Building Adaptivity into Execution

Zachary G. Ives

University of Pennsylvania

April 16, 2003

Data Integration Systems

- First generation: mostly concerned with query translation, data translation
  - TSIMMIS, Information Manifold, SIMS, many others
  - Automatically inferring wrappers for sources
  - Mostly prototypes for integrating web data
- Assumption: this was the “hard part” and the rest of the system would leverage conventional/distributed DB technology

It’s Not as Easy as It Sounds…

- How do we optimize a query here?
  - Conventional DBs: we control all, and we have stats on the tables
  - Distributed DBs: we control almost all, and we have stats on the tables
- What if someone else controls all of the data?
  - Statistics – how do you get them? Will they be up to date?
  - Costs – what about network congestion?
  - Reliability – we want maximal answers if a source fails
  - … And what if some of the sources might be large?
- Also: want to give answers as early as possible

The Tukwila System

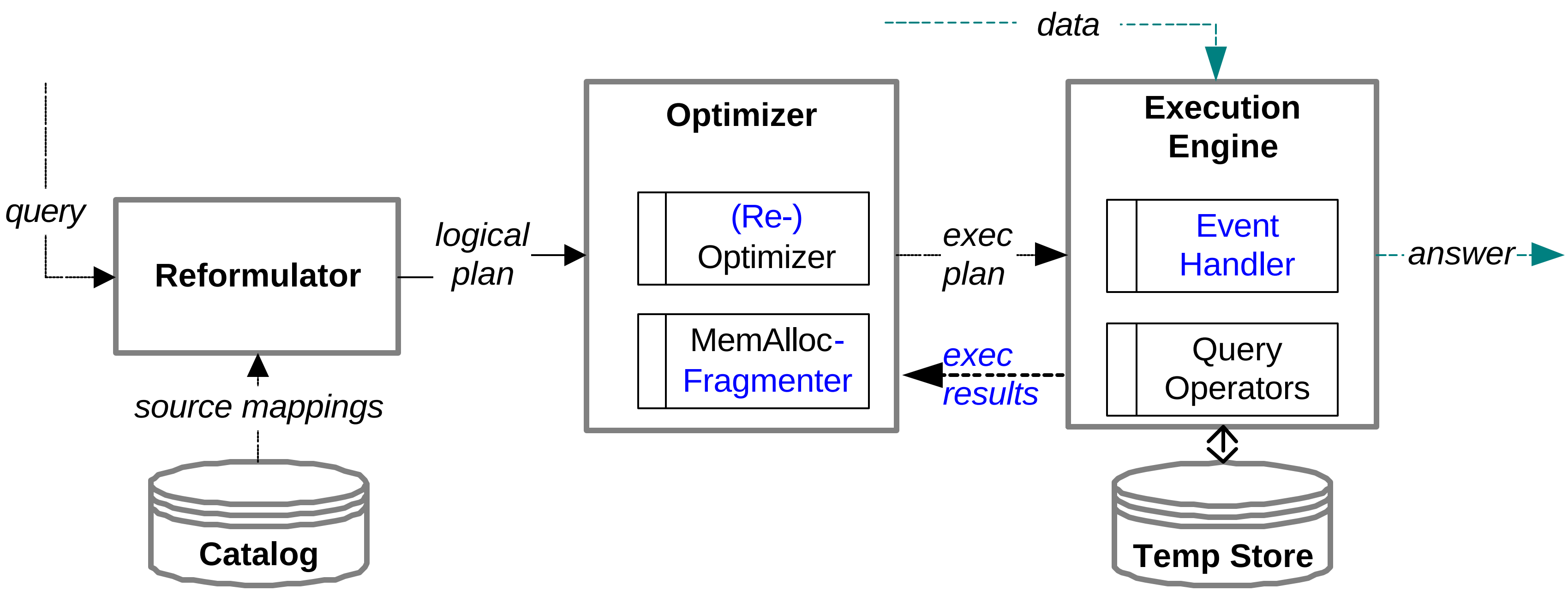
Image: Nat'l Wildlife Federation

- "Child of the Information Manifold"
  - Sources are described as queries over mediated schema ("local as view")
    - Successor to the Bucket Algorithm: MiniCon [Pottinger & Levy] (we'll discuss later)
  - Support for input bindings, etc.
- But focused on building scalable system:
  - Normal DB techniques for optimization and execution don't work well – how do we fix that?
  - Between 1999-2002:
    - Added support for XML in a novel way (we'll discuss this 3/3)
    - Tried to remedy the shortcomings of our initial approach

