Tukwila and Eddies

Two Adaptive Query Execution Systems

Outline

- Motivation
- Tukwila
- Eddies
- Evaluation

Data integration

- Extreme form of distributed database
- Sources of data all over the internet (unrelated)
- Query optimization is difficult to do in advance
  - Missing/obsolete statistics
  - Widely variable data arrival rate
  - Overlap/redundancy among data sources

Adaptive query execution

- Mariposa accepts that statistics may be unavailable or unreliable, so it asks for a cost estimate
  - But then execution is fixed – if an estimate is wrong, the query will be slow
- Adaptive query execution
  - Do not just try to *predict* the best execution plan
  - Instead, *react* to the data as it arrives and *adapt* the plan
- Emphasis on time to first result (online use)

Tukwila architecture

- Mediated schema for queries
- Data source catalog
  - Overlap information
  - Statistics (size of relations, time to access)
- Query rewriting (as discussed last class)
- Optimizer (may be called multiple times per query)
- Adaptive execution engine
- Stream wrappers to normalize data vs schema

Interleaved optimization

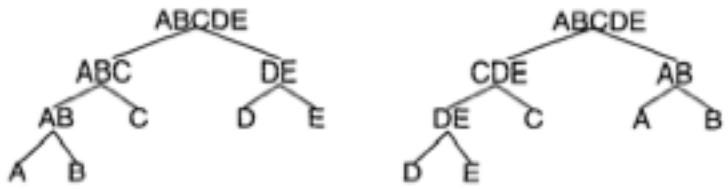
- The core of Tukwila's adaptive query engine
  - Guess an initial plan (does not need to be complete)
  - Produce a new plan when:
    - Reach the end of the current plan
    - Events trigger reoptimization
      - Timeout
      - Go over a row threshold
      - Etc
    - Events generated by rules, triggers inserted by optimizer

Fragments

- Tukwila can only reoptimize at certain points
  - Pipelining between operators prevents restructuring
- The optimizer splits a query plan (graph) into *fragments*
  - Each fragment executes atomically
  - Full pipelining *within* fragments
  - Results materialized *between* fragments

Example

```
Select * from A,B,C,D,E
where A.ssn =B.ssn
and   B.ssn=C.ssn
and   C.ssn=D.ssn
and   D.ssn=E.ssn
```



- Five table join
- If all tables are small, pipeline everything
- If size is unknown, start with a random set of independent joins
- hash join fragments (e.g., AB), materialize result, replan
- Can also try different orderings for *events* (e.g., timeout)

Dynamic collectors

- Perform union on disjoint result sets (e.g., overlap from different data sources)
- Can take action according to a *policy* (rules) to collect in a particular order, or switch on or off alternative sources
- Boolean combination of
  - Closed
  - Error
  - Timeout
  - Threshold

Double pipelined hash join

- Conventional hash join:
  - read *all* of the (hopefully!) smaller side into hash table
  - stream tuples from other side through hash (pipelined)
- Double pipelined hash join:
  - As rows come in from either side
    - Hash them
    - Probe against other side's partial hash table
  - Produces output tuples as soon as possible, but consumes much more memory (two hashes)

Double pipelined hash join vs RAM

- Expected to be used on relatively small tables (typical data integration scenario)
- Maintains a hash of both sides of the join (not just the smaller), so it consumes much more RAM
- Policy for overflow resembles that of hybrid hash join, but with a choice about which side to favor
  - Incremental Left Flush
  - Incremental Symmetric Flush

Memory overflow

- Incremental Left Flush
  - Stop reading left
  - Read right until end, paging out left as necessary
    - There may be a long pause here!
  - Continue reading left (standard hybrid hash)
- Incremental Symmetric Flush
  - Flush same bucket on both sides
  - Keep reading from either side as before
  - Steadier output, but may miss more

Data-driven iteration

- Normal query architecture is “pull”
  - Request of a tuple from the output side causes an operator to process (deterministically) until it produces one tuple
- Adaptive execution is data-driven
  - Feed operands to operator as they arrive
- Tukwila creates a thread per stream (inputs and output), where each thread tries to keep a small transfer buffer full

Discussion

- Q1: Would you use the double pipelined hash join if you were not doing data integration, why or why not?

Discussion

- Q2: The authors mention that the data is unpredictable due to the absence of statistics, arrival characteristics and overlap and redundancy among sources. Do you agree? Can you imagine ways to make the data more predictable? What would be the problems/challenges?

