THE UNIVERSITY of EDINBURGH

Advanced Databases

Spring 2020

Lecture #11:

Query Optimisation II

Milos Nikolic

Finding the "Best" Query Plan

Holy grail of any DBMS implementation

Challenge: There may be more than one way to answer a given query

- Which one of the join operators should we pick?
- With which parameters (block size, buffer allocation, ...)?
- Which join ordering?

Finding the "Best" Query Plan

The query optimiser

1. **Enumerates** all possible query execution plans

   If this yields too many plans, at least enumerate the "promising" plan candidates

2. Determines the **cost** (quality) of each plan
3. Chooses the **best** one as the final execution plan

Ideally: Want to find the best plan. **Practically**: Avoid worst plans!

Join Optimisation: Overview

We have translated the query into a graph of query blocks

Query blocks are essentially a multi-way product of relations with projections on top

We can estimate the cost of a given execution plan

Use result size estimates in combination with the cost for individual join algorithms

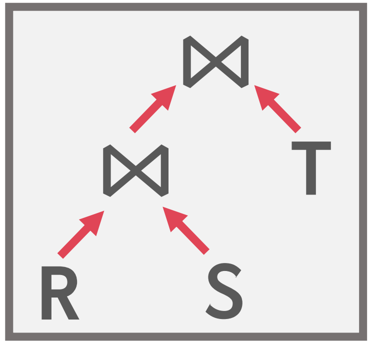
Task: enumerate all possible execution plans

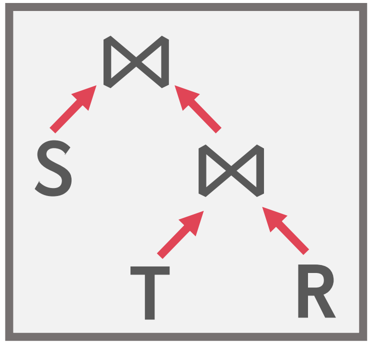
i.e., all possible 2-way join combinations for each query block

Example: three-way join

12 possible re-orderings

2 shown here





Enormous Search Space

# of relations n	# of different join trees
2	2
3	12
4	120
5	1,680
6	30,240
7	665,280
8	17,297,280
10	17,643,225,600

We have not even considered *different join algorithms*!

We need to restrict search space!

Join Trees: Dynamic Programming

Use **dynamic programming** to reduce the number of cost estimations

Find the cheapest plan for an *n*-way join ($R_1 \bowtie \ldots \bowtie R_n$) in ***n* passes**

- In pass *k*, find the best plans for all *k*-relation sub-queries
- Construct plans in pass *k* from best *i*-relation and *(k-i)*-relation subplans ($1 \leq i < k$)

Assumption: **"Principle of optimality"**

- To find optimal global plan suffices to only consider optimal plans of sub-queries
- Reduces considerably the search space, yet may lead to suboptimal plans

EXAMPLE: DYNAMIC PROGRAMMING

```
SELECT * FROM R, S, T
 WHERE R.A = S.A
   AND S.B = T.B
```

Pass #1 (best 1-relation plans): Find best access path to each relation (e.g., index, full table scans)

