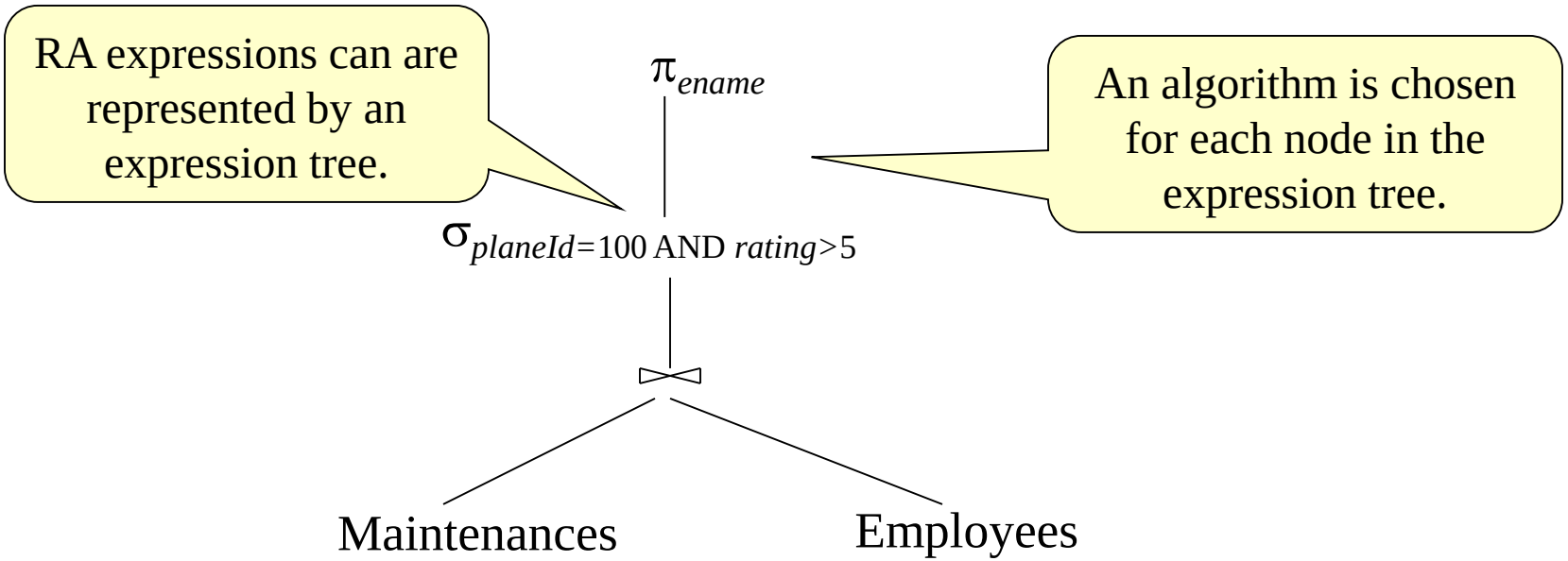
Query Evaluation

An SQL query and its RA equiv.

Employees (sin INT, ename VARCHAR(20), rating INT, address VARCHAR(90))
Maintenances (sin INT, planeId INT, day DATE, desc CHAR(120))

```
SELECT ename
FROM Employees NATURAL JOIN Maintenances
WHERE planeId = 100 AND rating > 5;
```

$\pi_{ename}(\sigma_{planeId=100 \text{ AND } rating>5}(\text{Employees} \bowtie \text{Maintenances}))$



Query Optimization

- SQL queries are translated into RA.

- Query evaluation plans are represented as **trees of relational operators**
  - with labels identifying the algorithm to use at each node.

- **Initial trees** can be transformed to **"better" trees**.
  - Process of finding **good evaluation plans** is called query optimization.

Running Example – Airline

Employees (sin INT, ename VARCHAR(20), rating INT, address VARCHAR(90))

Maintenances (sin INT, planeId INT, day DATE, desc CHAR(120))

- Assume for **Maintenances**:
  - a tuple is 160 bytes
  - a block can hold 100 tuples (16K block)
  - we have **1000 blocks** of such tuples.

- Assume for **Employees**:
  - a tuple is 130 bytes
  - a block can hold 120 tuples
  - we have **50 blocks** of such tuples.

Algorithms for selection

$\sigma_{R.attr = value}(R)$

- if no index on R.attr, just **scan R**.
  - On average half the number of blocks R is occupying.
    - E.g. if R is Maintenances, there will be 1000/2 block reads (I/Os).
- If there is an index on R.attr, **use the index**.
  - Typically 3 disk accesses
    - Assuming non-clustering B-Tree with 3 levels, with the root in MM.