Eddies

- More general and fine-grained than Tukwila
- Designed for parallelism
  - Keep the pipelines full
- The engine behind Telegraph
  - “intended to run queries over all the data available on line”

Challenges for Telegraph

- Hardware and workload complexity
  - Bursty access patterns
  - Heterogeneous hardware
- Data complexity
  - Non-alphanumeric data (e.g., objects)
  - Poor/no statistics for remote sources
- User interface “complexity”
  - Allow users to interact with query while it runs

What's an eddy?

- A scheduler for tuple processing
- Figures out which rows from the input tables to read next, and which operators to feed them to first
- Deals with three kinds of variation:
  - Operator cost
  - Operator selectivity
  - Rate at which data arrives from inputs

Varying operator cost

```
SELECT foo.x, bar.y
FROM foo, bar
WHERE fibo(foo.x) / 1000 = 0
AND fact(bar.y) / 1000 = 0
AND foo.id = bar.id
```

- Assume foo is sorted by x ascending, and bar is sorted by y descending
- Initially it is cheaper to process foo, but later it becomes much more expensive than processing bar

Varying selectivity

- SELECT * FROM foo WHERE n > 20 AND n < 40
- INPUT: foo
- OUTPUT: a very small subset of foo
- Selectivity of predicates depends on distribution of n. If foo is sorted by n, initially the first predicate will be highly selective, but later it won't be selective at all

Synchronization barriers

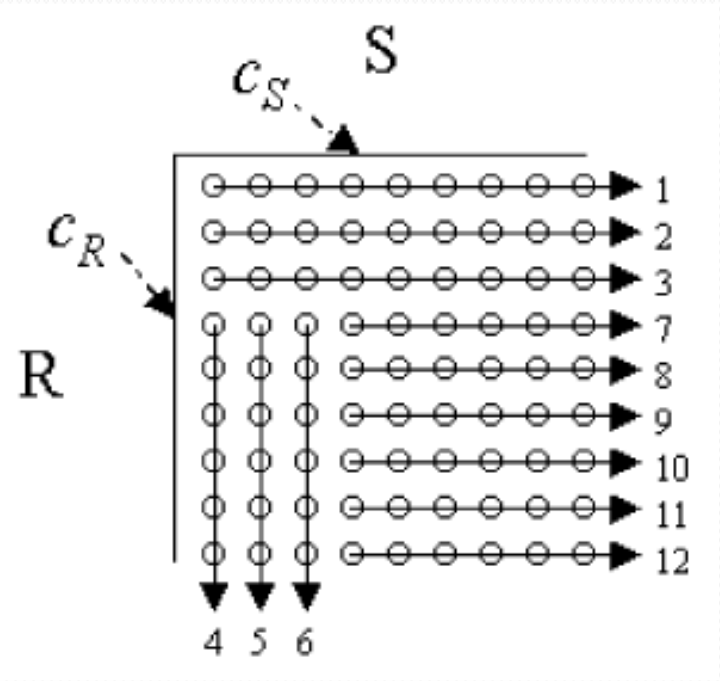
- We'd like to keep all operators busy all the time, but…
- Non-unary operators must wait for all operands
- Example: merge join (data already sorted)
- Table R returns data very slowly, and has many small values
- Table S returns data very quickly, but its values are mostly large
- Must wait for most of R before processing much of S

Moments of symmetry

- The common join operators are asymmetric
  - Nested-loops: all tuples of S for each tuple in R
  - Hash-join: all tuples of S before any tuple of R
- In both cases, reads of one side can proceed for some time before reaching a synchronization barrier
- At a synchronization barrier, it is possible to invert the asymmetric relation. This is a *moment of symmetry*.