R
S
T

R ⋈ S
T

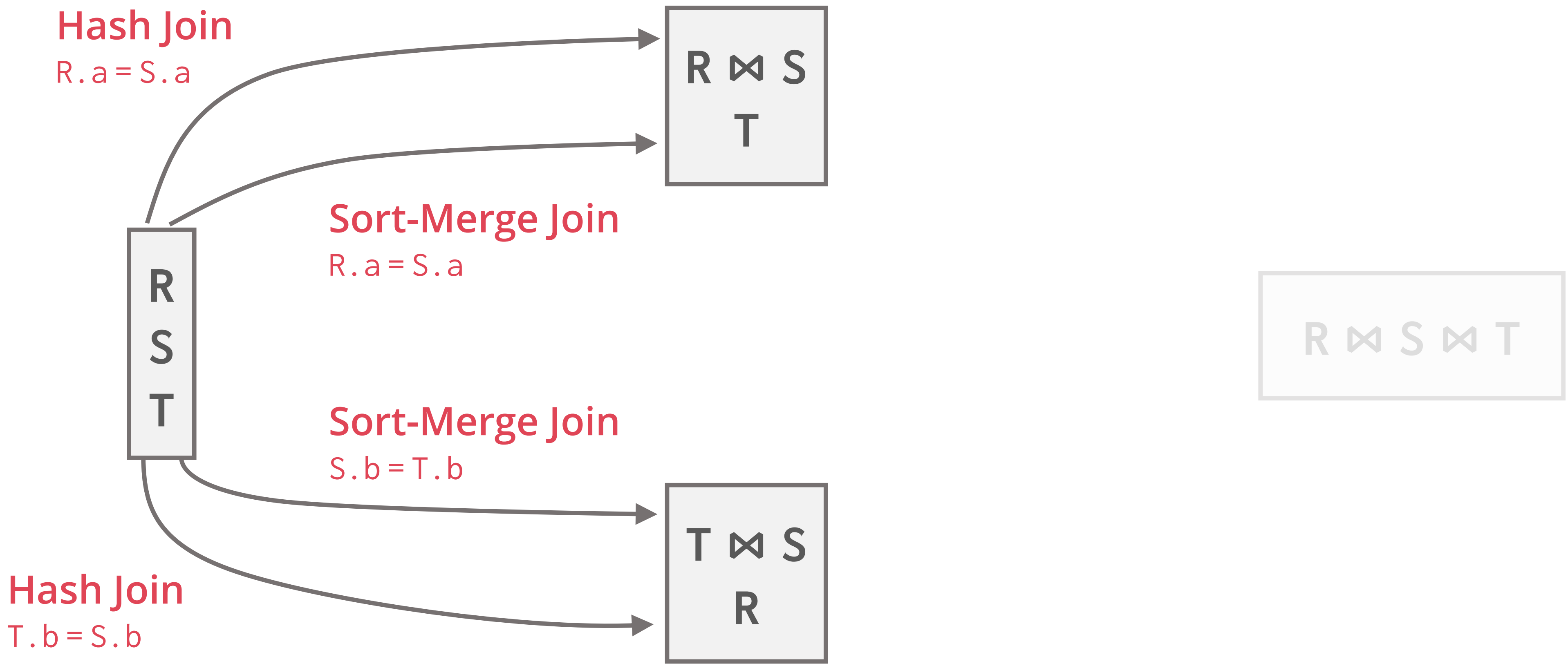
T ⋈ S
R

R ⋈ S ⋈ T

EXAMPLE: DYNAMIC PROGRAMMING

Pass #2 (best 2-relation plans): determine best join order (R ⋈ S or S ⋈ R), choose best candidate

