → more opportunities for adaptation

Adaptable join algorithms

- Pipelined hash join
  - Synchronization barriers:
  - Moments of symmetry:
  - Good for equijoins
- Block-based ripple join
  - Synchronization barriers:
  - Moments of symmetry:
  - Good for non-equality joins

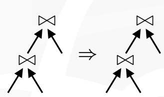
Example of adaptation



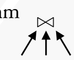
Block-based ripple join

Reordering beyond two-table joins

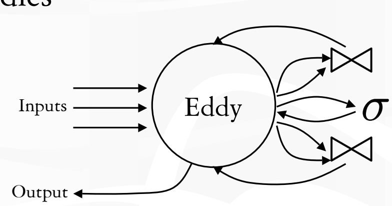
- Think of swapping "inners"
  - Can be done at a global moment of symmetry



- Intuition: like an *n*-way join operator
  - Except that each pair of inputs can be joined by a different algorithm



Eddies



- An iterator for tuple-routing
  - Essentially merges multiple participating operators into one *n*-ary operator implementing an adaptive query plan
  - Controls processing order dynamically within this plan

Some details

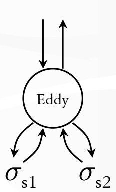
- Operators run as independent threads
- All edges are finite message queues
- Each tuple has a descriptor
  - A vector of ready bits
    - 1 if the corresponding operator is eligible to process the tuple
    - Eddy can turn on ready bits together (more flexible) or in order (more control)
  - A vector of done bits
    - 1 if the tuple has been processed by (returned from) the corresponding operator
    - Eddy returns the tuple if all done bits are set

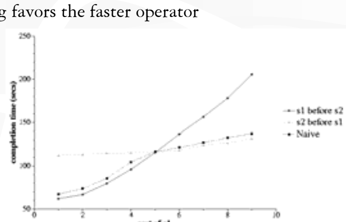
Naïve scheduling

- Tuples enter the eddy with low priority and receive high priority when they return from operators
  - Ensures that tuples flow completely through the eddy before new input tuples are admitted
- Operators fetch high-priority tuples as fast as they could

Experiment

- Two expensive selections with 50% selectivity each
  - Cost of s2 = 5; vary cost of s1
  - Naïve scheduling favors the faster operator

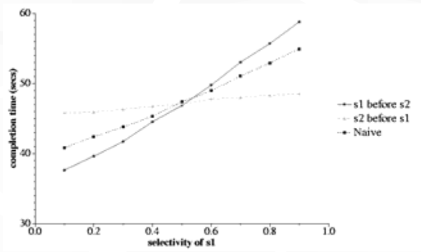




- Note: For "fair" comparison, static plans are implemented as eddies that set ready bits in order

Naïve is not enough

- ❖ Two expensive selections with identical cost
  - Selectivity of s2 = 50%; vary selectivity of s1

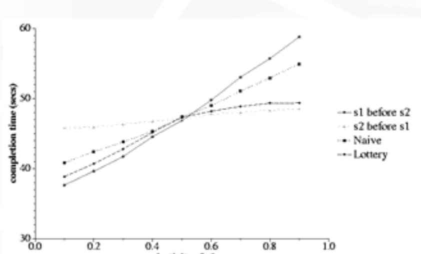


Eddies with lottery scheduling

- ❖ Operator gets 1 ticket when it takes a tuple
  - Favor operators that run fast (low cost)
- ❖ Operator loses a ticket when it returns a tuple
  - Favor operators with low selectivity
- ❖ When two operators compete for the same tuple, hold a lottery
- ❖ To improve adaptivity, use a window scheme that favors recent history

Performance

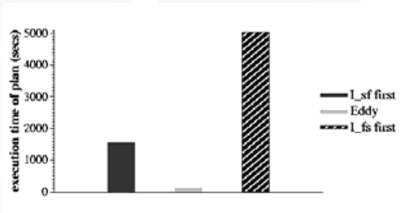
- ❖ Two expensive selections with identical cost
  - Selectivity of s2 = 50%; vary selectivity of s1



31

Performance in a volatile environment

- Two index joins
  - Slow: 5 seconds delay per lookup
  - Fast: no delay
  - Swap speeds after 30 seconds



32

Related work

- Query scrambling
  - Change execution order to avoid problems incurred by initial delays in receiving first tuples from remote sources
- Runtime re-optimization
  - Execute, monitor statistics, and re-optimize on the fly
- References
  - "Adaptive Query Processing: Technology in Evolution," by Hellerstein et al. *Data Engineering Bulletin*, 2000
  - "Adaptive Query Processing: A Survey," by Gounaris et al. *BNCOD* 2002