Novelties of Tukwila (in this Paper)

- Premise:
  - We start with little knowledge about data, sources, performance
  - Bad idea to stick with one plan or one scheduling!
- Solution: Build a “smarter” and more flexible runtime system!
  1. Rule-based core: optimizer can specify behaviors when events occur
  2. Integrate mid-query re-optimization at the core of execution and optimization
  3. Resurrect the pipelined hash join (invented for parallel DBs), but invent ways to handle memory constraints

Tukwila Architecture



Event-Based Control

- *Event-condition-action rules* allow optimizer to define changes in behavior at middle of pipeline
- Execution *events …*

  Timeout, *n* tuples read, operator opens, out of memory, execution step completes, …
- … trigger the rules
  - Test *conditions*

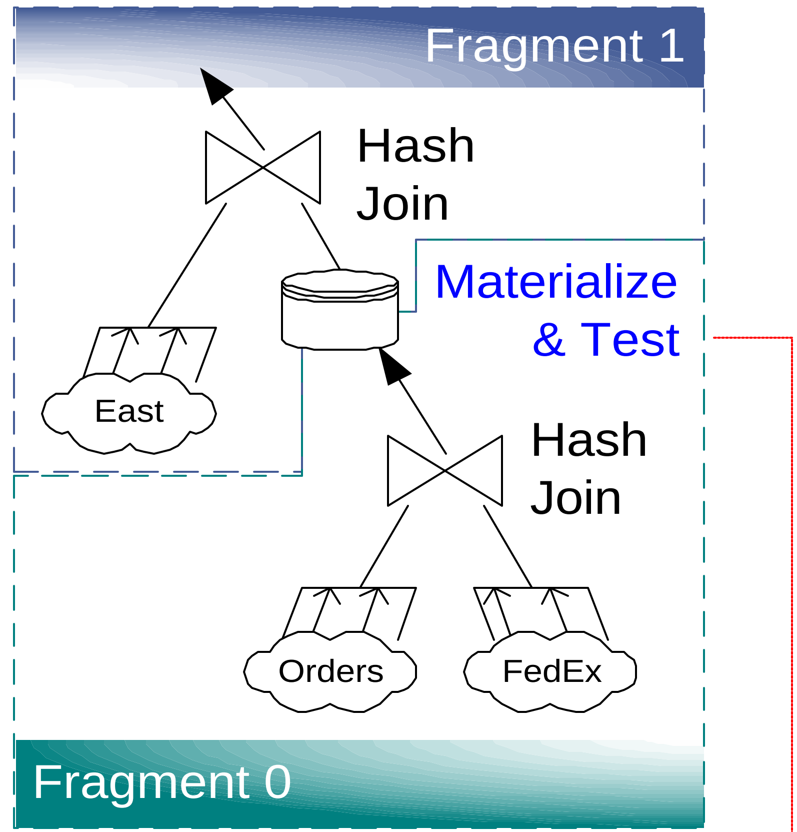
    Memory free, tuples read, operator state, …
  - Execute *actions*