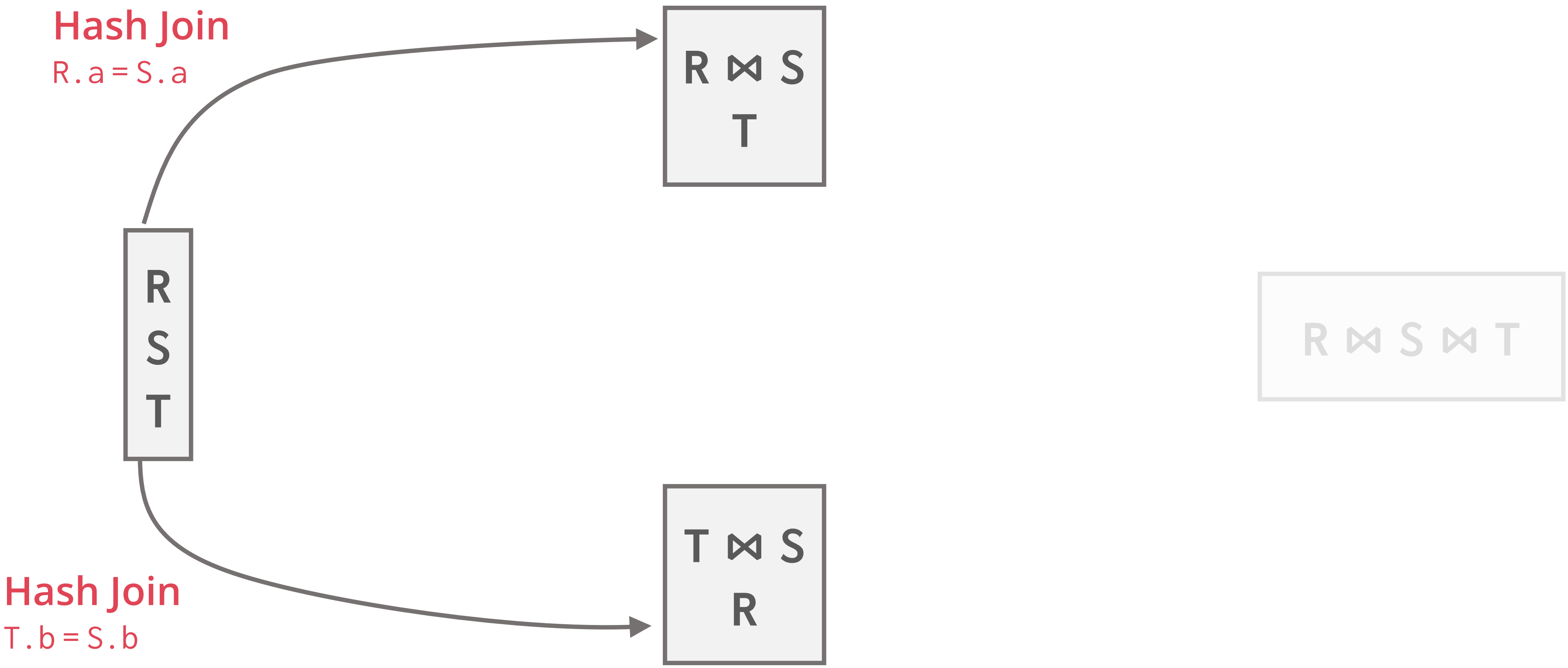
```
SELECT * FROM R, S, T
 WHERE R.A = S.A
   AND S.B = T.B
```



Example: Dynamic Programming

Pass #2 (best 2-relation plans): determine best join order (R ⋈ S or S ⋈ R), choose best candidate

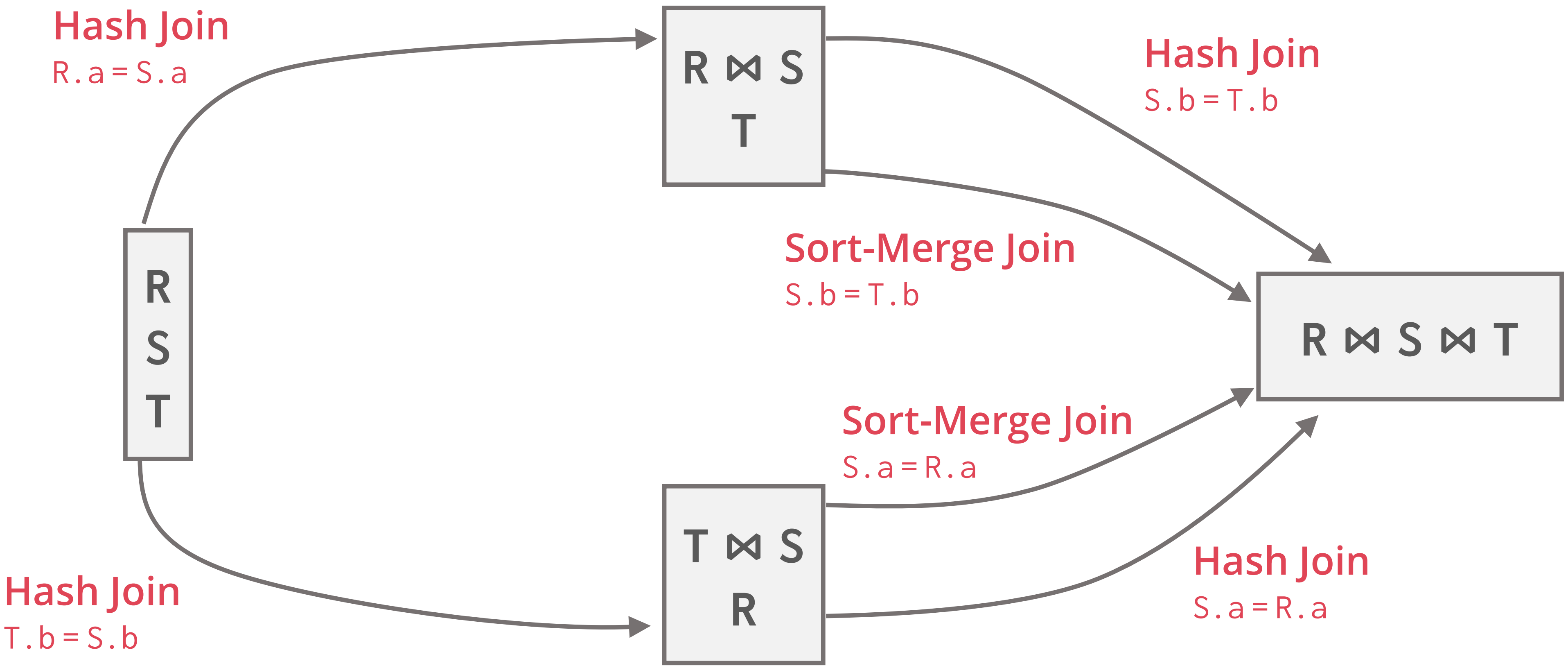
```
SELECT * FROM R, S, T
 WHERE R.A = S.A
   AND S.B = T.B
```



