Flow Algorithms for Parallel Query Optimization

Amol Deshpande, University of Maryland

Lisa Hellerstein, Polytechnic University, Brooklyn

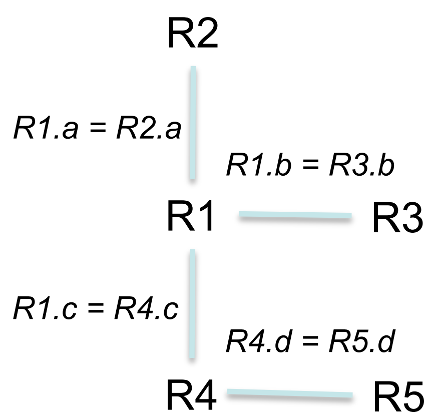
Introduction and Motivation

- Motivation: Parallel Query Processing
  - Increasing parallelism in computing
    - Shared-nothing clusters, multi-core technology, Grid, P2P…
  - Two ways to exploit parallelism:
    - Partitioned parallelism
      - Operator copies run in parallel on partitions of data
      - Harder to set up, more communication overheads
    - Pipelined Parallelism
      - Each operator run on a different processor
      - Better cache locality, easier to reason about
      - May be the only option in some scenarios
      - Cannot exploit the parallelism fully

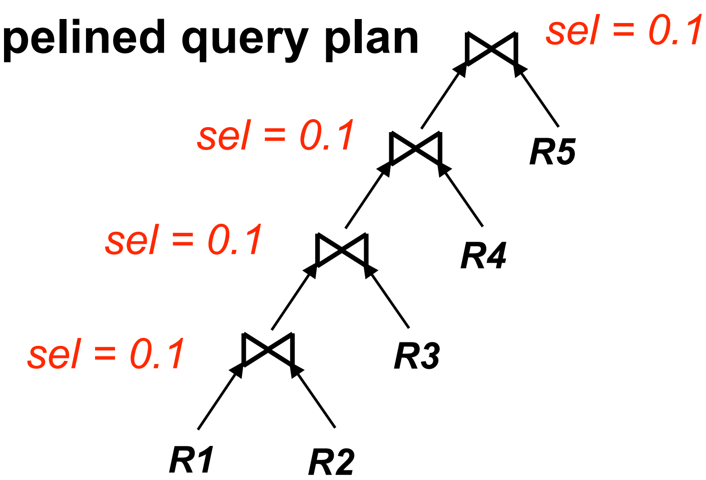
Motivation: Parallel Query Processing

Example query

select *
from R1, R2, R3, R4, R5
where R1.a = R2.a and
R1.b = R3.b and
R1.c = R4.c and
R4.d = R5.d

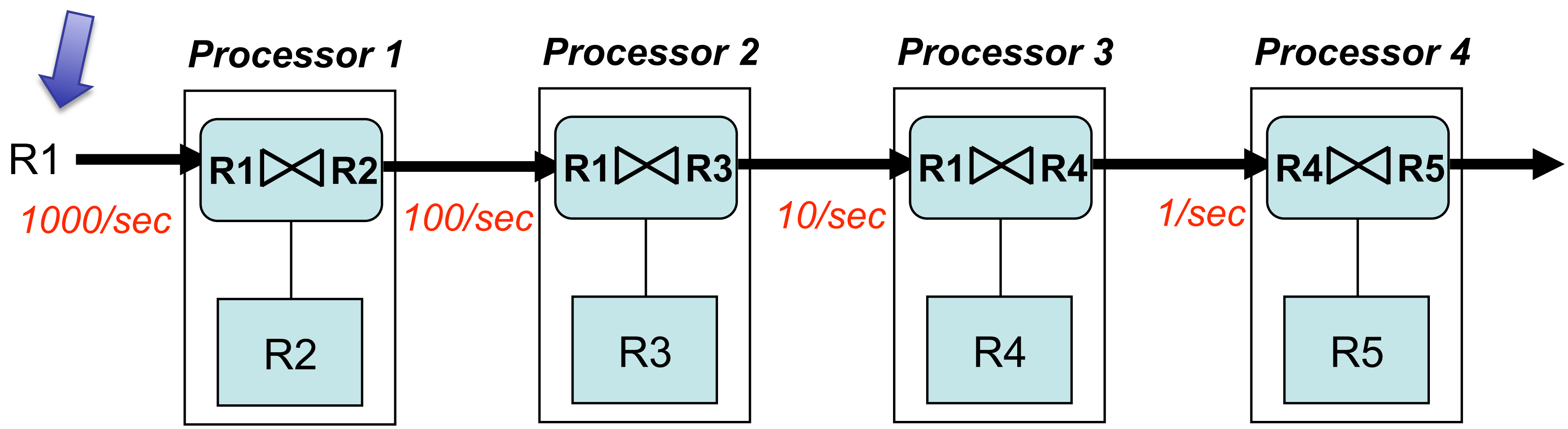


A pipelined query plan



Tuple Throughput = 1000 tuples/sec

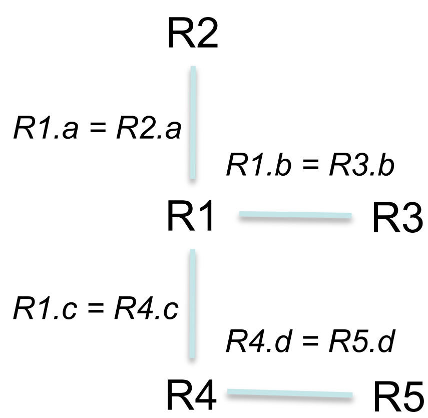
Driver relation



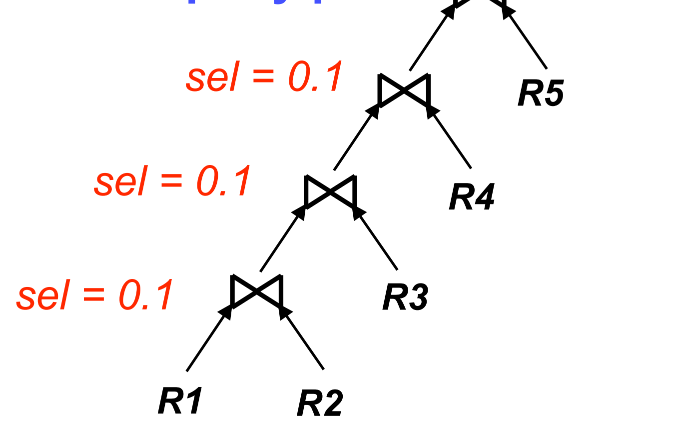
Motivation: Parallel Query Processing

Example query

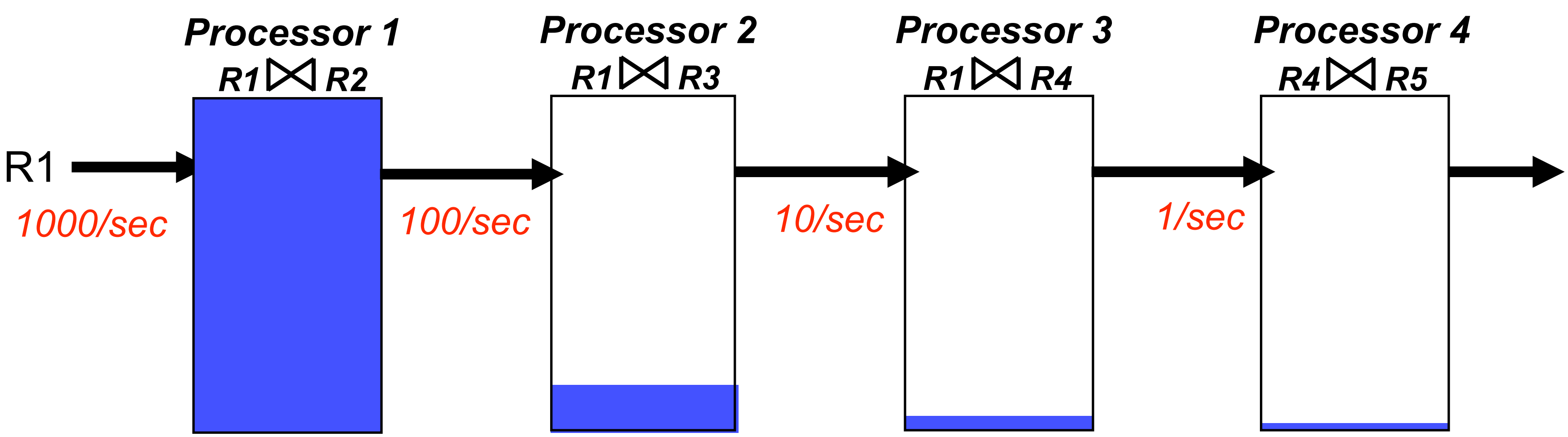
select *
from R1, R2, R3, R4, R5
where R1.a = R2.a and
R1.b = R3.b and
R1.c = R4.c and
R4.d = R5.d