$\sigma_{R.attr < value}(R)$

- If we have a **non-clustering index** we might better scan the table ignoring the index. Why?
- Of course, if we have a **clustering index**, we use it.

Algorithms for projection

- Given a projection we have to scan the relation and drop certain attributes of each tuple.
  - That's easy.

Algorithms for joins

- Joins are **expensive** and very **common**.

- Consider the natural join of **Maintenances** and **Employees.**
- Suppose **Employees** has an unclustered index (B-Tree) on the **SIN** column.

Index nested loops join

- Scan **Maintenances** and for each tuple **probe** **Employees** for matching tuples (using the index on **Employees.SIN**).
- Analysis:
  - For each of the 100,000 maintenance tuples, we try to access the corresponding employee with 3 I/Os (via the index).
  - So, 100,000*3 = 300,000 I/Os !!

Algorithms for joins (merge-join)

Sort-merge join

- **Sort** both tables **on the join column**, then scan them to find matches

- Analysis:
  - Sort Maintenances with 2PMMS, and Employees with 2PMMS
  - Cost for sort is
    - 2 * 2 * 1000 = 4000 I/Os for Maintenances and
    - 2 * 2 * 50 = 200 I/Os. for Employees
  - Then we merge-join. This requires an additional scan of both tables.
  - Thus the total cost is 4000+ 200+ 1000+ 50= **5250** I/Os. (Much better!!)

Algorithms for joins (sort-merge)

So, we have:

- **index nested loops join**: 300,000 I/Os
- **sort-merge join** : **5,250** I/Os.

- Why bother with **index nested loops join**?
- Well, "**index nested loops**" method has the nice property that it is **incremental**.
  - The cost of our example join is incremental in the number of **Maintenances** tuples that we process.
  - If some additional selection in the query allows us to consider only a small subset of **Maintenances** tuples, we can avoid computing the full join of **Maintenances** and **Employees**.
    - Suppose we only want the result of the join for plane **100** on **Mar 08, 2017**, and there are very few such maintenances, say only two.
    - For each of these two tuples, we probe **Employees** (twice), and we are done.

