Query Processing for Data Integration – Be Adaptive !

Vũ Tuyết Trinh

1

Context

- Covered basic design-time techniques:
  - Modeling sources, schema mapping
  - We'll use this stuff a lot going forward.
- Next: the run-time of data integration:
  - query processing
- Then:
  - Other architectures for data integration
  - Dataspaces and other fancy stuff.

2

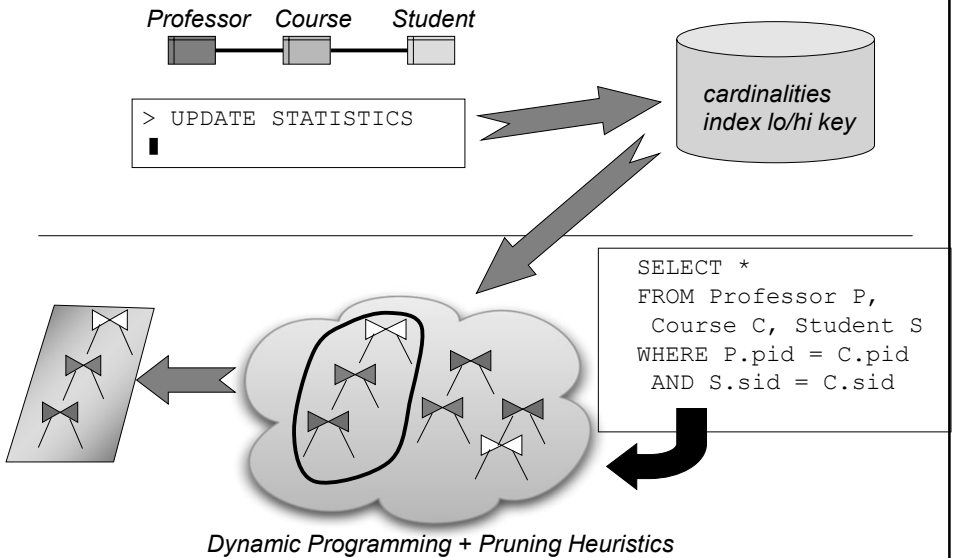
Some Issues

- Quick recap of traditional query processing
- Challenges in the context of data integration
- Techniques for adaptive query processing

3

Query Optimization and Processing

(As Established in System R [SAC+'79])

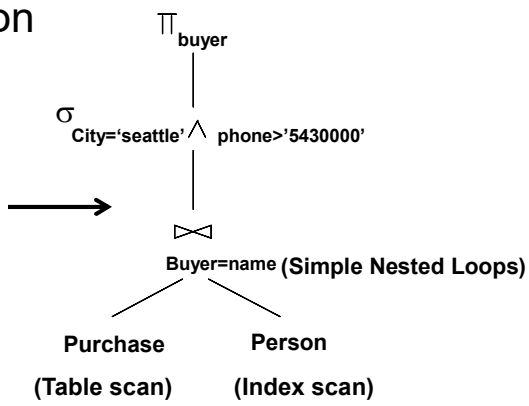


4

Traditional Query Processing

1. Query optimization

```
SELECT  S.sname
FROM Purchase P, Person Q
WHERE P.buyer=Q.name AND
      Q.city='seattle' AND
      Q.phone > '5430000'
```



2. Query execution

5

Meta-Data for Optimization

- Sizes of tables in the database
- Selectivity information:
  - How many rows per value in a column
  - How many rows per range
  - Cardinality of certain views
- Indexes on the data:
  - B-trees, hash indices, sorting order

6

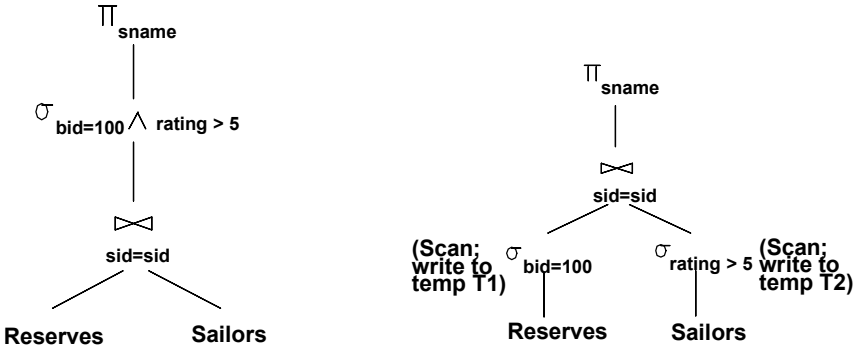
Query Optimization Steps

- Algebraic transformations
  - Can be very complex
- Select ordering of joins
  - Many possible orders. Can't explore them all
- Select algorithm for each node in the query plan
  - Many possible implementations of relational operators