Example: Dynamic Programming

Pass #3 (best 3-relation plans):
best 2-relation plans + one other relation

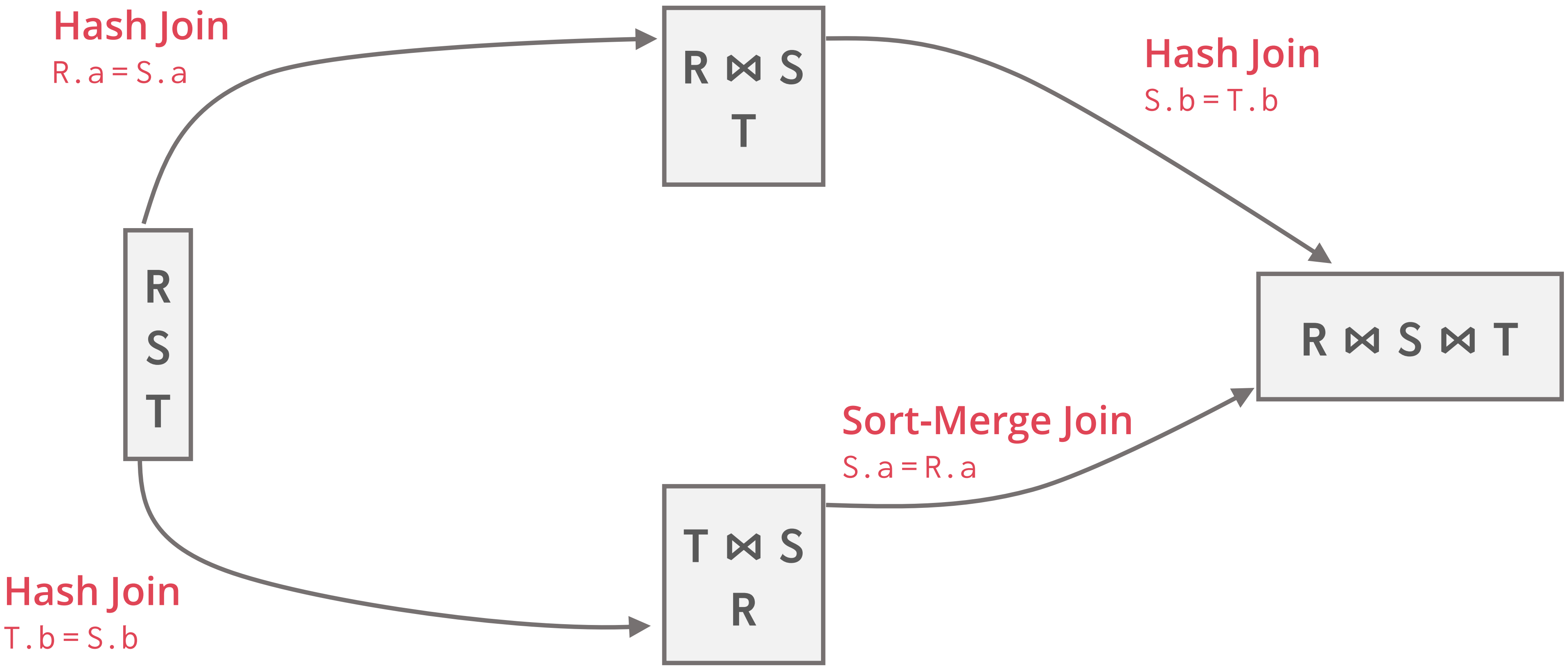
```
SELECT * FROM R, S, T
 WHERE R.A = S.A
   AND S.B = T.B
```