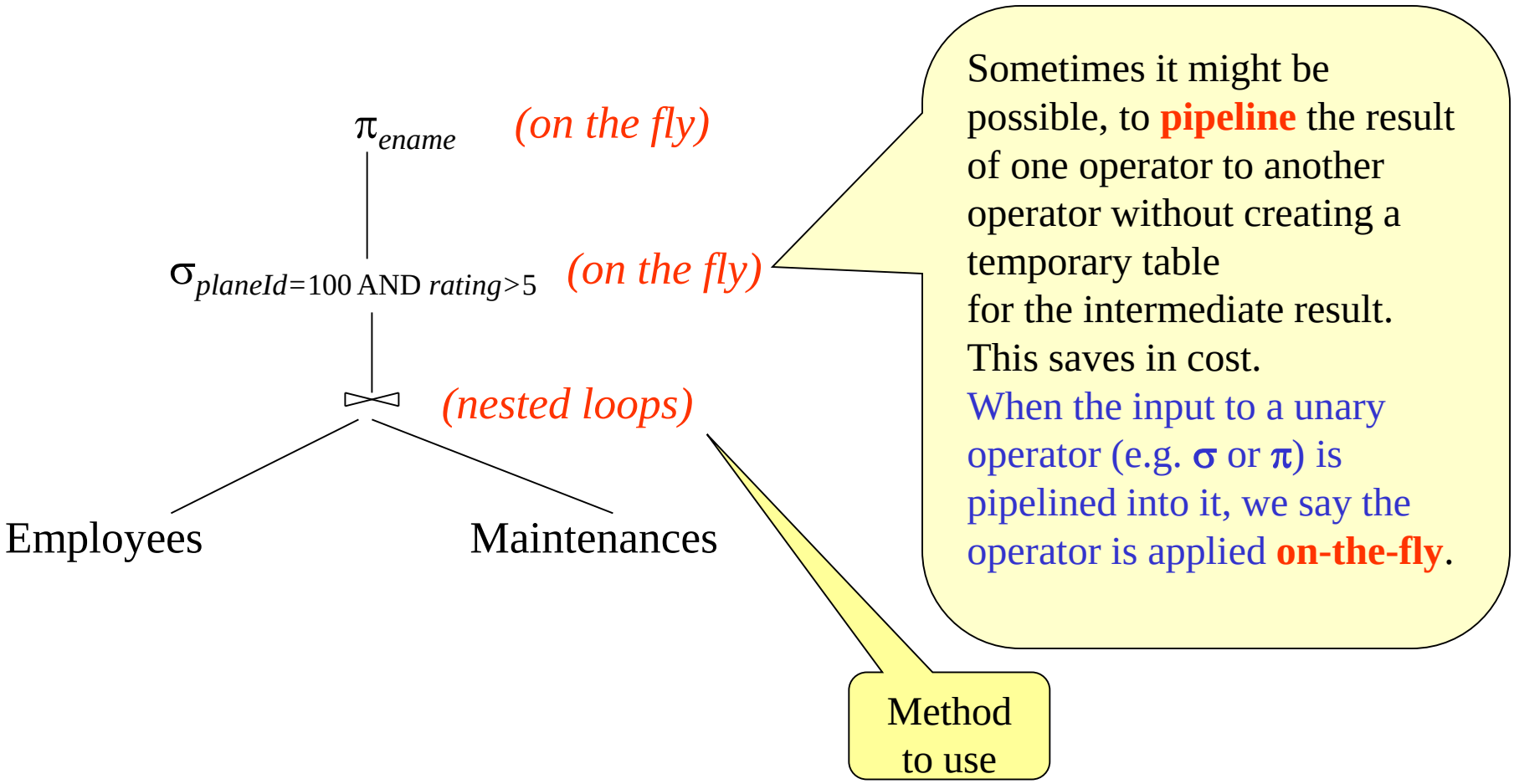
Optimization

- Observe the choice of index nested loops join is based on considering the query as a whole, including the **selection** on **Maintenances**, rather than just the join operation by itself.

- This leads us to the next topic, **query optimization**, which is the process of finding a good plan for the **entire query**.

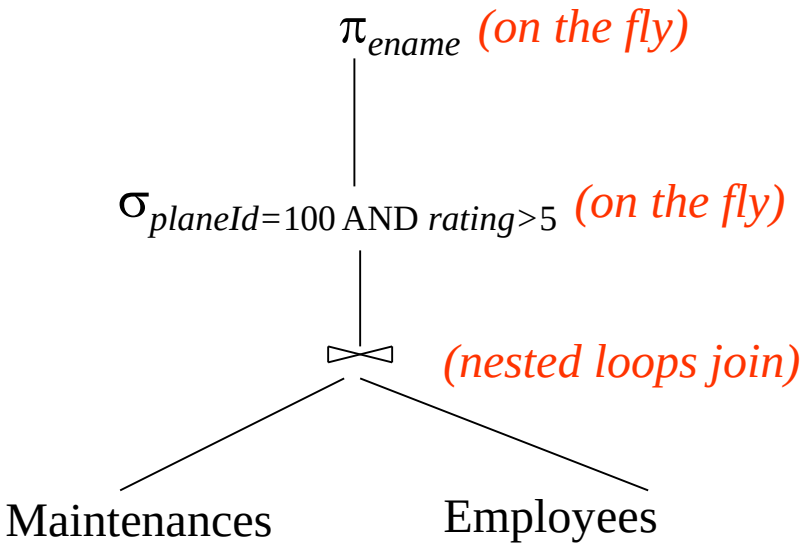
Query evaluations plans

- **Recall**: A **query evaluation plan** consists of an RA tree, with indications at each node for the method/algorithm to use.



Alternative query evaluation plans

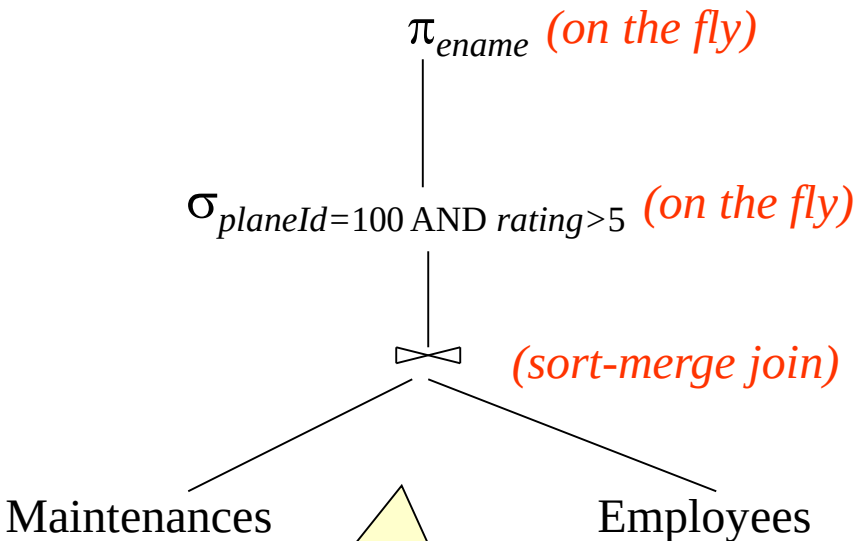
- Let's look at two (naïve plans)