    Re-optimize, reduce memory, activate operator, …

Interleaving Planning and Execution

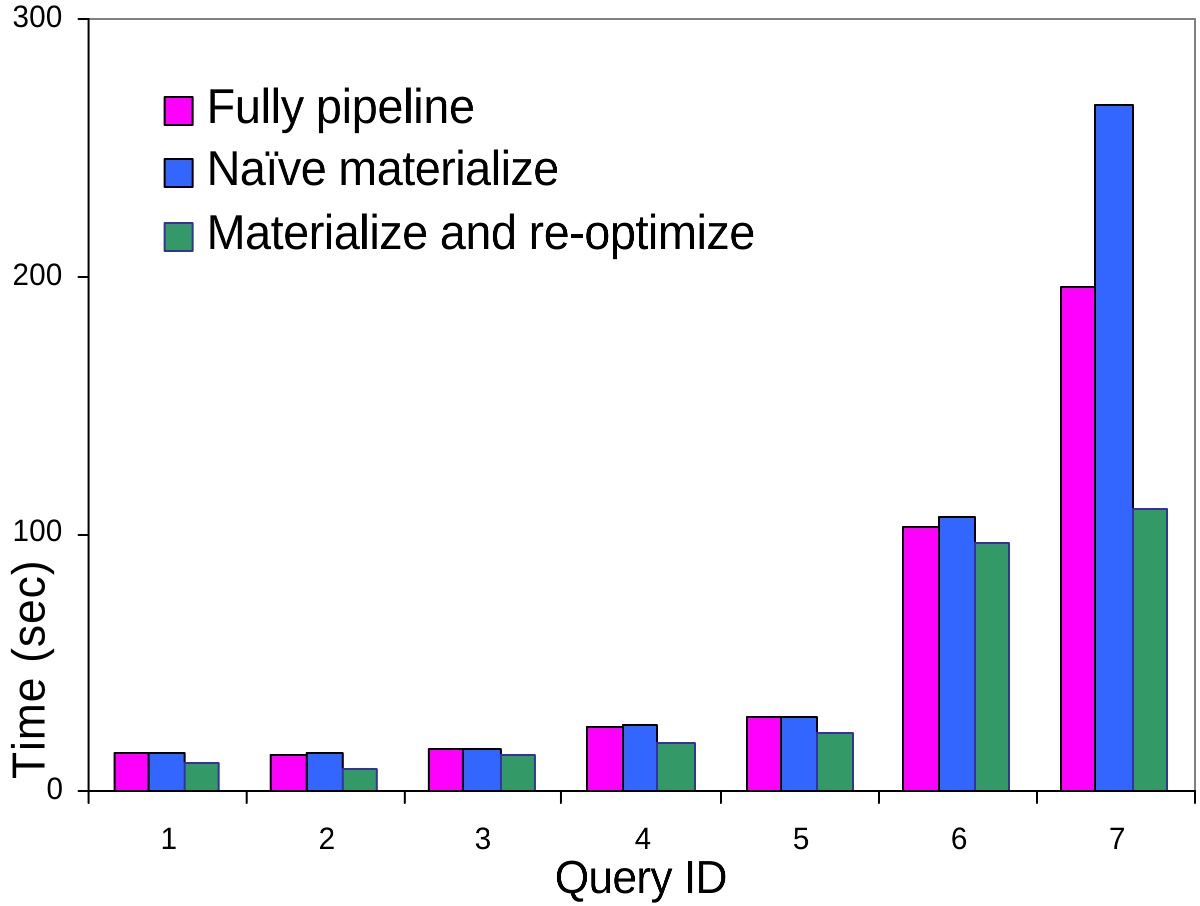
Generalization of [Kabra/DeWitt SIGMOD98] integrated into system

- Check at key points
- Plan in pipelined *fragments*
- Rules at boundaries test conditions
- Return simple statistics to optimizer
  - Optimizer does minimal re-computation of costs



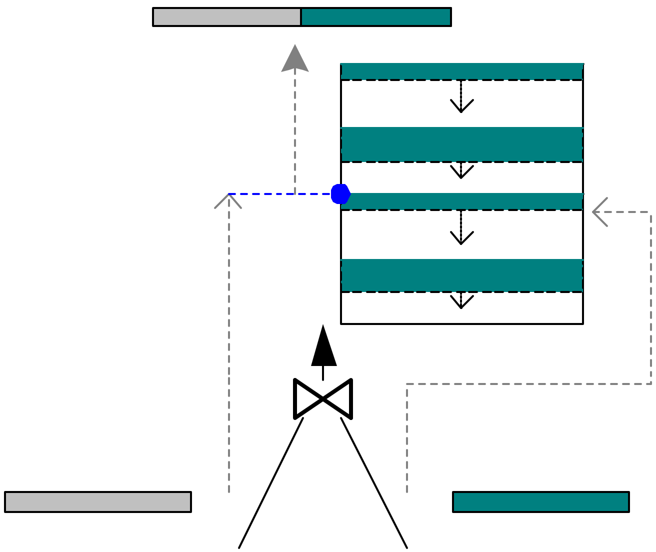
WHEN end_of_fragment(0)
IF card(result) > 100,000
THEN re-optimize