EXAMPLE: DYNAMIC PROGRAMMING

Pass #3 (best 3-relation plans):
best 2-relation plans + one other relation

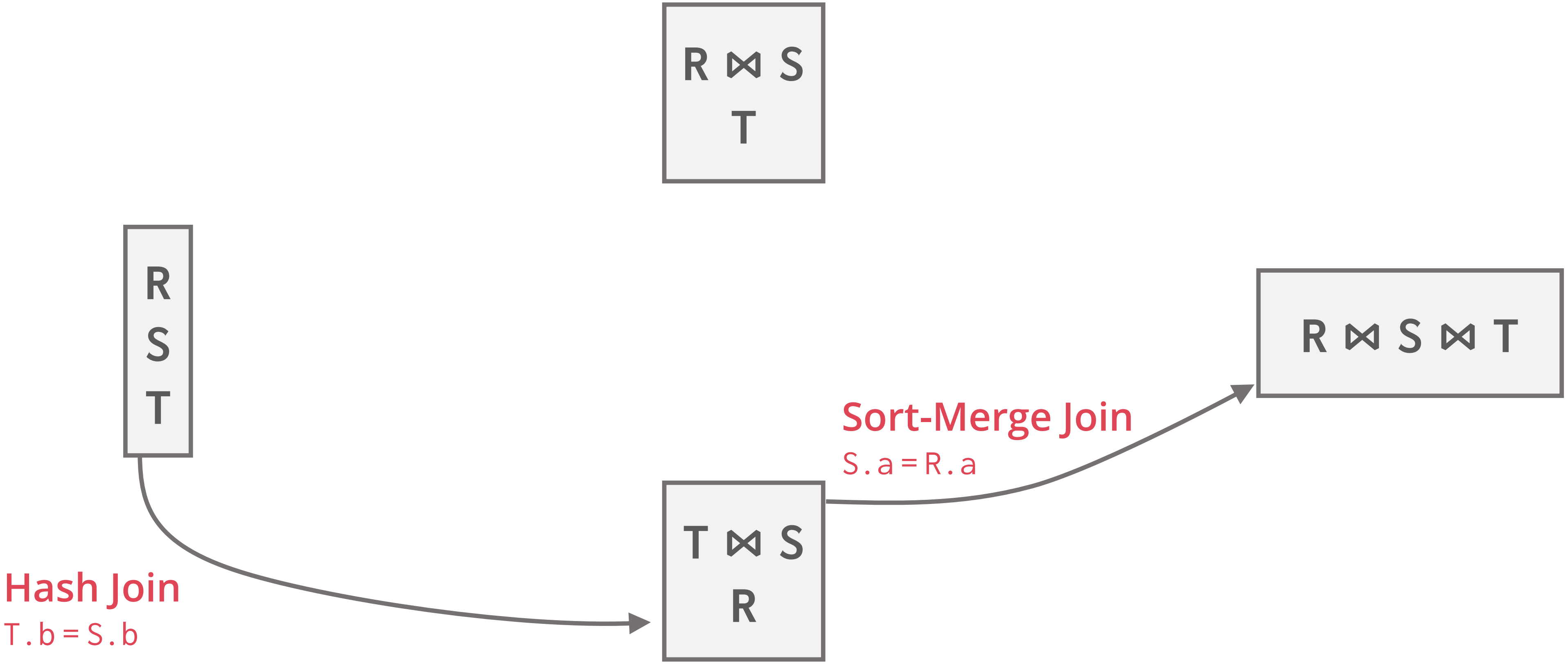
```
SELECT * FROM R, S, T
 WHERE R.A = S.A
   AND S.B = T.B
```