7

Algebraic Transformation

Predicate Pushdown



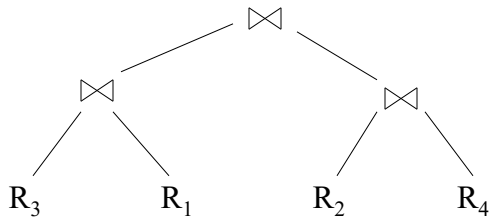
The earlier we process selections, less tuples we need to manipulate higher up in the tree.

Disadvantages?

8

Determining Join Ordering

- $R_1 \bowtie R_2 \bowtie \ldots \bowtie R_n$
- Join tree:

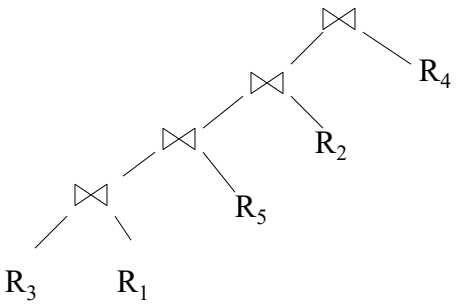


- A join tree represents a plan.

9

Left Deep Trees

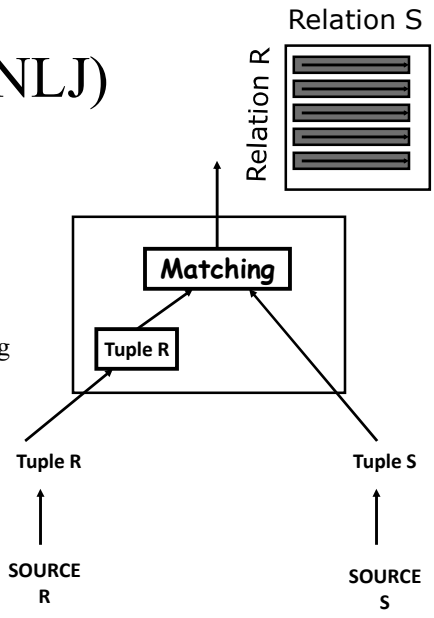
- Good for pipelining



10

Nested-loop-join (NLJ)

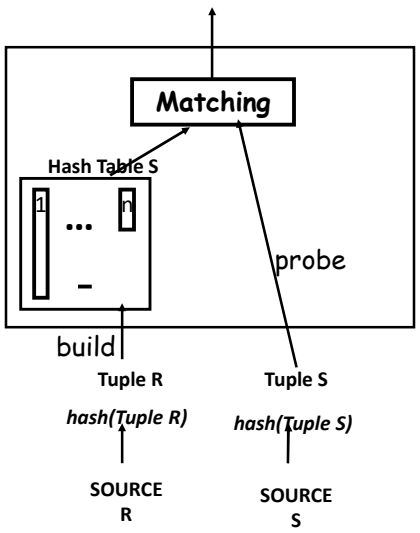
- Principle
  - reading the external relation R & iterating over the internal relation S
  - one-and-haft pass, non-blocking
- Variations
  - Tuple-based NLJ, block-based NLJ, index-based NLJ



11

Hash Join (HJ)

- Principle
  - using a hash table (HT) of R
  - two-pass, blocking algorithm
- Process
  - build the HT of R
  - probe every tuple of S with corresponding tuples in the HT of R



12

Cost of Hash-Join

- In partitioning phase, read+write both relations; 2(M+N).
- In matching phase, read both relations; M+N I/Os.

13

Sort-Merge Join

- Sort R and S on the join column, then scan them to do a ``merge'' on the join column.
  - Advance scan of R until current R-tuple >= current S tuple, then advance scan of S until current S-tuple >= current R tuple; do this until current R tuple = current S tuple.
  - At this point, all R tuples with same value and all S tuples with same value match; output <r, s> for all pairs of such tuples.
  - Then resume scanning R and S.

