THE UNIVERSITY of EDINBURGH

Advanced Database Systems

Spring 2024

Lecture #23:

Query Optimisation: Searching

R&G: Chapter 15

Query Optimisation

- Plan space
- Cost estimation
- Search algorithm

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!

Enumeration of Alternative Plans

There are two main cases:

- Single-table plans (base case)
- Multiple-table plans (induction)

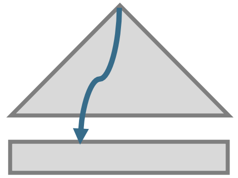
Single-table queries include selects, projects, and group-by / aggregate

- Consider each available access path (file scan vs. index)
- Choose the one with the least estimated cost

SINGLE-TABLE PLANS: COST ESTIMATES

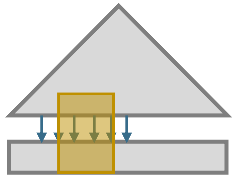
Index I on primary key matches selection:

Cost is **(Height(I) + 1) + 1** for a B+ tree *(variant **B** or **C**)*



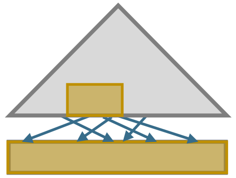
Clustered index I matching selection:

(NPages(I) + NPages(R)) * selectivity *(approximately)*



Non-clustered index I matching selection:

(NPages(I) + NTuples(R)) * selectivity *(approximately)*



Sequential scan of file

NPages(R)

Recall: Must also charge for duplicate elimination if required

Single-Table Plan: Example

```
SELECT * FROM Sailors
 WHERE rating = 8
```

NTuples(Sailors) = 40,000

NPages(Sailors) = 500

NKeys(rating) = 10

NPages(I) = 50

If we have an index **I** on *rating*:

Cardinality

= 1/ NKeys(rating) · NTuples(Sailors) = 1/10 · 40,000 = **4000 tuples**

Clustered index

1/ NKeys(rating) · (NPages(I) + NPages(Sailors)) = 1/10 · (50 + 500) = **55 pages** are retrieved

Unclustered index

1/ NKeys(rating) · (NPages(I) + NTuples(Sailors)) = 1/10 · (50 + 40,000) = **4005 pages** are retrieved

Costs on indexes are approximate as we might not need to retrieve all index pages

If we have an index **I** on *sid:*

Doing an index scan retrieves all pages & tuples

Clustered index: ~ (50 + 500) pages retrieved. Unclustered index: ~ (50 + 40,000) pages retrieved

Doing a file scan retrieves all file pages: 500

Multiple-Table Plans

We have translated the query into a graph of query blocks

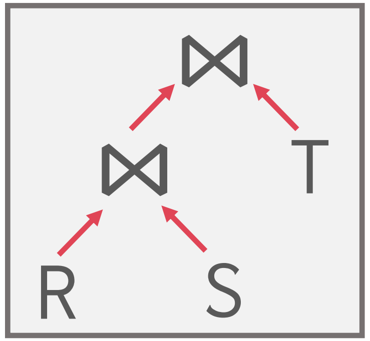
- Query blocks are essentially a multi-way product of relations with projections on top

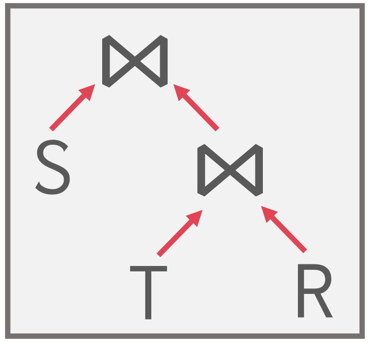
Task: enumerate all possible execution plans

