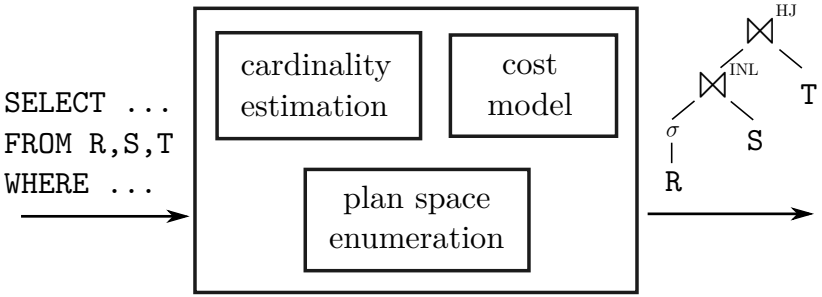
CMSC 848: Adaptive Query Processing

Instructor: Amol Deshpande
amol@cs.umd.edu

PostgreSQL Query Optimizer

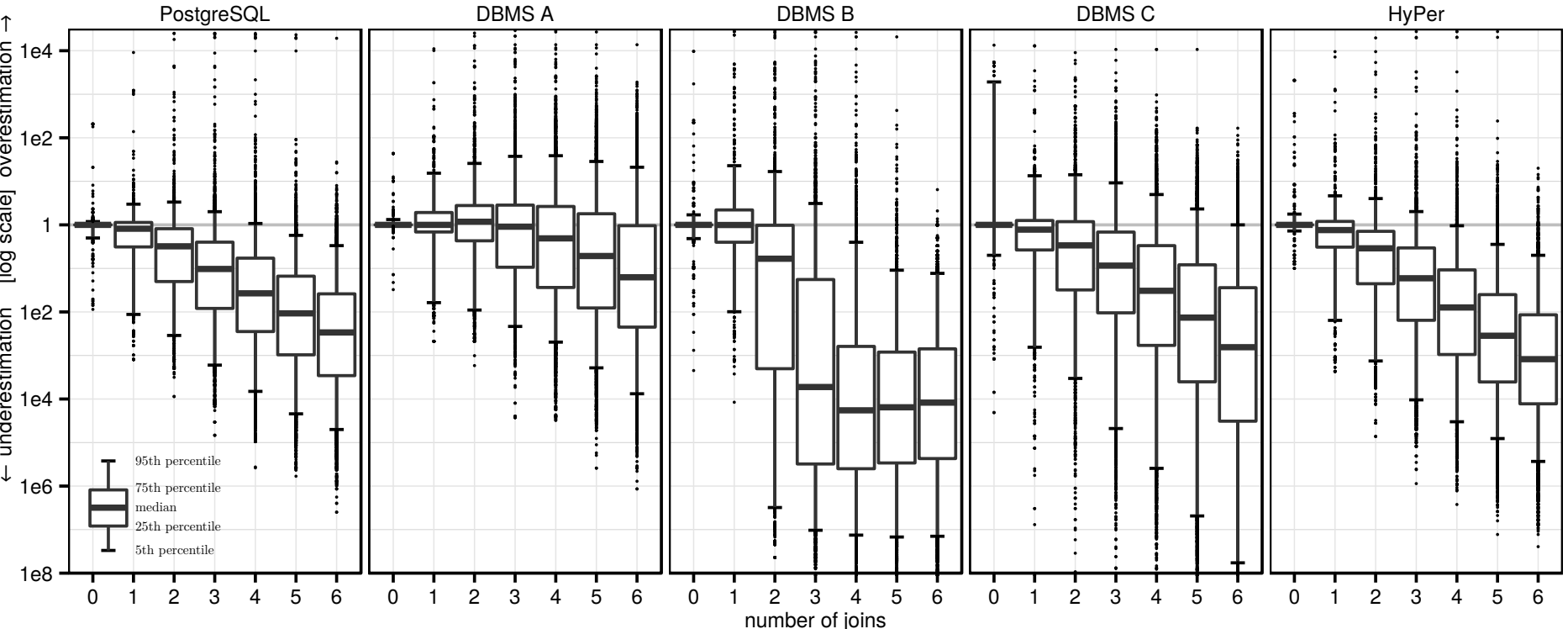
- Standard dynamic programming-based optimizer
  - Includes bushy plans, but no Cartesian products
- Statistics: Single-column histograms, min, max, most frequent values, etc.
  - Assume independence and uniformity outside of those
  - Especially for conjunctive predicates (like A = 10 and B = 20)



- Modified for the purposes of this paper to accept “cardinality injection”
  - i.e., use different cardinality estimates than the ones it computed
  - e.g., true cardinalities, or cardinalities per another system

Results: Cardinality Estimation

- q-error: ratio of correct result and estimate
- Base tables: sampling (Hyper and A) works better than histograms
- Huge underestimation seen as #joins increases
  - Underestimation generally worse – results in more aggressive plans (e.g., NL joins)
- Note: The experimental setup *may* naturally "select" for underestimates
  - (Missing enough details to be sure)



Results: What if we used "correct" estimates

- Used cardinality injection to use other systems' estimates or the true cardinalities
- Most bad plans boil down to NL joins
  - Disabling improves performance but doesn't fully solve the problem

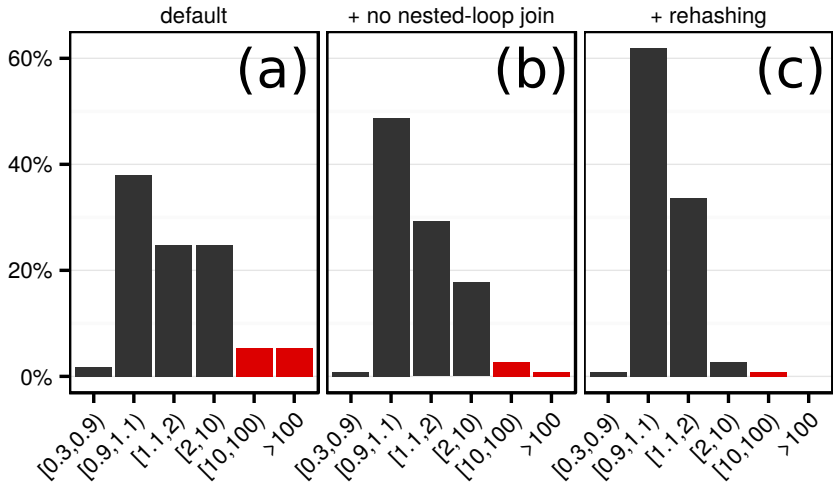


Figure 6: Slowdown of queries using PostgreSQL estimates w.r.t. using true cardinalities (primary key indexes only)

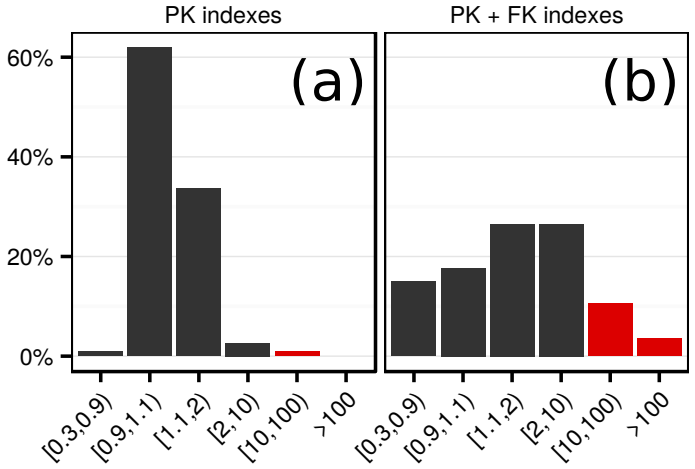


Figure 7: Slowdown of queries using PostgreSQL estimates w.r.t. using true cardinalities (different index configurations)

Results: Cost Models

- PostgreSQL uses a disk-oriented cost model – a weighted sum of I/O and CPU costs
  - No easy way to set the parameters
- Plot predicted costs vs actual costs – a linear line is the best outcome here
- Findings:
  - Default estimates result in fairly poor fit – predicted and actual costs quite different
  - Most of the error goes away if the optimizer has access to true cardinalities
  - Tuning the cost model doesn't really help that much
  - Using a much simpler cost model gives similar results
    - Just count the number of tuples being processed by each operator

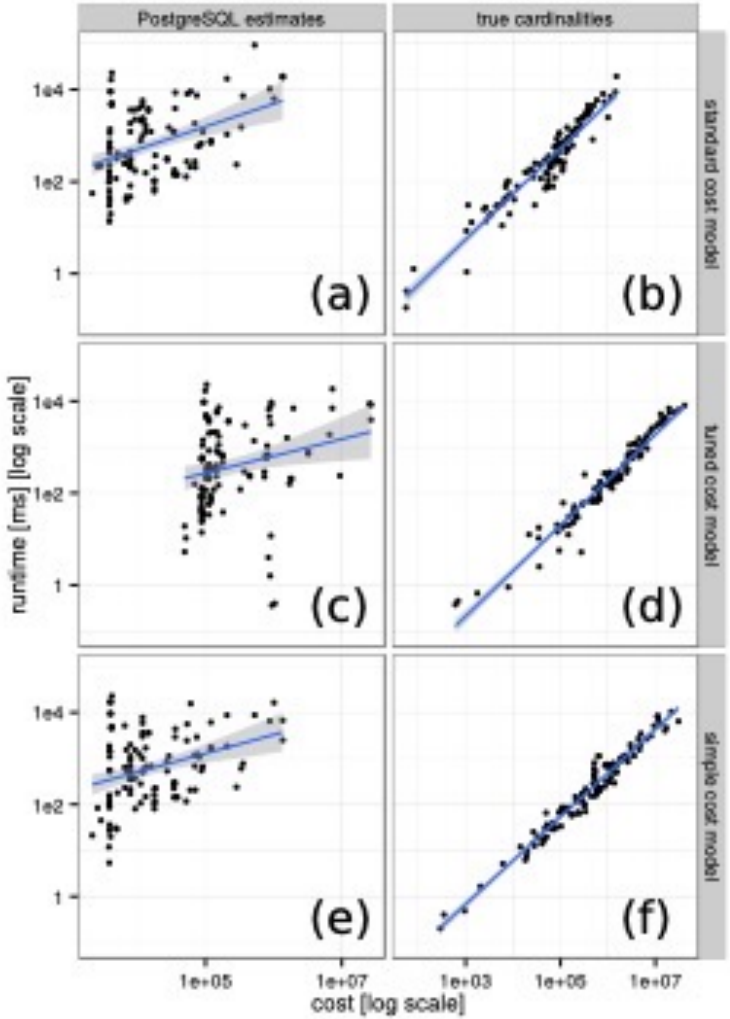


Figure 8: Predicted cost vs. runtime for different cost models

Results: Join Orders

Computed estimated costs with true cardinalities for 1000 random plans

Slowest or even median query plans much worse than optimal (several orders of magnitude in many cases)

Prior work from approx. 20 years ago that does this in more depth

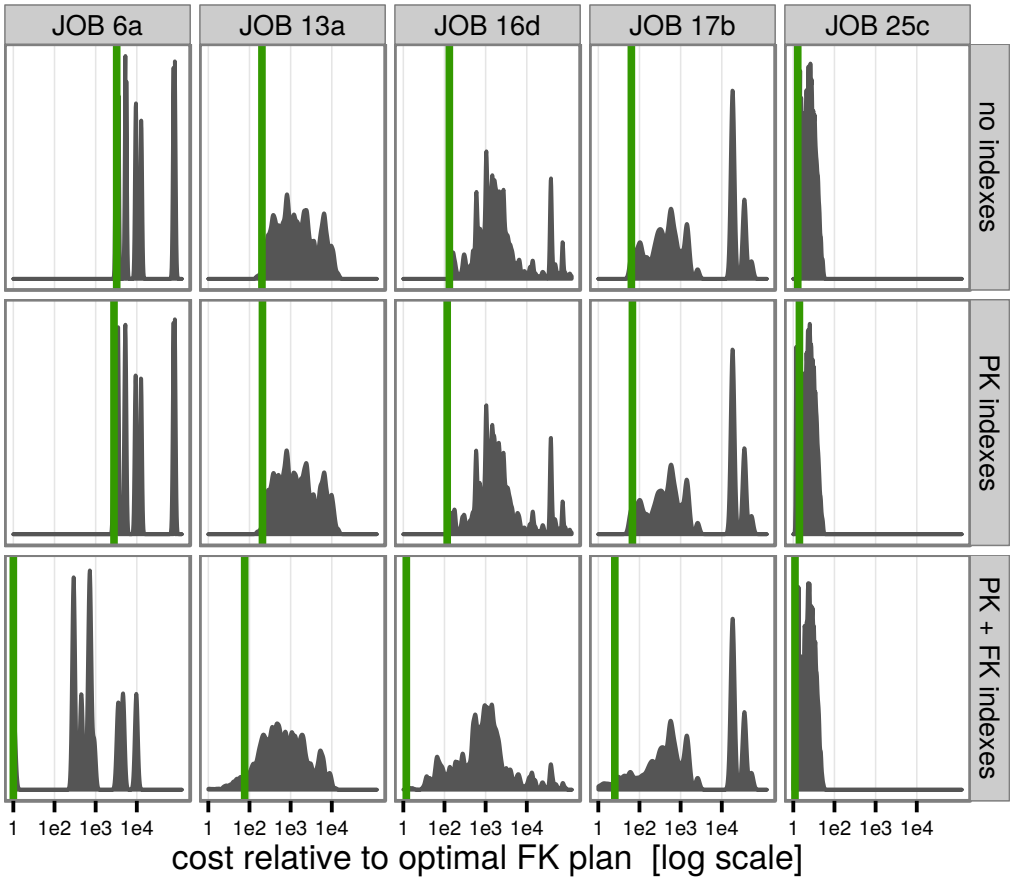


Figure 9: Cost distributions for 5 queries and different index configurations. The vertical green lines represent the cost of the optimal plan

Results: Join Orders

- Bushy trees important to consider

	PK indexes			PK + FK indexes		
	median	95%	max	median	95%	max
zig-zag	1.00	1.06	1.33	1.00	1.60	2.54
left-deep	1.00	1.14	1.63	1.06	2.49	4.50
right-deep	1.87	4.97	6.80	47.2	30931	738349

Table 2: Slowdown for restricted tree shapes in comparison to the optimal plan (true cardinalities)

- Exhaustive algorithms (DP or top-down) needed

	PK indexes						PK + FK indexes					
	PostgreSQL estimates			true cardinalities			PostgreSQL estimates			true cardinalities		
	median	95%	max	median	95%	max	median	95%	max	median	95%	max
Dynamic Programming	1.03	1.85	4.79	1.00	1.00	1.00	1.66	169	186367	1.00	1.00	1.00
Quickpick-1000	1.05	2.19	7.29	1.00	1.07	1.14	2.52	365	186367	1.02	4.72	32.3
Greedy Operator Ordering	1.19	2.29	2.36	1.19	1.64	1.97	2.35	169	186367	1.20	5.77	21.0

Table 3: Comparison of exhaustive dynamic programming with the Quickpick-1000 (best of 1000 random plans) and the Greedy Operator Ordering heuristics. All costs are normalized by the optimal plan of that index configuration

Correlations

- Single-table (e.g., R.A and R.B are correlated, throwing off estimation of R.A = 10 and R.B = 20)
  - Handled by the "sampling" techniques
  - Build multi-dimensional histograms (don't really work well)
  - Identify "soft" functional dependencies (i.e., very highly correlated columns)
    - e.g., "car make" and "car model" are highly correlated
    - Queries like: Make = Honda and Model = Accord are underestimated
    - But not a functional dependency: Model → Make is false

- Join-crossing Correlations

```
select *
from actors JOIN movies
where actors.location = 'Paris' and movies.language = 'French'
```

  - Unclear how one can benefit from capturing this correlation (even if one could)
  - Need a new operator or access method

Traditional Optimization not Robust Enough

- In traditional settings:
  - Queries over many tables
  - Unreliability of traditional cost estimation
  - Success, maturity make problems more apparent, critical
- In new environments:
  - e.g. data integration, web services, streams, P2P...
  - Unknown dynamic characteristics for data and runtime
  - Increasingly aggressive sharing of resources and computation
  - Interactivity in query processing
- Note two distinct themes lead to the same conclusion:
  - Unknowns: even static properties often unknown in new environments and often unknowable a priori
  - Dynamics: environment changes can be very high
- Motivates intra-query adaptivity