14

Cost of Sort-Merge Join

- R is scanned once; each S group is scanned once per matching R tuple.
- Cost: M log M + N log N + (M+N)
  - The cost of scanning, M+N, could be M*N (unlikely!)
- Sort-Merge Join vs. Hash Join:
  - Given a minimum amount of memory both have a cost of 3(M+N) I/Os. Hash Join superior on this count if relation sizes differ greatly. Also, Hash Join shown to be highly parallelizable.

15

But in Data Integration…

- Few statistics, if any
  - *autonomous sources*
- Unanticipated delays and failures
  - *network-bound sources*
  - *A query plan can look good but not work well*
- Desiderata:
  - optimize *time to first tuples*
  - *Reduce load on sources*

16

Overview of adaptive query processing

- Adaptive query processing [HFC+00]
  - receiving information from its environment
  - using this information to determine its behaviors
  - iterating the two above in a feedback loop
- Levels of adaptation
  - Operators algorithms: adaptive operators
  - Operators order: re-scheduling
  - Structure of plan: re-optimizing

17

Some Possible Routing Policies

Deterministic

- Monitor costs & selectivities continuously
- Re-optimize periodically using rank ordering (or A-Greedy for correlated predicates)

Lottery scheduling

Each operator runs in thread with an input queue

- "Tickets" assigned according to tuples input / output
- Route tuple to next eligible operator with room in queue, based on number of "tickets" and "backpressure"

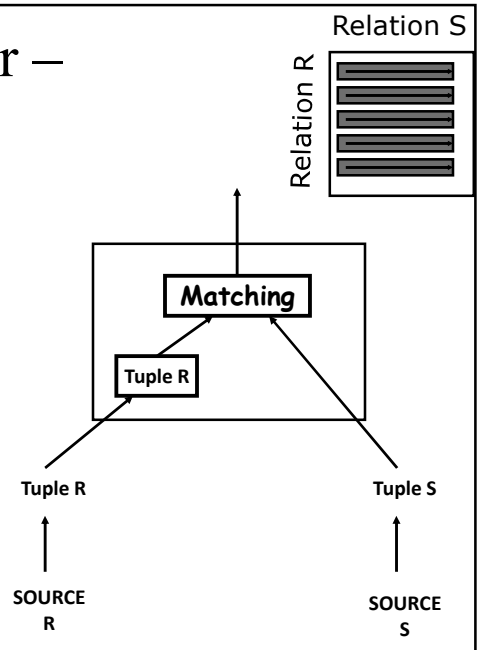
Content-based routing

- Different routes for different plans based on attribute values

18

Adaptive operator – RippleJoin

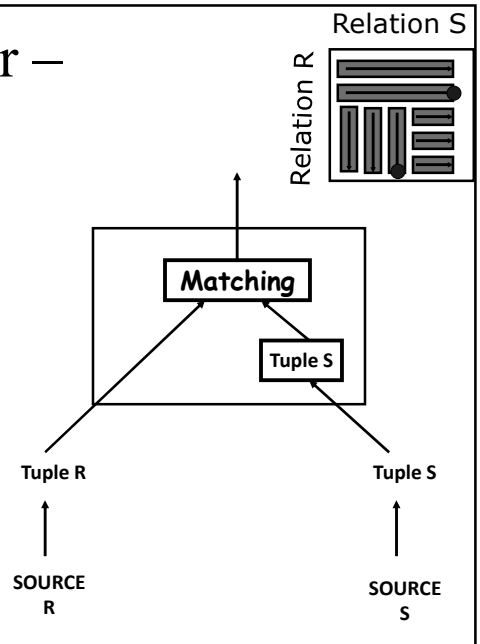
- extension of NLJ
- determining moments of symmetry
- dealing with
  - fluctuant arrival data rate



