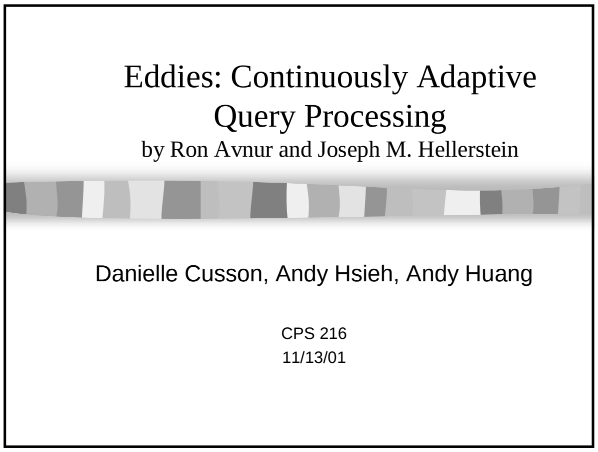
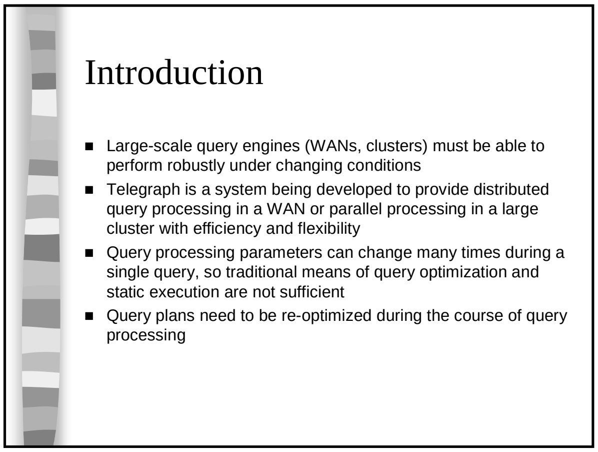


Eddies: Continuously Adaptive Query Processing

by Ron Avnur and Joseph M. Hellerstein

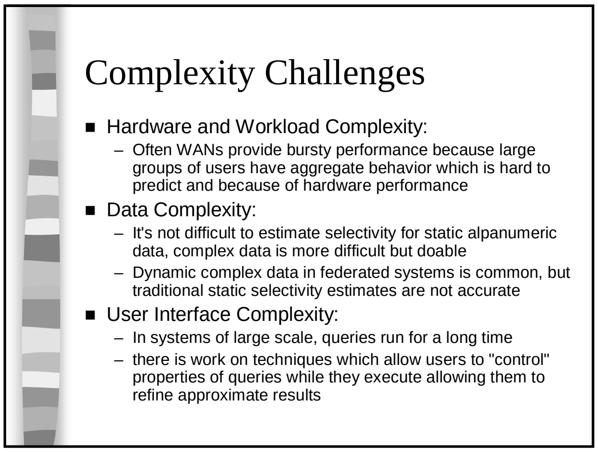
Danielle Cusson, Andy Hsieh, Andy Huang

CPS 216
11/13/01



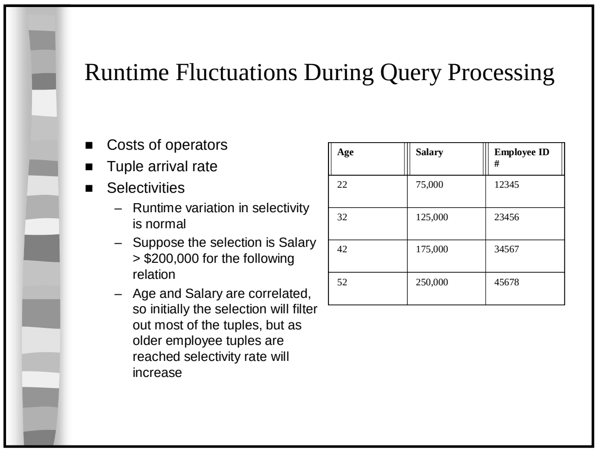
Introduction

- Large-scale query engines (WANs, clusters) must be able to perform robustly under changing conditions
- Telegraph is a system being developed to provide distributed query processing in a WAN or parallel processing in a large cluster with efficiency and flexibility
- Query processing parameters can change many times during a single query, so traditional means of query optimization and static execution are not sufficient
- Query plans need to be re-optimized during the course of query processing