Moments of symmetry example

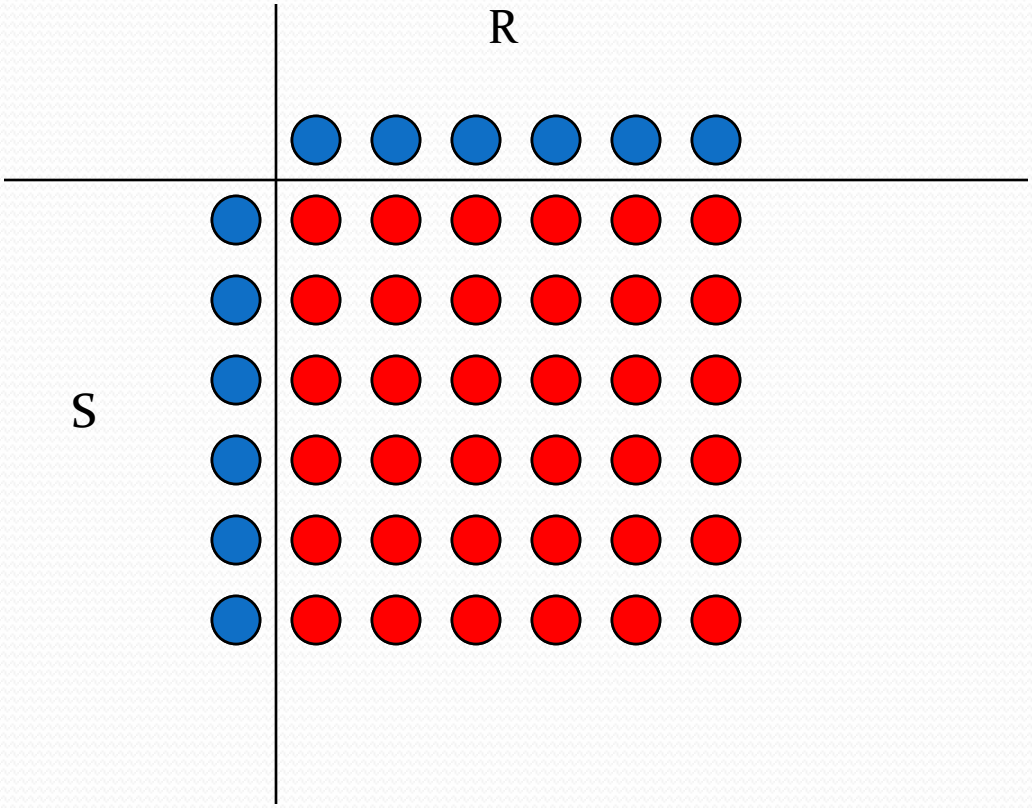
- Switching the inner and outer loops at moments of symmetry in a nested-loops join



Ripple join

- For maximum parallelism, we desire operators with frequent moments of symmetry: more freedom to process whatever operand is available
- The ripple join is a family of joins with very frequent moments of symmetry
- Not a pure stream operation: keeps the history of all tuples seen from either side (consumes much more memory)
- Double pipelined hash join is one example

Ripple join example



Eddies in operation

- N inputs, 1 output
- Each input tuple has a *ready* bit and a *done* bit for each operator in the eddy, representing the dependency tree
- After processing a tuple, an operator sets its *done* bit and returns it to the eddy. All done: output!
- As long as these dependencies are maintained, an eddy can *route* tuples freely to maximize throughput

Basic routing: backpressure

- Tuples buffered in priority queue
- Tuples arrive in low priority
- After any processing they are given high priority: forces a tuple all the way through an eddy ASAP
- Automatically handles variance in arrival rate and operator cost: slower operators spend more time processing result from fast operators than consuming new tuples

Routing: lottery scheduling

- Prioritizes more selective operators
- The ratio of tuples received to tuples produced is used as a probability of receiving the next tuple first, when multiple operators are available for the same tuple (e.g., multiple predicates)
- This does not account for selectivity changing over time
  - e.g., WHERE x LIKE ‘m%’ operating on sorted table
- Eddies use a simple window to handle this

Discussion

- Their general philosophy is: “we favor adaptability over best-case performance” Does this seem reasonable? In this case? In general: How does this compare with previous approaches that we’ve looked at?

Discussion

- Compare the lottery system here to the bidding in Mariposa. How is it similar? How is it different? Which would you rather use? Does it depend on the situation?