19

Adaptive operator – RippleJoin

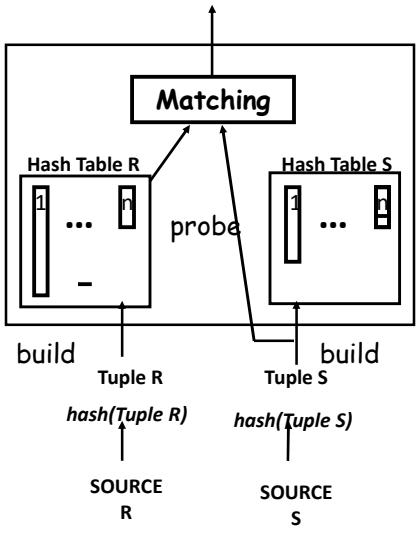
- extension of NLJ
- determining moments of symmetry
- dealing with
  - fluctuant arrival data rate



20

Adaptive operator - Symmetric Hash Join (SHJ)

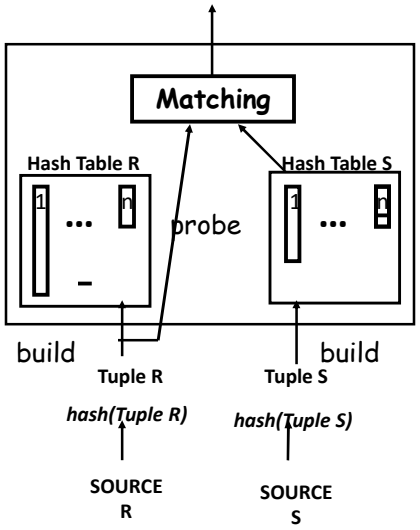
- extension of HJ
- using 2 hash tables
- dealing with
  - fluctuant arrival data rate



21

Adaptive operator - Symmetric Hash Join (SHJ)

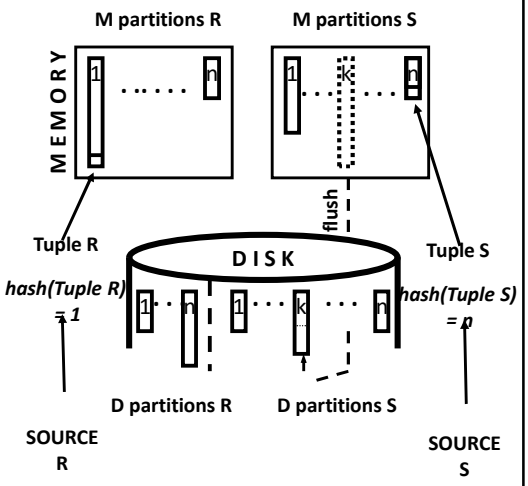
- extension of HJ
- using 2 hash tables
- dealing with
  - fluctuant arrival data rate



22

Adaptive operator - XJoin

- XJoin
  - extension of SHJ
  - storing a part of HTs to disk
  - dealing with
    - fluctuant arrival data rate and
    - Insufficient memory
- 3 phases of joint
  - mem – mem
  - mem – disk
  - cleaning

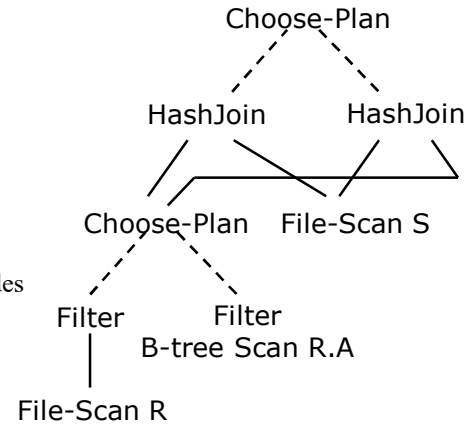


23

Dynamic query plan

- Problem
  - parameterized queries
  - lack of knowledge at compilation time
- Solution
  - combining different alternatives