- 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

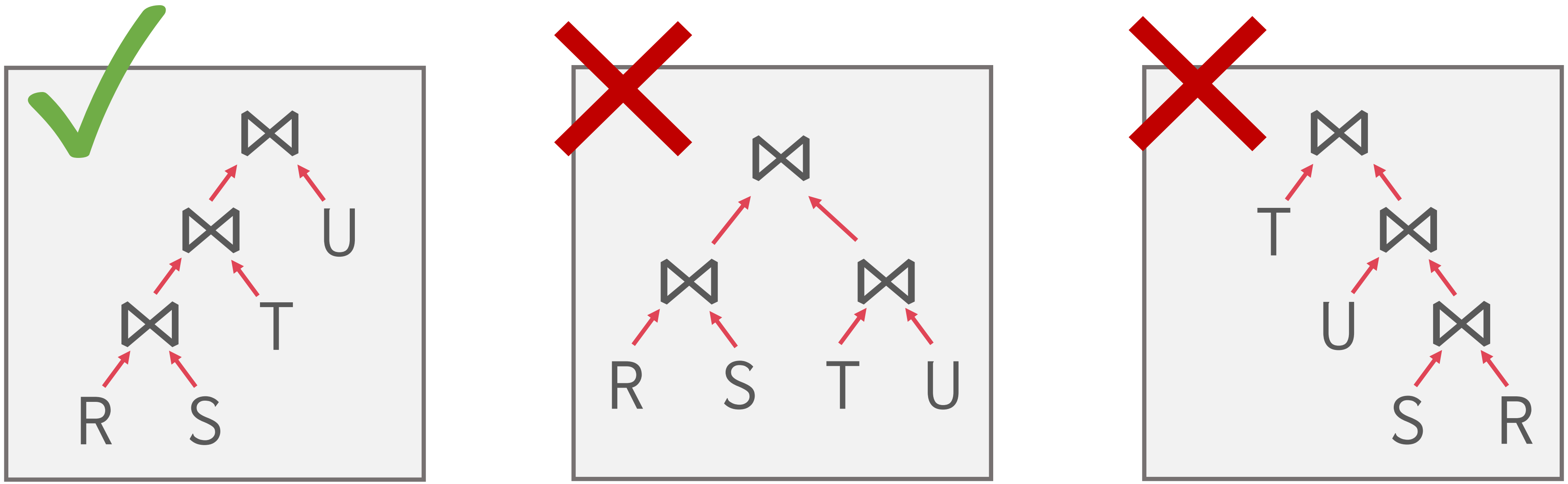
We have not even considered *different join algorithms*!

We need to restrict search space!

Multiple-Table Query Planning

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

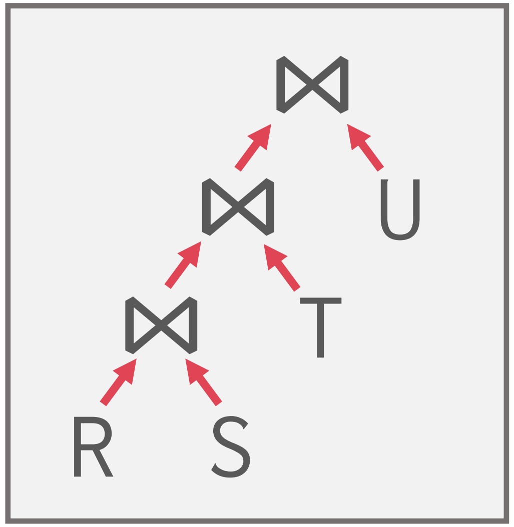
Only consider left-deep join trees



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

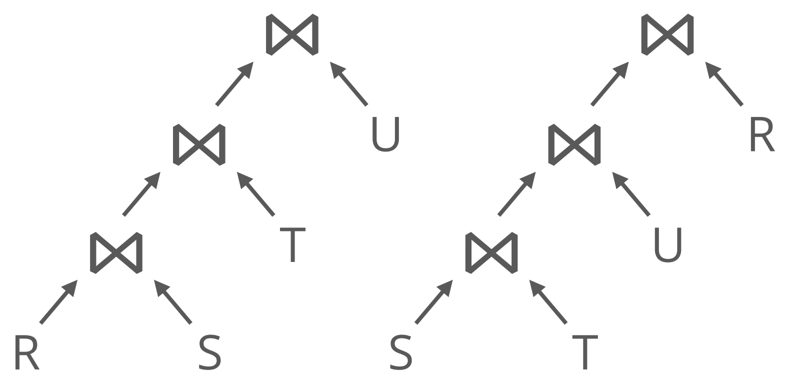


- Modern DBMSs may also consider non left-deep join trees

Multi-Table 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…

Use dynamic programming to reduce the number of cost estimations

The Principle of Optimality

- The best overall plan is composed of best decisions on the subplans
  - Optimal result has optimal substructure
- For example, the best left-deep plan to join tables R, S, T is either:
  - (The best plan for joining R, S) ⋈ T
  - (The best plan for joining R, T) ⋈ S
  - (The best plan for joining S, T) ⋈ R
- This is great!
  - When optimising a subplan (e.g., R ⋈ S), don’t worry how it will be used later (e.g., when joining with T)!
  - When optimizing a higher-level plan (e.g., R ⋈ S ⋈ T), reuse the best results of subplans (e.g., R ⋈ S)!