A pipelined query plan



Tuple Throughput = 1000 tuples/sec

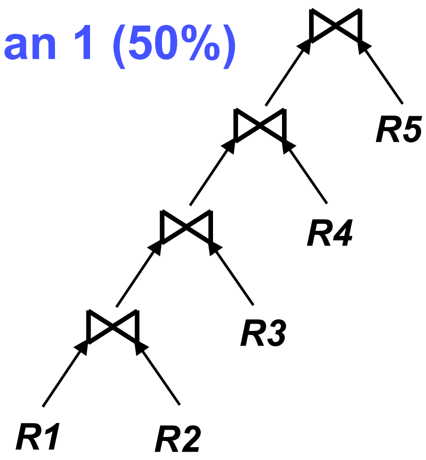


Proposed Solution: Interleaving Plans

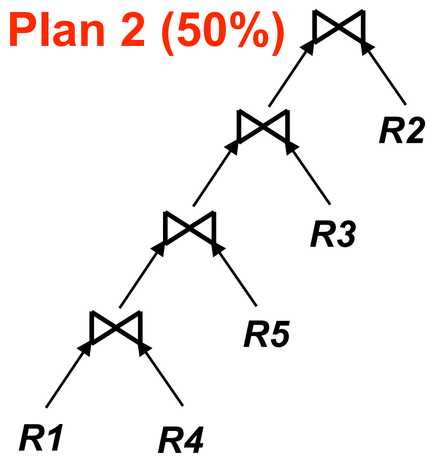
Example query

select *
from R1, R2, R3, R4, R5
where R1.a = R2.a and
R1.b = R3.b and
R1.c = R4.c and
R4.d = R5.d

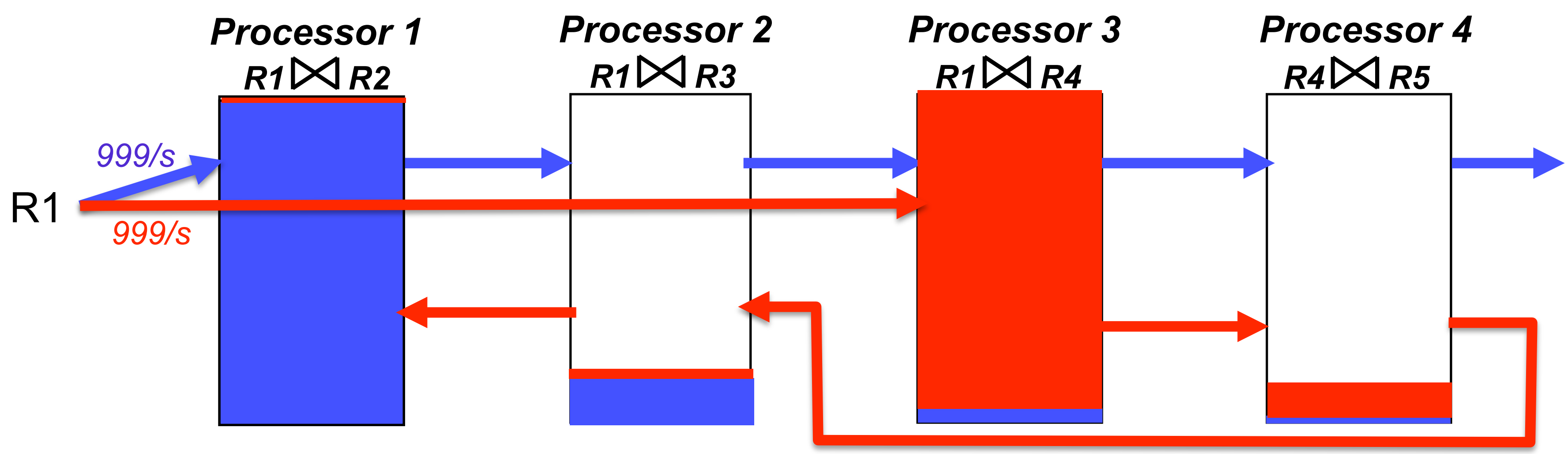
Plan 1 (50%)



Plan 2 (50%)



Tuple Throughput ≈ 1998 tuples/sec *(max = 2790 tuples/sec)*



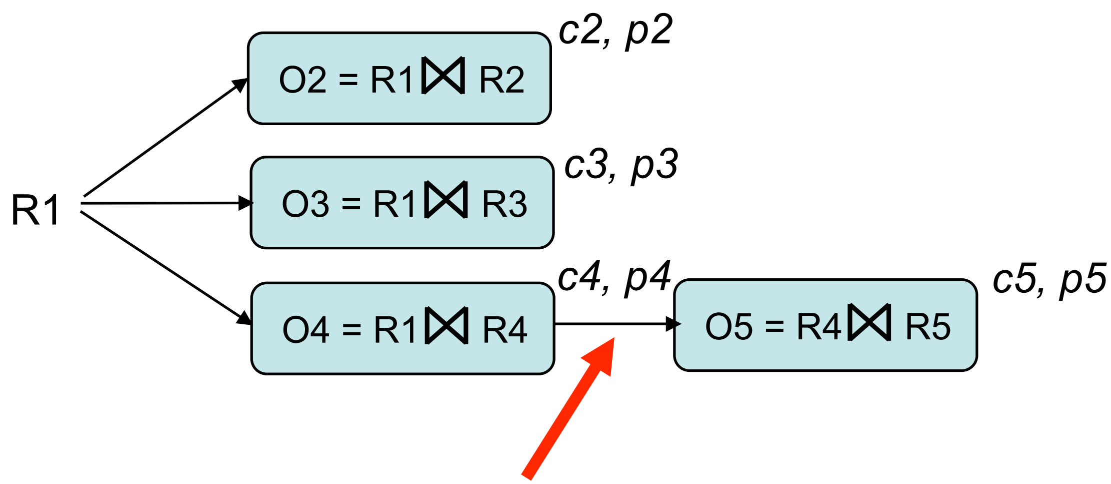
Introduction and Motivation

- Motivation: Selection ordering with precedence constraints

Given a driver relation, choosing a left-deep pipelined plan for a multi-way join query is equivalent to precedence-constrained selection ordering

Example query

select *
from R1, R2, R3, R4, R5
where R1.a = R2.a and
 R1.b = R3.b and
 R1.c = R4.c and
 R4.d = R5.d



R1 tuples need to join with R4 before joining with R5

Cost of *O4*: *c4 = average per-tuple cost of the join* $R1 \bowtie R4$

Selectivity of *O4*: *p4 = fanout of* $R1 \bowtie R4$

= average number of R4 matches for an R1 tuple ***(may be > 1)***

Introduction and Motivation

- Motivation: Selection ordering with precedence constraints
- Motivation: Query Processing over Web Services
  - Increasing abundance of web services and standardized APIs for querying them
    - Shopping, Web Search, Housing etc. …
  - Similar issues as pipelined query processing
    - Each web service == a processor
    - Typically limited number of requests allowed per minute
- Motivation: Similar to many problems in other domains
  - Sequential testing (e.g. for fault detection) [SF'01, K'01]
  - Learning with attribute costs [KKM'05]

Prior Related Work

- Rich literature on parallel and distributed query processing
  - Didn't consider interleaving plans
- Interleaving Plans for Selection Ordering [Condon et al., 2006]
  - Simpler types of queries
  - *$O(n^2)$* algorithm for computing the optimal plan
- Query Optimization over Web Services [Srivastava et al.,2006]
  - Algorithm for choosing an optimal *serial (single)* plan
  - Considered cyclic queries and a larger plan space
- Eddies [Avnur and Hellerstein, 2000] ?
  - Interleaving plans are not adaptive
  - Distributed eddies [Tian and DeWitt, 2006]: Similar metrics, but they focus on adaptivity