Some Related Topics

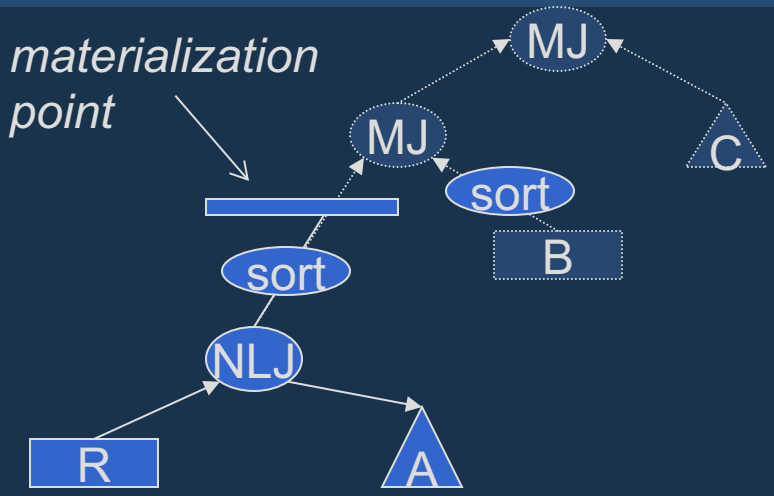
- Autonomic/self-tuning optimization
  - Chen and Roussoupolous: Adaptive selectivity estimation [SIGMOD 1994]
  - LEO (@IBM), SITS (@MSR): Learning from previous executions
- Robust/least-expected cost optimization
- Parametric optimization
  - Choose a collection of plans, each optimal for a different setting of parameters
  - Select one at the beginning of execution
- Competitive optimization
  - Start off multiple plans... kill all but one after a while
- Adaptive operators

  More details in our survey: “Adaptive Query Processing”; FnT 2007

AQP: Overview/Summary

- Low-overhead, evolutionary approaches
  - Typically apply to non-pipelined execution
  - **Late binding:** Don't instatntiate the entire plan at start
  - **Mid-query reoptimization:** At "materialization" points, review the remaining plan and possibly re-optimize
- Pipelined execution
  - No materialization points, so the above doesn't apply
  - The operators may contain complex states, raising correctness issues
  - **Eddies**
    - Always guarantee correct execution, but allows reordering during execution
- Lot of work in 1998-2008 timeframe -- not much since

Late Binding; Staged Execution



Normal execution: pipelines separated by materialization points

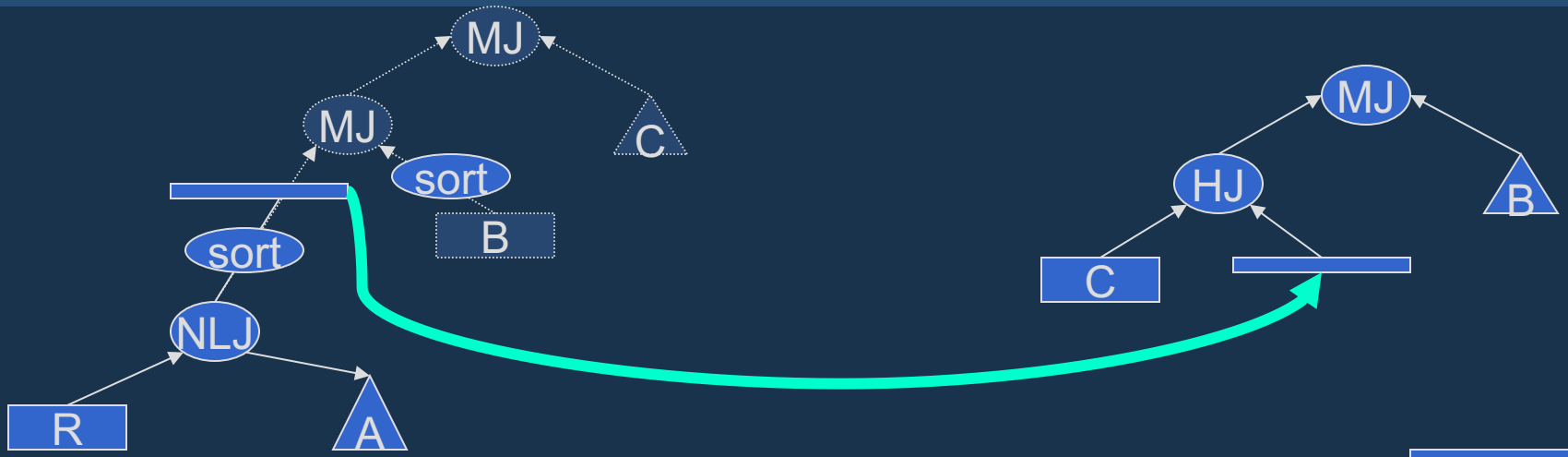
e.g., at a sort, GROUP BY, etc.

Materialization points make natural decision points where the *next* stage can be changed with little cost:

- Re-run optimizer at each point to get the next stage
- Choose among precomputed set of plans – *parametric* query optimization [INSS'92, CG'94, …]

Mid-query Reoptimization

[KD'98,MRS+04]



Choose *checkpoints* at which to monitor cardinalities — *Where?*

Balance overhead and opportunities for switching plans

If actual cardinality is too different from estimated, — *When?*

Avoid unnecessary plan re-optimization (where the plan doesn't change)

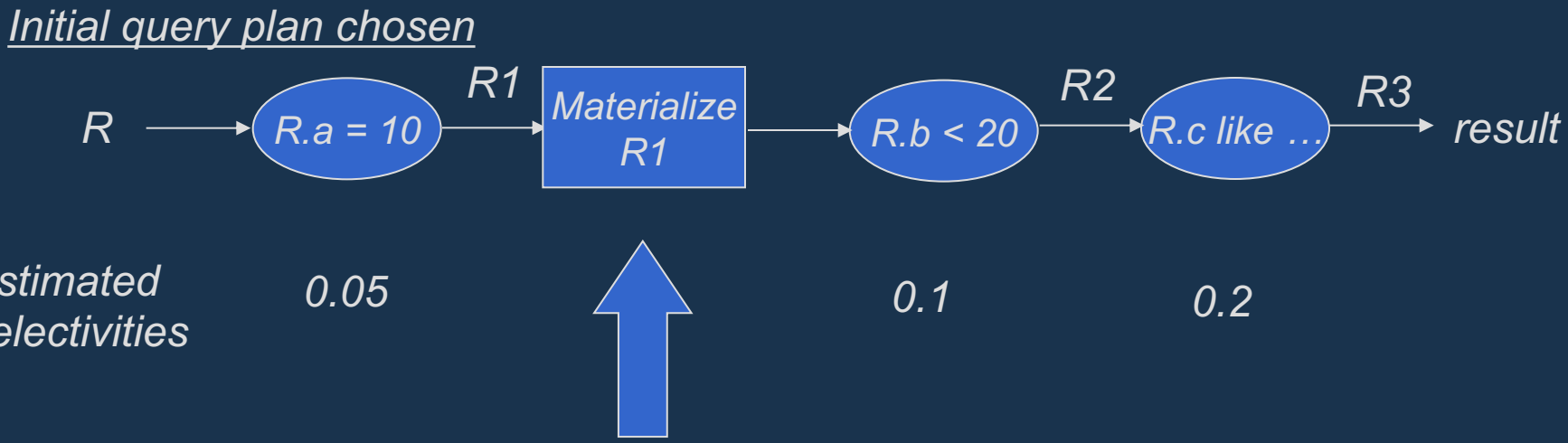
***Re-optimize* to switch to a new plan** — *How?*

Try to maintain previous computation during plan switching

- Most widely studied technique:
  - -- Federated systems (InterViso 90, MOOD 96), Red Brick, Query scrambling (96), Mid-query re-optimization (98), Progressive Optimization (04), Proactive Reoptimization (05), …

Mid-query Reoptimization

- At *materialization points,* re-evaluate the rest of the query plan
- Example:

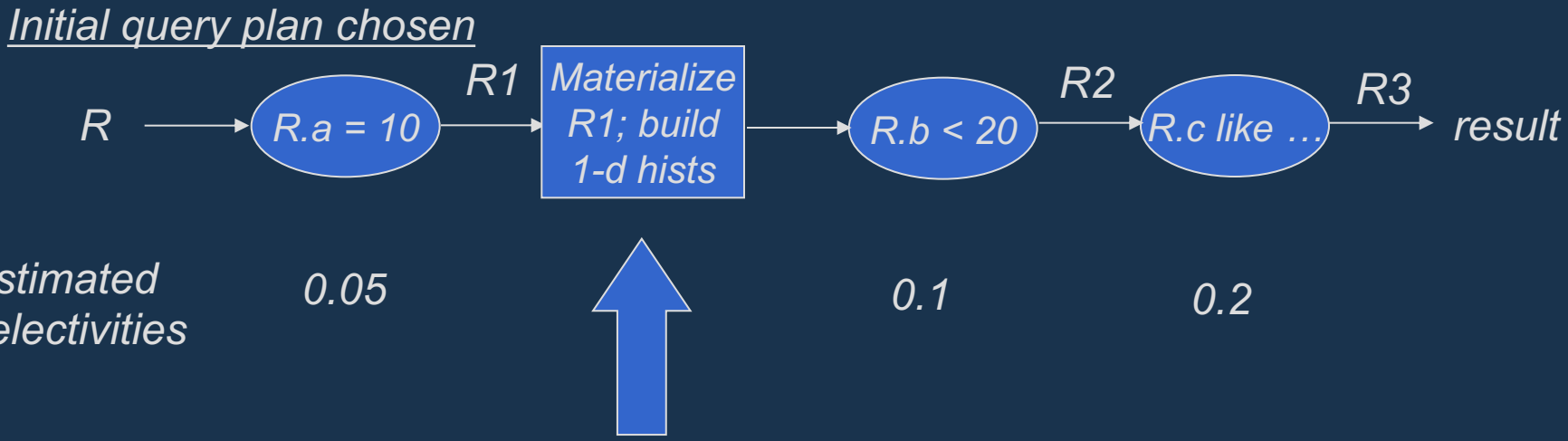


A *free* opportunity to re-evaluate *the rest of the query plan*

- Exploit by gathering information about the materialized result

Mid-query Reoptimization

- At *materialization points,* re-evaluate the rest of the query plan
- Example:

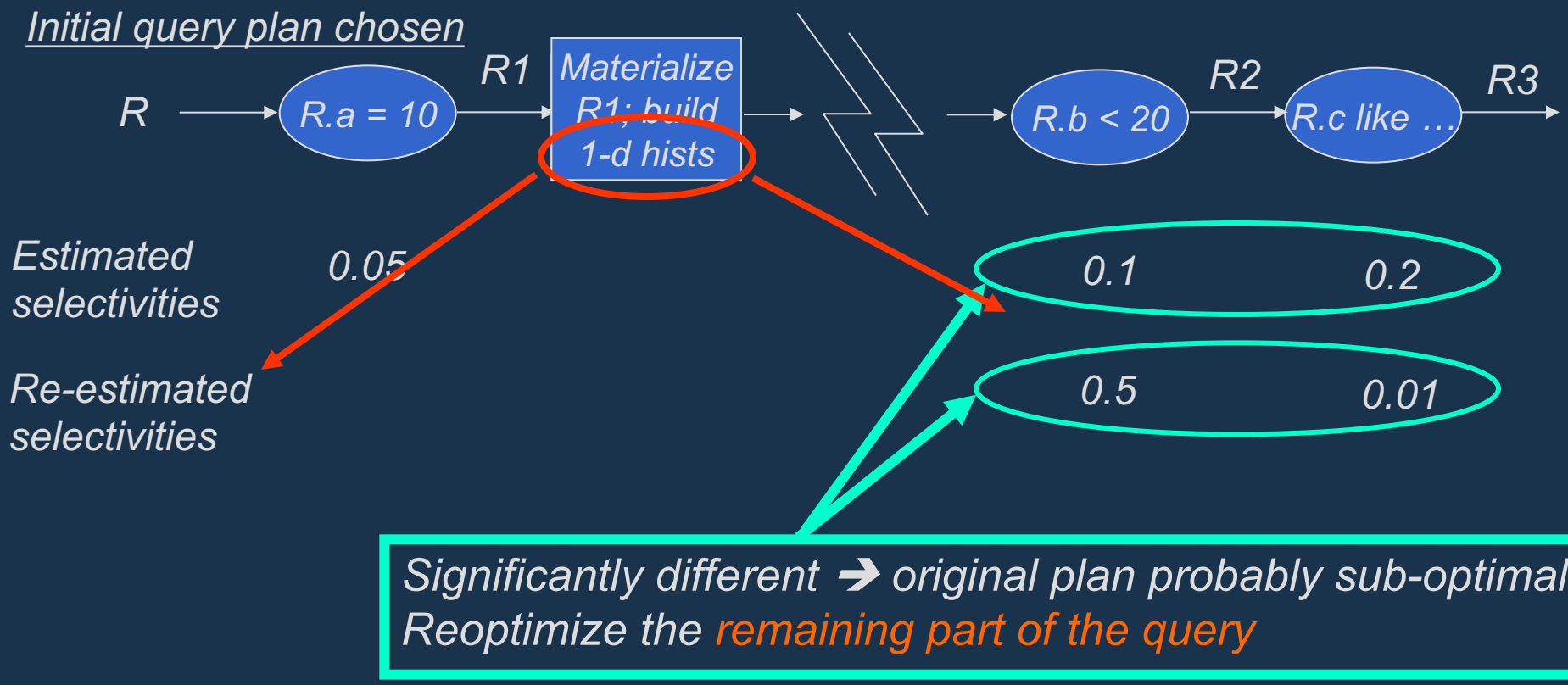


A *free* opportunity to re-evaluate *the rest of the query plan*

- Exploit by gathering information about the materialized result

Mid-query Reoptimization

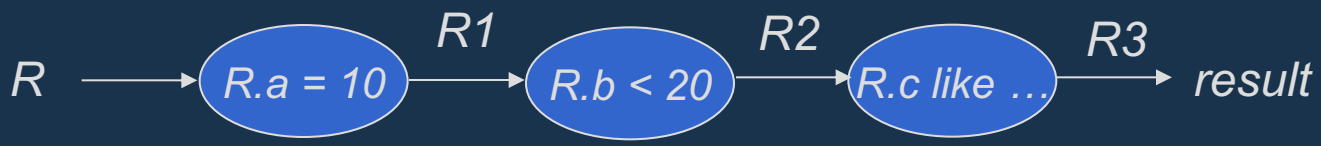
- At *materialization points,* re-evaluate the rest of the query plan
- Example:



Eddies [AH'00]

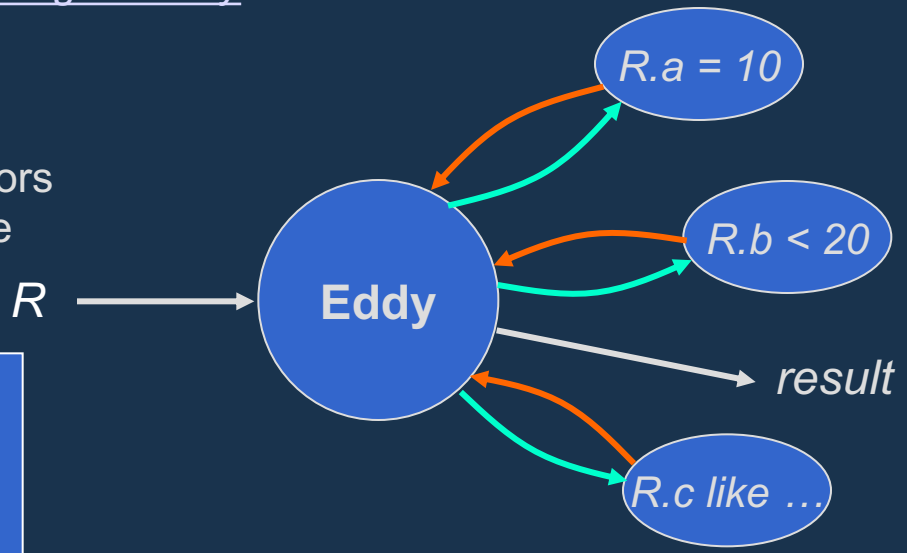
Query processing as routing of tuples through operators

A traditional pipelined query plan



Pipelined query execution using an eddy

An *eddy* operator
- Intercepts tuples from sources and output tuples from operators
- Executes query by routing source tuples through operators

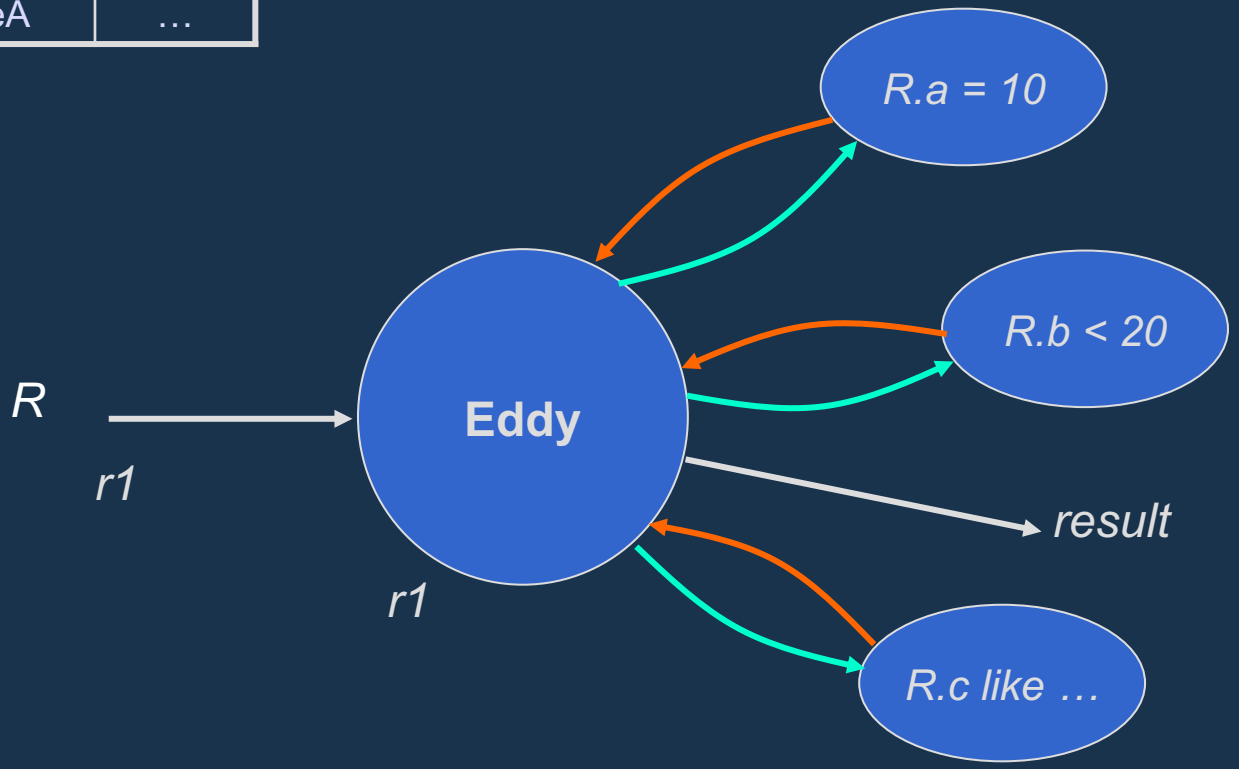


Encapsulates all aspects of adaptivity in a "standard" dataflow operator: measure, model, plan and actuate.