Cost for this plan:

300,000 I/Os for the join.

σ and π are done in the fly; no I/O cost for them.



Cost for this plan:

5,250 I/Os for the join.

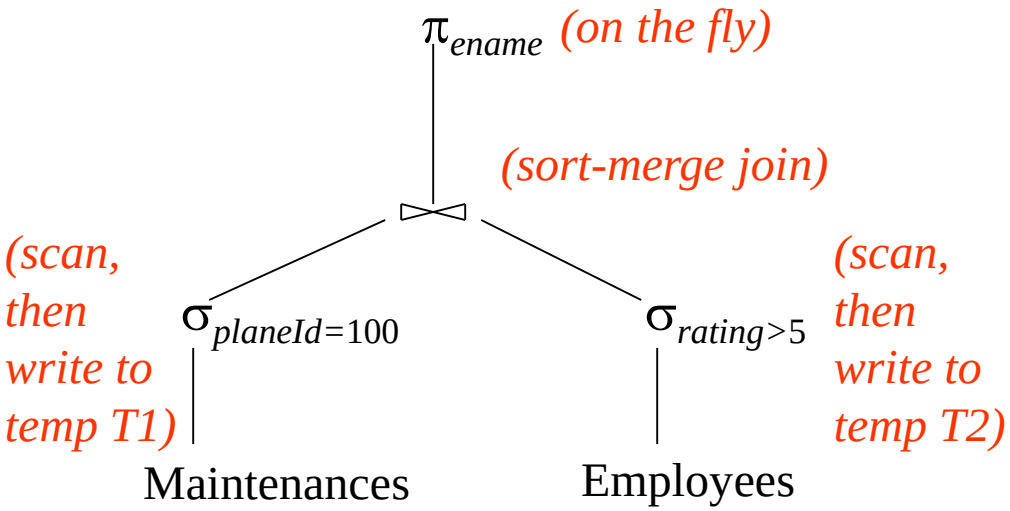
σ and π are done in the fly; no I/O cost for them.

We don't consider the cost of writing the final result, since it is the same for any plan.

Plan: Pushing selections I

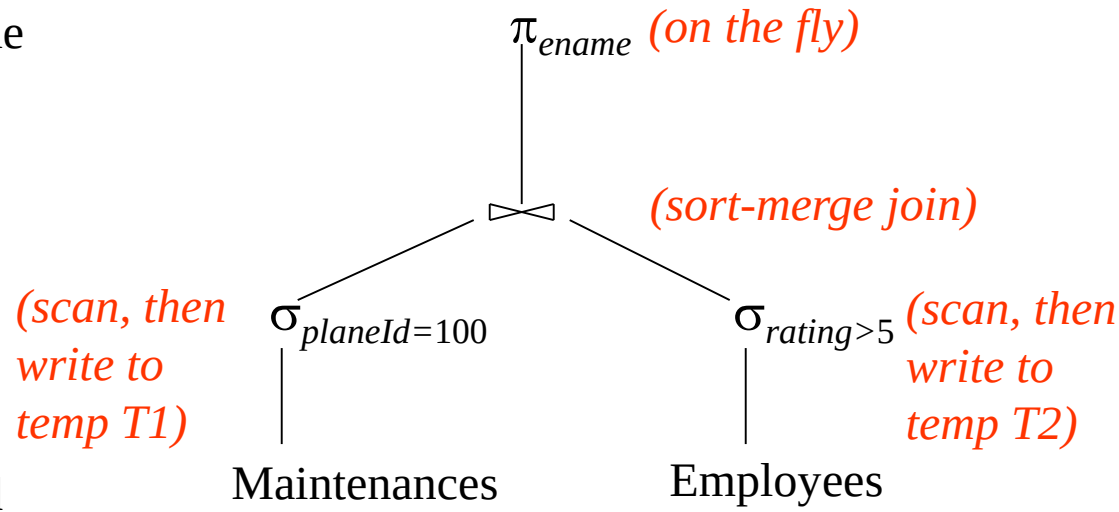
- Good **heuristic** for joins is to reduce the sizes of the tables to be joined as much as possible.

