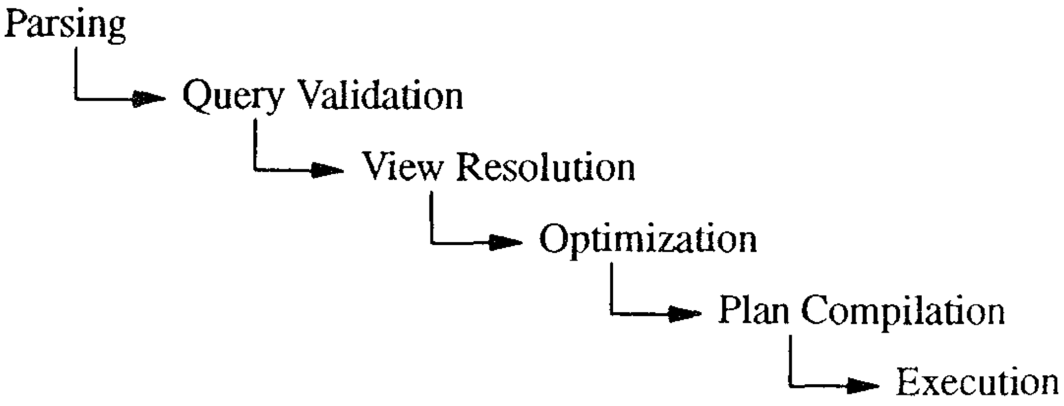
CMSC 724: Database Management Systems
Query Processing and Optimization

Instructor: Amol Deshpande
amol@cs.umd.edu

Outline

- Query evaluation techniques for large databases
- Skew avoidance strategies
- Query compilation
- Vectorization

Basics of Query Processing



Parsing

Query Validation

View Resolution

Optimization

Plan Compilation

Execution

Figure 2. Query processing steps.

Update queries usually handled through “deferred updates” (use standard read-only techniques to identify the modifications, and apply them afterwards.

Architectural Issues

- Logical algebra vs physical algebra
  - Latter is system-specific, and refers to the specific implementations of operators
  - Mapping from logical to physical operators is often not one-to-one
    - Most operator implementations usually handle subsequent selects and projects
    - A single logical operator may be broken up into multiple physical ones (e.g., “sort” is done separately from “merge” for ”sort-merge join”)
    - A “symmetric” logical operator may be implemented by an “asymmetric” physical operator

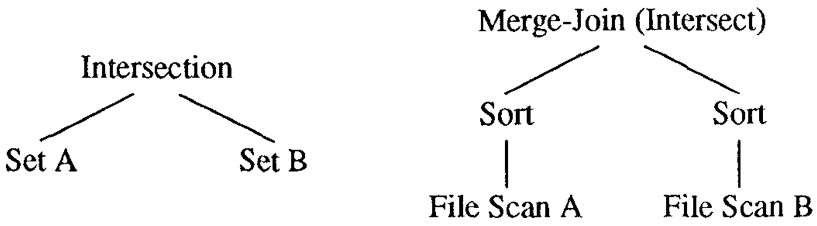
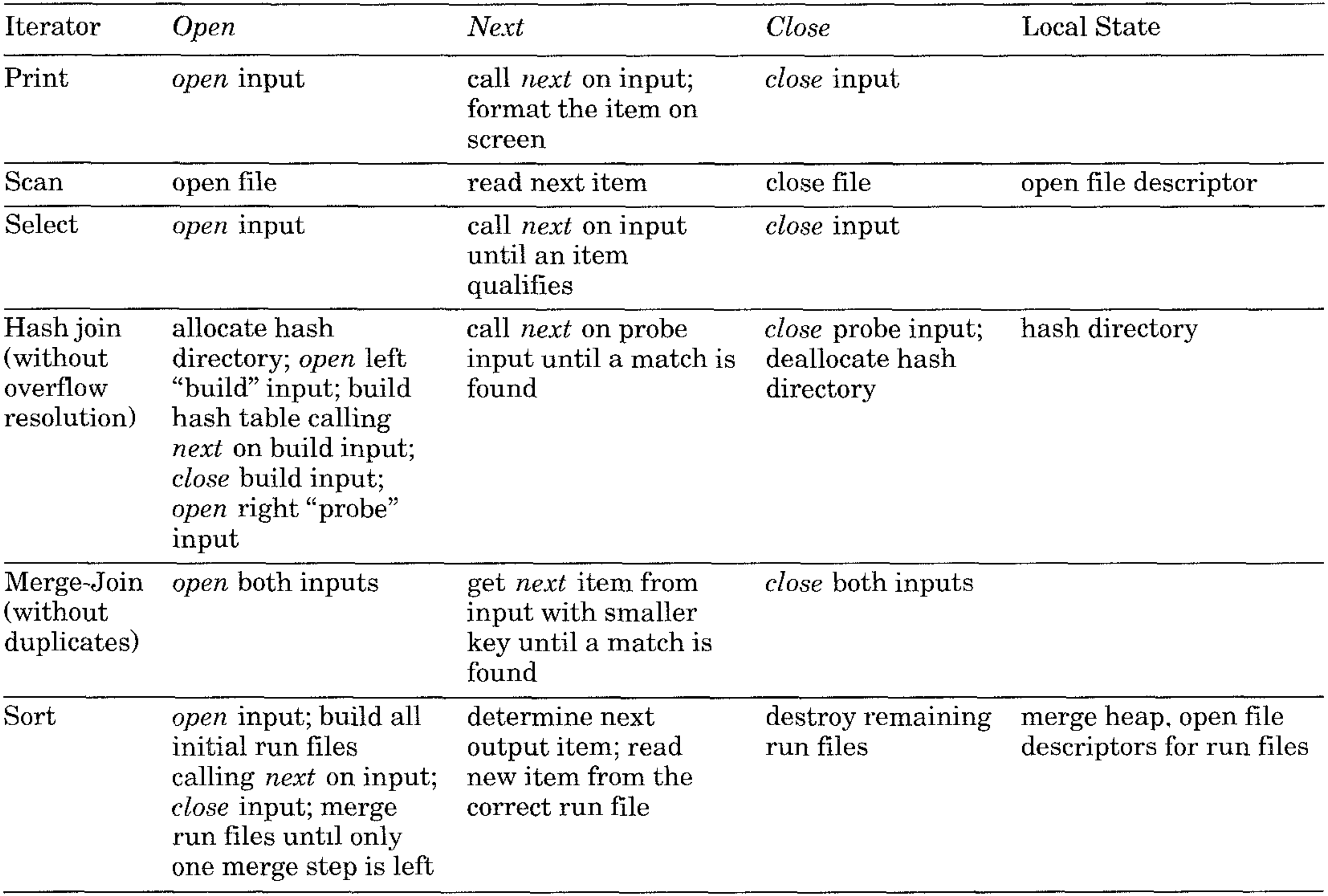


Figure 3. Logical and physical algebra expressions.

How to pass tuples between operators

- Materialization: Write out the results to a file, and the next operator reads it from the file
- Pipelining: Have both (or more) operators running at the same time (e.g., in different threads or processes), and use queues to transfer tuples
  - Hard to make this work efficiently (e.g., OS may switch to an operator that has no inputs, leading to wasted context switches)
- Iterator model: Have operators ”schedule” each other
  - When an operator needs more inputs, it “calls” the child operator(s)
  - No IPC needed – these are function calls
  - For Query Processing, can separate the work of an operator into:
    - initialization (init())
    - produce the next tuple (next())
    - clean up (close())
  - Main drawback (as we discuss later): too many function calls for modern architectures

Examples of Iterator Functions



Iterator	*Open*	*Next*	*Close*	Local State
Print	*open* input	call *next* on input; format the item on screen	*close* input	
Scan	open file	read next item	close file	open file descriptor
Select	*open* input	call *next* on input until an item qualifies	*close* input	
Hash join (without overflow resolution)	allocate hash directory; *open* left "build" input; build hash table calling *next* on build input; *close* build input; *open* right "probe" input	call *next* on probe input until a match is found	*close* probe input; deallocate hash directory	hash directory
Merge-Join (without duplicates)	*open* both inputs	get *next* item from input with smaller key until a match is found	*close* both inputs	
Sort	*open* input; build all initial run files calling *next* on input; *close* input; merge run files until only one merge step is left	determine next output item; read new item from the correct run file	destroy remaining run files	merge heap, open file descriptors for run files

Iterator Model and Parallel Execution

- Can add in “exchange” operators (that follow iterator model) to parallelize a query plan

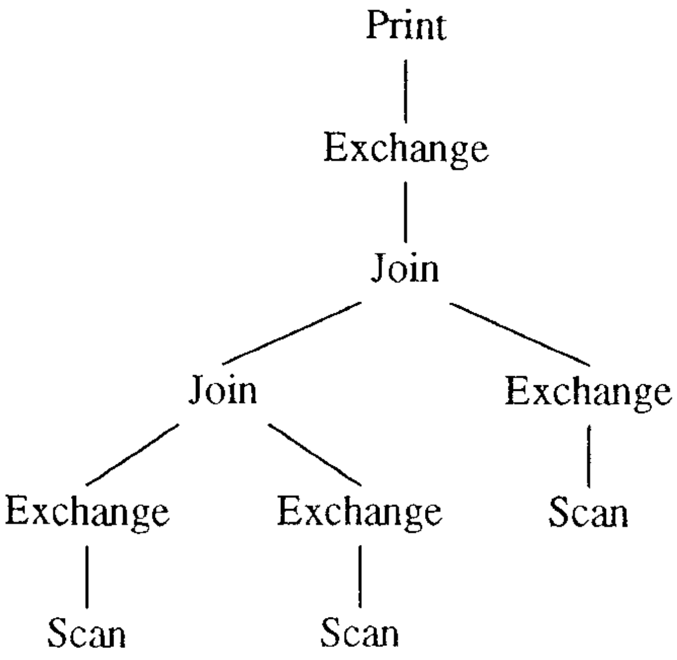


Figure 26. Operator model of parallelızation.

Types of Query Plans

- In some older papers, left-deep and right-deep are switched
  - Think of “left” as “outer” and “right” as “inner”
  - “Right-deep plans have only recently received more interest and may actually turn out to be very efficient, in particular in systems with ample memories” – refers to the ability to build many hash indexes at once, and today makes sense for “left-deep” plans

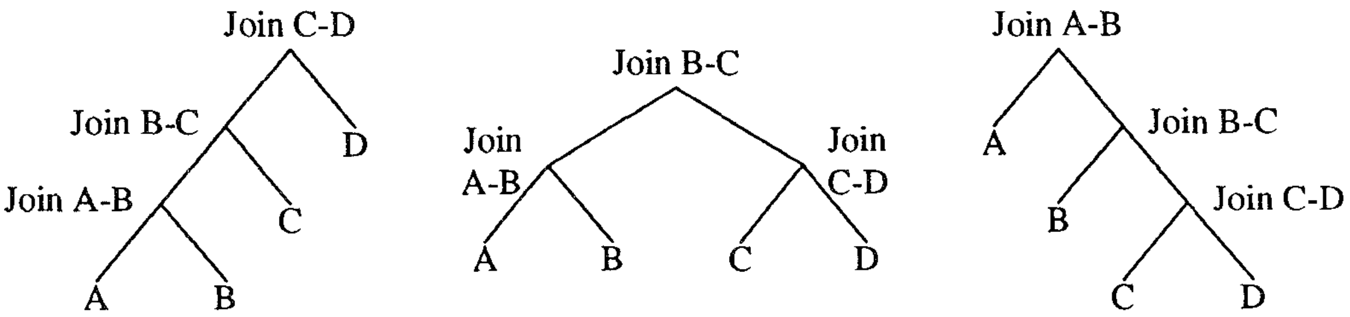
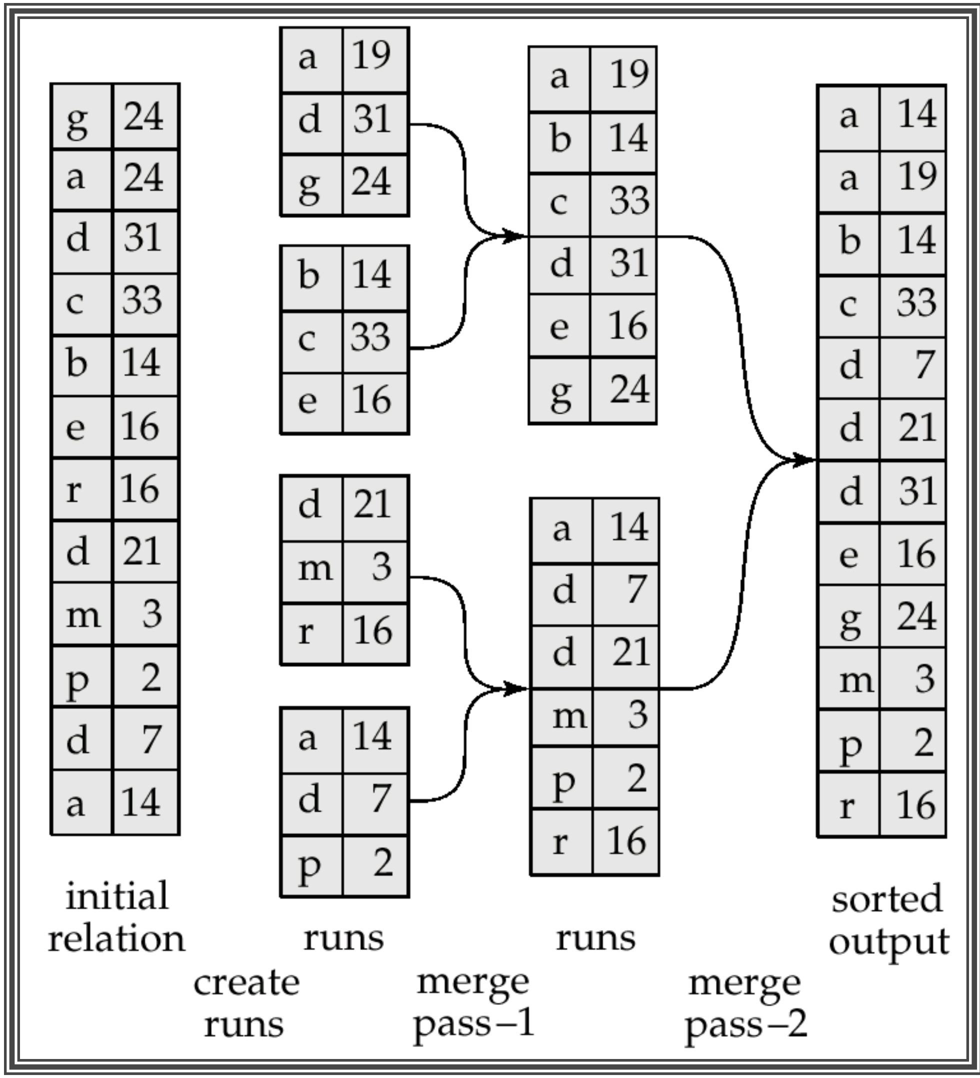


Figure 4. Left-deep, bushy, and right-deep plans.

- In general, may be a DAG (directed acyclic graph)
  - In case of common subexpressions

Sorting

- Volcano implementation:
  - open() does most of the work
    - If the input fits in memory, reads the entire input and does a quick-sort
    - If it doesn't fit in memory, uses external merge-sort except for the last merge
  - next() simply produces the tuples in the first case, and actually does the last merge in the second case
  - Probably better to do all the work in "next()" (with special-case code for the first call)



initial relation

g	24
a	24
d	31
c	33
b	14
e	16
r	16
d	21
m	3
p	2
d	7
a	14

create runs

runs

a	19
d	31
g	24

b	14
c	33
e	16

d	21
m	3
r	16

a	14
d	7
p	2

merge pass–1

runs

a	19
b	14
c	33
d	31
e	16
g	24

a	14
d	7
d	21
m	3
p	2
r	16

merge pass–2

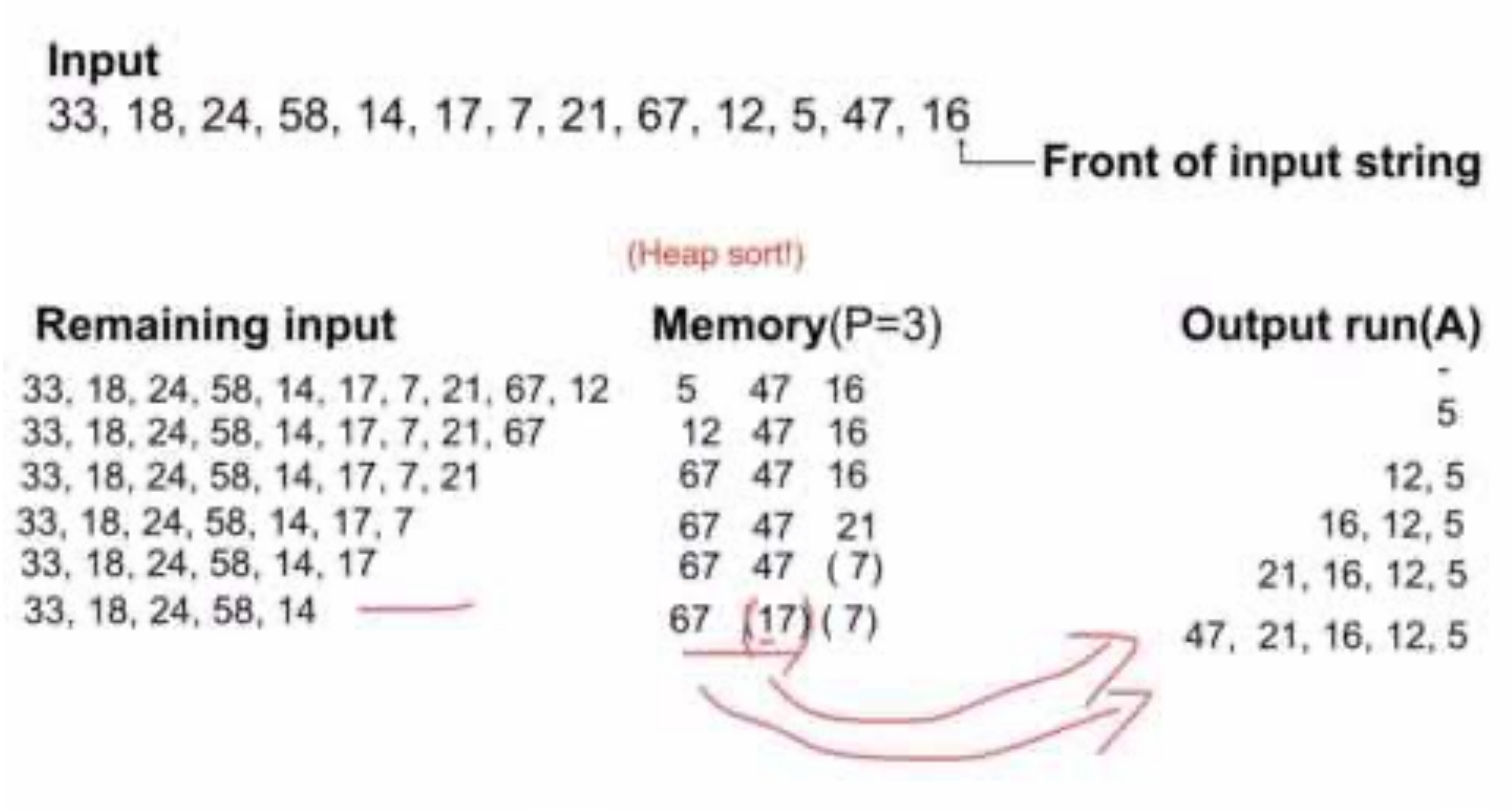
sorted output

a	14
a	19
b	14
c	33
d	7
d	21
d	31
e	16
g	24
m	3
p	2
r	16

Sorting: Creating the initial runs

- Say main memory = M blocks (of b tuples each)
- Option 1: Read M blocks at a time, quick-sort, and write out the “sorted run” to disk
  - Generates runs of size M
- Option 2: Replacement selection
  - Read M*b tuples in memory, and keep it (always) in sorted order
  - Write out the first tuple to disk as the first sorted run
  - Say the largest value written out so far is 1000
  - Read the next tuple from the original relation
    - If > 1000, add it to the same sorted run, and output the next tuple from that
    - If not, start (or add to) a second sorted run in memory
  - Keep doing this until the you the first sorted run in memory finishes is done (i.e., all new tuples get added to the second run)
  - Can use the Heap data structure to do this efficiently

Replacement Selection Example



Input

33, 18, 24, 58, 14, 17, 7, 21, 67, 12, 5, 47, 16

Front of input string

(Heap sort!)

Remaining input	**Memory**(P=3)	**Output run(A)**
33, 18, 24, 58, 14, 17, 7, 21, 67, 12	5 47 16	-
33, 18, 24, 58, 14, 17, 7, 21, 67	12 47 16	5
33, 18, 24, 58, 14, 17, 7, 21	67 47 16	12, 5
33, 18, 24, 58, 14, 17, 7	67 47 21	16, 12, 5
33, 18, 24, 58, 14, 17	67 47 (7)	21, 16, 12, 5
33, 18, 24, 58, 14	67 (17) (7)	47, 21, 16, 12, 5

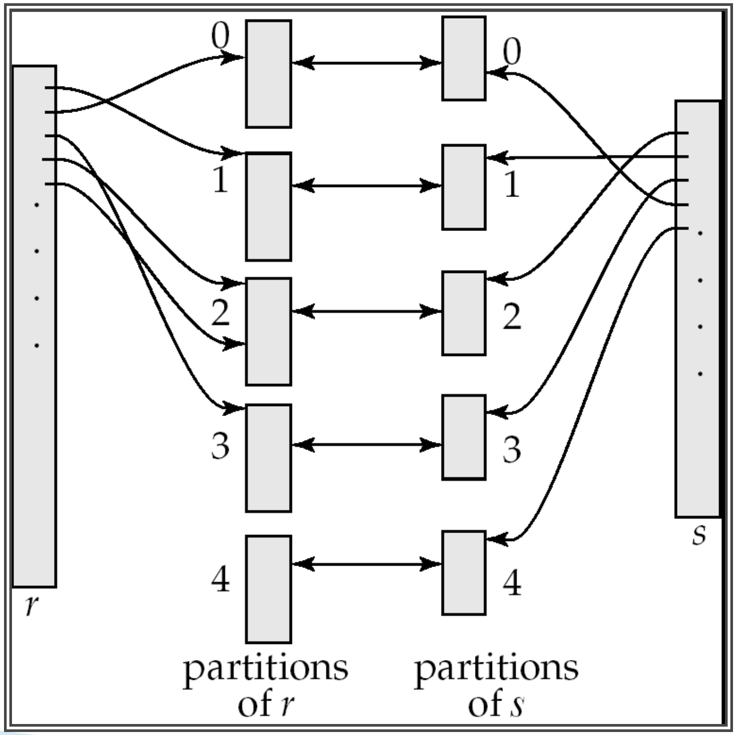
From: https://www.youtube.com/watch?v=LTpFZAd0cbE

Replacement Selection

- Need a data structure that efficiently supports removal of the smallest entry
  - The "heap" data structure works well
- Replacement selection results in larger runs → more efficient merge
  - If the input is already sorted or almost sorted, there is only one run
  - For random inputs, the runs are of size 2M
- But RS has more complex I/O patterns and there are other complications
  - Need to balance against the benefits of having fewer runs

Hashing

- Usually better when "equality matching" is required
- Basic idea:
  - "Build" a hash table on one of the inputs on the equality attribute(s)
  - "Probe" using the second input in any order
- What if the smaller input is too large?
  - Partition both the inputs using some criteria on the equality attribute (could be another hash function, or a range function)
  - Do partition-by-partition join

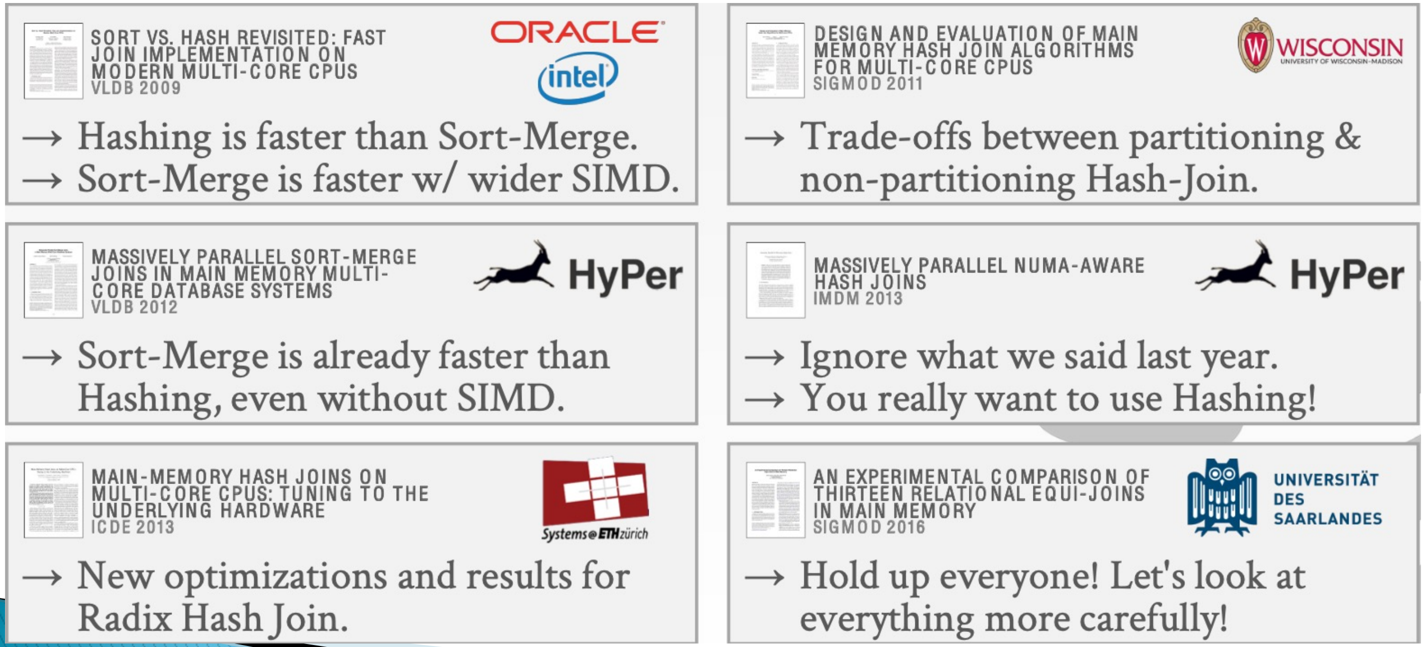


Hashing

- Usually better when “equality matching” is required
- Basic idea:
  - “Build” a hash table on one of the inputs on the equality attribute(s)
  - “Probe” using the second input in any order
- What if the smaller input is too large?
  - Partition both the inputs using some criteria on the equality attribute (could be another hash function, or a range function)
  - Do partition-by-partition join
- May need to do this “recursively”
  - Very unlikely to happen with today’s large memories
- Hybrid hash join
  - Keep one of the partitions in memory when doing the initial partitioning
  - Can be done in a reactive fashion
  - Works very well when the smaller input is just larger than memory

Sorting vs Hashing

- Most operators can be implemented using sorting or hashing
- Many papers written on which one is better
  - Depends a lot on the specific computing architecture
- Lot of recent work on multi-core sorting and hashing, and in shared-nothing settings



SORT VS. HASH REVISITED: FAST JOIN IMPLEMENTATION ON MODERN MULTI-CORE CPUS
VLDB 2009

ORACLE
intel

→ Hashing is faster than Sort-Merge.
→ Sort-Merge is faster w/ wider SIMD.

DESIGN AND EVALUATION OF MAIN MEMORY HASH JOIN ALGORITHMS FOR MULTI-CORE CPUS
SIGMOD 2011

WISCONSIN
UNIVERSITY OF WISCONSIN-MADISON

→ Trade-offs between partitioning & non-partitioning Hash-Join.

MASSIVELY PARALLEL SORT-MERGE JOINS IN MAIN MEMORY MULTI-CORE DATABASE SYSTEMS
VLDB 2012

→ Sort-Merge is already faster than Hashing, even without SIMD.

MASSIVELY PARALLEL NUMA-AWARE HASH JOINS
IMDM 2013

→ Ignore what we said last year.
→ You really want to use Hashing!

MAIN-MEMORY HASH JOINS ON MULTI-CORE CPUS: TUNING TO THE UNDERLYING HARDWARE
ICDE 2013

→ New optimizations and results for Radix Hash Join.

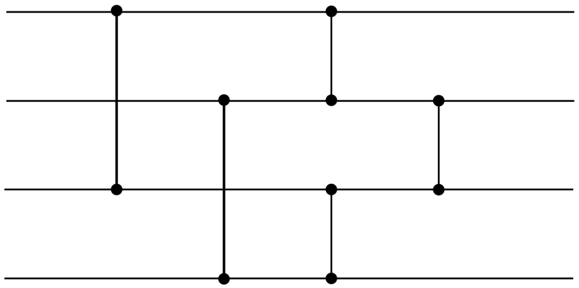
AN EXPERIMENTAL COMPARISON OF THIRTEEN RELATIONAL EQUI-JOINS IN MAIN MEMORY
SIGMOD 2016

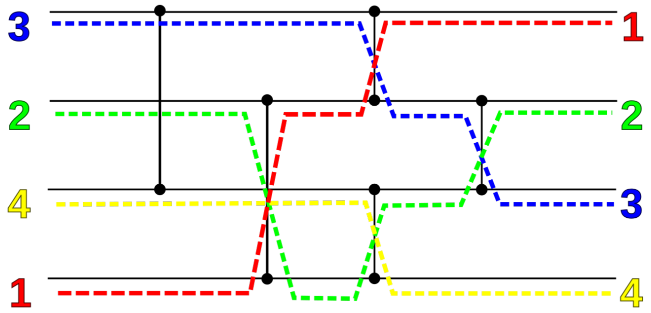
→ Hold up everyone! Let's look at everything more carefully!

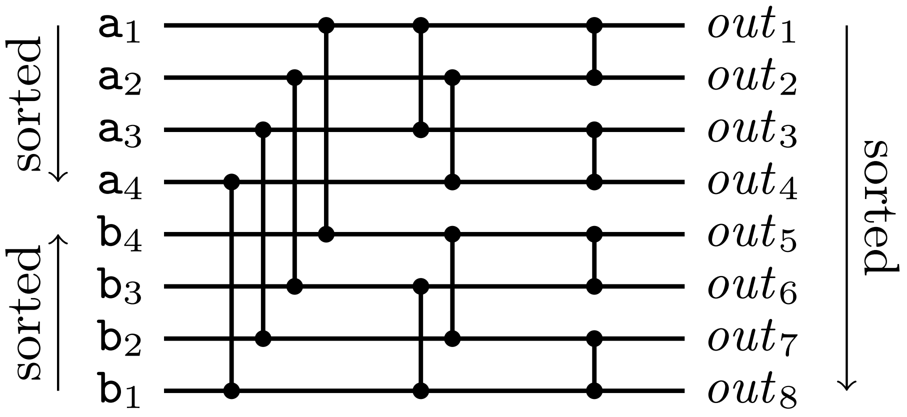
From Andy Pavlo's course slides

Multi-core: Sorting and Hashing

- Sorting and Bitonic Merge Networks
  - Fewer branches and more amenable to SIMD (vectorization)





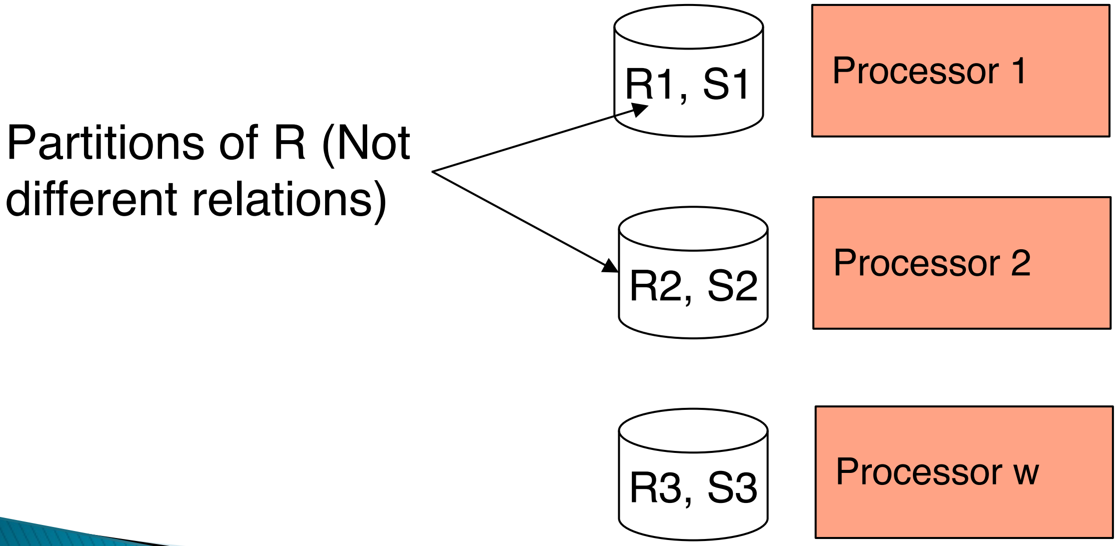


Outline

- Query evaluation techniques for large databases
- Skew avoidance strategies
- Query compilation
- Vectorization

Parallel Execution of Operators

- Assume shared-nothing model
- Relations are already partitioned across a set of machines



Partitions of R (Not different relations)

R1, S1

R2, S2

R3, S3

Processor 1

Processor 2

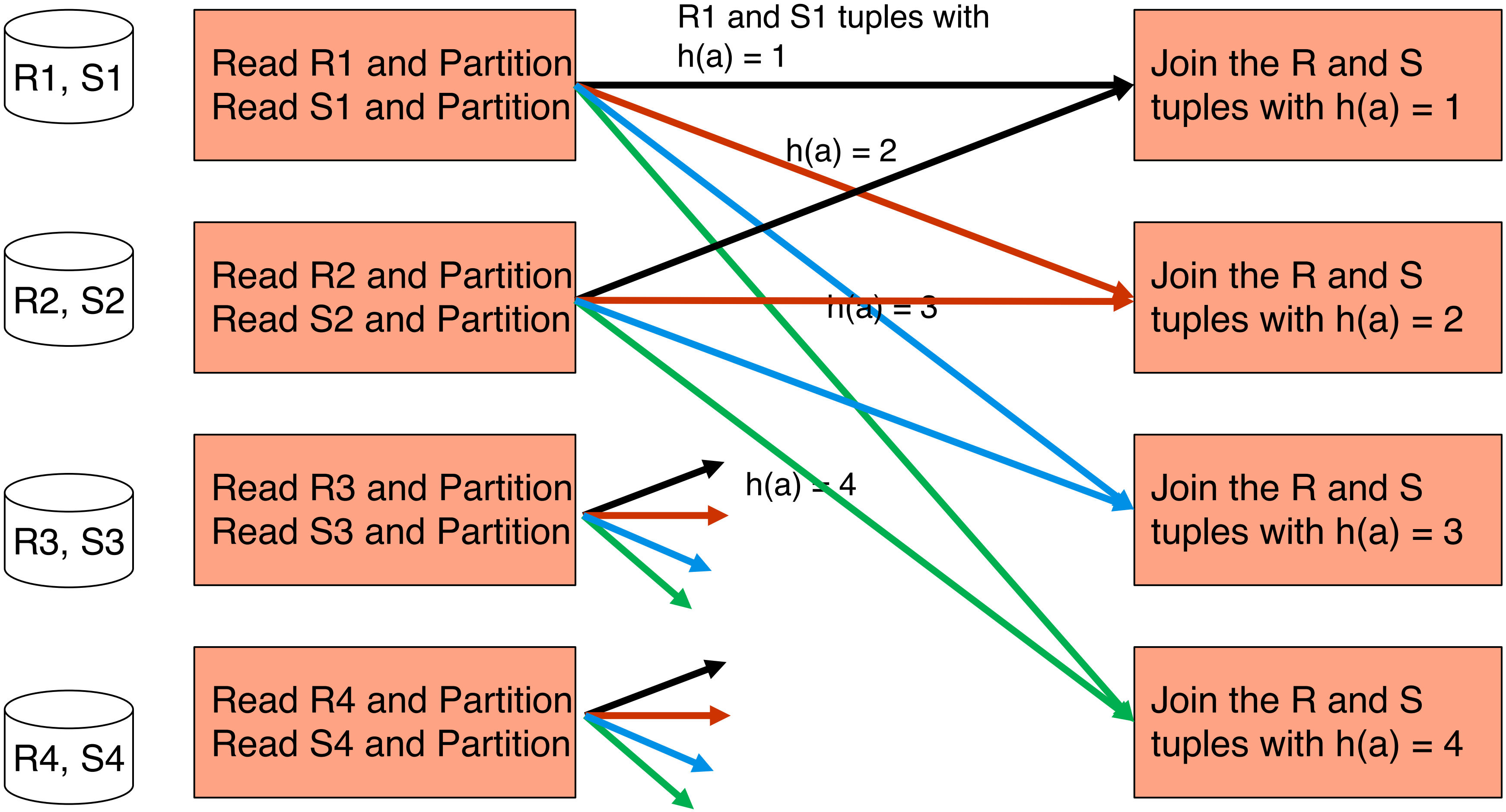
Processor w

Processor 1 can directly read R1, S1

If it wants R2, Processor 2 must read it and send it to Processor 1

Basic Parallel Hash Join

Can be same machines or different



R1, S1

R2, S2

R3, S3

R4, S4

Read R1 and Partition
Read S1 and Partition

Read R2 and Partition
Read S2 and Partition

Read R3 and Partition
Read S3 and Partition

Read R4 and Partition
Read S4 and Partition

R1 and S1 tuples with
h(a) = 1

h(a) = 2

h(a) = 3

h(a) = 4

Join the R and S
tuples with h(a) = 1

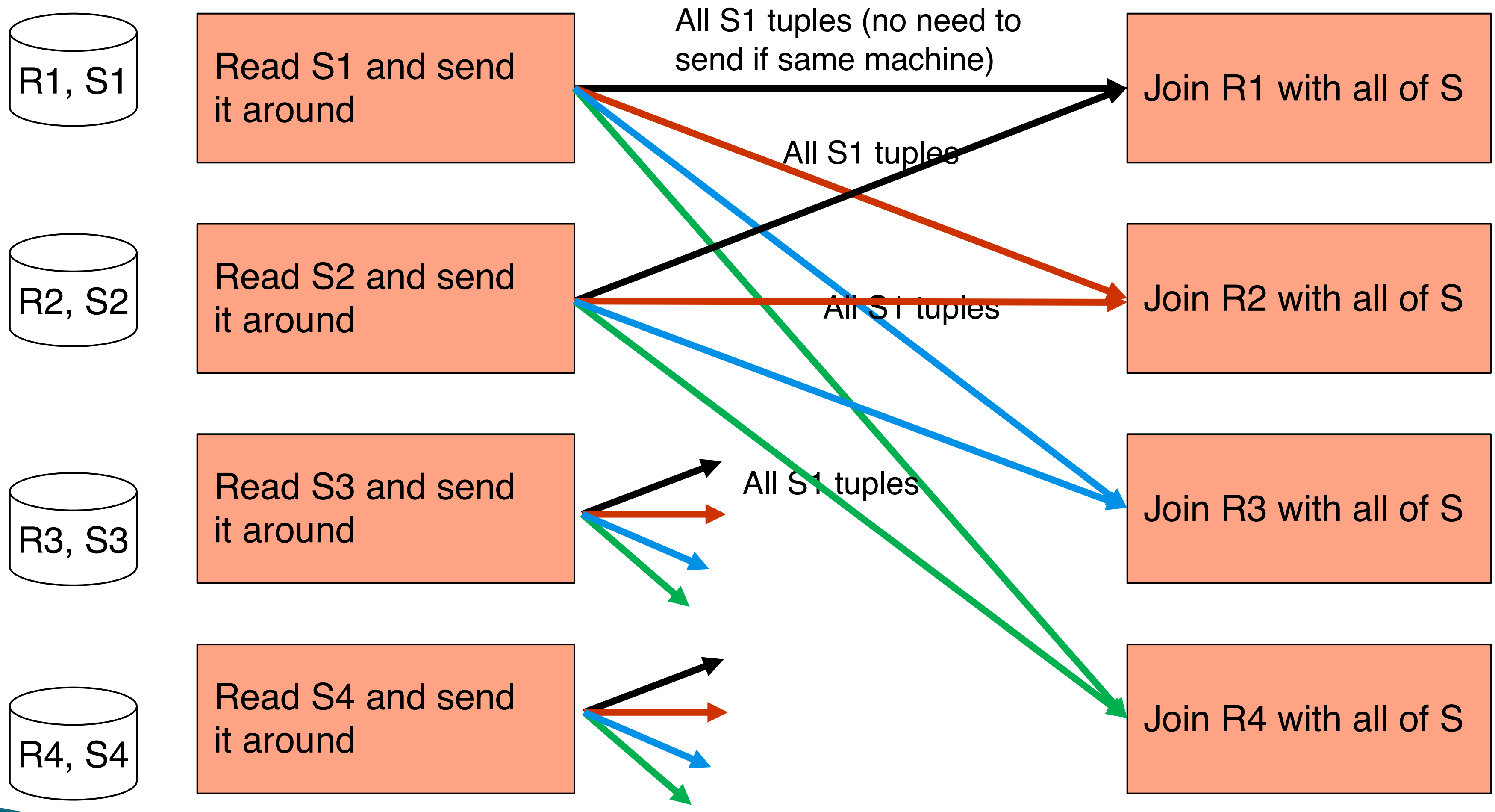
Join the R and S
tuples with h(a) = 2

Join the R and S
tuples with h(a) = 3

Join the R and S
tuples with h(a) = 4

Shuffle – typically expensive

Asymmetric Fragment and Replicate



R1, S1

Read S1 and send it around

All S1 tuples (no need to send if same machine)

Join R1 with all of S

All S1 tuples

R2, S2

Read S2 and send it around

All S1 tuples

Join R2 with all of S

R3, S3

Read S3 and send it around

All S1 tuples

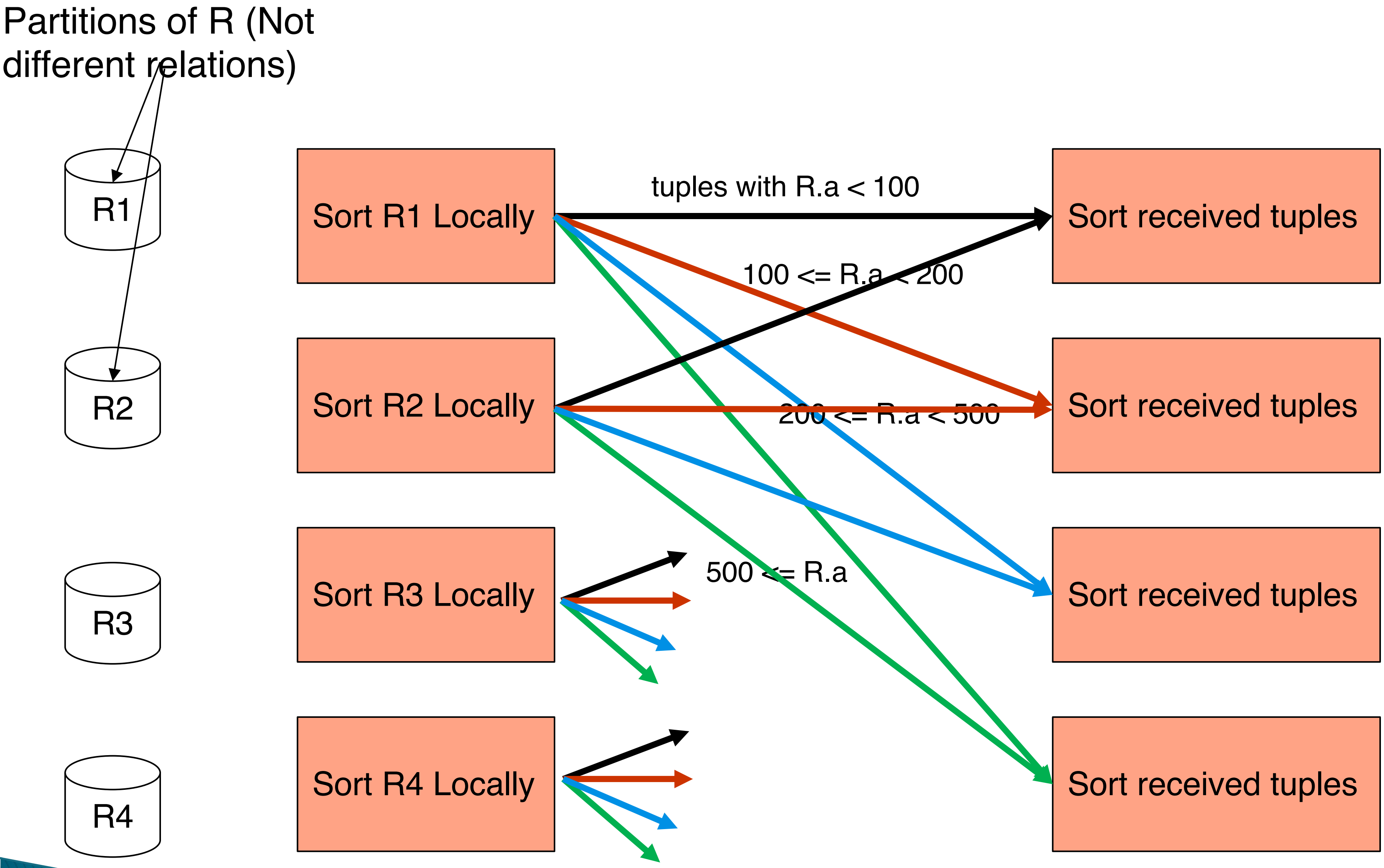
Join R3 with all of S

R4, S4

Read S4 and send it around

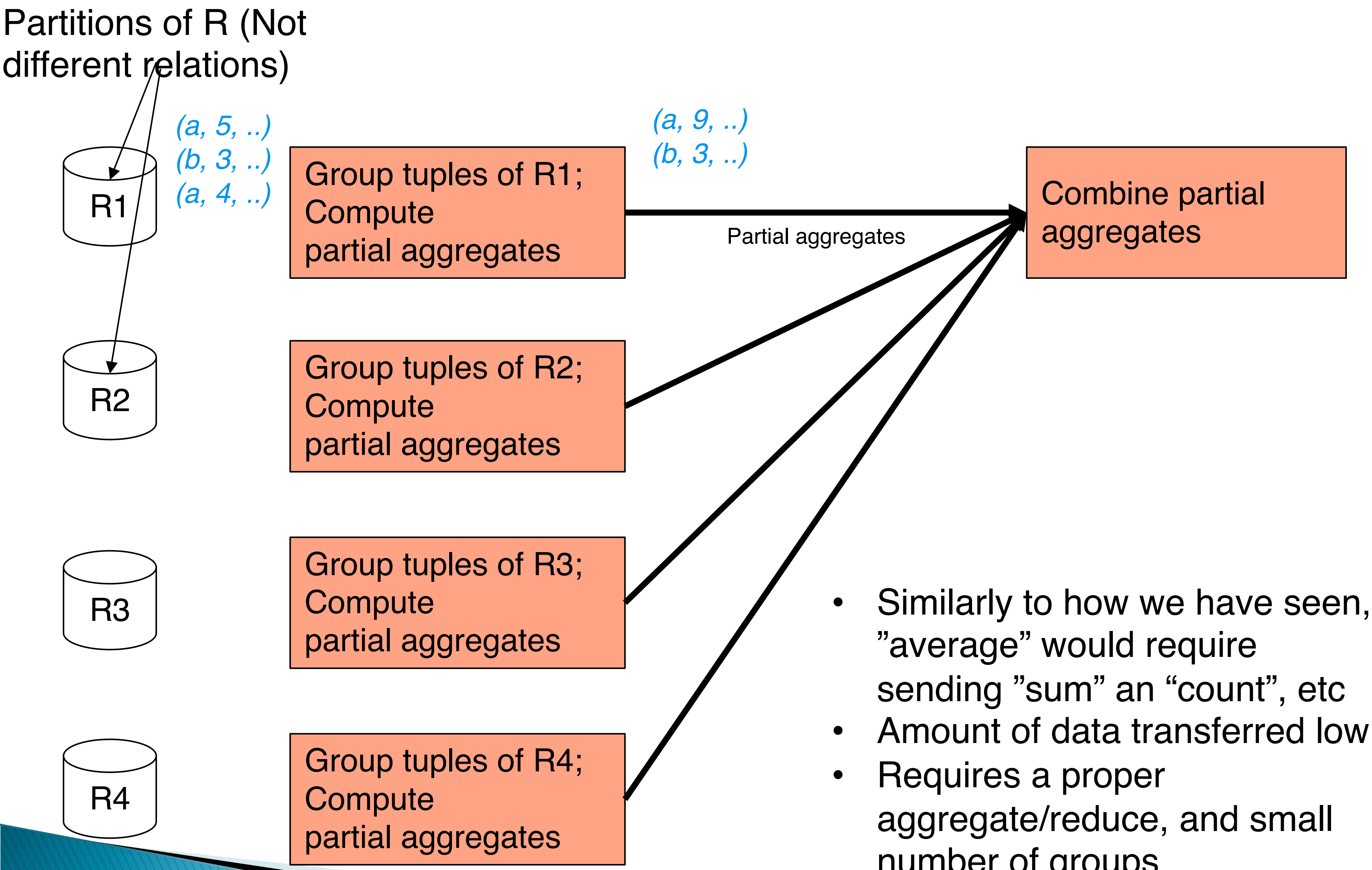
Join R4 with all of S

Basic Parallel Sort (similar skew issues)



Shuffle – typically expensive

Groupby Aggregate: Scenario 1



Partitions of R (Not different relations)

R1

R2

R3

R4

(a, 5, ..)
(b, 3, ..)
(a, 4, ..)

Group tuples of R1; Compute partial aggregates

Group tuples of R2; Compute partial aggregates

Group tuples of R3; Compute partial aggregates

Group tuples of R4; Compute partial aggregates

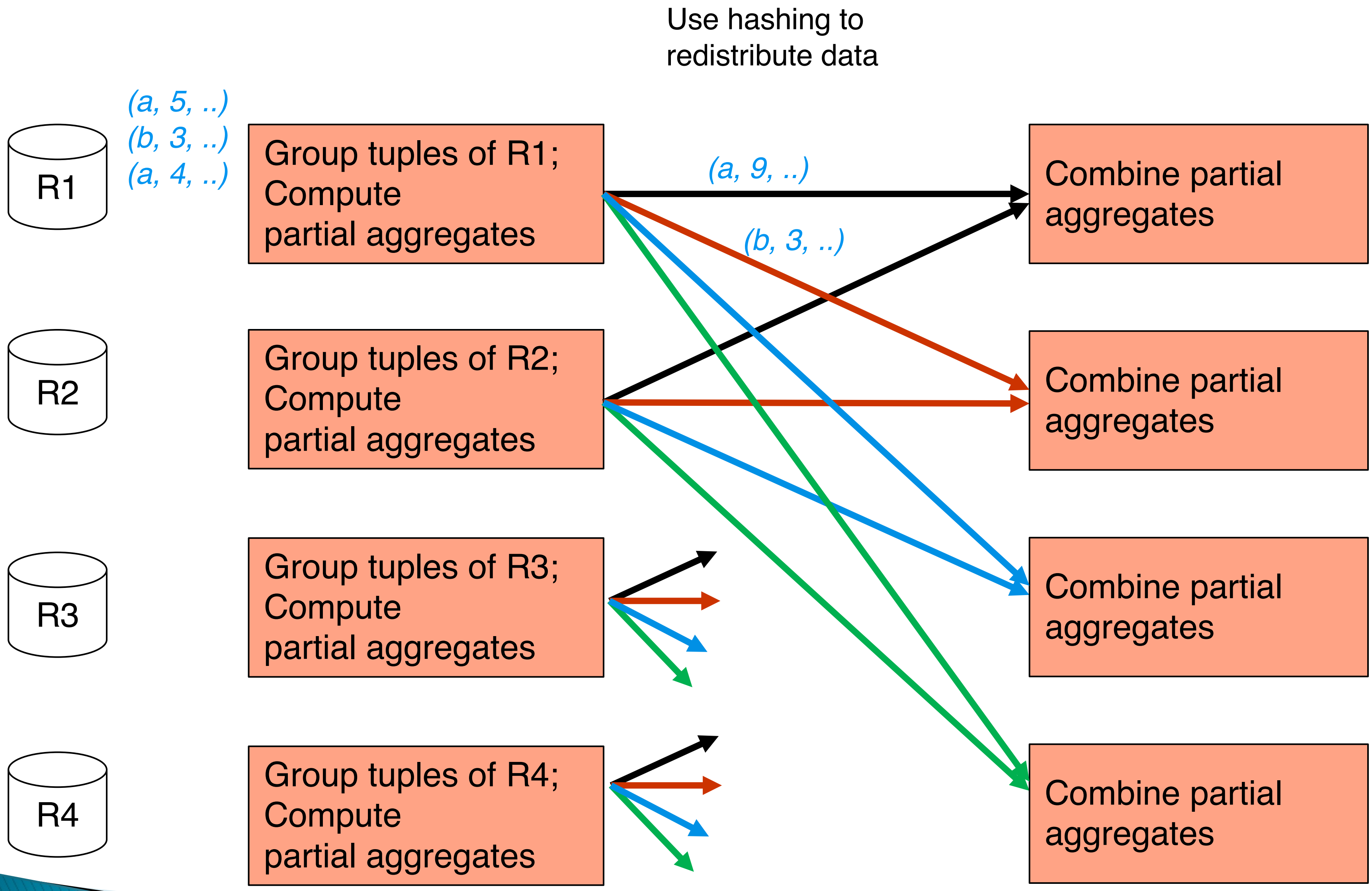
(a, 9, ..)
(b, 3, ..)