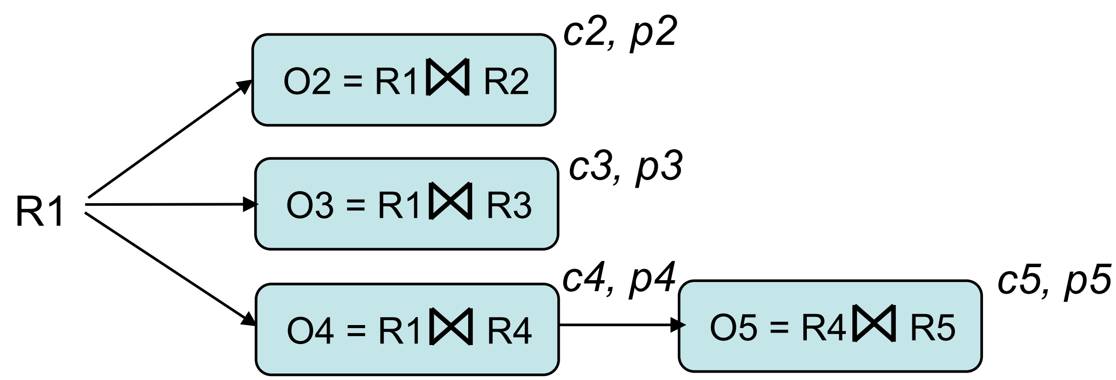
Outline

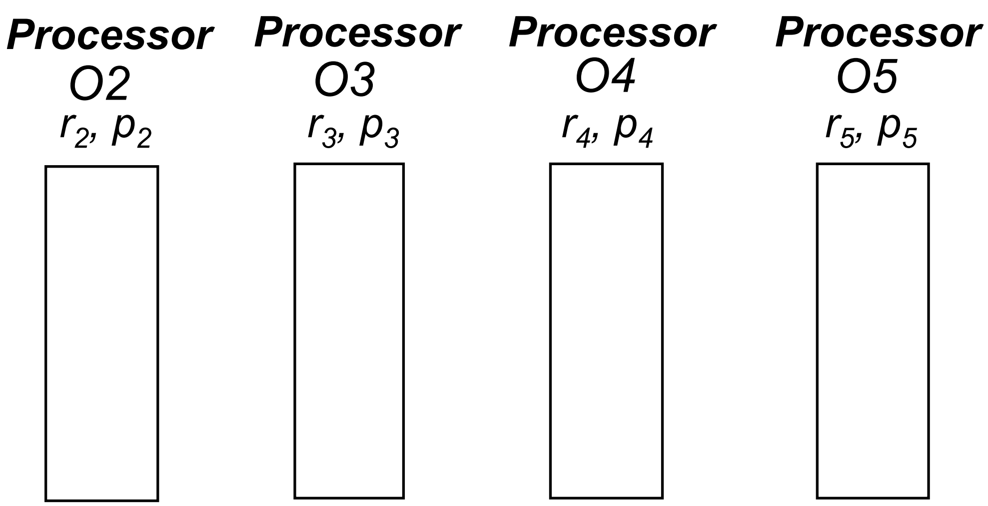
- Introduction and Motivation
- Problem Definition
- Algorithms for finding Optimal Interleaving Plans
  - Selective operators
  - Non-selective operators
- Experimental Results

Parallel Execution Model

- Each operator runs on a different processor

Example query
select *
from R1, R2, R3, R4, R5
where R1.a = R2.a and
  R1.b = R3.b and
  R1.c = R4.c and
  R4.d = R5.d



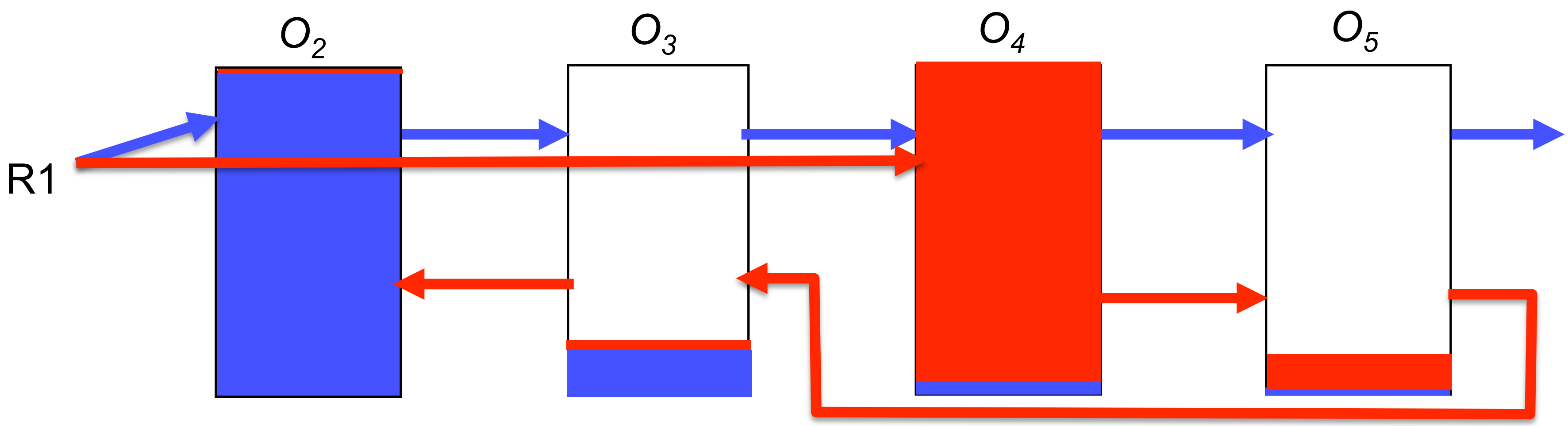


r_i = rate limit of operator O_i
= Number of tuples it can process per unit time
(also called *capacity*)

Can be computed using c_i

Interleaving Plans

- An interleaving plan defined by:
  - A set of permutations of the operators
  - A weight w_i for each permutation ($\Sigma\, w_i = 1$)



Interleaving plan: O2 → O3 →O4 → O5, *w = 0.5*

O4 → O5 →O3 → O2, *w = 0.5*

Problem Definition

- Given:
  - n selection operators $O_1, \ldots, O_n$
  - selectivity p_i and rate r_i for each operator O_i,
  - a precedence graph G over the operators

  Find the optimal interleaving plan that maximizes the tuple throughput *(and hence <u>total completion time)</u>*

- Definition: O_i is called selective if $p_i < 1$

- We assume tree-structured precedence constraints (correspond to queries with no cycles)

Outline

- Introduction and Motivation
- Problem Definition
- Algorithms for finding Optimal Interleaving Plans
  - Selective operators
  - Non-selective operators
- Experimental Results

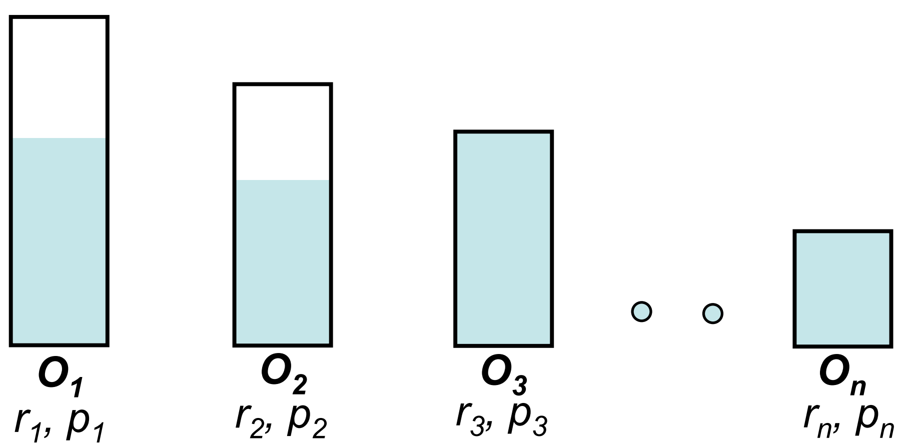
Overview of Approach

- View an interleaving plan as a collection of tuple flows
- Definition: An operator is *saturated* if it is processing at its rate limit
- Lemma: Saturation → Optimality
  - If all operators are saturated, we have an optimal solution
- Algorithm for when G is a forest of chains
- Recursively reduce the general case to forests of chains
- Combine the solutions for sub-problems

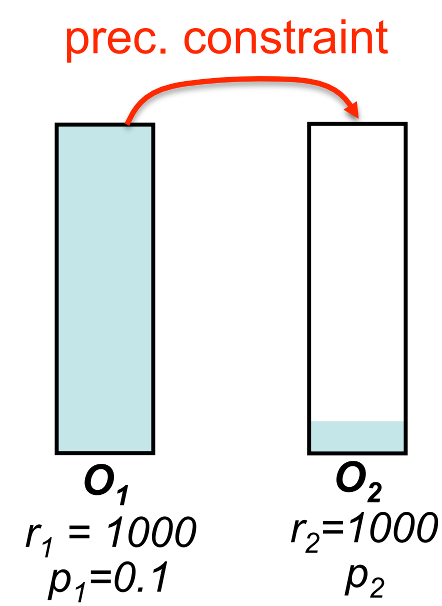
Saturated Suffix Lemma [CDHW'06]