Complexity Challenges

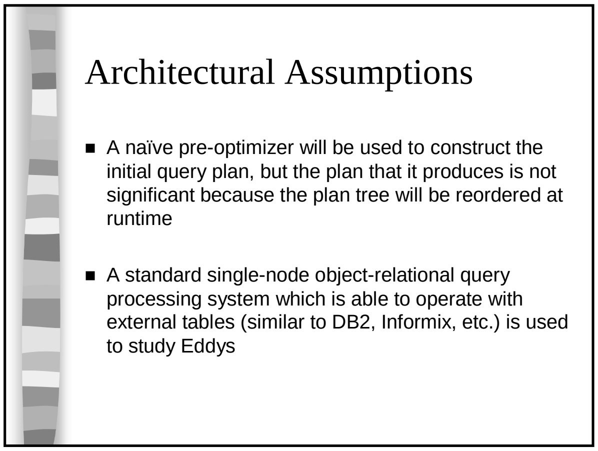
- Hardware and Workload Complexity:
  - Often WANs provide bursty performance because large groups of users have aggregate behavior which is hard to predict and because of hardware performance
- Data Complexity:
  - It's not difficult to estimate selectivity for static alpanumeric data, complex data is more difficult but doable
  - Dynamic complex data in federated systems is common, but traditional static selectivity estimates are not accurate
- User Interface Complexity:
  - In systems of large scale, queries run for a long time
  - there is work on techniques which allow users to "control" properties of queries while they execute allowing them to refine approximate results



Runtime Fluctuations During Query Processing

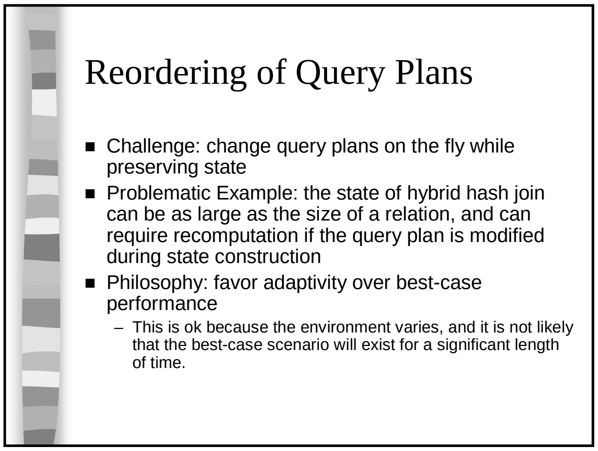
- Costs of operators
- Tuple arrival rate
- Selectivities
  - Runtime variation in selectivity is normal
  - Suppose the selection is Salary > $200,000 for the following relation
  - Age and Salary are correlated, so initially the selection will filter out most of the tuples, but as older employee tuples are reached selectivity rate will increase

Age	Salary	Employee ID #
22	75,000	12345
32	125,000	23456
42	175,000	34567
52	250,000	45678



Architectural Assumptions

- A naïve pre-optimizer will be used to construct the initial query plan, but the plan that it produces is not significant because the plan tree will be reordered at runtime
- A standard single-node object-relational query processing system which is able to operate with external tables (similar to DB2, Informix, etc.) is used to study Eddys



Reordering of Query Plans

- Challenge: change query plans on the fly while preserving state
- Problematic Example: the state of hybrid hash join can be as large as the size of a relation, and can require recomputation if the query plan is modified during state construction
- Philosophy: favor adaptivity over best-case performance
  - This is ok because the environment varies, and it is not likely that the best-case scenario will exist for a significant length of time.