- One approach is to apply selections early; i.e. 'push' the selection ahead of the join.



Plan: Pushing selections II

- Cost of applying $\sigma_{planeId=100}$ to **Maintenances** is:
  - scanning **Maintenances** (1000 blocks)
  - writing result to temporary table Tl.
- To estimate the size of T1, we reason as follows:
  - if we know that there are 100 planes, we can assume that maintenances are spread out uniformly across all planes and estimate the number of blocks in T1 to be 1000/100 = **10**.
- i.e. 1000+10 = **1010** I/Os.

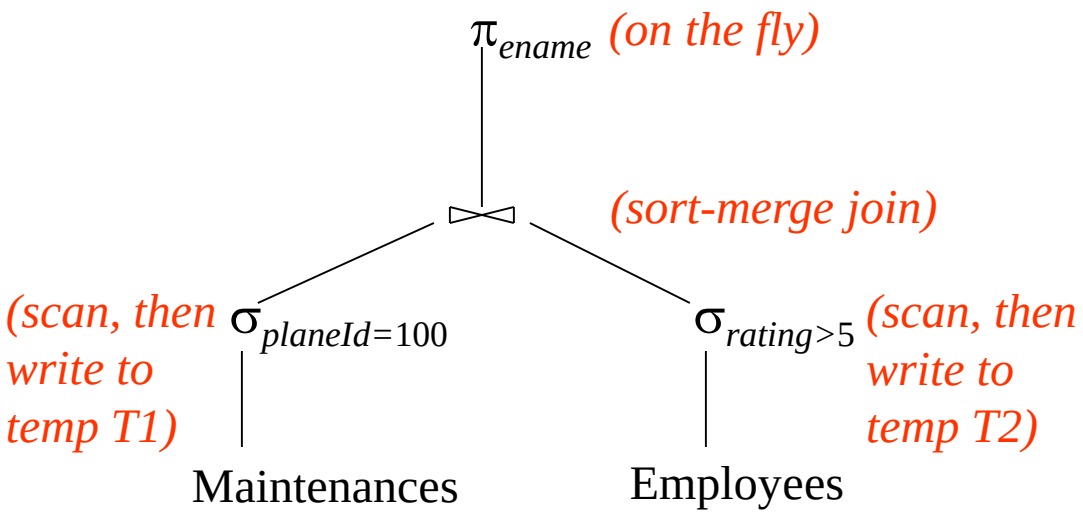


Plan: Pushing selections III

- Cost of applying *rating*> 5 to **Employees** is:
  - scanning **Employees** (50 blocks)
  - writing out the result to temporary table T2.

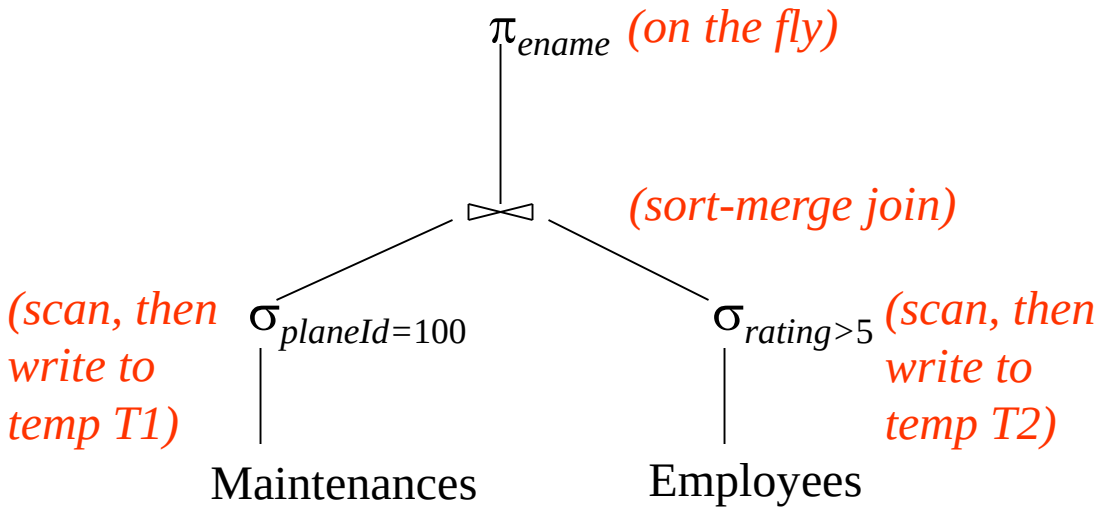
- If we assume that ratings are uniformly distributed over the range 1 to 10, we can approximately estimate the size of T2 as 25 blocks.