Partial aggregates

Combine partial aggregates

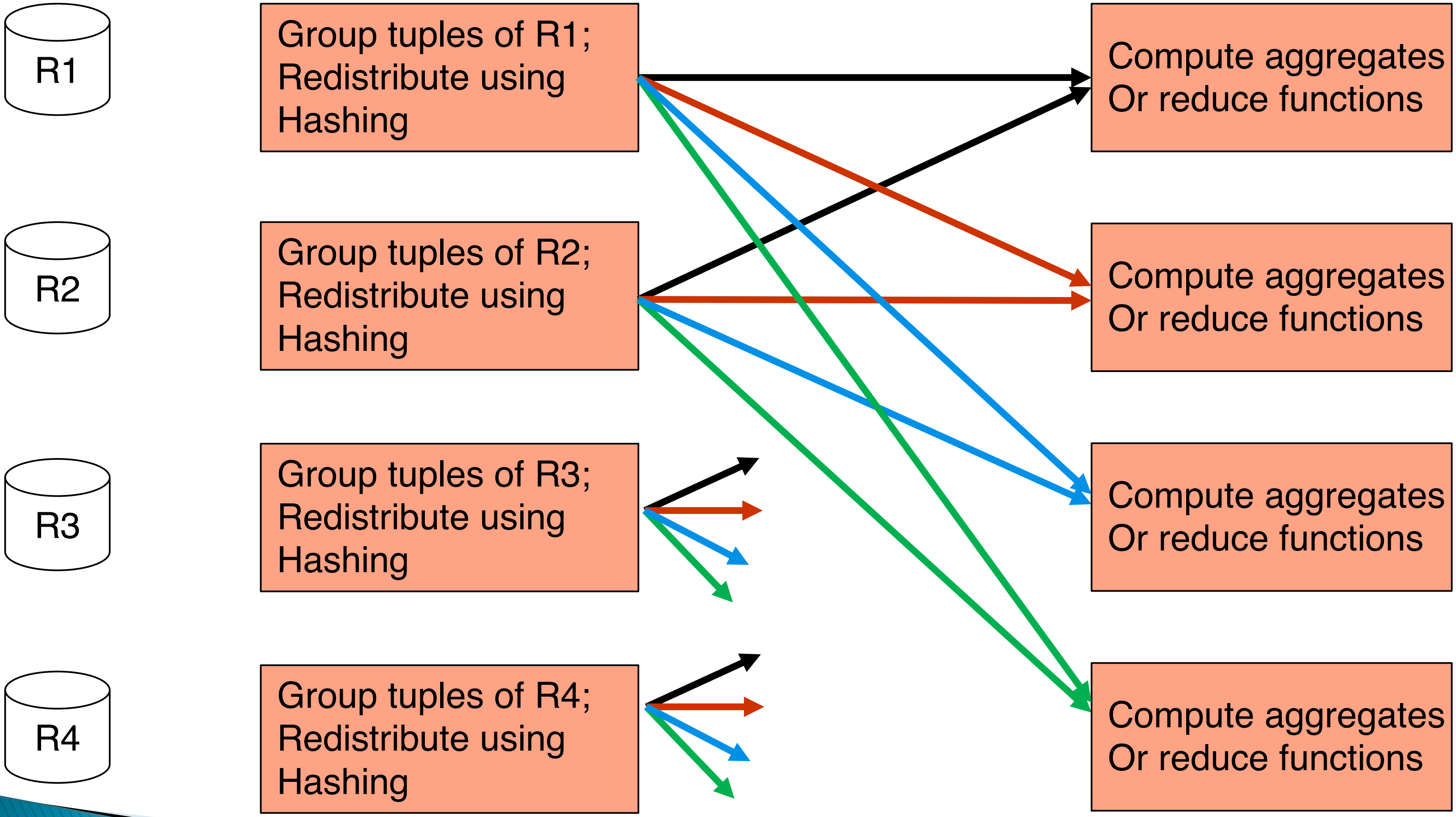
- Similarly to how we have seen, ”average” would require sending ”sum” an “count”, etc
- Amount of data transferred low
- Requires a proper aggregate/reduce, and small number of groups

Groupby Aggregate: Scenario 2 (fewer skew issues)


Use hashing to redistribute data
(a, 5, ..)
(b, 3, ..)
(a, 4, ..)
R1
Group tuples of R1; Compute partial aggregates
(a, 9, ..)
(b, 3, ..)
Combine partial aggregates
R2
Group tuples of R2; Compute partial aggregates
Combine partial aggregates
R3
Group tuples of R3; Compute partial aggregates
Combine partial aggregates
R4
Group tuples of R4; Compute partial aggregates
Combine partial aggregates

Groupby Aggregate: Scenario 3 (skew issues)

e.g., if we want to compute “median” or some other complex statistics – no “partial aggregation” possible


R1
R2
R3
R4
Group tuples of R1; Redistribute using Hashing
Group tuples of R2; Redistribute using Hashing
Group tuples of R3; Redistribute using Hashing
Group tuples of R4; Redistribute using Hashing
Compute aggregates Or reduce functions
Compute aggregates Or reduce functions
Compute aggregates Or reduce functions
Compute aggregates Or reduce functions

Types of Skew

- Tuple placement skew: tuples are distributed non-uniformly
- Selectivity Skew: Selectivity of a predicate varies across tuples
- Redistribution skew: re-partitioning introduces skew
- Join product skew: the join selectivity varies across partitions

Skew Avoidance Fundamentals

- Range Partitioning instead of hash partitioning
  - Can estimate the “splitting points” using sampling to distributed equally across machines
  - Typically focus on uniform split of the “build” relation (probe relations less important)
- Subset-Replicate
  - Need to handle the case of too many tuples with the same join value
  - e.g., {1, 1, 1, 1, 1, 1, 2, 3} should be split into {1, 1, 1, 1} and {1, 1, 2, 3} – but this requires sending the probe relation tuples with value 1 to both the machine (replicate)
- Weighting
  - Consider {1, 2, 3, 4, 4, 4, 4, 4, 4, 4, 4, 4, 6} split into {1, 2, 3, 4}, {4, 4, 4, 4}, {4, 4, 4, 6}
  - For the probe relation tuples with value 4: 1/8 should go to machine 1, 1/2 to machine 2, and 3/8 should go to machine 3
- Virtual processor partitioning and load scheduling
  - The above can’t handle join product skew well
  - If m machines, create a partitioning into 100m partitions, and then assign those to the m machines using round robin, or through a more complex cost-based manner

Implementation: Sampling

- Trickier in a disk-based model, especially with the data distributed across machines
- Stratified sampling:
  - Each processor takes a partial sample and we combine all of them
  - Not guaranteed to be a random sample of the relation
- Even on each processor, can't take a random sample of data
  - Instead choose a page randomly and use all tuples or some of the tuples from it

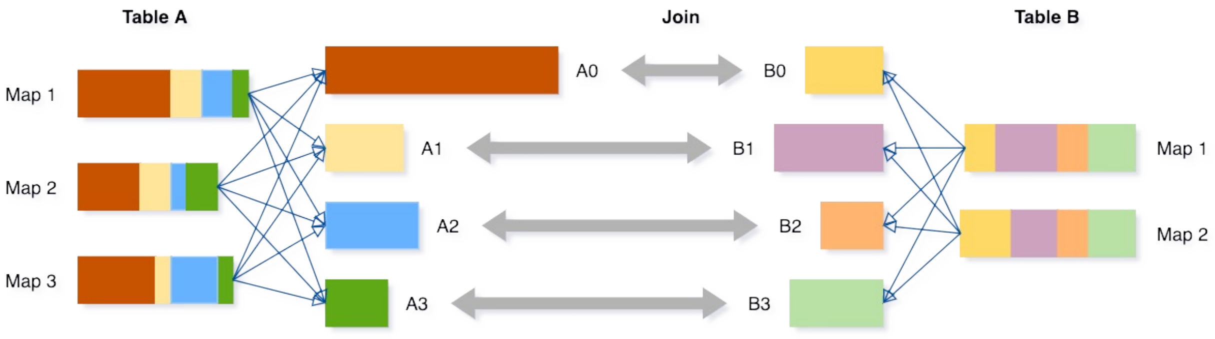
Implementation in Gamma

- Gamma uses a “split table” to do the re-partitioning
- Added a new type of split table for “range” partitioning
- Weighted range partitioning can be added easily
- Virtual processor partitioning causes problems
  - Split tables get large, and Gamma needs one page per entry in the split table
  - Instead, add another layer of abstraction on top

Summary of Experimental Results

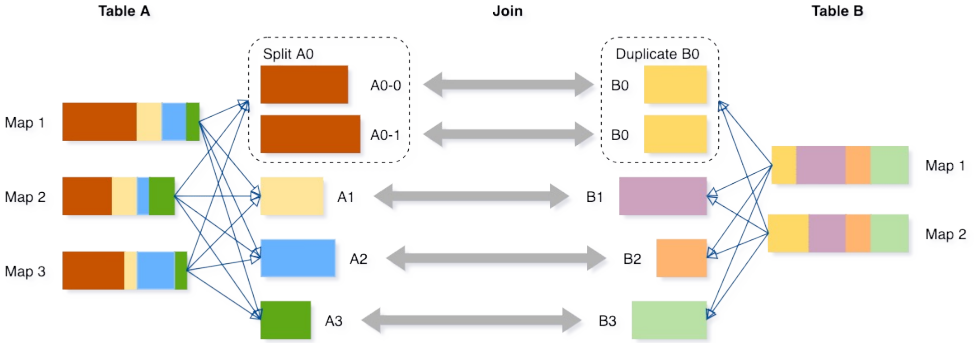
- Basic hybrid hash works well with low skew, but often doesn’t finish in presence of large skew
- Among the other techniques, Virtual Processor Range Partitioning w/ Round Robin works best across different scenarios
- Final recommendation:
  - Take a sample to see which of the relations (if either) is skewed
  - If neither is skewed, use basic parallel hash join to avoid any of the overheads
  - If one of them is skewed, use virtual process range partitioning w/ round robin – with more skewed relation as the building relation
- Over partitioning widely used today as well, in systems like Spark

Skew Optimization in Apache Spark 3.0



No AQE Skew Join

Looks to be similar to “subset-replicate”



With AQE Skew Join

https://databricks.com/blog/2020/05/29/adaptive-query-execution-speeding-up-spark-sql-at-runtime.html

Outline

- Query evaluation techniques for large databases
- Skew avoidance strategies
- Query compilation
- Vectorization

Motivation

- DBMSs built for 70’s-80’s hardware
- Current hardware is much much different
  - Need to rethink the design
- Key issues:
  - Pipelining → dependent code, branches bad
  - Multi-core
  - Caches
  - GPUs: lots of processing power, not clear how to use it Increasingly
  - NUMA architectures
  - FPGAs

Overview of “Modern” CPUs

- Heavy use of instruction pipelining
- Split a CPU instruction into large number of stages
  - 1993 Pentium: 5-stage pipeline, 2004 Penitum4: 31 pipeline stages
  - Example stages: IF = Instruction Fetch, ID = Instruction Decode, EX = Execute, MEM = Memory access, etc...
  - More stages –> simpler architecture
  - More stages necessitates speculative execution
  - More stages –> Wasted work because of dependent instructions and branch misprediction
- Super-scalar architectures
  - Large number of independent pipelines
  - Hard to keep feeding data into them in many cases

Overview of “Modern” CPUs

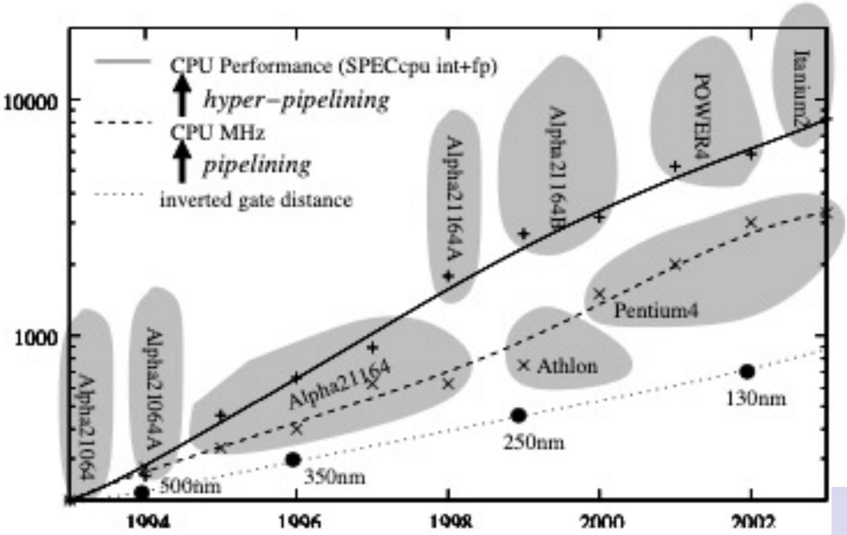
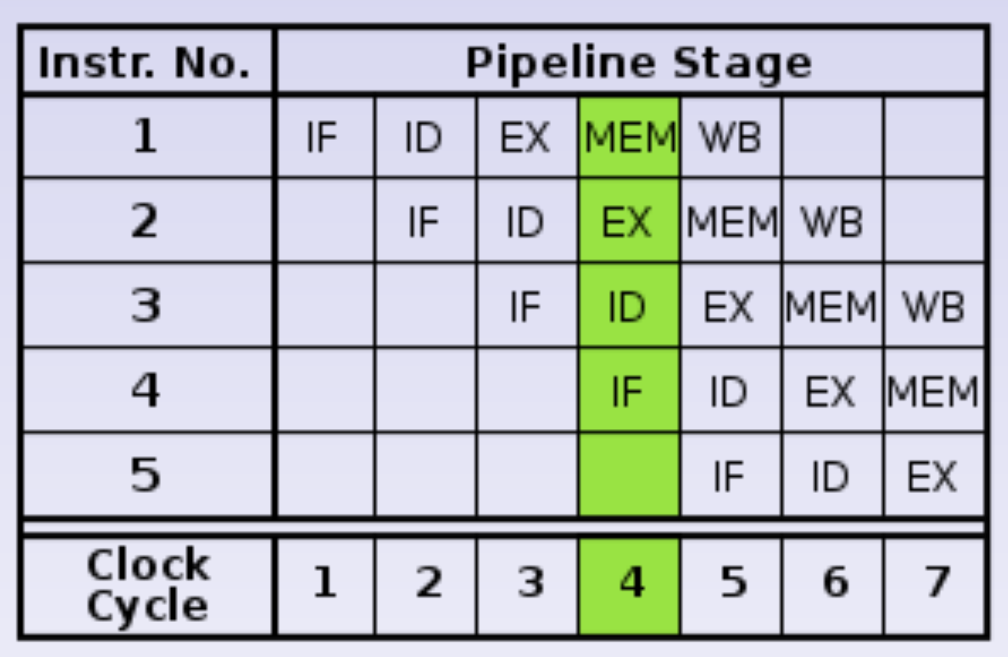


Figure 1: A Decade of CPU Performance



Instr. No.	Pipeline Stage						
1	IF	ID	EX	MEM	WB		
2		IF	ID	EX	MEM	WB	
3			IF	ID	EX	MEM	WB
4				IF	ID	EX	MEM
5					IF	ID	EX
Clock Cycle	1	2	3	4	5	6	7

Effect of Branch Mispredictions

```
int sel_lt_int_col_int_val(int n, int* res, int* in, int V) {
  for(int i=0,j=0; i<n; i++){
    /* branch version */
    if (src[i] < V)
      out[j++] = i;
    /* predicated version */
    bool b = (src[i] < V);
    out[j] = i;
    j += b;
  }
  return j;
}
```

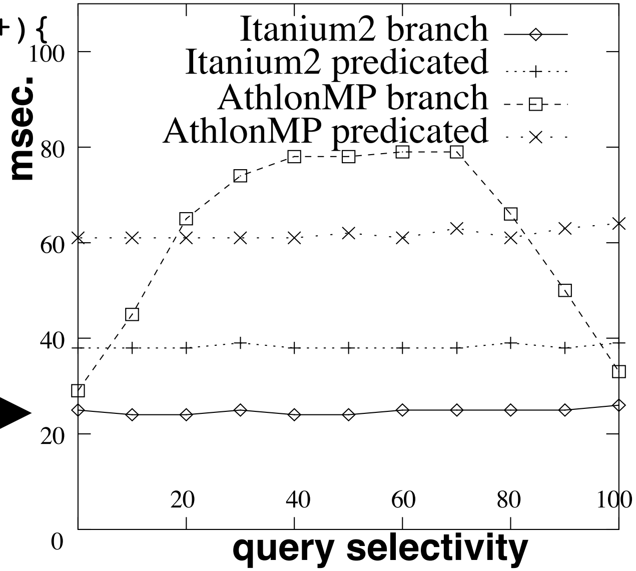
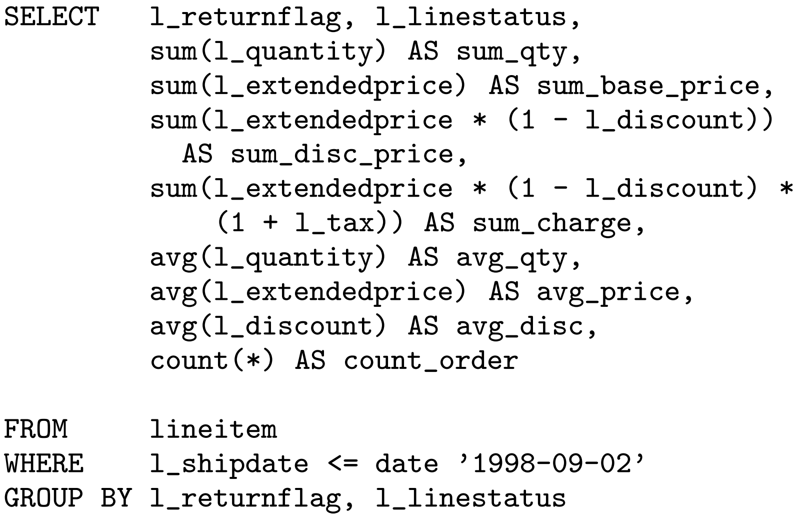


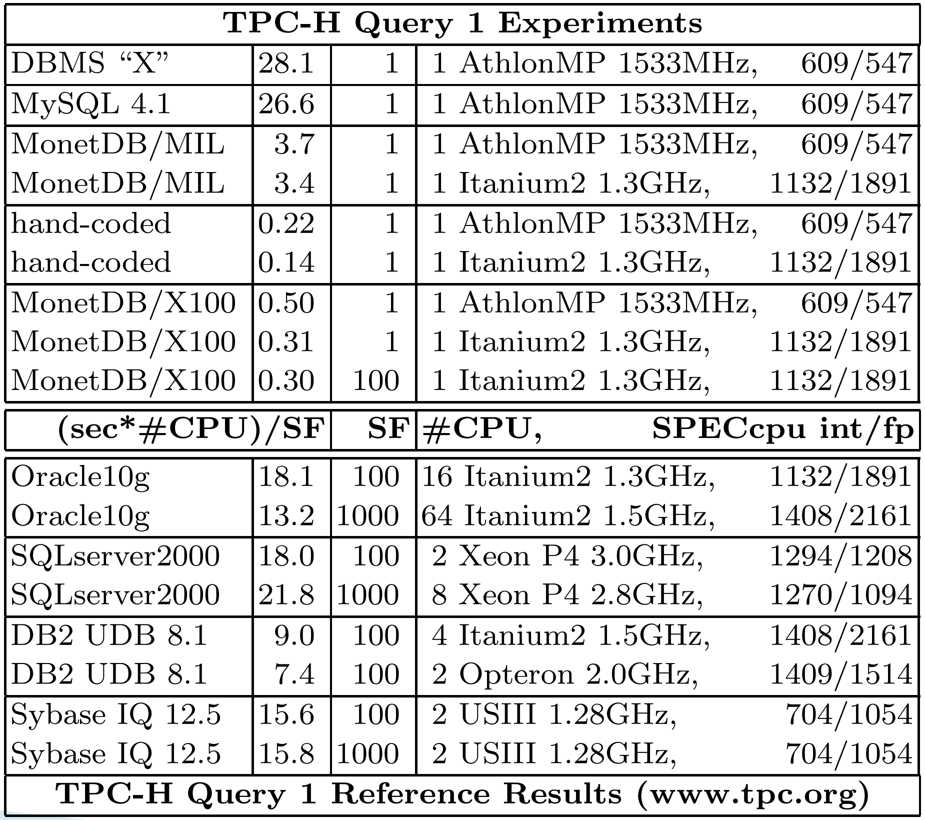
Figure 2: Itanium Hardware Predication Eliminates Branch Mispredictions

Itanium does hardware branch predication, i.e., execute both branches while condition is being evaluated (may be a little more complex than that)

Microbenchmarking: TPCH Query 1



```
SELECT   l_returnflag, l_linestatus,
         sum(l_quantity) AS sum_qty,
         sum(l_extendedprice) AS sum_base_price,
         sum(l_extendedprice * (1 - l_discount))
           AS sum_disc_price,
         sum(l_extendedprice * (1 - l_discount) *
             (1 + l_tax)) AS sum_charge,
         avg(l_quantity) AS avg_qty,
         avg(l_extendedprice) AS avg_price,
         avg(l_discount) AS avg_disc,
         count(*) AS count_order

FROM     lineitem
WHERE    l_shipdate <= date '1998-09-02'
GROUP BY l_returnflag, l_linestatus
```

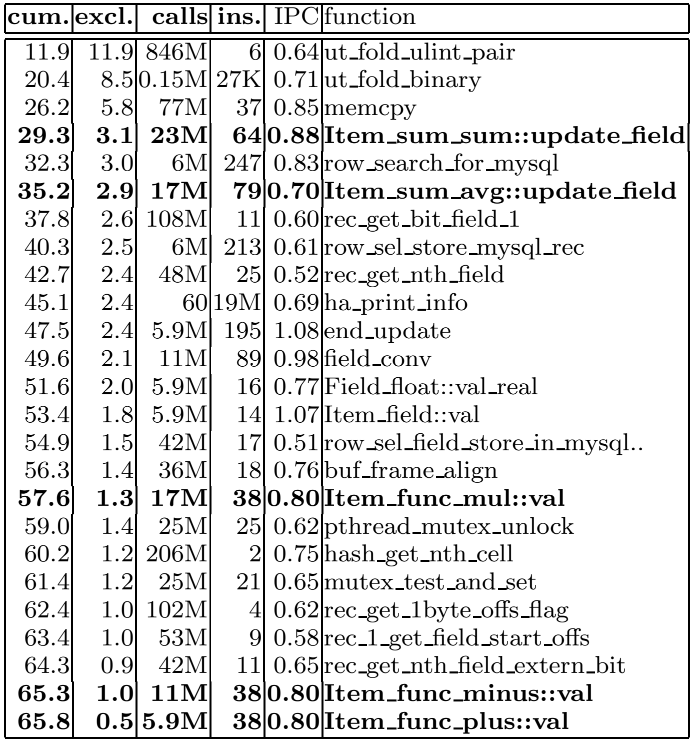


TPC-H Query 1 Experiments				
DBMS "X"	28.1	1	1 AthlonMP 1533MHz,	609/547
MySQL 4.1	26.6	1	1 AthlonMP 1533MHz,	609/547
MonetDB/MIL	3.7	1	1 AthlonMP 1533MHz,	609/547
MonetDB/MIL	3.4	1	1 Itanium2 1.3GHz,	1132/1891
hand-coded	0.22	1	1 AthlonMP 1533MHz,	609/547
hand-coded	0.14	1	1 Itanium2 1.3GHz,	1132/1891
MonetDB/X100	0.50	1	1 AthlonMP 1533MHz,	609/547
MonetDB/X100	0.31	1	1 Itanium2 1.3GHz,	1132/1891
MonetDB/X100	0.30	100	1 Itanium2 1.3GHz,	1132/1891
	(sec*#CPU)/SF	**SF**	**#CPU,**	**SPECcpu int/fp**
Oracle10g	18.1	100	16 Itanium2 1.3GHz,	1132/1891
Oracle10g	13.2	1000	64 Itanium2 1.5GHz,	1408/2161
SQLserver2000	18.0	100	2 Xeon P4 3.0GHz,	1294/1208
SQLserver2000	21.8	1000	8 Xeon P4 2.8GHz,	1270/1094
DB2 UDB 8.1	9.0	100	4 Itanium2 1.5GHz,	1408/2161
DB2 UDB 8.1	7.4	100	2 Opteron 2.0GHz,	1409/1514
Sybase IQ 12.5	15.6	100	2 USIII 1.28GHz,	704/1054
Sybase IQ 12.5	15.8	1000	2 USIII 1.28GHz,	704/1054
TPC-H Query 1 Reference Results (www.tpc.org)				

Microbenchmarking: TPCH Query 1

- Lot of overhead due to interpretation
  - Database code designed to handle arbitrary expressions, known only at run-time
  - Something that should take 4 cycles takes 49 cycles in MySQL
  - Likely due to inability to use loop pipelining
    - Should be able to evaluate an operation in parallel on all tuples
    - But compiler can't do that
    - The function call cost not getting amortized

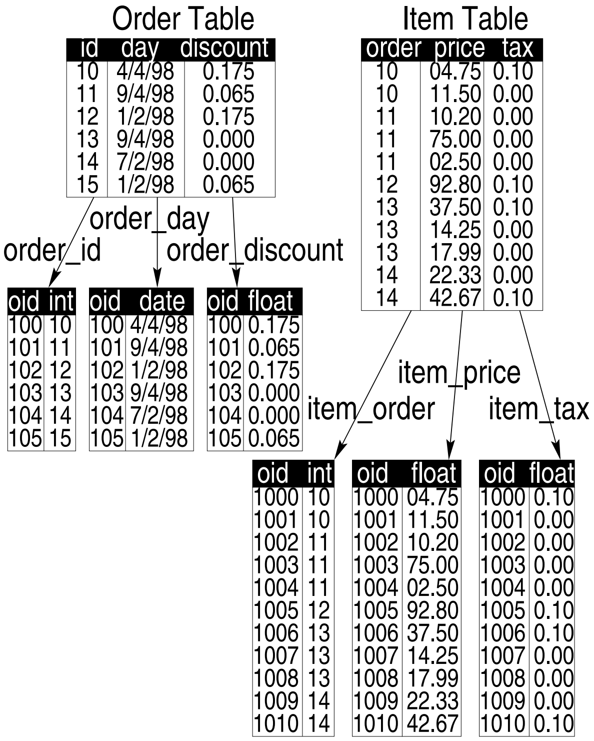
```
LOAD src1,reg1
LOAD src2,reg2
ADD reg1,reg2,reg3
STOR dst,reg3
```



cum.	**excl.**	**calls**	**ins.**	IPC	function
11.9	11.9	846M	6	0.64	ut_fold_ulint_pair
20.4	8.5	0.15M	27K	0.71	ut_fold_binary
26.2	5.8	77M	37	0.85	memcpy
29.3	**3.1**	**23M**	**64**	**0.88**	**Item_sum_sum::update_field**
32.3	3.0	6M	247	0.83	row_search_for_mysql
35.2	**2.9**	**17M**	**79**	**0.70**	**Item_sum_avg::update_field**
37.8	2.6	108M	11	0.60	rec_get_bit_field_1
40.3	2.5	6M	213	0.61	row_sel_store_mysql_rec
42.7	2.4	48M	25	0.52	rec_get_nth_field
45.1	2.4	60	19M	0.69	ha_print_info
47.5	2.4	5.9M	195	1.08	end_update
49.6	2.1	11M	89	0.98	field_conv
51.6	2.0	5.9M	16	0.77	Field_float::val_real
53.4	1.8	5.9M	14	1.07	Item_field::val
54.9	1.5	42M	17	0.51	row_sel_field_store_in_mysql..
56.3	1.4	36M	18	0.76	buf_frame_align
57.6	**1.3**	**17M**	**38**	**0.80**	**Item_func_mul::val**
59.0	1.4	25M	25	0.62	pthread_mutex_unlock
60.2	1.2	206M	2	0.75	hash_get_nth_cell
61.4	1.2	25M	21	0.65	mutex_test_and_set
62.4	1.0	102M	4	0.62	rec_get_1byte_offs_flag
63.4	1.0	53M	9	0.58	rec_1_get_field_start_offs
64.3	0.9	42M	11	0.65	rec_get_nth_field_extern_bit
65.3	**1.0**	**11M**	**38**	**0.80**	**Item_func_minus::val**
65.8	**0.5**	**5.9M**	**38**	**0.80**	**Item_func_plus::val**

MonetDB/MIL

- MonetDB uses the DSM model
- All operators take in a few BATs as input, and produce a BAT



id	day	discount
10	4/4/98	0.175
11	9/4/98	0.065
12	1/2/98	0.175
13	9/4/98	0.000
14	7/2/98	0.000
15	1/2/98	0.065

order	price	tax
10	04.75	0.10
10	11.50	0.00
11	10.20	0.00
11	75.00	0.00
11	02.50	0.00
12	92.80	0.10
13	37.50	0.10
13	14.25	0.00
13	17.99	0.00
14	22.33	0.00
14	42.67	0.10

oid	int
100	10
101	11
102	12
103	13
104	14
105	15

oid	date
100	4/4/98
101	9/4/98
102	1/2/98
103	9/4/98
104	7/2/98
105	1/2/98

oid	float
100	0.175
101	0.065
102	0.175
103	0.000
104	0.000
105	0.065

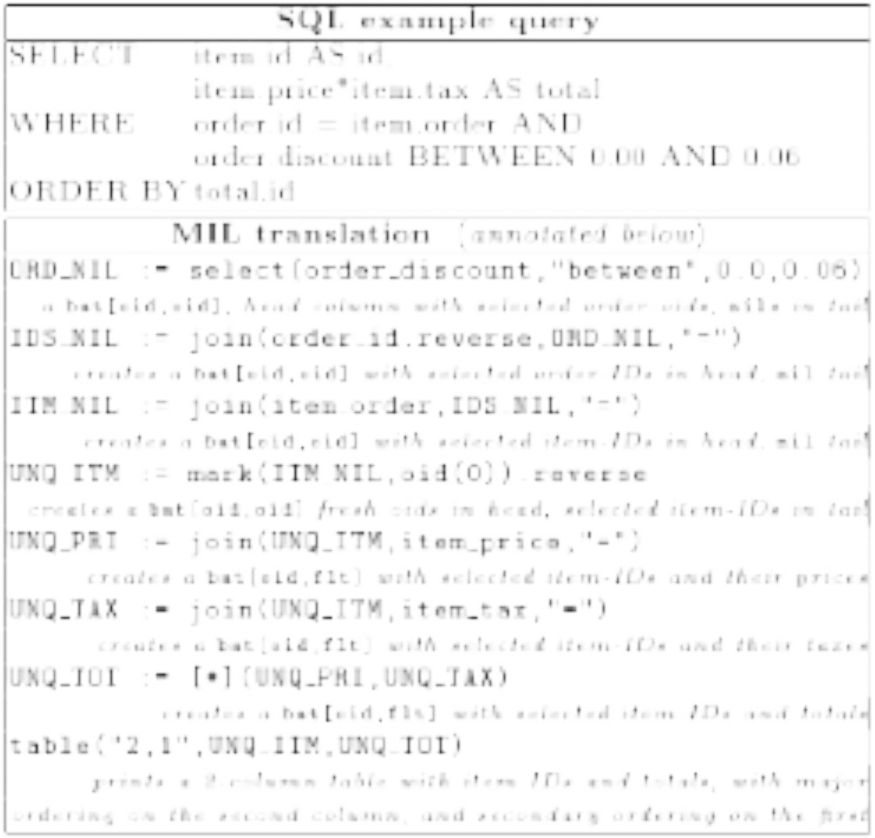
oid	int
1000	10
1001	10
1002	11
1003	11
1004	11
1005	12
1006	13
1007	13
1008	13
1009	14
1010	14

oid	float
1000	04.75
1001	11.50
1002	10.20
1003	75.00
1004	02.50
1005	92.80
1006	37.50
1007	14.25
1008	17.99
1009	22.33
1010	42.67

oid	float
1000	0.10
1001	0.00
1002	0.00
1003	0.00
1004	0.00
1005	0.10
1006	0.10
1007	0.00
1008	0.00
1009	0.00
1010	0.10

Fig. 3. Mapping of Relational Tables.

BATs = Binary Association table



SQL example query

```
SELECT     item.id AS id,
           item.price*item.tax AS total
WHERE      order.id = item.order AND
           order.discount BETWEEN 0.00 AND 0.06
ORDER BY total,id
```

MIL translation *(annotated below)*

```
ORD_NIL  := select(order_discount,"between",0.0,0.06)
   a bat[oid,oid], head column with selected order oids, nils in tail
IDS_NIL  := join(order_id.reverse,ORD_NIL,"=")
      creates a bat[oid,oid] with selected order IDs in head, nil tail
ITM_NIL  := join(item_order,IDS_NIL,"=")
       creates a bat[oid,oid] with selected item-IDs in head, nil tail
UNQ_ITM  := mark(ITM_NIL,oid(0)).reverse
  creates a bat[oid,oid] fresh oids in head, selected item-IDs in tail
UNQ_PRI  := join(UNQ_ITM,item_price,"=")
       creates a bat[oid,flt] with selected item-IDs and their prices
UNQ_TAX  := join(UNQ_ITM,item_tax,"=")
        creates a bat[oid,flt] with selected item-IDs and their taxes
UNQ_TOT  := [*](UNQ_PRI,UNQ_TAX)
            creates a bat[oid,flt] with selected item IDs and totals
table("2,1",UNQ_ITM,UNQ_TOT)
       prints a 2-column table with item IDs and totals, with major
ordering on the second column, and secondary ordering on the first
```

Fig. 4. A simple SQL query and a MIL translation.

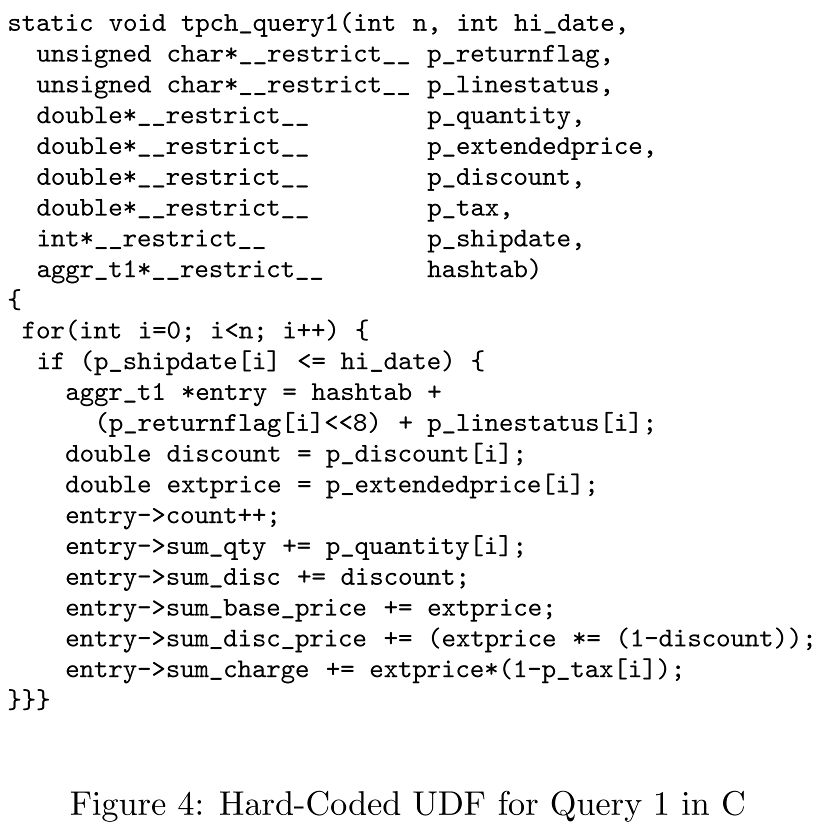
MonetDB/MIL

- MonetDB uses the DSM model
- All operators take in a few BATs as input, and produce a BAT

- Performance better than alternatives
- However:
  - All MIL operations become memory-bound instead of CPU-bound
  - For large enough BATs, every MIL operation (effectively) brings the input BATs from memory to cache, and writes out the output BAT to memory

Hardcoded UDF

- Much faster than anything else
- Proposed solution comes close to achieving that



```
static void tpch_query1(int n, int hi_date,
  unsigned char*__restrict__ p_returnflag,
  unsigned char*__restrict__ p_linestatus,
  double*__restrict__        p_quantity,
  double*__restrict__        p_extendedprice,
  double*__restrict__        p_discount,
  double*__restrict__        p_tax,
  int*__restrict__           p_shipdate,
  aggr_t1*__restrict__       hashtab)
{
 for(int i=0; i<n; i++) {
  if (p_shipdate[i] <= hi_date) {
    aggr_t1 *entry = hashtab +
      (p_returnflag[i]<<8) + p_linestatus[i];
    double discount = p_discount[i];
    double extprice = p_extendedprice[i];
    entry->count++;
    entry->sum_qty += p_quantity[i];
    entry->sum_disc += discount;
    entry->sum_base_price += extprice;
    entry->sum_disc_price += (extprice *= (1-discount));
    entry->sum_charge += extprice*(1-p_tax[i]);
}}}
```

Figure 4: Hard-Coded UDF for Query 1 in C

MonetDB/X100

- Key Idea: Combine the best of Volcano Iterator Model and Vectorized execution
- Use the iterator model as the top-level model
- But transfer batches of data (i.e., fragments of BATs) between the operators – called *vectors*
- Continue to use DSM model for efficient disk-to-memory transfer
- Decompression of data happens when it is loaded into Cache
  - Bandwidth doesn’t matter for Cache as much
- Ensure that loop pipelining is clear to the compiler
- Combine multiple operations together through compilation
  - Similar to query compilation ideas

MonetDB/X100 Example

```
Aggr(
  Project(
    Select(
      Table(lineitem),
      < (shipdate, date('1998-09-03'))),
    [ discountprice = *( -( flt('1.0'), discount),
                         extendedprice) ]),
  [ returnflag ],
  [ sum_disc_price = sum(discountprice) ])
```

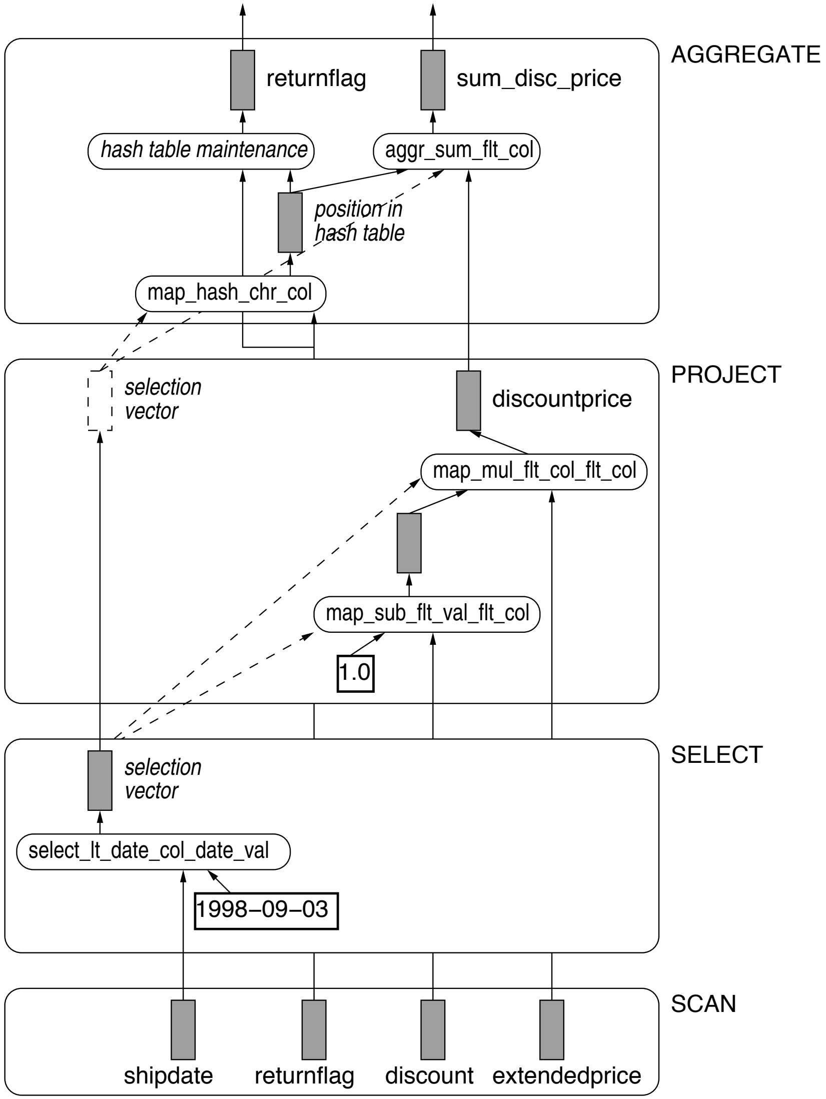
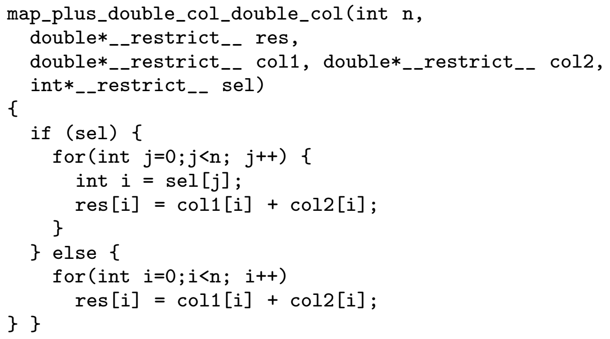


Figure 6: Execution scheme of a simplified TPC-H Query 1 in MonetDB/X100

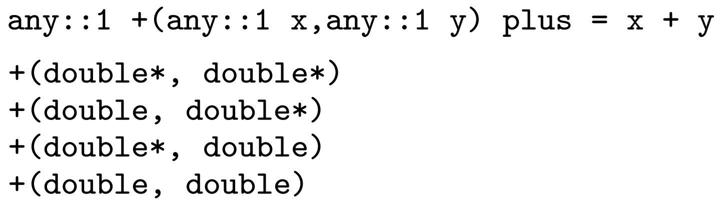
MonetDB/X100 Example



```
map_plus_double_col_double_col(int n,
  double*__restrict__ res,
  double*__restrict__ col1, double*__restrict__ col2,
  int*__restrict__ sel)
{
  if (sel) {
    for(int j=0;j<n; j++) {
      int i = sel[j];
      res[i] = col1[i] + col2[i];
    }
  } else {
    for(int i=0;i<n; i++)
      res[i] = col1[i] + col2[i];
} }
```

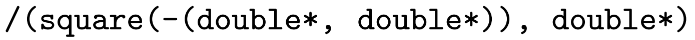
Code generated for a floating point addition
Easy for compiler to aggressively optimize

Generated from primitive patterns



```
any::1 +(any::1 x,any::1 y) plus = x + y

+(double*, double*)
+(double, double*)
+(double*, double)
+(double, double)
```

Can also do compound primitives

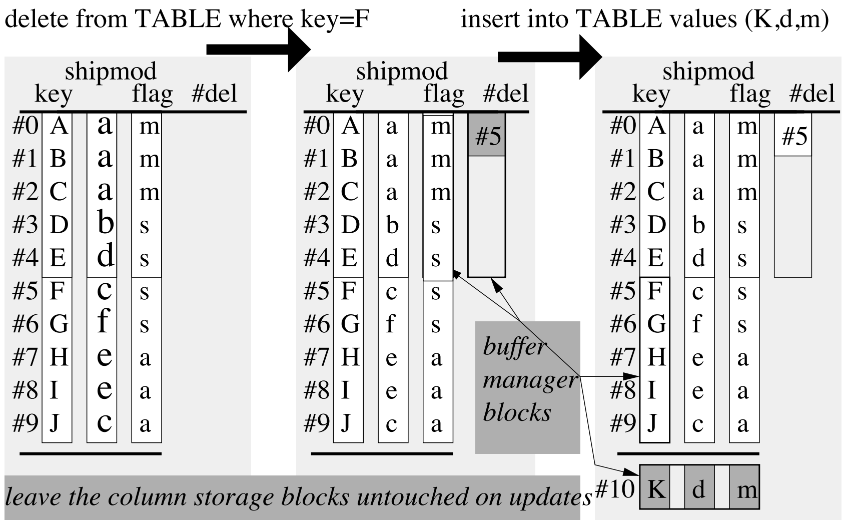


```
/(square(-(double*, double*)), double*)
```

Primitive generated when the system is built
Could switch to doing this more dynamically instead (may have been done later)

MonetDB/X100 Data Storage

- Decomposition storage model
- Uses a “delta” structure to handle updates
- Also does lightweight compression (looks to be dictionary encoding)


delete from TABLE where key=F
insert into TABLE values (K,d,m)
buffer manager blocks
leave the column storage blocks untouched on updates

	key	shipmod	flag	#del
#0	A	a	m	
#1	B	a	m	
#2	C	a	m	
#3	D	b	s	
#4	E	d	s	
#5	F	c	s	
#6	G	f	s	
#7	H	e	a	
#8	I	e	a	
#9	J	c	a	

	key	shipmod	flag	#del
#0	A	a	m	#5
#1	B	a	m	
#2	C	a	m	
#3	D	b	s	
#4	E	d	s	
#5	F	c	s	
#6	G	f	s	
#7	H	e	a	
#8	I	e	a	
#9	J	c	a	

	key	shipmod	flag	#del
#0	A	a	m	#5
#1	B	a	m	
#2	C	a	m	
#3	D	b	s	
#4	E	d	s	
#5	F	c	s	
#6	G	f	s	
#7	H	e	a	
#8	I	e	a	
#9	J	c	a	
#10	K	d	m	

Figure 8: Vertical Storage and Updates

Outline

- Query evaluation techniques for large databases
- Skew avoidance strategies
- Query compilation
- Vectorization

Motivation

- Basic iterator model makes too many function calls
- Vectorization helps, but makes too many copies
- Benefits of "pipelining" are lost

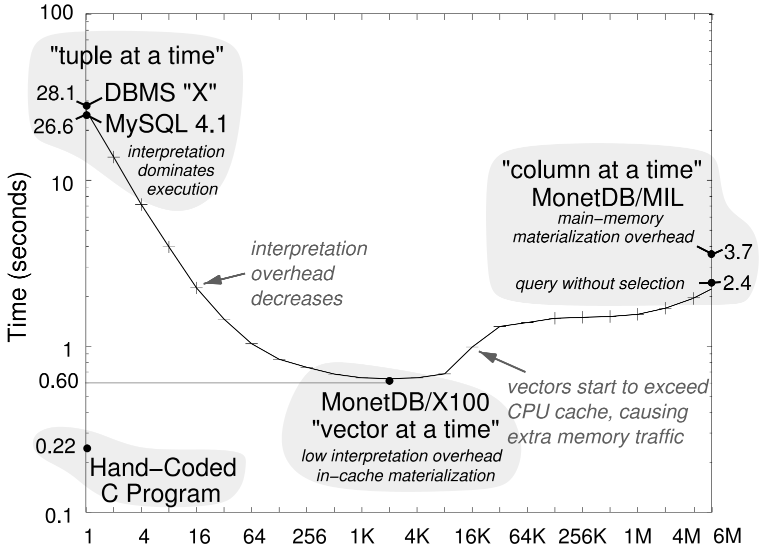
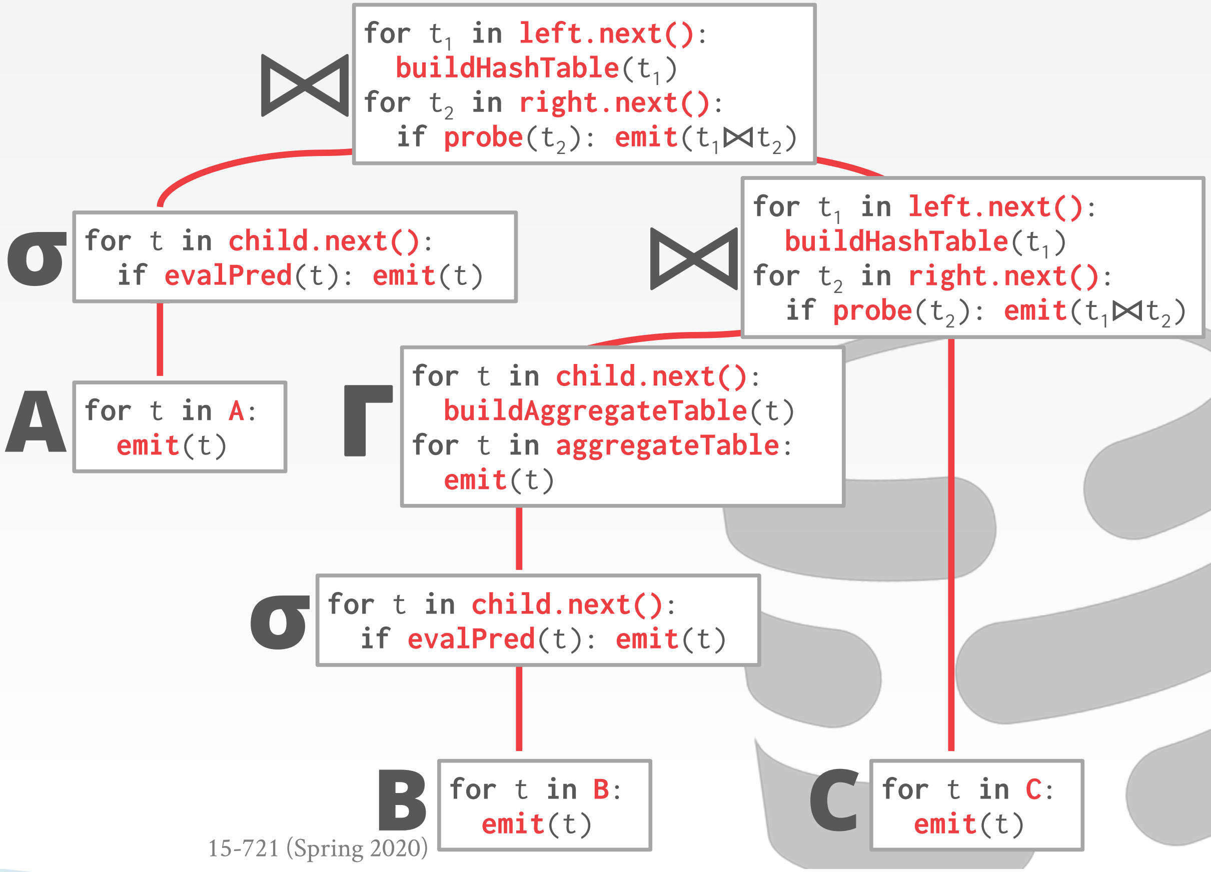


Figure 1: Hand-written code vs. execution engines for TPC-H Query 1 (Figure 3 of [16])

Motivation

- Basic iterator model makes too many function calls
- This doesn't count the additional function calls due to expression interpretation

```
SELECT *
  FROM A, C,
   (SELECT B.id, COUNT(*)
      FROM B
     WHERE B.val = ? + 1
     GROUP BY B.id) AS B
  WHERE A.val = 123
    AND A.id = C.a_id
    AND B.id = C.b_id
```



⋈

```
for t1 in left.next():
  buildHashTable(t1)
for t2 in right.next():
  if probe(t2): emit(t1⋈t2)
```

σ

```
for t in child.next():
  if evalPred(t): emit(t)
```

A

```
for t in A:
  emit(t)
```

⋈

```
for t1 in left.next():
  buildHashTable(t1)
for t2 in right.next():
  if probe(t2): emit(t1⋈t2)
```

Γ

```
for t in child.next():
  buildAggregateTable(t)
for t in aggregateTable:
  emit(t)
```

σ

```
for t in child.next():
  if evalPred(t): emit(t)
```

B

```
for t in B:
  emit(t)
```

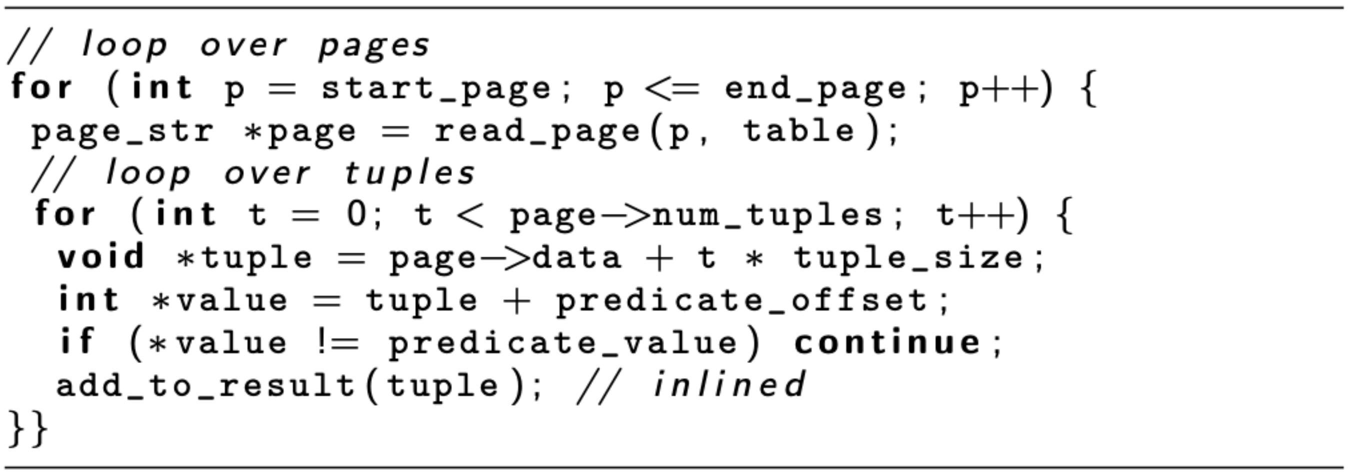
C

```
for t in C:
  emit(t)
```

From Andy Pavlo's slides

HIQUE [ICDE 2010]

- Generated C++ Code using Templates
- Example below: Hardcodes "int" nature of the value, as well as the offset of the attribute within the tuple
  - For a query like: select * from R where R.A = 10



```
// loop over pages
for (int p = start_page; p <= end_page; p++) {
 page_str *page = read_page(p, table);
 // loop over tuples
 for (int t = 0; t < page->num_tuples; t++) {
  void *tuple = page->data + t * tuple_size;
  int *value = tuple + predicate_offset;
  if (*value != predicate_value) continue;
  add_to_result(tuple); // inlined
}}
```

Listing 1. Optimized table scan-select

Motivation

- Two issues with HIQUE:
  - Still using iterator model, leading to function calls across operators
  - Compilation cost very high

- Goals for the Hyper work:
  - Processing is ”data-centric” – data kept in CPU registers as long as possible
  - Data pushed towards the operator, rather than pulled by them
  - Queries compiled into native machine code using LLVM

Motivation

- Basic iterator model makes too many function calls
- Vectorization helps, but makes too many copies
- Benefits of “pipelining” are lost

- Goals for this work:
  - Processing is ”data-centric” – data kept in CPU registers as long as possible
  - Data pushed towards the operator, rather than pulled by them
  - Queries compiled into native machine code using LLVM

Key Idea

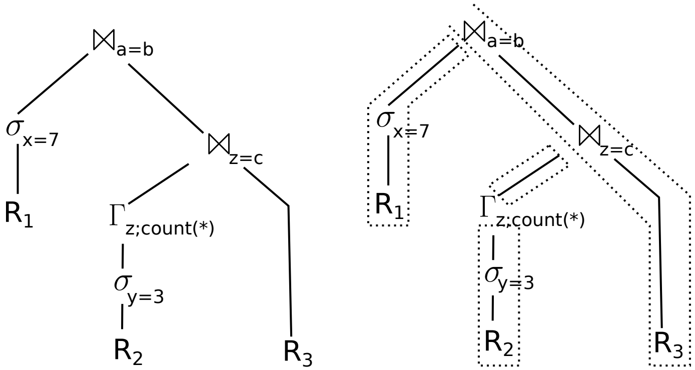
- Pipeline breakers: Operators that remove tuples from CPU registers
- Iterator (pull-based) model can’t keep data in CPU registers because of function calls – end up evicting register contents
- Block-oriented (vectorized) execution break pipelines – produce batches that go beyond register capacity