- Given an interleaving plan, IF:
  - A set of operators is saturated (processing at their rate limit), and
  - No flow from a saturated operator to an unsaturated operator

  THEN the plan is optimal.
- The actual permutations used irrelevant
- However not *necessary* when there are precedence constraints



Steady state with saturated suffix



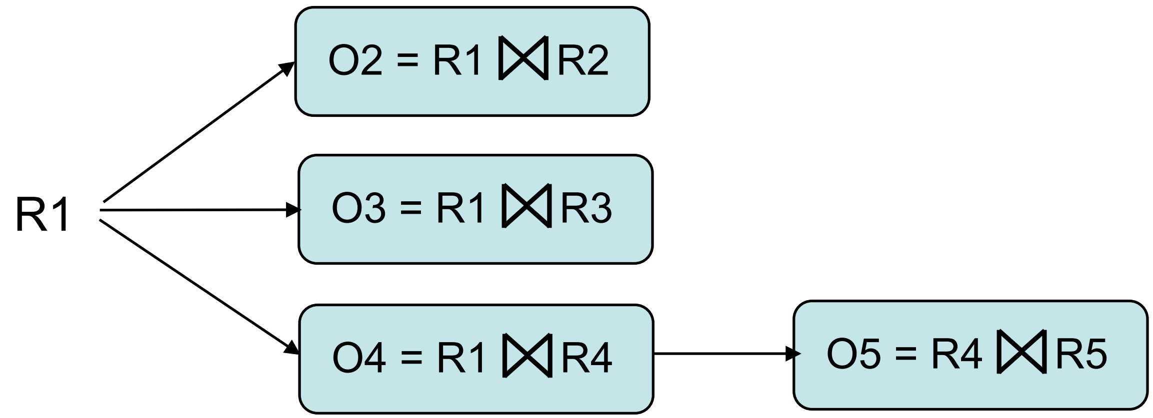
Overview of Approach

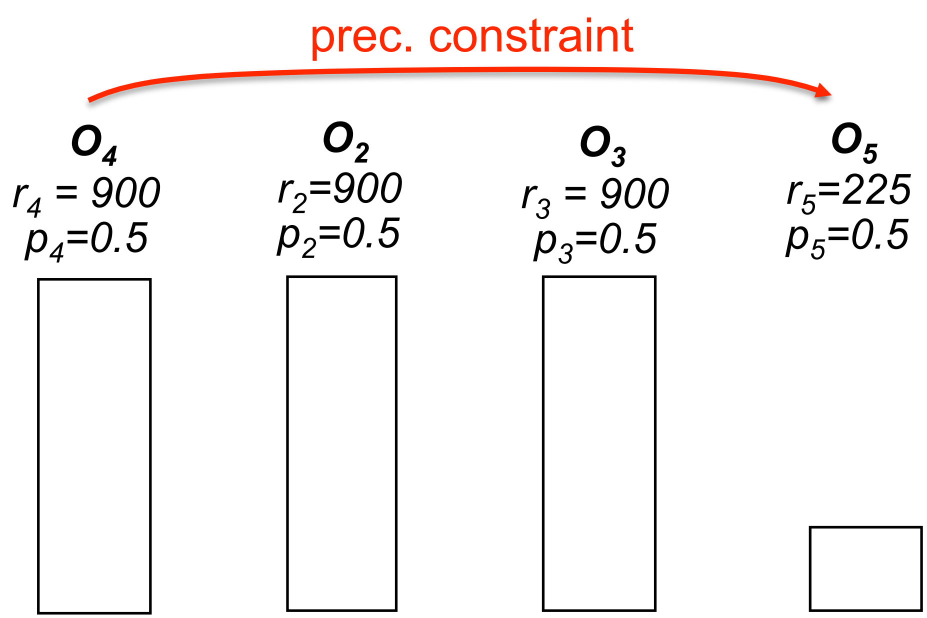
- View an interleaving plan as a collection of tuple *flows*
- Definition: An operator is *saturated* if it is processing at its rate limit
- Lemma: Saturation → Optimality
  - If all operators are saturated, we have an optimal solution
- Algorithm for when *G* is a *forest of chains*
- Recursively reduce the general case to *forests of chains*
- Combine the solutions for sub-problems

Algorithm: *G = forest of chains*

Example query

select *
from R1, R2, R3, R4, R5
where R1.a = R2.a and
R1.b = R3.b and
R1.c = R4.c and
R4.d = R5.d



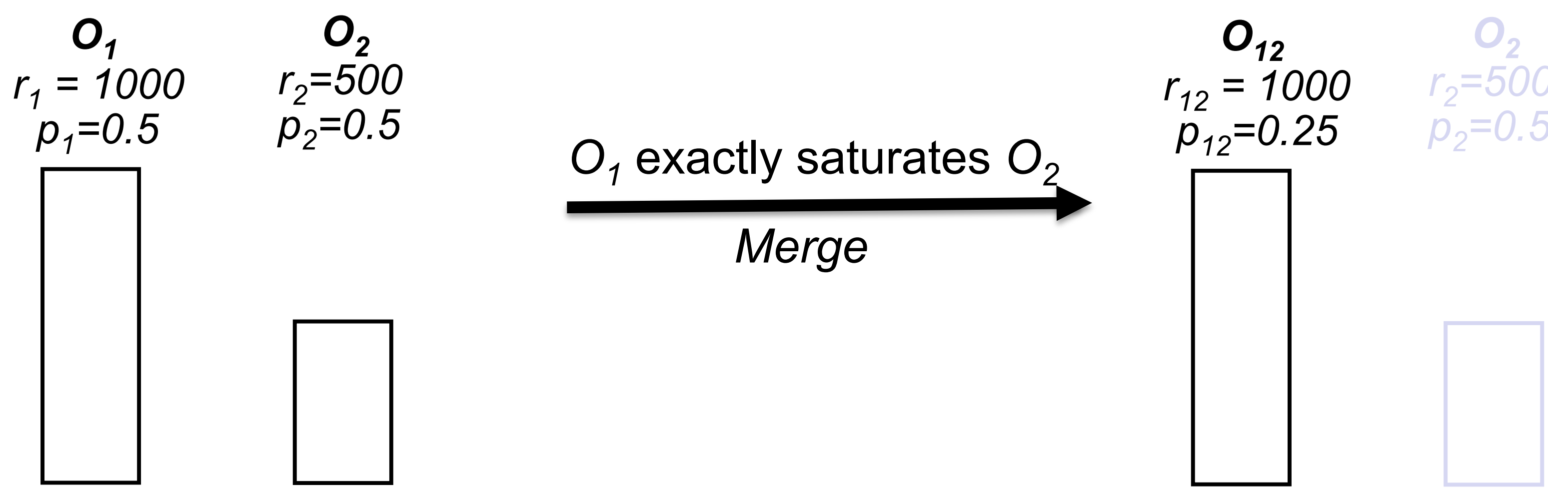


Algorithm: *G = forest of chains*

- Sort in non-increasing order by rate
- Start adding flow from left-to-right till:
  - Cond 1: A parent can exactly saturate a child (merge and recurse)
  - Cond 2: A node can exactly saturate its *predecessor* (merge and recurse)
  - Cond 3: No more flow can be added (→ saturation → optimality)

- Merging two operators:

  O_2 is merged into O_1 → O_2 is applied immediately after O_1



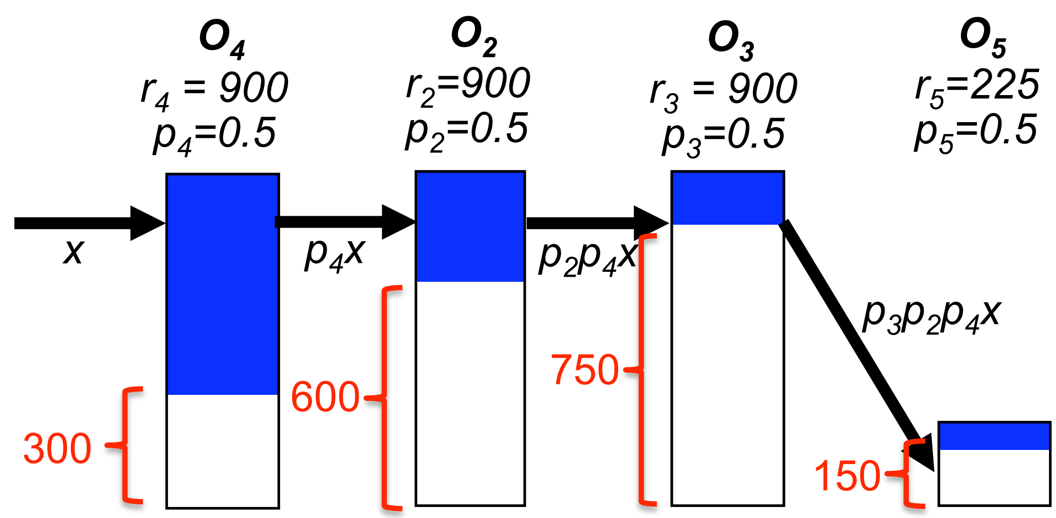
Algorithm: *G = forest of chains*

- Sort in non-increasing order by rate
- Start adding flow from left-to-right till:
  - Cond 1: A parent can exactly saturate a child (merge and recurse)
  - Cond 2: A node can exactly saturate its *predecessor* (merge and recurse)
  - Cond 3: No more flow can be added (→ saturation → optimality)