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 (index vs. full table scans)

R
S
T

R ⋈ S
T

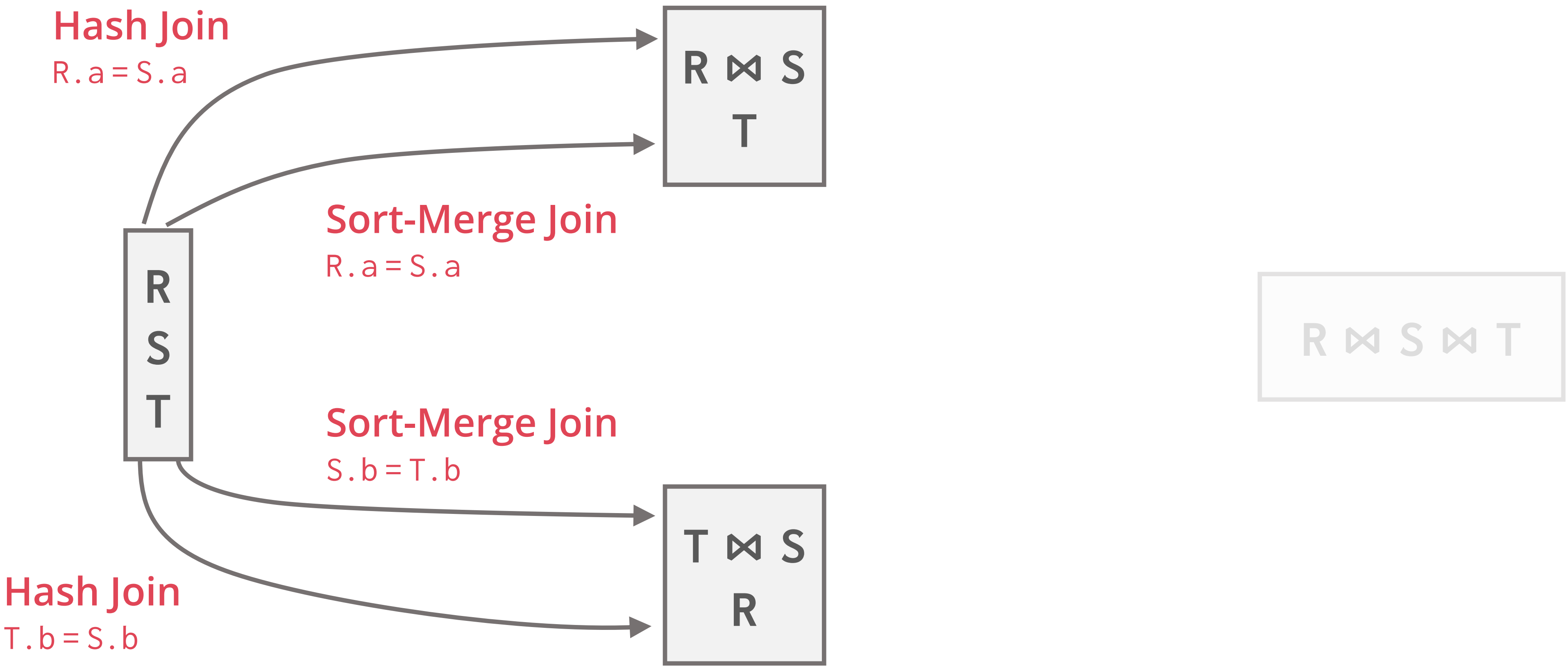
T ⋈ S
R

R ⋈ S ⋈ T

Example: Dynamic Programming

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

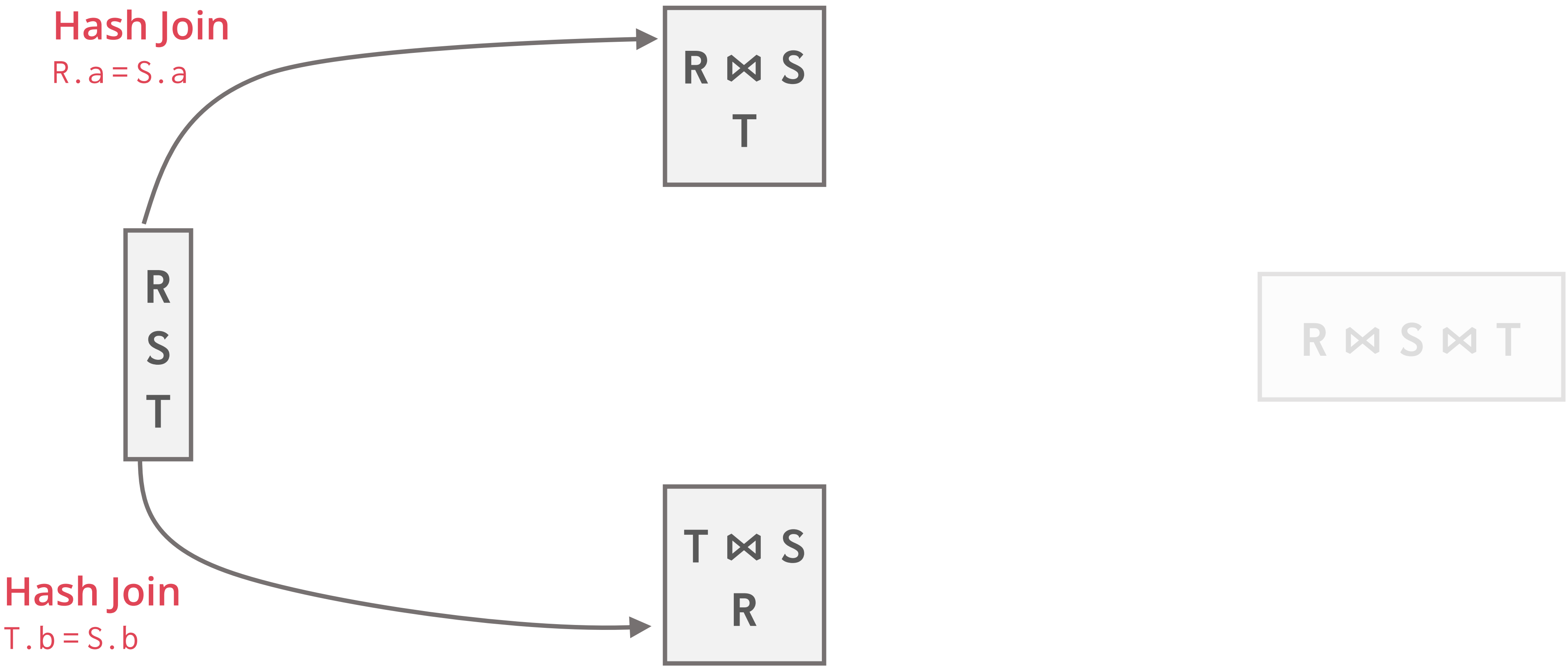
```
SELECT * FROM R, S, T
 WHERE R.A = S.A
   AND S.B = T.B
```