- Instead: “push” data through the operators until they reach a pipeline breaker

Example

```
select      *
from        R1,R3,
            (select     R2.z,count(*)
             from       R2
             where      R2.y=3
             group by R2.z) R2
where       R1.x=7 and R1.a=R3.b and R2.z=R3.c
```

Figure 2: Example Query



original

with pipeline boundaries

Figure 3: Example Execution Plan for Figure 2

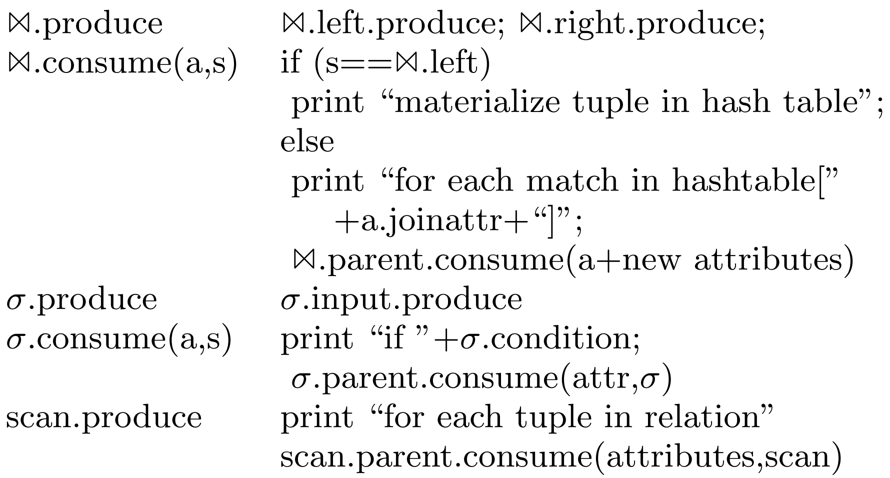
```
initialize memory of $\bowtie_{a=b}$, $\bowtie_{c=z}$, and $\Gamma_z$
for each tuple $t$ in $R_1$
    if $t.x = 7$
        materialize $t$ in hash table of $\bowtie_{a=b}$
for each tuple $t$ in $R_2$
    if $t.y = 3$
        aggregate $t$ in hash table of $\Gamma_z$
for each tuple $t$ in $\Gamma_z$
    materialize $t$ in hash table of $\bowtie_{z=c}$
for each tuple $t_3$ in $R_3$
    for each match $t_2$ in $\bowtie_{z=c}[t_3.c]$
        for each match $t_1$ in $\bowtie_{a=b}[t_3.b]$
            output $t_1 \circ t_2 \circ t_3$
```

Figure 4: Compiled query for Figure 3

NOTE: We typically build hash tables on the “right” side
Here they are building it on the “left” side

Push Model

- Instead of implementing:
  - next()
- Each operator implements:
  - produce(): ask the operator to produce its next tuple (but doesn’t wait)
  - consume(attributes, source): ask the operator to consume a new tuple from its children



```
⋈.produce          ⋈.left.produce; ⋈.right.produce;
⋈.consume(a,s)     if (s==⋈.left)
                    print “materialize tuple in hash table”;
                   else
                    print “for each match in hashtable[”
                        +a.joinattr+“]”;
                    ⋈.parent.consume(a+new attributes)
σ.produce          σ.input.produce
σ.consume(a,s)     print “if ”+σ.condition;
                    σ.parent.consume(attr,σ)
scan.produce       print “for each tuple in relation”
                   scan.parent.consume(attributes,scan)
```

Figure 5: A simple translation scheme to illustrate the *produce*/*consume* interaction

Code Generation

- Tried C++ initially, but:
  - Optimizing C++ compiler really slow (remember this is being done for each query, in addition to parsing, optimization, etc)
  - Can lead to suboptimal performance because of less control
- Instead use LLVM == Low Level Virtual Machine, Compiler Framework
  - Hides the problem of register allocation
  - Portable across machine architectures
  - Strongly typed ➔ easy to catch bugs early
  - Full strength optimizing compiler
- Can mix C++ and LLVM code

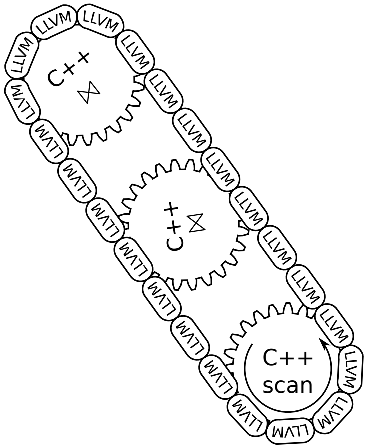


Figure 6: Interaction of LLVM and C++

More…

- Compilation cost itself can be significant for large OLAP queries
  - Later work looked into doing this in an on-demand fashion (ICDE 2018)
  - Execute interpreted code while the query is being compiled
- Other bottlenecks can start showing up
- Vectorization or Compilation?
  - At odds to some extent
  - Later work compared both of them, and tried to harmonize
- Most high-performance systems today use query compilation
  - PostgreSQL supports this today for a small set of operations like expression evaluation

Outline

- Query evaluation techniques for large databases
- Skew avoidance strategies
- Query compilation
- Vectorization
- Query Optimization: Overview

Query Optimization

- Goal: Given a SQL query, find the best "physical operator" tree to execute the query
  - Large number of logically equivalent algebraic representations for a query
  - Many operator trees for each algebraic expression

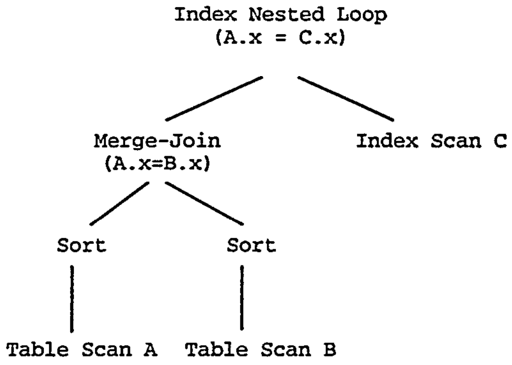


Figure 1. Operator Tree

- For "cost-based" optimization, we need:
  - A space of plans to search through (search space)
  - Cost estimation techniques
  - Enumeration/search algorithm
- Heuristic optimizers typically use "rules"
  - e.g., push down selections as much as possible – typically a good idea but not always

System R Query Optimizer (1979)

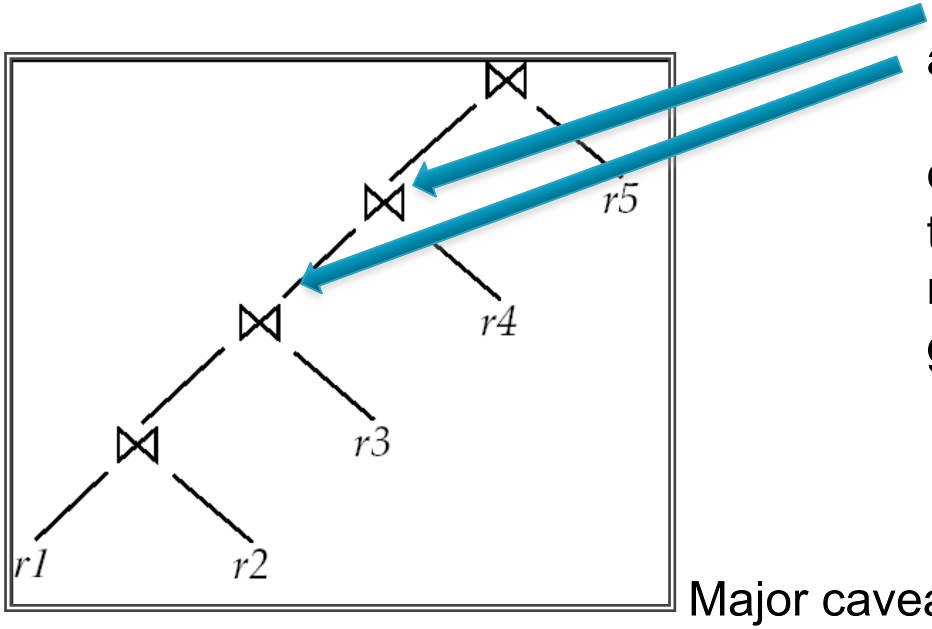
- Focused on SPJ queries (select-project-join)

- Search space:
  - Linear (left-deep) plans
  - Each join can be nested loop or sort-merge (no hash joins)
  - Each scan node either an index scan or a sequential scan

- Cost estimation done using:
  - A set of statistics: #data pages for a relation, #distinct values in a column
  - Formulas for estimating intermediate result sizes
    - Relied on “magic” constants for anything not covered by the statistics
  - Formulas for CPU and I/O cost for each operator

System R Query Optimizer (1979)

- Search algorithm: Bottom-up Dynamic Programming
  - Insight: the best overall plan uses the best plan for any subexpression inside of it



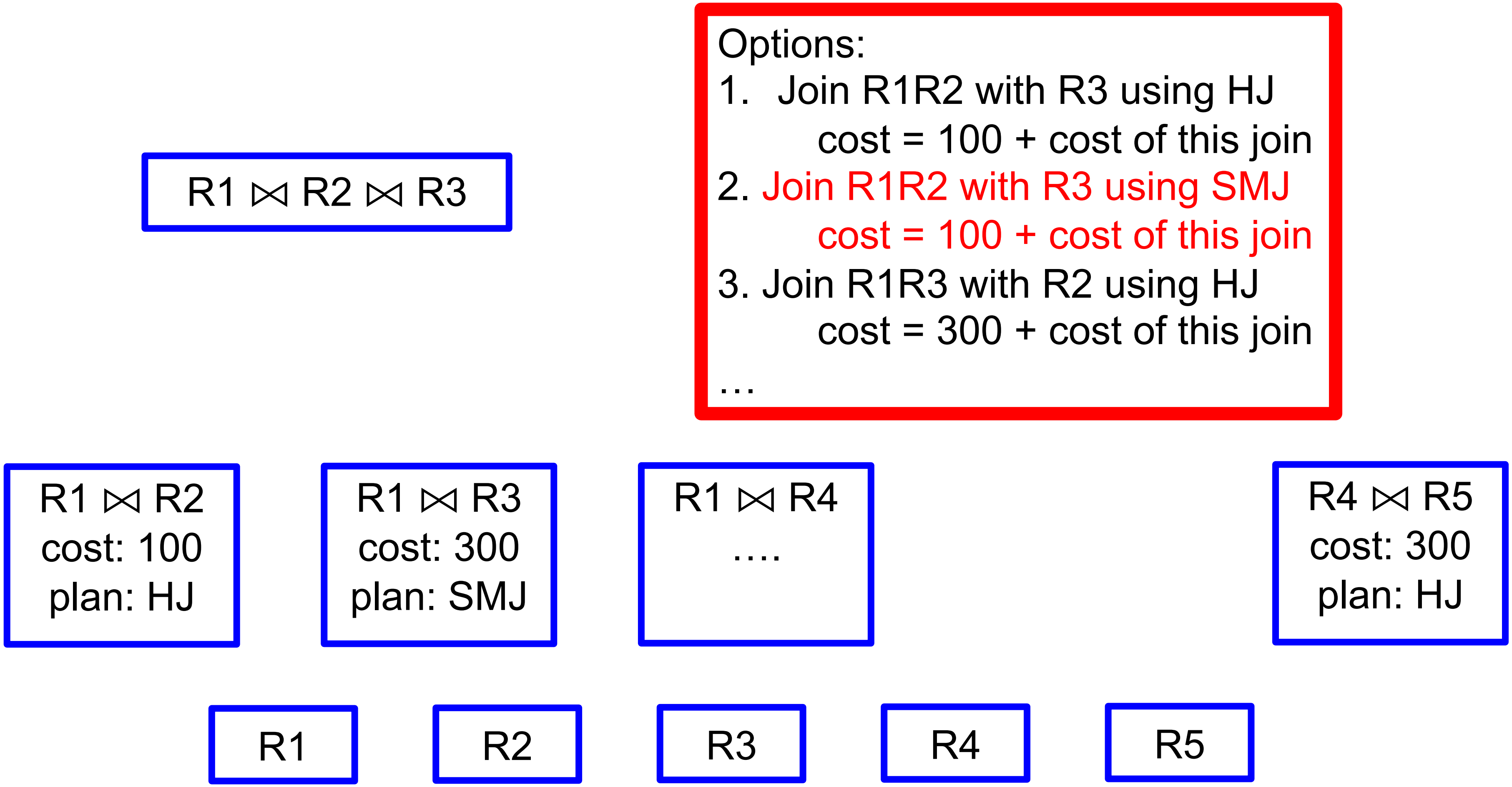
The best overall plan should use the "best" plan for (r1 join r2 join r3 join r4) and the "best" plan for (r1 join r2 join r3)..

e.g., if the best plan for r1 – r2 – r3 was to join r1 and r3 first and then join with r2, we can just substitute that plan, and get an overall better plan

Major caveat: the alternate plan should not miss any "physical properties" that are important
e.g., if the original plan produce r1-r2-r3 in sorted order by D, and the alternate doesn't, the substitution may change the cost of the next join (with r4)

Dynamic Programming Algo.

- Join R1, R2, R3, R4, R5



Options:

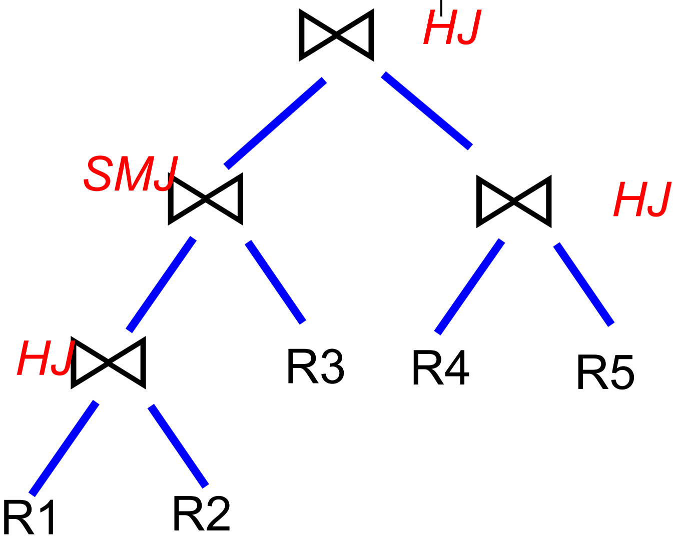
1. Join R1R2 with R3 using HJ
   cost = 100 + cost of this join
2. Join R1R2 with R3 using SMJ
   cost = 100 + cost of this join
3. Join R1R3 with R2 using HJ
   cost = 300 + cost of this join

…

R1 ⋈ R2 ⋈ R3 ⋈ R4 ⋈ R5
cost: 1200
plan: *HJ(R1R2R3, R4R5)*

R1 ⋈ R2 ⋈ R3 ⋈ R4
cost: 700
plan: *HJ(R1R2R3, R4)*

....



R1 ⋈ R2 ⋈ R3
cost: 400
plan: *SMJ(R1R2, R3)*

....

R1 ⋈ R2
cost: 100
plan: HJ

R1 ⋈ R3
cost: 300
plan: SMJ

R1 ⋈ R4
....

....

R4 ⋈ R5
cost: 300
plan: HJ

R1

R2

R3

R4

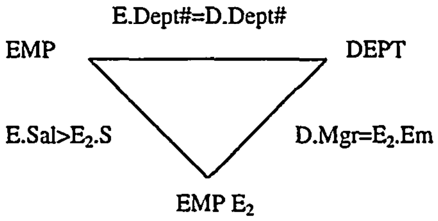
R5

System R Query Optimizer (1979)

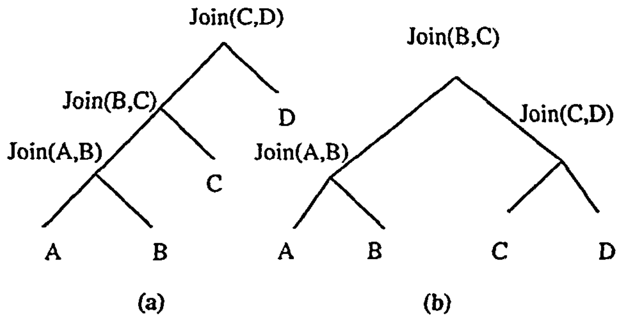
- Interesting orders
  - Sort orders is an important physical property for the query executor (given the reliance on sort-merge joins)
  - So keep track of the sort order in which results are generated
  - Two plans for a subexpression are NOT comparable if the sort orders are different
  - ➔ For each subexpression, more than one plan may be maintained with different sort orders

- Can be generalized to handle “incomparable-ness” in general
  - e.g., one subplan may have better CPU but worse Memory, and the other subplans may have better Memory but worse CPU

Search Space: Reordering



- Intermediate representations
  - Query graphs commonly used in research papers, but only capture a simple subset
  - QGM Structure used in Starburst (will cover later)
  - Many others just use an “operator tree” or an “expression tree”



- Join ordering
  - Bushy plans commonly considered today
  - Significantly add to the search complexity
  - Cartesian products may be allowed in some cases

- Outerjoins
  - Only commute with joins in some cases (will cover later)
  - e.g., Join(R, S LOJ T) = Join(R, S) LOJ T

Search Space: Reordering

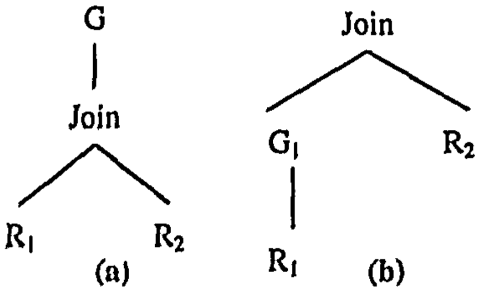
- Group-By and Joins
  - Pushing group by below a join results in significant reductions in tuples being joined

```
select R1.A, sum(R1.B)
from R1, R2
where R1.A = R2.A
group by R1.A
```

equivalent to

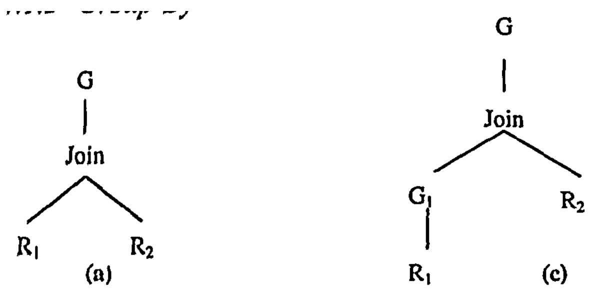
```
select x.A, x.sumB
from R2, (select A, sum(R1.B) as sumB
            from R1
            group by A) x
where R2.A = x.A
```

only if: A is a primary key of R2



Search Space: Reordering

- Group-By and Joins
  - Pushing group by below a join results in significant reductions in tuples being joined



```
select R1.A, sum(R1.B)
from R1, R2
where R1.A = R2.A
group by R1.A
```

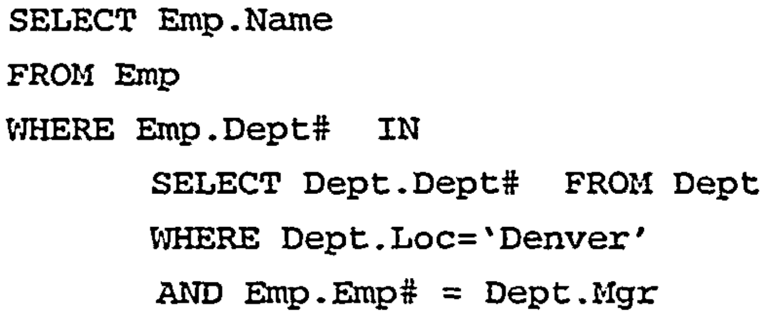
equivalent to

```
select R2.C, sum(x.sumB)
from R2, (select A, sum(R1.B) as sumB
          from R1
          group by A) x
where R2.A = x.A
group by R2.C
```

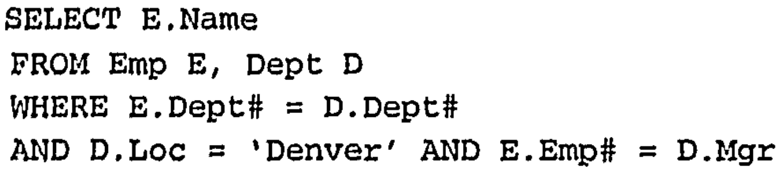
only if: in R2, A → C

Search Space: Subqueries

- Collapsing nested subqueries results in more optimization opportunities
  - Need to be very careful: NULLs, Distincts, Aggregates, etc., cause problems



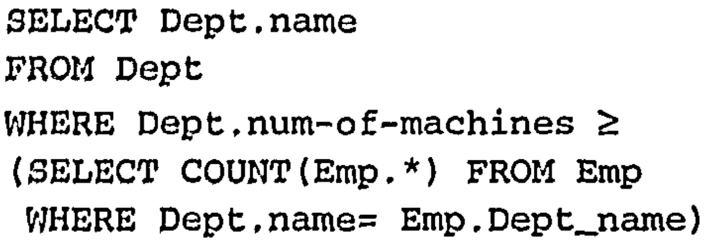
```
SELECT Emp.Name
FROM Emp
WHERE Emp.Dept#  IN
       SELECT Dept.Dept#  FROM Dept
       WHERE Dept.Loc='Denver'
       AND Emp.Emp# = Dept.Mgr
```



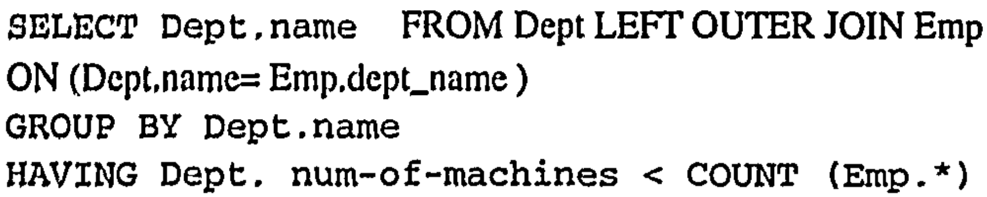
```
SELECT E.Name
FROM Emp E, Dept D
WHERE E.Dept# = D.Dept#
AND D.Loc = 'Denver' AND E.Emp# = D.Mgr
```

Search Space: Subqueries

- Collapsing nested subqueries results in more optimization opportunities
  - Need to be very careful: NULLs, Distincts, Aggregates, etc., cause problems



```
SELECT Dept.name
FROM Dept
WHERE Dept.num-of-machines ≥
(SELECT COUNT(Emp.*) FROM Emp
 WHERE Dept.name= Emp.Dept_name)
```



```
SELECT Dept.name   FROM Dept LEFT OUTER JOIN Emp
ON (Dept.name= Emp.dept_name )
GROUP BY Dept.name
HAVING Dept. num-of-machines < COUNT (Emp.*)
```

LOJ is essential here
Otherwise will miss depts with no employees

Search Space: Semijoins for Optimizing

```
CREATE VIEW DepAvgSal As (
       SELECT  E.did,  Avg(E.Sal)  AS  avgsal
       FROM Emp E
       GROUP BY E.did)
SELECT E.eid, E.sal
FROM Emp E, Dept D, DepAvgSal V
WHERE  E.did   = D.did  AND  E.did  =  V.did
AND E.age < 30 AND D.budget > 100k
AND E.sal > V.avgsal
```

```
CREATE VIEW partialresult AS
(SELECT E.id, E.sal, E.did
 FROM Emp E, Dept D
 WHERE E.did=D.did AND E.age < 30
 AND D.budget > 100k)
CREATE VIEW Filter AS
(SELECT DISTINCT P.did FROM PartialResult P)
CREATE VIEW LimitedAvgSal AS
(SELECT E.did, Avg(E.Sal) AS avgsal
FROM Emp E, Filter F
WHERE E.did = F.did GROUP BY E.did)

SELECT P.eid, P.sal
FROM PartialResult P, LimitedDepAvgSal V
WHERE P.did = V.did AND P.sal > V.avgsal
```

- Say only a few departments (say 10) satisfy the join condition out of, say 10000
  - Only need to compute the “view” tuples for those 10 departments
- So we are passing information “sideways” from the main block into the nested block

Statistics and Cost Estimation

- In general: more information about the data ➔ better estimates
- Single-column statistics
  - min, max, #distinct, #bytes, etc.
  - Histograms for value distributions (e.g., to estimate #tuples satisfying “age < 20”)
  - Many different types of histograms proposed over the years
- Multi-column statistics
  - Correlations among attributes a major issue for estimates
  - Queries of type: “SSN = 0123 and Name = ‘John Smith’” pretty common
    - Independence assumption ➔ huge underestimation of the result size
  - Many proposals for capturing correlations, but hard to make work in practice
- Propagation of errors
  - Even if estimates lower in the query plan are pretty good, estimates for more complex subexpressions become erroneous very quickly

Enumeration Architectures

- Need the optimization algorithm to be "extensible"
  - So it can handle new physical operators, new transformations, new cost estimation approaches, easicly
- Starburst:
  - Uses a rule engine and an intermediate representation called QGM to do query rewrites/transformations
  - Uses a somewhat generalized bottom-up query optimizer
- Volcano/Cascades:
  - Transformation rules to map algebraic expressions
  - Implementation rules to map algebraic expression into an operator tree
  - Uses a "top-down" query optimizer
    - Starts with the overall expression and tries to find all possible ways to get to it
    - Uses "memoization" to keep avoid redoing work
  - Formed the basis of the Microsoft database systems

More…

- Distributed and Parallel Databases
  - Much bigger search space (can place operators anywhere, and can partition them)
  - What to optimize for? Communication cost? Total resources? Response time?
  - Standard approach is to generate a single-machine query plan and then parallelize it (2-phase optimization)
- User-defined Functions
  - Need to consider the cost of executing those (can be hard to estimate)
- Materialized views
  - Given a set of materialized views, hard to decide if those can be used in place of the original relations (undecidable in general)
- …

Outline

- Query evaluation techniques for large databases
- Skew avoidance strategies
- Query compilation
- Vectorization
- Query Optimization: Overview
- How good are the query optimizers, really?

JOB Benchmark

- Build using the IMDB dataset
  - 21 tables, total of 3.6 GB in CSV format
- 113 SPJ queries – no aggregates or subqueries
- More realistic than the commonly used TPC-H/DS benchmarks (or synthetic benchmarks)

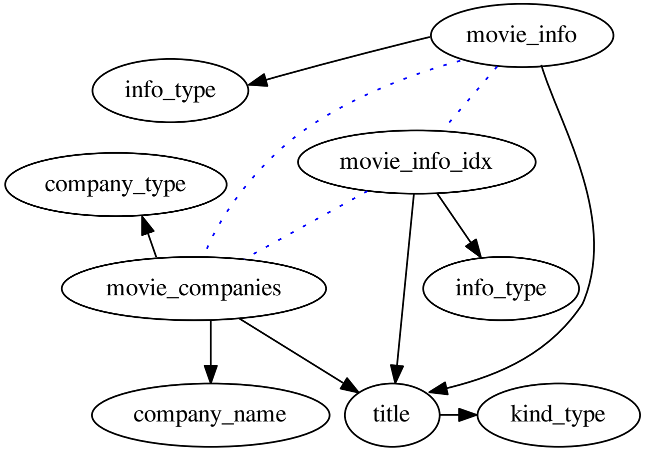
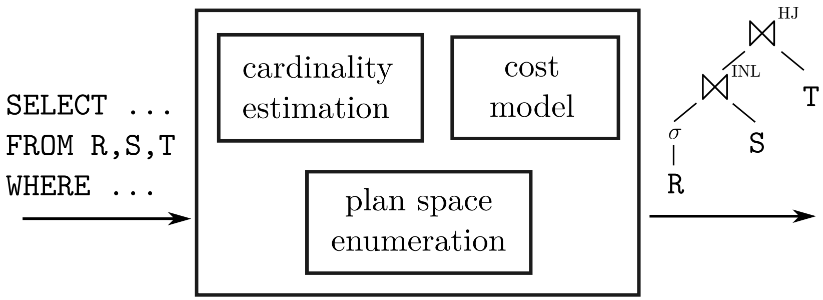


Figure 2: Typical query graph of our workload

```
SELECT cn.name, mi.info, miidx.info
FROM company_name cn, company_type ct,
     info_type it, info_type it2, title t,
     kind_type kt, movie_companies mc,
     movie_info mi, movie_info_idx miidx
WHERE cn.country_code =’[us]’
AND ct.kind = ’production companies’
AND it.info = ’rating’
AND it2.info = ’release dates’
AND kt.kind = ’movie’
AND ... -- (11 join predicates)
```

PostgreSQL Query Optimizer

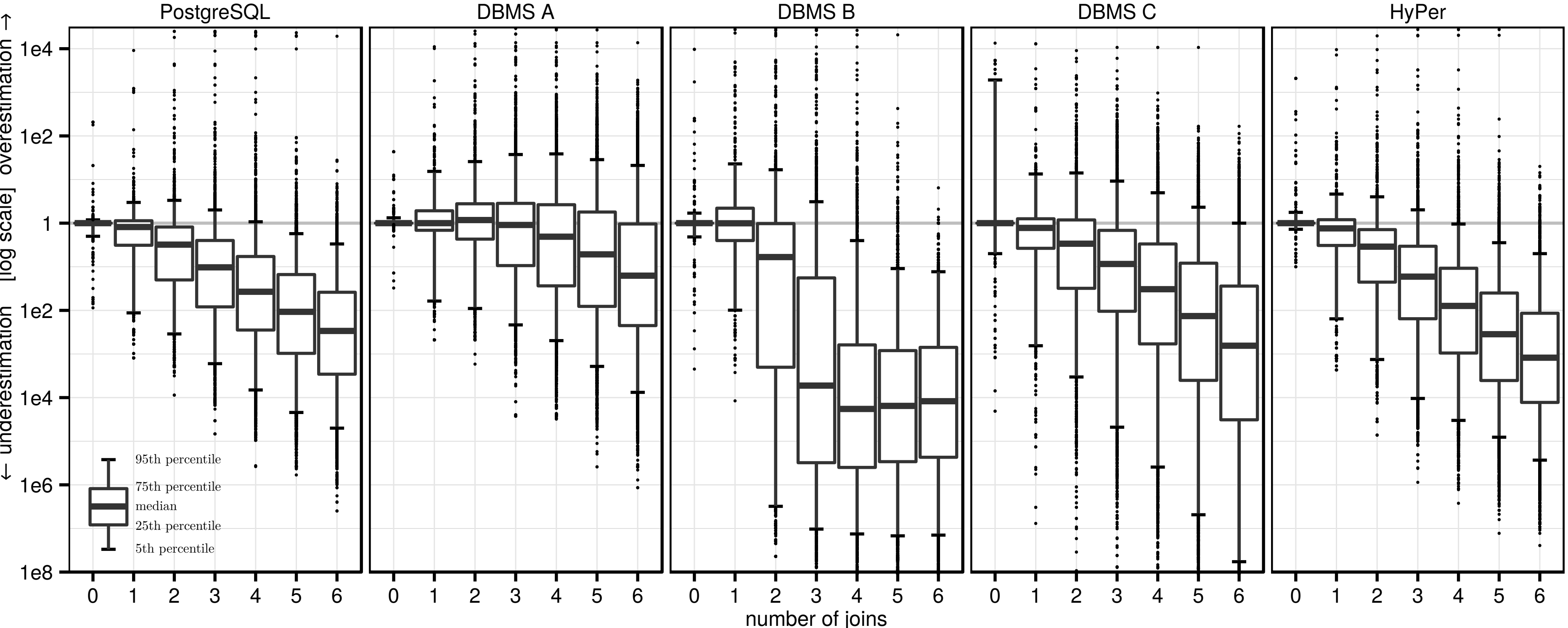
- Standard dynamic programming-based optimizer
  - Includes bushy plans, but no Cartesian products
- Statistics: Single-column histograms, min, max, most frequent values, etc.
  - Assume independence and uniformity outside of those
  - Especially for conjunctive predicates (like A = 10 and B = 20)



- 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

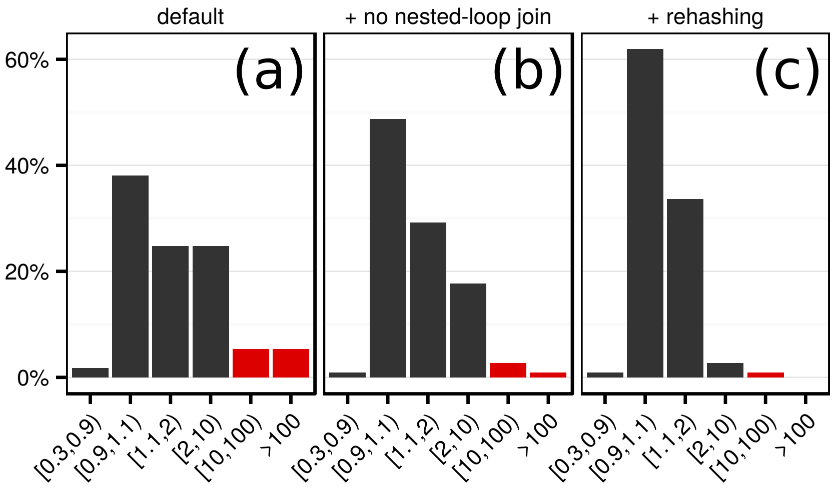


Figure 6: Slowdown of queries using PostgreSQL estimates w.r.t. using true cardinalities (primary key indexes only)

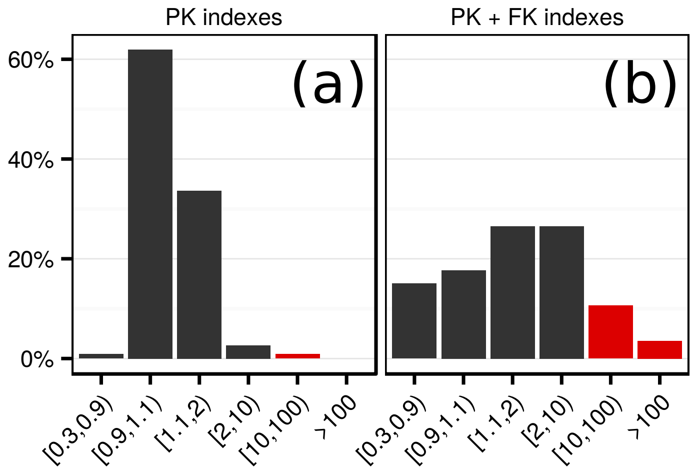


Figure 7: Slowdown of queries using PostgreSQL estimates w.r.t. using true cardinalities (different index configurations)

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

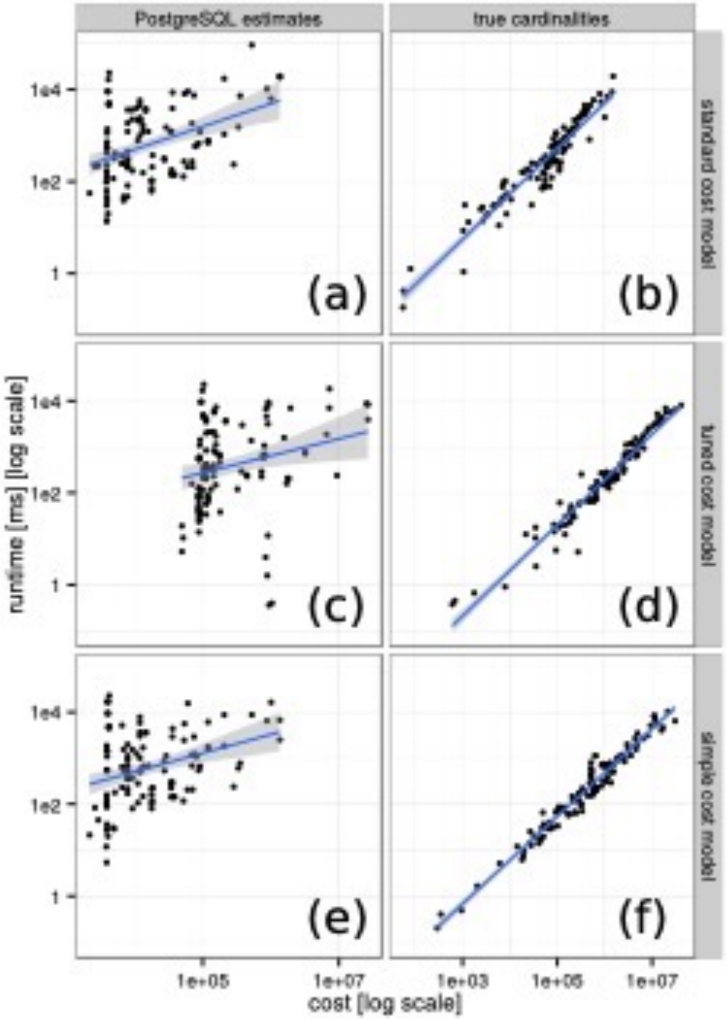


Figure 8: Predicted cost vs. runtime for different cost models

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

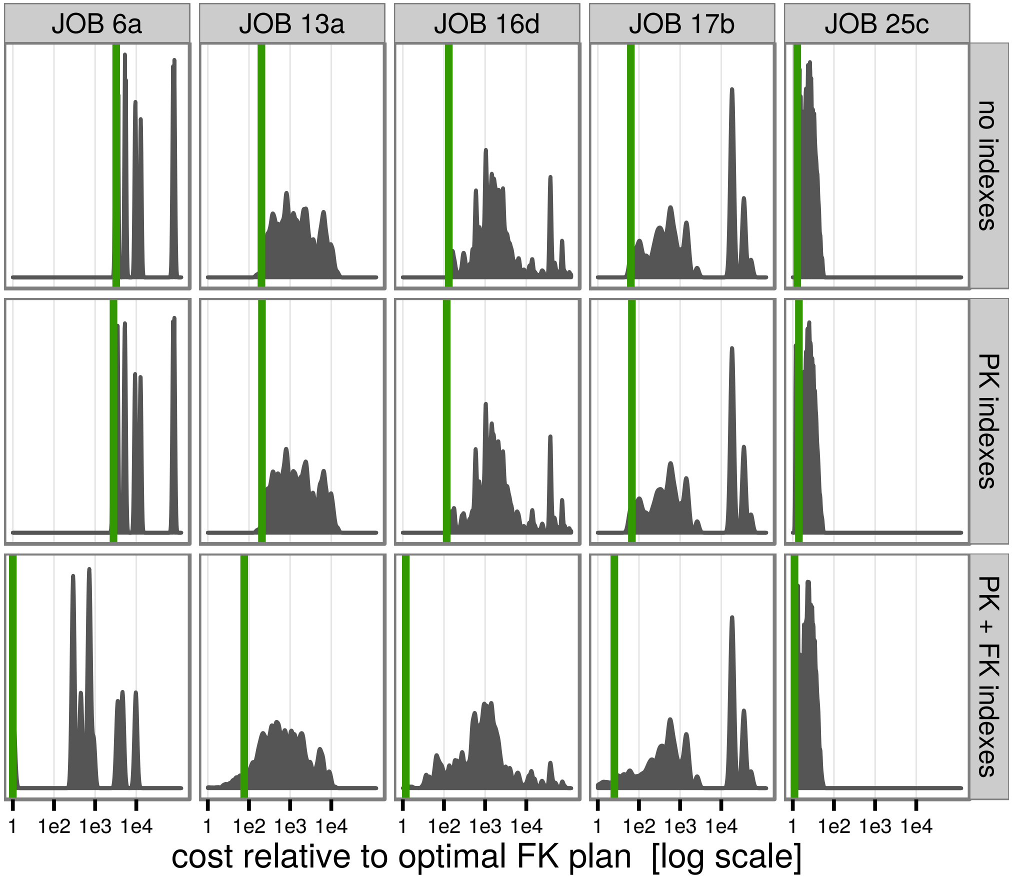
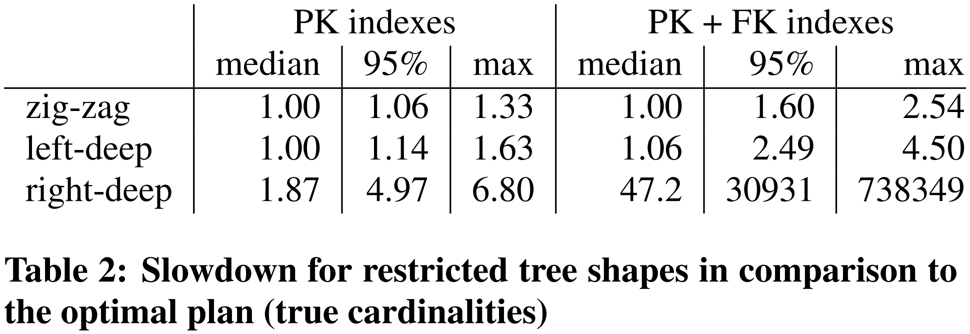


Figure 9: Cost distributions for 5 queries and different index configurations. The vertical green lines represent the cost of the optimal plan

Results: Join Orders

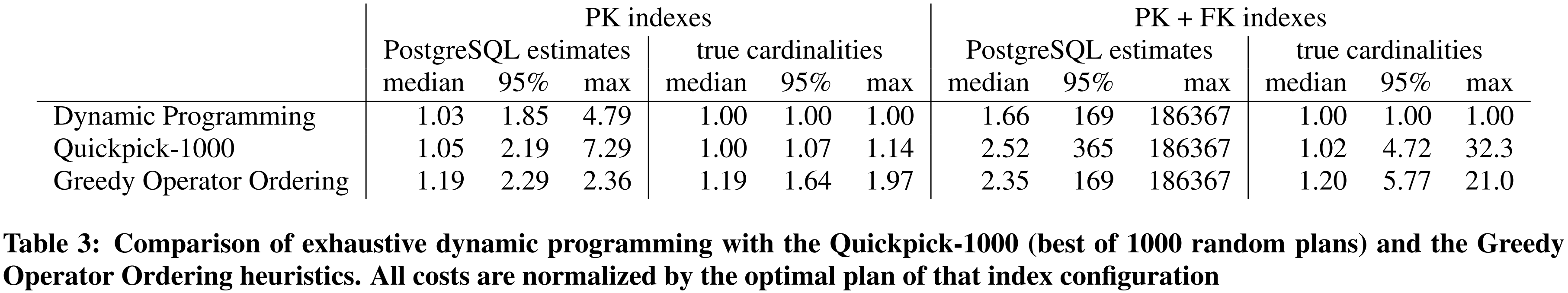
- Bushy trees important to consider



	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'
```

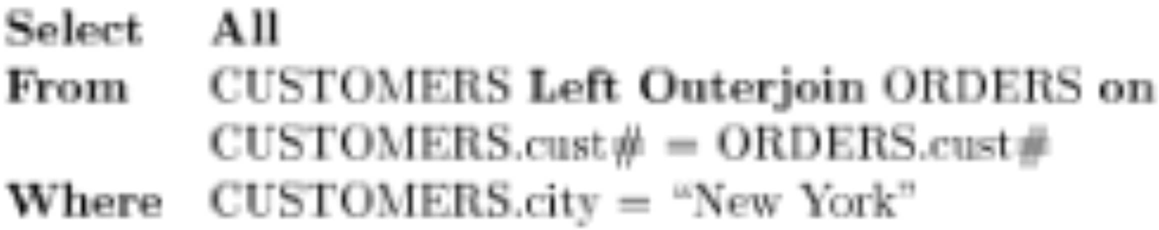
  - Unclear how one can benefit from capturing this correlation (even if one could)
  - Need a new operator or access method

Outline

- Query evaluation techniques for large databases
- Skew avoidance strategies
- Query compilation
- Vectorization
- Query Optimization: Overview
- How good are the query optimizers, really?
- Reordering for Outerjoins

Outerjoins

- Preserve the tuples even when there are no matches

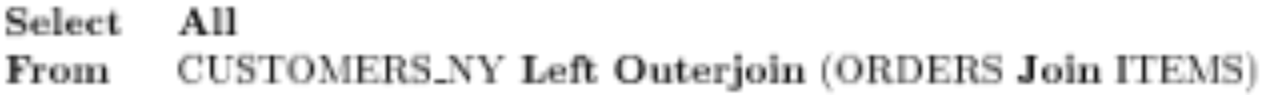


```
Select   All
From     CUSTOMERS Left Outerjoin ORDERS on
         CUSTOMERS.cust# = ORDERS.cust#
Where    CUSTOMERS.city = "New York"
```

- Many common use cases
  - Database merging
  - Hierarchical views (e.g., in document stores)
  - Nested queries
  - …
- However, unclear how to reorder/commute joins and outerjoins in general

Outerjoins: Optimization Opportunities

- Consider:



```
Select   All
From     CUSTOMERS_NY Left Outerjoin (ORDERS Join ITEMS)
```

- Possibly only a few tuples in CUSTOMERS_NY
  - So better to do: customers_NY LOJ orders first
- But no way to combine with “items” after that:
  - (customers_ny LOJ orders) JOIN items: will throw away customers without orders (the join attribute will be NULL)
  - (customers_ny LOJ orders) LOJ items: will keep orders with no items

- Basically need a new join operator with desired behavior

Some Terminology

- **Outerunion:** Union of two relations that don’t have the same schema, done by adding extra columns with NULLs as needed

$$R_1 \uplus R_2 = (R_1 \times \{\mathrm{null}_{S_2 - S_1}\}) \cup (R_2 \times \{\mathrm{null}_{S_1 - S_2}\}),$$

- **Outerjoins using Outerunions:**

$$R_1 \xrightarrow{p} R_2 = (R_1 \overset{p}{\bowtie} R_2) \uplus (R_1 - \pi_{\mathrm{sch}(R_1)}(R_1 \overset{p}{\bowtie} R_2)).$$

$$R_1 \overset{p}{\leftrightarrow} R_2 = (R_1 \overset{p}{\bowtie} R_2) \uplus (R_1 - \pi_{\mathrm{sch}(R_1)}(R_1 \overset{p}{\bowtie} R_2)) \uplus (R_2 - \pi_{\mathrm{sch}(R_2)}(R_1 \overset{p}{\bowtie} R_2)).$$

- **Example:**

R

A	B	C
a	c	b
d	f	a
c	d	b

S

C	D	E
b	g	a
d	a	f

$R \overset{R.C=S.C}{\bowtie} S$

A	B	R.C	S.C	D	E
a	c	b	b	g	a
c	d	b	b	g	a

$R \overset{R.C=S.C}{\rightarrow} S$

A	B	R.C	S.C	D	E
a	c	b	b	g	a
c	d	b	b	g	a
d	f	a			

$R \overset{R.C=S.C}{\leftrightarrow} S$

A	B	R.C	S.C	D	E
a	c	b	b	g	a
c	d	b	b	g	a
d	f	a			
			d	a	f

Outerjoin Simplification

- Some basic identities
  - Note: can't push down the selection on the RHS here

$$R_1 \xrightarrow{p_1 \wedge p_2} R_2 = R_1 \xrightarrow{p_1} (\sigma_{p_2} R_2), \text{ if } \mathrm{sch}(p_2) \subseteq \mathrm{sch}(R_2). \quad (1)$$

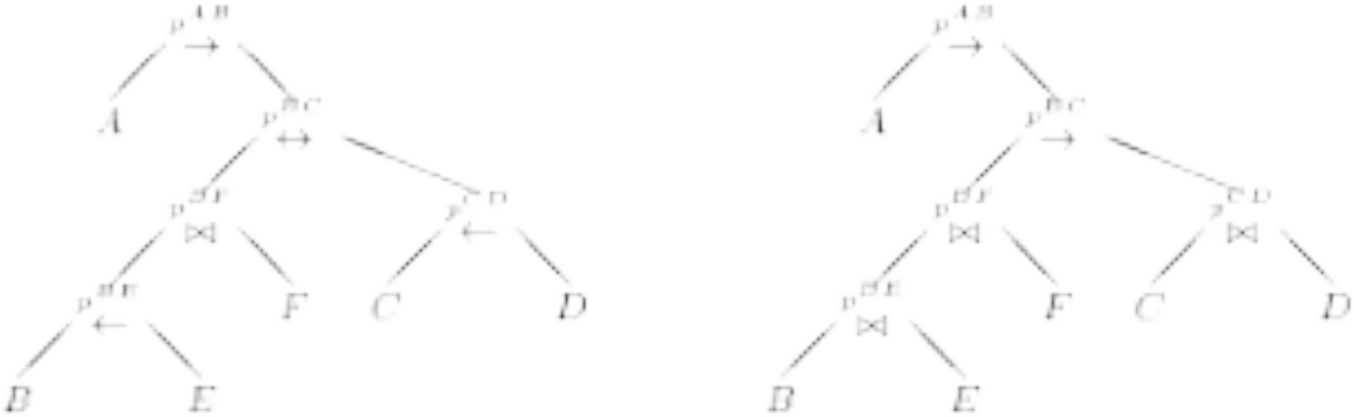
$$\sigma_{p_1}(R_1 \xrightarrow{p_2} R_2) = (\sigma_{p_1} R_1) \xrightarrow{p_2} R_2, \text{ if } \mathrm{sch}(p_1) \subseteq \mathrm{sch}(R_1). \quad (2)$$

- A predicate "rejects" NULLs on attributes A if it evaluates to FALSE or UNKNOWN if all attributes in A are NULL
- Can replace outerjoins with joins if a subsequent predicates rejects NULLs

$$\sigma_{p_1}(R_1 \xrightarrow{p_2} R_2) = \sigma_{p_1}(R_1 \overset{p_2}{\bowtie} R_2), \text{ if } p_1 \text{ rejects nulls on } \mathrm{sch}(R_2).$$

$$\sigma_{p_1}(R_1 \overset{p_2}{\leftrightarrows} R_2) = \sigma_{p_1}(R_1 \xleftarrow{p_2} R_2), \text{ if } p_1 \text{ rejects nulls on } \mathrm{sch}(R_2).$$

Outerjoin Simplification: Example



(a) Original query.

(b) Simplified query.

- The top join condition (p^{AB}) will not evaluate to true for any tuple with all B attributes = NULL
  - This assume p^{AB} is on an attribute coming from B alone

Join and Outerjoin Associativity

$$(R_1 \overset{p^{12}}{\bowtie} R_2) \overset{p^{13} \wedge p^{23}}{\bowtie} R_3 = R_1 \overset{p^{12} \wedge p^{13}}{\bowtie} (R_2 \overset{p^{23}}{\bowtie} R_3). \tag{5}$$

$$(R_1 \overset{p^{12}}{\bowtie} R_2) \overset{p^{23}}{\rightarrow} R_3 = R_1 \overset{p^{12}}{\bowtie} (R_2 \overset{p^{23}}{\rightarrow} R_3). \tag{6}$$

$$(R_1 \overset{p^{12}}{\rightarrow} R_2) \overset{p^{23}}{\rightarrow} R_3 = R_1 \overset{p^{12}}{\rightarrow} (R_2 \overset{p^{23}}{\rightarrow} R_3), \tag{7}$$

if p^{23} rejects nulls on $\mathrm{sch}(R_2)$.

$$(R_1 \overset{p^{12}}{\leftarrow} R_2) \overset{p^{23}}{\rightarrow} R_3 = R_1 \overset{p^{12}}{\leftarrow} (R_2 \overset{p^{23}}{\rightarrow} R_3). \tag{8}$$

$$(R_1 \overset{p^{12}}{\leftrightarrow} R_2) \overset{p^{23}}{\leftrightarrow} R_3 = R_1 \overset{p^{12}}{\leftrightarrow} (R_2 \overset{p^{23}}{\leftrightarrow} R_3), \tag{9}$$

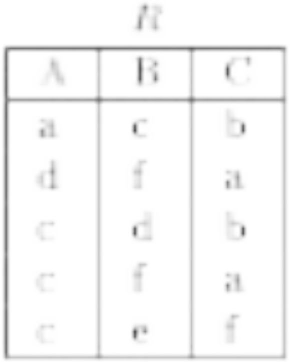
if p^{12} and p^{23} reject nulls on $\mathrm{sch}(R_2)$.

$$(R_1 \overset{p^{12}}{\leftrightarrow} R_2) \overset{p^{23}}{\rightarrow} R_3 = R_1 \overset{p^{12}}{\leftrightarrow} (R_2 \overset{p^{23}}{\rightarrow} R_3), \tag{10}$$

if p^{23} rejects nulls on $\mathrm{sch}(R_2)$.

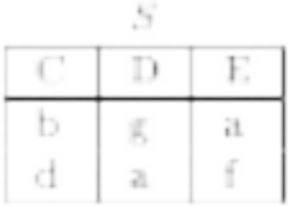
Generalized Outerjoin

- A new binary join operator that "preserves" a subset of the attributes
- Not much harder to implement than a standard outerjoin, but may need additional duplicate elimination step



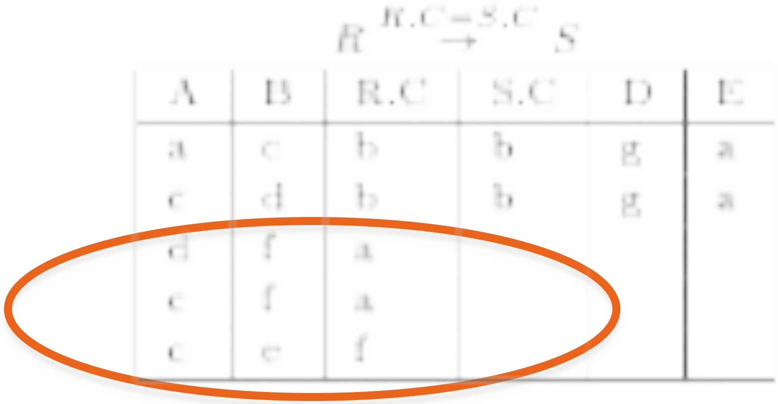
R

A	B	C
a	c	b
d	f	a
c	d	b
c	f	a
c	e	f



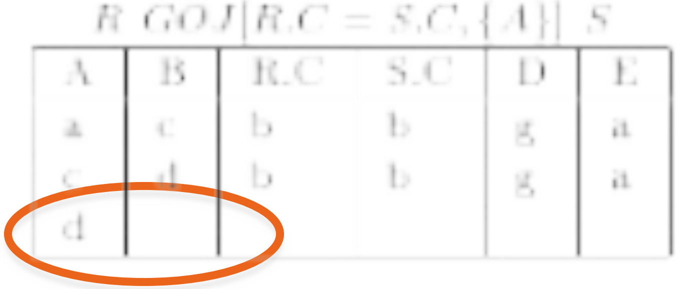
S

C	D	E
b	g	a
d	a	f



$R \xrightarrow{R.C=S.C} S$

A	B	R.C	S.C	D	E
a	c	b	b	g	a
c	d	b	b	g	a
d	f	a			
c	f	a			
c	e	f			



$R\ GOJ[R.C = S.C, \{A\}]\ S$

A	B	R.C	S.C	D	E
a	c	b	b	g	a
c	d	b	b	g	a
d					

Generalized Outerjoin

Two new equivalences

$$R_1 \overset{p^{12}}{\rightarrow} (R_2 \overset{p^{23}}{\bowtie} R_3) = (R_1 \overset{p^{12}}{\rightarrow} R_2)\ GOJ[p^{23}, \mathrm{sch}(R_1)]\ R_3, \quad \text{if } p^{23} \text{ rejects nulls on } \mathrm{sch}(R_2). \tag{11}$$

$$R_1 \overset{p^{12}}{\leftrightarrow} (R_2 \overset{p^{23}}{\bowtie} R_3) = (R_1 \overset{p^{12}}{\leftrightarrow} R_2)\ GOJ[p^{23}, \mathrm{sch}(R_1)]\ R_3, \quad \text{if } p^{23} \text{ rejects nulls on } \mathrm{sch}(R_2). \tag{12}$$

Helps solve the original problem

$$\mathrm{CUSTOMERS\_NY} \rightarrow (\mathrm{ORDERS} \bowtie \mathrm{ITEMS}) =$$
$$(\mathrm{CUSTOMERS\_NY} \rightarrow \mathrm{ORDERS})\ \mathrm{GOJ}[\mathrm{sch}(\mathrm{CUSTOMERS\_NY})]\ \mathrm{ITEMS}.$$

Completeness

- For a class of “simple” queries, the set of equivalences is “complete”
  - i.e., you can get to any possible and correct reordering by repeated application of the rules

- Bunch of caveats though…
  - So in practice, possible that we miss out on some optimization opportunities

Aside: Semijoins and Distributed Settings

R = 10M Tuples

A	B	C	D	E
1	..	..	..	..
2	..	..	..	..
3	..	..	..	..
8	..	..	..	..
8	..	..	..	..
9	..	..	..	..

Site A

But only a small number of join results

Say: 1M tuples from R match with 1M tuples of S

S = 10M Tuples

A	F	G	H	I
2	..	..	..	..
4	..	..	..	..
8	..	..	..	..
9	..	..	..	..
9	..	..	..	..
9	..	..	..	..

Site B

Aside: Semijoins and Distributed Settings

R = 10M Tuples

A	B	C	D	E
1	..	..	..	..
2	..	..	..	..
3	..	..	..	..
8	..	..	..	..
8	..	..	..	..
9	..	..	..	..

Site A

Option 1: Send R to Site B
Comm Cost = 50M

Option 2: Send S to Site A
Comm Cost = 50M

Option 3:
(1) Send R.A to Site B
(2) Send matching tuples to Site A

Comm Cost = 10M + 1M*5
= 15M

S = 10M Tuples

A	F	G	H	I
2	..	..	..	..
4	..	..	..	..
8	..	..	..	..
9	..	..	..	..
9	..	..	..	..
9	..	..	..	..

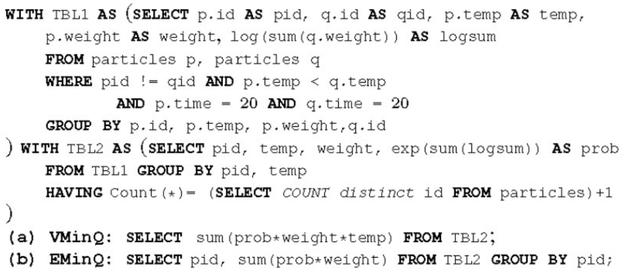
Site B

Outline

- Query evaluation techniques for large databases
- Skew avoidance strategies
- Query compilation
- Vectorization
- Query Optimization: Overview
- How good are the query optimizers, really?
- Reordering for Outerjoins
- Query Rewriting
    - Starburst
    - Unnesting arbitrary queries
    - APPLY (SQL Server)

Why Query Rewrite?

- Many queries are written in a way that forces a procedural execution
  - Use of WITH clause or Views to simplify
  - Procedural code easier for users to write
  - Modern frameworks/query languages often not that declarative
  - Automated translation of other DSLs into SQL
  - Program synthesis?



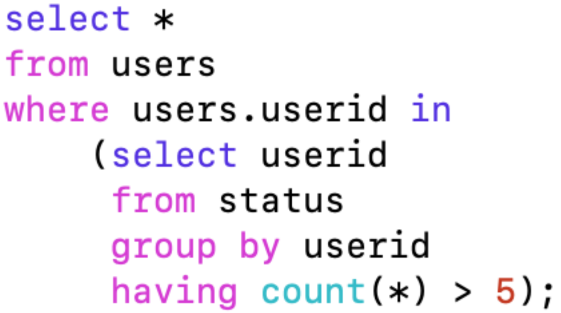
```
WITH TBL1 AS (SELECT p.id AS pid, q.id AS qid, p.temp AS temp,
    p.weight AS weight, log(sum(q.weight)) AS logsum
    FROM particles p, particles q
    WHERE pid != qid AND p.temp < q.temp
           AND p.time = 20 AND q.time = 20
    GROUP BY p.id, p.temp, p.weight,q.id
) WITH TBL2 AS (SELECT pid, temp, weight, exp(sum(logsum)) AS prob
    FROM TBL1 GROUP BY pid, temp
    HAVING Count(*)= (SELECT COUNT distinct id FROM particles)+1
)
(a) VMinQ: SELECT sum(prob*weight*temp) FROM TBL2;
(b) EMinQ: SELECT pid, sum(prob*weight) FROM TBL2 GROUP BY pid;
```

- Harder for optimizers to deal with
  - Join order optimization usually goes block-by-block ➔ significant benefits in reducing the number of blocks
  - Redundant DISTINCTs etc., lead to unnecessary work

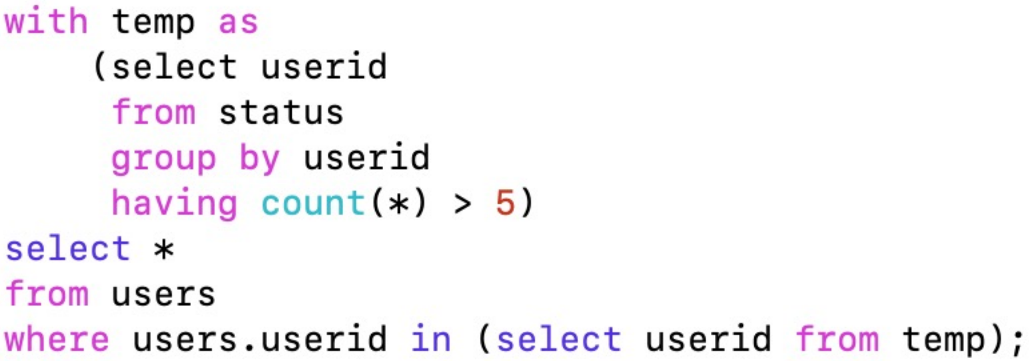
Two Main Issues

- Merging of select blocks
  - Different “blocks” get created because of:
    - WITH, Views
    - Table expressions in FROM (e.g., select * from R, (select S.A, max(S.B) from S group by S.A) X)…)
    - Table expressions in WHERE/SELECT/HAVING etc. (e.g., where R.A in (select S.A from S))
    - Scalar expressions in WHERE/SELECT/HAVING etc. (e.g., where R.A = (select max(S.A) from S)))
- Correlations Across Blocks
  - When an ”lower” block refers to an “upper” block
  - Forces a “dependent” ”nested-loops” execution
    - For every tuple in the outer block, the inner block is executed

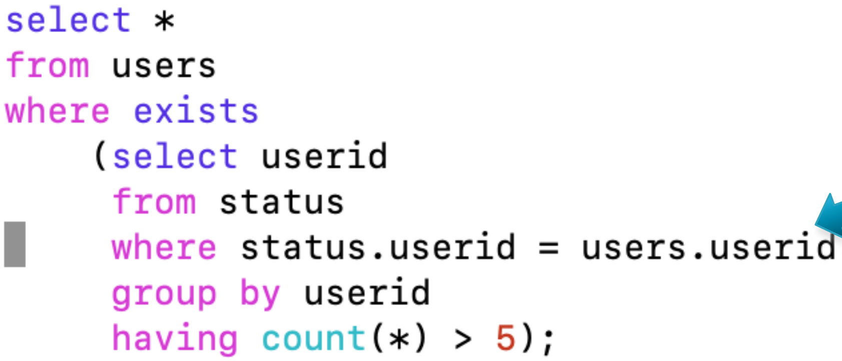
Example