Synchronization Barriers

- A synchronization barrier occurs when one table-scan must wait until another table-scan produces a required value
- Example: Merge Join on sorted, duplicate-free inputs
  - Each tuple consumed is taken from the relation whose last tuple had the lower value.
  - SlowLow, FastHi relations
- Barriers limit concurrency
- Overhead is caused by both frequency of barriers and the differences in arrival times of two inputs at a barrier

Moments of Symmetry

- Moment of Symmetry: When join algorithms reach synchronization barriers, scheduling dependency between the two input relations ends and often the order of the inputs can be changed without disturbing the state of the join
- Merge Join: MoS at any time, symmetric
- Nested Loops Join: MoS at the end of each inner loop, asymmetric
- Hybrid Hash Join: no MoS

Commutativity and Reordering Issues

- Natural joins commute, therefore a tree of n-1 binary joins can be considered a single n-ary join
- The binary joins are not required to be the same join algorithm, although they must commute
- Certain cases such as index nested-loops join or algorithms which only work for certain types of joins place constraints on the reordering of the query plan, in this work these constraints are adhered to

Join Algorithms for Dynamic Reordering

- The best join algorithms for dynamic reordering provide:
  - Frequent Moments of Symmetry
  - Adaptive or non-existent barriers
  - Minimal ordering constraints
- Ripple Join family provides best adaptivity:
  - Pipelined Hash Join (aka hash ripple, Xjoin)
    - No synchronization barriers
    - Continuous symmetry
    - Good for equijoin
  - Simple (or Block) Ripple Join
    - Synchronization barriers at corners
    - Moments of Symmetry at corners
    - Good for non-equijoin

River

- A dataflow, multithreaded query engine, analogous to many commercial parallel database engines
- Contain module (query operators) communicate via a fixed dataflow graph (query plan)
- Each module runs as independent thread
- Can exploit barrier-free algorithms by reading from various inputs at independent rates

Pre-Optimization

- Initial optimizer to construct spanning tree and algorithms used in each join
- Later will use eddies to reorder tables among joins

Eddy in River

- Implemented via a module in a river
- Arbitrary number of input relations, a number of unary and binary modules, and a single output relation
- Merge multiple unary and binary operators into a single n-ary operator
- Fixed-sized buffer of tuples to be processed by one or more operators
- Each operator takes one or two inputs
- Output stream returns tuples to eddy

Ready and Done Bits

- Purpose: preserve ordering constraints while maximizing opportunities for tuples to follow different possible ordering of operators
- Ready bits - indicate tuples are eligible to be operated
- Eddy will only pass a tuple to operator if Ready bit is on
- Done bits - tuple has been processed
- If all Done bits are on, the tuple is sent to eddy's output, otherwise send tuple to another operator

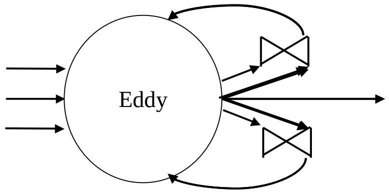
Eddy Advantages

- Flexible in shapes of trees: represents the full class of bushy trees corresponding to the set of join nodes
- Flexible in reordering operators: no requirement that all operators in the eddy be at moment of symmetry except the operator fetching new tuples

Eddy Routing

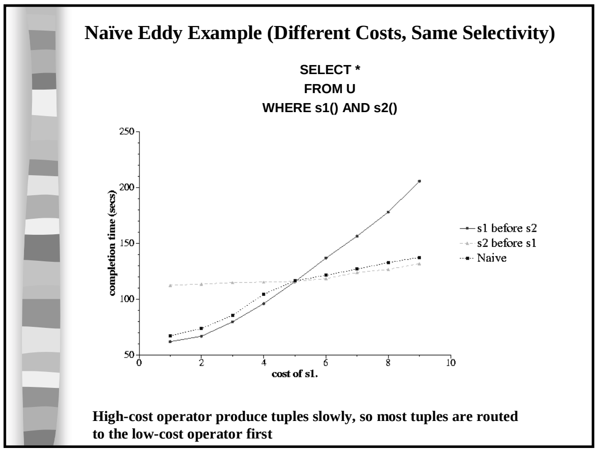
- Directs the flow of tuples individually through various operators to the output
- Priority queue as tuple buffer
- High priority to tuples having Ready bits set