Example: Dynamic Programming

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

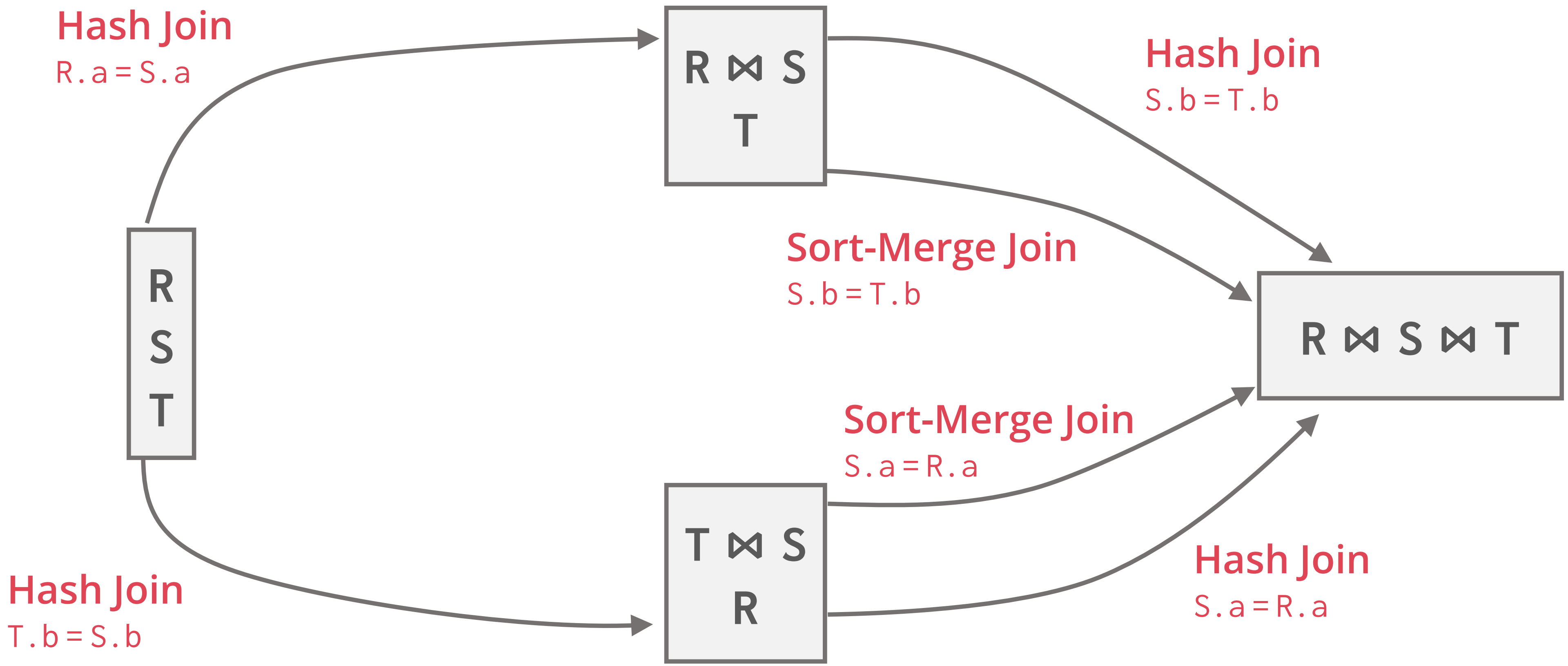
```
SELECT * FROM R, S, T
 WHERE R.A = S.A
   AND S.B = T.B
```