- i.e. 50 + 25 = **75** I/Os



Plan: Pushing selections IV

- For **sort-merge join** of T1 and T2,
  they are first sorted, then join-merged.
- 2PMMS for each one:
  - cost of sorting T1 is
    2 * 2 * 10 = 40 I/Os
  - cost of sorting T2 is
    2 * 2 * 25 = 100 I/Os
- To join-merge the sorted T1 and T2,
  we need to scan them. Cost:
    10+25=35 I/Os.



- Final projection is done on-the-fly
- Ignore cost of writing the final result.

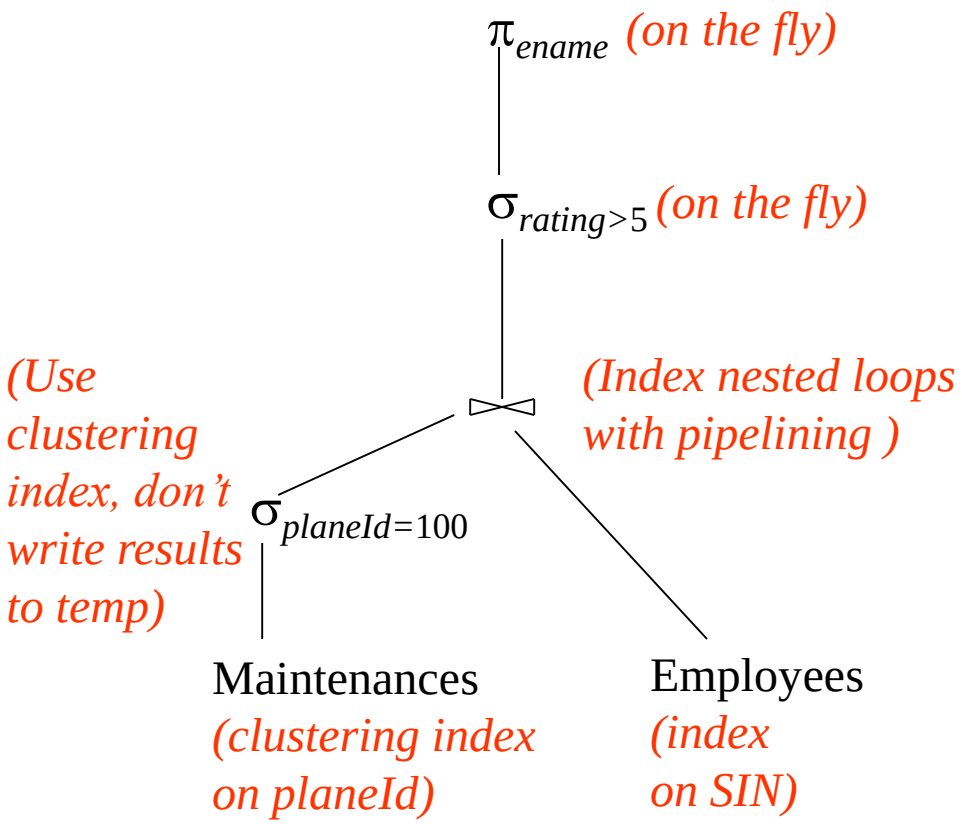
- The total cost of the plan is:
  (1010+75)+(40+100+35) =
    1260 I/Os

Pushing projections

- A further refinement is to push projections, just like we pushed selections past the join.
- Observe that only the *SIN* attribute of T1 and the *SIN* and *ename* attributes of T2 are really required.
- When we scan **Maintenances** and **Employees** to do the selections, we could also eliminate unwanted columns.
  - This on-the-fly projection reduces the sizes of the temporary tables T1 and T2.
  - The reduction in the size of T1 is substantial because only an integer field is retained.
  - In fact, T1 now fits within three blocks (and be all can be in MM), and we can perform a block nested loops join with a single scan of T2.