Experimental Results: Interleaving Planning and Execution



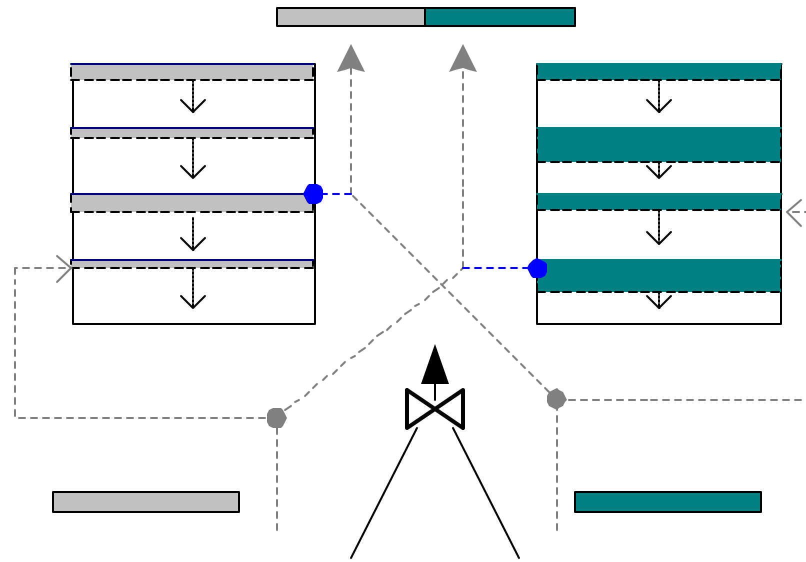
Four-table joins from scaled TPC-D

Adaptive Operators: Double Pipelined Join



Hybrid Hash Join

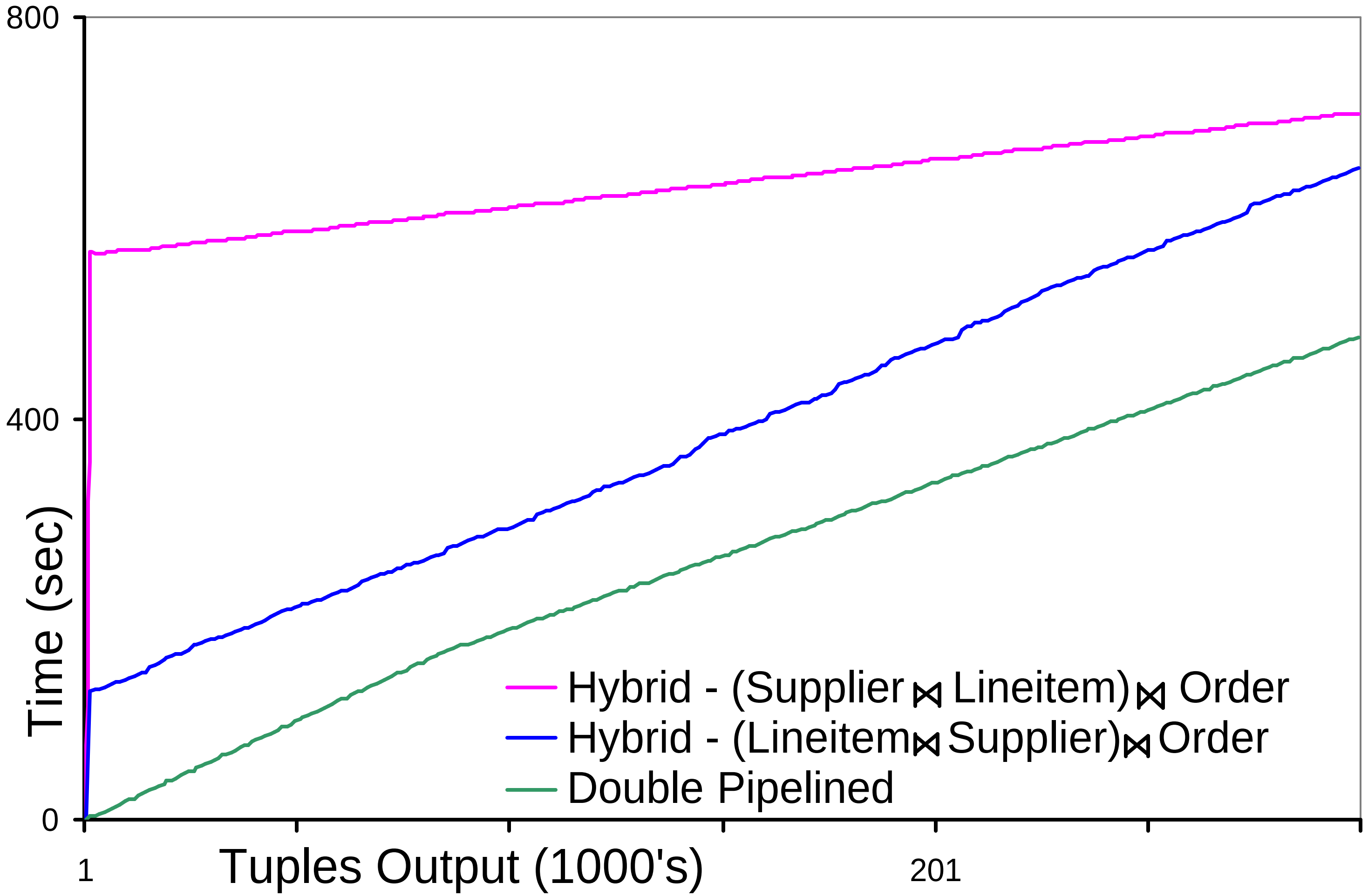
- ✗ No output until hash built
- ✗ Asymmetric (build vs. probe) (why is this bad?)