Eddy Routing (cont’d)



Naïve Eddy

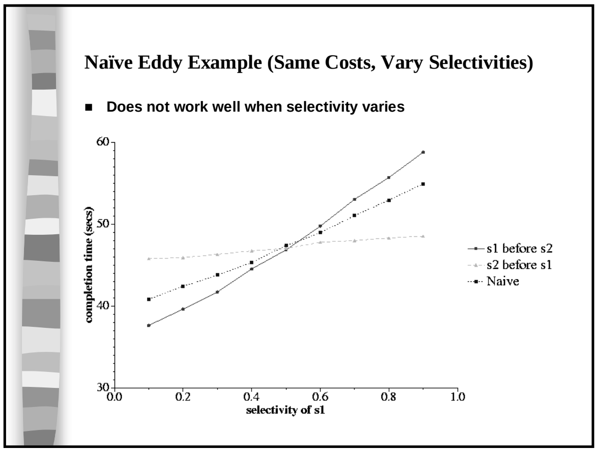
- Works well for handling operators with different costs but same selectivities



Naïve Eddy Example (Different Costs, Same Selectivity)

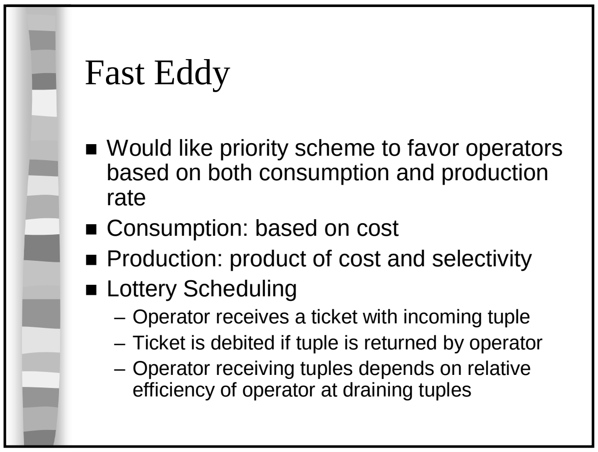
SELECT *
FROM U
WHERE s1() AND s2()

High-cost operator produce tuples slowly, so most tuples are routed to the low-cost operator first



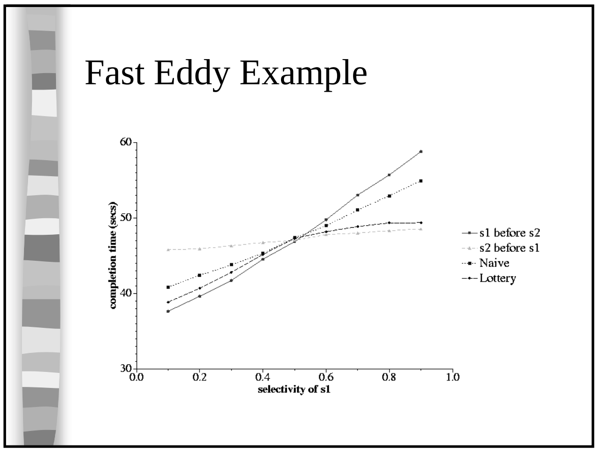
Naïve Eddy Example (Same Costs, Vary Selectivities)