Step 1:
Send 600 units along
O4 → O2 →O3 → O5

Cond 1 satisfied for O4 and O5
Merge O5 into O4

Cond 2 satisfied for O2 and O4
Merge O4 into O2

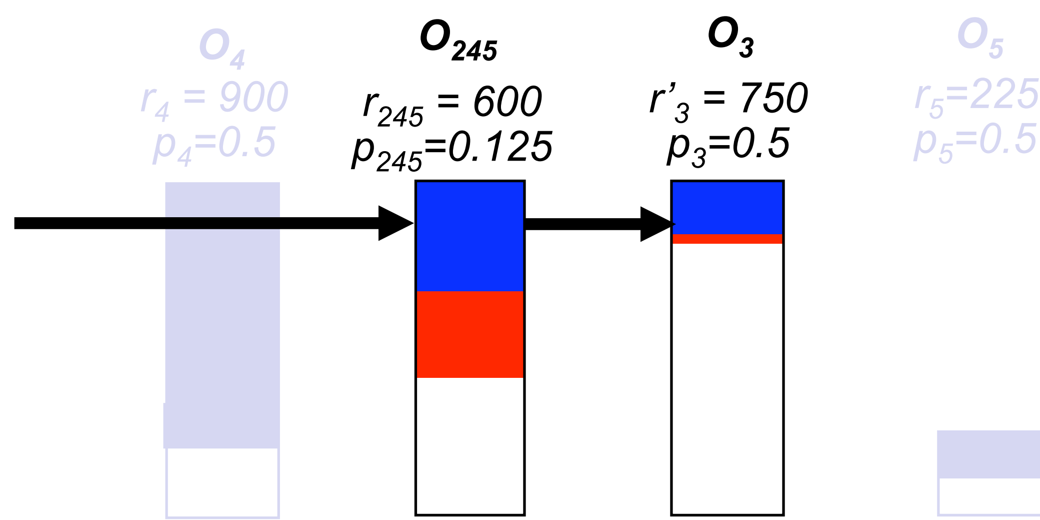


Algorithm: *G = forest of chains*

- Sort in non-increasing order by rate
- Start adding flow from left-to-right till:
  - Cond 1: A parent can exactly saturate a child (merge and recurse)
  - Cond 2: A node can exactly saturate its *predecessor* (merge and recurse)
  - Cond 3: No more flow can be added (→ saturation → optimality)

Step 2:
Send 240 units along
O2 → O4 →O5 → O3

Cond 2 satisfied for O3 and O245
Merge O245 into O3



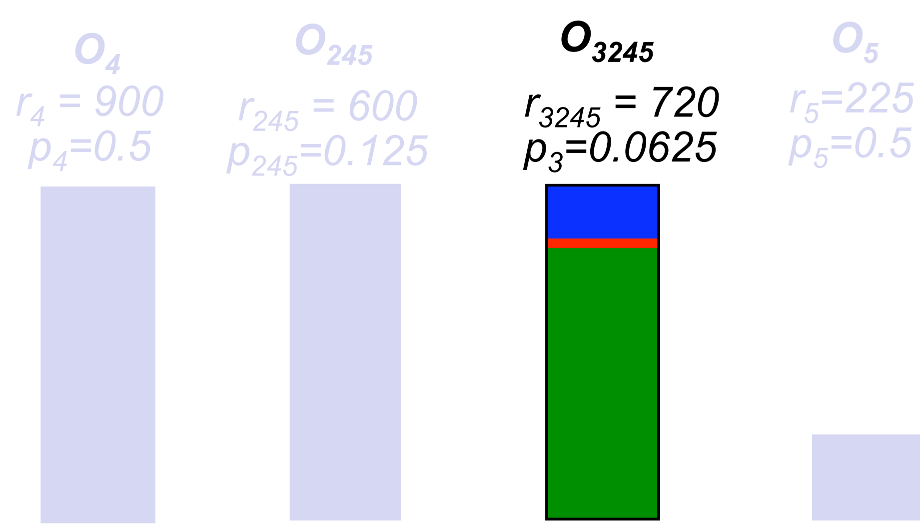
Algorithm: *G = forest of chains*

- Sort in non-increasing order by rate
- Start adding flow from left-to-right till:
  - Cond 1: A parent can exactly saturate a child (merge and recurse)
  - Cond 2: A node can exactly saturate its *predecessor* (merge and recurse)
  - Cond 3: No more flow can be added (→ saturation → optimality)

Step 3:
Send 720 units along
O3 → O2 →O4 → O5

Cond 3 satisfied for O3245

All operators are saturated
→ Optimality



Algorithm: *G = forest of chains*

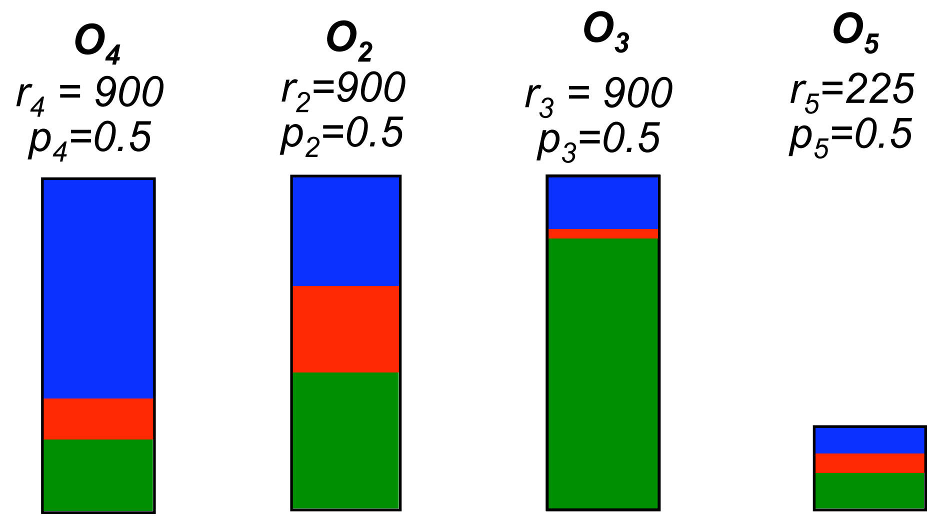
- Sort in non-increasing order by rate
- Start adding flow from left-to-right till:
  - Cond 1: A parent can exactly saturate a child (merge and recurse)
  - Cond 2: A node can exactly saturate its *predecessor* (merge and recurse)
  - Cond 3: No more flow can be added (→ saturation → optimality)

Final Interleaving Plan:

O4 → O2 → O3 → O5, $w = \frac{600}{1560}$

O2 → O4 → O5 → O3, $w = \frac{240}{1560}$

O3 → O2 → O4 → O5, $w = \frac{720}{1560}$



Algorithm: *G = forest of chains*

- Sort in non-increasing order by rate
- Start adding flow from left-to-right till:
  - Cond 1: A parent can exactly saturate a child (merge and recurse)
  - Cond 2: A node can exactly saturate its *predecessor* (merge and recurse)
  - Cond 3: No more flow can be added ($\rightarrow$ saturation $\rightarrow$ optimality)

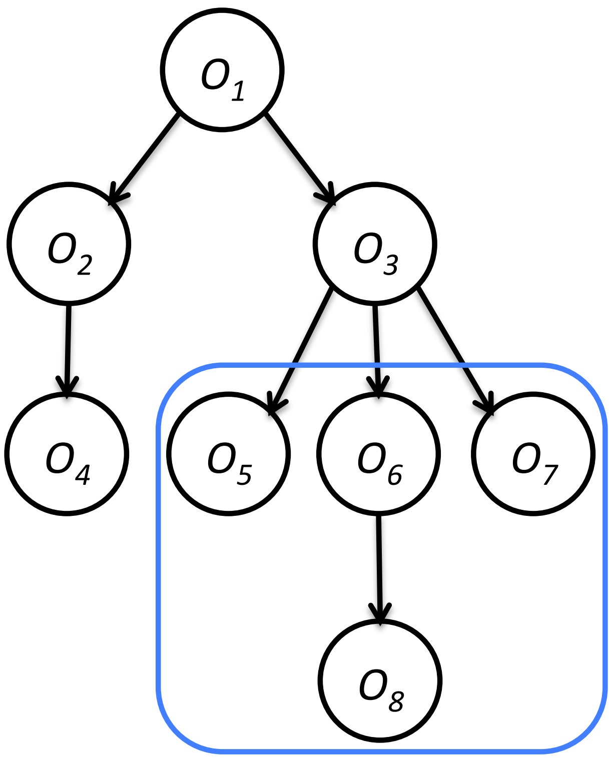
- Theorem: The algorithm runs in $O(n^2 \log n)$ time and finds an interleaving plan with at most $4n - 3$ distinct permutations.