EXAMPLE: DYNAMIC PROGRAMMING

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

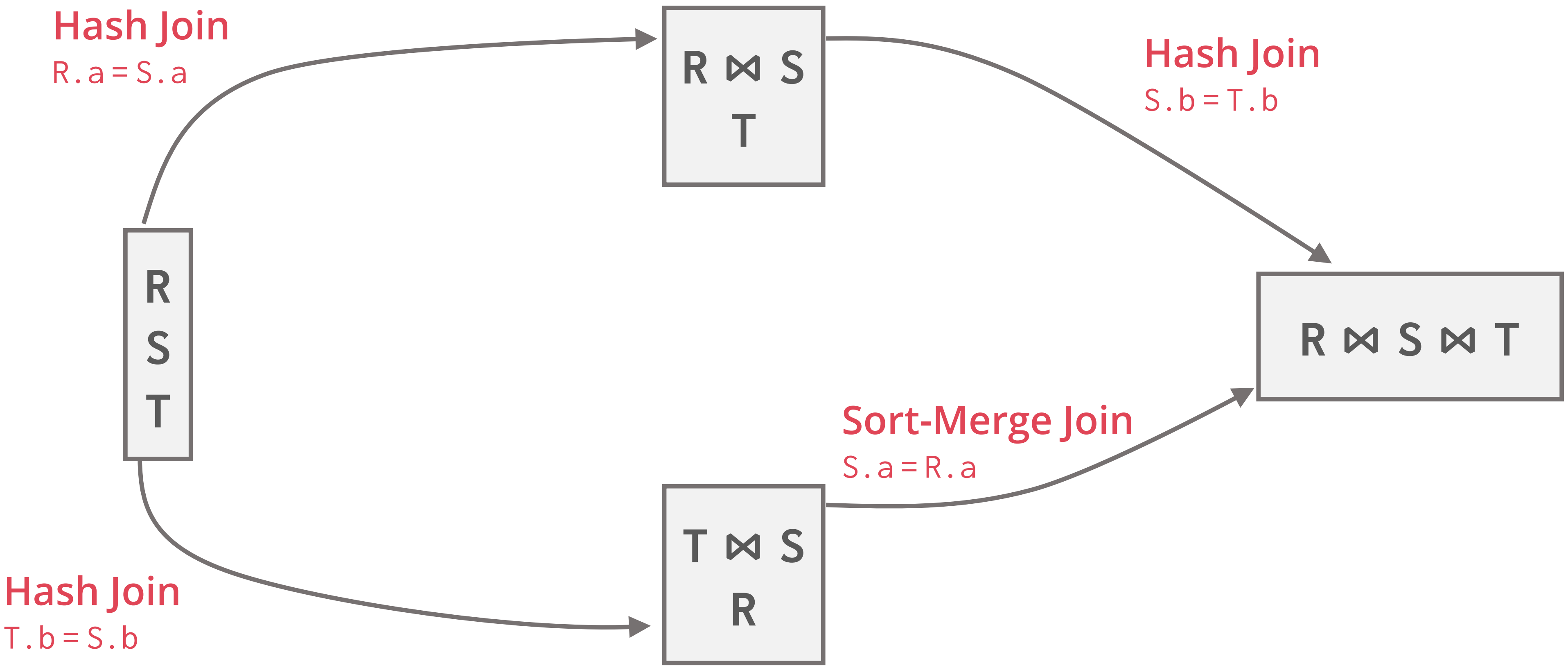
Pass #3 (best 3-relation plans):
best 2-relation plans + one other relation