```
select *
from users
where users.userid in
    (select userid
     from status
     group by userid
     having count(*) > 5);
```



```
with temp as
    (select userid
     from status
     group by userid
     having count(*) > 5)
select *
from users
where users.userid in (select userid from temp);
```



```
select *
from users
where exists
    (select userid
     from status
     where status.userid = users.userid
     group by userid
     having count(*) > 5);
```

Correlated

Join Operators and Implementations

R(A, B), and S(B, C)

R Natural Join S

```
ht = dict()
for r in R:
      if r.B in ht:
           ht[r.B].append(r)
      else:
           ht[r.B] = [r]
for s in S:
      for r in ht.get(s.B, []):
           yield (s, r)
```

Most other join operators built as minor modifications (special cases) of this basic code

S Semi Join R (build on R)

```
ht = set()
for r in R:
      ht.add(r.B)
for s in S:
      if s.B in ht:
           yield s
```

R Semi Join S (build on R)

```
ht = dict()
for r in R:
      if r.B not in ht:
           ht[r.B].append(r)
      else:
           ht[r.B] = [r]
for s in S:
      for r in ht[s.B]:
            yield r
      ht[s.B] = [] -- avoid
                     duplicates
```

Join Operators and Implementations

R(A, B), and S(B, C)

R Natural Join S

```
ht = dict()
for r in R:
      if r.B in ht:
           ht[r.B].append(r)
      else:
           ht[r.B] = [r]
for s in S:
      for r in ht.get(s.B, []):
           yield (s, r)
```

S Anti Join R

```
ht = set()
for r in R:
      ht.add(r.B)
for s in S:
      if s.B not in ht:
           yield s
```

R Anti Join S

```
ht = dict()
for r in R:
      if r.B not in ht:
           ht[r.B].append(r)
      else:
           ht[r.B] = [r]
for s in S:
      ht[s.B] = [] --- remove r
for r in ht.values():
       yield r
```

Join Operators and Implementations

R(A, B), and S(B, C)

R Natural Join S

```
ht = dict()
for r in R:
      if r.B in ht:
           ht[r.B].append(r)
      else:
           ht[r.B] = [r]
for s in S:
      for r in ht.get(s.B, []):
           yield (s, r)
```

R Full Outer Join S

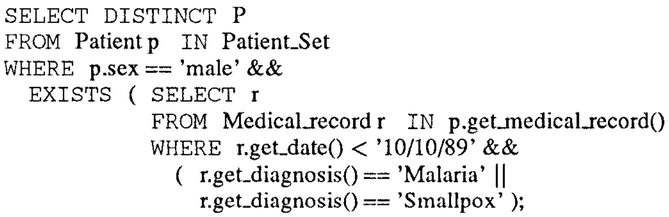
```
ht = dict()
found_set = set()
for r in R:
      if r.B in ht:
           ht[r.B].append(r)
      else:
           ht[r.B] = [r]
for s in S:
      if s.B in ht:
          found_set.add(s.B)
          for r in ht[s.B]:
               yield (s, r)
      else:
          yield (NULLS, s)

for x in ht:
     if x not in found_set:
          for r in ht[x]:
              yield(r, NULLS)
```

Outline

- Query evaluation techniques for large databases
- Skew avoidance strategies
- Query compilation
- Vectorization
- Query Optimization: Overview
- How good are the query optimizers, really?
- Reordering for Outerjoins
- Query Rewriting
  - Starburst
  - Unnesting (de-correlating) arbitrary queries
  - APPLY (SQL Server)

Query Rewrite in Starburst

- Goals:
  - Make queries more “declarative” by removing procedural parts
  - Perform natural heuristics like predicate pushdown



```
SELECT DISTINCT P
FROM Patient p   IN Patient_Set
WHERE p.sex == 'male' &&
  EXISTS ( SELECT r
           FROM Medical_record r   IN p.get_medical_record()
           WHERE r.get_date() < '10/10/89' &&
             ( r.get_diagnosis() == 'Malaria' ||
               r.get_diagnosis() == 'Smallpox' );
```

A query with a “path expression”

Option 1:

- For each male patient
  - Go through their records and check if any matches

Basically a “nested-loops join”

Highly inefficient if very few matching records

Option 2:

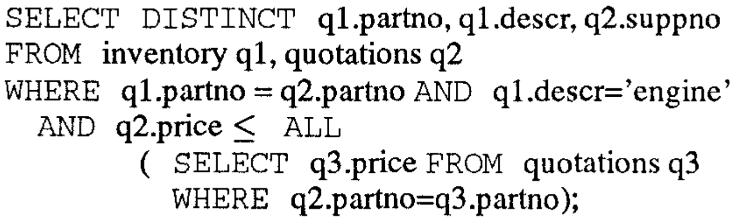
Better to find the matching records first, and then look for patients

Need to convert to a join first

select distinct p
from patient p in patient_set, record r in record_set
where [[r conditions]] and p.sex = ‘male’ and [[join conditions]]

Query Graph Model (QGM)

- Internal representation – goal is to have as few "select boxes" as possible
- Most optimizers today use "operator tree" representations



```
SELECT DISTINCT q1.partno, q1.descr, q2.suppno
FROM inventory q1, quotations q2
WHERE q1.partno = q2.partno AND q1.descr='engine'
  AND q2.price ≤ ALL
        ( SELECT q3.price FROM quotations q3
          WHERE q2.partno=q3.partno);
```

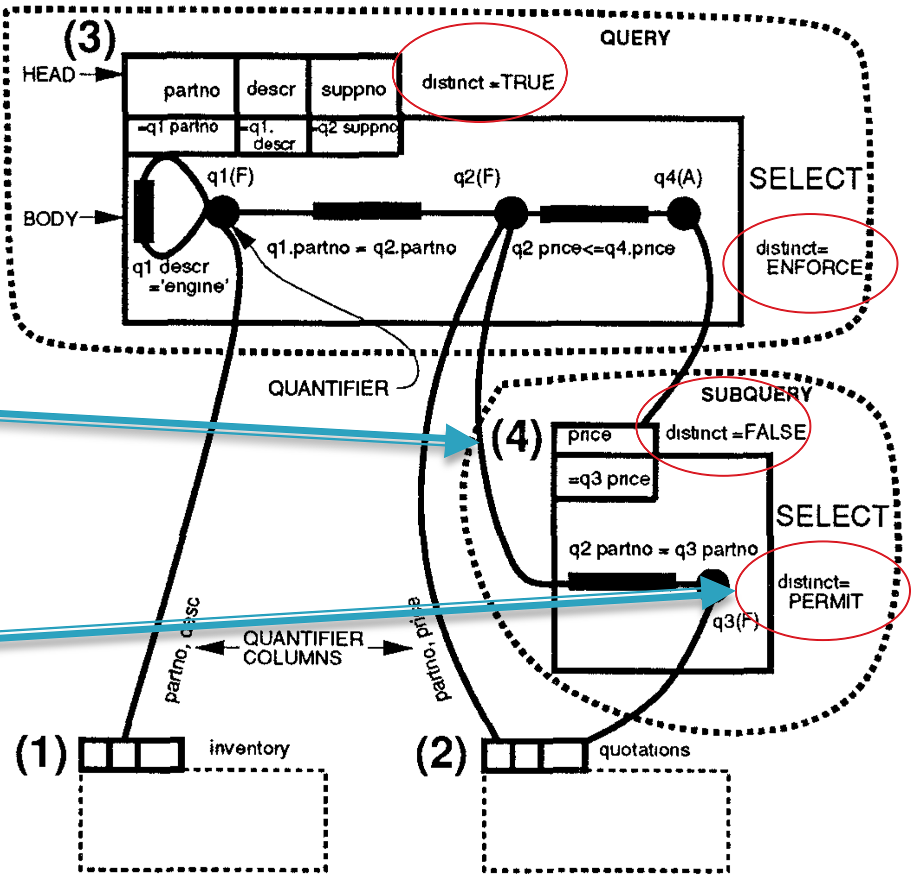


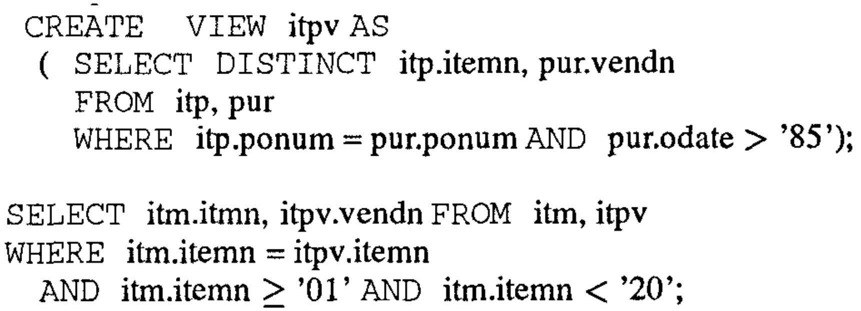
Figure 1: Example QGM graph

Correlation

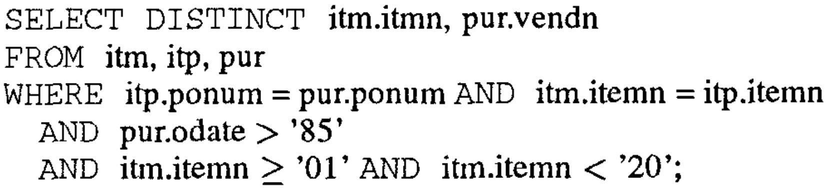
Need to be careful with "distincts"
In some cases, the above operator doesn't care
- Can use that for optimization

Rewrite Rules: SELMERGE

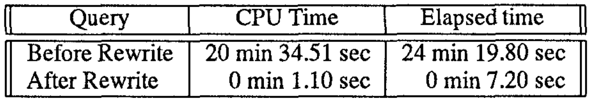
- Take two select boxes connected by “F” quantifier, and merge them
  - Typical example: A “view” or “select expression” in FROM



```
CREATE  VIEW  itpv AS
 ( SELECT DISTINCT  itp.itemn, pur.vendn
   FROM  itp, pur
   WHERE  itp.ponum = pur.ponum AND  pur.odate > ’85’);

SELECT  itm.itmn, itpv.vendn FROM  itm, itpv
WHERE  itm.itemn = itpv.itemn
  AND  itm.itemn ≥ ’01’ AND  itm.itemn < ’20’;
```



```
SELECT DISTINCT  itm.itmn, pur.vendn
FROM  itm, itp, pur
WHERE  itp.ponum = pur.ponum AND  itm.itemn = itp.itemn
  AND  pur.odate > ’85’
  AND  itm.itemn ≥ ’01’ AND  itm.itemn < ’20’;
```



Query	CPU Time	Elapsed time
Before Rewrite	20 min 34.51 sec	24 min 19.80 sec
After Rewrite	0 min 1.10 sec	0 min 7.20 sec

Table 3: Example 1, Before and After Rewrite

- Need to be careful with DISTINCTS
  - upper.body.distinct = PERMIT ➔ don’t care
  - upper.head.distinct = TRUE and upper.body.distinct = ENFORCE ➔ upper box is required to eliminate duplicates, so we are okay
  - Other cases a bit more complicated
  - Can’t apply the rule if:
    - upper.head.distinct = FALSE, upper.body.distinct = PRESERVE, and lower.body.distinct = ENFORCE: No easy way to reconcile the DISTINCT requirements for the lower and the upper box after merge
    - Need to apply some of the other rules before we can use SELMERGE

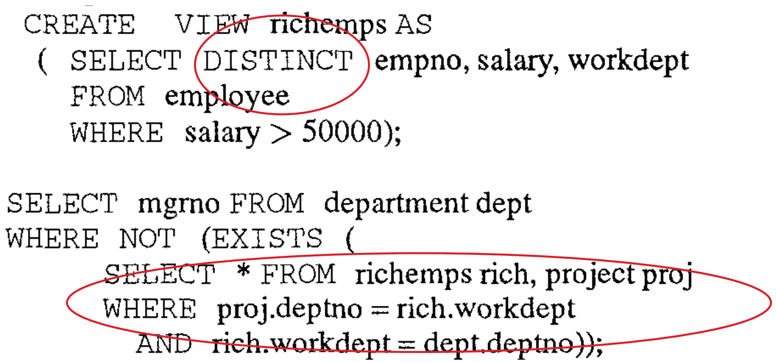
Rewrite Rules: Distinct pullup/pushdown

- Distinct Pullup: If all F quantifiers are guaranteed to either:
  - Have a single tuple
  - Have a primary key of the attributes in the output
  - THEN, we can set: head.distinct = TRUE, and body.distinct = PRESERVE
  - Why? More “distinct = true” will make it easier to do merges

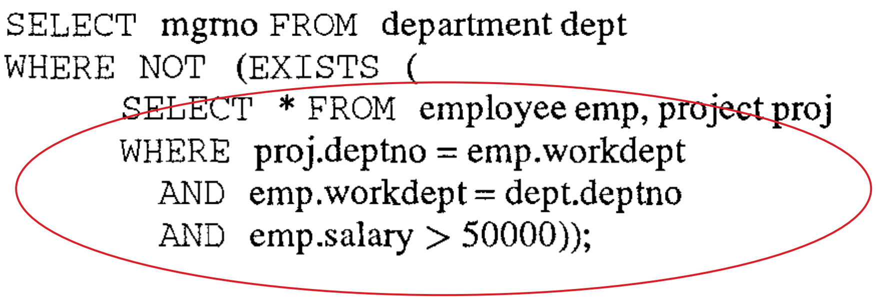
- Distinct pulldown
  - If a box has body.distinct = PERMIT or ENFORCE, it can tell its children to set body.distinct = PERMIT
    - i.e., the upper box either doesn’t care about duplicates, or is going to enforce DISTINCT → lower boxes don’t need to worry about it
  - If all parents of a box don’t care about duplicates (i.e., have body.distinct = PERMIT), then we can set body.distinct = PERMIT for this box as well

Rewrite Rules: EorAPDFR

- If a quantifier has type = E (existential) or A (all), then for the lower box: set body.distinct = PERMIT

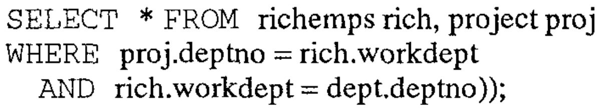


```
CREATE  VIEW richemps AS
 ( SELECT DISTINCT empno, salary, workdept
   FROM employee
   WHERE salary > 50000);

SELECT mgrno FROM department dept
WHERE NOT (EXISTS (
      SELECT * FROM richemps rich, project proj
      WHERE proj.deptno = rich.workdept
        AND rich.workdept = dept.deptno));
```



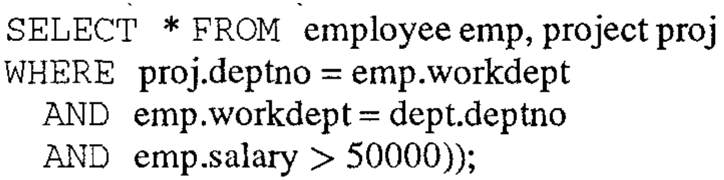
```
SELECT mgrno FROM department dept
WHERE NOT (EXISTS (
      SELECT * FROM employee emp, project proj
      WHERE proj.deptno = emp.workdept
        AND emp.workdept = dept.deptno
        AND emp.salary > 50000));
```

Possible that:



```
SELECT * FROM richemps rich, project proj
WHERE proj.deptno = rich.workdept
  AND rich.workdept = dept.deptno));
```

!=



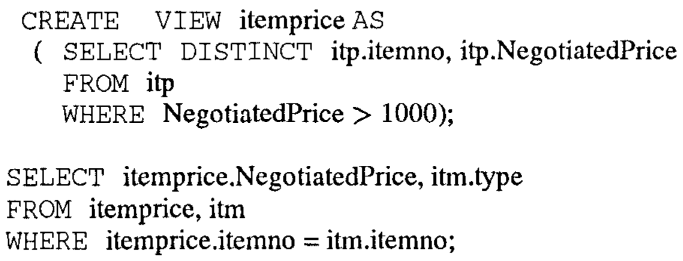
```
SELECT * FROM employee emp, project proj
WHERE proj.deptno = emp.workdept
  AND emp.workdept = dept.deptno
  AND emp.salary > 50000));
```

But EXISTS ➔ we don't care about duplicates

Bad example? DISTINCT in the view doesn't matter because of empno

Rewrite Rules: ADDKEYS

- The one situation that SELMERGE doesn’t handle for F quantifiers
    - upper.head.distinct = FALSE, upper.body.distinct = PRESERVE, and lower.body.distinct = ENFORCE
        - The output of upper box has duplicates, Upper box ”maintains” duplicates, Lower box removes duplicates
    - No way to reconcile if you merge



```
CREATE  VIEW itemprice AS
 ( SELECT DISTINCT itp.itemno, itp.NegotiatedPrice
   FROM itp
   WHERE NegotiatedPrice > 1000);

SELECT itemprice.NegotiatedPrice, itm.type
FROM itemprice, itm
WHERE itemprice.itemno = itm.itemno;
```

!=

```
select itp.NegotiatedPrice, itm.type
from itp, itm
where itp.itemno = itm.itemno
      and itp.NegotiatedPrice> 1000
```

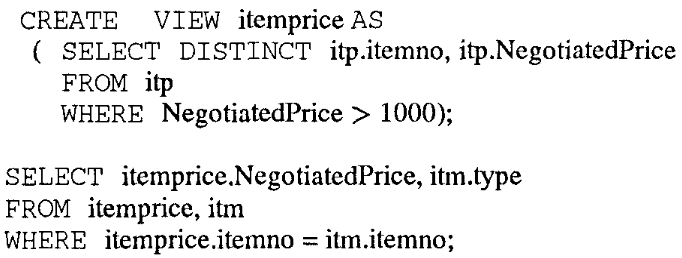
OR

```
select DISTINCT itp.NegotiatedPrice, itm.type
from itp, itm
where itp.itemno = itm.itemno
      and itp.NegotiatedPrice> 1000
```

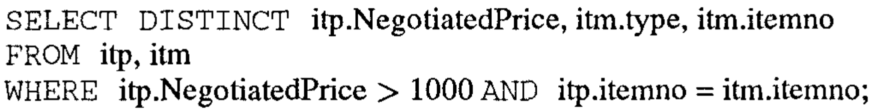
For the same itemno, there may be two tuples in itp with the same NegotiatedPrice
-- The view eliminates the duplicates, so the final query will only have one instance
-- Right query 1 doesn’t ➔ the final result would have two entries corresponding to that itemno
-- Right query 2 doesn’t handle the case when two different items of the same type have the same NegotiatedPrice

Rewrite Rules: ADDKEYS

- The one situation that SELMERGE doesn't handle for F quantifiers
  - upper.head.distinct = FALSE, upper.body.distinct = PRESERVE, and lower.body.distinct = ENFORCE
    - The output of upper box has duplicates, Upper box "maintains" duplicates, Lower box removes duplicates
  - No way to reconcile if you merge
- Instead, add the KEY to the upper box output



```
 CREATE  VIEW itemprice AS
  ( SELECT DISTINCT itp.itemno, itp.NegotiatedPrice
    FROM itp
    WHERE NegotiatedPrice > 1000);

SELECT itemprice.NegotiatedPrice, itm.type
FROM itemprice, itm
WHERE itemprice.itemno = itm.itemno;
```



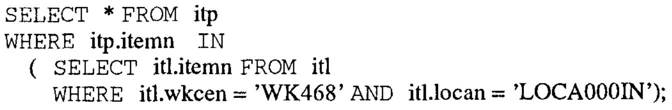
```
SELECT DISTINCT itp.NegotiatedPrice, itm.type, itm.itemno
FROM itp, itm
WHERE itp.NegotiatedPrice > 1000 AND itp.itemno = itm.itemno;
```

The two queries are not the same
But we can throw away the itemno at the end

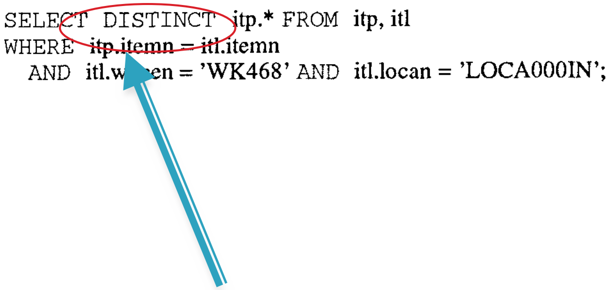
Use of this rule guarantees that F boxes can always be merged

Rewrite Rules: E to F Quantifier Conversion

- Under certain conditions, an quantifier of type E can be converted to F, and then merged using SELMERGE

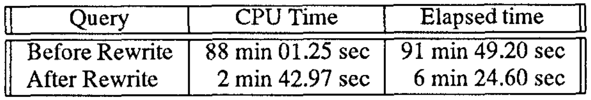


```
SELECT * FROM itp
WHERE itp.itemn IN
  ( SELECT itl.itemn FROM itl
    WHERE itl.wkcen = 'WK468' AND itl.locan = 'LOCA000IN');
```



```
SELECT DISTINCT itp.* FROM itp, itl
WHERE itp.itemn = itl.itemn
  AND itl.wkcen = 'WK468' AND itl.locan = 'LOCA000IN';
```

Without this distinct, we may get duplicates in the output



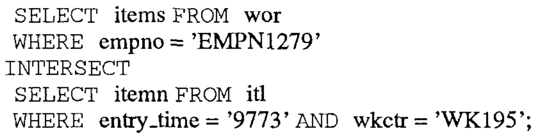
Query	CPU Time	Elapsed time
Before Rewrite	88 min 01.25 sec	91 min 49.20 sec
After Rewrite	2 min 42.97 sec	6 min 24.60 sec

Table 10: Example 4, Before and After Rewrite

Intersect to Exists

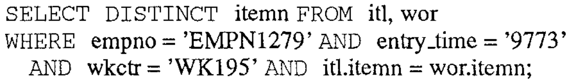
- Replace an INTERSECT to an EXISTS, which can then be merged using SELMERGE

Should be itemn



```
 SELECT  items FROM  wor
 WHERE  empno = ’EMPN1279’
INTERSECT
 SELECT  itemn FROM  itl
 WHERE  entry_time = ’9773’ AND  wkctr = ’WK195’;
```

```
SELECT DISTINCT itemn from wor
WHERE empno = ‘EMPN1279’ AND
          EXISTS (SELECT * FROM itl
                    WHERE entry_time = ‘9773’ AND
                              wkctr = ‘WK195’ AND
                              wor.itemn = itl.itemn);
```



```
SELECT  DISTINCT  itemn FROM  itl, wor
WHERE  empno = ’EMPN1279’ AND  entry_time = ’9773’
   AND  wkctr = ’WK195’ AND  itl.itemn = wor.itemn;
```

Starburst Rule Engine

- Each rule is a pair of functions in a procedural language like C
  - a condition function that checks a condition
  - a action function that takes an action
- All functions read/manipulate the QGM (a C object), so need to be written in C
- Need more logic to decide how to fire rules, when to stop etc.
- No discussion of “aggregates” or other types of joins
  - The basic set of rules doesn’t cover those

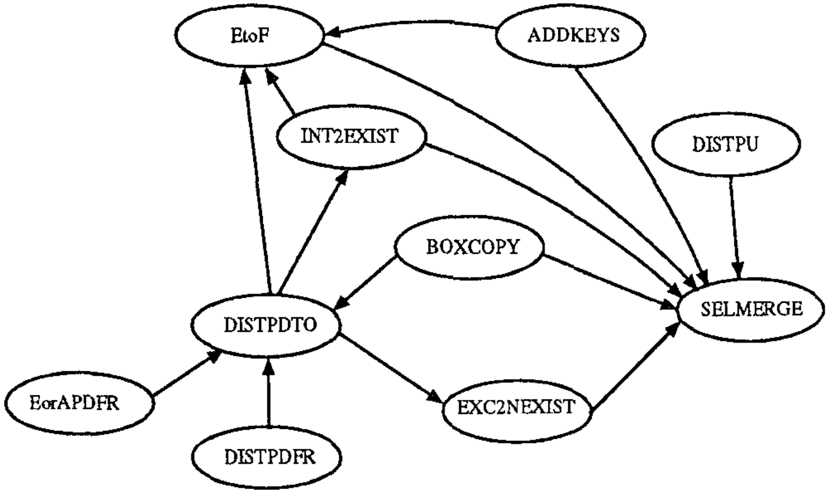


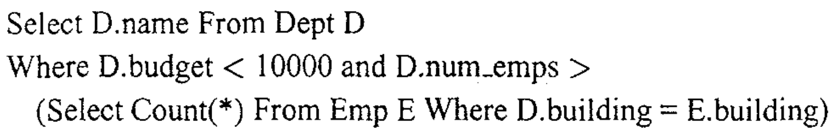
Figure 2: Triggering Interactions Between Rules

Outline

- Query evaluation techniques for large databases
- Skew avoidance strategies
- Query compilation
- Vectorization
- Query Optimization: Overview
- How good are the query optimizers, really?
- Reordering for Outerjoins
- Query Rewriting
  - Starburst
  - Unnesting (de-correlating) arbitrary queries
  - APPLY (SQL Server)

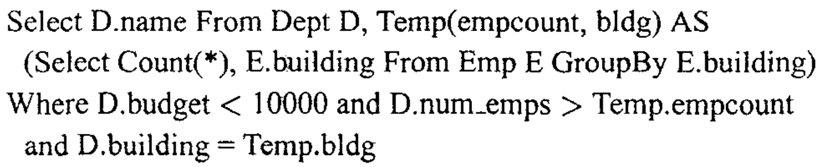
Decorrelating Subqueries

- Discussion from: “Complex Query Decorrelation”; ICDE 1996
- Correlations lead to “nested-loops” execution



```
Select D.name From Dept D
Where D.budget < 10000 and D.num_emps >
  (Select Count(*) From Emp E Where D.building = E.building)
```

[Kim; 1982]

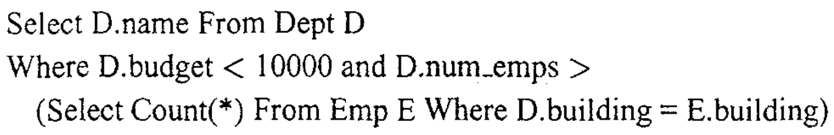


```
Select D.name From Dept D, Temp(empcount, bldg) AS
 (Select Count(*), E.building From Emp E GroupBy E.building)
Where D.budget < 10000 and D.num_emps > Temp.empcount
 and D.building = Temp.bldg
```

- Only works for simple equality correlated predicates
- COUNT computation must be done for all departments, whether or not they have budget < 10000
- “Count bug”: doesn’t handle departments from a building with 0 employees

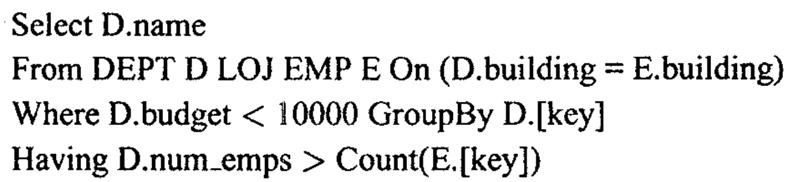
Decorrelating Subqueries

- Discussion from: “Complex Query Decorrelation”; ICDE 1996
- Correlations lead to “nested-loops” execution



```
Select D.name From Dept D
Where D.budget < 10000 and D.num_emps >
  (Select Count(*) From Emp E Where D.building = E.building)
```

[Dayal; 1987]

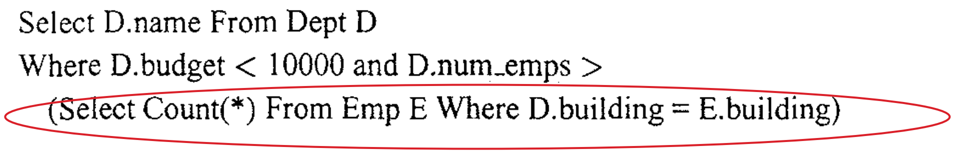


```
Select D.name
From DEPT D LOJ EMP E On (D.building = E.building)
Where D.budget < 10000 GroupBy D.[key]
Having D.num_emps > Count(E.[key])
```

- Outerjoin takes care of buildings with 0 employees
- Collapsed query -- better optimizations possible
- Still may lead to redundant work if multiple departments share a building
- Limited application

”Magic” Decorrelation

- Discussion from: “Complex Query Decorrelation”; ICDE 1996
- A correlated subquery can be seen as a function call



```
Select D.name From Dept D
Where D.budget < 10000 and D.num_emps >
  (Select Count(*) From Emp E Where D.building = E.building)
```

Say we had a function CS(x)

- x is a building
- CS(x) returns the number of employees in that building

Further, let’s say the function is a “table”

TEMP(x, CS_of_x), with x as the key

Can write the query as:

```
select D.name from Dept D, TEMP
where D.budget < 1000 and
          D.num_emps > TEMP. CS_of_x and
          D.building = TEMP.x
```

Just need to figure out how to compute TEMP

e.g.,

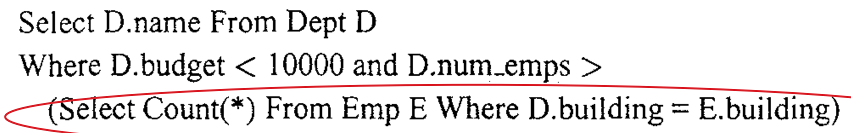
```
with TEMP as
   (select E.building, count(*) as CS_of_x
    from Emp E
    group by E.building)
```

Not efficient: computes the counts for all buildings, not just the necessary one

Also: doesn’t handle buildings with 0 employees

"Magic" Decorrelation

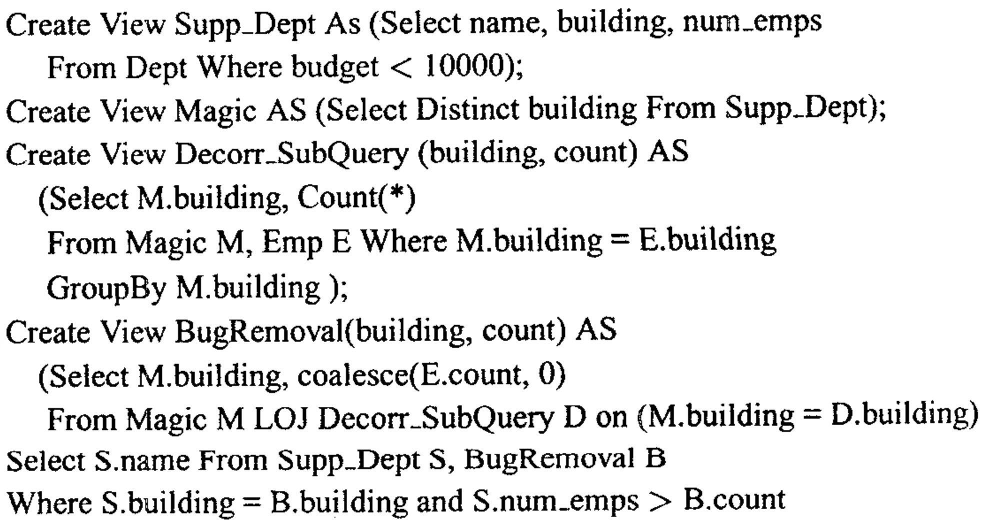
- Discussion from: "Complex Query Decorrelation"; ICDE 1996
- Instead, somehow restrict the computation only to the ones of interest



```
Select D.name From Dept D
Where D.budget < 10000 and D.num_emps >
   (Select Count(*) From Emp E Where D.building = E.building)
```

Only do the counts for buildings of interest

Need to add the buildings with 0 count



```
Create View Supp_Dept As (Select name, building, num_emps
    From Dept Where budget < 10000);
Create View Magic AS (Select Distinct building From Supp_Dept);
Create View Decorr_SubQuery (building, count) AS
   (Select M.building, Count(*)
    From Magic M, Emp E Where M.building = E.building
    GroupBy M.building );
Create View BugRemoval(building, count) AS
   (Select M.building, coalesce(E.count, 0)
    From Magic M LOJ Decorr_SubQuery D on (M.building = D.building)
Select S.name From Supp_Dept S, BugRemoval B
Where S.building = B.building and S.num_emps > B.count
```

Unnesting Arbitrary Queries

- Goal: A generic approach to ”decorrelate” any subquery

- students: {[id, name, major, year, . . .]}
- exams: {[sid, course, curriculum, date, . . .]}

```
Q1: select s.name,e.course
    from   students s,exams e
    where  s.id=e.sid and
           e.grade=(select min(e2.grade)
                    from exams e2
                    where s.id=e2.sid)
```

```
Q1’: select s.name,e.course
     from   students s,exams e,
            (select e2.sid as id, min(e2.grade) as best
             from exams e2
             group by e2.sid) m
     where  s.id=e.sid and m.id=s.id and
            e.grade=m.best
```

Correlation makes it harder to merge
Makes this a “dependent” join

Unnesting Arbitrary Queries

- A hard-to-decorrelate subquery

- students: {[id, name, major, year, . . .]}
- exams: {[sid, course, curriculum, date, . . .]}

```
Q2:
select s.name, e.course
from   students s, exams e
where  s.id=e.sid and
  (s.major = ’CS’ or s.major = ’Games Eng’) and
  e.grade>=(select avg(e2.grade)+1                          --one grade worse
            from exams e2                                   --than the average grade
            where s.id=e2.sid or                            --of exams taken by
                  (e2.curriculum=s.major and                          --him/her or taken
                  s.year>e2.date))                                    --by elder peers
```

General Approach to Unnesting

- Similar to “magic sets” or “sideways information passing”
- In essence, duplicate the input tables sufficiently

```
Q2:
select s.name, e.course
from   students s, exams e
where  s.id=e.sid and
  (s.major = ’CS’ or s.major = ’Games Eng’) and
  e.grade>=(select avg(e2.grade)+1
            from exams e2
            where s.id=e2.sid or
                  (e2.curriculum=s.major and
                  s.year>e2.date))
```

General Approach to Unnesting

- Similar to “magic sets” or “sideways information passing”
- In essence, duplicate the input tables sufficiently

```
Q2:
select s.name, e.course
from   students s, exams e
where  s.id=e.sid and
  (s.major = ’CS’ or s.major = ’Games Eng’) and
  e.grade>=  X.avg_grade_plus_1
             and X.s-id = s.id
             and X.s-year = s.year
             and X.s-major = s.major
```

```
(select avg(e2.grade)+1
 from exams e2
 where s.id=e2.sid or
         (e2.curriculum=s.major and
         s.year>e2.date))
```

This is a scalar function with three parameters: s.id, s.year, s.major

What if we had a table:
X(s-id, s-year, s-major, avg_grade_plus_1)
with primary key (s-id, s-year, s-major) ?

NOTE: In general, a scalar function can be seen as a “lookup table” with the parameter(s) as the key

General Approach to Unnesting

- Similar to “magic sets” or “sideways information passing”
- In essence, duplicate the input tables sufficiently

```
(select avg(e2.grade)+1
 from exams e2
 where s.id=e2.sid or
        (e2.curriculum=s.major and
        s.year>e2.date))
```

How to compute X?

```
X
==
select s.id, s.year, s.major,
    avg(e2.grade) + 1 as avg_grade_plus_1
from exams e2, students s
where s.id = e2.sid or
      (e2.curriculum = s.major and
        s.year > e2.date))
group by s.id, s.year, s.major
```

This is a scalar function with three parameters: s.id, s.year, s.major

What if we had a table:
X(s-id, s-year, s-major, avg_grade_plus_1)
with primary key (s-id, s-year, s-major) ?

NOTE: In general, a scalar function can be seen as a “lookup table” with the parameter(s) as the key

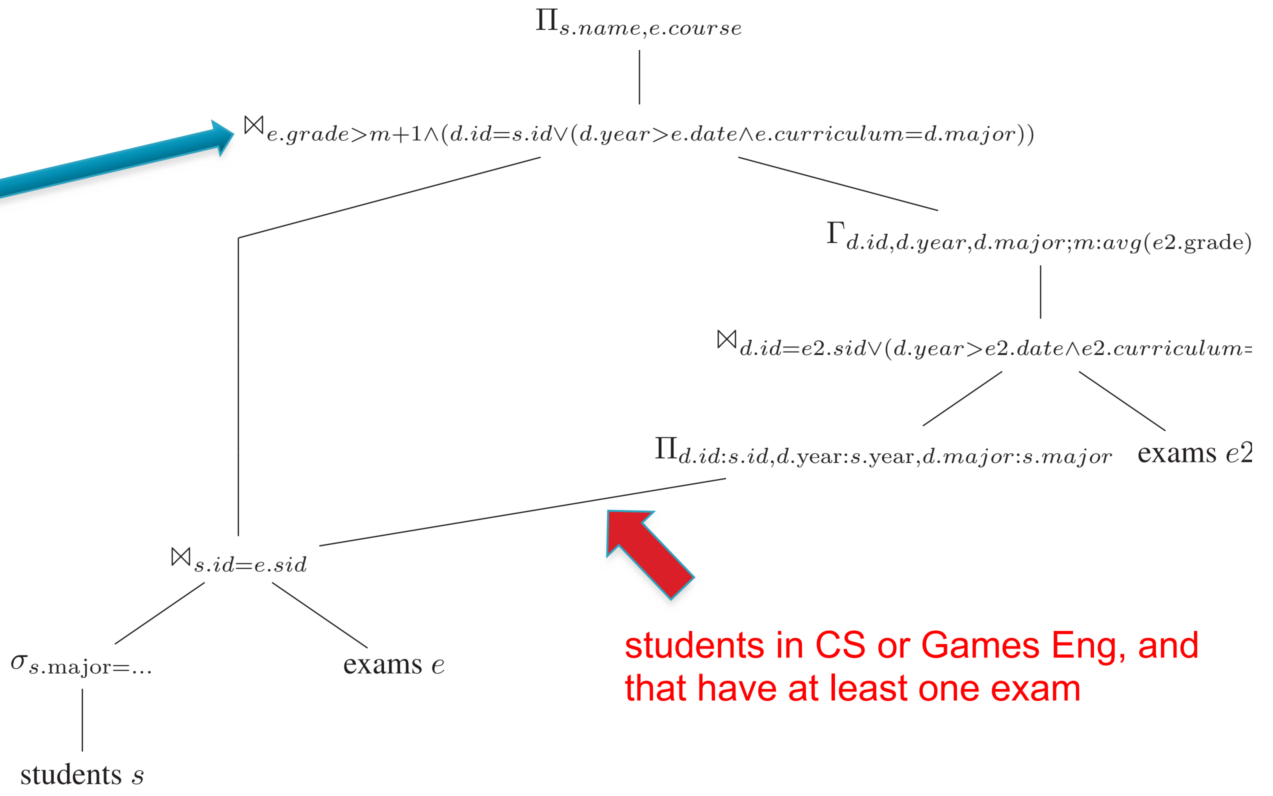
Can optimize this further
Only need to do this for students in CS or Games Eng, and that have at least one exam

General Approach to Unnesting

- Similar to “magic sets” or “sideways information passing”
- In essence, duplicate the input tables sufficiently

This seems wrong to me. No reason why e.curriculum = d.major or d.year > e.date

The only join conditions there should be:
e.grade > m+1 and
d.id = s.id and
d.major = s.major and
d.year = s.year



$\Pi_{s.name,e.course}$
$\bowtie_{e.grade>m+1 \wedge (d.id=s.id \vee (d.year>e.date \wedge e.curriculum=d.major))}$
$\Gamma_{d.id,d.year,d.major;m:avg(e2.grade)}$
$\bowtie_{d.id=e2.sid \vee (d.year>e2.date \wedge e2.curriculum=}$
$\Pi_{d.id:s.id,d.year:s.year,d.major:s.major}$
exams $e2$
$\bowtie_{s.id=e.sid}$
$\sigma_{s.major=...}$
exams e
students s

students in CS or Games Eng, and that have at least one exam

Figure 9: Query Q2, Optimized Form with Sideways Information Passing

Machinery

- Dependent Join Notation
  - For every tuple *t1* in *T1*, for every tuple *t2* in *T2(t1):*
    - Output *(t1, t2)*, if *p(t1, t2) is* True

$$T_1 \ltimes\!\!\rtimes_p T_2 \quad := \quad \{t_1 \circ t_2 | t_1 \in T_1 \wedge t_2 \in T_2(t_1) \wedge p(t_1 \circ t_2)\}.$$

- Apply notation

$$\mathrm{R}\ \mathcal{Apply}_{\mathcal{JN}}\,\mathrm{E}(\boldsymbol{r}) = \mathcal{UA}_{r\,\in\,\mathrm{R}}\,(\{r\}\ \mathcal{JN}\,\mathrm{E}(\boldsymbol{r})).$$

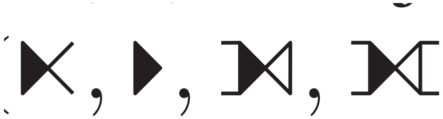
  - For every tuple *r* in *R,* compute *E(r)*, and then “join” with {r} using JN
  - If JN = Crossproduct: then produce a tuple each for each t in E(r) -- as (r, t)
    - An inner join will have an additional predicate to check on (r, t)
  - If JN = Outerjoin: then in addition, if E(r) = empty, produce (r, NULLs)
  - If JN = Semijoin: produce just (r) if E(r) is NOT empty
  - If JN = Antijoin: produce just (r) if E(r) is empty

Machinery

- Dependent Join Notation
  - For every tuple *t1* in *T1*, for every tuple *t2* in *T2(t1):*
    - Output (*t1, t2), if p(t1, t2) is* True

$$T_1 \bowtie_p T_2 \quad := \quad \{t_1 \circ t_2 | t_1 \in T_1 \wedge t_2 \in T_2(t_1) \wedge p(t_1 \circ t_2)\}.$$

- Dependent join notation can also be extended to handle the other types of joins



- Group by operator

$$\Gamma_{A;a:f}(e) \quad := \quad \{x \circ (a : f(y)) | x \in \Pi_A(e) \wedge y = \{z | z \in e \wedge \forall a \in A : x.a = z.a\}\}$$

Simple Unnesting

- Rules that allow removing the “dependence” in some cases
  - The other paper has a list of rules like this