Eddies [AH'00]

An R Tuple: r1

a	b	c	…
15	10	AnameA	…



Eddies [AH'00]

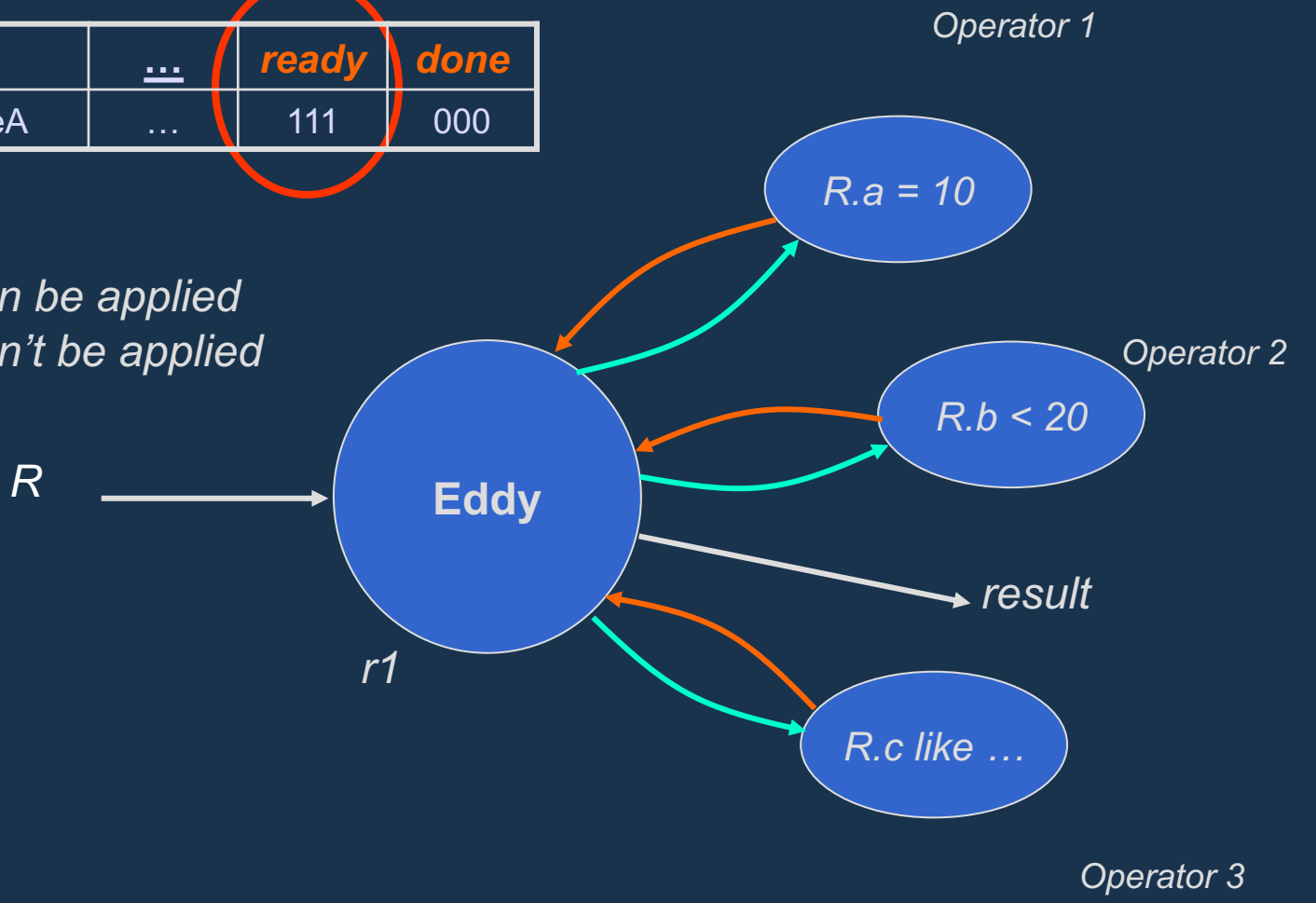
An R Tuple: r1

a	b	c	…	ready	done
15	10	AnameA	…	111	000

ready bit i :

1 → operator i *can be applied*

0 → operator i *can't be applied*



Eddies [AH'00]

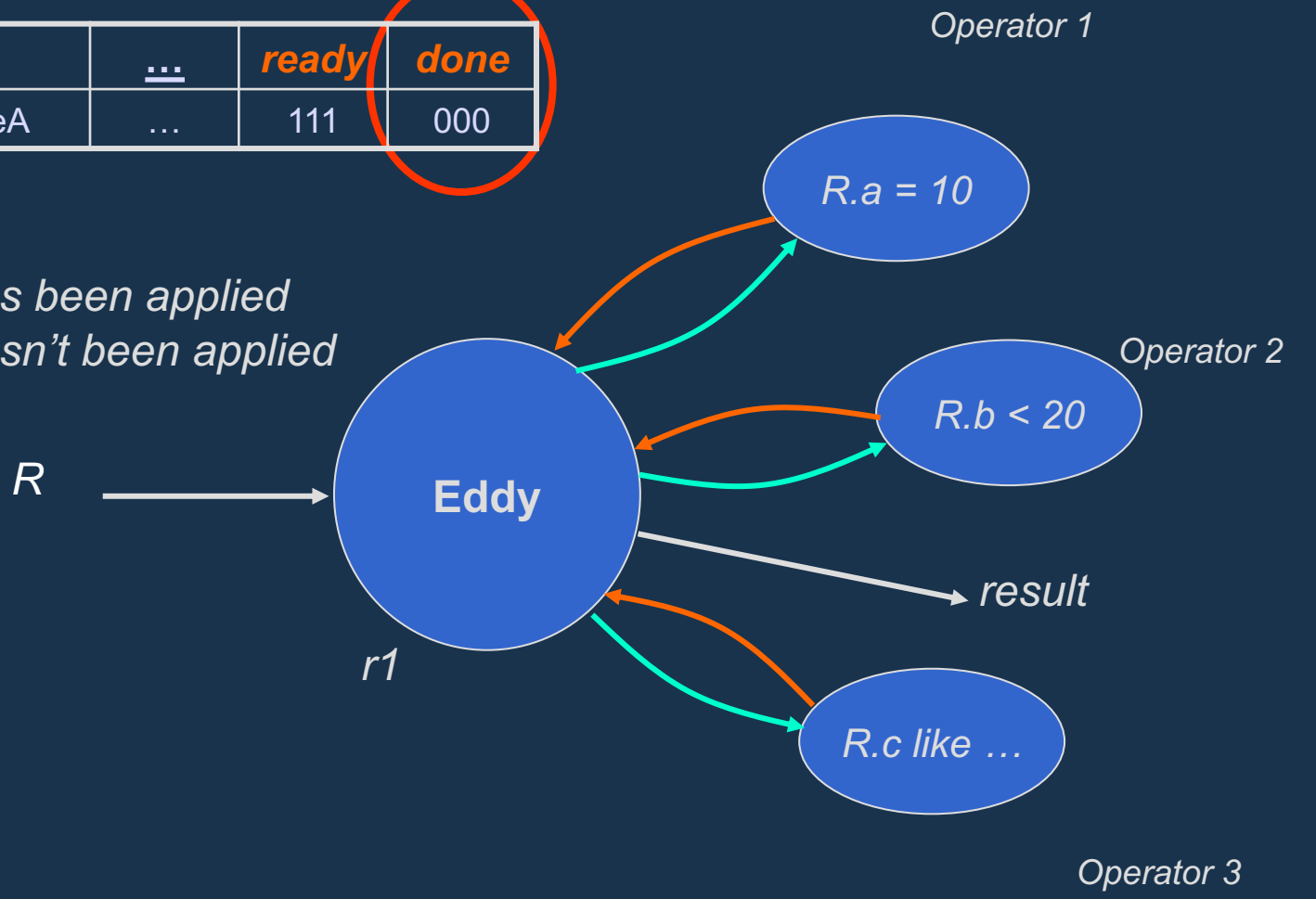
An R Tuple: r1

a	b	c	…	ready	done
15	10	AnameA	…	111	000

done bit i :

1 → operator i *has been applied*

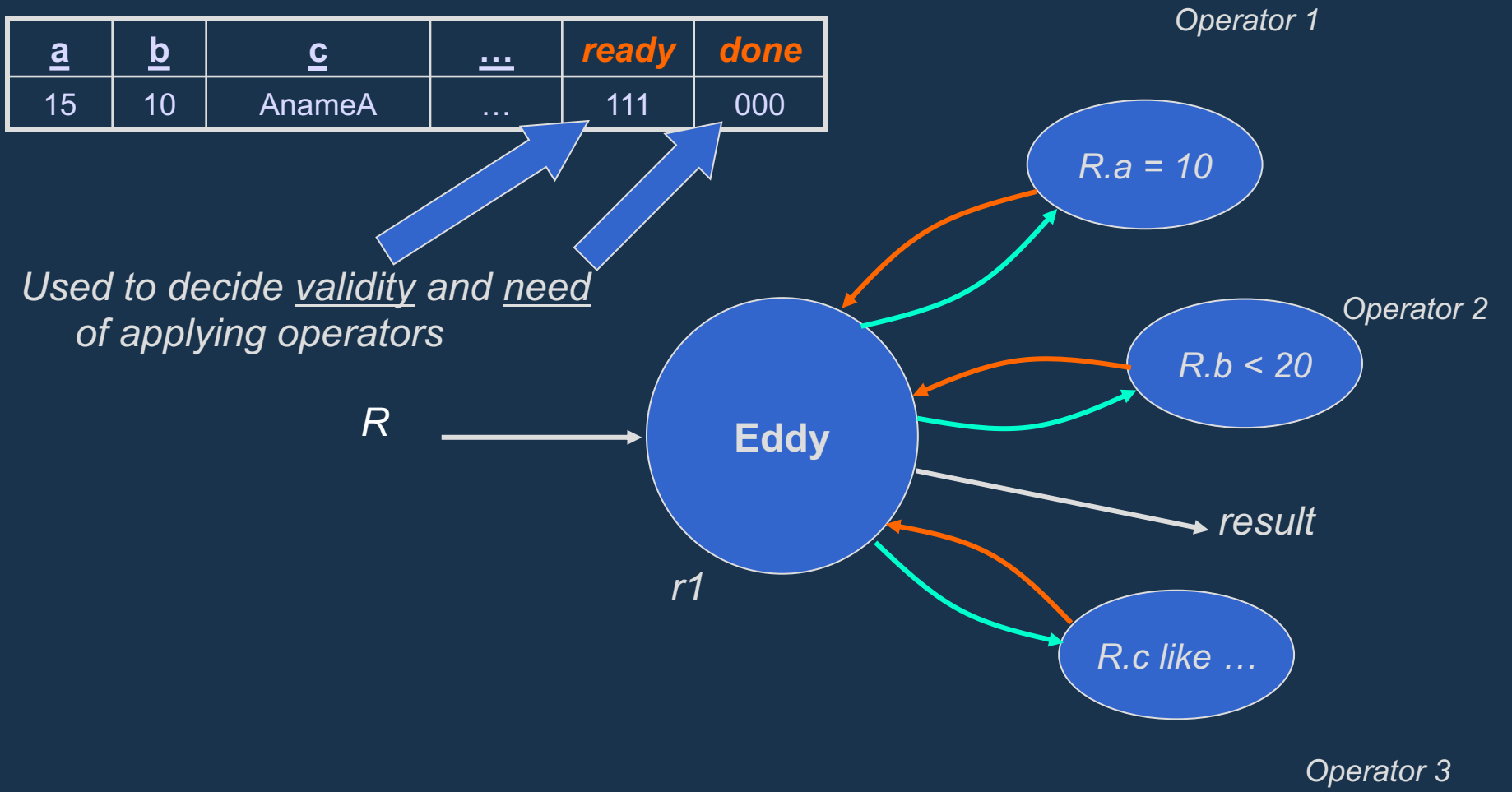
0 → operator i *hasn't been applied*



Eddies [AH'00]

An R Tuple: r1

a	b	c	…	ready	done
15	10	AnameA	…	111	000

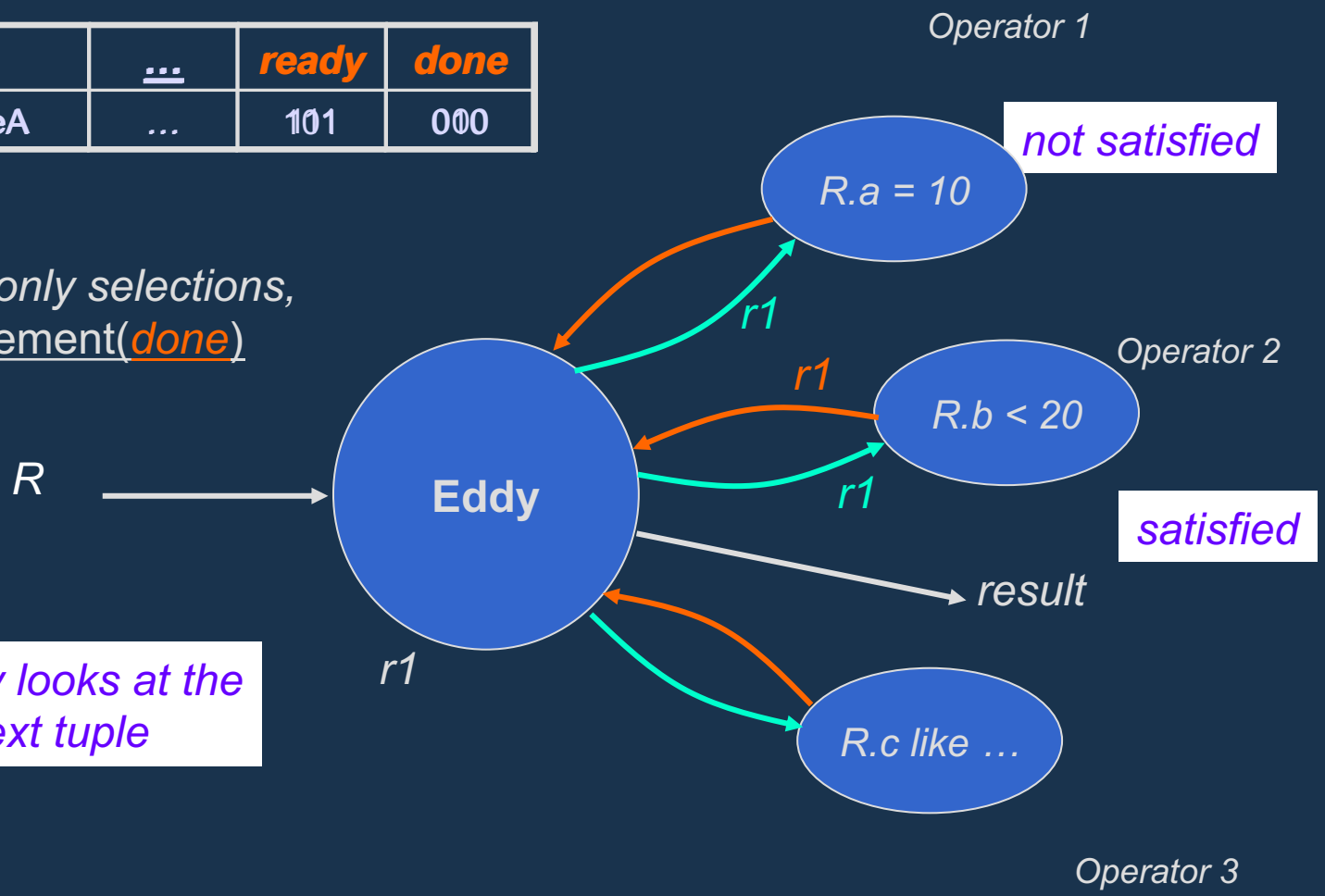


Eddies [AH'00]