Example: Dynamic Programming

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

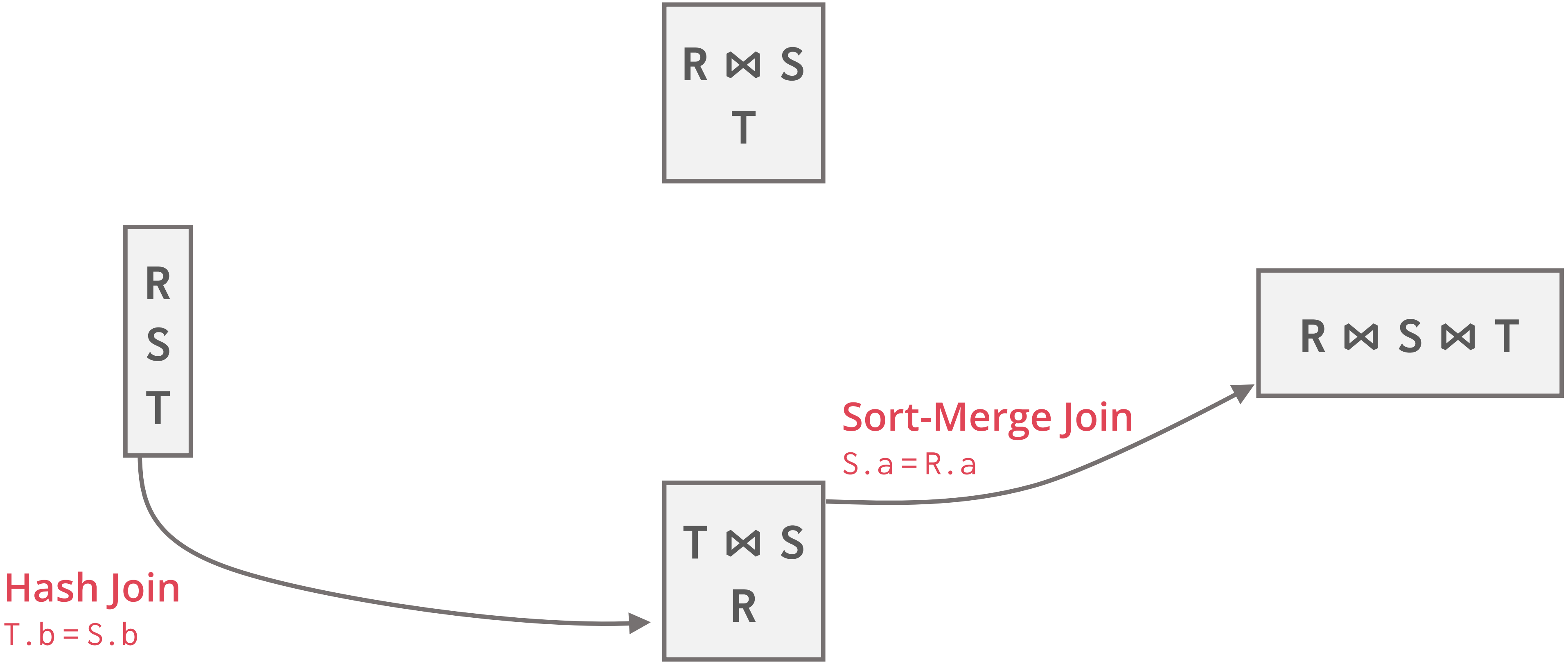
```
SELECT * FROM R, S, T
 WHERE R.A = S.A
   AND S.B = T.B
```