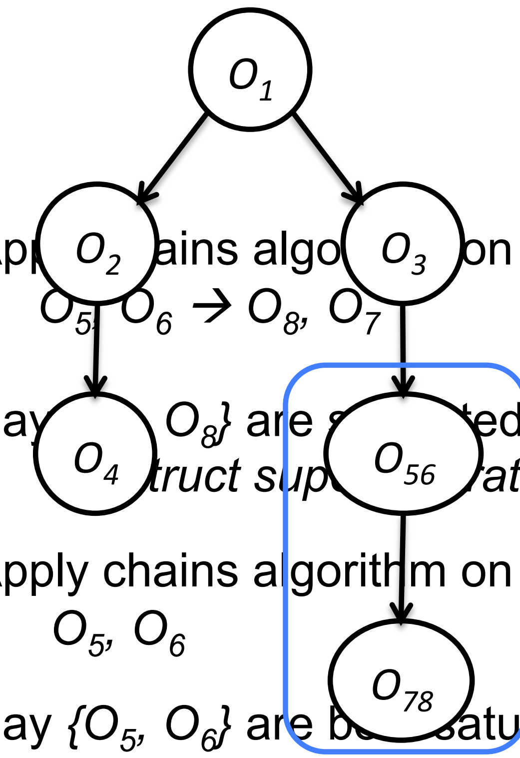
Overview of Approach

- View an interleaving plan as a collection of tuple *flows*
- Definition: An operator is *saturated* if it is processing at its rate limit
- Lemma: Saturation → Optimality
  - If all operators are saturated, we have an optimal solution
- Algorithm for when *G* is a *forest of chains*
- Recursively reduce the general case to *forests of chains*
- Combine the solutions for sub-problems

General Case

Eliminate one *fork* at a time using the *Chains* algorithm





Apply chains algorithm on chains
$O_5, O_6 \rightarrow O_8, O_7$

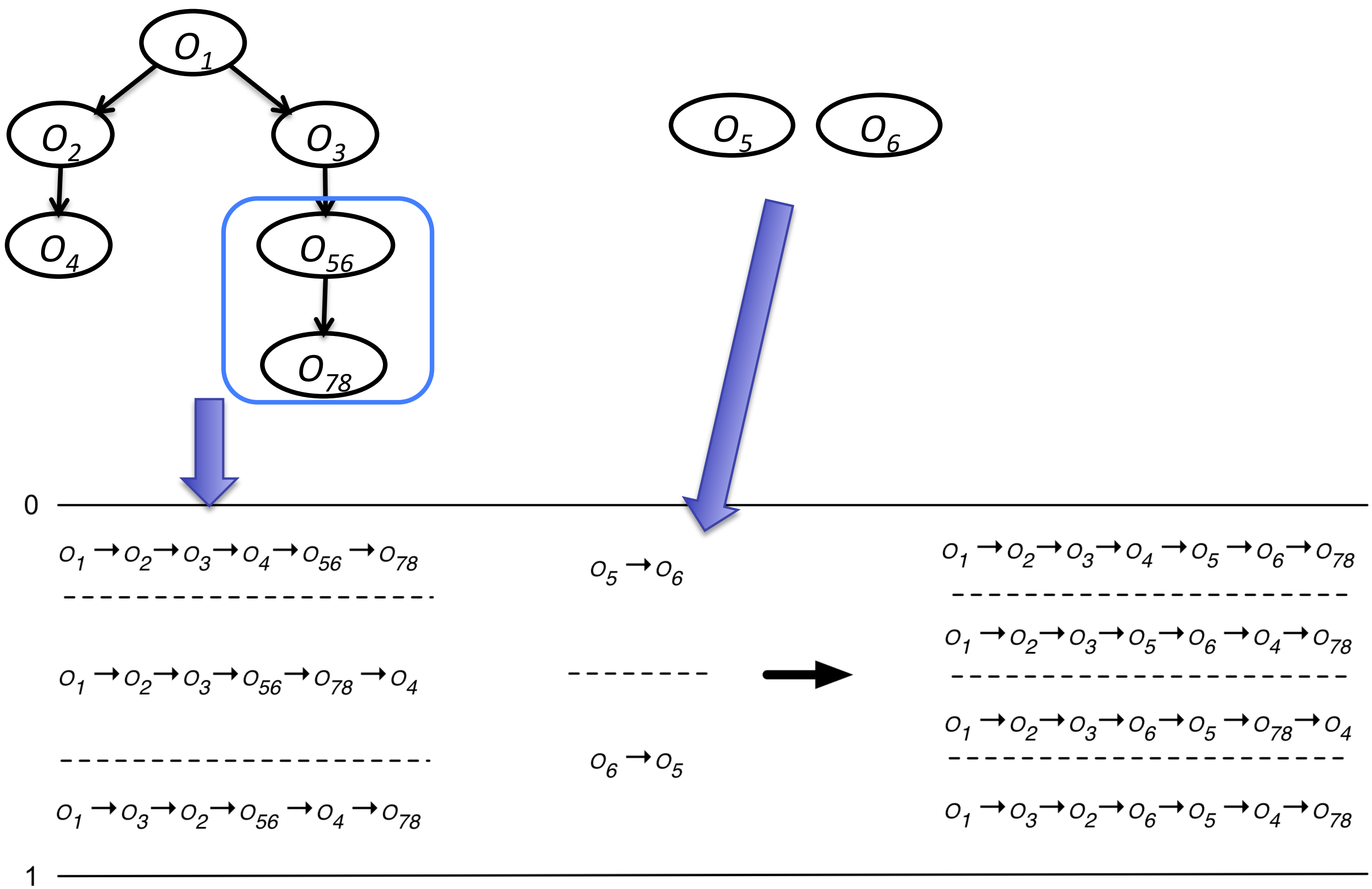
Say $\{O_7, O_8\}$ are saturated
Construct super-operator O_{78}

Apply chains algorithm on chains
O_5, O_6

Say $\{O_5, O_6\}$ are both saturated
Construct super-operator O_{56}

General Case

Combine the solutions found for the sub-problems and the recursive problem



General Case

- Theorem: The algorithm runs in $O(n^3)$ time and finds an interleaving plan with at most $4n$ distinct permutations.

Outline

- Introduction and Motivation
- Problem Definition
- Algorithms for finding Optimal Interleaving Plans
  - Selective operators
  - Non-selective operators
- Experimental Results

Non-selective Operators

- The saturated suffix lemma does not hold:
  - *Saturation does not imply optimality*

- Summary of results:
  - All non-selective operators and tree-structured precedence constraints
    - Can be solved using the same algorithm
  - Mixture of selective and non-selective operators
    - $O(n^2 \log n)$ algorithm for when G is a forest of chains
  - General case still open
    - Known to be polynomial

Outline

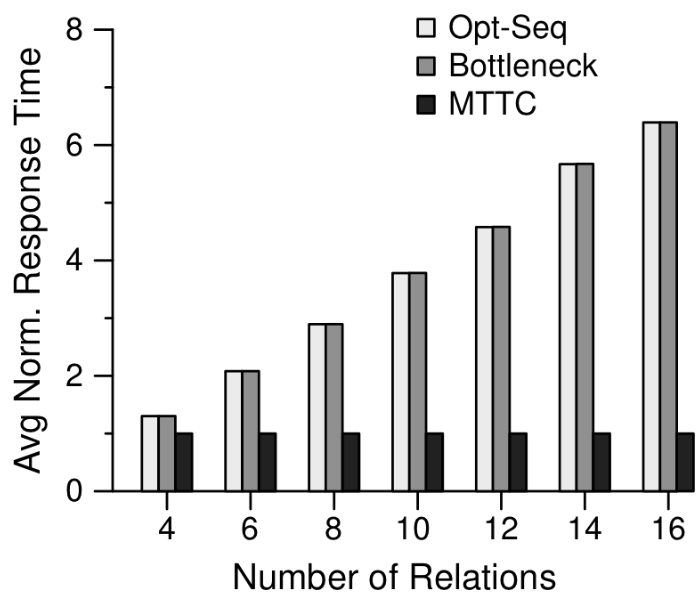
- Introduction and Motivation
- Problem Definition
- Algorithms for finding Optimal Interleaving Plans
  - Selective operators
  - Non-selective operators
- Experimental Results