An R Tuple: r1

a	b	c	…	ready	done
15	10	AnameA	…	101	000

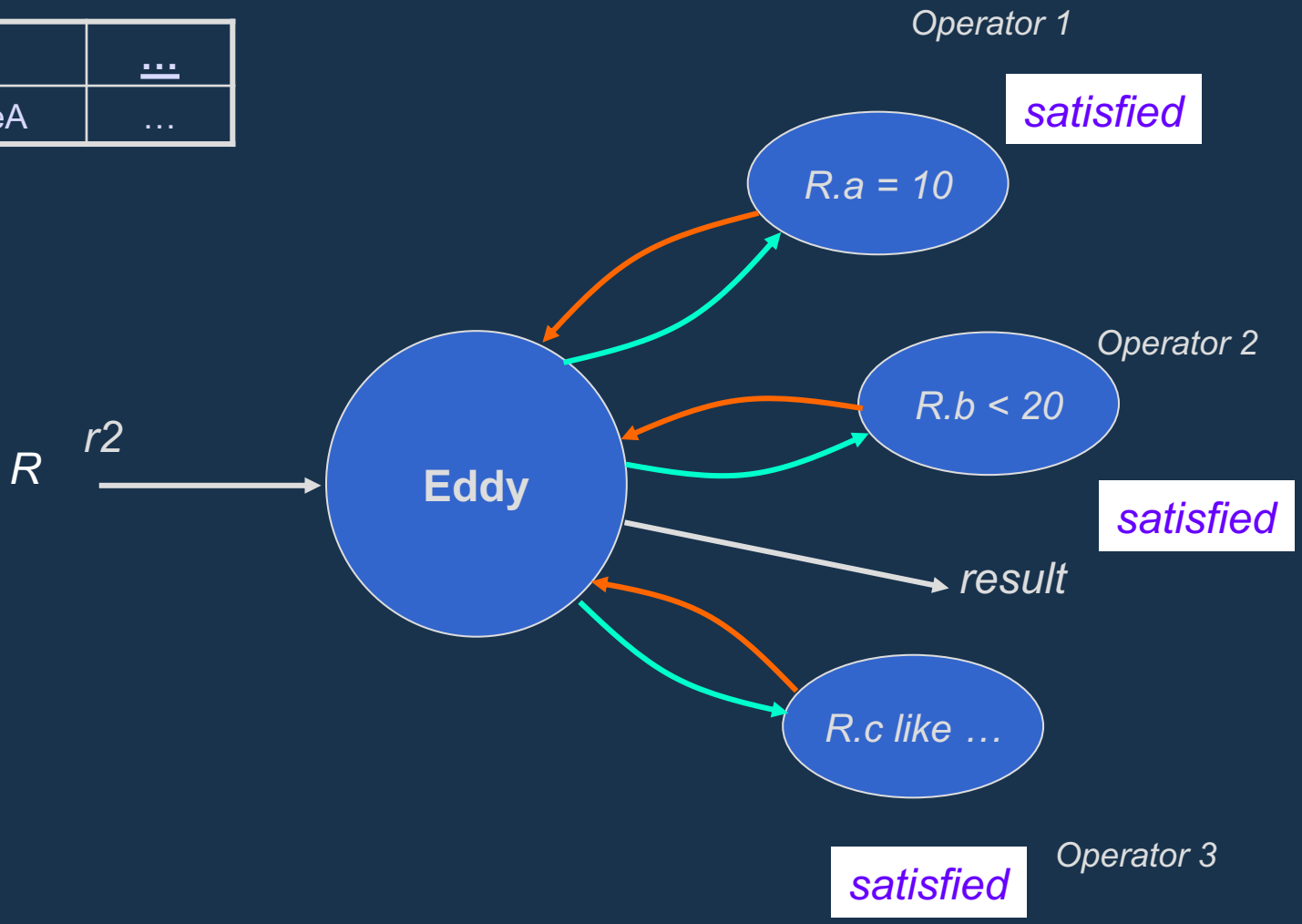
For a query with only selections,
ready = complement(*done*)



Eddies [AH'00]

An R Tuple: r2

a	b	c	…
10	15	AnameA	…

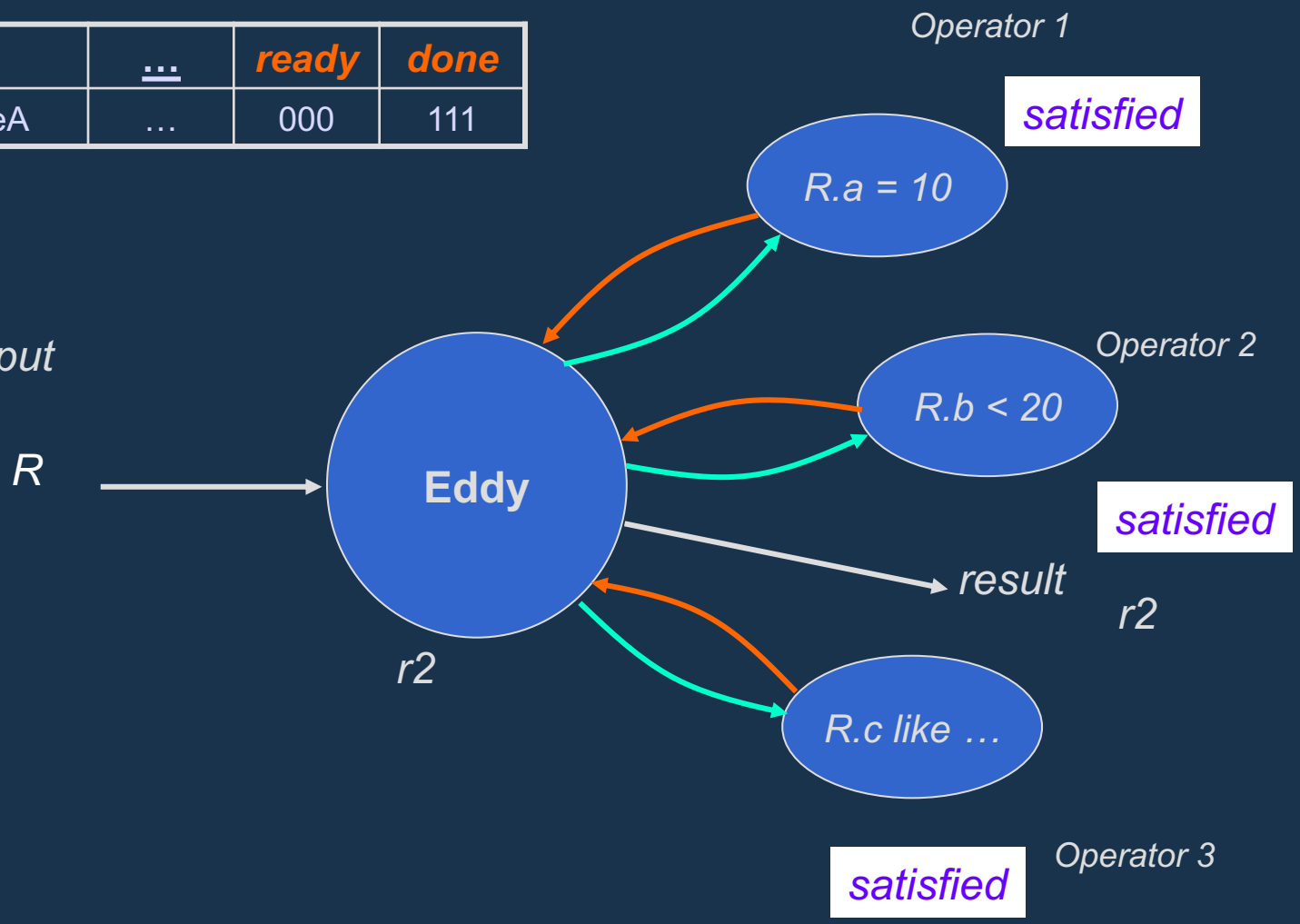


Eddies [AH'00]

An R Tuple: r2

a	b	c	…	ready	done
10	15	AnameA	…	000	111

if done = 111,
send to output



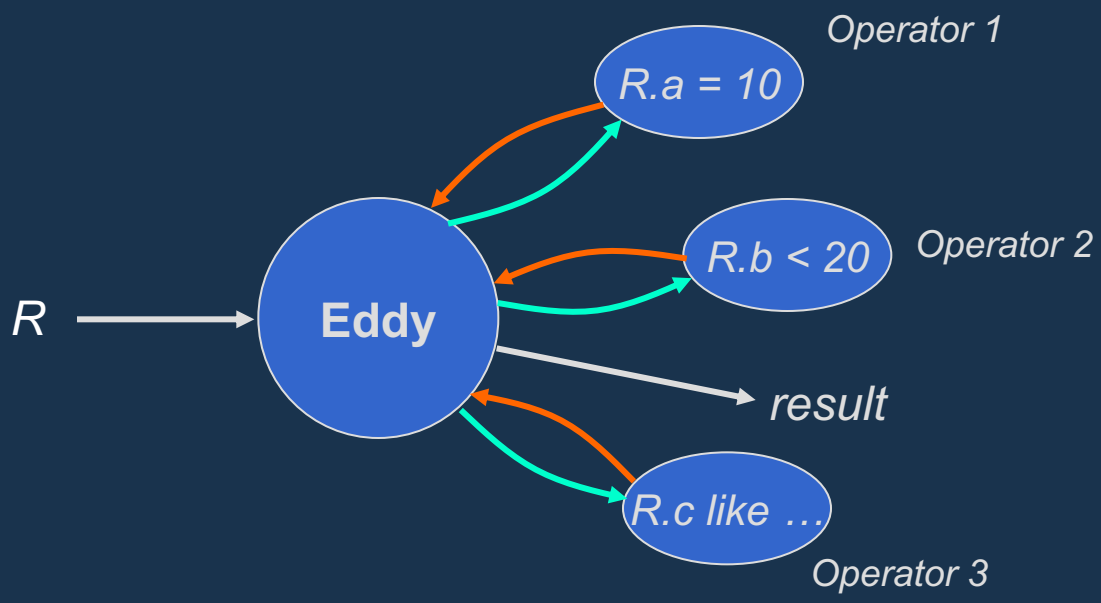
Eddies [AH'00]

Adapting order is easy

- Just change the operators to which tuples are sent
- Can be done on a per-tuple basis
- Can be done in the middle of tuple's "pipeline"

How are the *routing decisions* made?

Using a *routing policy*



Routing Policies that Have Been Studied

Deterministic [D03]

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

Lottery scheduling [AH00]

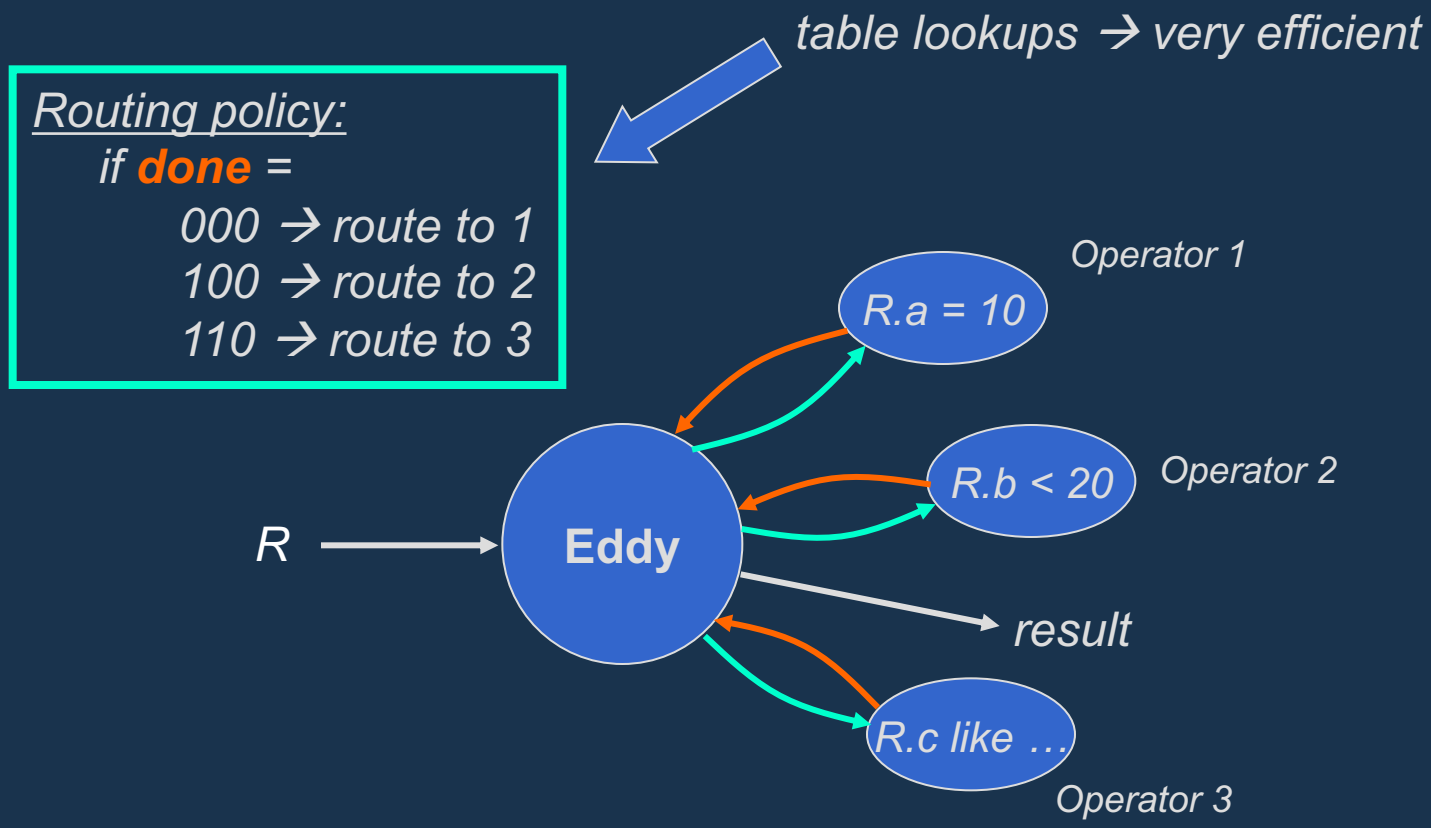
- Each operator runs in thread with an input queue
- “Tickets” assigned according to tuples input / output
- Route tuple to next eligible operator with room in queue, based on number of “tickets” and “backpressure”

Content-based routing [BBDW05]