  →Choose-Plan operator decides an alternative at *starting* query execution

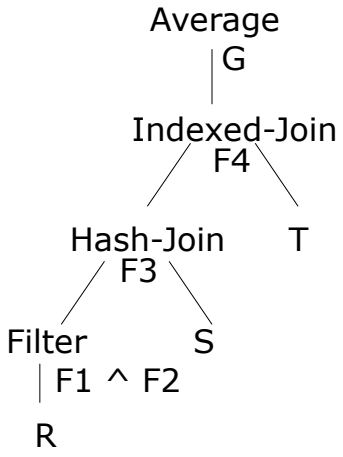


24

Mid-query re-optimization

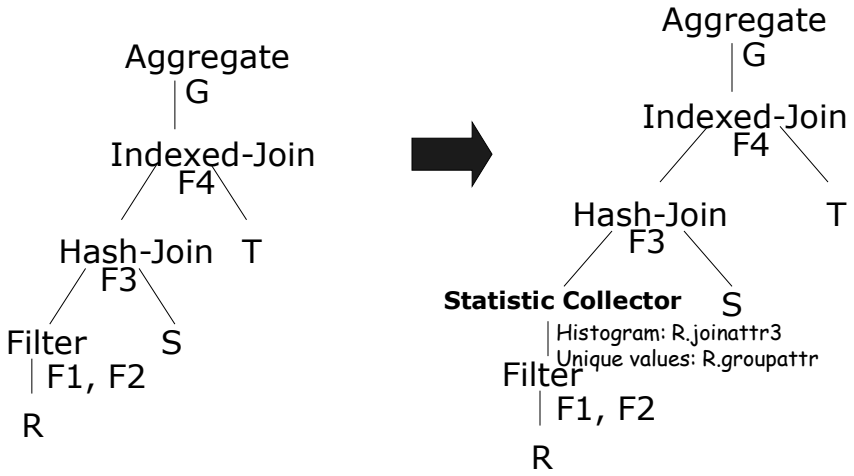
- Problem
  - inaccurate or unknown statistics
- Solution
  - collecting statistics at mid-query

  →a Static-Collector operator to
    - *collect* statistics on data
    - *materialize* intermediate results