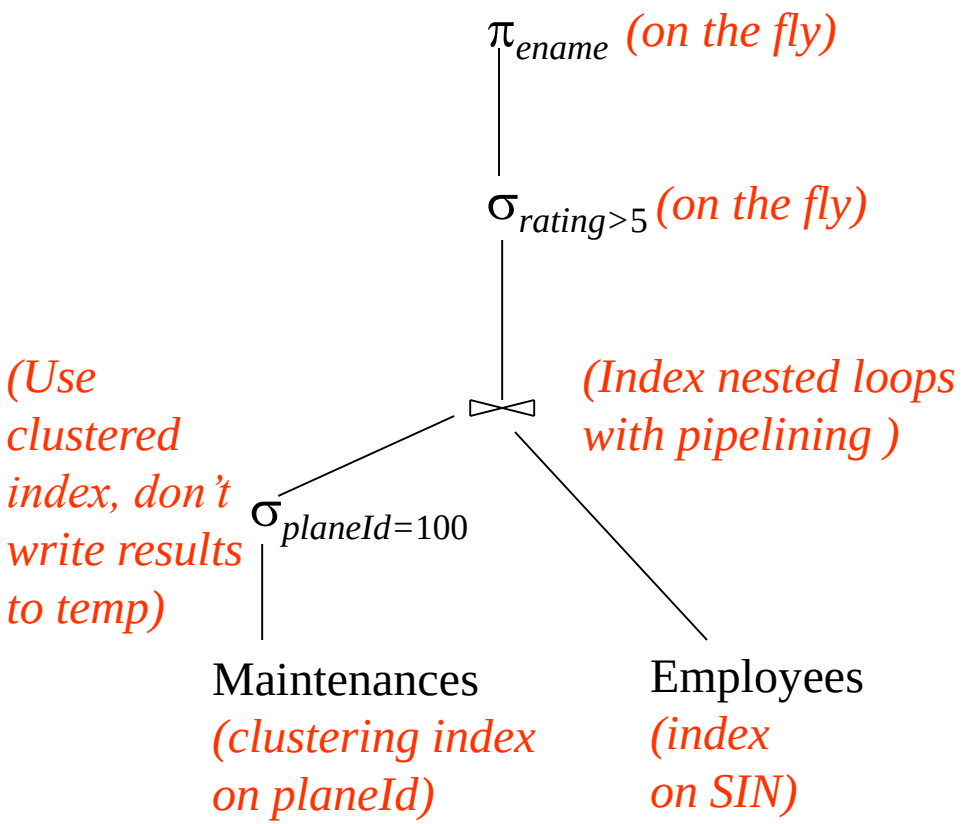
Plan using Indexes I

- Suppose we have a clustering B-Tree index on the *planeId* field of **Maintenances** and another, non-clustering, B-Tree index on the *SIN* field of **Employees**.
- We can use the plan in the figure.