- Different routes for different plans based on attribute values

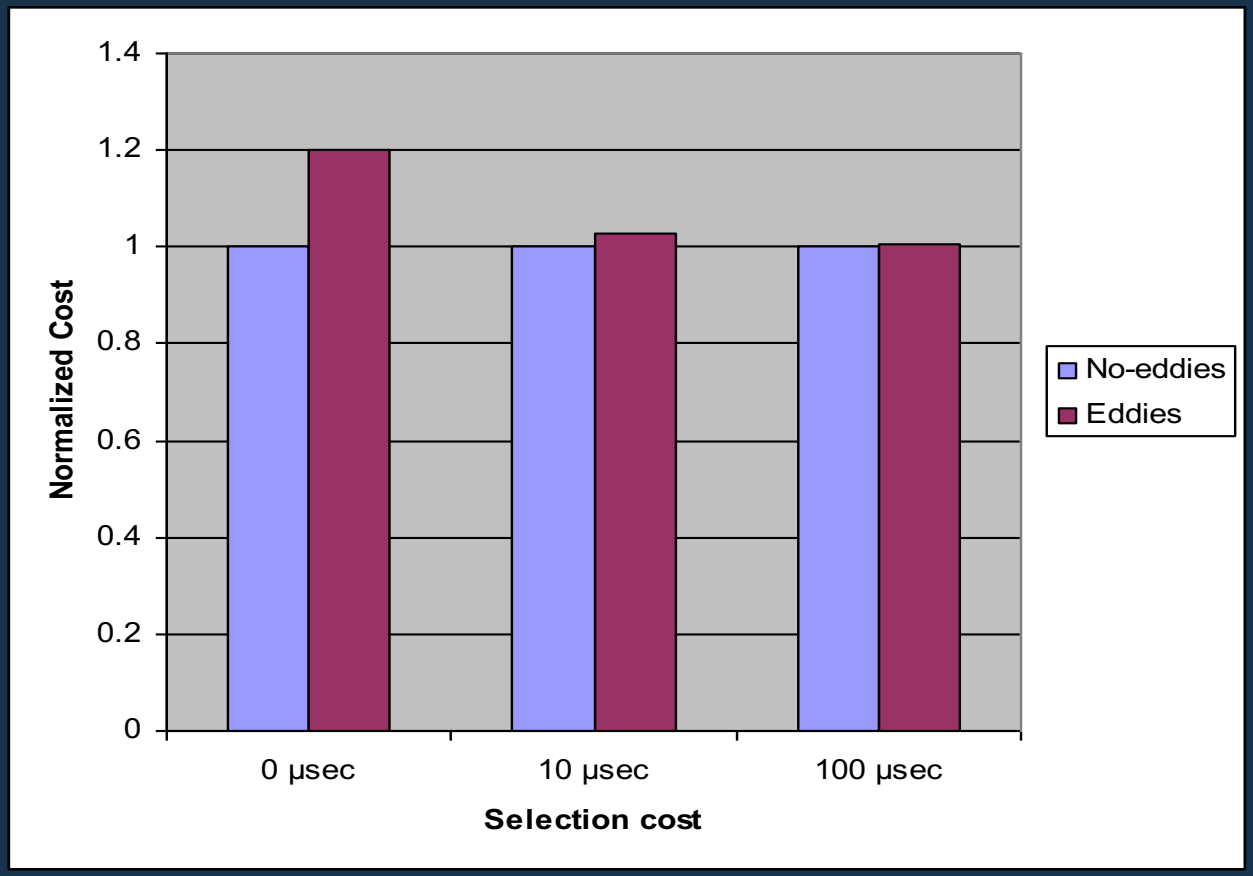
Routing Policy 1: Non-adaptive

- Simulating a single static order
  - E.g. operator 1, then operator 2, then operator 3



Overhead of Routing

- PostgreSQL implementation of eddies using *bitset lookups* [Telegraph Project]
- Queries with 3 selections, of varying cost
  - Routing policy uses a *single static order,* i.e., no adaptation



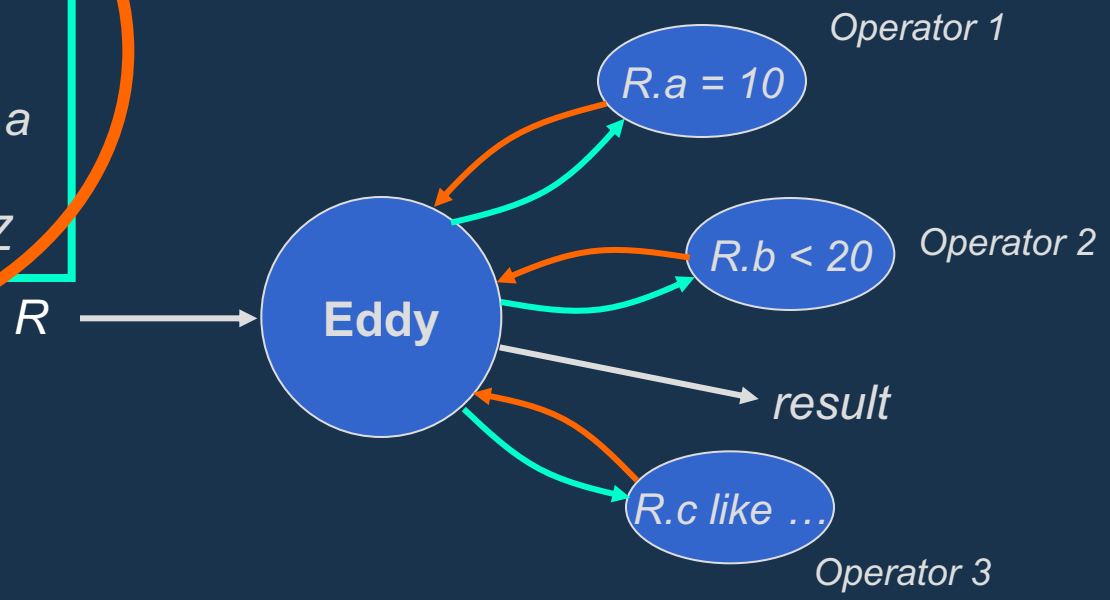
Routing Policy 2: Deterministic

- Monitor costs and selectivities *continuously*
- Reoptimize *periodically* using KBZ

Can use specialized policies for correlated predicates

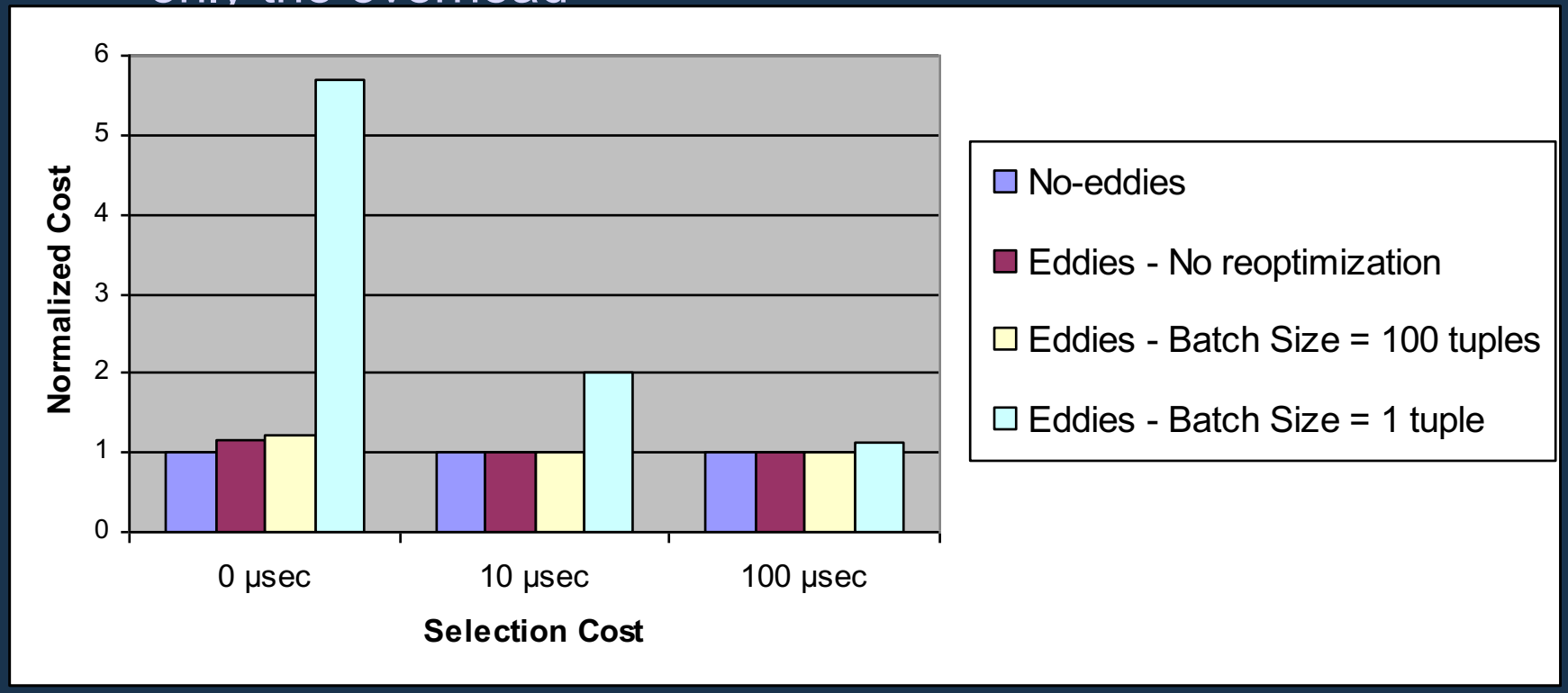
Statistics Maintained:
Costs of operators
Selectivities of operators

Routing policy:
Use a single order for a batch of tuples
Periodically apply KBZ



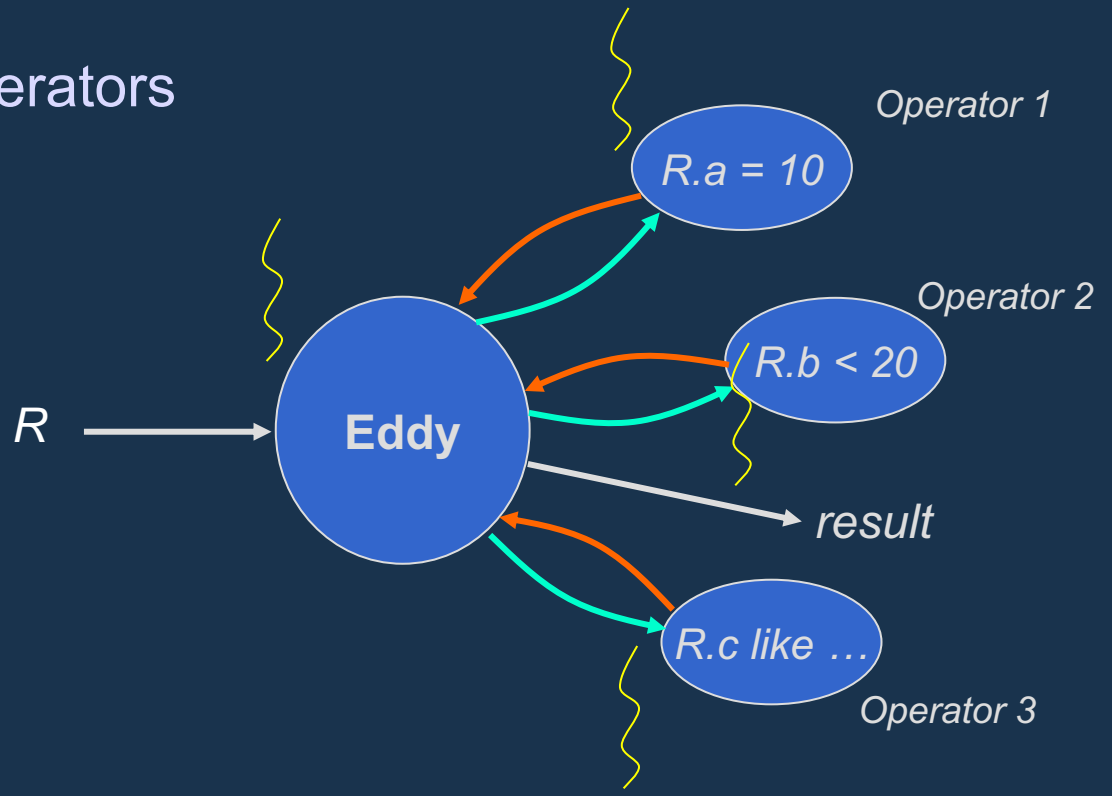
Overhead of Routing and Reoptimization

- Adaptation using *batching*
  - Reoptimized every *X* tuples using monitored selectivities
  - Identical selectivities throughout ➔ experiment measures only the overhead



Routing Policy 3: Lottery Scheduling

- Originally suggested routing policy [AH'00]
- Applicable only if each operator runs in a separate thread
- Uses two easily obtainable pieces of information for making routing decisions:
  - *Busy/idle status* of operators
  - *Tickets* per operator

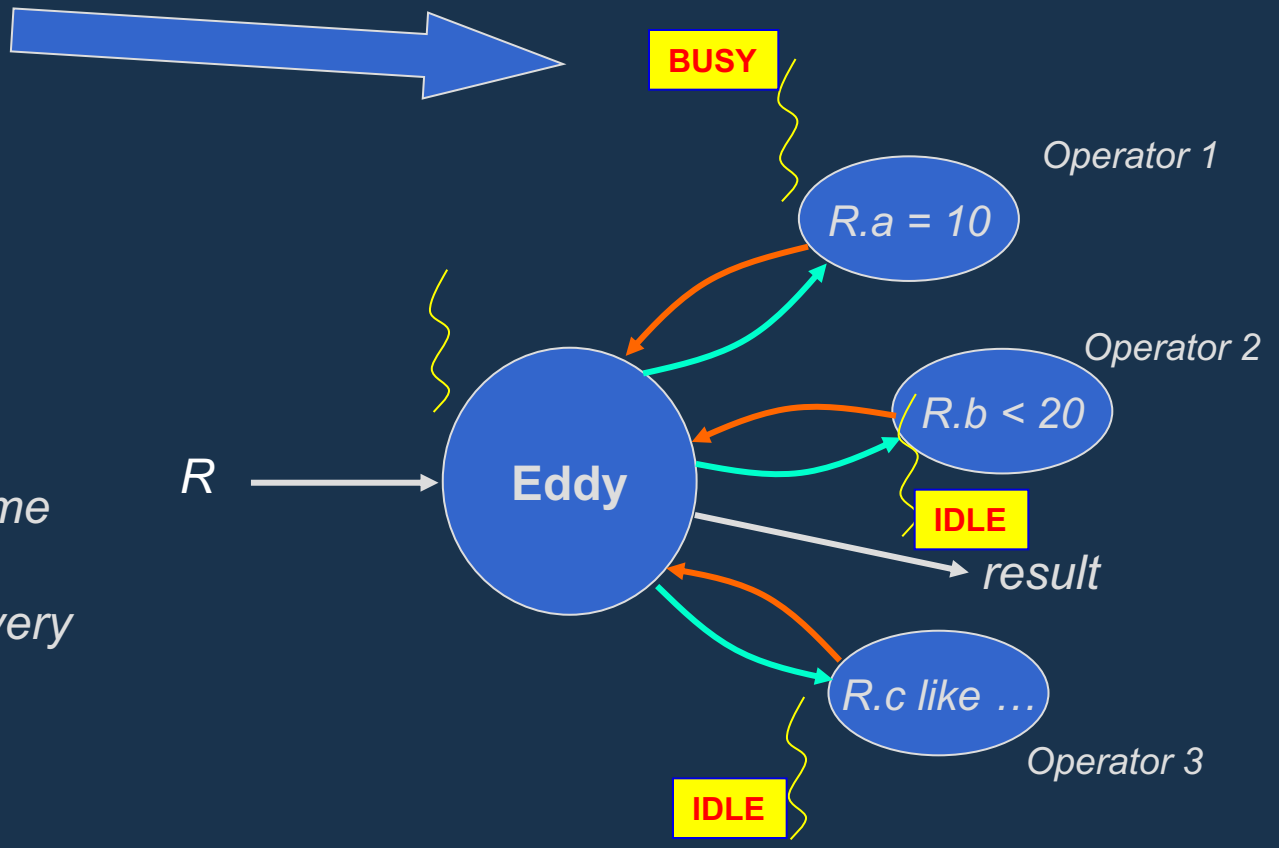


Routing Policy 3: Lottery Scheduling

- Routing decisions based on busy/idle status of operators

Rule:
IF operator busy,
THEN do not route more tuples to it

Rationale:
Every thread gets equal time
SO IF an operator is busy,
THEN its cost is perhaps very high