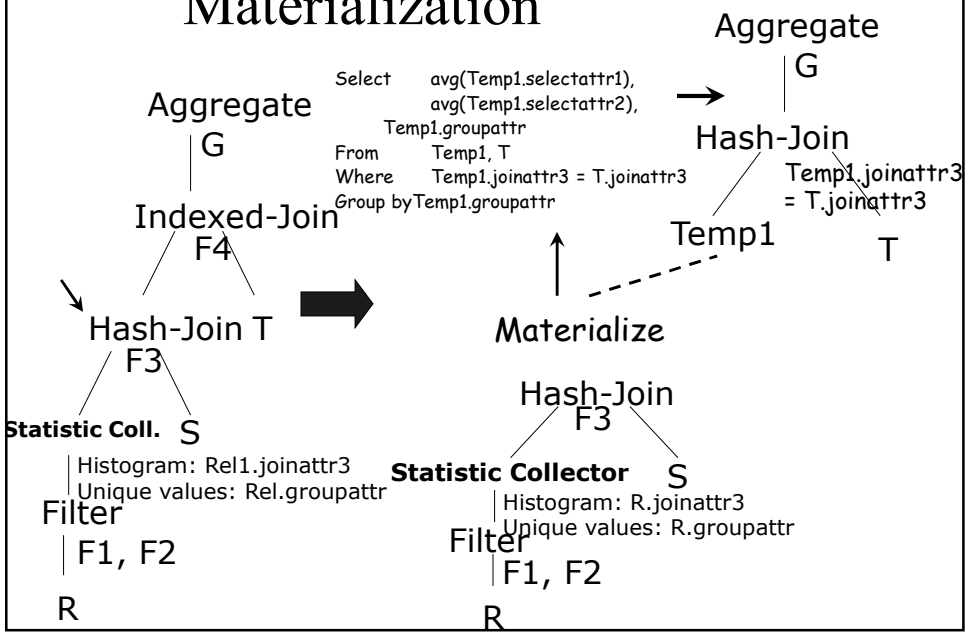
25

Statistic collection



26

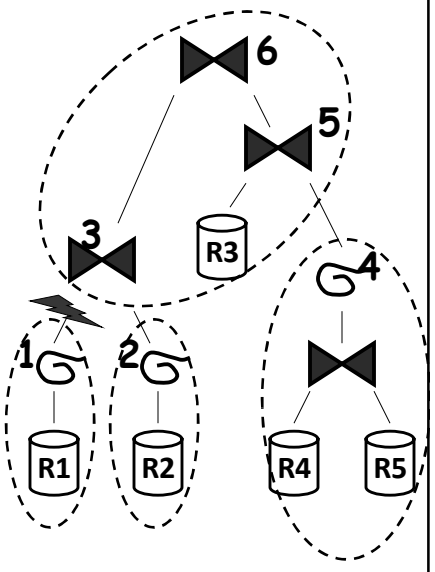
Materialization



27

Query scrambling

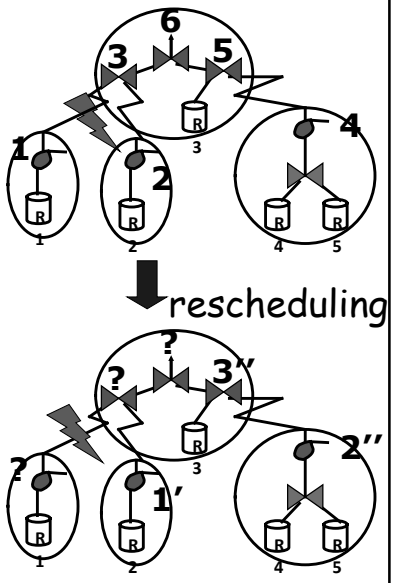
- Problem
  - network delays
- Solution
  - hiding delayed sources
  - running non-blocked operators
- 2 phases
  - rescheduling
  - operators synthesis



28

Phase 1: Rescheduling

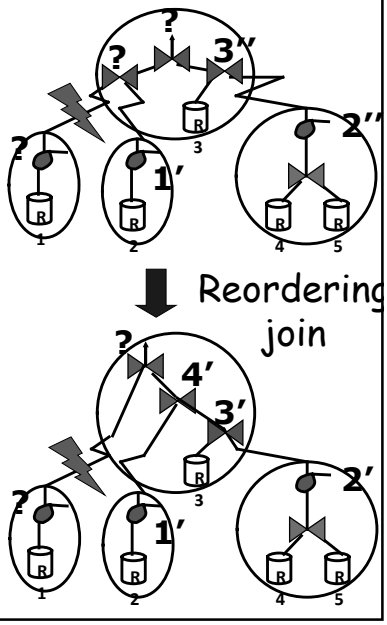
- Principle
  - running sub-tree that is independent from delayed sources or blocked operators (*non-blocked sub-tree*)
  - materializing intermediate results
- Process
  - processing a non-blocked sub-tree
  - if return of delayed sources, continuing
  - otherwise, processing other non-blocked sub-tree
  - if no sub-tree for processing
    → *phase 2*



29

Phase 2: Operator synthesis

- Principle
  - combining materialized data through new join operators
- Process
  - generating new join operator for materialized intermediate result
  - if delay is left, continuing
  - otherwise, generating other operators for new materialized data
  - if nothing to do, waiting data



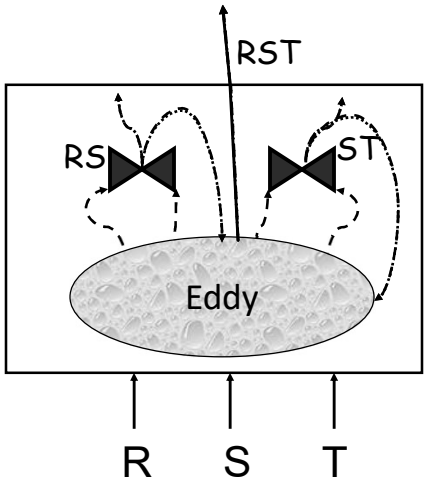
30

Eddy [AH00]

- Problem
  - Run-time fluctuations: cost, selectivity, rate
- Solution
  - per-tuple operator reordering

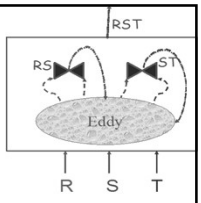
  →Eddy: *routing* tuples between operators (*modules*)

  no query plan
  but Eddy & *modules*



31

Eddy: easily routing ?



- Necessary properties of operator algorithms
  - Synchronization barrier
    - one input relation frozen, waiting for the other.

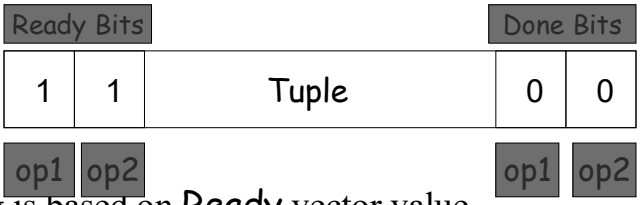
    →adaptive or non existent synchronization barriers
  - Moment of symmetry
    - points enabling to interchange the order of relations without affecting the output.