Performance Study

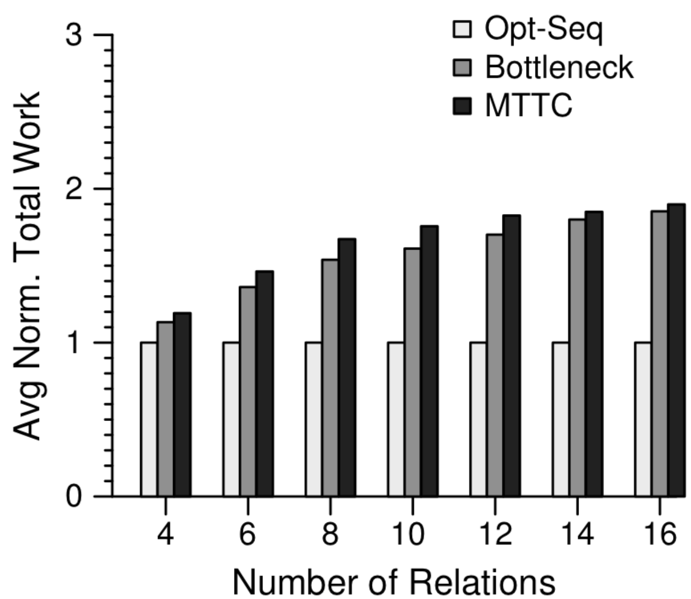
- Compared Techniques:
  - OPT-SEQ: Serial plan found using *rank ordering*
    - Optimal for centralized case
  - BOTTLENECK [Srivastava et al.; 2006]
    - Optimal serial plan for parallel execution
  - MTTC: Proposed algorithm
- Setup:
  - Synthetic datasets: *costs* and *selectivities* chosen randomly
  - Different query types: *star, path, random*
- Comparison metrics:
  - *Response time* (total time to execute the query)
  - *Total work* (across all processors)

Performance Study

- Star queries (1)



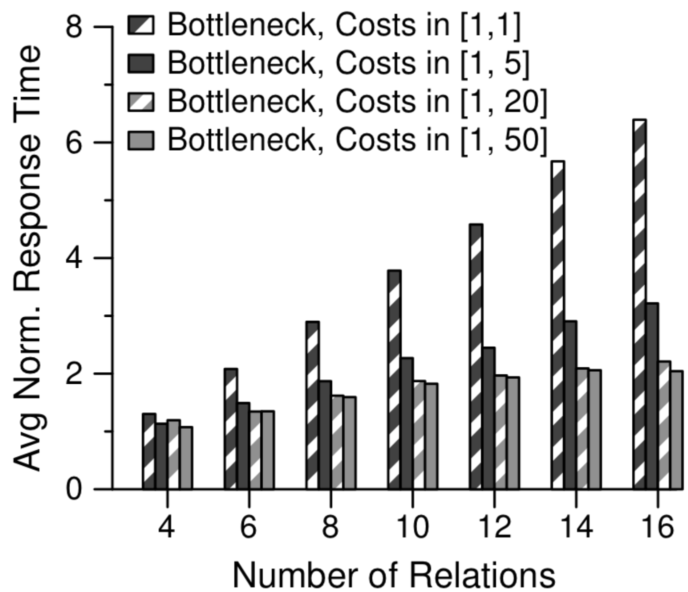
(i) Sel in [0, 1], Costs in [1, 1]



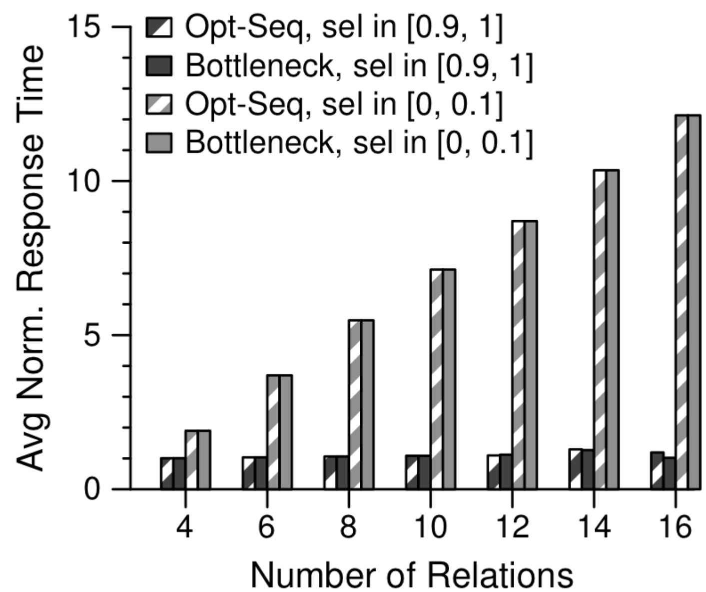
(ii) Sel in [0, 1], Costs in [1, 1]

Performance Study

- Star queries (2)



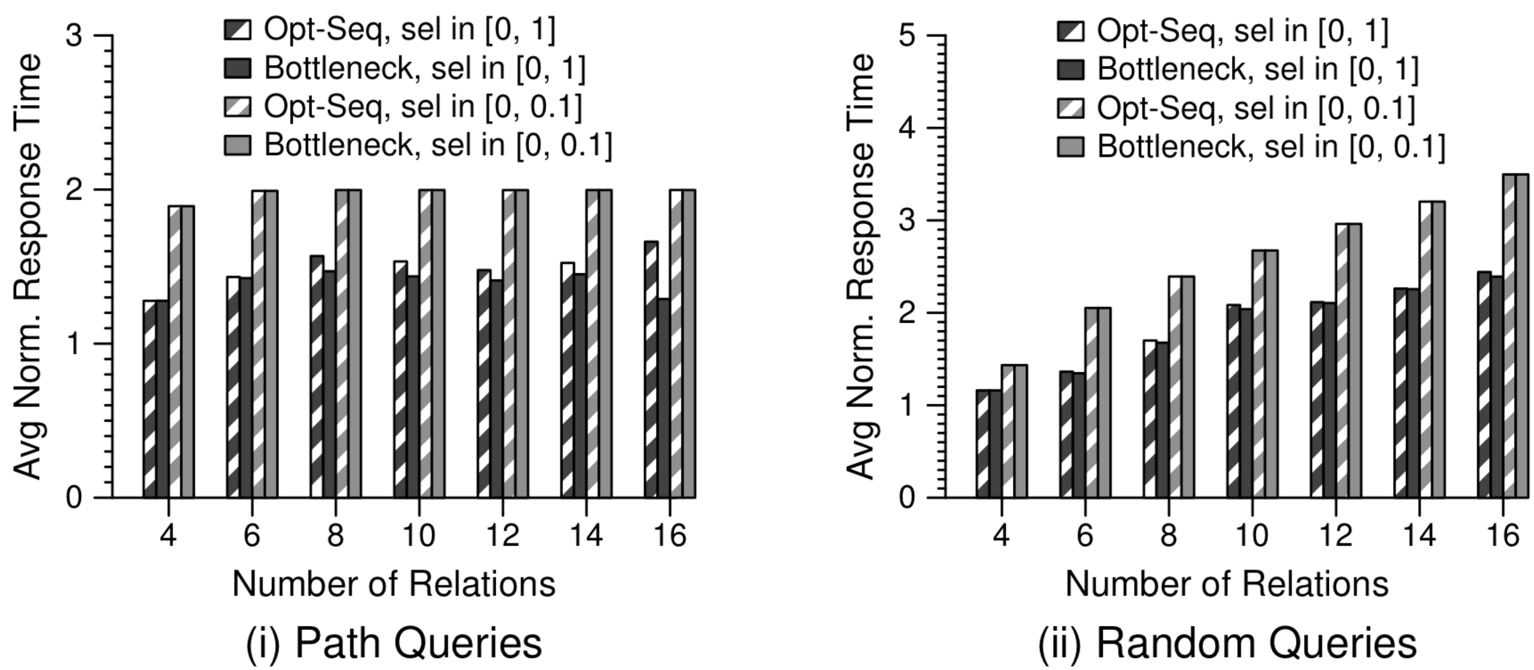
(i) Sel in [0, 1]