```
select ...
from   lineitem l1 ...
where  exists (select *
               from lineitem l2
               where l2.l_orderkey = l1.l_orderkey)
```

$$l_1 \ltimes (\sigma_{l_1.okey=l_2.okey}(l_2))$$

$$l_1 \ltimes_{l_1.okey=l_2.okey} (l_2)$$

Note: SQL doesn’t have SEMIJOIN so
SQL representation can’t be simplified

General Approach

- Push dependent joins down into the query until it can be simplified
- Requires duplication of expressions

$$T_1 \mathbin{\text{⧑}}_p T_2 \quad \equiv \quad T_1 \bowtie_{p \wedge T_1 =_{\mathcal{A}(D)} D} (D \mathbin{\text{⧑}} T_2)$$

$$D := \Pi_{\mathcal{F}(T_2) \cap \mathcal{A}(T_1)}(T_1).$$

Nested loops execution: For every tuple in T1, find the “parameters” (values of A(T1)) and run T2 subquery

Instead, let’s find all possible sets of parameters (“magic set”) and
Run T2 subquery for all of them simultaneously

If T2 doesn’t have any free parameters any more (if F(T2) = A(T1)), then this dependent join is a normal join

$$D \mathbin{\text{⧑}} T \quad \equiv \quad D \bowtie T \quad \text{if} \quad \mathcal{F}(T) \cap \mathcal{A}(D) = \emptyset.$$

Example

- Push dependent joins down into the query until it can be simplified
- Requires duplication of expressions

Still a dependent join

Needs a couple more transformations to do the join first and then the aggregate

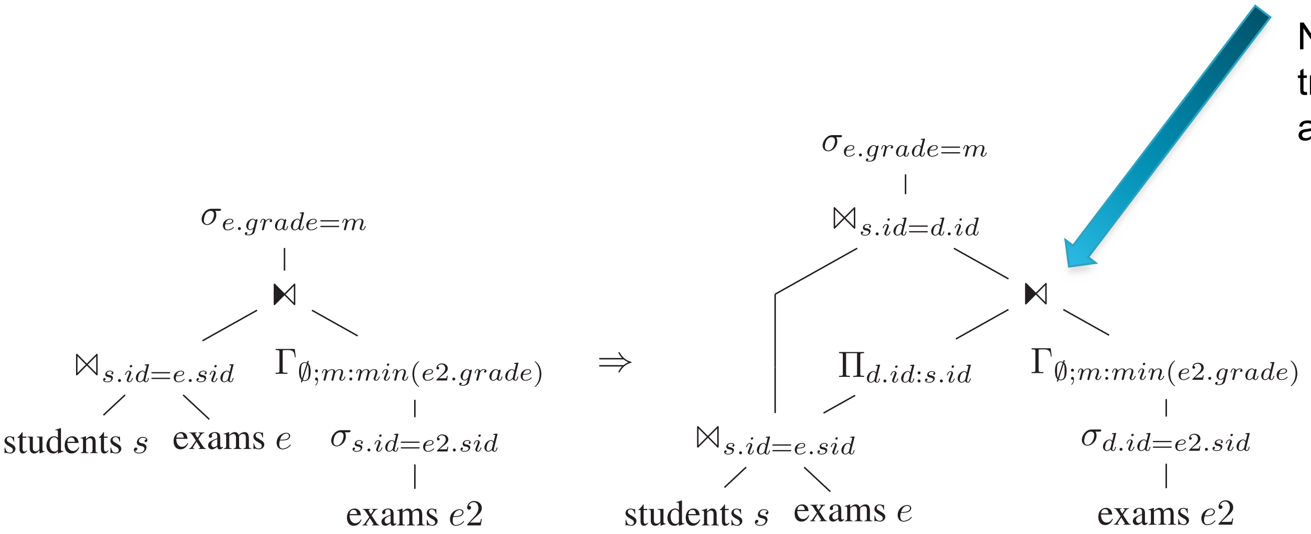


Figure 1: Example Application of Dependent Join “Push-Down”

Transformation Rules

$$D \mathbin{\blacktriangleright\!\!\triangleleft} T \quad\equiv\quad D \bowtie T \quad \text{if} \quad \mathcal{F}(T) \cap \mathcal{A}(D) = \emptyset.$$

$$D \mathbin{\blacktriangleright\!\!\triangleleft} \sigma_p(T_2) \quad\equiv\quad \sigma_p(D \mathbin{\blacktriangleright\!\!\triangleleft} T_2).$$

$$D \mathbin{\blacktriangleright\!\!\triangleleft} (T_1 \bowtie_p T_2) \equiv \begin{cases} (D \mathbin{\blacktriangleright\!\!\triangleleft} T_1) \bowtie_p T_2 & : \quad \mathcal{F}(T_2) \cap \mathcal{A}(D) = \emptyset \\ T_1 \bowtie_p (D \mathbin{\blacktriangleright\!\!\triangleleft} T_2) & : \quad \mathcal{F}(T_1) \cap \mathcal{A}(D) = \emptyset \\ (D \mathbin{\blacktriangleright\!\!\triangleleft} T_1) \bowtie_{p \wedge \text{natural join } D} (D \mathbin{\blacktriangleright\!\!\triangleleft} T_2) & : \quad \text{otherwise.} \end{cases}$$

$$D \mathbin{\blacktriangleright\!\!\triangleleft} (T_1 ⟕_p T_2) \equiv \begin{cases} (D \mathbin{\blacktriangleright\!\!\triangleleft} T_1) ⟕_p T_2 & : \quad \mathcal{F}(T_2) \cap \mathcal{A}(D) = \emptyset \\ (D \mathbin{\blacktriangleright\!\!\triangleleft} T_1) ⟕_{p \wedge \text{natural join } D} (D \mathbin{\blacktriangleright\!\!\triangleleft} T_2) & : \quad \text{otherwise.} \end{cases}$$

$$D \mathbin{\blacktriangleright\!\!\triangleleft} (T_1 ⟗_p T_2) \;\equiv\; (D \mathbin{\blacktriangleright\!\!\triangleleft} T_1) ⟗_{p \wedge \text{natural join } D} (D \mathbin{\blacktriangleright\!\!\triangleleft} T_2).$$

$$D \mathbin{\blacktriangleright\!\!\triangleleft} (T_1 \ltimes_p T_2) \equiv \begin{cases} (D \mathbin{\blacktriangleright\!\!\triangleleft} T_1) \ltimes_p T_2 & : \quad \mathcal{F}(T_2) \cap \mathcal{A}(D) = \emptyset \\ (D \mathbin{\blacktriangleright\!\!\triangleleft} T_1) \ltimes_{p \wedge \text{natural join } D} (D \mathbin{\blacktriangleright\!\!\triangleleft} T_2) & : \quad \text{otherwise.} \end{cases}$$

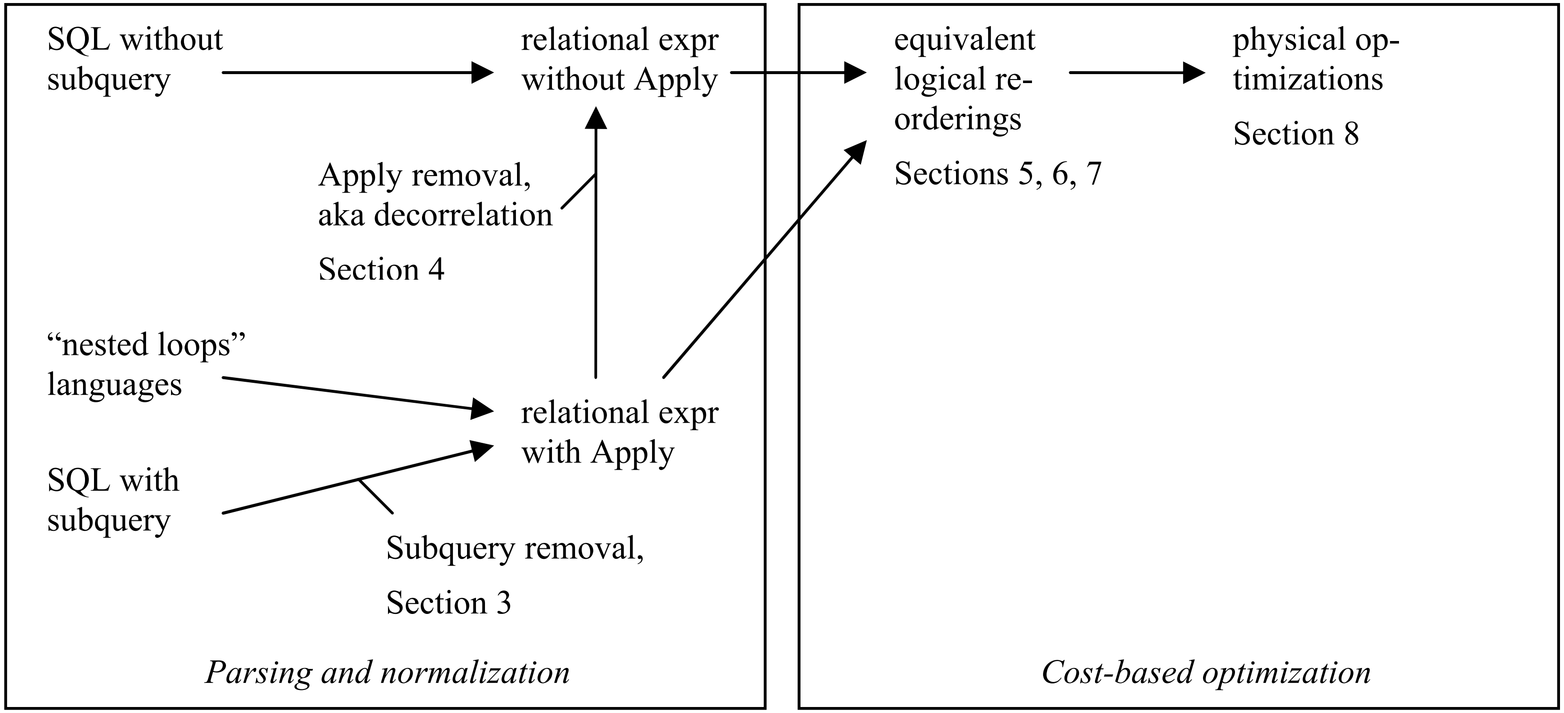
$$D \mathbin{\blacktriangleright\!\!\triangleleft} (T_1 \triangleright_p T_2) \equiv \begin{cases} (D \mathbin{\blacktriangleright\!\!\triangleleft} T_1) \triangleright_p T_2 & : \quad \mathcal{F}(T_2) \cap \mathcal{A}(D) = \emptyset \\ (D \mathbin{\blacktriangleright\!\!\triangleleft} T_1) \triangleright_{p \wedge \text{natural join } D} (D \mathbin{\blacktriangleright\!\!\triangleleft} T_2) & : \quad \text{otherwise.} \end{cases}$$

… and more

Outline

- Query evaluation techniques for large databases
- Skew avoidance strategies
- Query compilation
- Vectorization
- Query Optimization: Overview
- How good are the query optimizers, really?
- Reordering for Outerjoins
- Query Rewriting
  - Starburst
  - Unnesting (de-correlating) arbitrary queries
  - APPLY (SQL Server)

Query Processing in SQL Server



SQL without subquery

relational expr without Apply

Apply removal, aka decorrelation

Section 4

"nested loops" languages

relational expr with Apply

SQL with subquery

Subquery removal,

Section 3

Parsing and normalization

equivalent logical re-orderings

Sections 5, 6, 7

physical op-timizations

Section 8

Cost-based optimization

Figure 1: Overview of plan generation for subqueries in SQL Server and structure of the paper

Algebraic Representation

- Apply

$$\mathrm{R}\ \mathcal{Apply}_{\mathcal{JN}}\ \mathrm{E}(\boldsymbol{r}) = \mathcal{UA}_{r\,\in\,\mathrm{R}}\ (\{r\}\ \mathcal{JN}\ \mathrm{E}(\boldsymbol{r})).$$

```
SELECT *, (SELECT C_NAME
FROM CUSTOMER
 WHERE C_CUSTKEY = O_CUSTKEY)
 FROM ORDERS
```

→

ORDERS $\mathcal{Apply}_{\mathcal{OJ}}$ (π [C_NAME] σ [C_CUSTKEY = **O_CUSTKEY**] CUSTOMER)

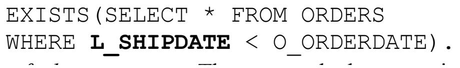
- Available to use in SQL as well -- kinda like “flatMap”

```
SELECT *
FROM MYTABLE
OUTER APPLY CHOP_WORDS(MYTABLE.COL)
```

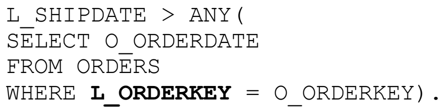
Types of Subqueries

- In the FROM clause -- called "derived table"
  - Also through use of Views, WITH clause
  - Not correlated in most cases
  - Some implementation may allow correlations there, but semantics unclear

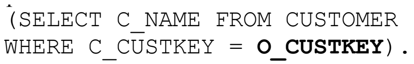
- SELECT, WHERE, etc.
  - Existential
  - Quantified comparison (ALL, ANY)
  - IN, NOT IN
  - Scalar valued



```
EXISTS(SELECT * FROM ORDERS
WHERE L_SHIPDATE < O_ORDERDATE).
```



```
L_SHIPDATE > ANY(
SELECT O_ORDERDATE
FROM ORDERS
WHERE L_ORDERKEY = O_ORDERKEY).
```



```
(SELECT C_NAME FROM CUSTOMER
WHERE C_CUSTKEY = O_CUSTKEY).
```

Subquery Removal

- Basic algorithm (SQREM) relatively straightforward
  - Look for subqueries with correlated variables, and replace with an APPLY and appropriate parameterized expression
- A few special cases to deal with:
  - Queries that may return an exception
    - Need a special Max1Row operator

```
SELECT *, (SELECT C_NAME
FROM CUSTOMER

WHERE C_CUSTKEY = O_CUSTKEY)
FROM ORDERS
```

  - EXISTS and NOT EXISTS replaced with APPLY with Semijoins or Antijoins
  - Conditional execution:
    - CASE WHEN EXISTS(E1(r)) THEN E2(r) ELSE 0 END.

$$(\mathrm{R}\ \mathit{Apply}[\mathit{semijoin,\ probe\ as}\ \mathrm{b}]\ E_1(\mathbf{r}))\ \mathit{Apply}[\mathit{outerjoin,\ pass\text{-}through}\ \mathrm{b}{=}\mathit{1}]\ \mathit{max1row}(E_2(\mathbf{r})).$$

    - Apply-semijoin-probe adds the result as a Boolean to the rows, and pass-through only executes if b = 1

Subquery Removal

- Basic algorithm (SQREM) relatively straightforward
  - Look for subqueries with correlated variables, and replace with an APPLY and appropriate parameterized expression
- A few special cases to deal with:
  - Disjunction of subqueries

$$p(\mathbf{r}) \text{ OR EXISTS}(E_1(\mathbf{r})) \text{ OR EXISTS}(E_2(\mathbf{r})),$$

    - Gets converted to -- CT(1) is a table with a single 1

$$\text{R } \mathcal{A}pply_{SJ} ((\sigma_{p(\mathbf{r})} \text{ CT}(1) \;\mathcal{UA}\; E_1(\mathbf{r}) \;\mathcal{UA}\; E_2(\mathbf{r}))$$

  - Universal quantification
    - Dealing with NULLs can be a pain -- semantics often counter-intuititve
      - e.g., 5 <>ALL {NULL, 5} is FALSE, but 5<>ALL {NULL, 1} is UNKNOWN

Apply Removal

- (1) No increase in the size of the expression

$$\text{ORDERS } \mathcal{A}pply_{\mathcal{OJ}} (\sigma[\text{C\_CUSTKEY} = \textbf{O\_CUSTKEY}] \text{ CUSTOMER}) = \text{ORDERS } \mathcal{OJ} [\text{C\_CUSTKEY} = \text{O\_CUSTKEY}] \text{ CUSTOMER}$$

- (2) Requires duplication

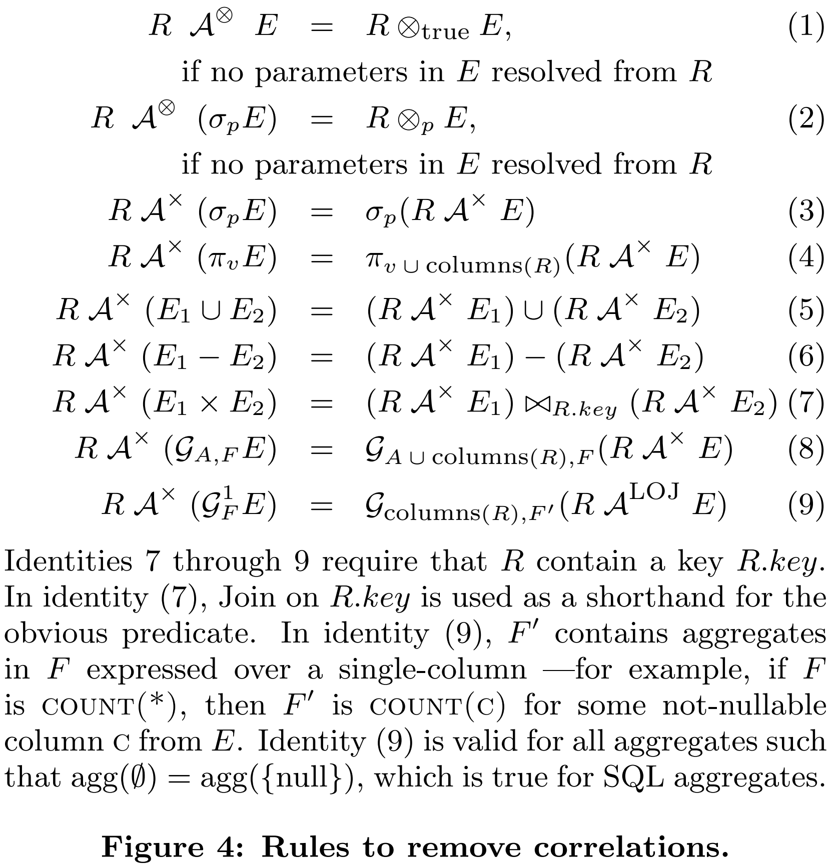
$$\text{R } \mathcal{A}pply_{\mathcal{JN}} (\sigma_{\mathbf{R.a} = S.a} \text{ S}) \ \mathcal{UA} \ \sigma_{\mathbf{R.b} = T.b} \text{ T}) \qquad = \text{R } \mathcal{JN}_{R.a = S.a} \text{ S } \mathcal{UA} \text{ R } \mathcal{JN}_{R.b = T.b} \text{ T}$$

- (3) Not possible: e.g., pass-through, or use of Max1Row
- (4) Opaque (user-defined) Functions

MYTABLE *Apply* CHOP_WORDS(**MYTABLE.COL**)

Apply Removal

- Rules from the previous paper [2001]



$$
\begin{aligned}
R\ \mathcal{A}^{\otimes}\ E &= R \otimes_{\text{true}} E, & (1)\\
&\text{if no parameters in } E \text{ resolved from } R\\
R\ \mathcal{A}^{\otimes}\ (\sigma_p E) &= R \otimes_p E, & (2)\\
&\text{if no parameters in } E \text{ resolved from } R\\
R\ \mathcal{A}^{\times}\ (\sigma_p E) &= \sigma_p (R\ \mathcal{A}^{\times}\ E) & (3)\\
R\ \mathcal{A}^{\times}\ (\pi_v E) &= \pi_{v \,\cup\, \text{columns}(R)} (R\ \mathcal{A}^{\times}\ E) & (4)\\
R\ \mathcal{A}^{\times}\ (E_1 \cup E_2) &= (R\ \mathcal{A}^{\times}\ E_1) \cup (R\ \mathcal{A}^{\times}\ E_2) & (5)\\
R\ \mathcal{A}^{\times}\ (E_1 - E_2) &= (R\ \mathcal{A}^{\times}\ E_1) - (R\ \mathcal{A}^{\times}\ E_2) & (6)\\
R\ \mathcal{A}^{\times}\ (E_1 \times E_2) &= (R\ \mathcal{A}^{\times}\ E_1) \bowtie_{R.key} (R\ \mathcal{A}^{\times}\ E_2) & (7)\\
R\ \mathcal{A}^{\times}\ (\mathcal{G}_{A,F} E) &= \mathcal{G}_{A \,\cup\, \text{columns}(R),F} (R\ \mathcal{A}^{\times}\ E) & (8)\\
R\ \mathcal{A}^{\times}\ (\mathcal{G}^1_F E) &= \mathcal{G}_{\text{columns}(R),F'} (R\ \mathcal{A}^{\text{LOJ}}\ E) & (9)
\end{aligned}
$$

Identities 7 through 9 require that R contain a key $R.key$. In identity (7), Join on $R.key$ is used as a shorthand for the obvious predicate. In identity (9), F' contains aggregates in F expressed over a single-column —for example, if F is COUNT(*), then F' is COUNT(C) for some not-nullable column C from E. Identity (9) is valid for all aggregates such that $\text{agg}(\emptyset) = \text{agg}(\{\text{null}\})$, which is true for SQL aggregates.

Figure 4: Rules to remove correlations.

Magic Sets

- As discussed earlier: the goal is to reduce the number of times a subquery is executed

Reordering Semi-joins and Anti-joins

- SJ and AJ are basically “filters”, and can be moved around as a block

$$(\mathcal{G}_{A,F}R)\ \mathcal{SJ}_{p(A,S)}\ S = \mathcal{G}_{A,F}\ (R\ \mathcal{SJ}_{p(A,\ S)}\ S)$$

$$(\mathcal{G}_{A,F}R)\ \mathcal{ASJ}_{p(A,\ S)}\ S = \mathcal{G}_{A,F}\ (R\ \mathcal{ASJ}_{p(A,\ S)}\ S)$$

- Can sometimes convert to joins, but may introduce additional costs

```
SELECT COUNT(*)
FROM ORDERS
WHERE  O_ORDERDATE = '1995-01-01'
    AND EXISTS(SELECT *
    FROM CUSTOMER, SUPPLIER, LINEITEM
    WHERE
       L_ORDERKEY = O_ORDERKEY
       AND S_SUPPKEY = L_SUPPKEY
       AND C_CUSTKEY = O_CUSTKEY
       AND C_NATIONKEY = S_NATIONKEY
       AND L_SHIPDATE BETWEEN '1995-01-01'
       AND dateadd(dd, 7, '1995-01-01')
```

Option 1: Evaluate “EXISTS” for each Order (nested-loops)

-- obviously a bad idea

Option 2: Do the join between the three relations first

-- those three tables may be large

Option 3: Convert the semijoin to a JOIN

$$R\ \mathcal{SJ}_{p(R,S)}\ S = \mathcal{G}_{key(R),Any(R)}\ (R\ \mathcal{Join}_{p(R,\ S)}\ S)$$

This allows doing the join between the four tables arbitrarily

Only works in some cases

Option 4???: Magic sets

Optimizing General Apply

- Caching
  - If the result is small of the inner loop is small enough, just cache it and reuse it
  - If parameterized, then can build an index using the parameters as the key

- Asynchronous pre-fetch
  - Start retrieving the required rows for the “next” outer tuple

- Sort the outer relation, and execute APPLY in that order
  - Can give some benefits because of natural temporal correlations

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

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

AQP: Overview/Summary

- We will start with a general overview of AQP as presented in a later survey and tutorial

- Then go through the three papers (first two quickly, and the last one in more detail)
  - First two will be covered in the tutorial

Slides Adapted From:

Adaptive Query Processing Tutorial
VLDB 2008

Amol Deshpande, University of Maryland

Zachary G. Ives, University of Pennsylvania

Vijayshankar Raman, IBM Almaden Research Center

Thanks to Joseph M. Hellerstein, University of California, Berkeley

Query Processing: Adapting to the World

Data independence facilitates modern DBMS technology

- Separates specification ("what") from implementation ("how")
- Optimizer maps declarative query → algebraic operations

Platforms, conditions are constantly changing:

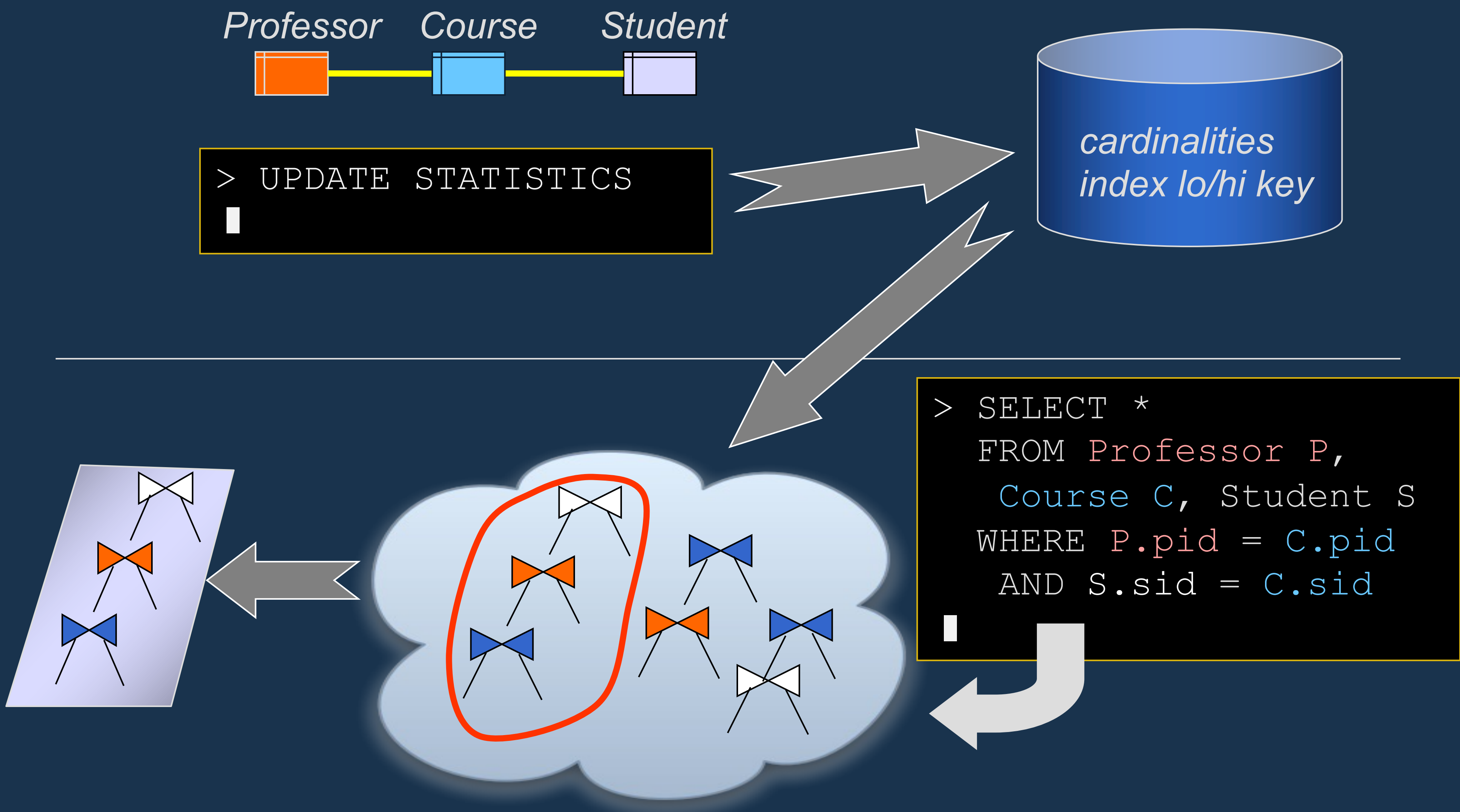
$$\frac{dapp}{dt} << \frac{denv}{dt}$$

Query processing **adapts** implementation to runtime conditions

- Static applications → dynamic environments

Query Optimization and Processing

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



```
> UPDATE STATISTICS
▌
```

```
> SELECT *
  FROM Professor P,
   Course C, Student S
  WHERE P.pid = C.pid
   AND S.sid = C.sid
▌
```

Dynamic Programming + Pruning Heuristics

Traditional Optimization Is Breaking

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, sensor nets, hosting
- Unknown and 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*: ${}^{denv}\!/\!_{dt}$ can be very high

Motivates *intra-query adaptivity*

A Call for Greater Adaptivity

System R adapted query processing as stats were updated

- Measurement/analysis: periodic
- Planning/actuation: once per query
- Improved thru the late 90s (see [Graefe ’93] [Chaudhuri ’98])
  - Better measurement, models, search strategies

INGRES adapted execution many times per query

- Each tuple could join with relations in a different order
- Different plan space, overheads, frequency of adaptivity
  - Didn’t match applications & performance at that time

Recent work considers adaptivity in new contexts

Tutorial Focus

By necessity, we will cover only a piece of the picture here

- Intra-query adaptivity:
  - autonomic / self-tuning optimization [CR'94, CN'97, BC'02, …]
  - robust / least expected cost optimization [CHG'02, MRS+'04, BC'05, ...]
  - parametric or competitive optimization [A'93, INSS'92, CG'94, …]
  - adaptive operators, e.g., memory adaptive sort & hash join [NKT'88, KNT'89, PCL'93a, PCL'93b,…]
- Conventional relations, rather than streams
- Single-site, single query computation

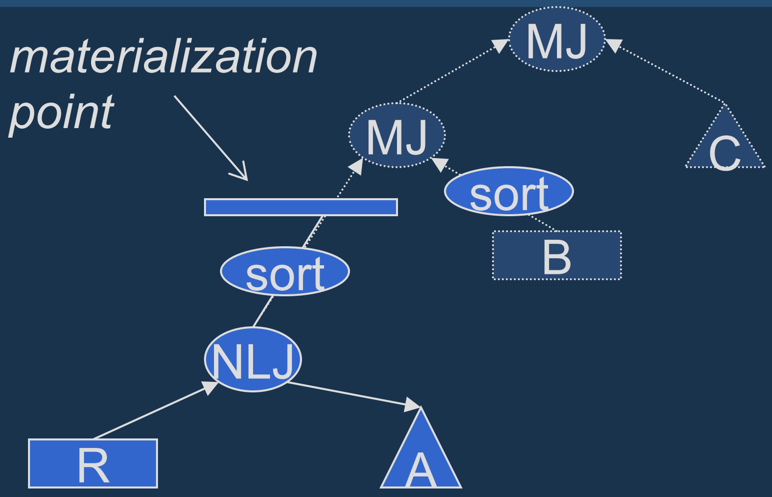
- For more depth, see our survey in now Publishers' *Foundations and Trends in Databases*, Vol. 1 No. 1

Tutorial Outline

- Motivation
- Non-pipelined execution
- Pipelined execution
  - Selection ordering
  - Multi-way join queries
- Putting it all in context
- Recap/open problems

Low-Overhead Adaptivity:
Non-pipelined Execution

Late Binding; Staged Execution



Normal execution: pipelines separated by materialization points

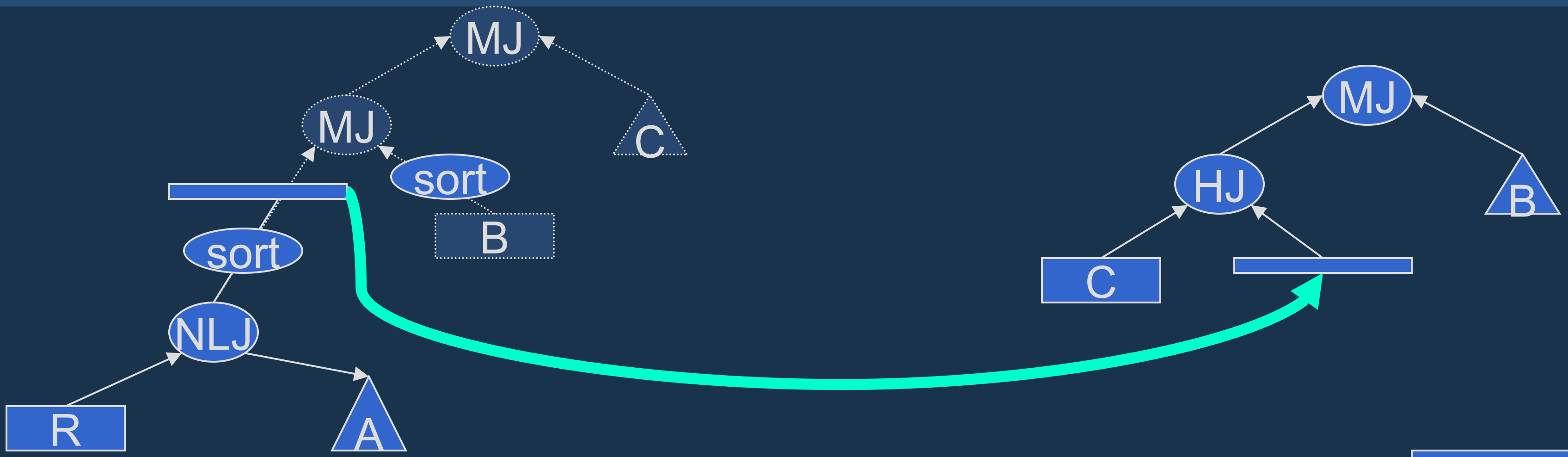
e.g., at a sort, GROUP BY, etc.

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

Balance overhead and opportunities for switching plans

Where?

If actual cardinality is too different from estimated,

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

When?

***Re-optimize* to switch to a new plan**

Try to maintain previous computation during plan switching

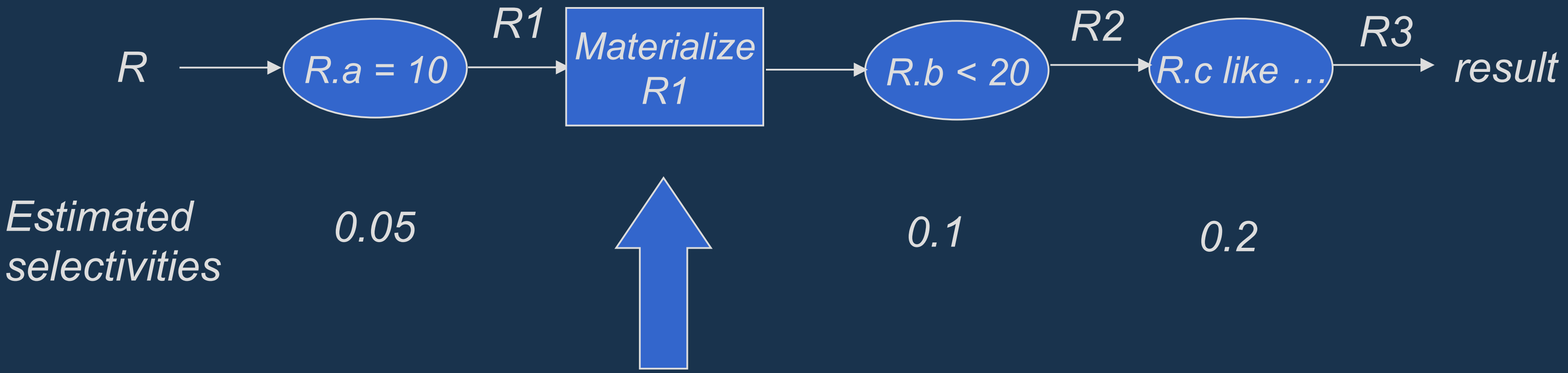
How?

- 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:

<u>*Initial query plan chosen*</u>

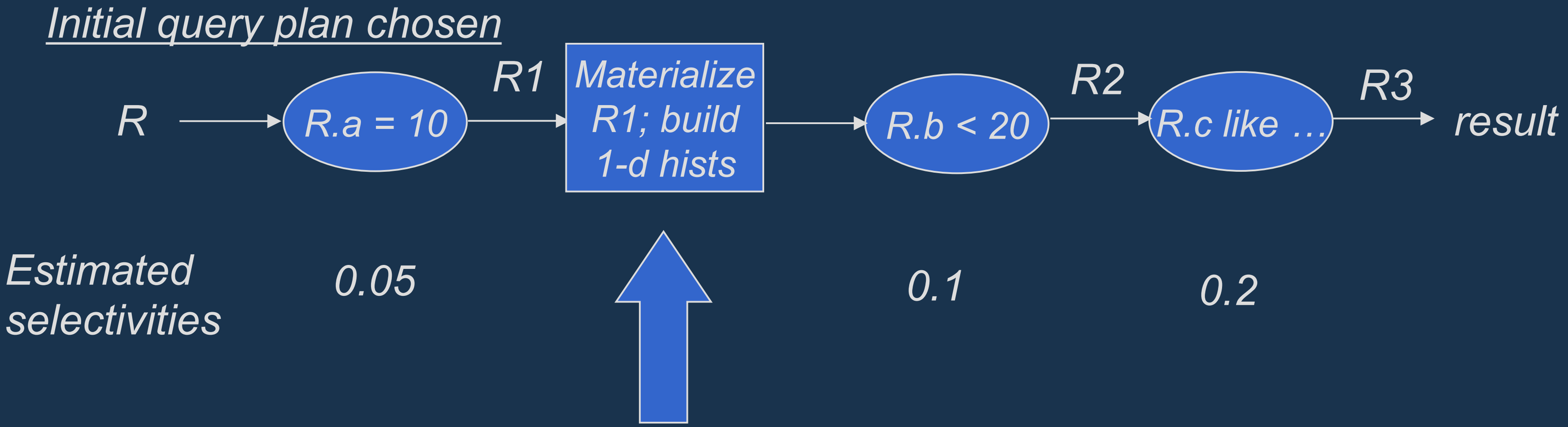


A *free* opportunity to re-evaluate *<u>the rest of the query plan</u>*

- Exploit by gathering information about the materialized result

Mid-query Reoptimization

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

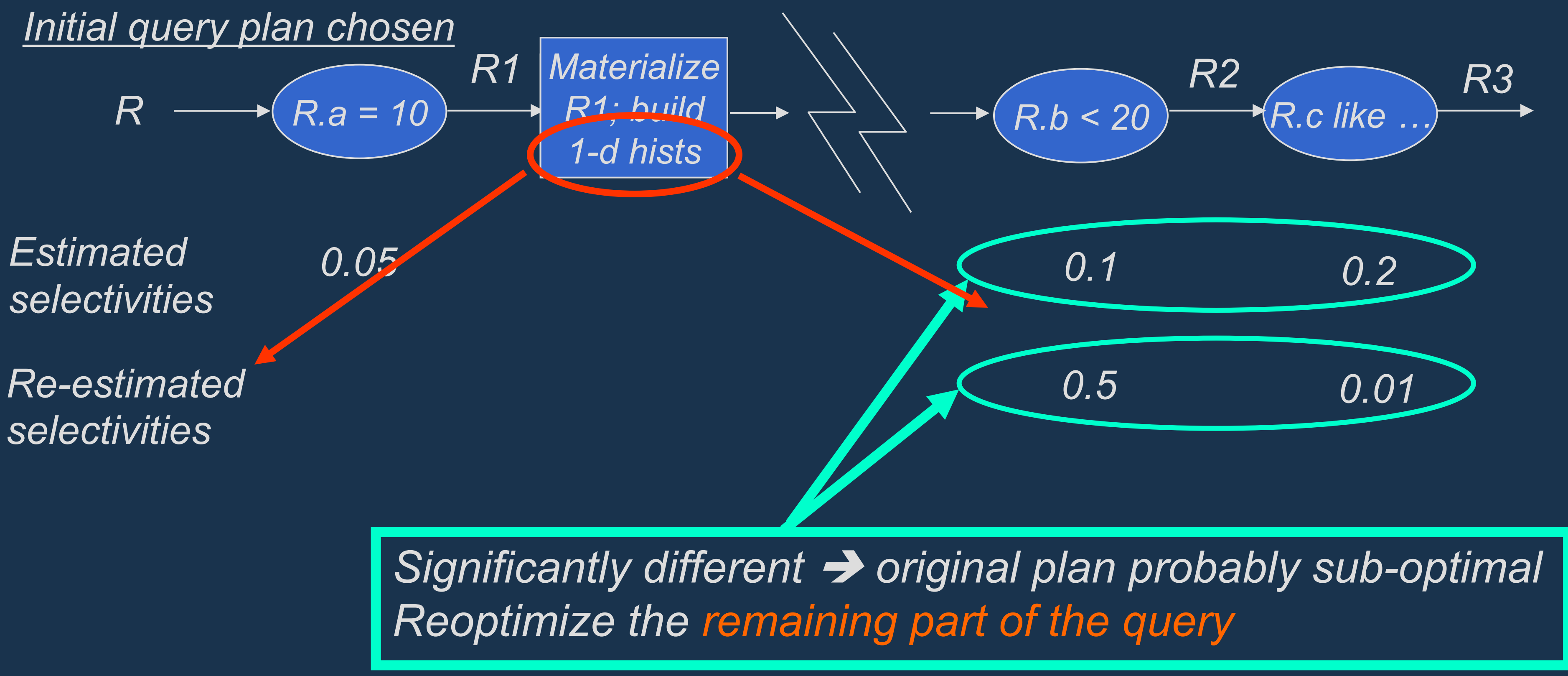


A *free* opportunity to re-evaluate *the rest of the query plan*

- Exploit by gathering information about the materialized result

Mid-query Reoptimization

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



Initial query plan chosen

R

R.a = 10

R1

Materialize R1; build 1-d hists

R.b < 20

R2

R.c like …

R3

Estimated selectivities

0.05

0.1

0.2

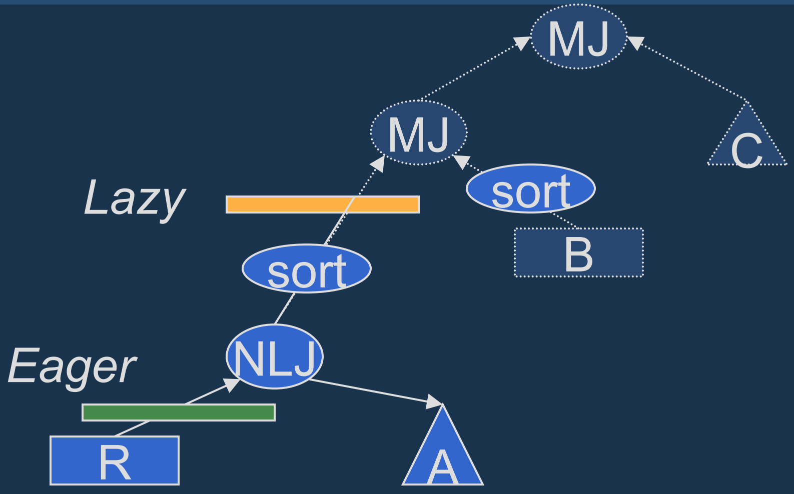
Re-estimated selectivities

0.5

0.01

Significantly different ➔ original plan probably sub-optimal
Reoptimize the remaining part of the query

Where to Place Checkpoints?



More checkpoints ➔ more opportunities for switching plans

Overhead of (simple) monitoring is small [SLMK'01]

Consideration: it is easier to switch plans at some checkpoints than others

Lazy checkpoints: placed above materialization points

- No work need be wasted if we switch plans here

Eager checkpoints: can be placed anywhere

- May have to discard some partially computed results
- Useful where optimizer estimates have high uncertainty

When to Re-optimize?

- Suppose actual cardinality is different from estimates: how high a difference should trigger a re-optimization?

- Idea: do not re-optimize if current plan is still the best

1. Heuristics-based [KD'98]:

   e.g., re-optimize < time to finish execution

2. Validity range [MRS+04]: precomputed range of a parameter (e.g., a cardinality) within which plan is optimal
   - Place eager checkpoints where the validity range is narrow
   - Re-optimize if value falls outside this range
   - Variation: bounding boxes [BBD'05]

How to Reoptimize

Getting a better plan:

- Plug in actual cardinality information acquired during this query (as possibly histograms), and re-run the optimizer

Reusing work when switching to the better plan:

- Treat fully computed intermediate results as materialized views
  - Everything that is under a materialization point
- Note: It is optional for the optimizer to use these in the new plan

- Other approaches are possible (e.g., query scrambling [UFA'98])

Pipelined Execution

Adapting Pipelined Queries

Adapting pipelined execution is often necessary:

- Too few materializations in today's systems
- Long-running queries
- Wide-area data sources
- Potentially endless data streams

The tricky issues:

- Some results may have been delivered to the user
  - Ensuring correctness non-trivial
- Database operators build up *state*
  - Must reason about it during adaptation
  - May need to manipulate state

Adapting Pipelined Queries

We discuss three subclasses of the problem:

- *Selection ordering (stateless)*
  - Very good analytical and theoretical results
  - Increasingly important in web querying, streams, sensornets
  - Certain classes of join queries reduce to them
- *Select-project-join queries (stateful*)
  - *History-independent* execution
    - Operator state largely independent of execution history

      → Execution decisions for a tuple independent of prior tuples
  - *History-dependent* execution
    - Operator state depends on execution history
    - Must reason about the state during adaptation

Pipelined Execution Part I:
Adaptive Selection Ordering

Adaptive Selection Ordering

Complex predicates on single relations common

- e.g., on an employee relation:

  ((*salary > 120000*) AND (*status* = 2)) OR

  ((*salary* between *90000* and *120000*) AND (*age < 30*) AND (*status* = *1*)) OR …

Selection ordering problem:

Decide the order in which to evaluate the individual predicates against the tuples

We focus on *conjunctive predicates* (containing only AND’s)

Example Query

```
select * from R
where R.a = 10 and R.b < 20
and R.c like ‘%name%’;
```

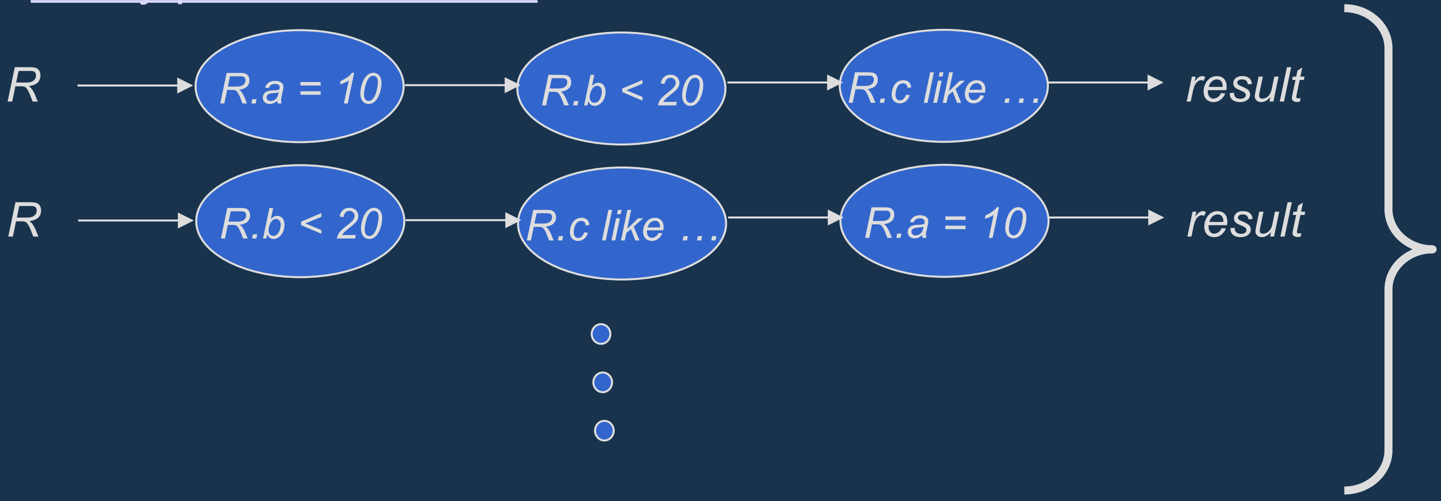
Basics: Static Optimization

Find a *single order of the selections* to be used for *all tuples*

Query