Example: Dynamic Programming

Pass #3 (best 3-relation plans):
best 2-relation plans + one other relation

```
SELECT * FROM R, S, T
 WHERE R.A = S.A
   AND S.B = T.B
```



Dynamic Programming Algorithm

```
Function find_join_tree_dp(q(R_1, ..., R_n))

for i = 1 to n do
  optPlan[{R_i}] = access_plans(R_i)
  prune_plans(optPlan[{R_i}])
for i = 2 to n do
  foreach S ⊆ {R_1, ..., R_n} such that |S| = i do
    optPlan[S] = ∅
    foreach T ⊂ S with T ≠ ∅ do
      optPlan[S] = optPlan[S] ∪ possible_joins(optPlan[T], optPlan[S\T])
    prune_plans(optPlan[S])
return optPlan[{R_1, ..., R_n}]
```

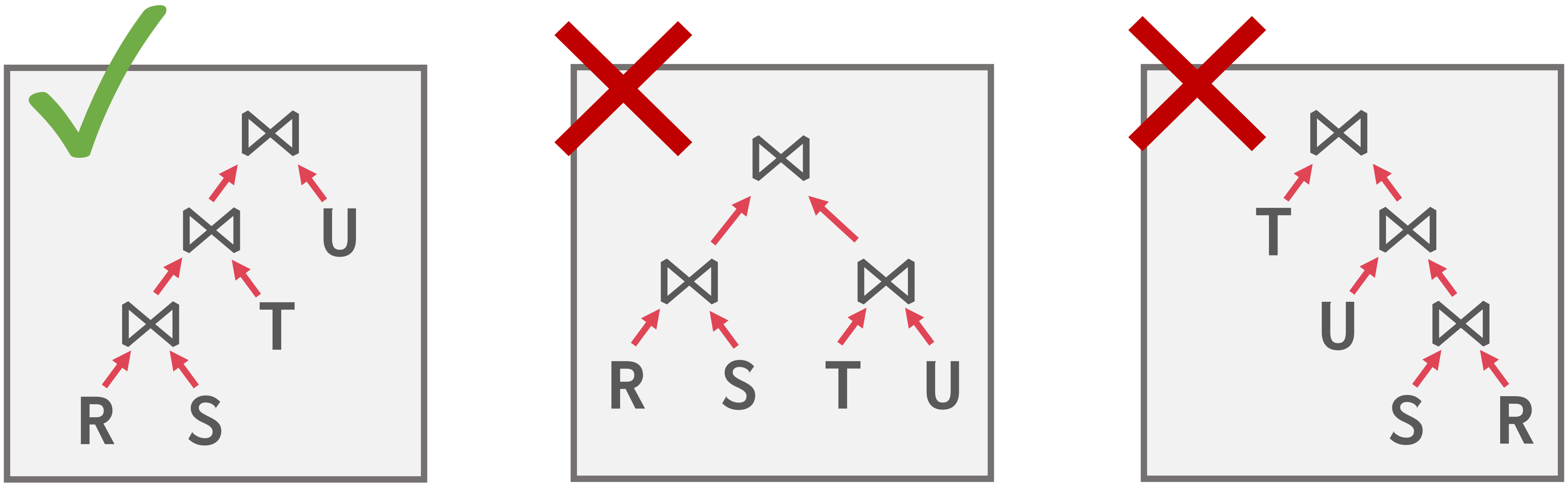
`possible_joins(`R, S`)` enumerates the possible joins between R and S (e.g., NL join, SM join)