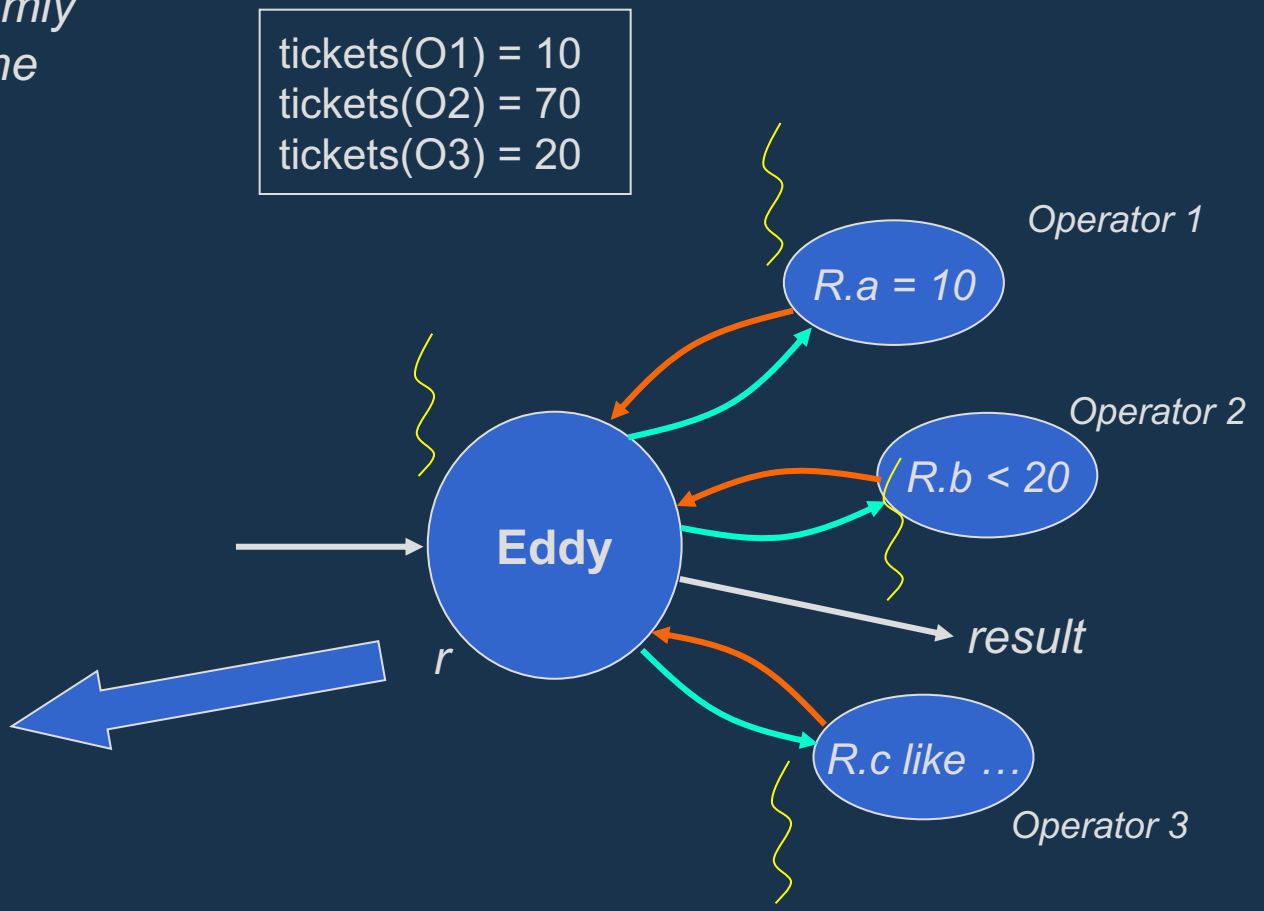


Routing Policy 3: Lottery Scheduling

- Routing decisions based on tickets

Rules:

1. *Route a new tuple randomly weighted according to the number of tickets*



Will be routed to:

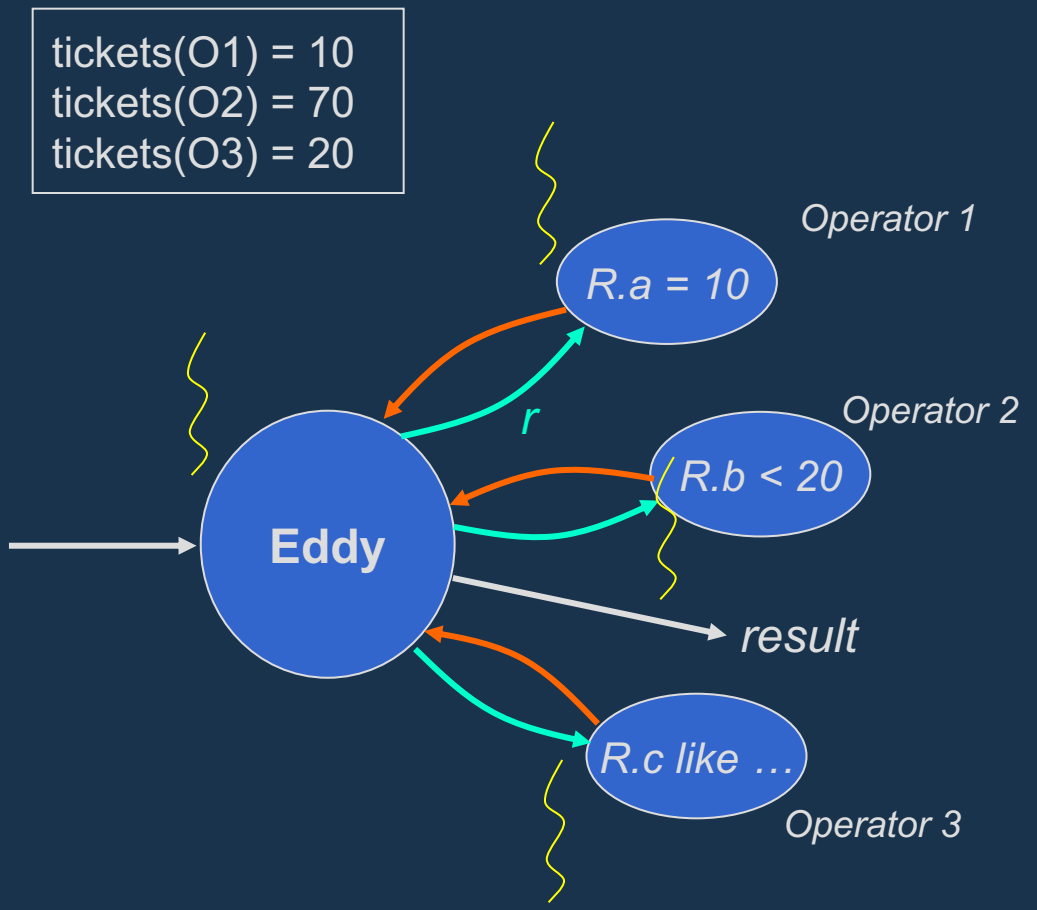
O1 w.p. 0.1
O2 w.p. 0.7
O3 w.p. 0.2

Routing Policy 3: Lottery Scheduling

- Routing decisions based on tickets

Rules:

1. *Route a new tuple randomly weighted according to the number of tickets*

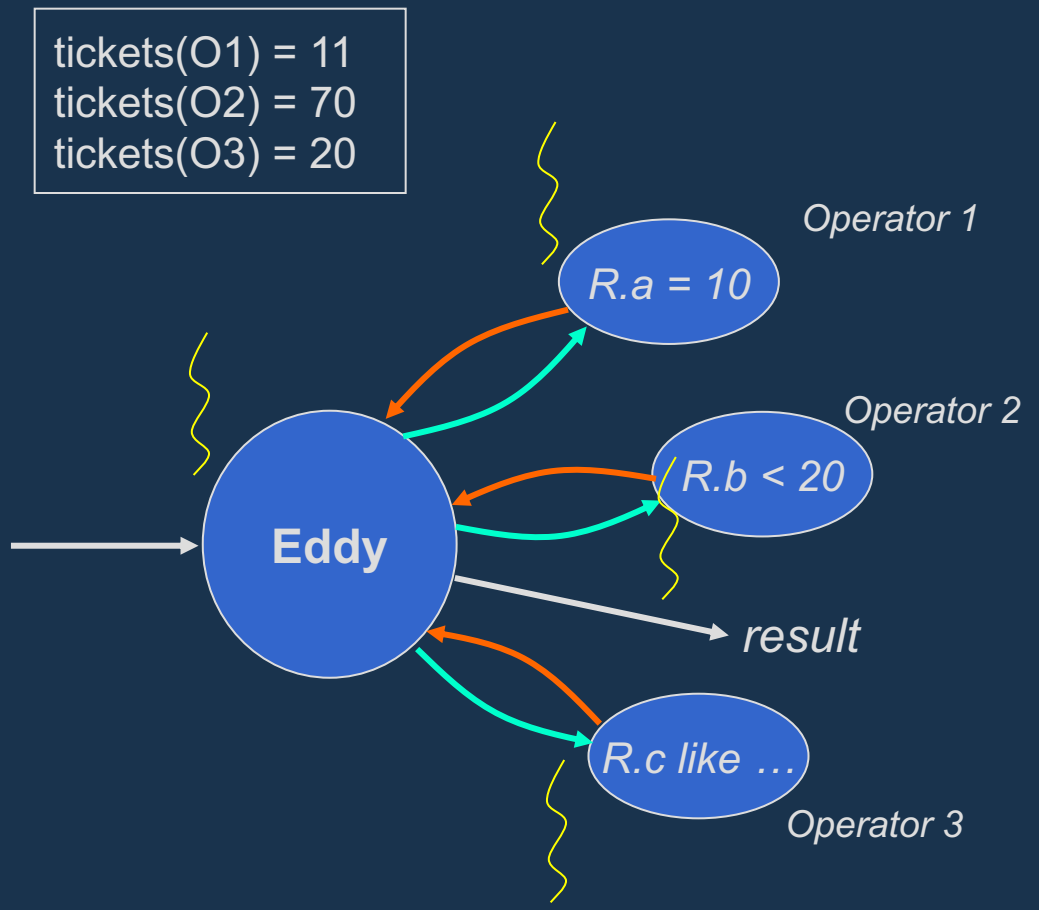


Routing Policy 3: Lottery Scheduling

- Routing decisions based on tickets

Rules:

1. *Route a new tuple randomly weighted according to the number of tickets*
2. *route a tuple to an operator O_i tickets(O_i) ++;*

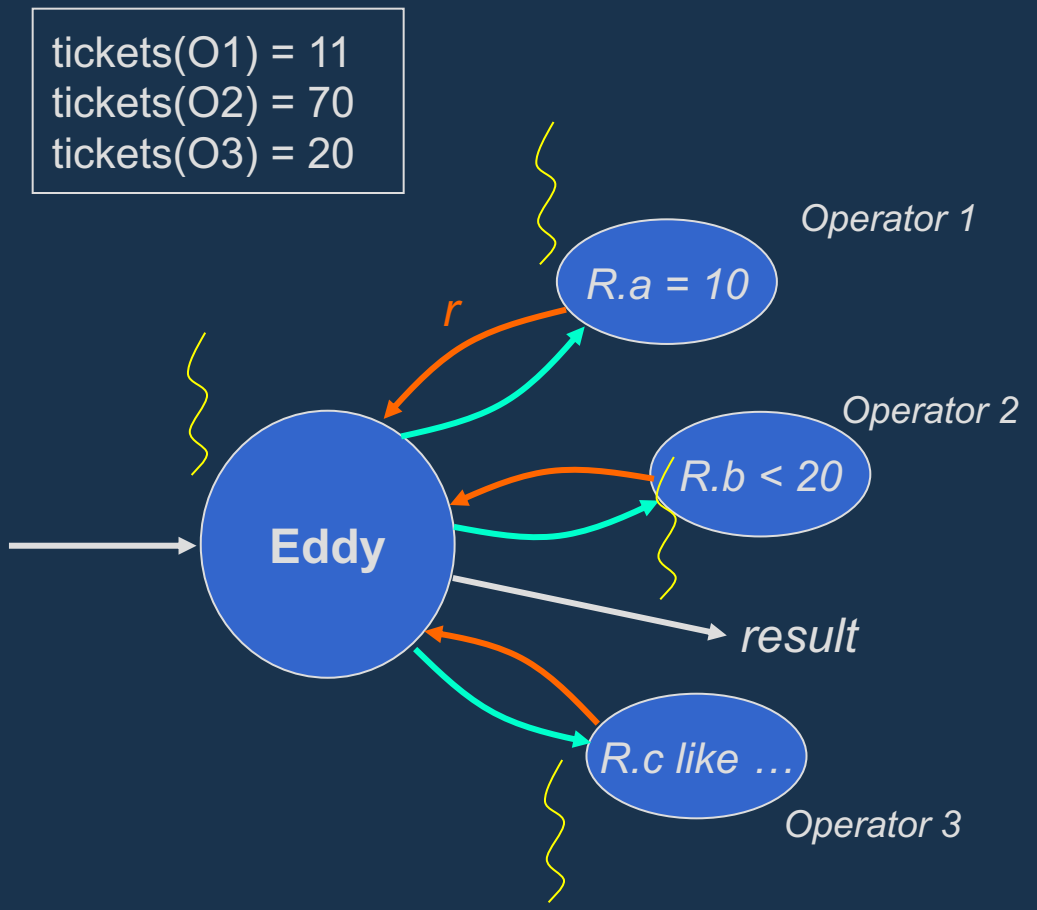


Routing Policy 3: Lottery Scheduling

- Routing decisions based on tickets

Rules:

1. *Route a new tuple randomly weighted according to the number of tickets*
2. *route a tuple to an operator O_i*
   tickets(O_i) ++;
3. *O_i returns a tuple to eddy*
   tickets(O_i) --;

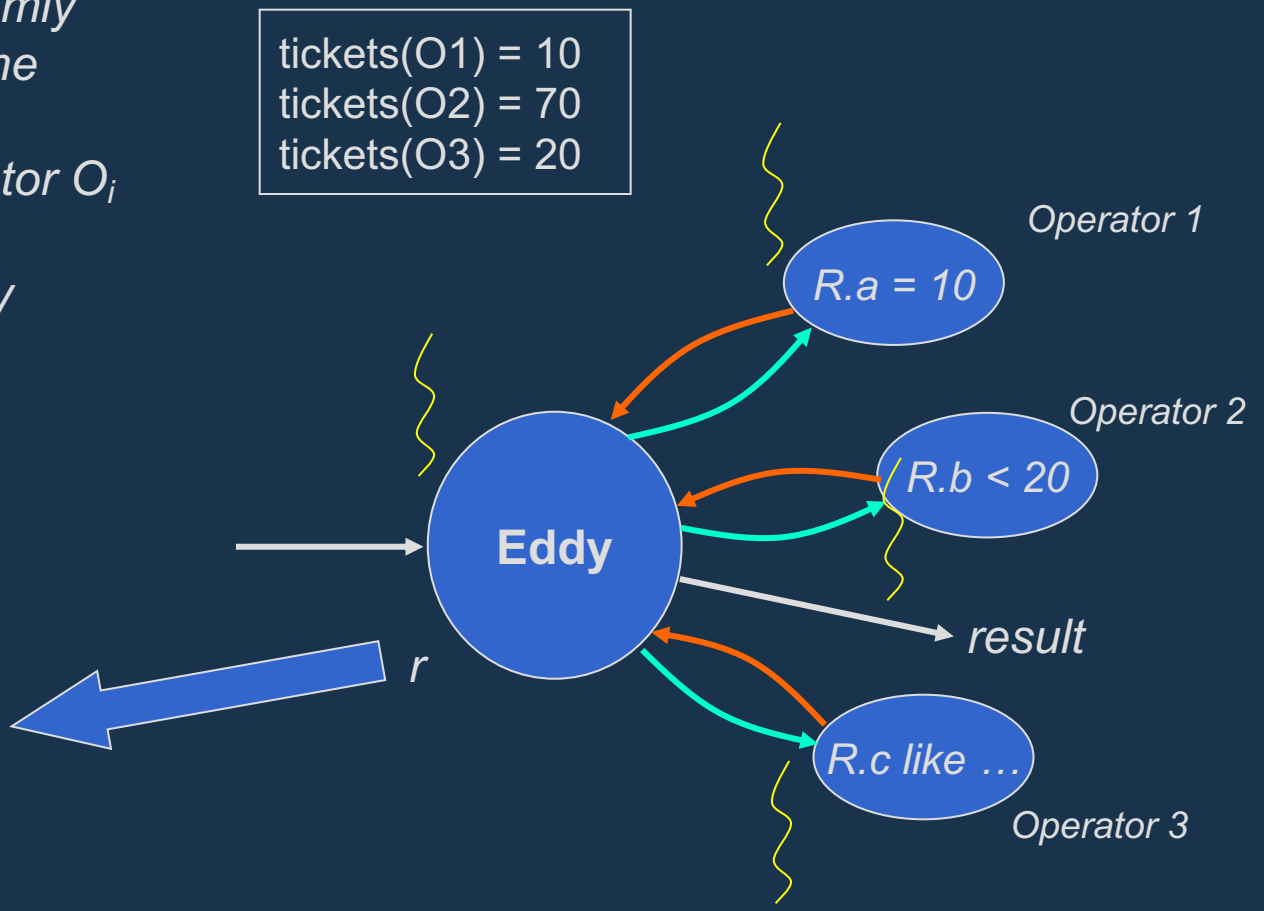


Routing Policy 3: Lottery Scheduling

- Routing decisions based on tickets

Rules:

1. *Route a new tuple randomly weighted according to the number of tickets*
2. *route a tuple to an operator O_i tickets(O_i) ++;*
3. *O_i returns a tuple to eddy tickets(O_i) --;*



Will be routed to:

O2 w.p. 0.777
O3 w.p. 0.222

Routing Policy 3: Lottery Scheduling

- Routing decisions based on tickets

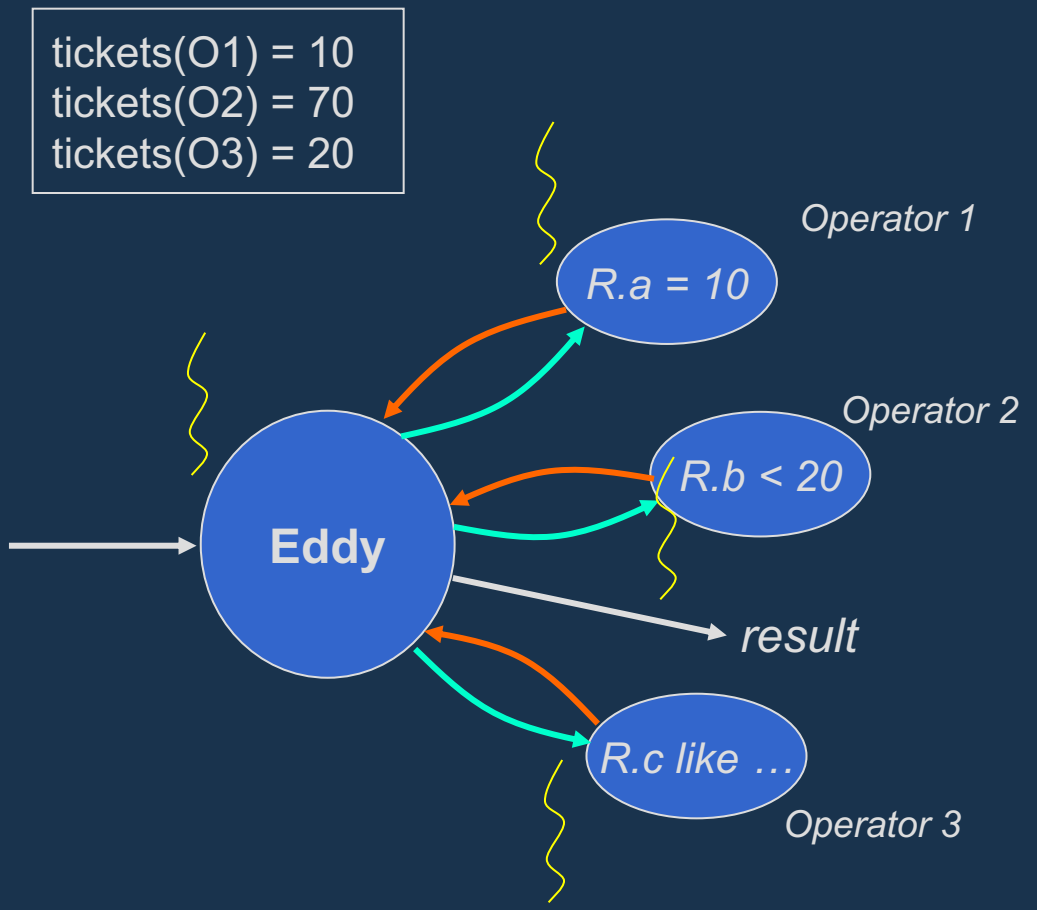
Rules:

1. *Route a new tuple randomly weighted according to the number of tickets*
2. *route a tuple to an operator O_i tickets(O_i) ++;*
3. *O_i returns a tuple to eddy tickets(O_i) --;*

Rationale:

Tickets(O_i) roughly corresponds to (1 - selectivity(O_i))

So more tuples are routed to highly selective operators

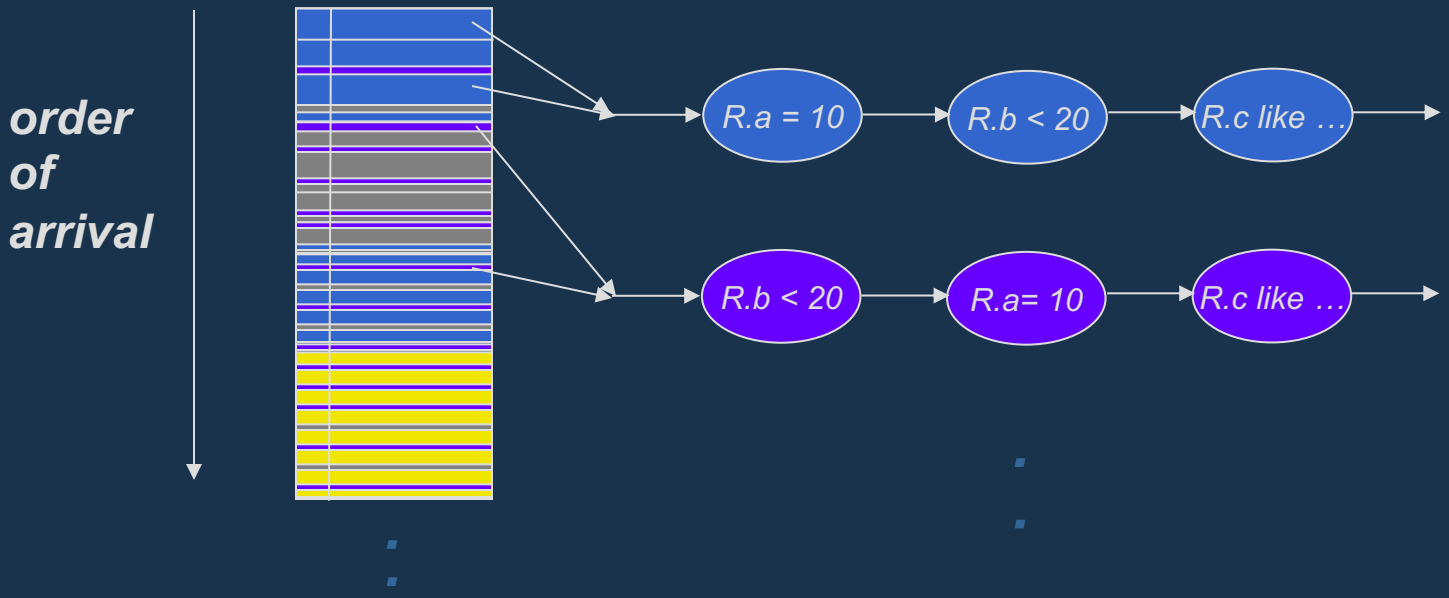


Routing Policy 3: Lottery Scheduling

- Effect of the combined lottery scheduling policy:
  - Low cost operators get more tuples
  - Highly selective operators get more tuples
  - Some tuples are knowingly routed according to sub-optimal orders
    - To *explore*
    - Necessary to detect selectivity changes over time

Eddies: Post-Mortem

- Plan Space explored
  - Allows arbitrary "*horizontal partitioning*"
  - Not necessarily correlated with order of arrival



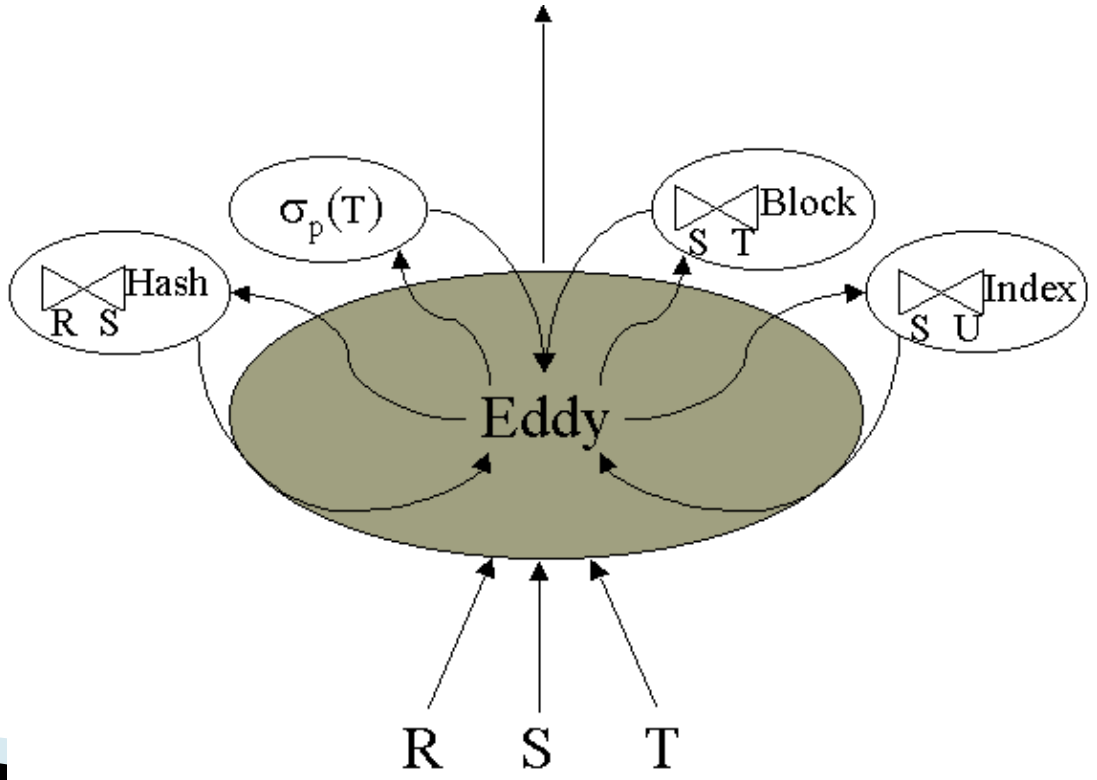
In a later paper, we looked at optimizing for horizontal partitioning directly

Outline

- Query evaluation techniques for large databases, Skew Avoidance, Query compilation/vectorization
- Query Optimization: Overview, How good are the query optimizers, really?, Reordering for Outerjoins, Query Rewriting
- Adaptive Query Processing
  - Eddies
  - Progressive Query Optimization
  - Compilation and adaptivity

Overview

- Continuously ”reorder” operators as the query is executing
  - By changing the “order” in which tuples visit operators
  - Obviate the need for selectivity estimation and optimization entirely
  - Naturally handles situations where the selectivities change over time (for long-running queries)

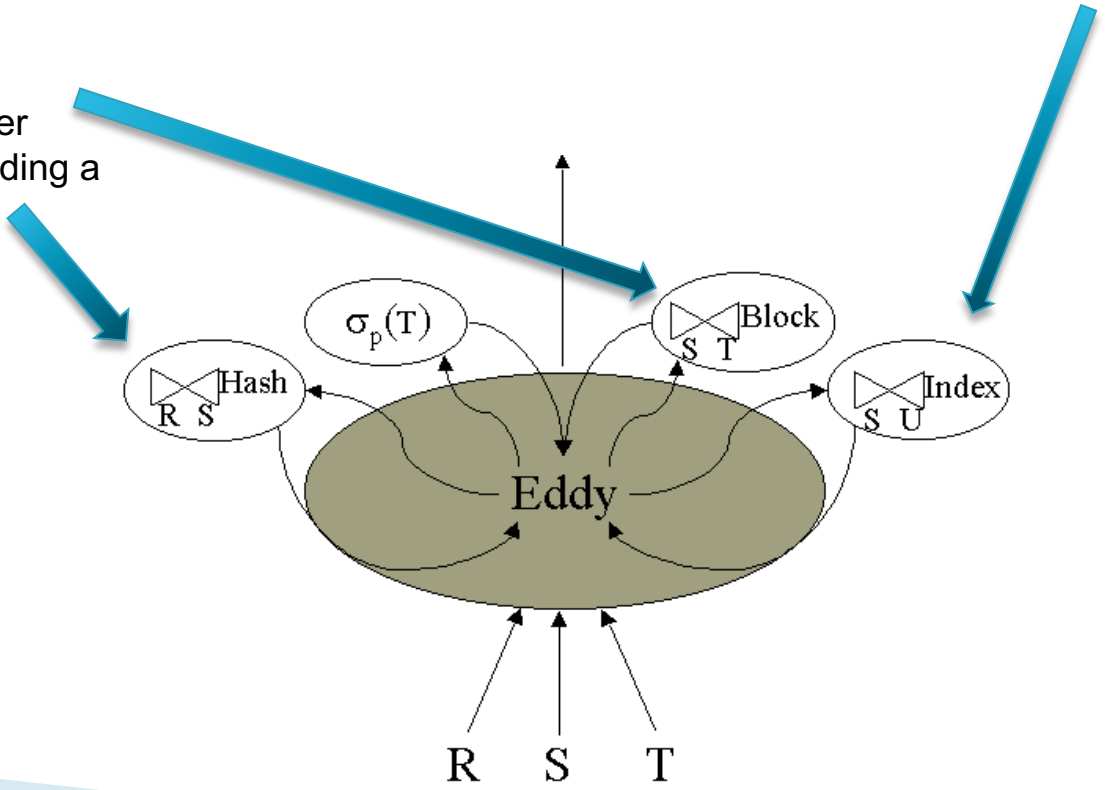


Eddies and Joins

- Selections are arbitrarily reorderable
- What about joins?

- An index lookup can be treated as a “selection”
- Send an S tuple, get back augmented tuples
- Note: decision to use the index cannot be “adapted”

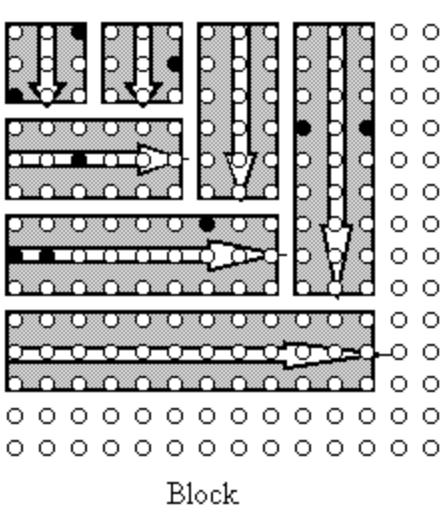
- These two are tricky
- Nested loops requires iterating over all of inner
- Hash join requires building a hash table on inner

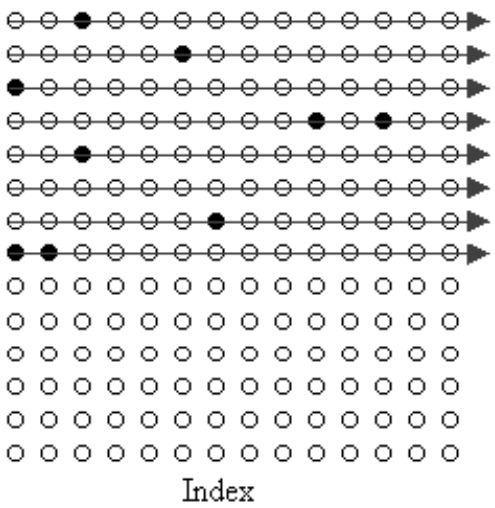


Reorderability of Plans

- Synchronization Barriers
  - Many operators explicitly enforce an order in which tuples must be read from the inputs
  - e.g., Sort-merge joins: at most points, the next tuple to read must be read from a specific input
  - Hash joins: need to read all of ”inner” before outer tuples can be read
- Moments of Symmetry
  - Sort-merge join is symmetric
  - But Nested-loops is not
    - However, can change the outer/inner at specific points
- Join operators with more moments of symmetric preferred
  - e.g., Symmetric Hash Join Operator

Reorderability of Plans





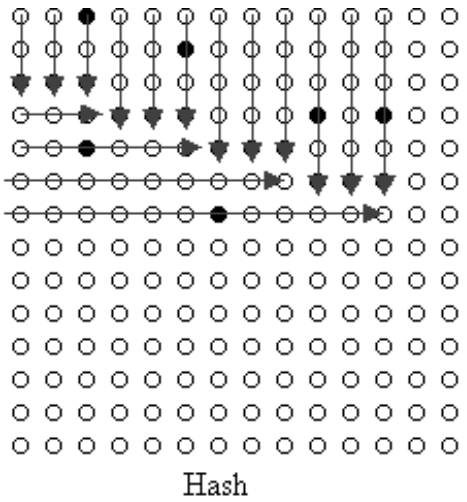


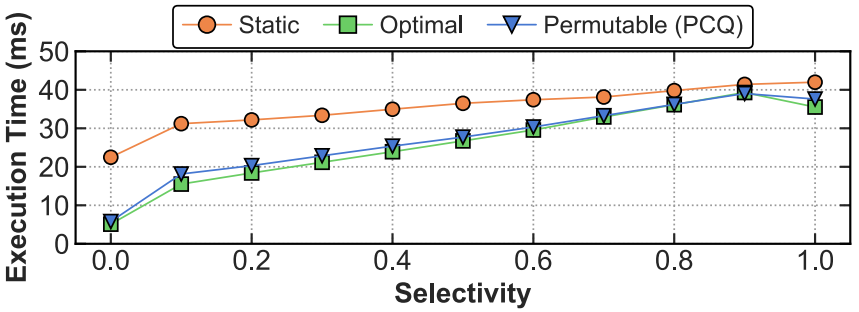
Figure 3: Tuples generated by block, index, and hash ripple join. In block ripple, all tuples are generated by the join, but some may be eliminated by the join predicate. The arrows for index and hash ripple join represent the *logical* portion of the cross-product space checked so far; these joins only expend work on tuples satisfying the join predicate (black dots). In the hash ripple diagram, one relation arrives 3× faster than the other.