    →frequent moments of symmetry
- Routing
  - Principle: favors the production of result tuples
  - 2 policies: Naïve vs. Fast

32

Eddy: tuple structure

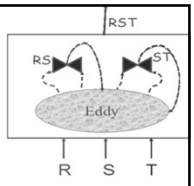
- Eddy's tuple = tuple + READY + DONE



- Routing is based on **Ready** vector value

33

Naïve Eddy

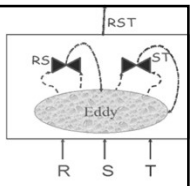



```
select *
from  R
where s1() and s2()
```

- Static query execution
  - either *s1() before s2()*
  - or *s2() before s1()*
- Statistics
  - 50% selectivity
  - Cost(s2) = 5 sec & Cost(s1) = {1..9} sec.
- Naïve Eddy routing policy:
  - all operators fetch from Eddy as fast as possible
  - previously-seen tuples precede new tuples

34

Fast Eddy



Loop:
- Given a tuple
- Determine which operators are eligible to process tuple
- If more than 1 operator, hold a lottery with chance of winning proportional to # tickets
- Operator gets 1 ticket if it gets a tuple
- Operator loses 1 ticket if it returns a tuple

Until every tuple can no longer be processed by any operator

Operator	# tickets
A	
B	

tickets tracks efficiency of an operator in dropping tuples from system

The system learns ordering of the operators

35

What is "*inappropriate*" in DB query processing ?

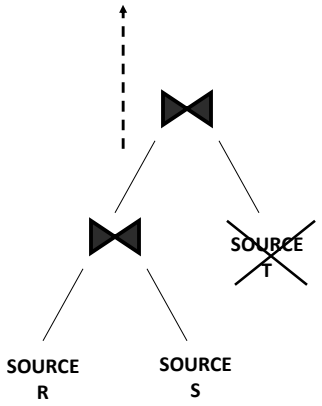
- DB queries : what (data) I want
  - *(may be) so large and not appropriate*
- Batch processing: wait for a query to complete
  - *long even interminable*

➔Need to
- enable client controls
- return feedbacks

36

Building partial results

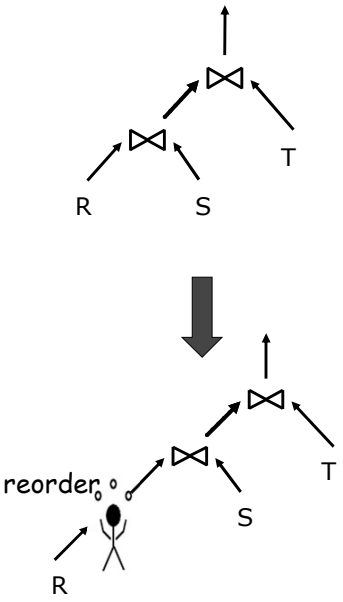
- Problem
  - unavailable data
  - client preference on some data
- Overview of solutions
  - returning incomplete results
  - routing data to client … *according to client interest*



37

Juggle

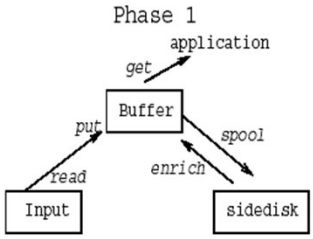
- Context
  - online query processing, i.e. users perceive data being processed over time
  - user interests as preferences
- Solution: prioritize processing for "interesting" tuples based on user-specified preferences
  - Prefetch & Spool (P&S)
  - reorder

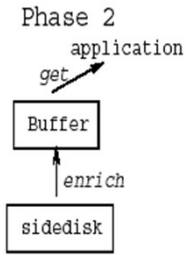


38

P&S technique

- prefetch from input, spools onto auxiliary side disk if needed
- juggle data between buffer and side disk, to keep buffer full of "*interesting*" items
  - getNext chooses best item currently on buffer
- reordering policy: determines
  - what getNext returns,
  - enrich/spool decisions