```
select * from R
where R.a = 10 and R.b < 20
and R.c like '%name%';
```

Query plans considered



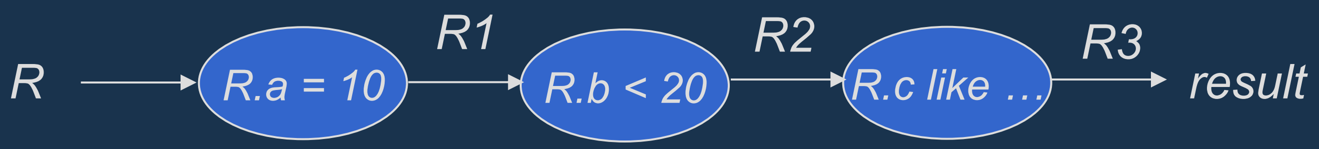
3! = 6 distinct plans possible

Static Optimization

Cost metric: CPU instructions

Computing the cost of a plan

- Need to know the *costs* and the *selectivities* of the predicates



costs	*c1*		*c2*		*c3*
selectivities	*s1*		*s2*		*s3*
cost per tuple	*c1*	+	*s1 c2*	+	*s1 s2 c3*

Independence assumption

$cost(plan) = |R| * (c1 + s1 * c2 + s1 * s2 * c3)$

Static Optimization

Rank ordering algorithm for *independent* selections [IK'84]

- Apply the predicates in the decreasing order of *rank:*

  (1 – s) / c

  where s = selectivity, c = cost

For *correlated* selections:

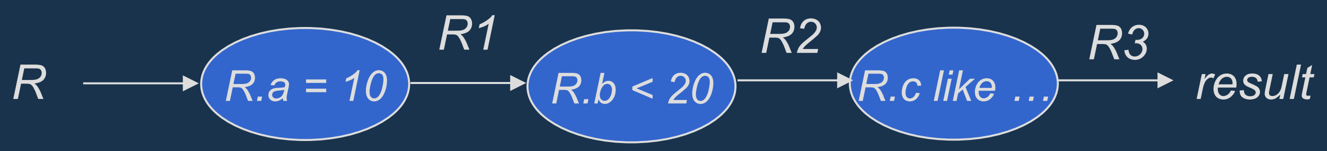
- NP-hard under several different formulations
  - e.g. when given a random sample of the relation

- Greedy algorithm, shown to be 4-approximate [BMMNW'04]:
  - Apply the selection with the highest *(1 - s)/c*
  - Compute the selectivities of remaining selections over the *result*
    - *Conditional selectivities*
  - Repeat

Eddies [AH'00]

Query processing as routing of tuples through operators

A traditional pipelined query plan



R

R.a = 10

R1

R.b < 20

R2

R.c like …

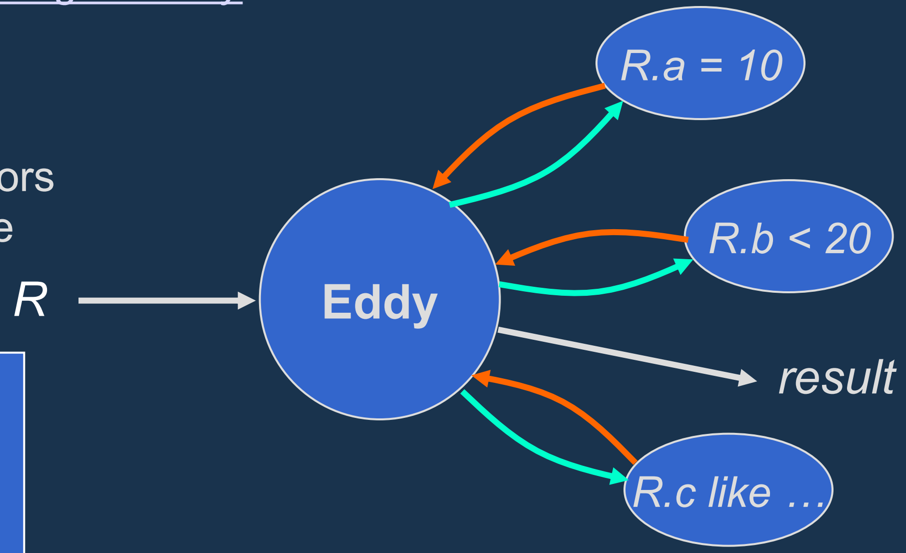
R3

result

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



R

Eddy

R.a = 10

R.b < 20

result

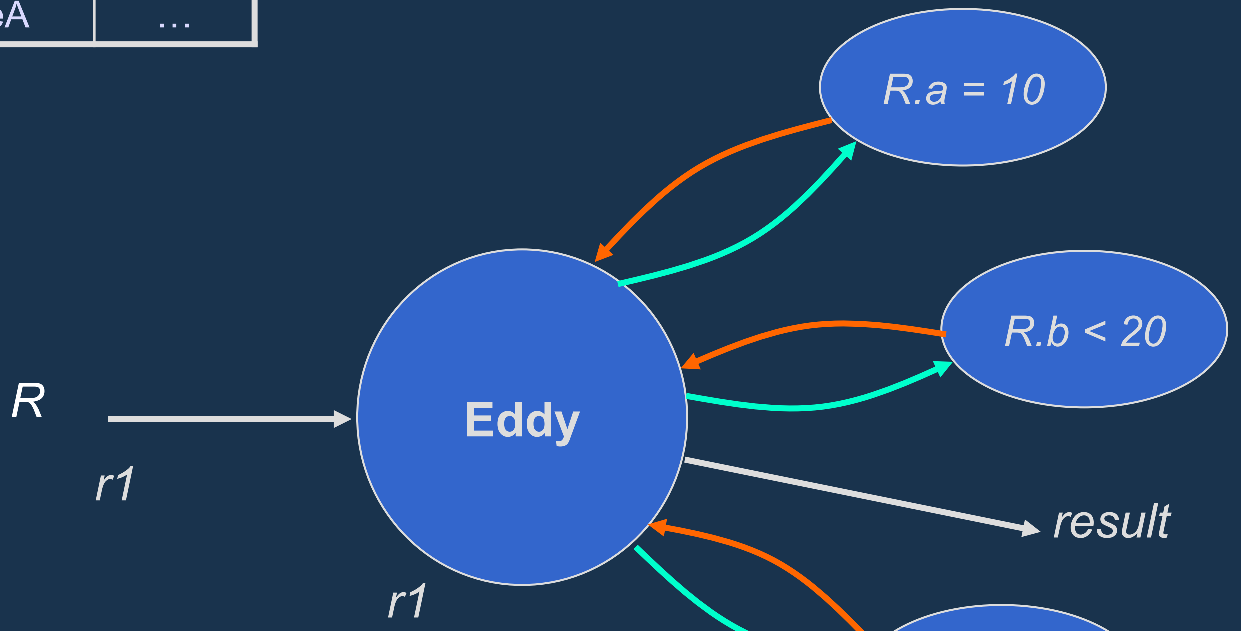
R.c like …

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]

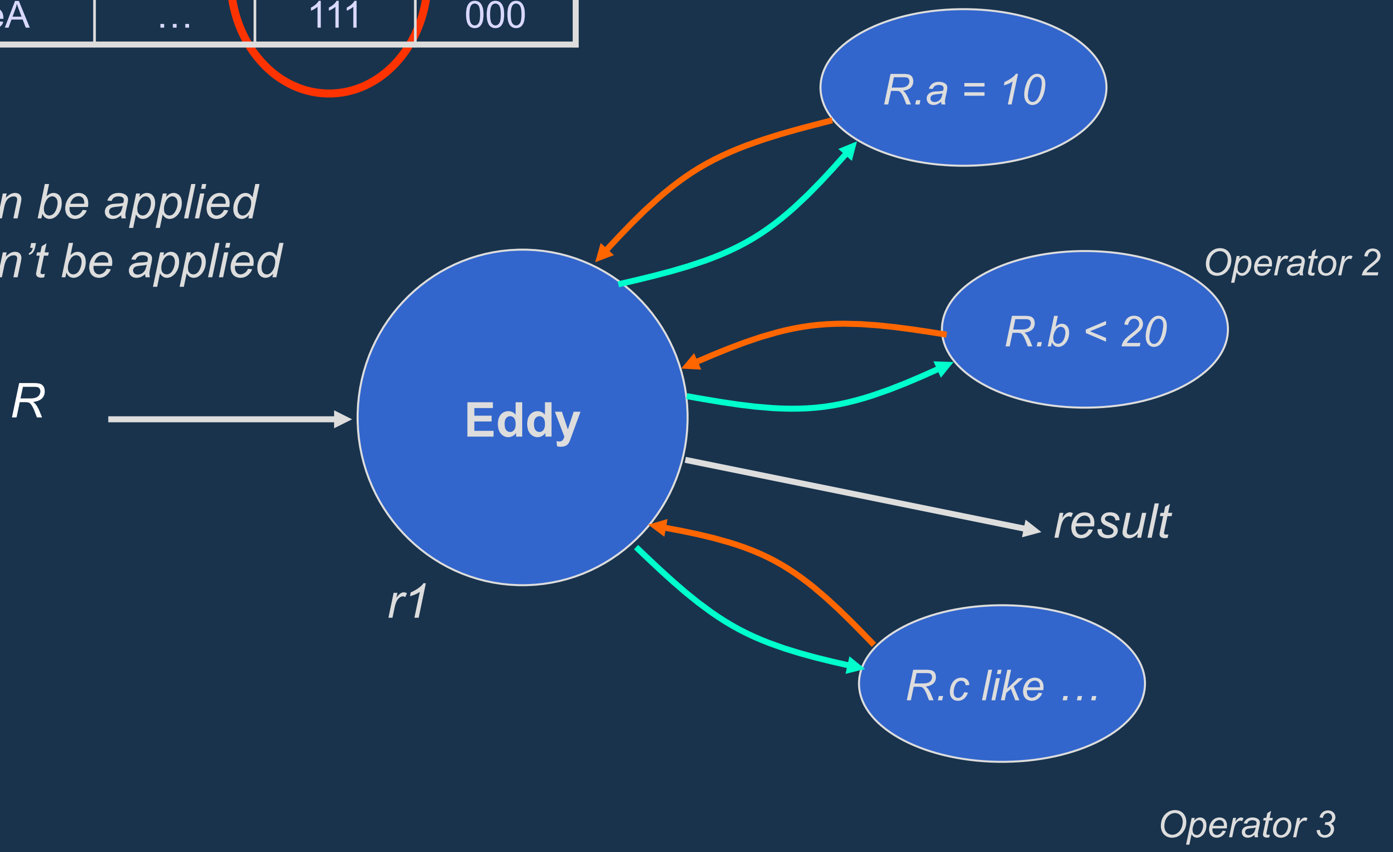
An R Tuple: r1

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*



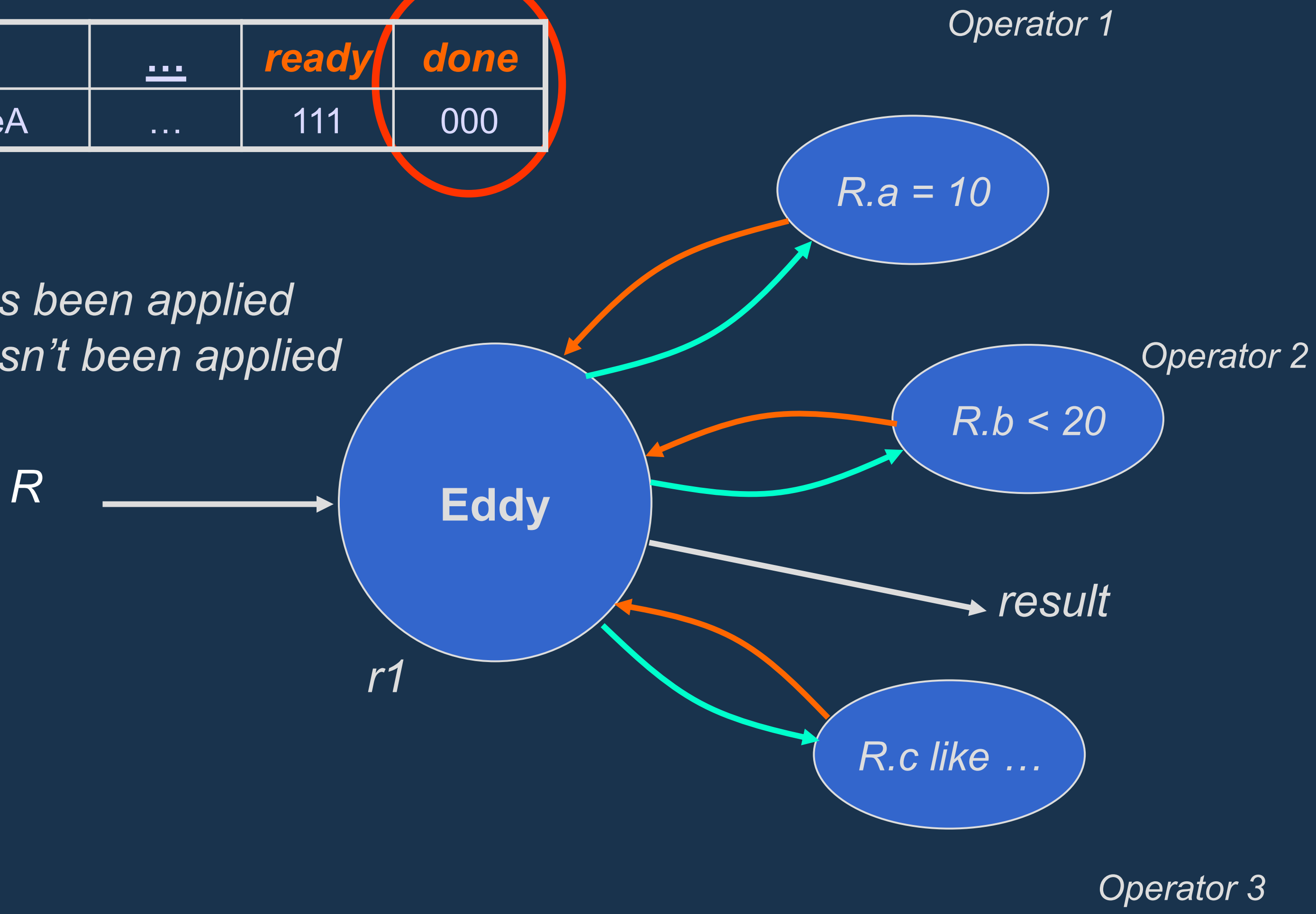
Eddies [AH'00]

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*

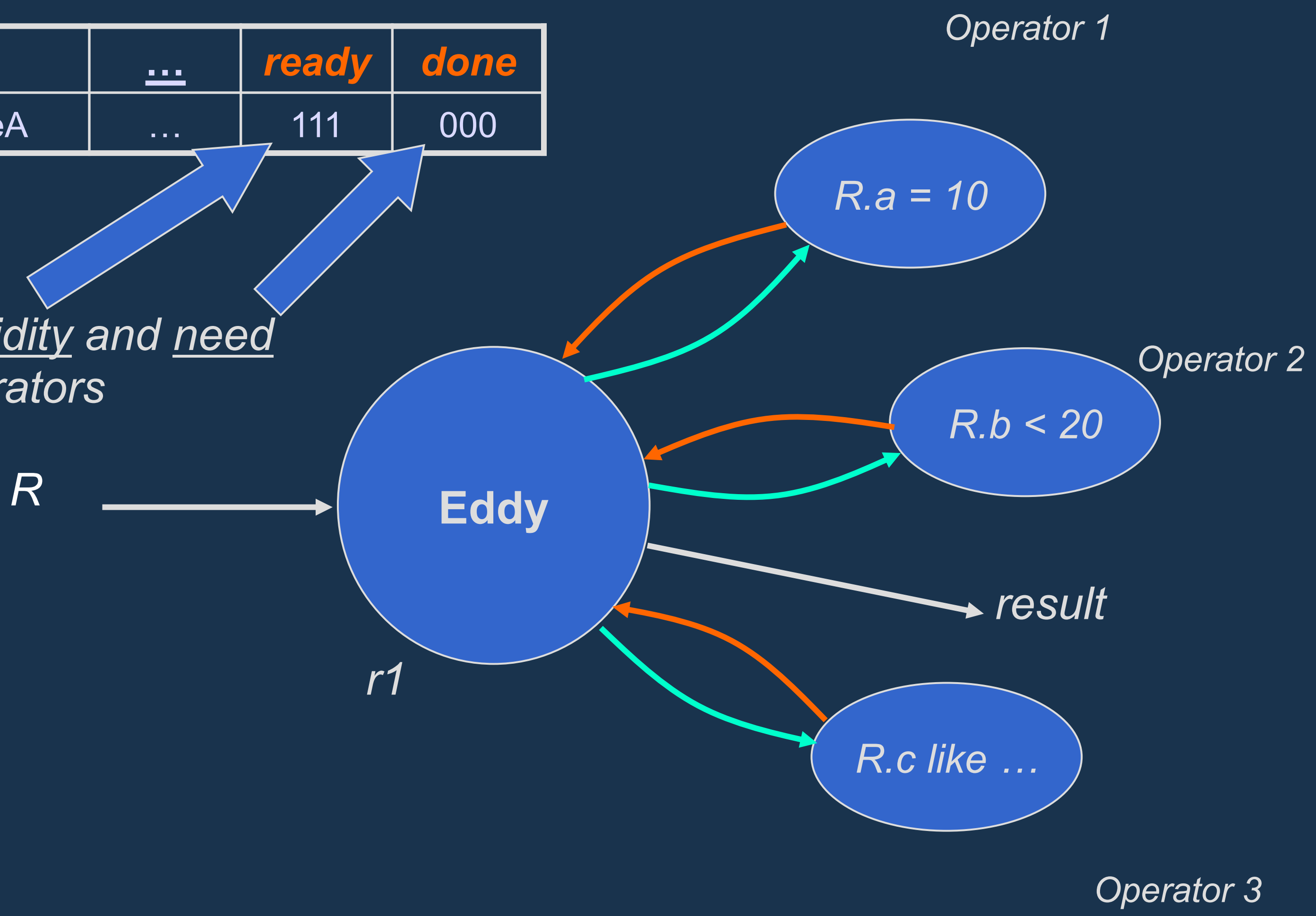


Eddies [AH’00]

An R Tuple: r1

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

Used to decide validity and need of applying operators

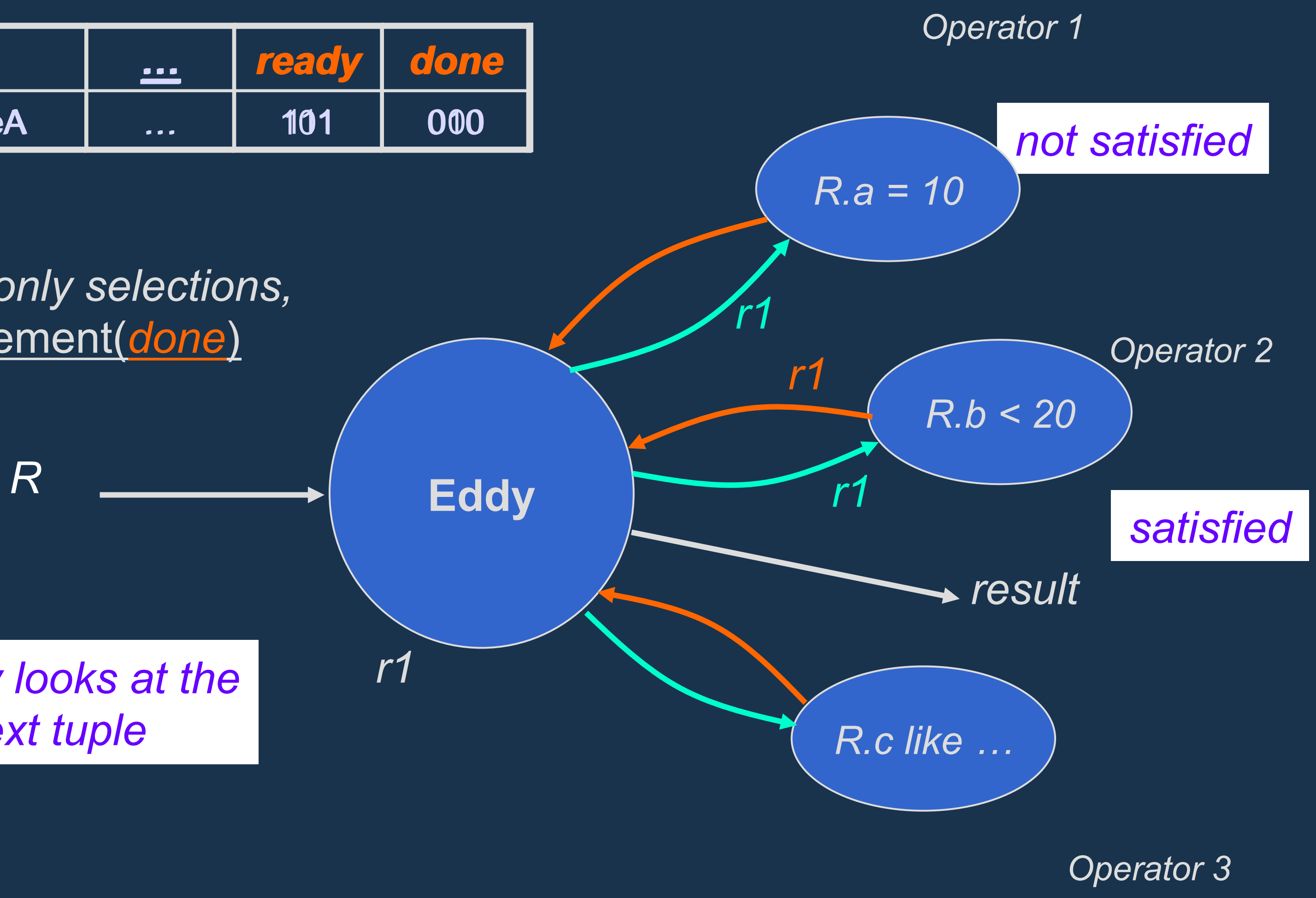


Eddies [AH'00]

An R Tuple: r1

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

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



not satisfied

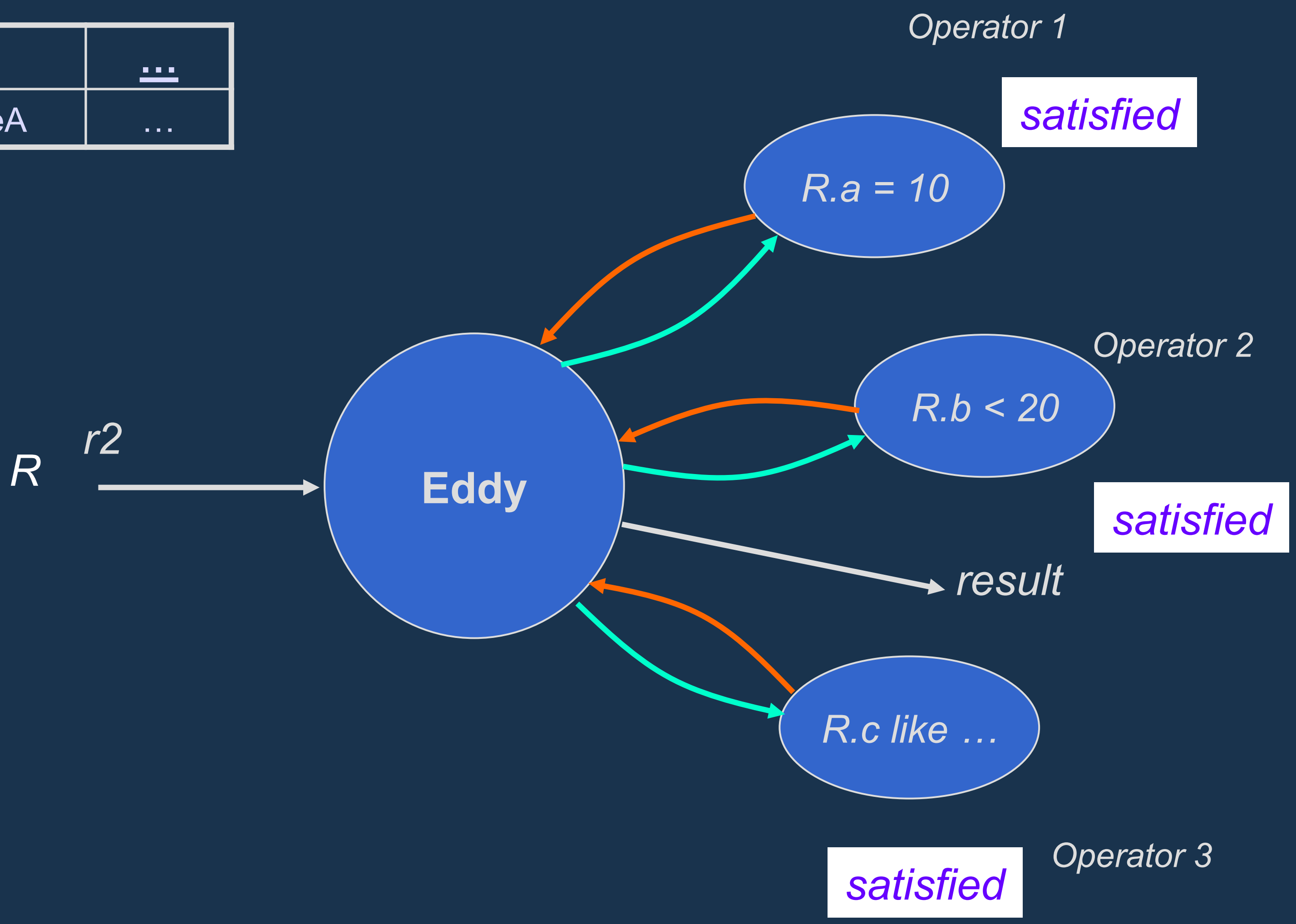
satisfied

eddy looks at the next tuple

Eddies [AH'00]

An R Tuple: r2

a	b	c	...
10	15	AnameA	…



Operator 1

satisfied

R.a = 10

Operator 2

R.b < 20

satisfied

R

r2

Eddy

result

R.c like …

satisfied

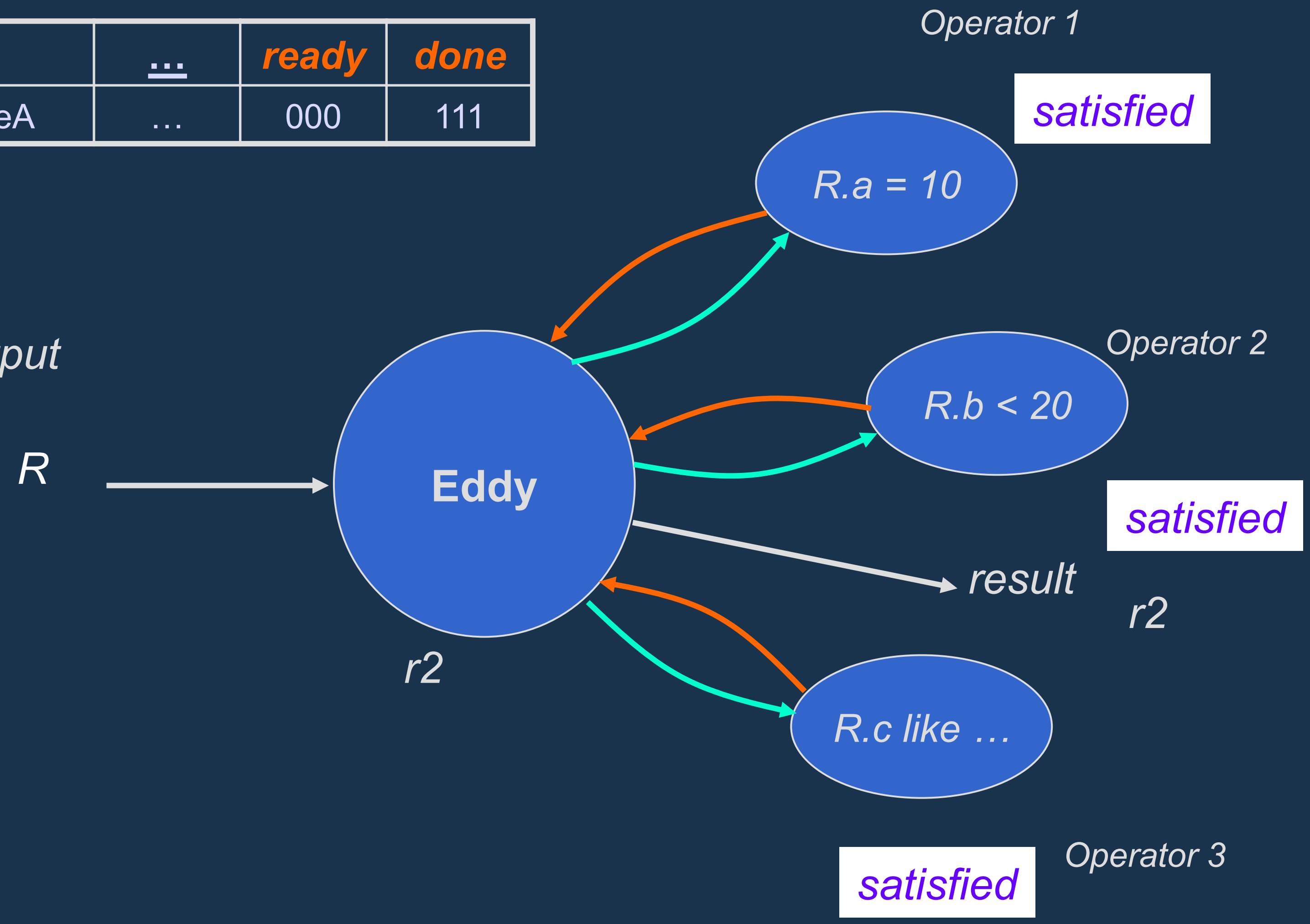
Operator 3

Eddies [AH'00]

An R Tuple: r2

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

if done = 111,
send to output



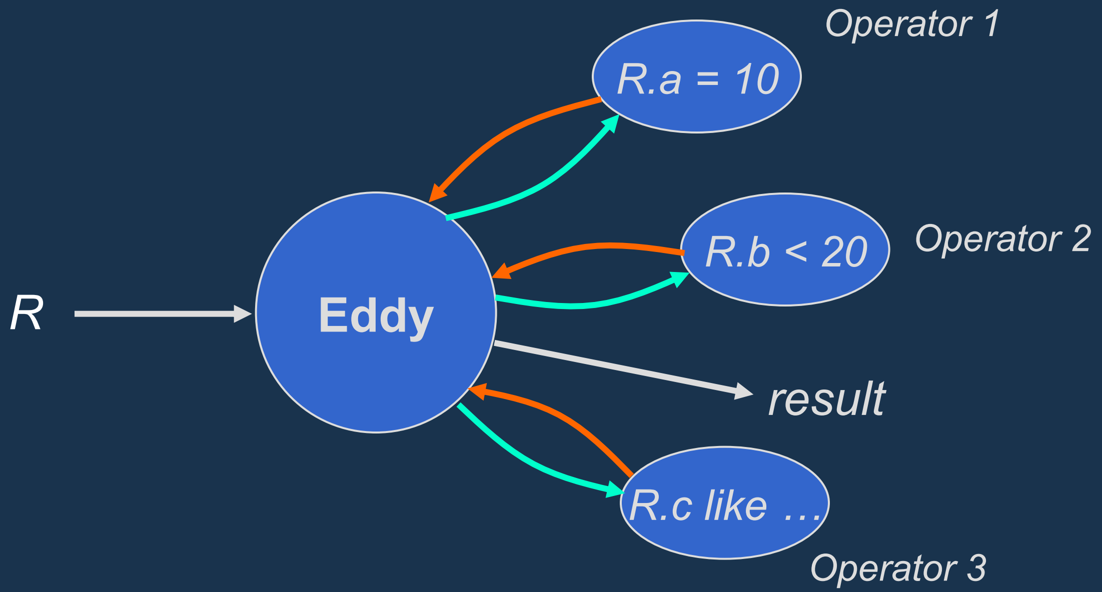
Eddies [AH'00]

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

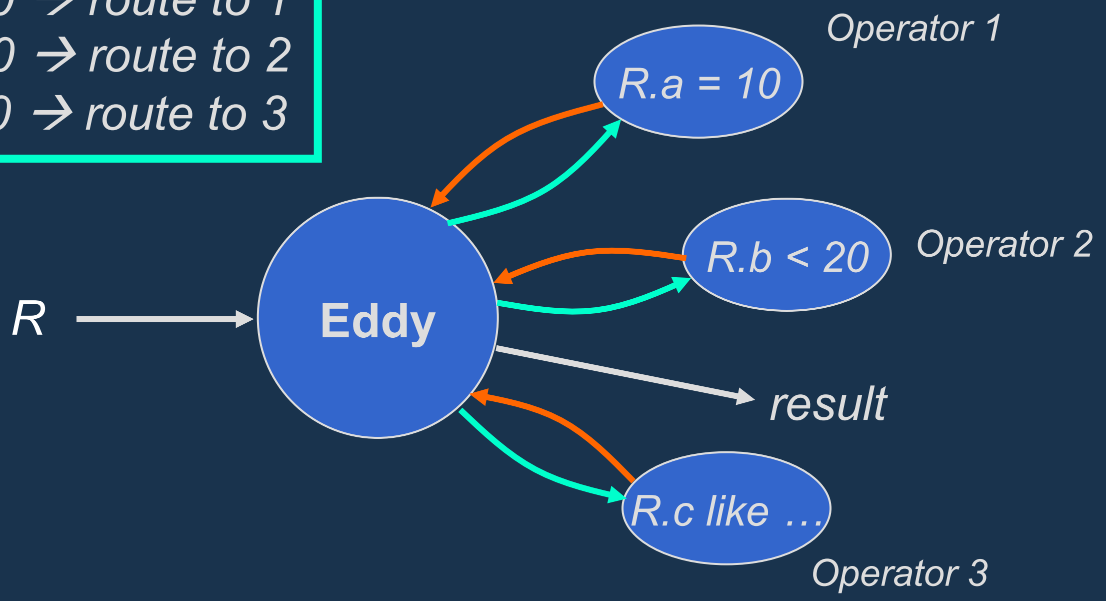
Routing Policy 1: Non-adaptive

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

table lookups → very efficient

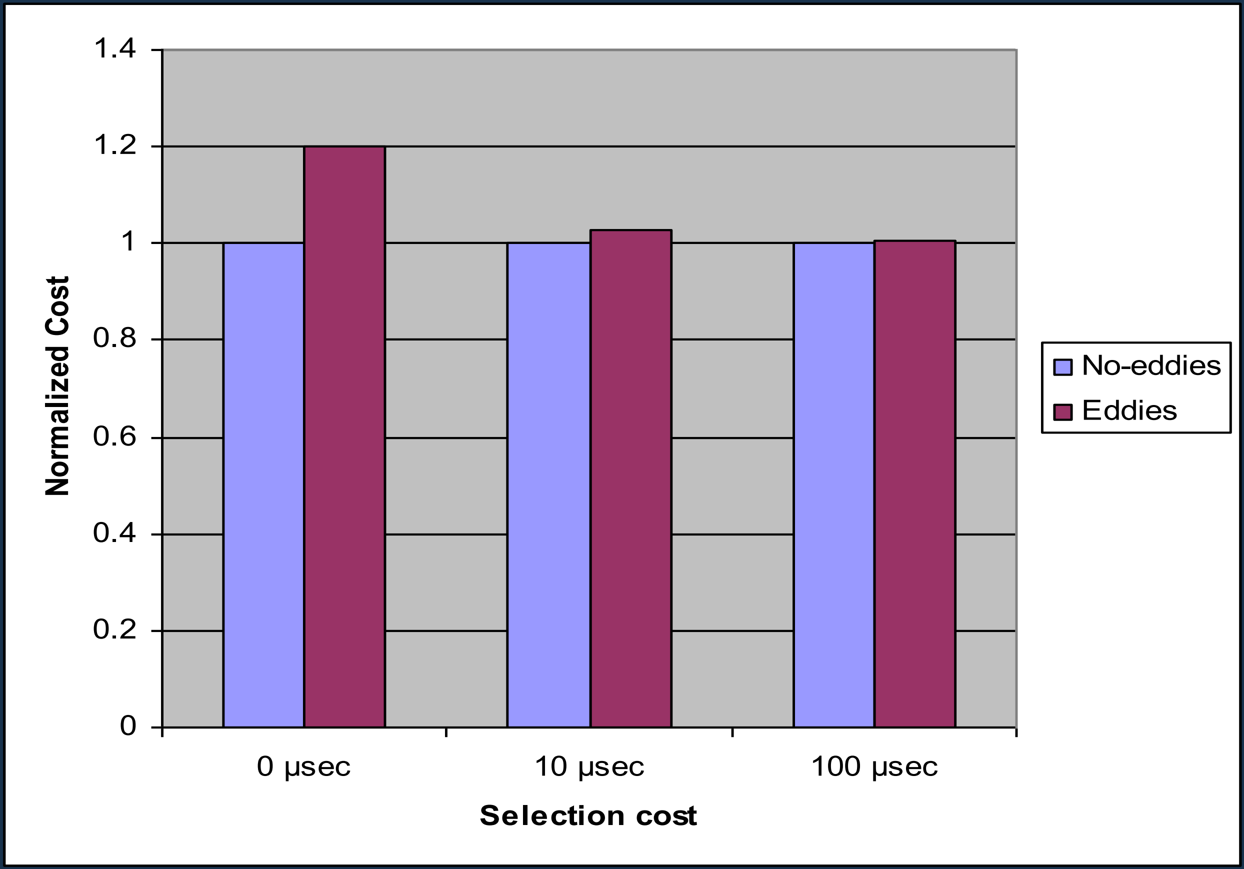
Routing policy:

```
    if done =
        000 → route to 1
        100 → route to 2
        110 → route to 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:

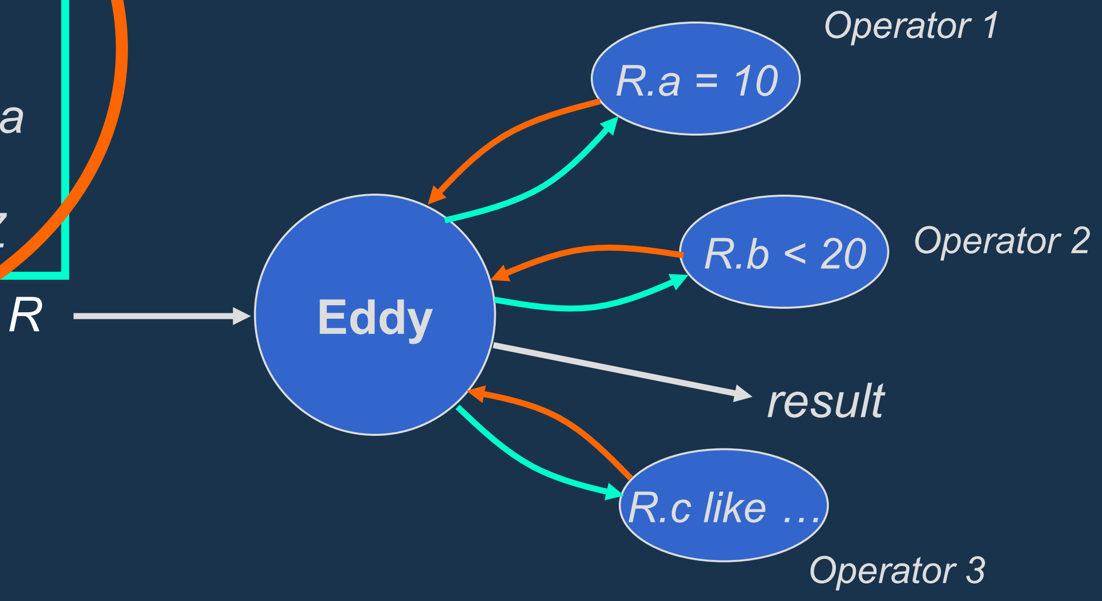
Costs of operators

Selectivities of operators

Routing policy:

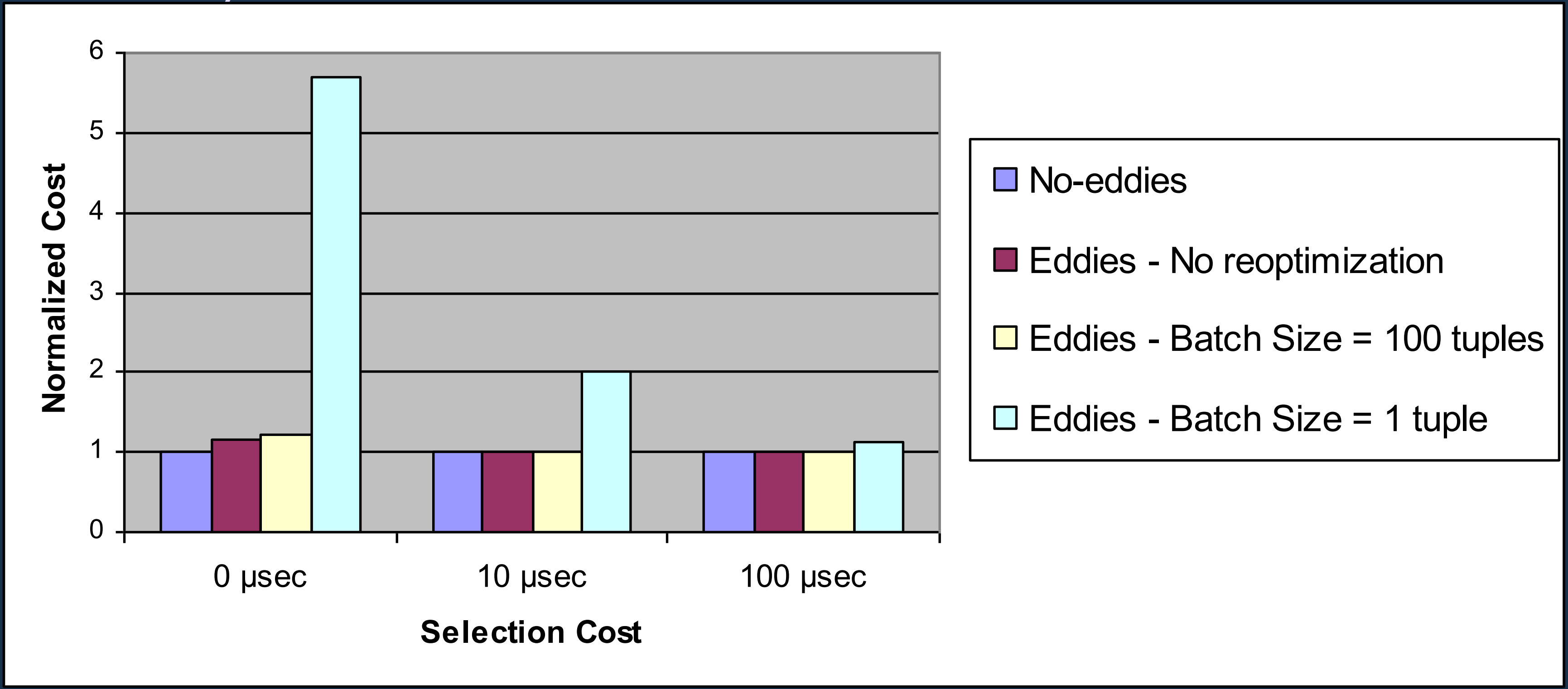
Use a single order for a batch of tuples

Periodically apply KBZ



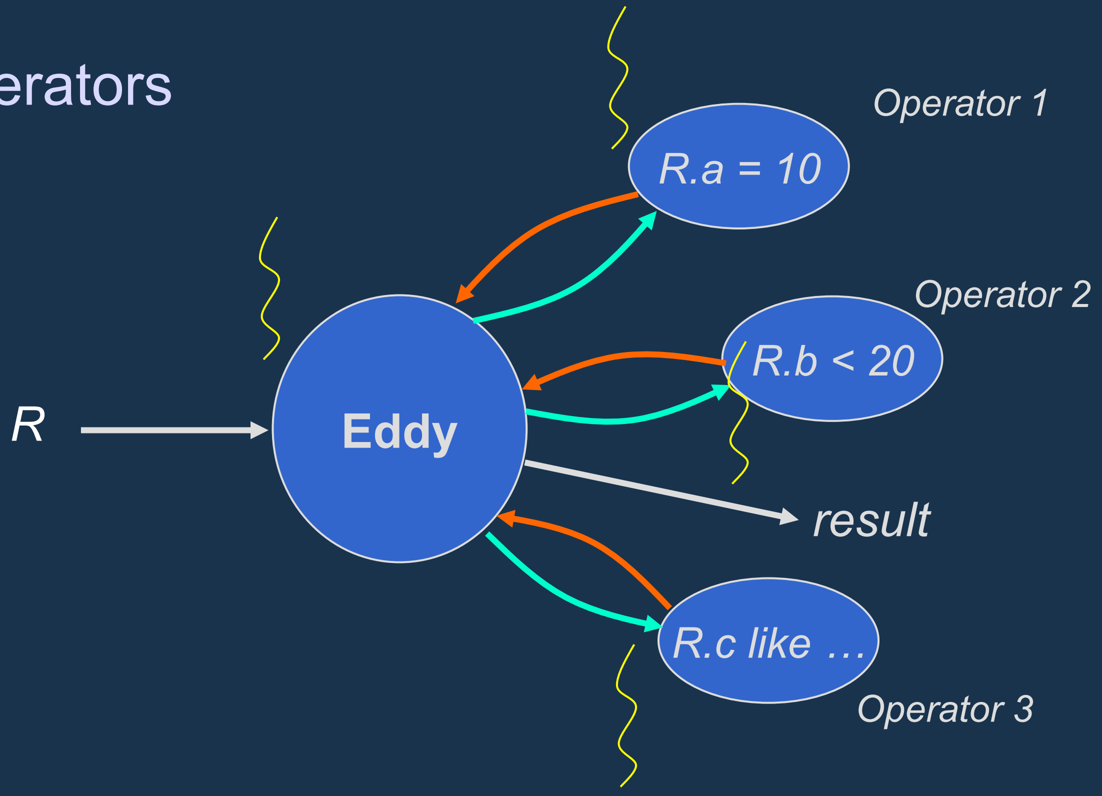
Overhead of Routing and Reoptimization

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



Routing Policy 3: Lottery Scheduling

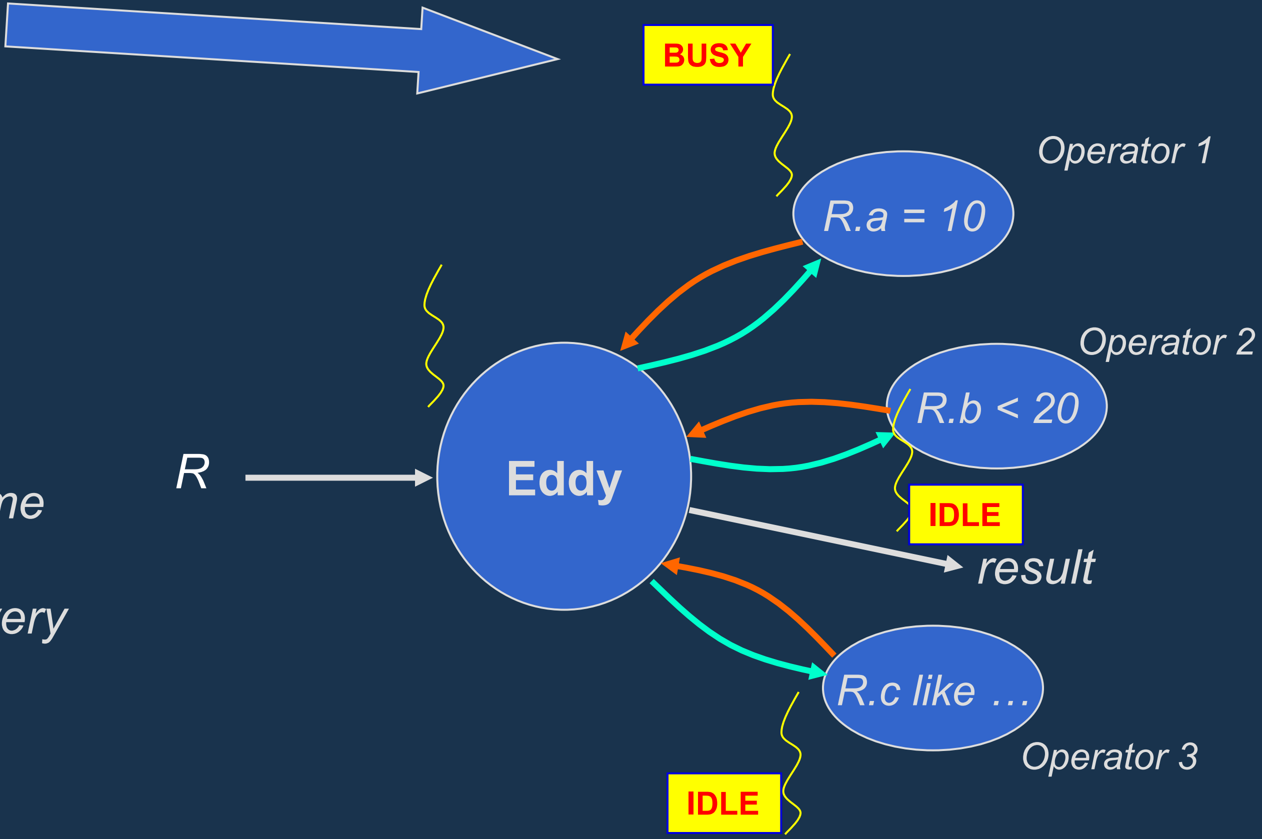
- Originally suggested routing policy [AH'00]
- Applicable only if each operator runs in a separate thread
- Uses two easily obtainable pieces of information for making routing decisions:
  - *Busy/idle status* of operators
  - *Tickets* per operator



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



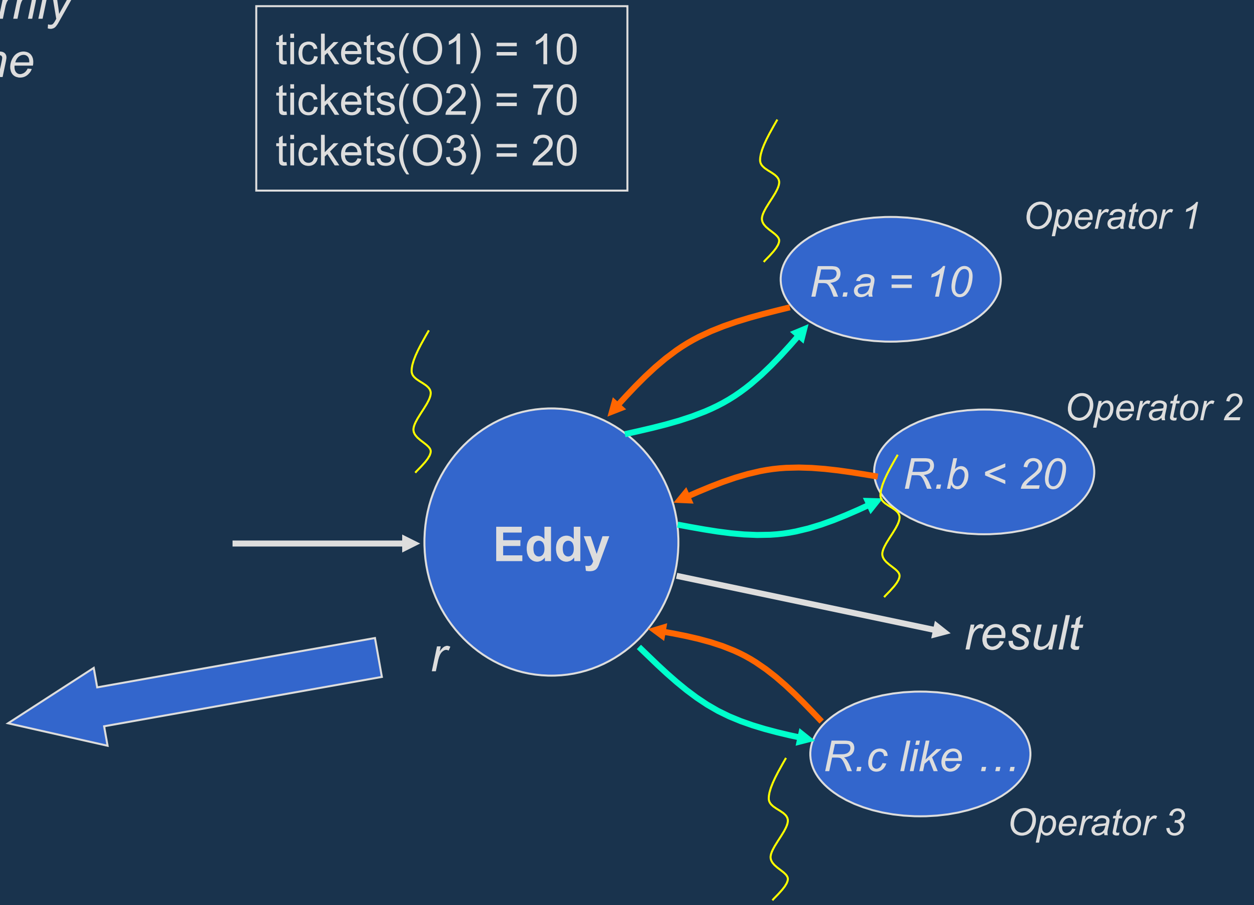
Rationale:
Every thread gets equal time
SO IF an operator is busy,
THEN its cost is perhaps very high

Routing Policy 3: Lottery Scheduling

- Routing decisions based on tickets

Rules:

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



tickets(O1) = 10
tickets(O2) = 70
tickets(O3) = 20

Will be routed to:

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

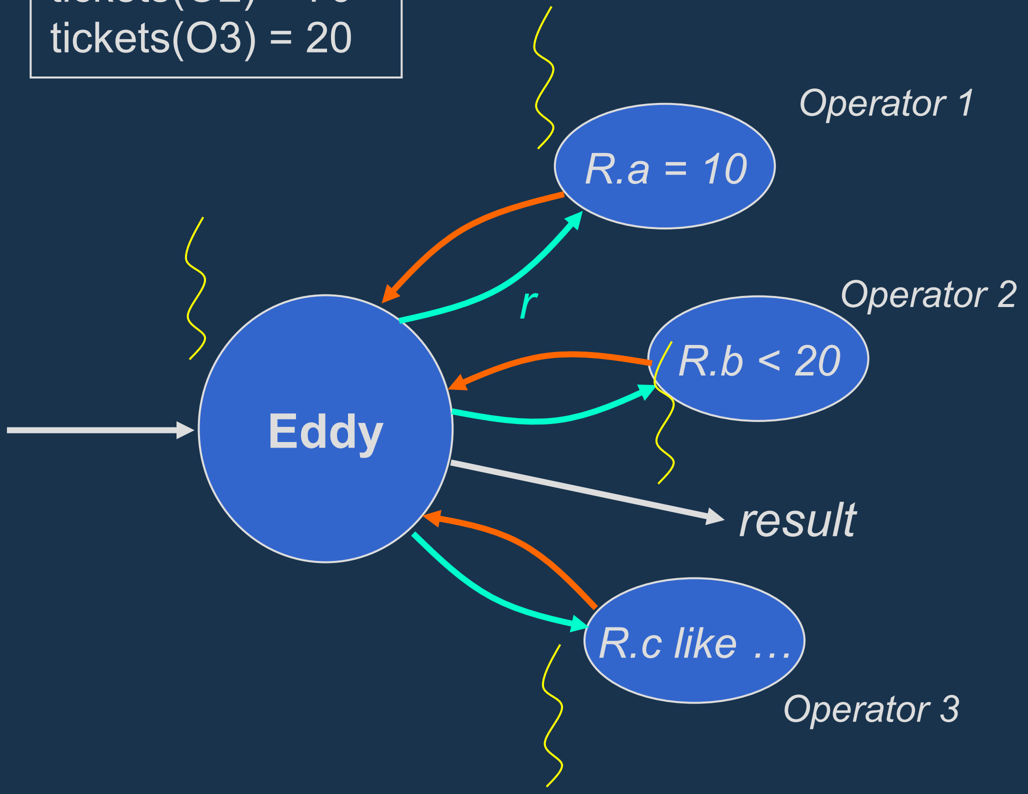
Routing Policy 3: Lottery Scheduling

- Routing decisions based on tickets

Rules:

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

tickets(O1) = 10
tickets(O2) = 70
tickets(O3) = 20



Routing Policy 3: Lottery Scheduling

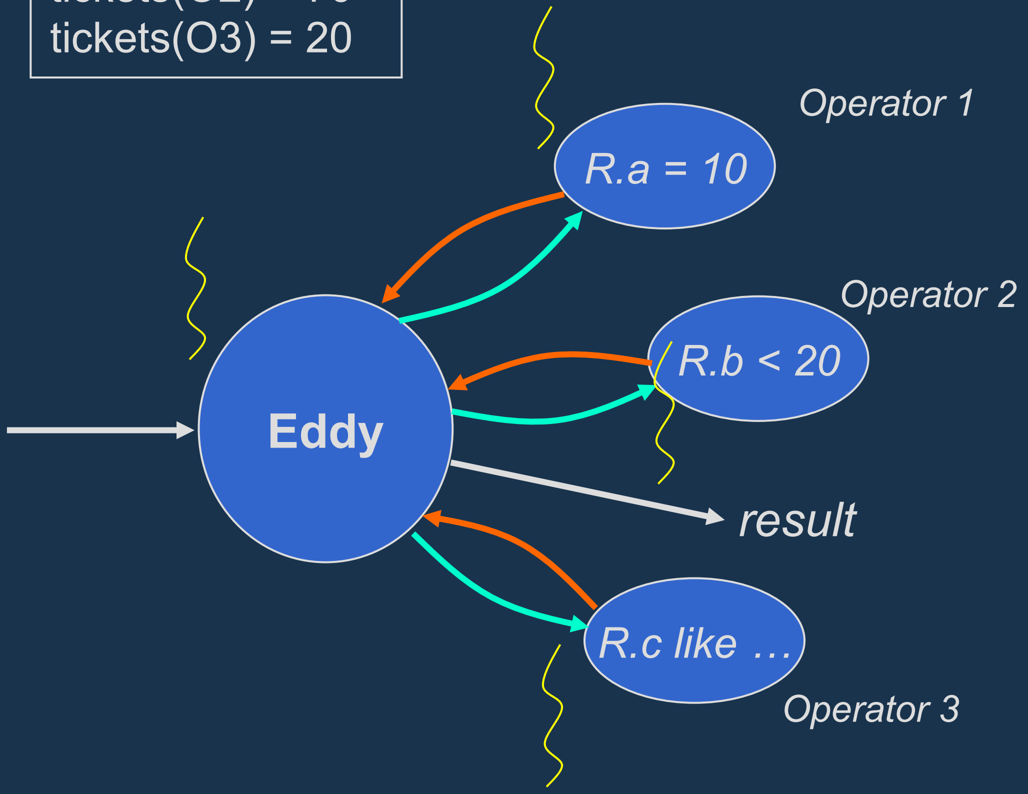
- Routing decisions based on tickets

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) ++;

tickets(O1) = 11
tickets(O2) = 70
tickets(O3) = 20

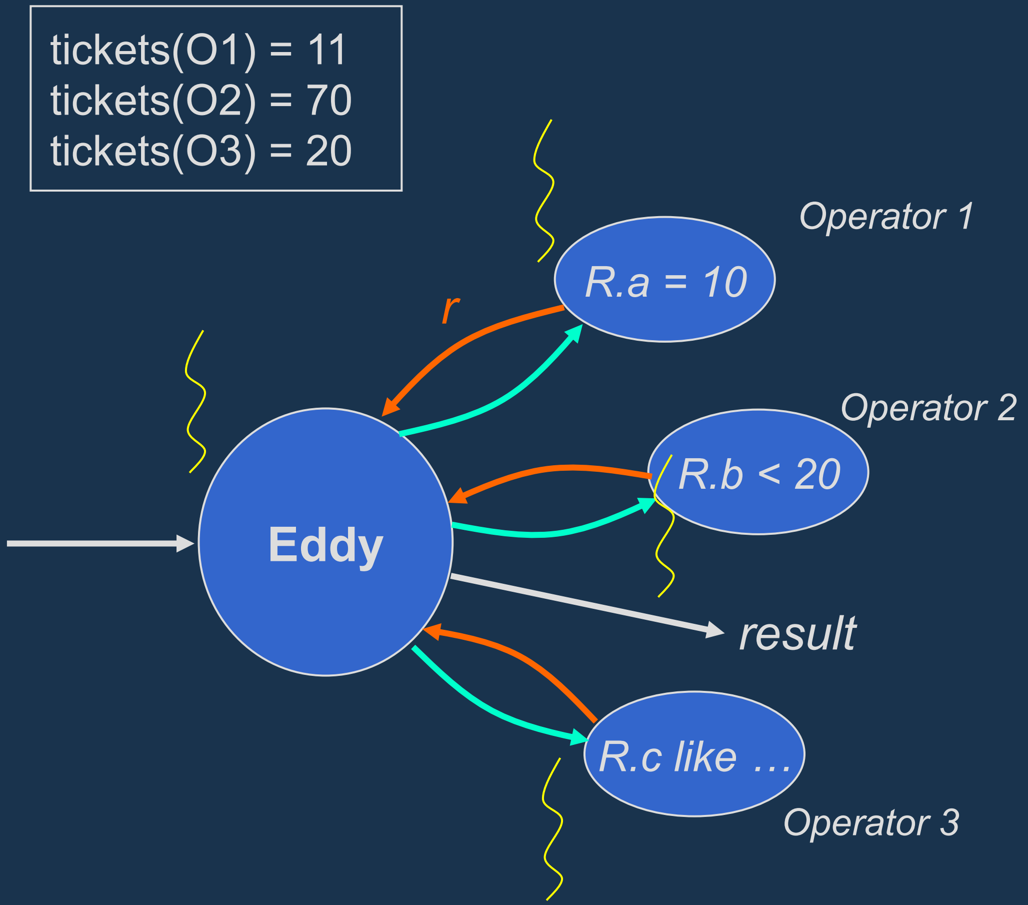


Routing Policy 3: Lottery Scheduling

- 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) --;



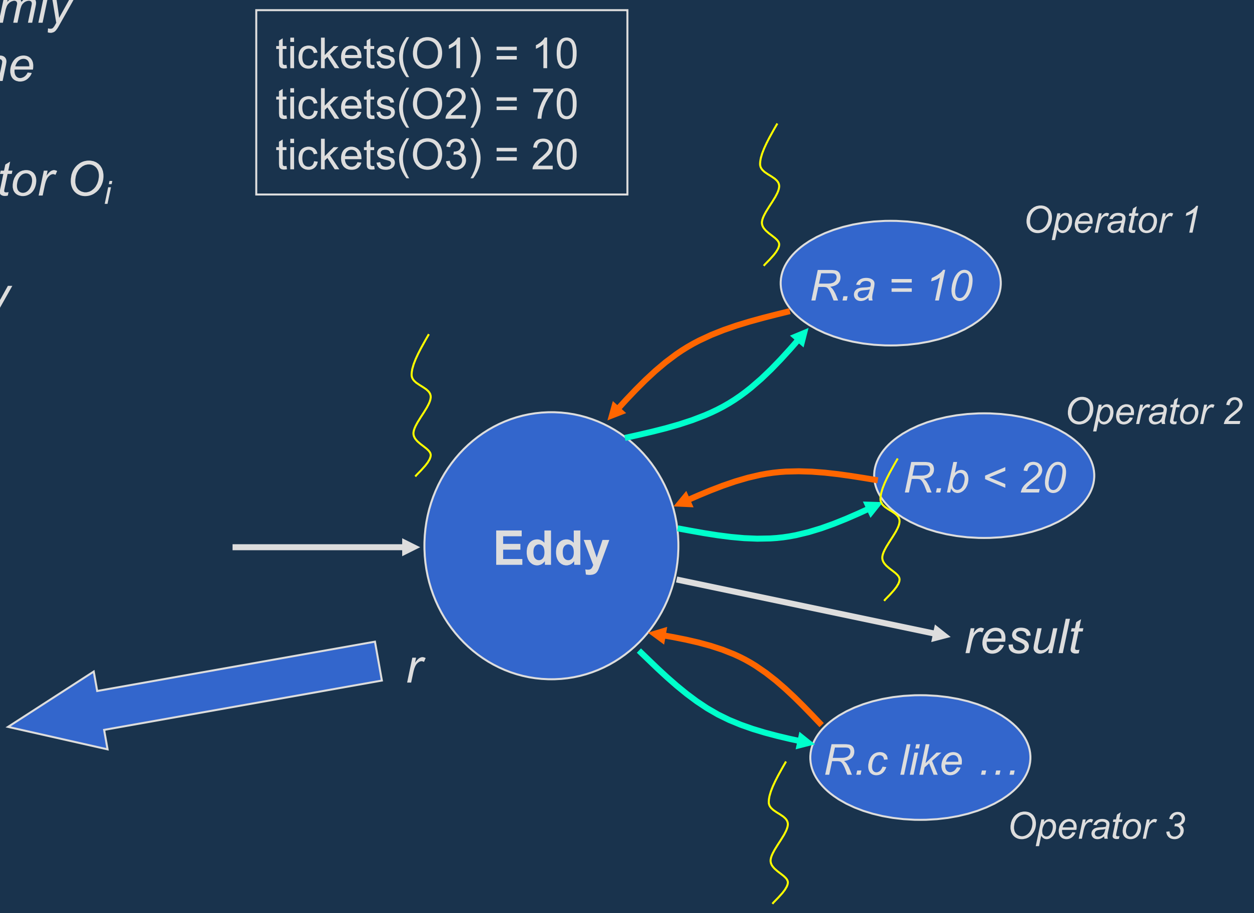
tickets(O1) = 11
tickets(O2) = 70
tickets(O3) = 20

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*
   $tickets(O_i)$ --;



tickets(O1) = 10
tickets(O2) = 70
tickets(O3) = 20

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:

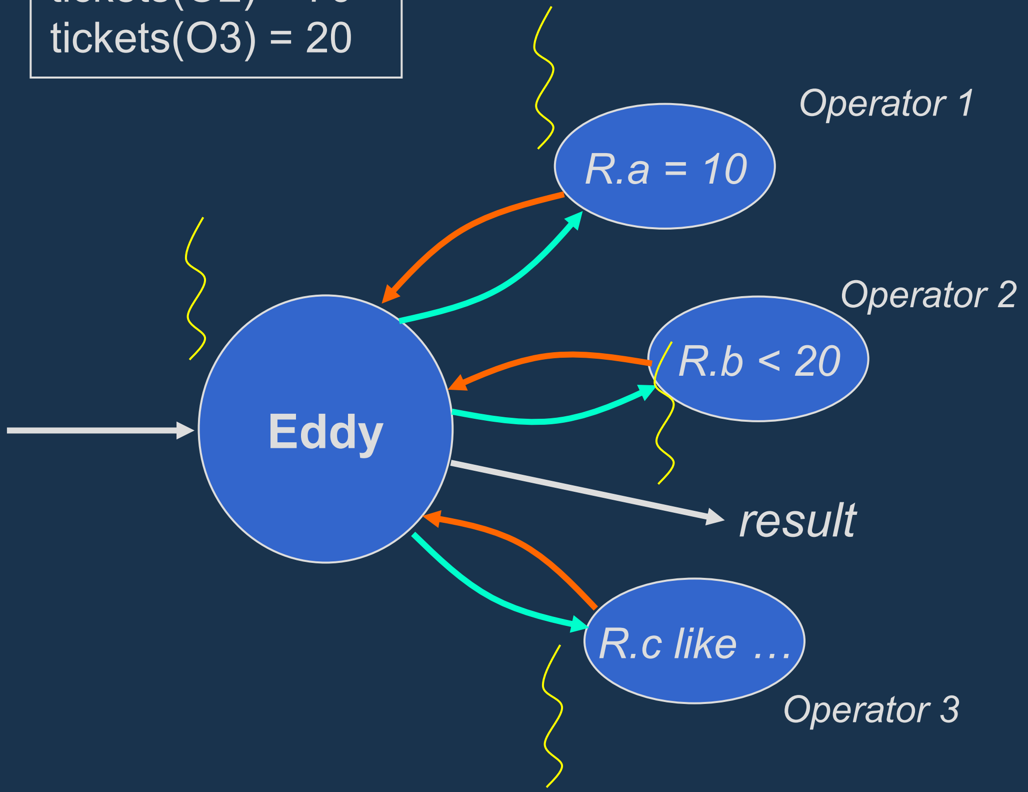
1. *Route a new tuple randomly weighted according to the number of tickets*
2. *route a tuple to an operator O_i*
   $tickets(O_i)$ ++;
3. *O_i returns a tuple to eddy*
   $tickets(O_i)$ --;

Rationale:

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

So more tuples are routed to highly selective operators

tickets(O1) = 10
tickets(O2) = 70
tickets(O3) = 20

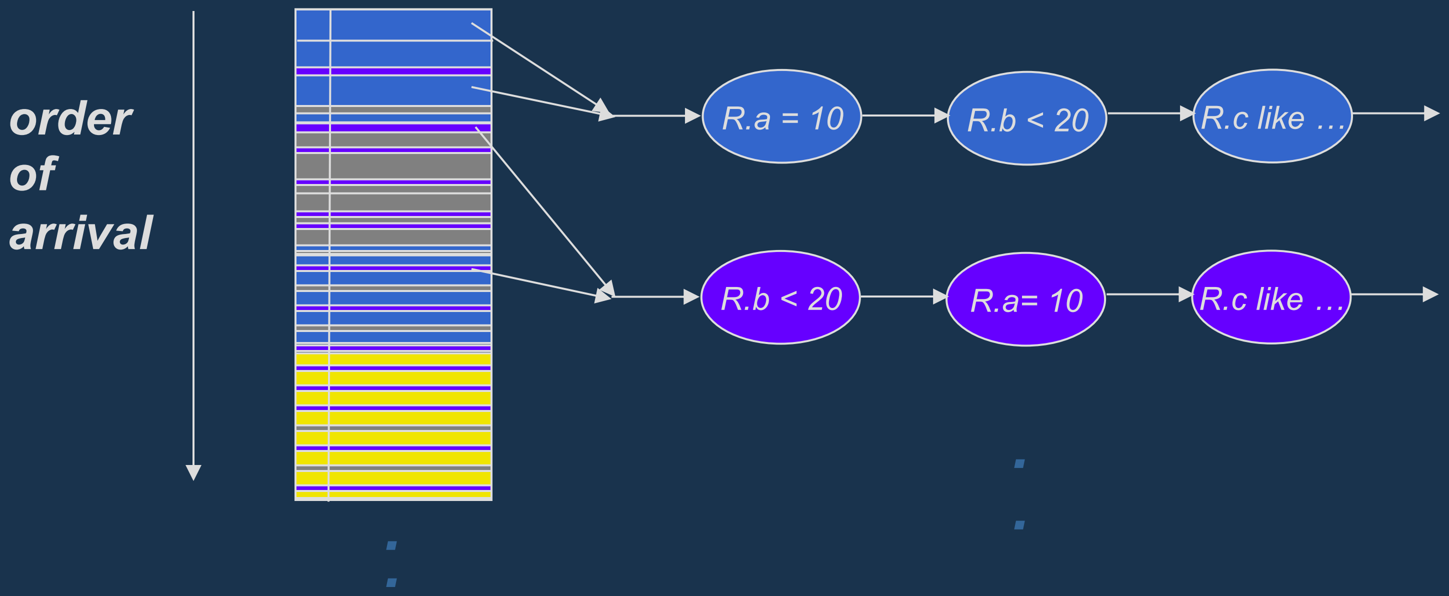


Routing Policy 3: Lottery Scheduling

- 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 <u>arbitrary</u> "*horizontal partitioning*"
  - Not necessarily correlated with order of arrival



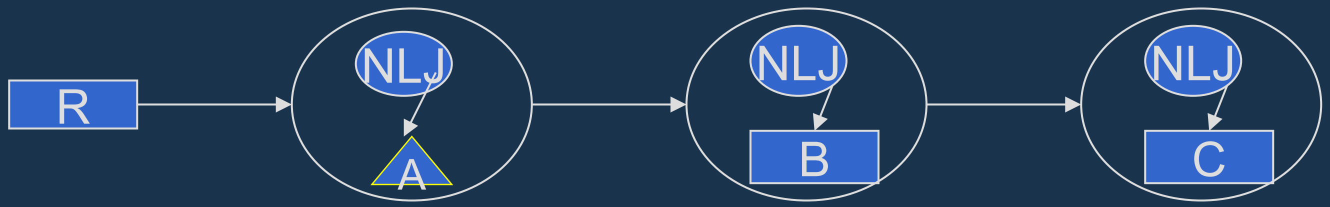
In a later paper, we looked at optimizing for horizontal partitioning directly

Pipelined Execution Part II: Adaptive Join Processing

Adaptive Join Processing: Outline

- Single streaming relation
  - Left-deep pipelined plans
- Multiple streaming relations
  - Execution strategies for multi-way joins
  - History-independent execution
  - History-dependent execution

Left-Deep Pipelined Plans

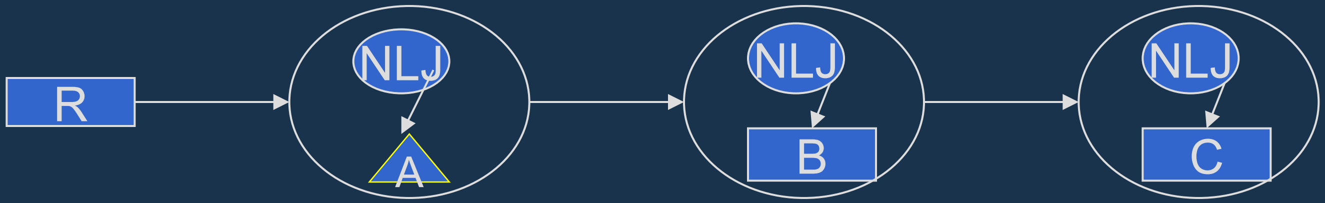


Simplest method of joining tables

- Pick a *driver* table (R). Call the rest *driven* tables
- Pick access methods (AMs) on the driven tables (*scan, hash, or index*)
- Order the driven tables
- Flow R tuples through the driven tables

```
For each r ∈ R do:
look for matches for r in A;
for each match a do:
        look for matches for <r,a> in B;
        …
```

Adapting a Left-deep Pipelined Plan



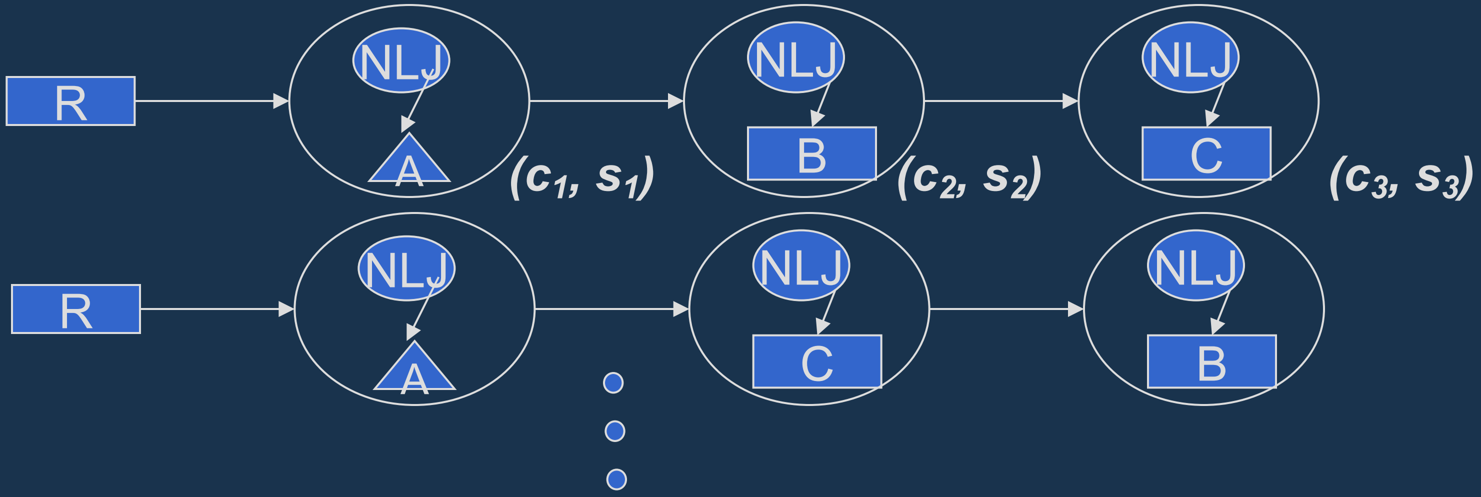
Simplest method of joining tables

- Pick a *driver* table (R). Call the rest *driven* tables
- Pick access methods (AMs) on the driven tables
- Order the driven tables
- Flow R tuples through the driven tables

Almost identical to selection ordering

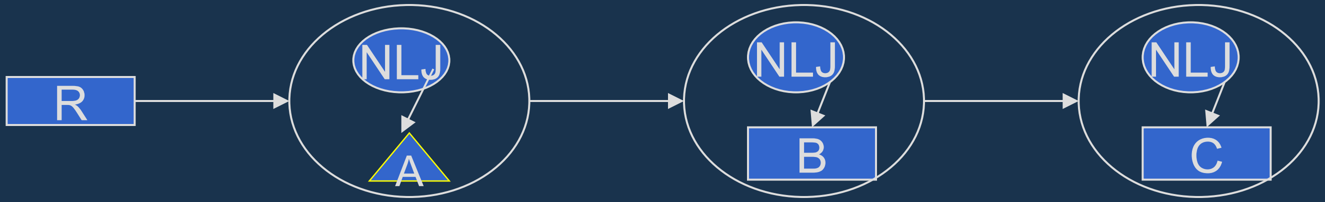
```
For each $r \in$ R do:
look for matches for r in A;
for each match a do:
        look for matches for <r,a> in B;
        …
```

Adapting the Join Order



- Let c_i = cost/lookup into i'th driven table,
  s_i = fanout of the lookup
- As with selection, cost = $|R| \times (c_1 + s_1c_2 + s_1s_2c_3)$
- Caveats:
  - Fanouts $s_1, s_2, \ldots$ can be > 1
  - Precedence constraints
  - Caching issues
- Can use *rank ordering, A-greedy* for adaptation (subject to the caveats)

Adapting a Left-deep Pipelined Plan



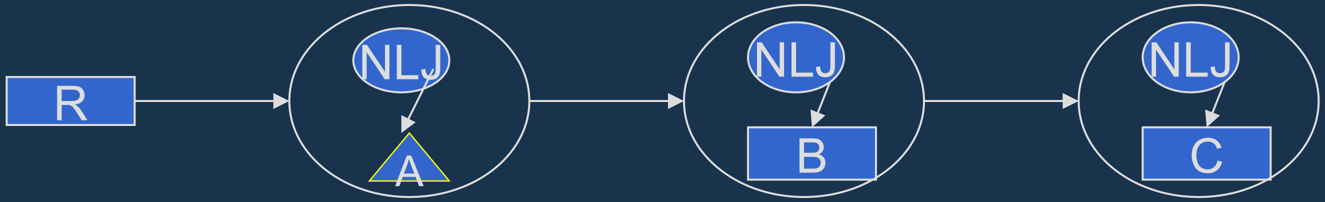
Simplest method of joining tables

- Pick a *driver* table (R). Call the rest *driven* tables
- Pick access methods (AMs) on the driven tables
- Order the driven tables
- Flow R tuples through the driven tables

?

```
For each r ∈ R do:
look for matches for r in A;
for each match a do:
        look for matches for <r,a> in B;
        …
```

Adapting a Left-deep Pipelined Plan



Key issue: Duplicates

Adapting the choice of driver table

[L+07] Carefully use indexes to achieve this

Adapting the choice of access methods

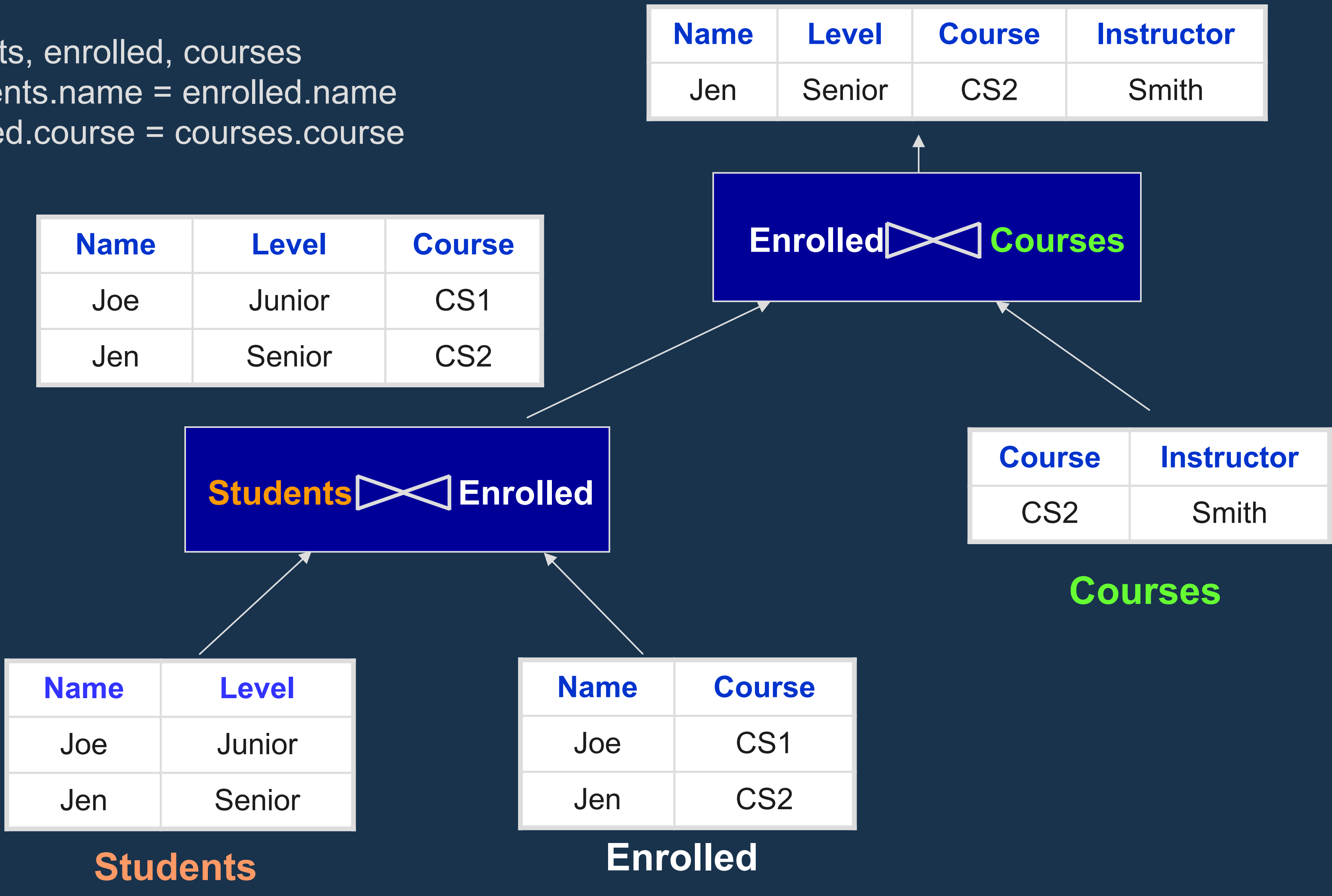
- Static optimization: explore all possibilities and pick best
- Adaptive: Run multiple plans in parallel for a while, and then pick one and discard the rest [Antoshenkov' 96]
  - Cannot easily explore combinatorial options

Adaptive Join Processing: Outline

- Single streaming relation
  - Left-deep pipelined plans
- Multiple streaming relations
  - Execution strategies for multi-way joins
  - History-independent execution
    - MJoins
  - History-dependent execution
    - Eddies with joins
    - Corrective query processing

Example Join Query & Database

```
select *
from students, enrolled, courses
where students.name = enrolled.name
  and enrolled.course = courses.course
```



Name	Level	Course	Instructor
Jen	Senior	CS2	Smith

Enrolled ⋈ Courses

Name	Level	Course
Joe	Junior	CS1
Jen	Senior	CS2

Students ⋈ Enrolled

Course	Instructor
CS2	Smith

Courses

Name	Level
Joe	Junior
Jen	Senior

Students

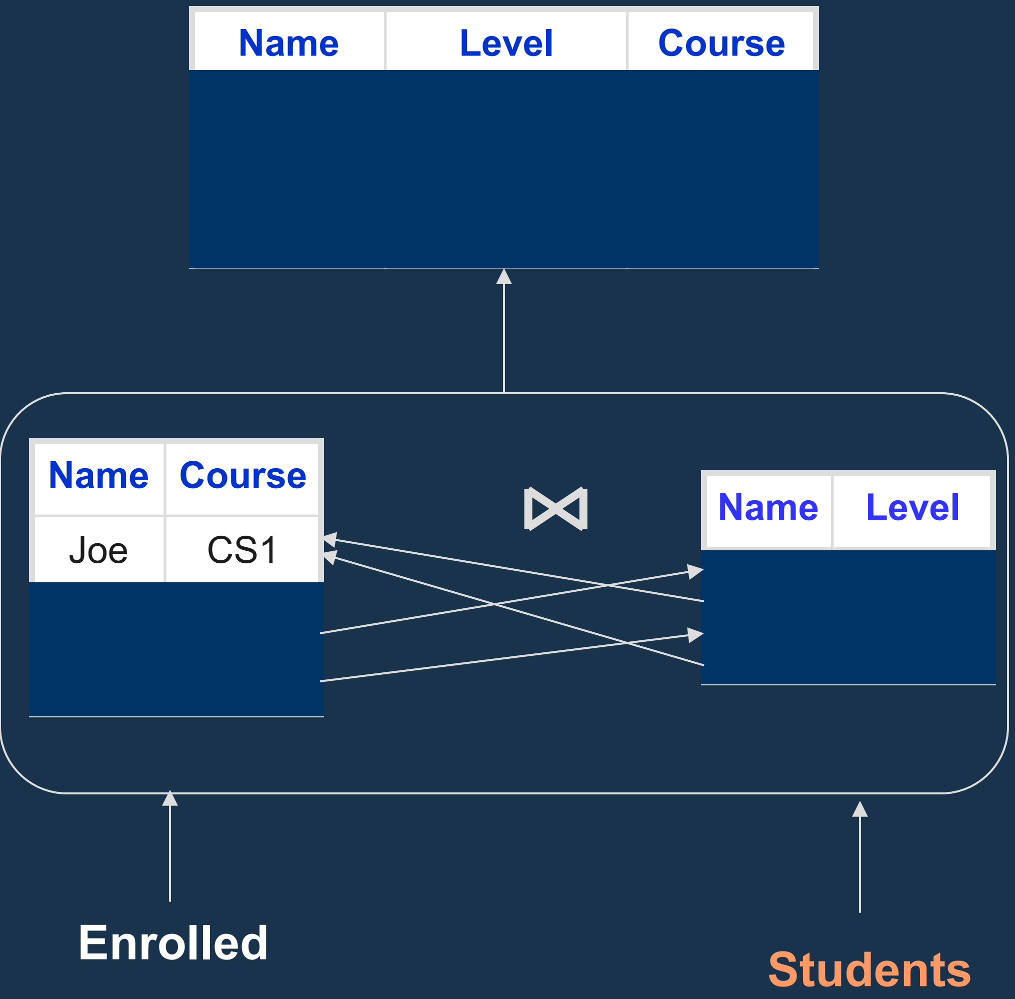
Name	Course
Joe	CS1
Jen	CS2

Enrolled

Symmetric/Pipelined Hash Join

[RS86, WA91]

select * from students, enrolled where students.name = enrolled.name



Name	Level	Course

Name	Course
Joe	CS1

Name	Level

- Simultaneously builds and probes hash tables on both sides
- Widely used:
  - adaptive query processing
  - stream joins
  - online aggregation
  - …
- Naïve version degrades to NLJ once memory runs out
  - Quadratic time complexity
  - memory needed = sum of inputs
- Improved by XJoins [UF 00], Tukwila DPJ [IFFLW 99]

Multi-way Pipelined Joins over Streaming Relations

Alternatives

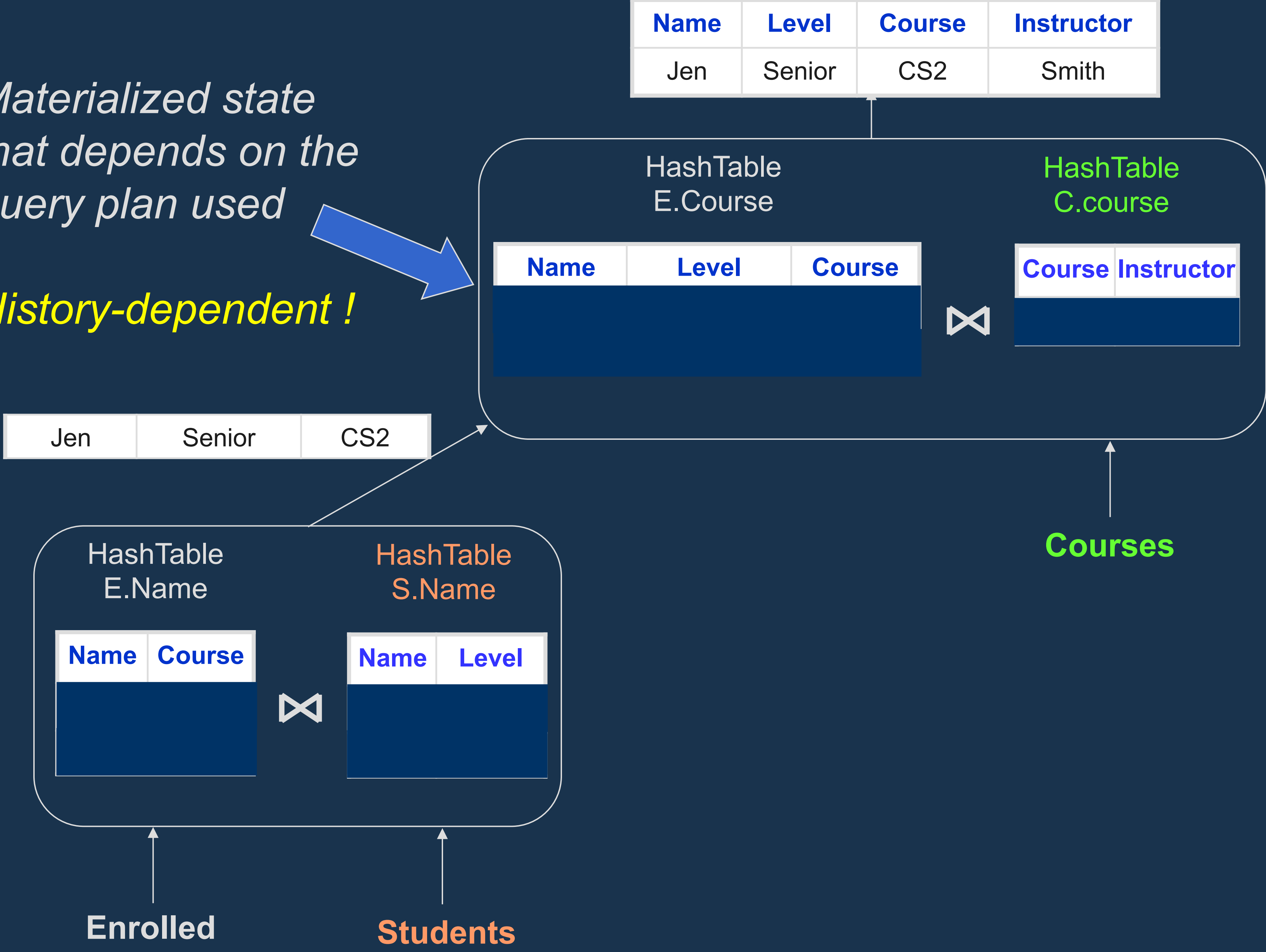
- Using binary join operators

- Using a single n-ary join operator (MJoin) [VNB'03]

- Some other options explored in the literature

Materialized state that depends on the query plan used

History-dependent !



Name	Level	Course	Instructor
Jen	Senior	CS2	Smith

HashTable E.Course

Name	Level	Course

⋈

HashTable C.course

Course	Instructor

Jen	Senior	CS2

HashTable E.Name

Name	Course

⋈

HashTable S.Name

Name	Level

Multi-way Pipelined Joins over Streaming Relations

Three alternatives

- Using binary join operators
  - *History-dependent execution*
  - Hard to reason about the impact of adaptation
  - May need to migrate the state when changing plans
- Using a single n-ary join operator (MJoin) [VNB'03]

Probing Sequences

Students tuple: Enrolled, then Courses

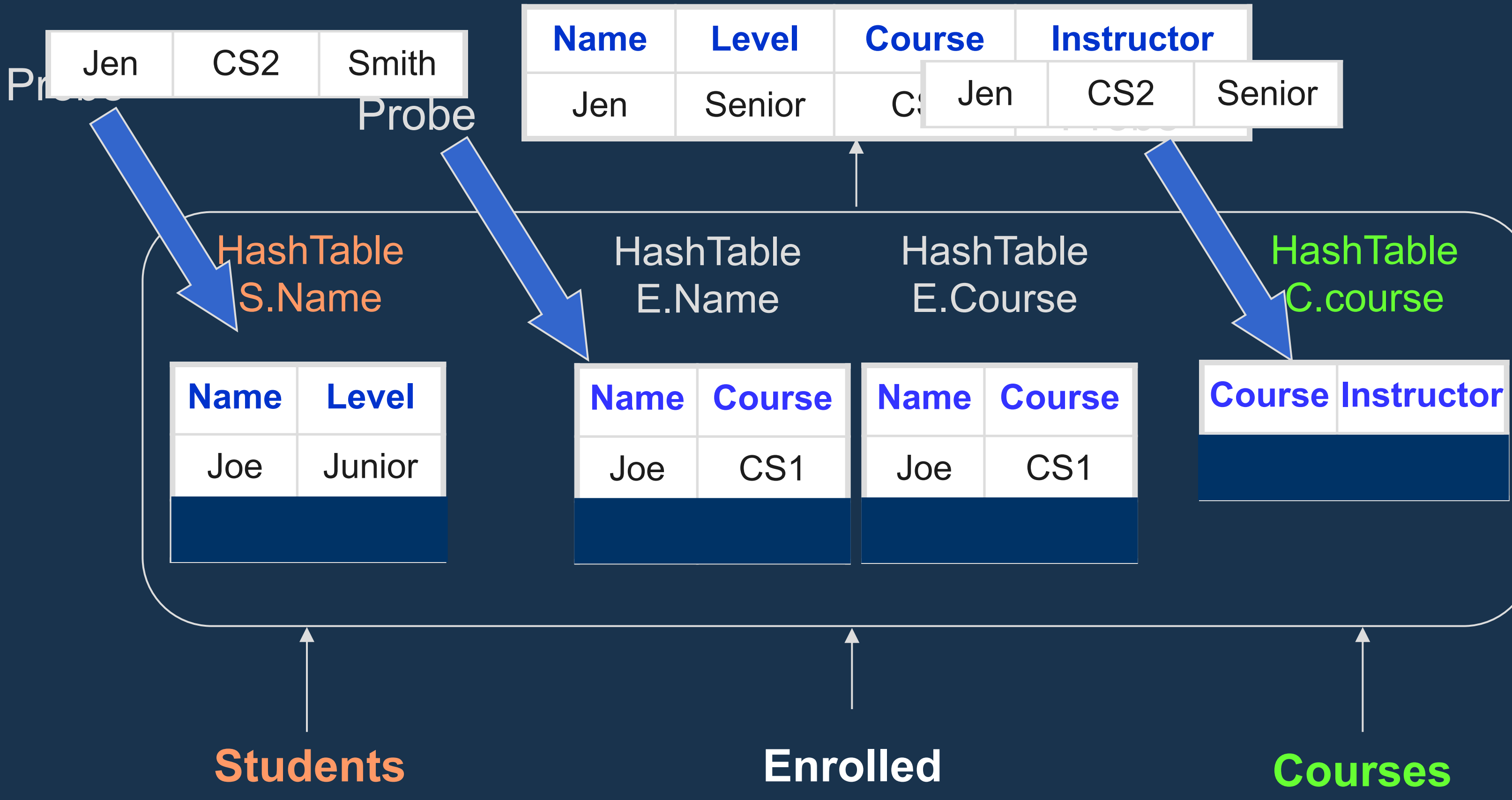
Enrolled tuple: Students, then Courses

Courses tuple: Enrolled, then Students

Hash tables contain all tuples that arrived so far

Irrespective of the probing sequences used

History-independent execution !



Jen	CS2	Smith

Name	Level	Course	Instructor
Jen	Senior	C	

Jen	CS2	Senior

HashTable S.Name

Name	Level
Joe	Junior

HashTable E.Name

Name	Course
Joe	CS1

HashTable E.Course

Name	Course
Joe	CS1

HashTable C.course

Course	Instructor

MJoins [VNB'03]

Choosing probing sequences

- For each relation, use a left-deep pipelined plan (based on hash indexes)
- Can use selection ordering algorithms

  Independently for each relation

Adapting MJoins

- Adapt each probing sequence independently

  e.g., StreaMon [BW'01] used A-Greedy for this purpose

A-Caching [BMWM'05]

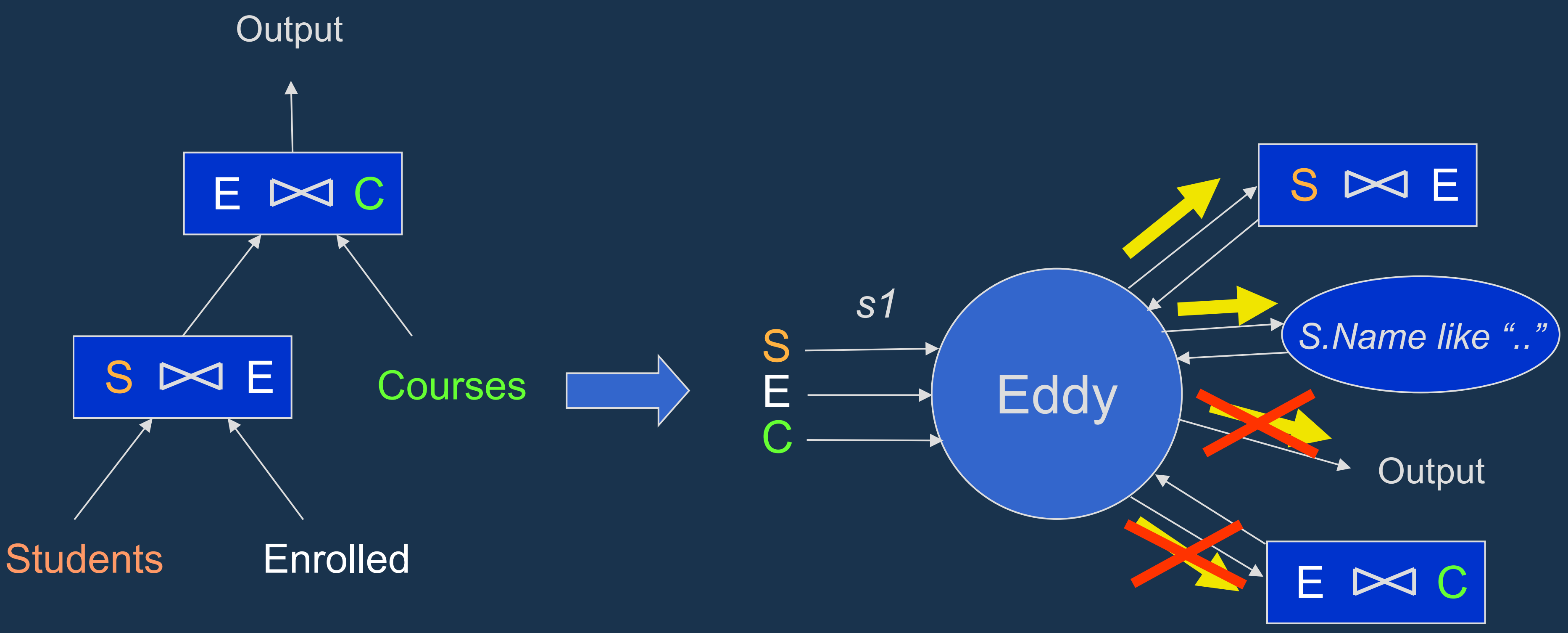
- Maintain intermediate caches to avoid recomputation
- Alleviates some of the performance concerns

Adaptive Join Processing: Outline

- Single streaming relation
  - Left-deep pipelined plans
- Multiple streaming relations
  - Execution strategies for multi-way joins
  - History-independent execution
    - MJoins
    - SteMs
  - History-dependent execution
    - Eddies with binary joins

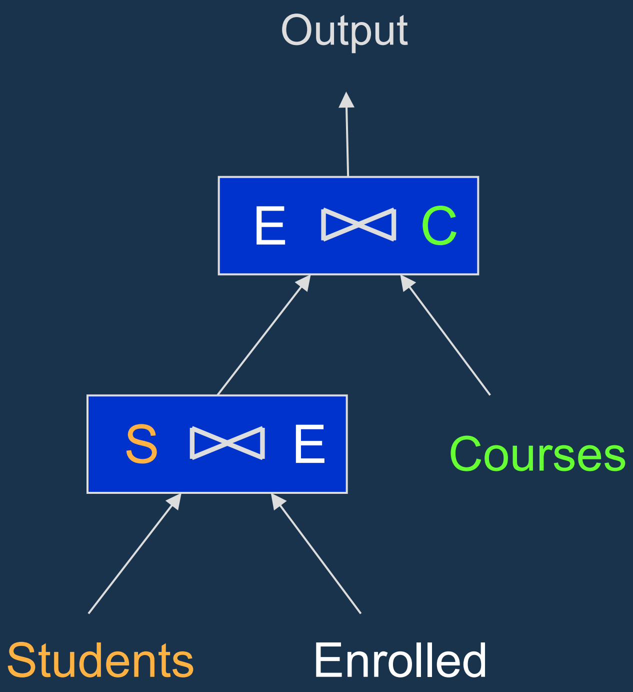
Eddies with Binary Joins [AH'00]

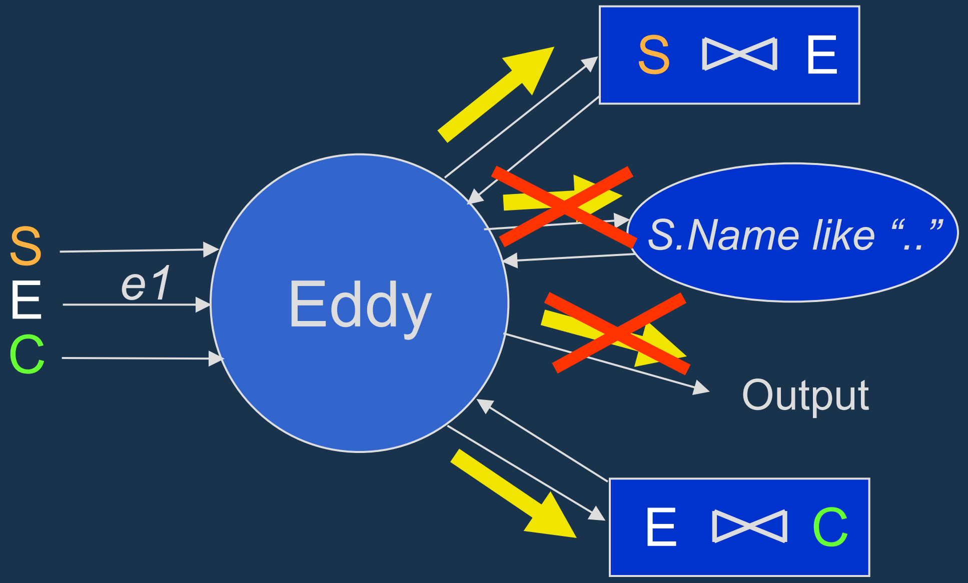
For correctness, must obey routing constraints !!



Eddies with Binary Joins [AH'00]

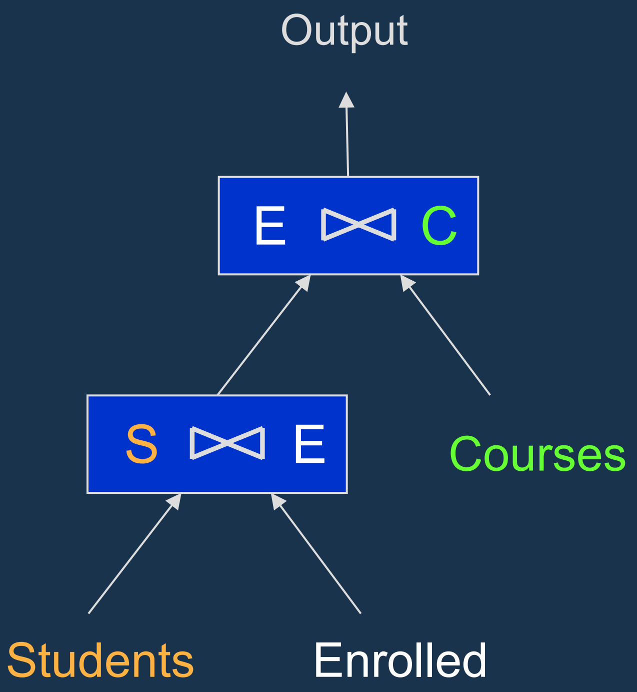
For correctness, must obey routing constraints !!

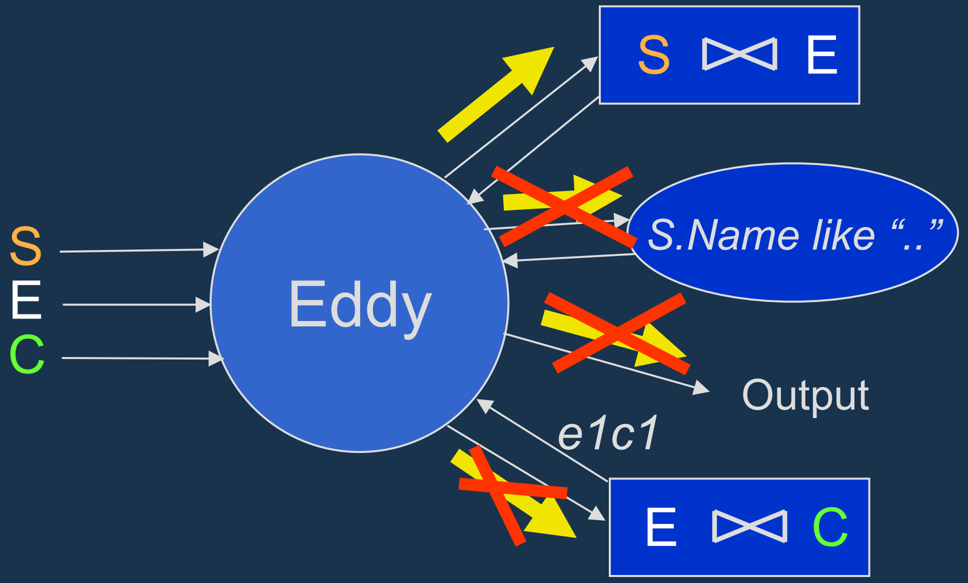




Eddies with Binary Joins [AH'00]

For correctness, must obey routing constraints !!
Use some form of *tuple-lineage*



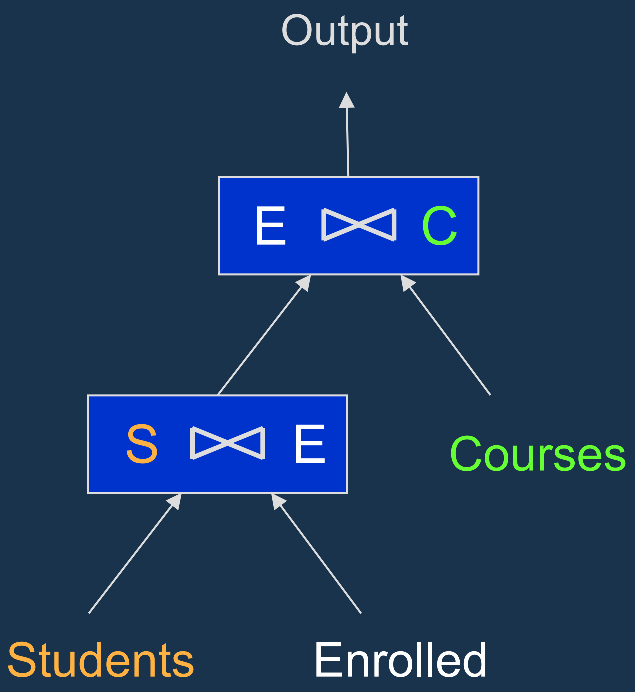


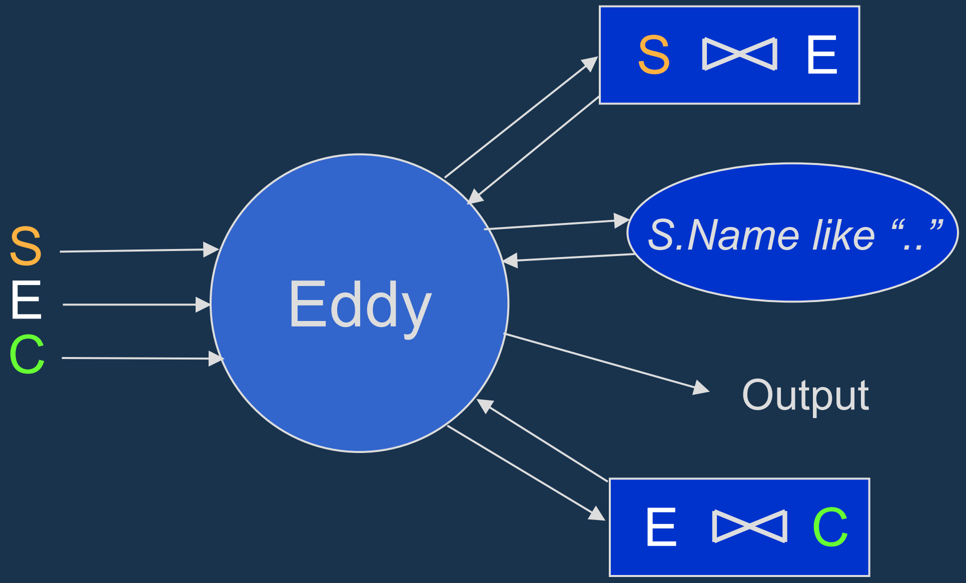
Eddies with Binary Joins [AH'00]

Can use any join algorithms

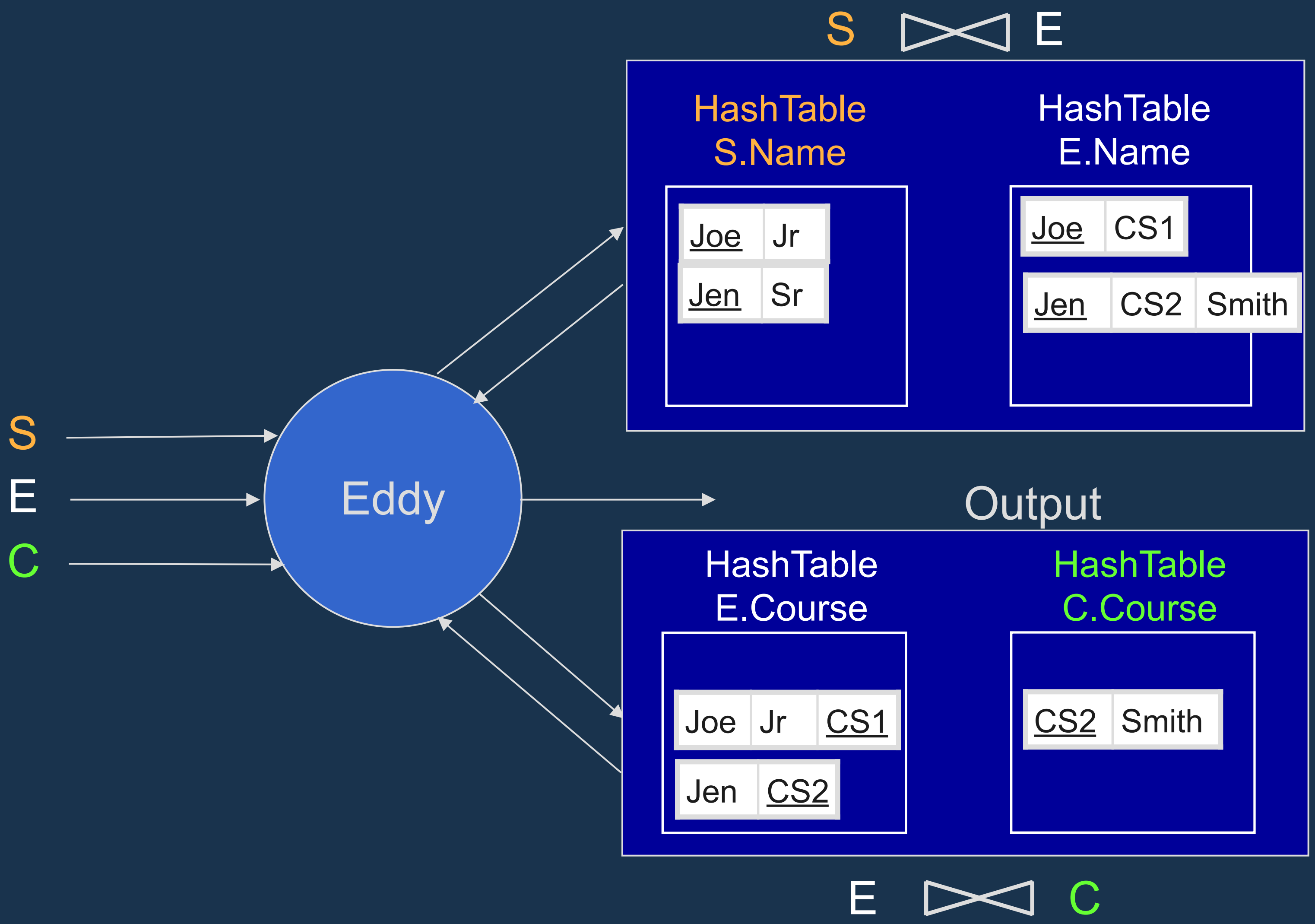
But, *pipelined* operators preferred

Provide quick feedback





Eddies with Symmetric Hash Joins



S ⋈ E

HashTable S.Name

Joe	Jr
Jen	Sr

HashTable E.Name

Joe	CS1	
Jen	CS2	Smith

S

E

C

Eddy

Output

HashTable E.Course

Joe	Jr	CS1
Jen	CS2	

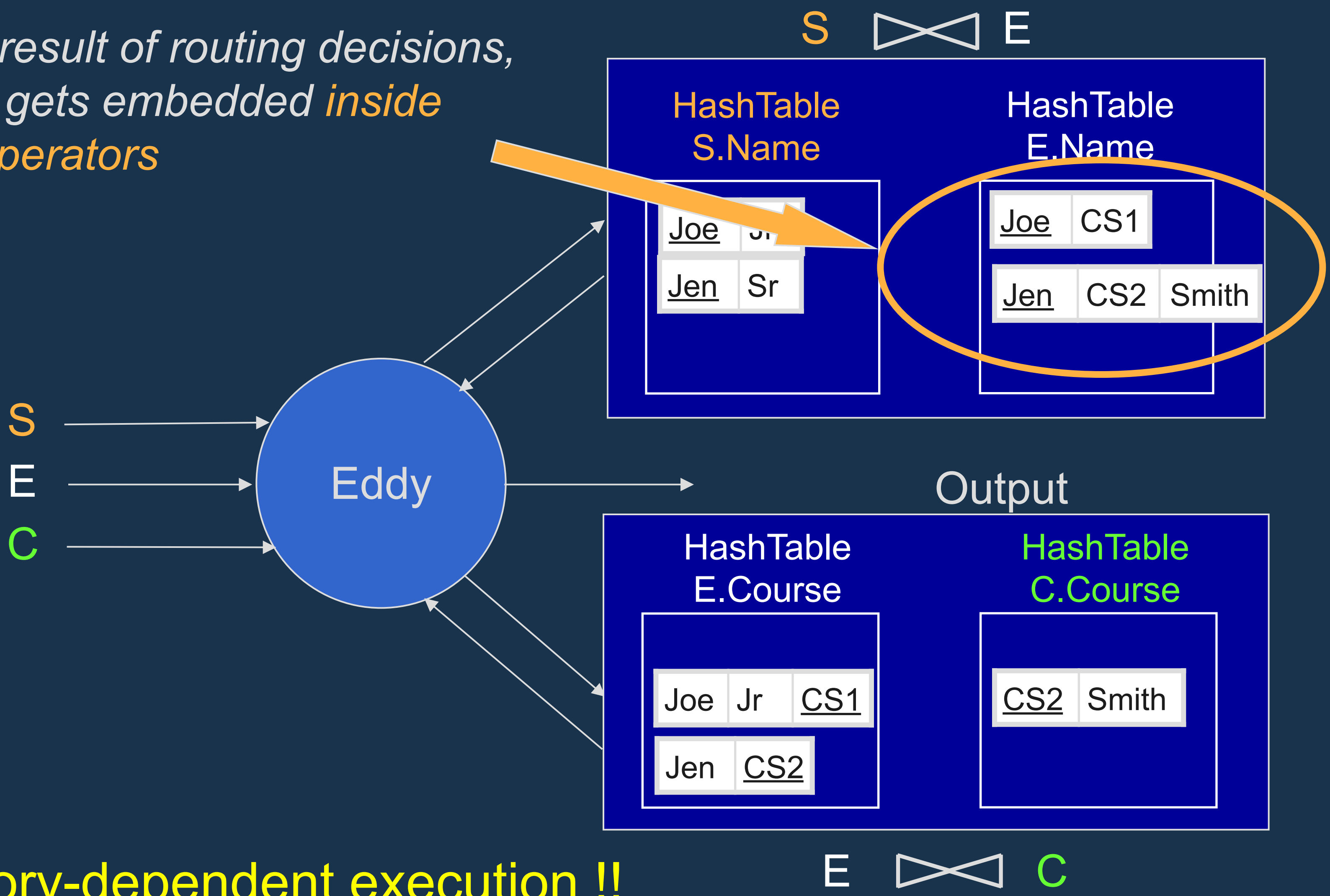
HashTable C.Course

CS2	Smith

E ⋈ C

Burden of Routing History [DH'04]

As a result of routing decisions, state gets embedded inside the operators



History-dependent execution !!