(ii) Costs in [1, 1]

Performance Study

- Path queries, randomly generated query graphs



Conclusions and Future Work

- Proposed *interleaving plans* to fully exploit parallelism in a database system
- Fast algorithms for finding optimal *interleaving plans*
  - Use few permutations, so easy to deploy

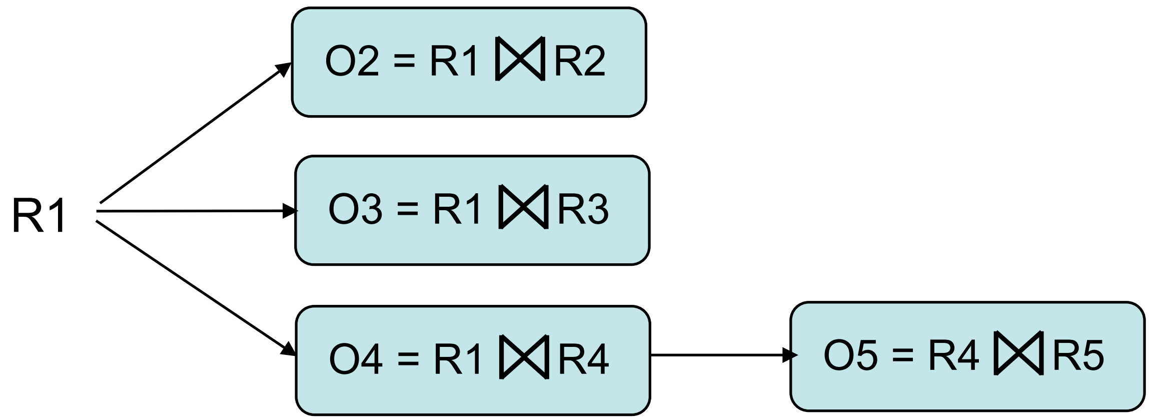
- Open questions:
  - Cyclic precedence constraints
  - Correlated predicates
  - Other types of queries (Bushy plans, MJoins)

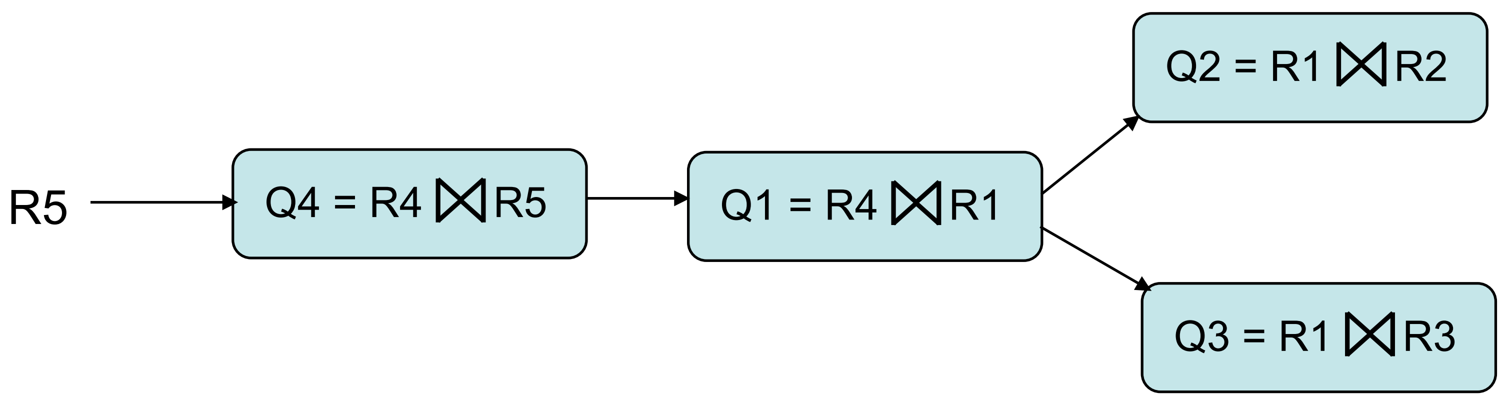
- Thank you !!

Reduction

- Given a multi-way join query and a driver relation, choosing a left-deep plan is equivalent to precedence-constrained selection ordering

Example query
select *
from R1, R2, R3, R4, R5
where R1.a = R2.a and
R1.b = R3.b and
R1.c = R4.c and
R4.d = R5.d



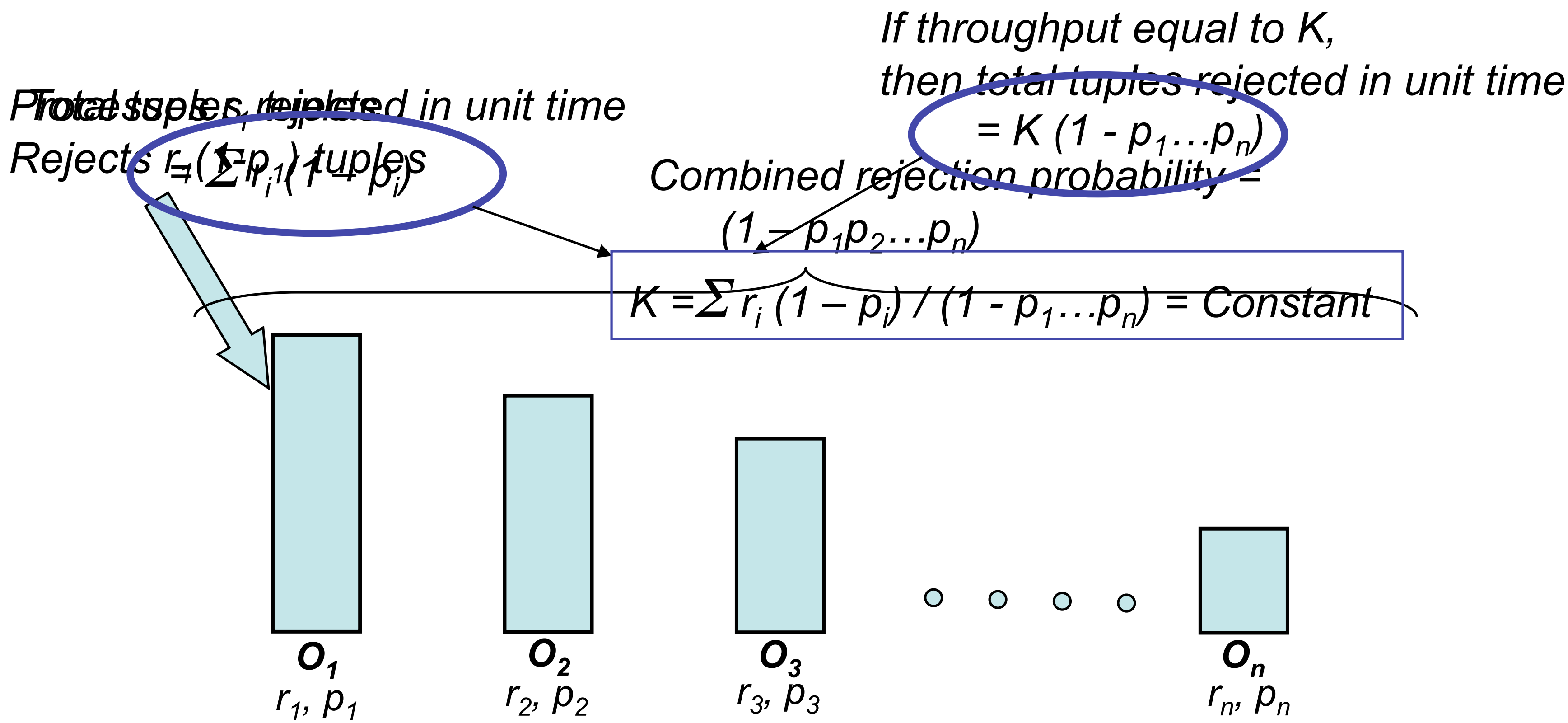


Algorithms for finding serial plans

- Rank ordering: Order the operators in the increasing order of rank = ci/(1 – pi)
  - Optimal in a centralized scenario
  - Oblivious to the parallelization
- Bottleneck [Srivastava et al.; 2006]: Order the operators in the decreasing order of ri
  - Finds the optimal serial plan in the parallel setting for selective operators
  - Different plan space considered for non-selective operators

Saturation → Optimality

- Case: Full Saturation
  - All operators are processing at their rate limit
  - Precedence constraints irrelevant
- Proof:



Saturated steady state – Throughput *K*