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
```



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

```
SELECT S.sid, COUNT(*) AS number
  FROM Sailors S
  JOIN Reserves R ON S.sid = R.sid
  JOIN Boats B ON R.bid = B.bid
 WHERE B.color = 'red'
GROUP BY S.sid
```

Sailors:

B+ tree on *sid*

Reserves:

Clustered B+ tree on *bid*

B+ tree on *sid*

Boats:

B+ tree on *color*

Pass 1: Best plan for each relation

- **Sailors**, **Reserves**: File scan
- **Boats**: B+ tree on color
- Also B+ tree on Sailors.sid as interesting order (output sorted on *sid*)
- Also B+ tree on Reserves.bid as interesting order (output sorted on *bid*)
- Also B+ tree on Reserves.sid as interesting order (output sorted on *sid*)

Example: Pass 2

Pass 2: Best 2-relation plans

```
// for each left-deep logical plan
foreach plan P in Pass 1:
  foreach FROM table T not in P:
    // for each physical plan
    foreach access method M on T:
      foreach join method ⋈:
        generate P ⋈ M(T)
```

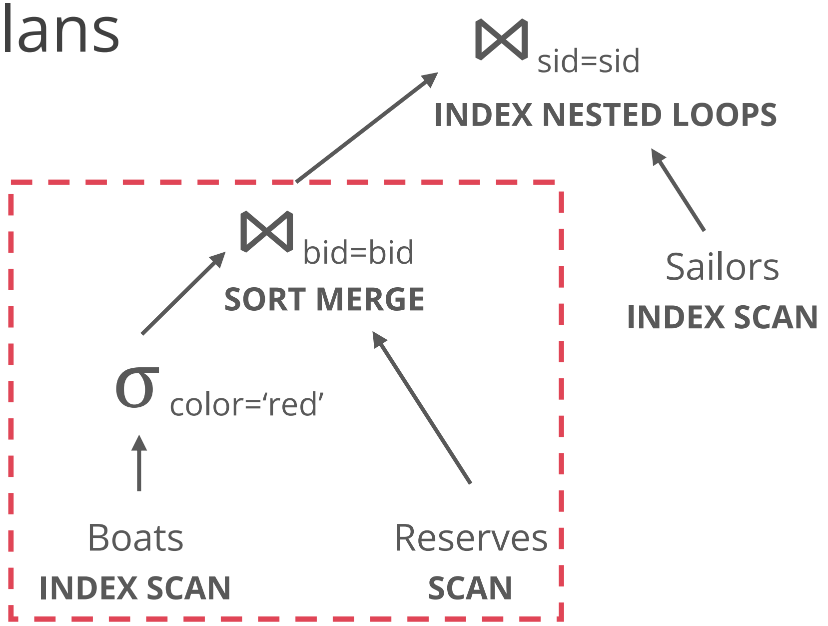
Eliminate cross products

Retain cheapest plan for each (pair of relations, order)

Example: Pass 3

Using **Pass 2 plans** as outer relations, generate plans for the next join in the same way as Pass 2

Example: the marked subplan is the best plan for { Reserves, Boats } and interesting order on Boats.bid and Reserves.bid



Then, add cost for group-by / aggregate:

This is the cost to sort the result by *sid*

… unless it has already been sorted by a previous operator

Finally, choose the cheapest plan

Summary

Query optimisation 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 optimiser is the most complex part of database systems!