Recap: Eddies with Binary Joins

Routing constraints enforced using tuple-level lineage

Must choose access methods, join spanning tree beforehand

- SteMs relax this restriction [RDH’03]

The operator state makes the behavior unpredictable

- Unless only one streaming relation

Routing policies explored are same as for selections

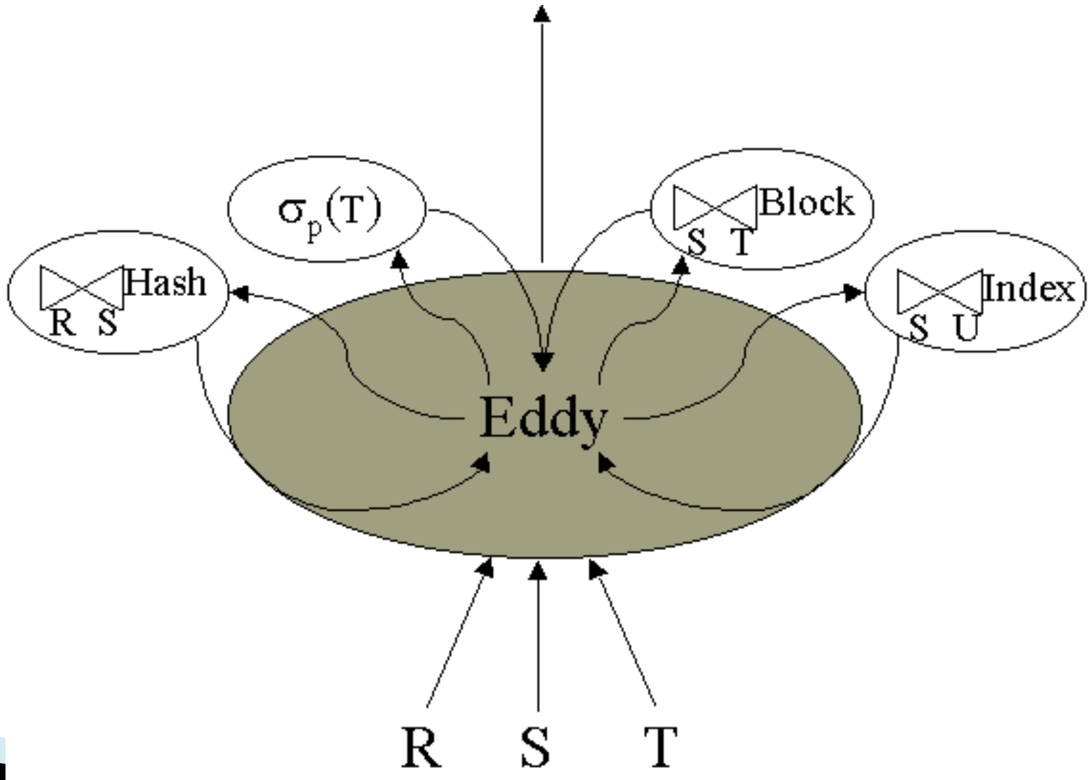
- Can tune policy for interactivity metric [RH’02]

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)

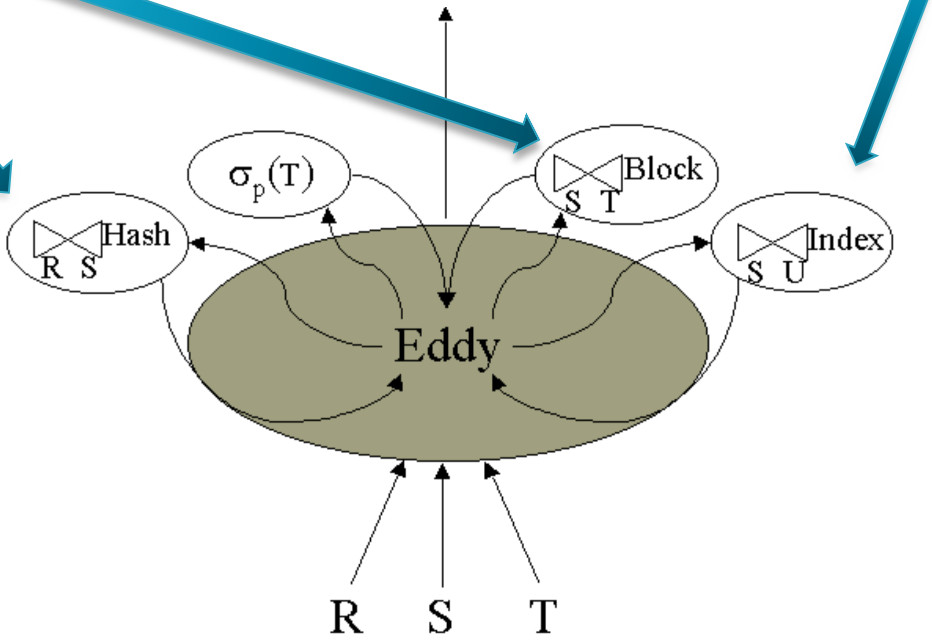


Eddies and Joins

- 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”

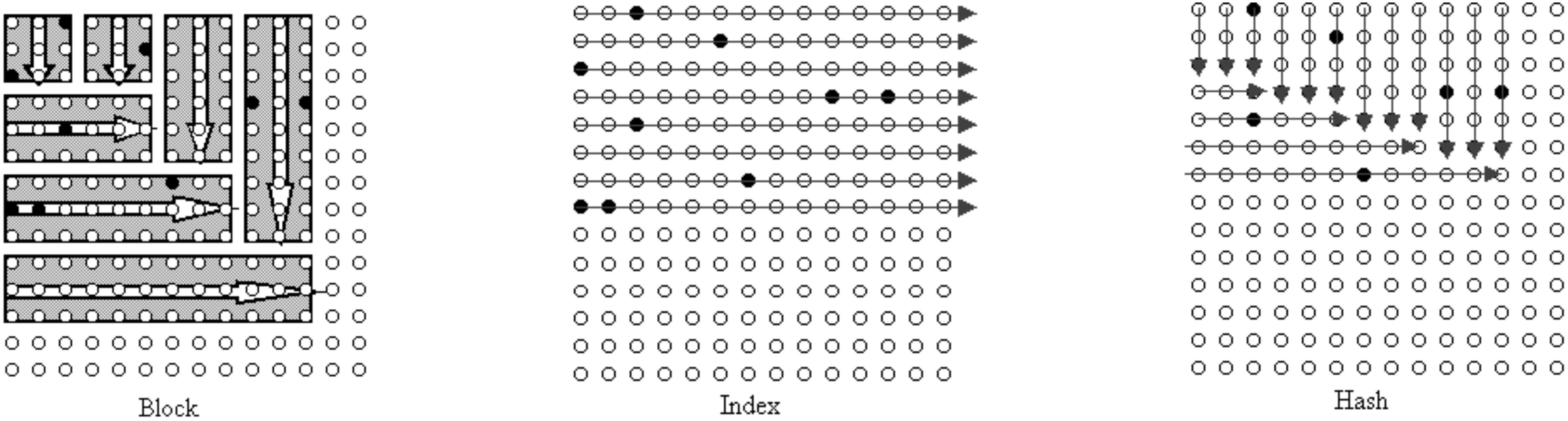
- These two are tricky
- Nested loops requires iterating over all of inner
- Hash join requires building a hash table on inner



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



Block

Index

Hash

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

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

- Trigger re-optimization during query execution if errors too high
- Through use of CHECK operators inserted into the query plan
  - Succeeds if the observed values within a range around the estimates
- If optimizer estimates accurate, the only overhead is the “couting” done by CHECK

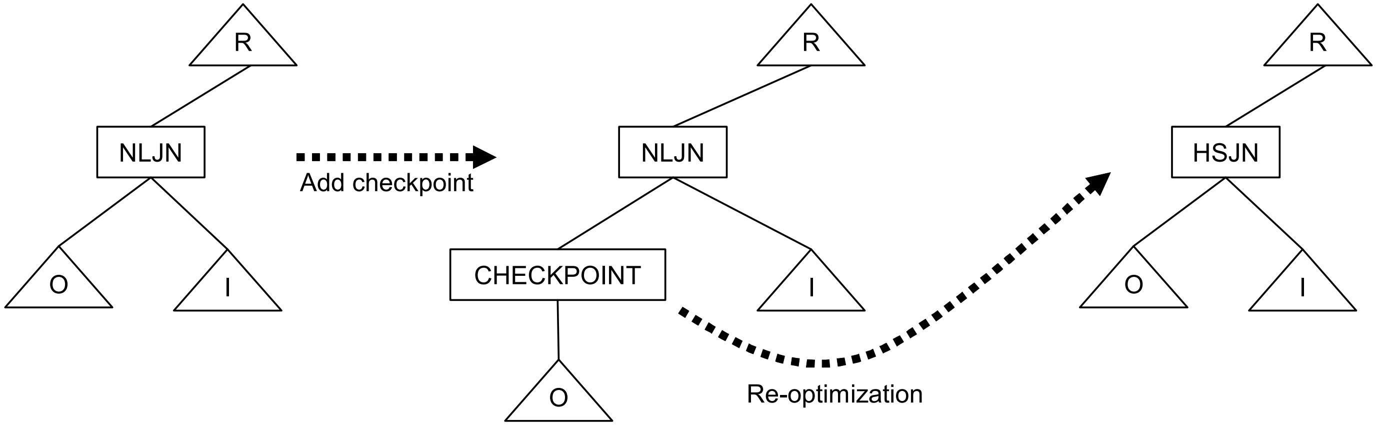


Figure 2: Adding CHECK to the outer of a NLJN

Overview

- Trigger re-optimization during query execution if errors too high
- Through use of CHECK operators inserted into the query plan
  - Succeeds if the observed values within a range around the estimates
- If optimizer estimates accurate, the only overhead is the "couting" done by CHECK
- If CHECK detects significant error, then "reoptimize"
  - Partial results made available to the optimizer to use if it wants (in the form of a materialized view)

Architecture

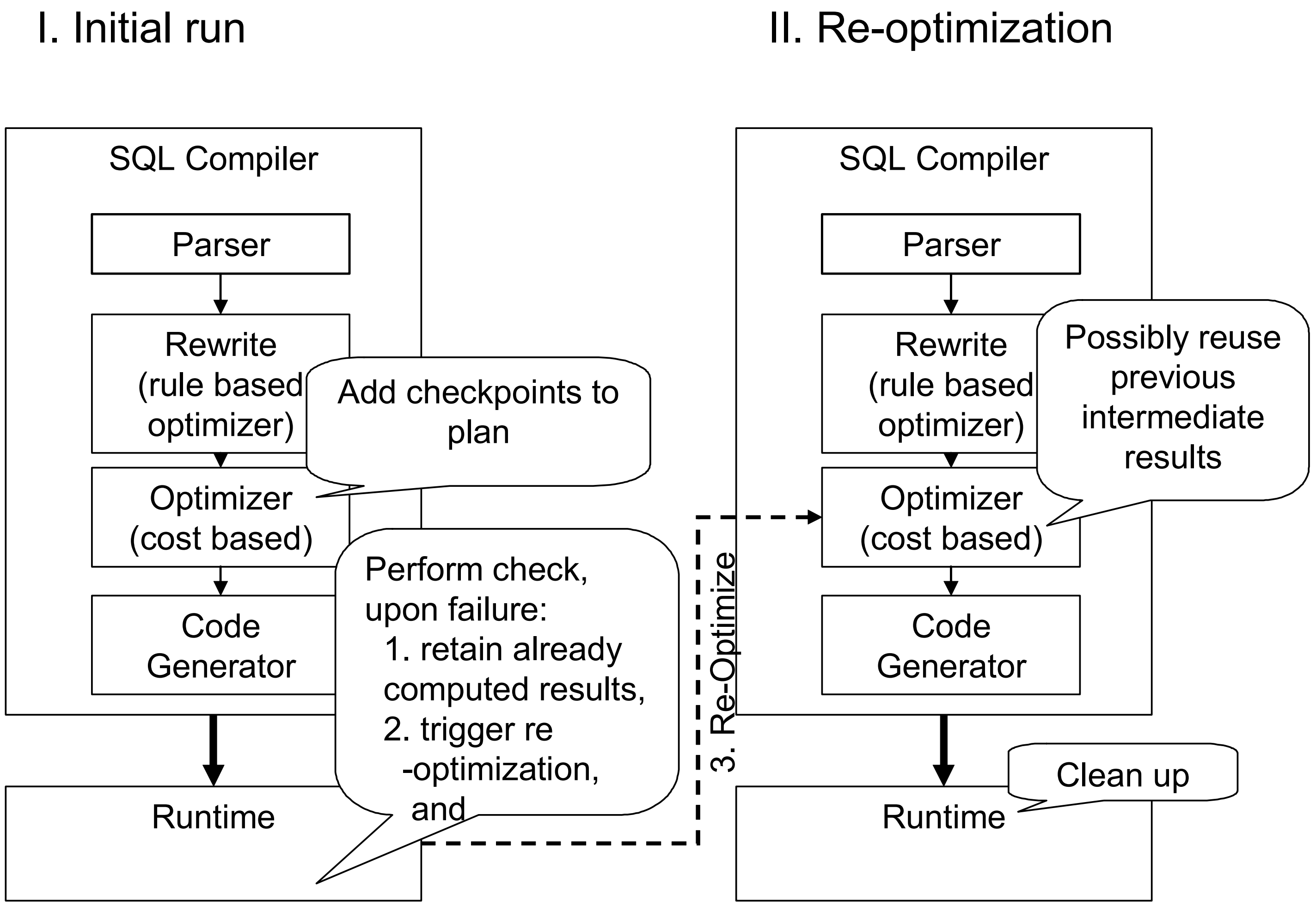
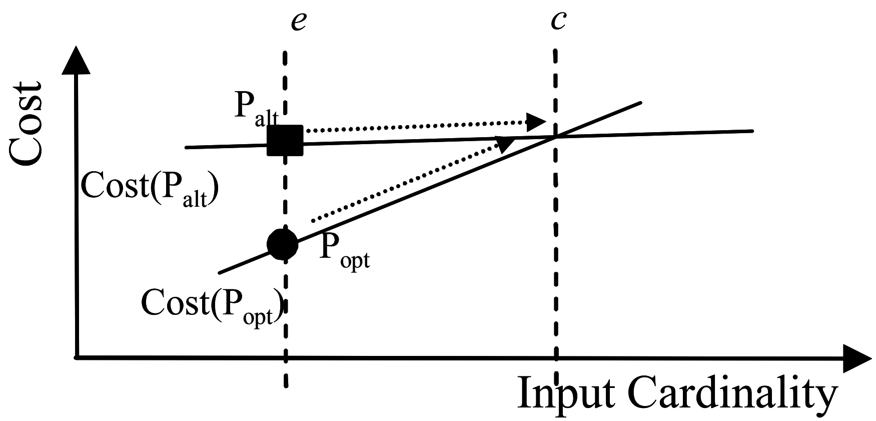


Figure 1: Progressive Optimization architecture

Computing Validity Ranges

- Helps only re-optimize when necessary
- The general problem is that of "parametric" optimization
  - i.e., find the best plan for each combination of parameters
  - Too expensive
- Instead:
  - Consider P1 and P2 -- two identical plans except for the top operator
  - Let cost(P1) < cost(P2) per the estimates → we would choose P1 over P2
  - Let "x" denote an edge into the top operator, and let "result(x) = e" denote the result flowing along "x"
  - Figure out: at what value of |result(x)|, we would have chosen P2 instead



Computing Validity Ranges

- Helps only re-optimize when necessary
- The general problem is that of “parametric” optimization
    - i.e., find the best plan for each combination of parameters
    - Too expensive
- Instead:
    - Consider P1 and P2 -- two identical plans except for the top operator
    - Let cost(P1) < cost(P2) per the estimates → we would choose P1 over P2
    - Let “x” denote an edge into the top operator, and let “result(x) = e” denote the result flowing along “x”
    - Figure out: at what value of |result(x)|, we would have chosen P2 instead
- Use numerical techniques to find these validity ranges

Reusing Partial Results

- Treat it as a materialized view, and let the optimizer decide
- If the plan under CHECK has a side-effect (e.g., update), then must reuse that plan (i.e., not redo that portion)
- In many cases, better not to use the partial result

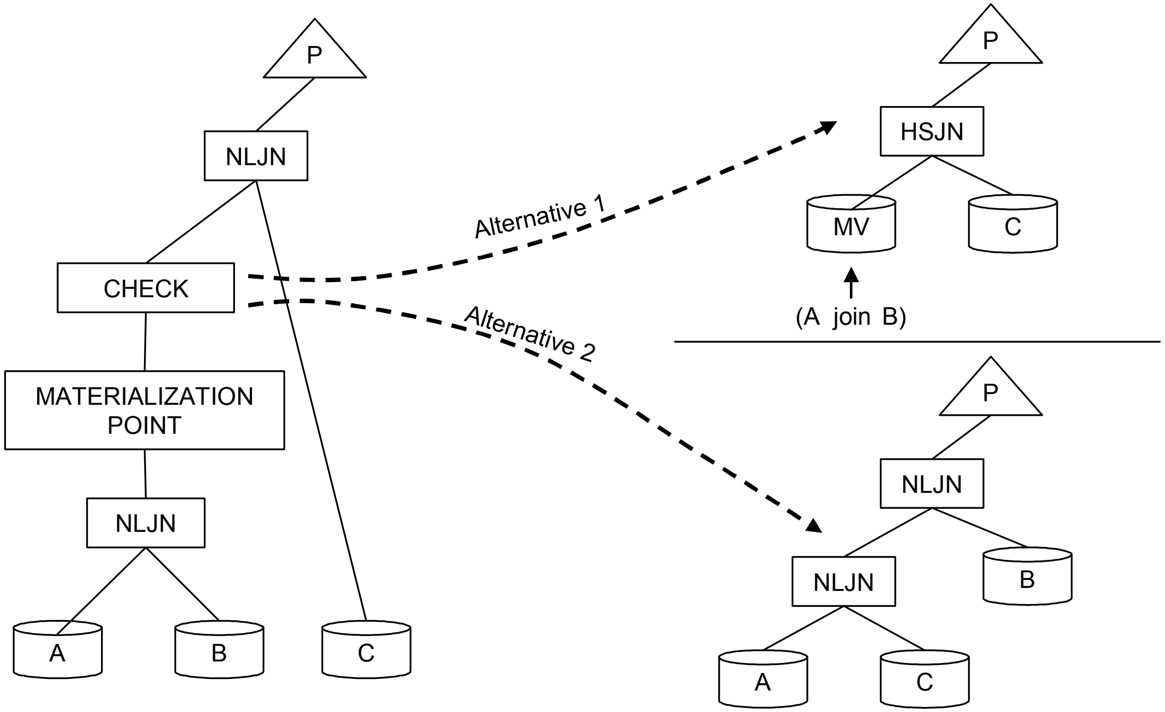


Figure 6: Two alternatives considered in re-optimization

Lazy vs Eager Checking

- If there is already a materialization point, can add CHECK there for free (lazy)
- Can add explicit materialization along with a CHECK
  - Extra overhead in doing that
- Eager CHECKs don't wait for materialization
- ECWC (Eager without compensation)
  - There is a materialization afterwards → no results will be output to the user
  - So can easily reoptimize without worrying about compensation

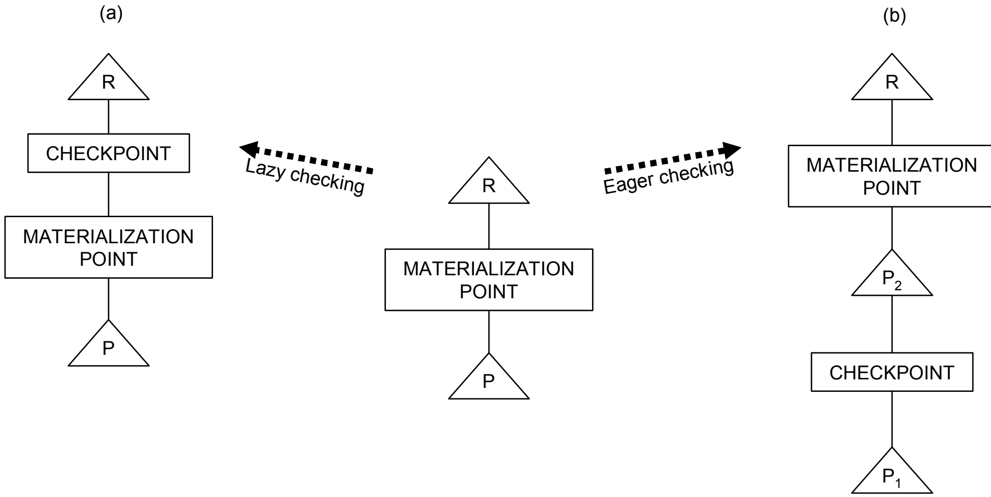


Figure 7: Lazy checking (LC) and eager checking without compensation (ECWC)

Eager Checking

- With Buffering: Buffer results until you are sure things are okay
  - Delays the pipeline for some time

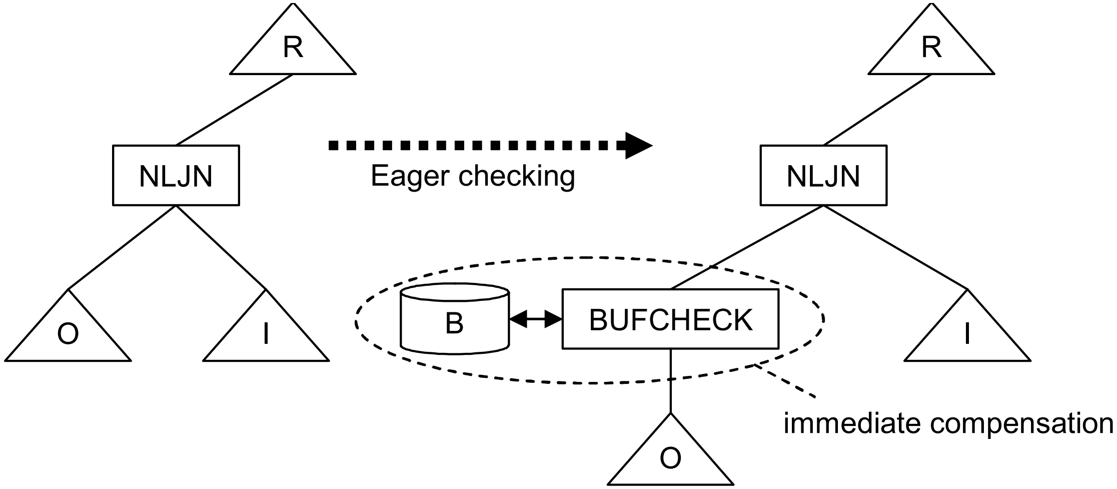
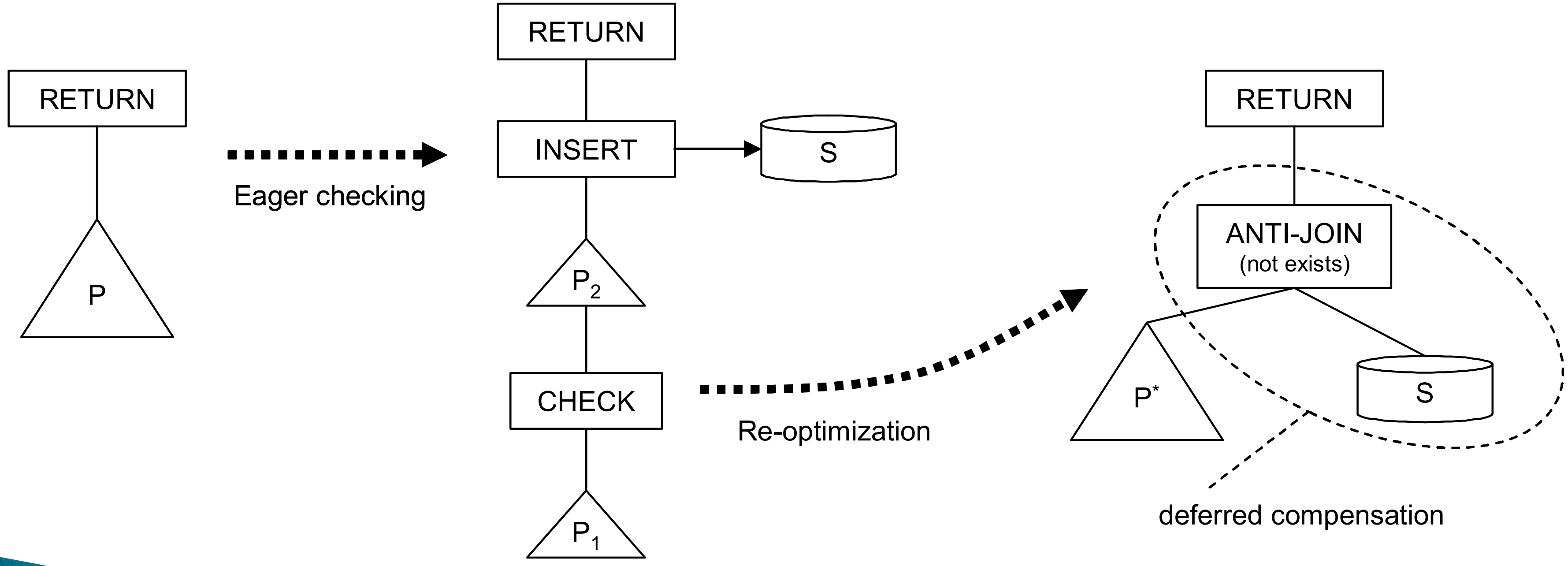


Figure 8: Eager checking with Buffering

Eager Checking

- With Deferred Compensation
  - Keep track of what tuples have already been output
  - Check that side table before outputting new tuples after reoptimization
  - Potentially a lot of repeated work



Experiments

- Degradation in some cases -- sometimes two errors cancelled each other out in the original plan

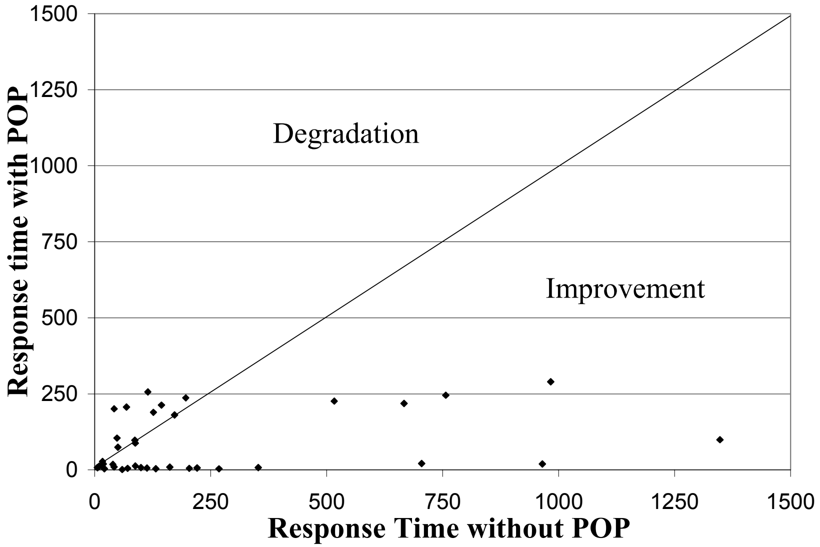


Figure 15: Scatter Plot of Response Times with and without POP on the DMV database

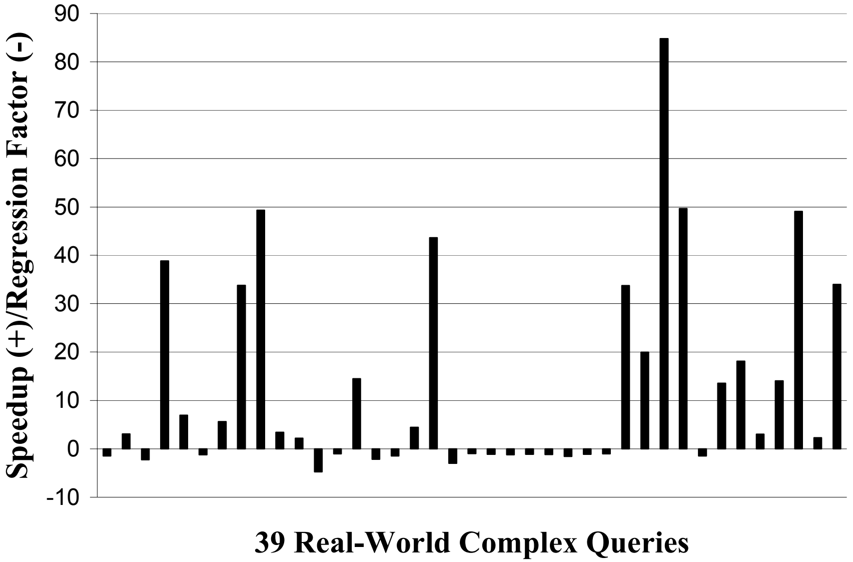


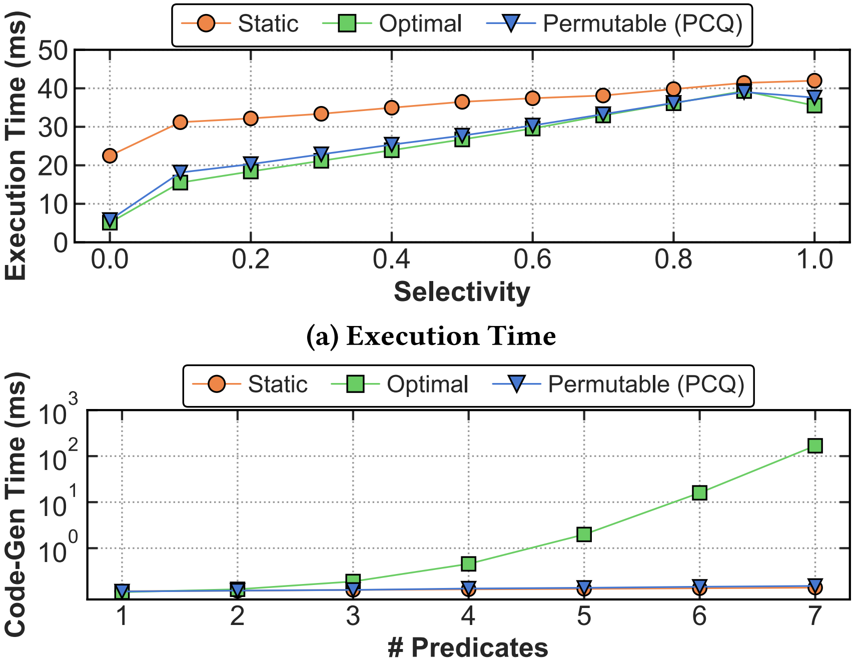
Figure 16: Speedup and Regression of each Query

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

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

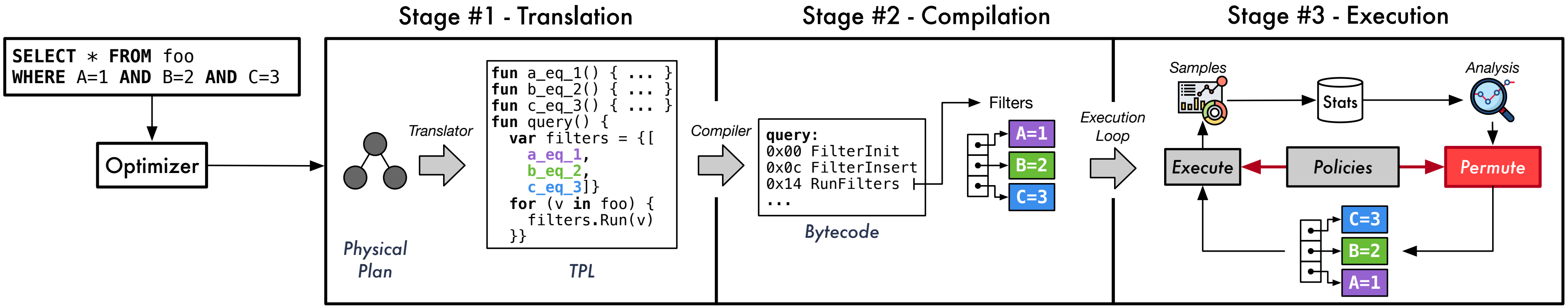
(b) Code-Generation Time

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)



```
SELECT * FROM foo
WHERE A=1 AND B=2 AND C=3
```

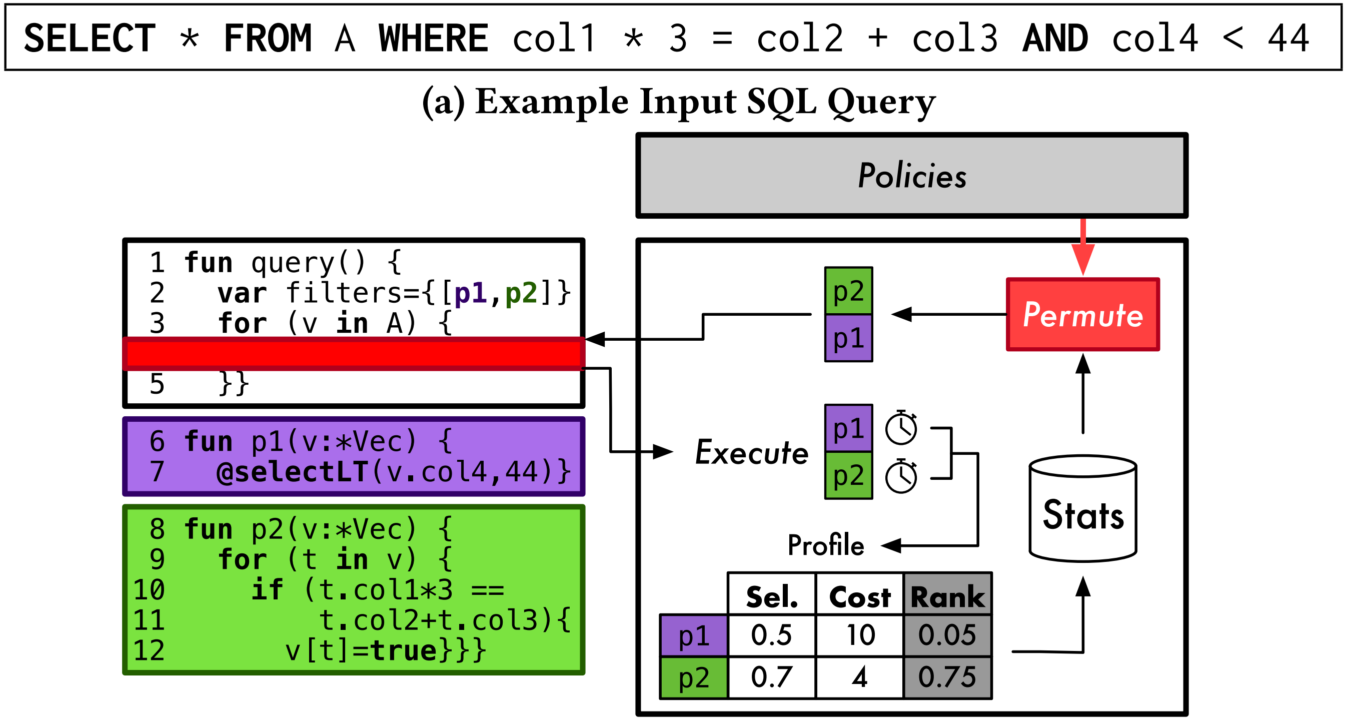
```
fun a_eq_1() { ... }
fun b_eq_2() { ... }
fun c_eq_3() { ... }
fun query() {
  var filters = {[
    a_eq_1,
    b_eq_2,
    c_eq_3]}
  for (v in foo) {
    filters.Run(v)
  }}
```

```
query:
0x00 FilterInit
0x0c FilterInsert
0x14 RunFilters
...
```

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

```
fun query() {
  var filters={[p1,p2]}
  for (v in A) {
[redacted]
  }}
```

```
fun p1(v:*Vec) {
  @selectLT(v.col4,44)}
```

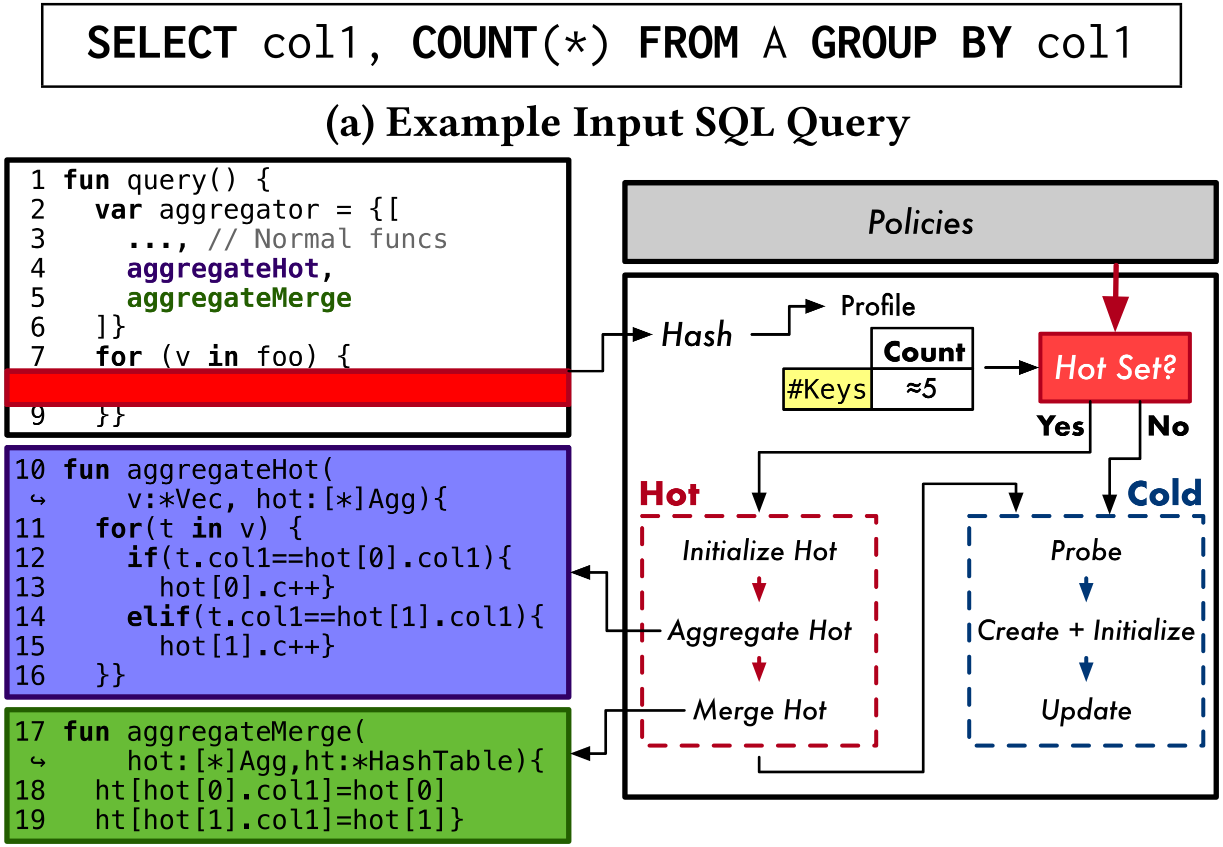
```
fun p2(v:*Vec) {
  for (t in v) {
    if (t.col1*3 ==
          t.col2+t.col3){
      v[t]=true}}}
```

	Sel.	Cost	Rank
p1	0.5	10	0.05
p2	0.7	4	0.75

(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

```
fun query() {
  var aggregator = {[
    ..., // Normal funcs
    aggregateHot,
    aggregateMerge
  ]}
  for (v in foo) {
  }}
```

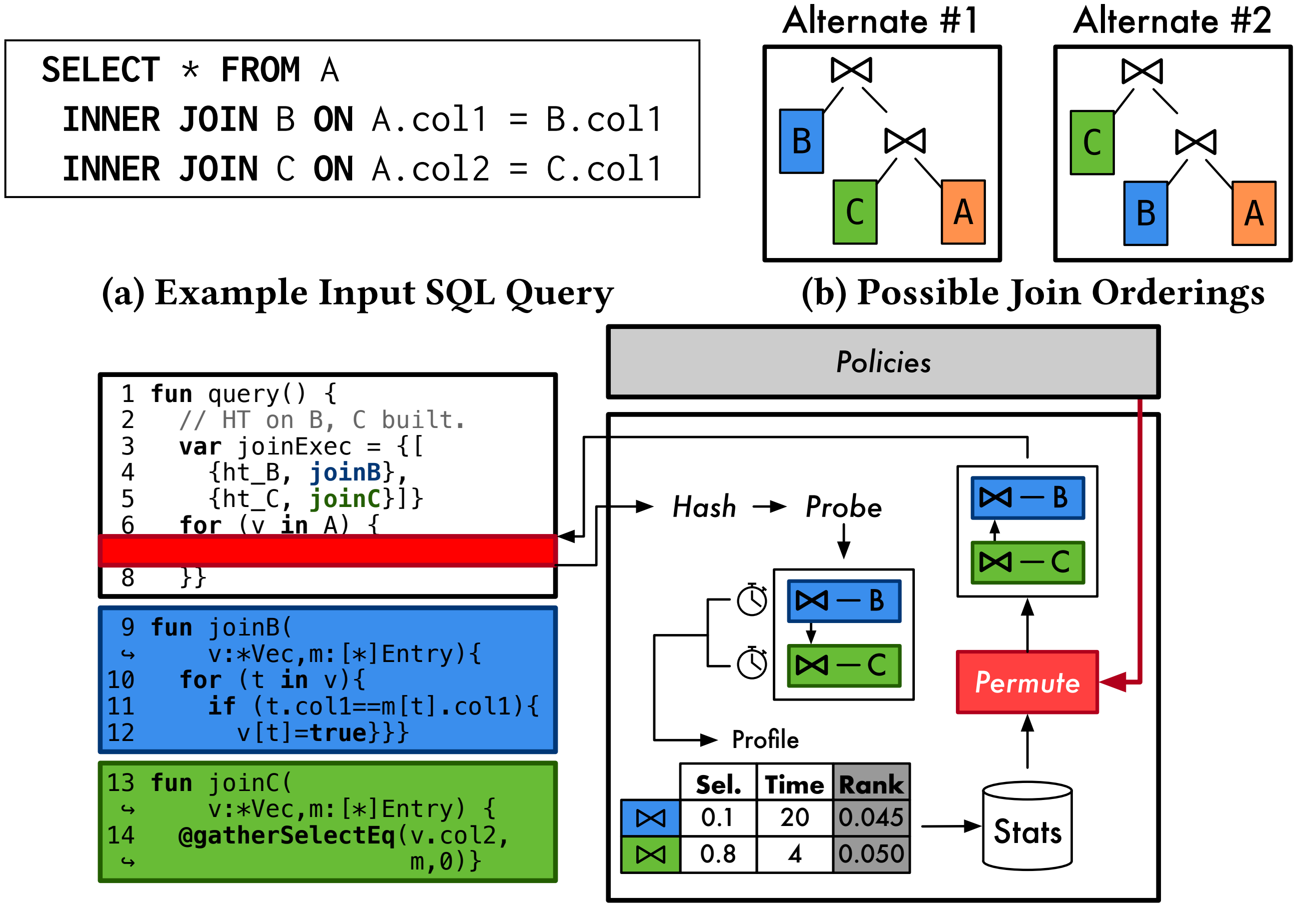
```
fun aggregateHot(
    v:*Vec, hot:[*]Agg){
  for(t in v) {
    if(t.col1==hot[0].col1){
      hot[0].c++}
    elif(t.col1==hot[1].col1){
      hot[1].c++}
  }}
```

```
fun aggregateMerge(
    hot:[*]Agg,ht:*HashTable){
  ht[hot[0].col1]=hot[0]
  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



```
SELECT * FROM A
  INNER JOIN B ON A.col1 = B.col1
  INNER JOIN C ON A.col2 = C.col1
```

(a) Example Input SQL Query

(b) Possible Join Orderings

```
fun query() {
  // HT on B, C built.
  var joinExec = {[
    {ht_B, joinB},
    {ht_C, joinC}]}
  for (v in A) {

  }}
```

```
fun joinB(
    v:*Vec,m:[*]Entry){
  for (t in v){
    if (t.col1==m[t].col1){
      v[t]=true}}}
```

```
fun joinC(
    v:*Vec,m:[*]Entry) {
  @gatherSelectEq(v.col2,
                  m,0)}
```

	Sel.	Time	Rank
⋈	0.1	20	0.045
⋈	0.8	4	0.050

(c) Generated Code and Execution of Permutable 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

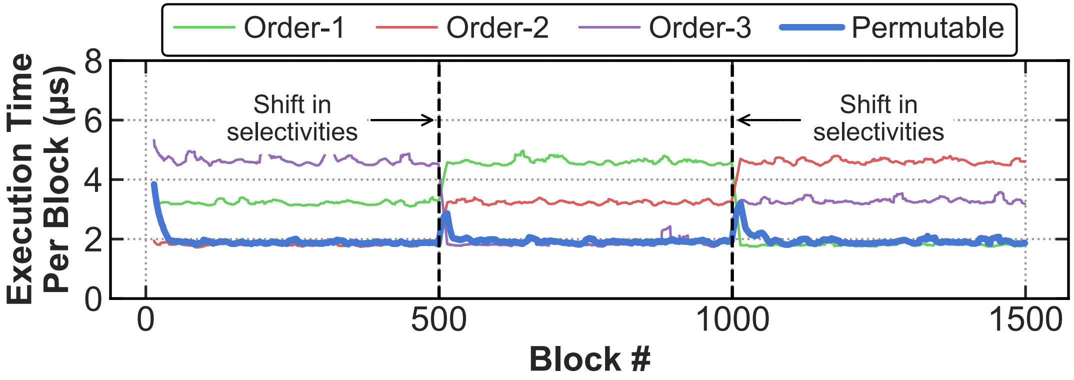


Figure 6: Performance Over Time – Execution time of three static filter orderings and our PCQ filter during a sequential table scan.

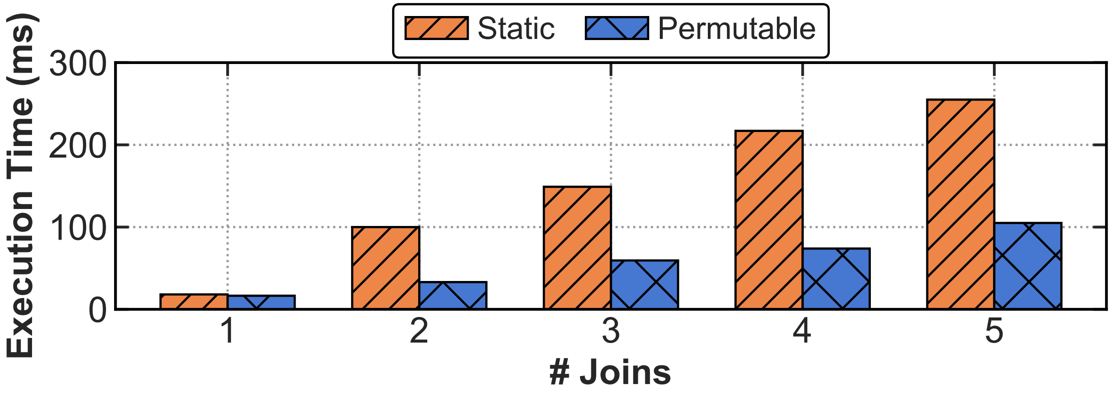
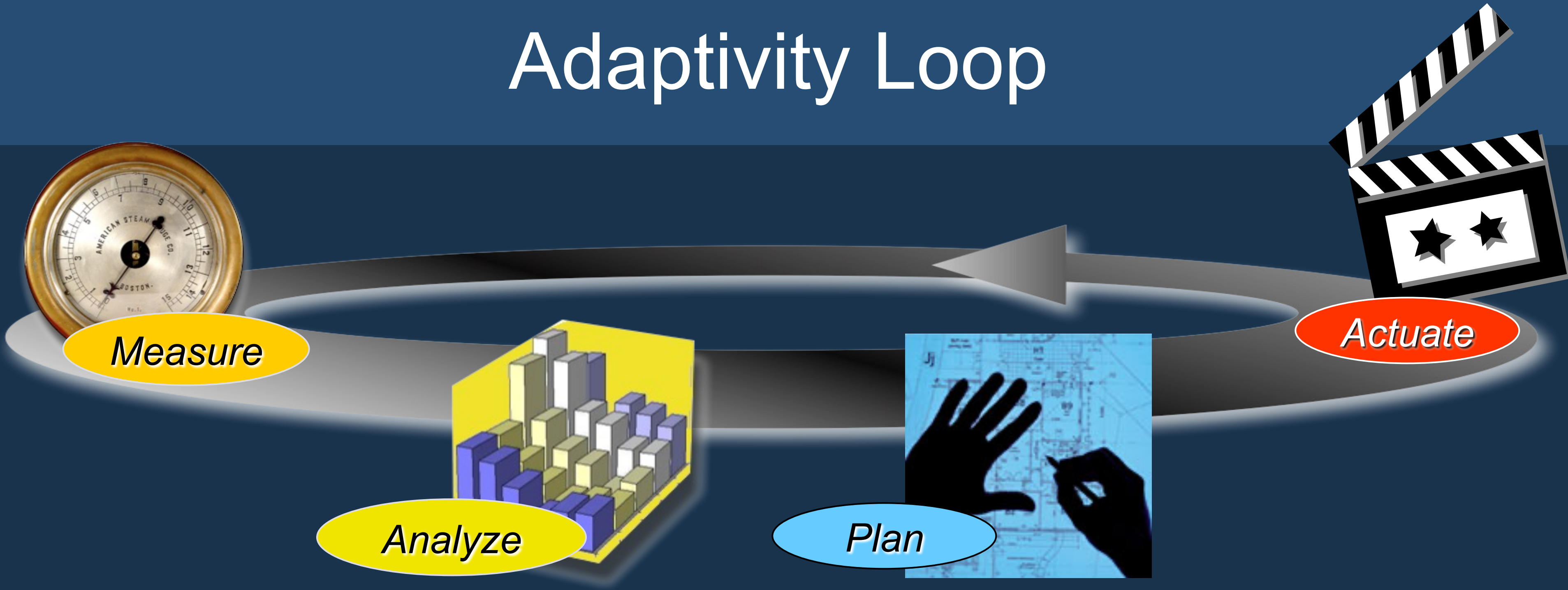


Figure 12: Varying Number of Joins – Execution time to perform a multi-step join while keeping the overall join selectivity at 10%.

Adaptivity Loop



Measure what ?

Cardinalities/selectivities, operator costs, resource utilization

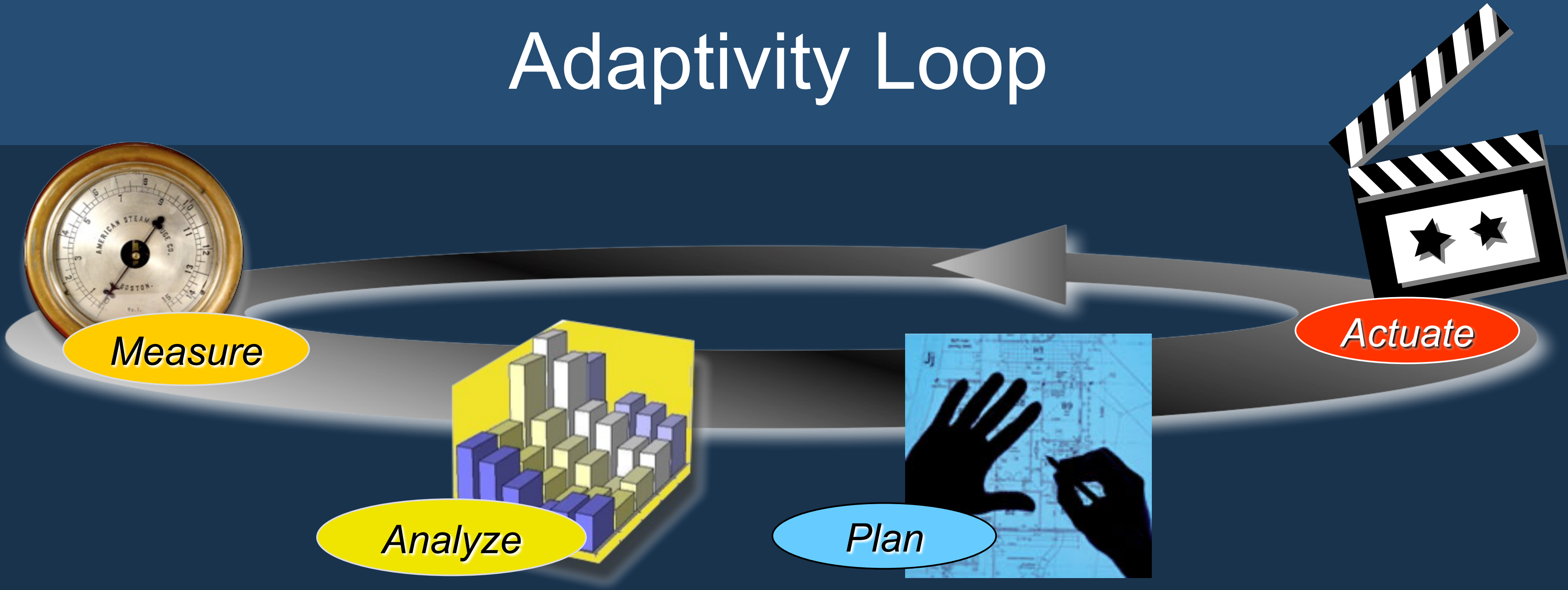
Measure when ?

Continuously (eddies); using a random sample (A-greedy); at materialization points (mid-query reoptimization)

Measurement overhead ?

Simple counter increments (mid-query) to very high

Adaptivity Loop



Analyze/replan what decisions ?

(Analyze actual vs. estimated selectivities)

Evaluate costs of alternatives and switching (keep state in mind)

Analyze / replan when ?

Periodically; at materializations (mid-query); at conditions (A-greedy)

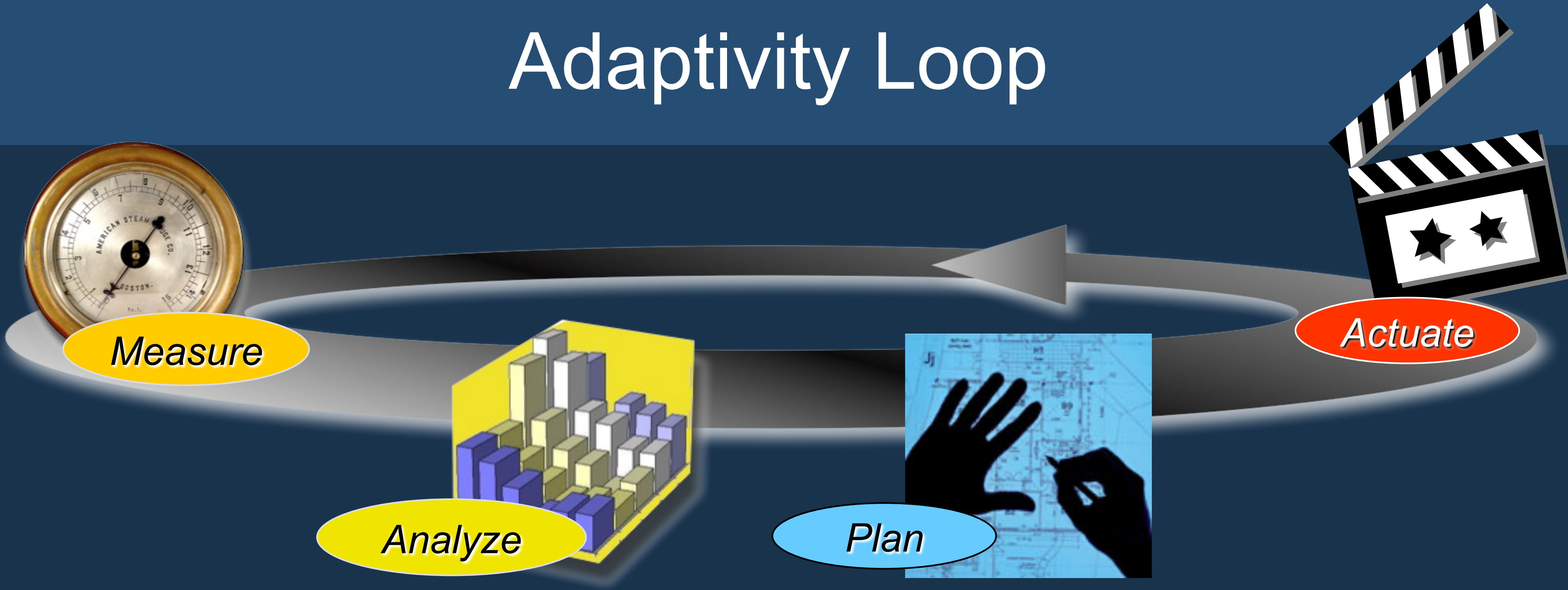
Plan how far ahead ?

Next tuple; batch; next stage (staged); possible remainder of plan (CQP)

Planning overhead ?

Switch stmt (parametric) to dynamic programming (CQP, mid-query)

Adaptivity Loop



Actuation: How do they switch to the new plan/new routing strategy ?

Actuation overhead ?

At the end of pipelines → free (mid-query)

During pipelines:

History-independent → Essentially free (selections, MJoins)

History-dependent → May need to migrate state (STAIRs, CAPE)

Recap/Thoughts

- Not much work on adaptive query processing in the last 10 years
  - SkinnerDB [2019] another relevant work
- More work on adapting the execution of a single operator
  - e.g., changing things based on available resources
- Likely to re-emerge as an important topic in the next few years
  - As QP in many systems becomes more mature…
  - As SQL starts becoming more and more common as the query language (e.g., in Spark, Pandas, etc).