- Does not work well when selectivity varies

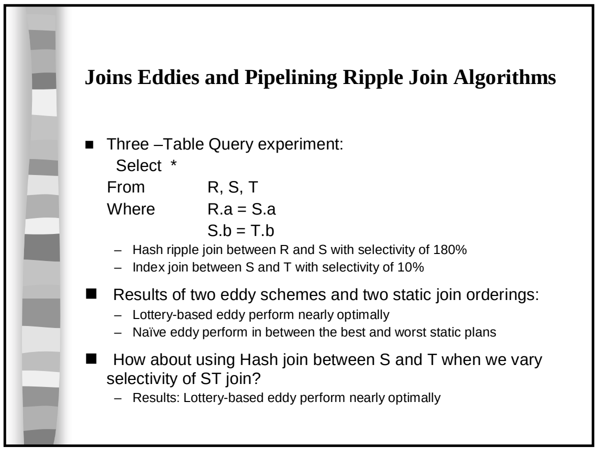


Fast Eddy

- Would like priority scheme to favor operators based on both consumption and production rate
- Consumption: based on cost
- Production: product of cost and selectivity
- Lottery Scheduling
  - Operator receives a ticket with incoming tuple
  - Ticket is debited if tuple is returned by operator
  - Operator receiving tuples depends on relative efficiency of operator at draining tuples

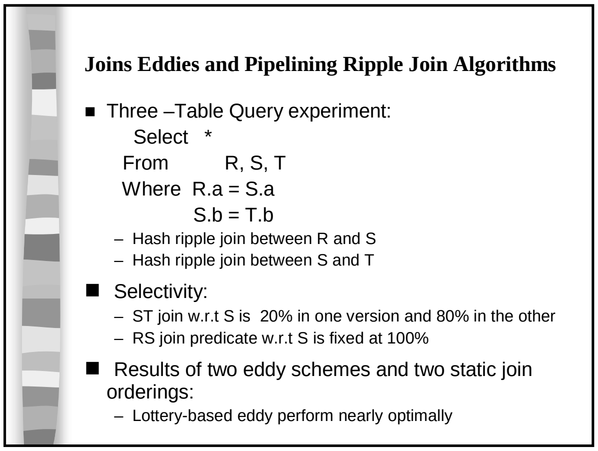


Fast Eddy Example



Joins Eddies and Pipelining Ripple Join Algorithms

- Three –Table Query experiment:
  Select *
  From R, S, T
  Where R.a = S.a
  S.b = T.b
  - Hash ripple join between R and S with selectivity of 180%
  - Index join between S and T with selectivity of 10%
- Results of two eddy schemes and two static join orderings:
  - Lottery-based eddy perform nearly optimally
  - Naïve eddy perform in between the best and worst static plans
- How about using Hash join between S and T when we vary selectivity of ST join?
  - Results: Lottery-based eddy perform nearly optimally



Joins Eddies and Pipelining Ripple Join Algorithms

- Three –Table Query experiment:
  Select *
  From R, S, T
  Where R.a = S.a
  S.b = T.b
  - Hash ripple join between R and S
  - Hash ripple join between S and T
- Selectivity:
  - ST join w.r.t S is 20% in one version and 80% in the other
  - RS join predicate w.r.t S is fixed at 100%
- Results of two eddy schemes and two static join orderings:
  - Lottery-based eddy perform nearly optimally

Dynamic Fluctuation

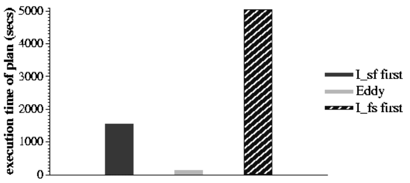
- Problem with static performance environment
  - Lottery scheme weights all experiences equally
  - Observation from distant past affect the lottery as much as recent observation
- Fix?
  - Window Scheme
  - Time is partitioned into windows
  - Banked tickets and escrow tickets

Dynamic Fluctuation

- 3-table equijoin query example setup
  - Two tables are external and used as “inner” relations
  - The third relation has 30,000 tuples
  - Two phases in experiment
    - Phase 1: One index (I_fs) is fast and I_sf is slow
    - Phase 2: Begin after 30 seconds and two index swap speed

Dynamic Fluctuation

- Two Index Join
  - Slow: 5 second delay
  - Fast: no delay
  - Toggle after 30 seconds



Dynamic Fluctuation

- What happen when we fix the costs and modify the selectivity?