Eddies

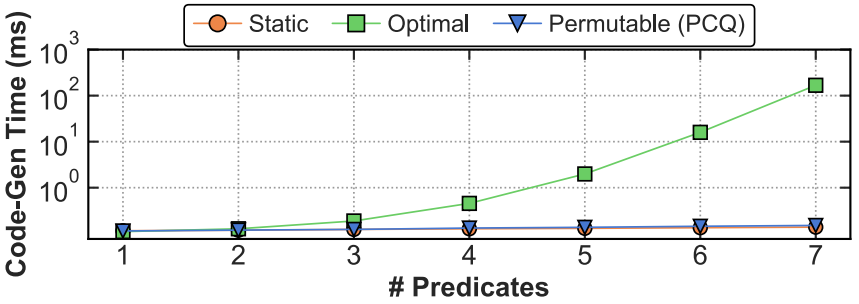
- Implemented in the context of River project
- Eddy is a separate module that talks to all other operators
  - Uses “ready” and “done” bitsets to direct traffic
- Lottery scheduling-based routing policy
  - Promising initial results, but bunch of caveats

Motivation

- Adaptive query processing (POP-style) works well with interpretable query plans, but not as well with compilation
  - Compiling a new query plan too expensive



(a) Execution Time



(b) Code-Generation Time

Figure 1: Reoptimizing Compiled Queries – PCQ enables near-optimal execution through adaptivity with minimal compilation overhead.

Permutable Compiled Queries (PCQ)

- Adaptive query processing (POP-style) works well with interpretable query plans, but not as well with compilation
  - Compiling a new query plan too expensive

- Instead:
  - Precompile a bunch of different plans at optimization time itself
  - Add indirections to the compiled code to make it easy to switch/permute operators
  - Add hooks for collecting runtime performance metrics
    - To be used to decide whether to switch

Permutable Compiled Queries (PCQ)

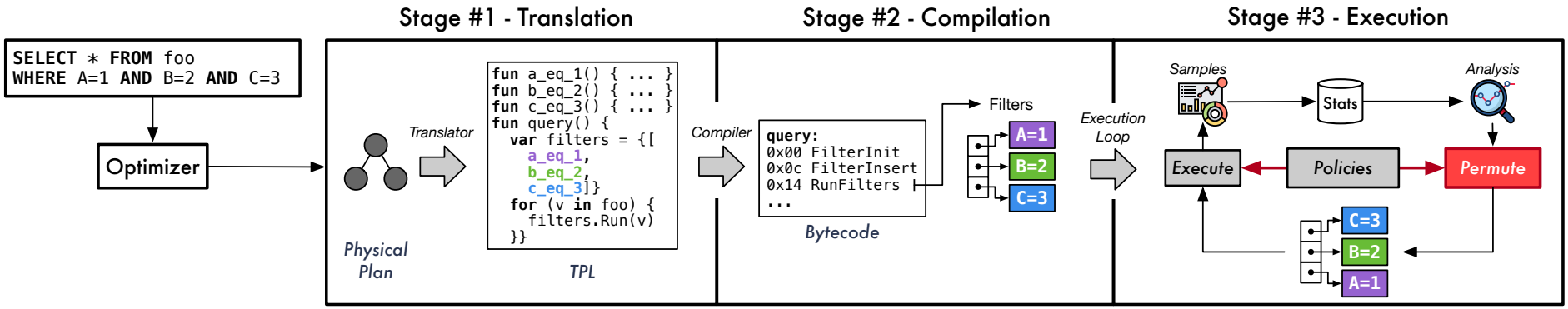


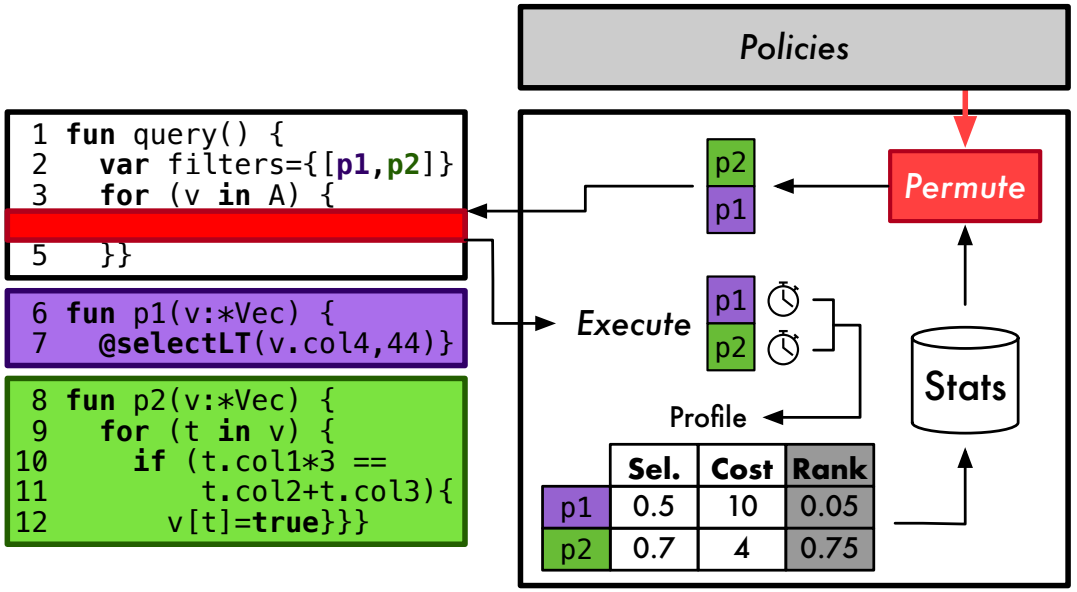
Figure 2: System Overview – The DBMS translates the SQL query into a DSL that contains indirection layers to enable permutability. Next, the system compiles the DSL into a compact bytecode representation. Lastly, an interpreter executes the bytecode. During execution, the DBMS collects statistics for each predicate, analyzes this information, and permutes the ordering to improve performance.

Adaptive Filter Ordering

Vectorization effect???
The code suggests filters applied to all tuples, so no point in reordering

```
SELECT * FROM A WHERE col1 * 3 = col2 + col3 AND col4 < 44
```

(a) Example Input SQL Query



(b) Generated Code and Execution of Permutable Filter

Figure 3: Filter Reordering – The Translator converts the query in (a) into the TPL on the left side of (b). This program uses a data structure template with query-specific filter logic for each filter clause. The right side of (b) shows how the policy collects metrics and then permutes the ordering.

Adaptive Aggregations

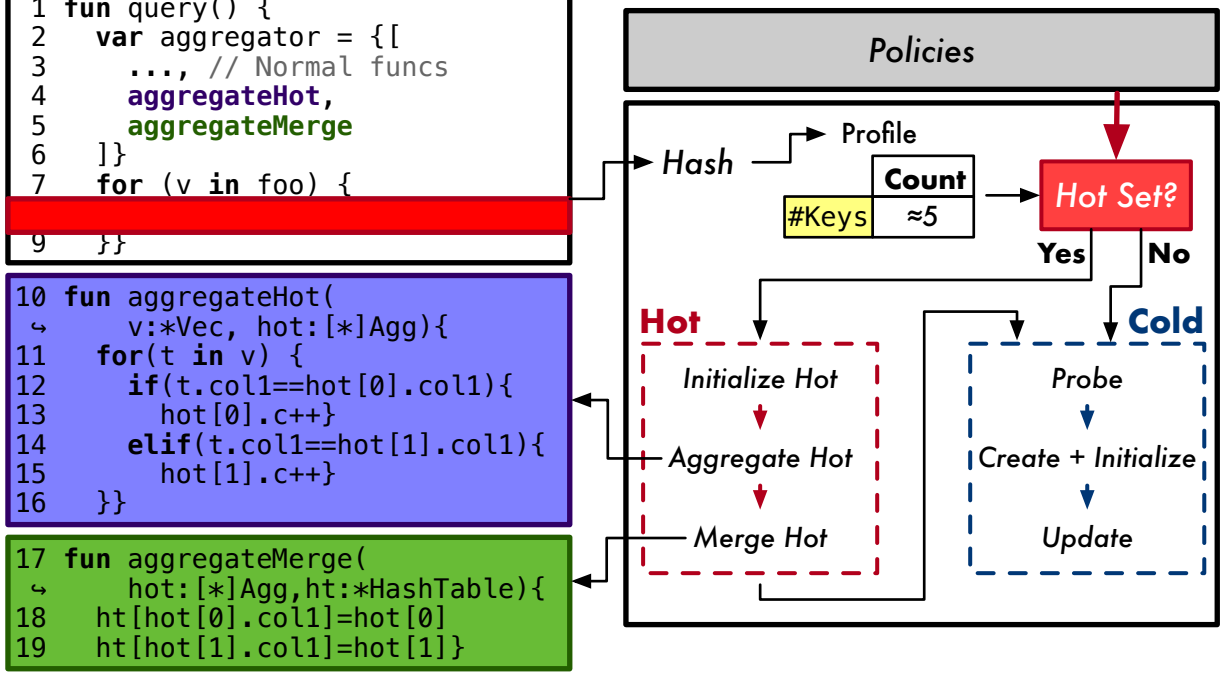
```
SELECT col1, COUNT(*) FROM A GROUP BY col1
```

(a) Example Input SQL Query

```
1  fun query() {
2    var aggregator = {[
3      ..., // Normal funcs
4      aggregateHot,
5      aggregateMerge
6    ]}
7    for (v in foo) {
8      aggregator.aggregate(v)
9    }}
```

```
10 fun aggregateHot(
 ↪     v:*Vec, hot:[*]Agg){
11   for(t in v) {
12     if(t.col1==hot[0].col1){
13       hot[0].c++}
14     elif(t.col1==hot[1].col1){
15       hot[1].c++}
16   }}
```

```
17 fun aggregateMerge(
 ↪     hot:[*]Agg,ht:*HashTable){
18   ht[hot[0].col1]=hot[0]
19   ht[hot[1].col1]=hot[1]}
```



(b) Generated Code and Execution of Adaptive Aggregation

Figure 4: Adaptive Aggregations – The input query in (a) is translated into TPL on the left side of (b). The right side of (b) steps through one execution of PCQ aggregation.

Adaptive Joins

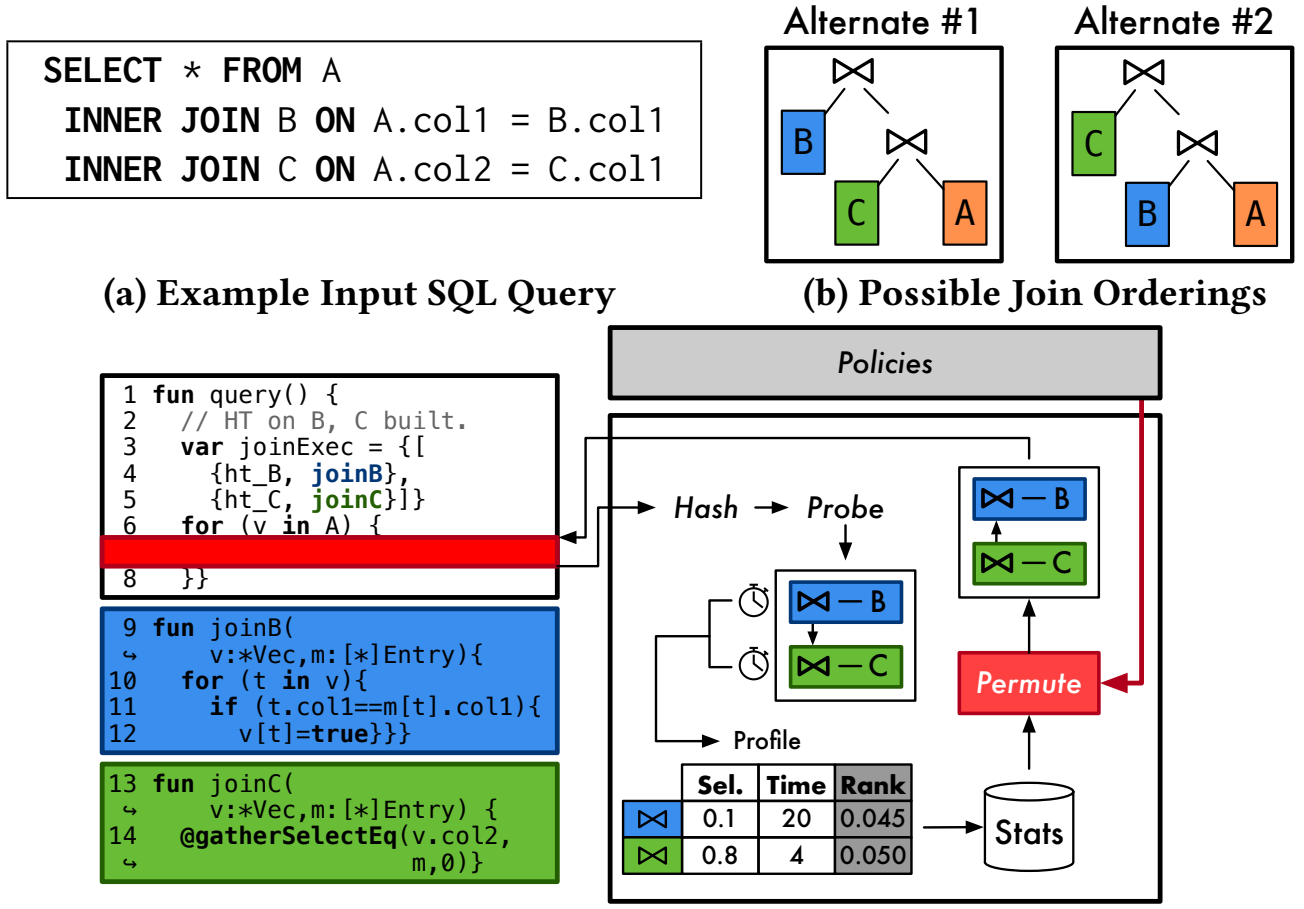


Figure 5: Adaptive Joins – The DBMS translates the query in (a) to the program in (c). The right side of (c) illustrates one execution of a permutable join that includes a metric collection step.

Experimental Evaluation

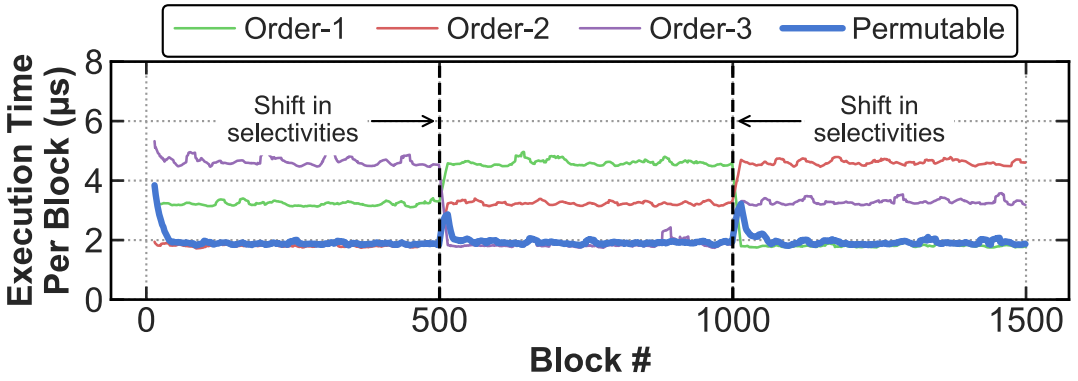


Figure 6: Performance Over Time – Execution time of three static filter orderings and our PCQ filter during a sequential table scan.

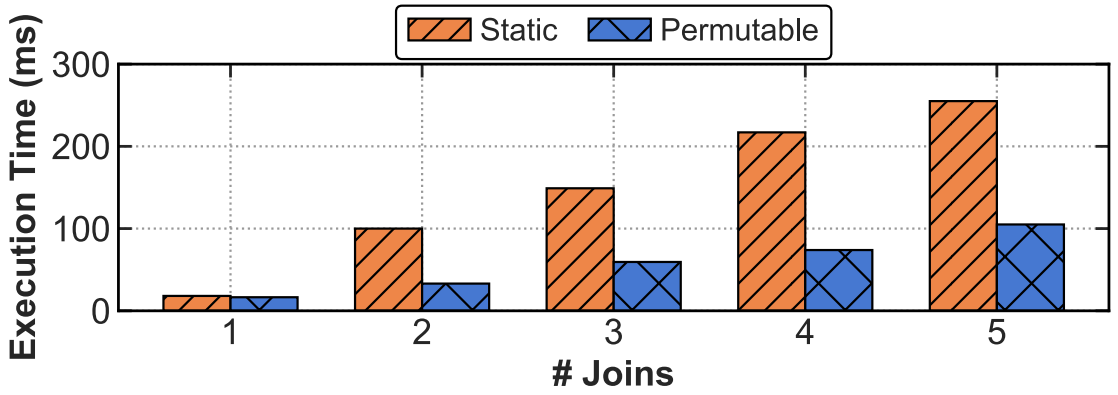


Figure 12: Varying Number of Joins – Execution time to perform a multi-step join while keeping the overall join selectivity at 10%.