CS262, Spring 2007 (Hellerstein/Brewer): Parallel DBs and MapReduce

- **Quick Primer on the Relational Algebra**
  - selection
  - projection
  - cartesian product
  - union
  - set difference
  - join

- **Iterators and Other Dataflow Architectures**
  - An OO view of iterators
    - open(): initialize state
    - next(): produce a single record
    - close(): clean up state
    - inputs[]: a set of input iterators
  - The output of next() is well typed. In SQL it's a tuple with some schema.
  - Iterators are "piped together" by parameterizing parents with an array of children. In relational DBs usually 1 or 2 children.
  - Parents' methods invoke children's methods. Usually parent.open() invokes child.open() and sometimes child.next(). Usually parent.next() invokes child.next() or nothing. Usually parent.close() invokes child.close().
  - Typically, iterators in a single-site query processor make synchronous calls to their children. This is a "pull" model, like sucking data through a straw. **This couples dataflow and control flow** among iterators. Problems? Alternatives?
    - Thread per operator. Problems? Solution?
    - MapReduce's approach?
  - P2 solution: Push/Pull with boundary conditions
    - Empty + Pull: puller posts a callback/continuation
    - Full + "push": accept push, post a callback
  - PIER solution: Async Iterators
    - Control flow is pull
    - Dataflow is push
  - Fjords (TelegraphCQ)
    - Iterators can yield to caller without returning data
    - Push and Pull queue iterators
  - Bottom line:
    - Rendezvous in Space and Time
      - Level of indirection in both.
      - Where does the message go? Call stack vs. queue
      - Who arrives first? ("push vs. pull")
      - Coupling of control-flow and dataflow?
        - If coupled, rate mismatches should not happen
        - Can be decoupled by threads, or by some event scheduler
          - Threads wait on queues
          - Events and the role of callbacks/continuations in scheduling them.
Iterators and Other Dataflow Architectures

Typically, iterators in a single-site query processor make synchronous calls to their children. This is a "pull" model, like sucking data through a straw. This couples dataflow and control flow among iterators. Problems? Alternatives?

Bottom line: Rendezvous in Space and Time

How are rate mismatches buffered?
- Trivial with infinite buffers
- In pull, the caller allocates/reserve result space
- In push, the caller must find space

The Exchange Operator: Parallelizing Iterators

- Annoyance: "parallelizing" a query executor is a hassle.
- Solution: encapsulate the parallelism in a query operator, not in the QP infrastructure.
  - another classic level-of-indirection scheme, this time for dataflows.
- We want to enable partition and pipelined parallel dataflows over iterators -- including setup, teardown, and runtime logic -- in a clean encapsulated way.
- The exchange operator: an operator you pop into any single-site dataflow graph as desired -- anonymous to the other operators.

Implementation:
- Note: Volcano was done with processes, but today you'd use threads
  - splits the graph into two threads. The lower thread has an X-OUT iterator at the top. The upper thread has an X-IN iterator at the bottom.
  - X-OUT is a driver for the lower iterator. Says next() a bunch of times, constructs a packet, pushes that packet via IPC (or network comm) onto a queue in X-IN's "port". X-IN responds to next() when it has tuples in its queue.
  - Why is push beneficial?
  - Flow control: semaphore on the port dictates the maximum degree to which the producer can get ahead of the consumer.

Benefits of exchange:
- oOpaquely handles setup and teardown of clones (in an SMP...for shared-nothing, would need to have daemons at each site, and a protocol to request clone spawning)
- at the top of a local subplan, allows pipeline parallelism: turns iterator-based, unithreaded "pull" into network-based, cross-thread "push".
- at the top of a local subplan, allows decoupling of children's scheduling.
- inside a subplan, can mix pull and push to get the best of both

"Extensibility" features of Volcano and exchange:
- operators don't interpret records, support functions do; goes for partitioning as well
- basically just a level of indirection for anything data-dependent
- Volcano had a much more aggressive extensibility story for query optimization
  - rule-based query optimizer generator -- evolved from Graefe's thesis work on the Exodus optimizer generator. Later work on the Cascades optimizer evolved this further
  - contrasts with Starburst's rule-based optimizer, which is more like an extensible version of the System R optimizer (more standard).
  - if you're interested in using query optimization ideas for building up dataflows of arbitrary operators (not just relational operators), you should know this material

Eddies

- Modern Motivation: poor selectivity estimation. performance fluctuation is minor compared to this.
  - Lots of interest in this kind of adaptive query processing in the traditional DB shops
  - Also of course on the inter-query timescales
- The 2-selection example
  - rank ordering intuition
The eddy operator. Simultaneously encapsulates
- monitoring of all operator inputs/outputs
- decisions on orderings of operators
- decisions on orderings of tuples (though not in this paper)
- Cost: marking tuples with ready/done bits

- fix selectivities, vary costs.
  - detail: priority to previously-seen tuples
  - queue backpressure prevents slow operators from consuming fresh tuples before fast operators

- fix costs, vary selectivities
  - backpressure is not enough. Need to account for selectivities.
  - Joint metric: lottery tickets for discarding inputs, accounts for input rate (cost) and output rare
    (function of cost and selectivity)

- Need a way to forget the past.

Joins
- This story gets much more interesting in later work: SteMs and STAIRS
  - SteMs (Raman, ICDE ’03): unary operators, not binary joins. Gives join algorithm flexibility,
    spanning tree flexibility
  - STAIRS (Deshpande, VLDB ’04): Lifting the burden of state history!

- Lots of theory work since then
  - Mostly on selections (the "Pipelined Filter" problem)
  - Optimization goal: distributional (expected case)? adversarial (worst case)?
  - Correlation makes life tricky!
  - But see static "conditional plans" for ideas, Deshpande ICDE ’05

- Lots of room for ML techniques here
  - This is a reinforcement learning problem
  - Subject of previous class projects, but never pursued to completion

- Lots of room for practical issues
  - Good start on this in Postgres (Amol Deshpande)
  - Again, never really pursued to completion

- Industry would LOVE us to make this less radical, more practical
  - There is a host of work on less aggressive adaptive query processing techniques
  - See Foundations and Trends in Databases 1(1)