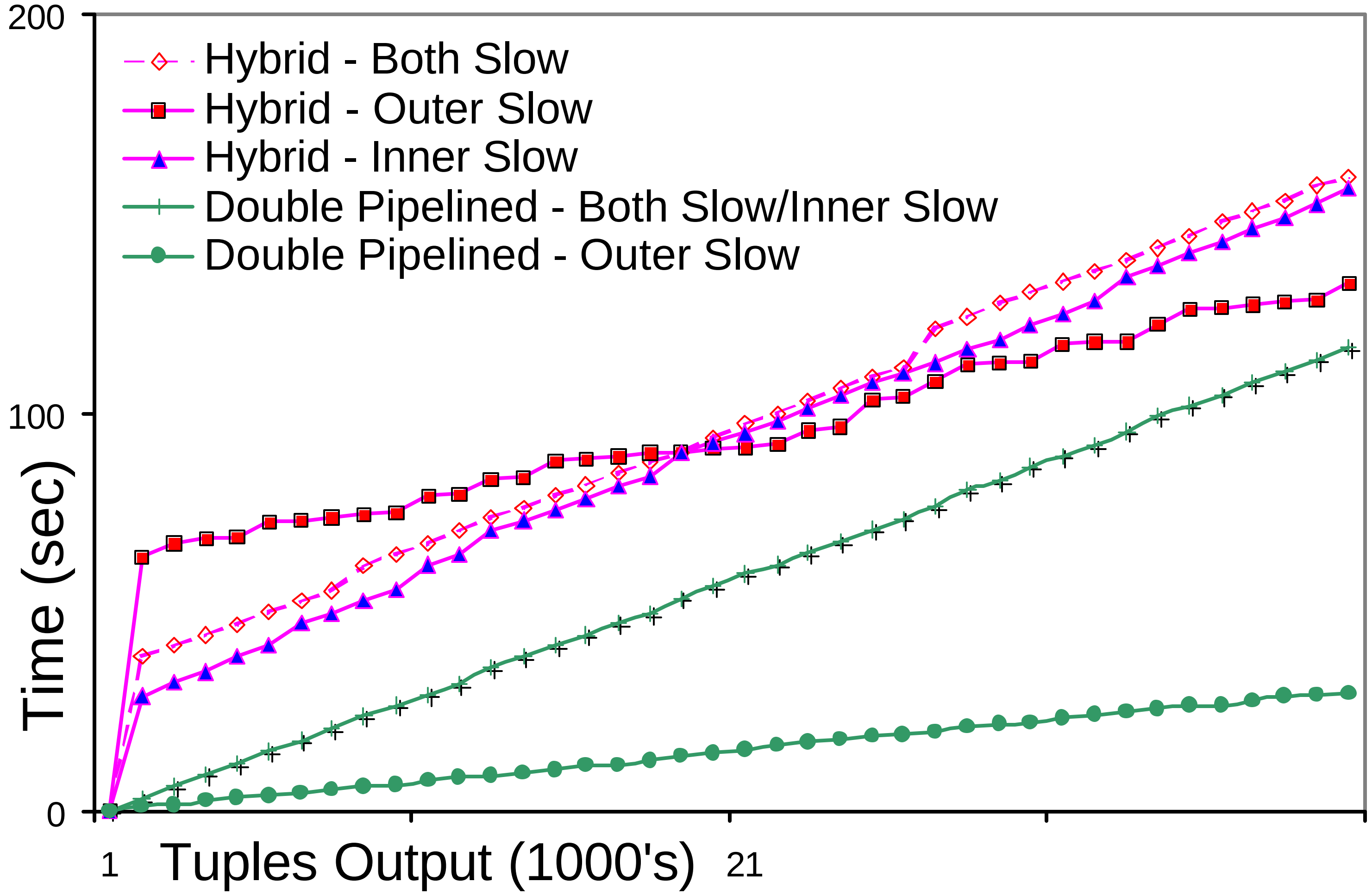
Pipelined Hash Join

- ✓ Outputs data immediately
- ✓ Symmetric (why is this good?)
- ✗ More memory

Double Pipelined and Hash Join—Tuples Output vs. Time - LAN



Double Pipelined Join - Wide Area/Internet



Problem: Memory Usage

- We need two hash tables in memory…

- Recall how a hybrid hash join works:
  - Load build relation until we run out of memory
  - Repeat until we’ve read the build relation:
    - Select a few buckets, page them out
    - Read some more data
  - Load data from the probe relation:
    - If it hashes to a bucket that’s in memory, probe & join
    - Else page to tempfile
    - After probe relation consumed, join tempfile with swapped buckets

Handling Overflow

Extend principles of hybrid hash algorithm:

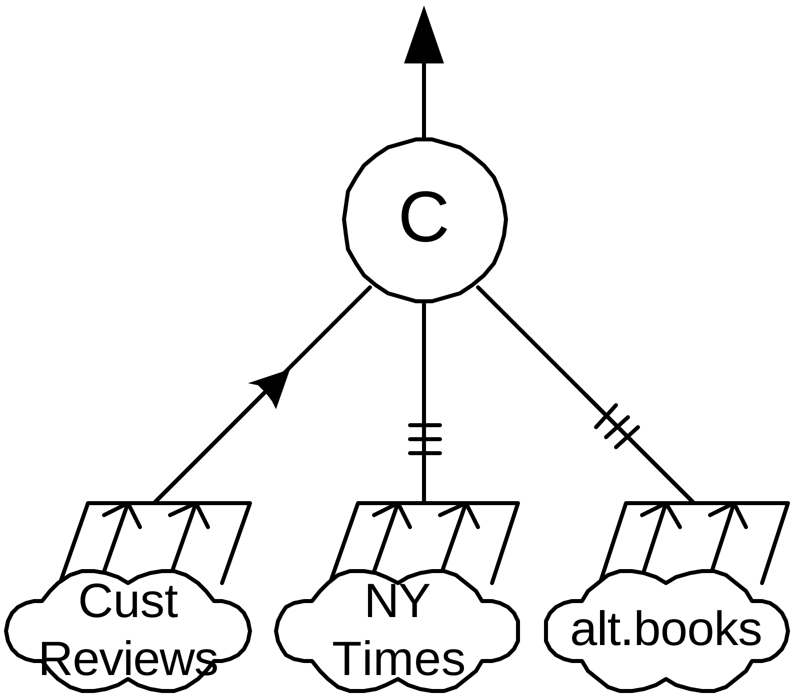
- *Incremental left flush* – degrade into hybrid hash
  - Pause pipelining left, flush some of its hash table
  - Read remainder of right, pipeline left as in HHJ
  - ➢ Abrupt pause, then steady output of tuples
- *Symmetric flush* – lose some “coverage”
  - Flush same hash bucket in both tables simultaneously, continue to fully pipeline
  - ➢ Output production tapers off as more flushes

➢ Expensive, but get first tuples faster than otherwise!

Adaptive Operators: Collector

Utilize mirrors and overlapping sources to produce results quickly

- Dynamically adjust to source speed & availability
- Scale to many sources without exceeding net bandwidth
- *Policy* expressed via rules



WHEN timeout(CustReviews)
DO activate(NYTimes),
activate(alt.books)

Brief Retrospective on this Paper

1. Rule-based core:
   - Nicely unifies adaptive behaviors, supports custom responses to events
   - But hard to generate rules, except for basic ones
2. Integrated mid-query re-optimization
   - … Let's defer this to last!
3. Pipelined hash join with overflow handling
   - (Simultaneously resurrected by Urhan & Franklin)
   - A success: everyone doing distributed querying uses this technique now

Mid-Query Re-optimization in a Data Integration Context

- Benefits:
  - Can keep us from going too far down the wrong path if we have huge intermediate results
- Drawbacks:
  - How do we decide where to break the pipelines, given that we don't know how big anything is?
  - May quickly find that we're running a bad plan – no way to change until we finish the 1st pipeline
  - What about early initial answers?

- Can you think of some alternatives…?