Evaluation: operator cost

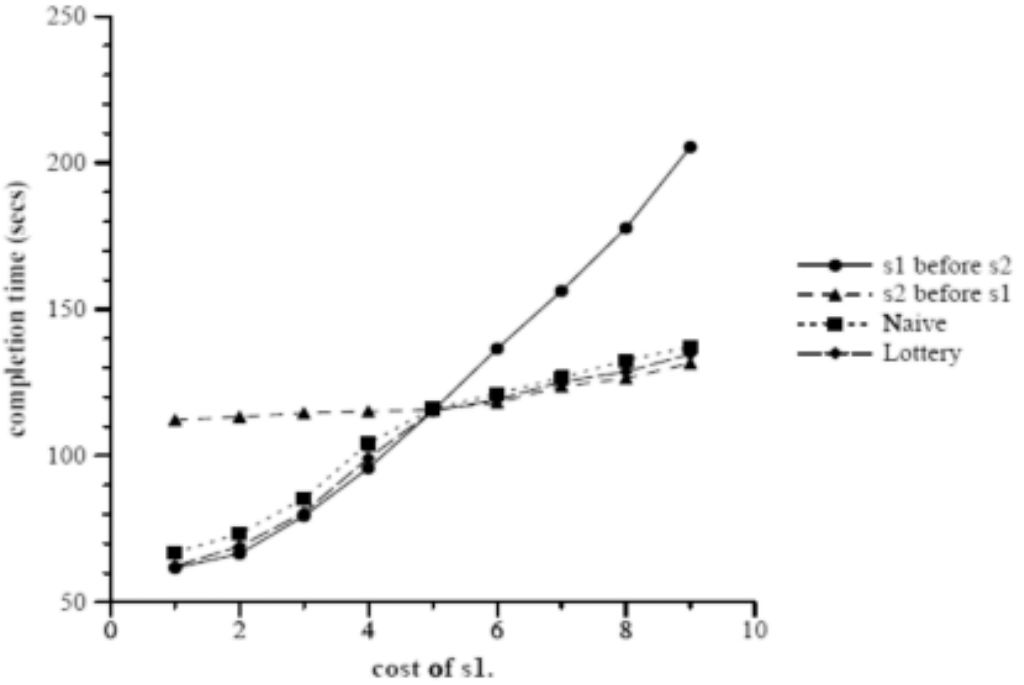


Figure 4: Performance of two 50% selections, $s2$ has cost 5, $s1$ varies across runs.

Evaluation: selectivity variance

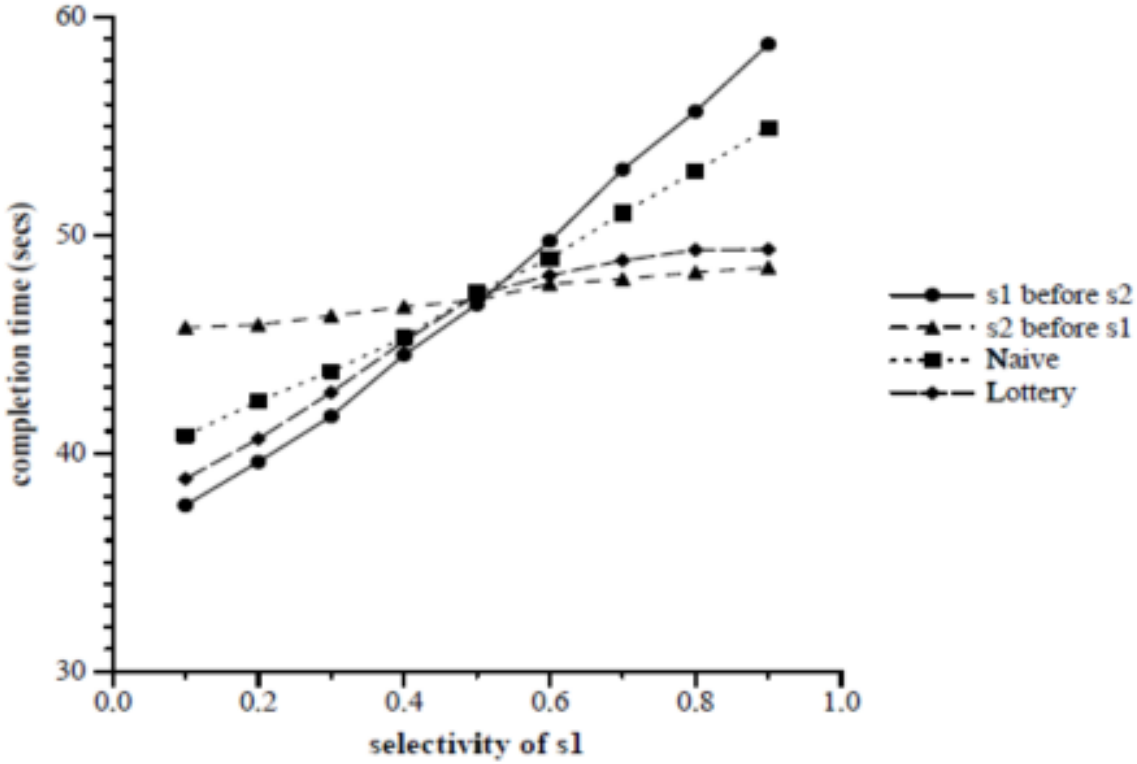


Figure 5: Performance of two selections of cost 5, $s2$ has 50% selectivity, $s1$ varies across runs.

Discussion

- Which would you rather use: Tukwila or Eddies? Why?