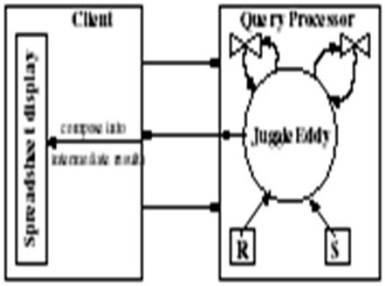




39

JuggleEddy

- Context
  - unavailable data
  - client preference (may be changed during QE)
  - spreadsheet display client application
- Solution: return continually results to client
  - based on hash client
  →defining key columns
  - routing tuples

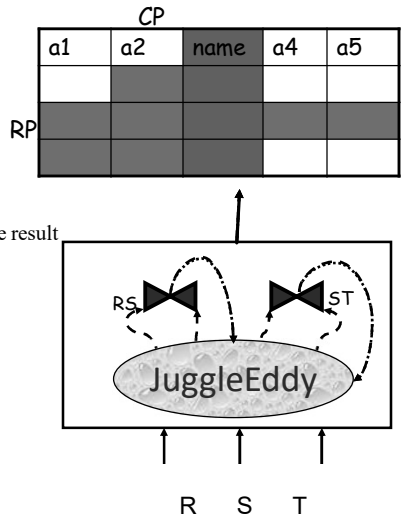


40

Partial results in JuggleEddy

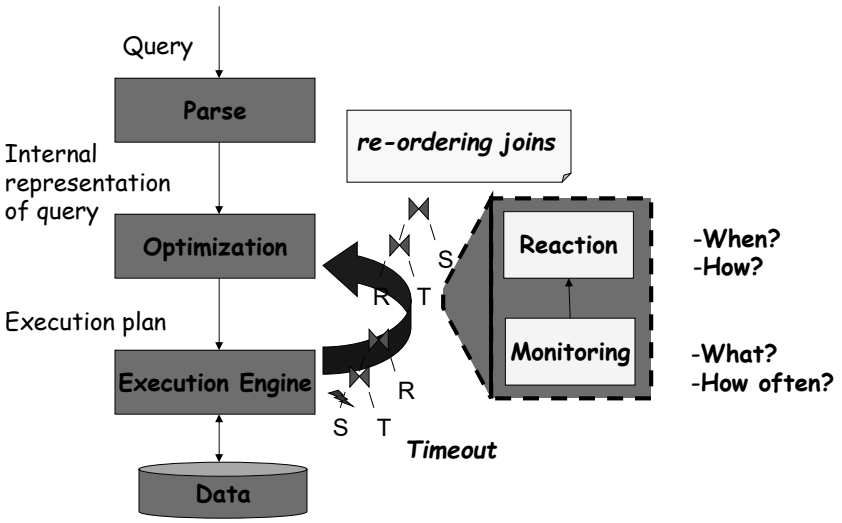
- Some definitions
  - partial result
    - ~any tuple produced by QP
  - intermediate result
    - ~incomplete result displayed at client
  - key column
    - ~ set of columns allowing to identify an unique result tuple
  - column properties (CPs)
  - row properties (RPs)
  - row column properties (RCPs)
  - benefit of a partial result tuple

    $\Sigma Benefit(cell[r,c]) = \Sigma RP(c)*CP(c)*RCP(r,c)$



41

Adaptive Query Processing Framework



42

Query Processing Summary

- We don't have the usual statistics
  - Hence, need to be adaptive
- Adaptivity needs to trade off several factors
- From a theoretical viewpoint, this is still an open problem.
  - And what if results are assumed to be approximate?

43

References

- J.M. Hellerstein, M.J. Franklin, S. Chandrasekaran, A. Deshpande, K. Hildrum, S. Madden, V. Raman, M.A. Shah. Adaptive Query Processing: Technology in Evolution. IEEE Data Engineering Bulletin 23(2), 2000
- Goetz Graefe and Karen Ward. Dynamic query evaluation plans. Proceedings ACM SIGMOD International Conference on Management of Data 1989
- Navin Kabra, David J. DeWitt: Efficient Mid-Query Re-Optimization of Sub-Optimal Query Execution Plans. Proceedings ACM SIGMOD International Conference on Management of Data 1998
- Tolga Urhan and Michael Franklin. XJoin: A Reactively-Scheduled Pipelined Join Operator. IEEE Data Engineering Bulletin, 2000
- Peter J. Haas and Joseph M. Hellerstein. Ripple Joins for Online Aggregation. SIGMOD 99'.

44