`prune_plans(`*set*`)` discards all but the best plan from *set*

Join Tree Shapes

Fundamental decision in IBM's **System R** (late 1970):

Only consider left-deep join trees



left-deep

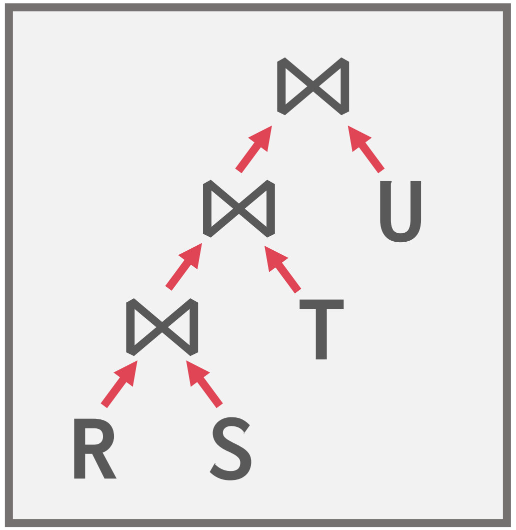
bushy
(everything else)

right-deep

LEFT-DEEP JOIN TREES

DBMSs often prefer left-deep join trees

- The inner (rhs) relation always is a base relation
- Allows the use of index nested loops join
- Allows for **fully pipelined plans** where intermediate results are not written to temporary files
  - Should be factored into global cost calculation
  - Not all left-deep trees are fully pipelined (e.g., sort-merge join)
  - Pipelining requires **non-blocking** operators



Modern DBMSs may also consider non left-deep join trees

Multi-Relation Query Planning

- **System R**-style join order enumeration
  - Left-deep tree #1, Left-deep tree #2…
  - Eliminate plans with cross products immediately
- Enumerate the plans for each operator
  - Hash, Sort-Merge, Nested Loop…
- Enumerate the access paths for each table
  - Index #1, Index #2, Sequential scan…

Interesting Orders

System R-style query optimisers also consider **interesting orders**

- Sorting orders of the input tables that may be beneficial later in the query plan
  - e.g., for a sort-merge join, projection with duplicate removal, order-by clause
- Determined by ORDER BY and GROUP BY clauses in the input query or join attributes of subsequent joins (to facilitate merging)

For each subset of relations, retain only:

- Cheapest plan overall, plus
- Cheapest plan for each **interesting order** of the tuples

Example: Interesting Orders

Consider the join query:

```
SELECT * FROM Orders O, Lineitem L
 WHERE O.o_orderkey = L.l_orderkey
```

`Orders` has an unclustered index `OK_IDX` on column `o_orderkey`

Possible table access plans (1-relation plans) are:

`Orders`	**Full table scan** (arbitrary order). Cost: #pages(`Orders`)
	Index scan (sorted on `o_orderkey`). Cost = #pages(`OK_IDX`) + *k* * #pages(`Orders`), *k* is #index entries/page
`Lineitem`	**Full table scan**. Cost = #pages(`Lineitem`)

Example: Interesting Orders

The full table scan is the cheapest access method for both tables

Although more expensive than a full table scan, the use of the index (**order enforcement**) may pay off later on in the overall plan

- An index scan of `OK_IDX` yields the scan output in `o_orderkey` order
- This is beneficial for sort-merge join as we need to sort only `Lineitem`
- This could turn out to be the best 2-relation plan!

System R-style optimisers would keep both full table scan and index scan as best 1-relation plans for `Orders`

Conclusion

Query optimization is an important task in a relational DBMS

Explores a set of alternative plans

- Must prune search space; typically, left-deep plans only
- Uses dynamic programming for join orderings

Must estimate cost of each plan that is considered

- Must estimate the size of result and cost for each plan node

Query optimizer is the most complex part of database systems!