  Given : two operators of cost C with selectivity swap between low to hi

  Either Static plan : nc + 1/2nc (n is number of tuples)

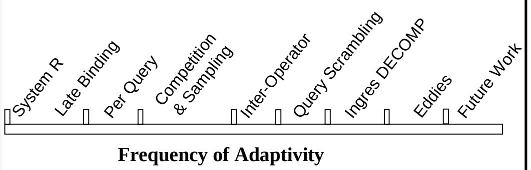
  Optimal dynamic plan: nc

  Ratio is: 3:2
- With more operators, adaptively to changes in selective can become more significant

Delayed Delivery

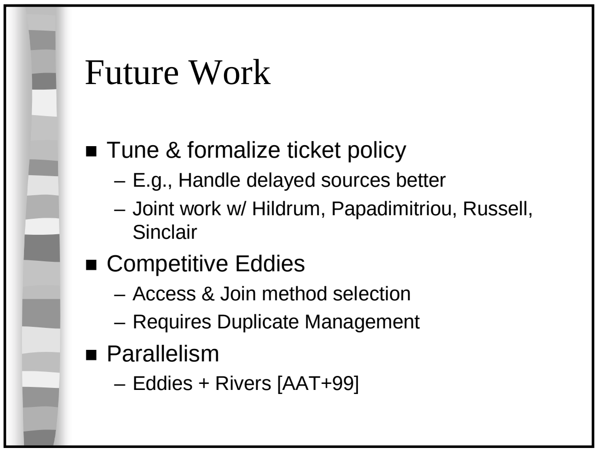
- What happen when we add an initial delay?
  - Eddy even with a lottery and a window-based forgetting scheme- does not adapt well as it could
- Observation:
  - Eddy incorrectly favors the RS join when no tuples are streaming in
- Reason:
  - Eddy observes that the RS join doe not produce any output tuples when given S tuples
  - It awards most S tuples to the RS join initially
  - ST join is left to fetch and hash T tuples
- Problem:
  - Wastes resources that could have been spent joining S tuples with T tuples during the delay

Related Work



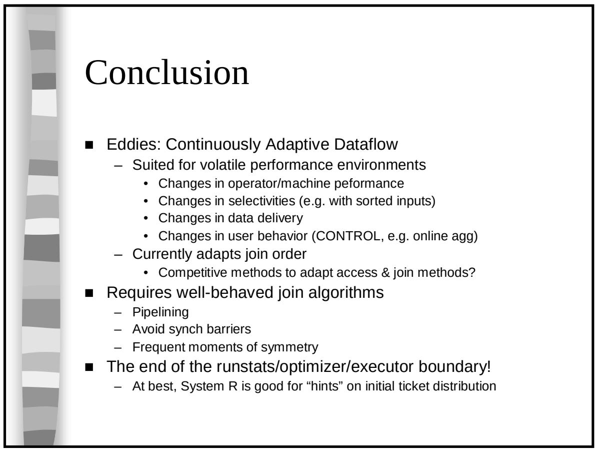
- Late Binding: *Dynamic, Parametric* [HP88,GW89,IN+92,GC94,AC+96,LP97]
- Per Query: *Mariposa* [SA+96], *ASE* [CR94]
- Competition: *RDB* [AZ96]
- Inter-Op: [KD98], *Tukwila* [IF+99]
- Query Scrambling: [AF+96,UFA98]

- Survey: Hellerstein, Franklin, et al., DE Bulletin 2000



Future Work

- Tune & formalize ticket policy
  - E.g., Handle delayed sources better
  - Joint work w/ Hildrum, Papadimitriou, Russell, Sinclair
- Competitive Eddies
  - Access & Join method selection
  - Requires Duplicate Management
- Parallelism
  - Eddies + Rivers [AAT+99]



Conclusion

- Eddies: Continuously Adaptive Dataflow
  - Suited for volatile performance environments
    - Changes in operator/machine peformance
    - Changes in selectivities (e.g. with sorted inputs)
    - Changes in data delivery
    - Changes in user behavior (CONTROL, e.g. online agg)
  - Currently adapts join order
    - Competitive methods to adapt access & join methods?
- Requires well-behaved join algorithms
  - Pipelining
  - Avoid synch barriers
  - Frequent moments of symmetry
- The end of the runstats/optimizer/executor boundary!
  - At best, System R is good for “hints” on initial ticket distribution