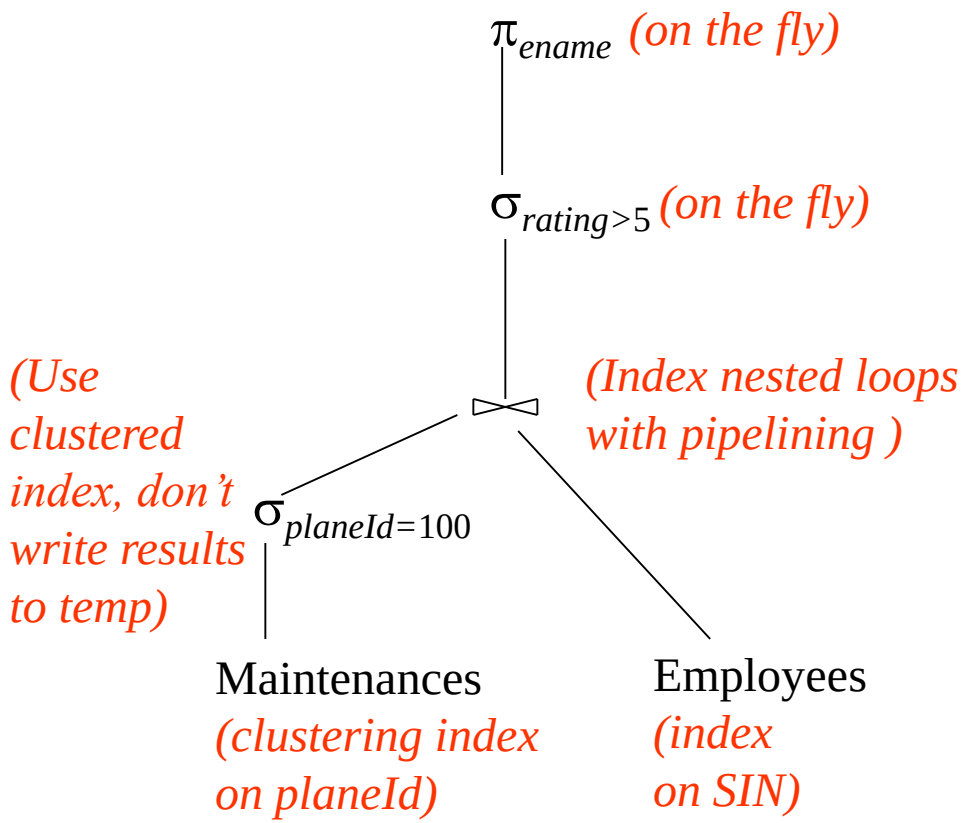
Plan using Indexes II

- Assume that there are 100 planes and assume that the maintenances are spread out uniformly across all planes.
  - We can estimate the number of selected tuples to be 100,000/100=1000.
  - Since the index on *planeId* is clustering, these 1000 tuples appear consecutively and therefore, the cost is:
  - 10 blocks I/Os + 2 I/Os
    - (the 2 I/Os are for finding the first block via the index.



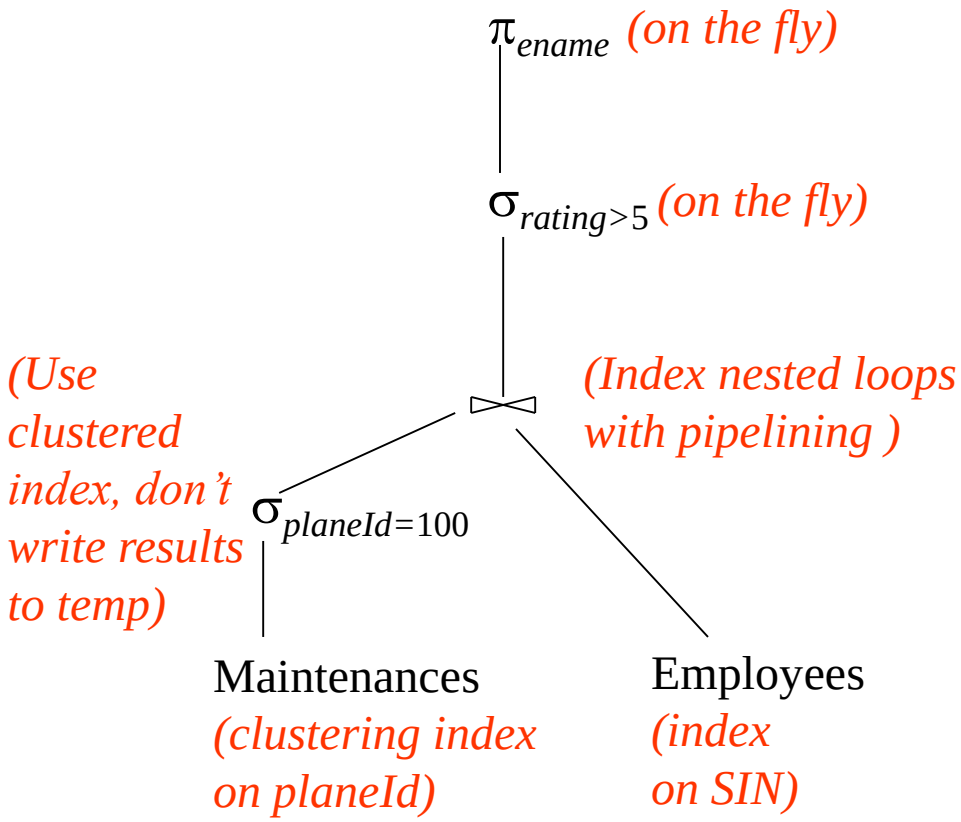
Plan using Indexes III

- For each selected tuple, we retrieve matching **Employees** tuples using the index on the *SIN* field;
- The join field *SIN* is a key for **Employees**. Therefore, at most one **Employees** tuple matches a given **Maintenances** tuple.
  - The cost of retrieving this matching tuple is 3 I/Os.
  - So, for the 1000 Maintenances tuples we get 3000 I/O's.
- For each tuple in the result of the join, we perform the selection *rating>5* and the projection of *ename* on-the-fly.
- So, total: 3000+12 = **3012** I/Os.



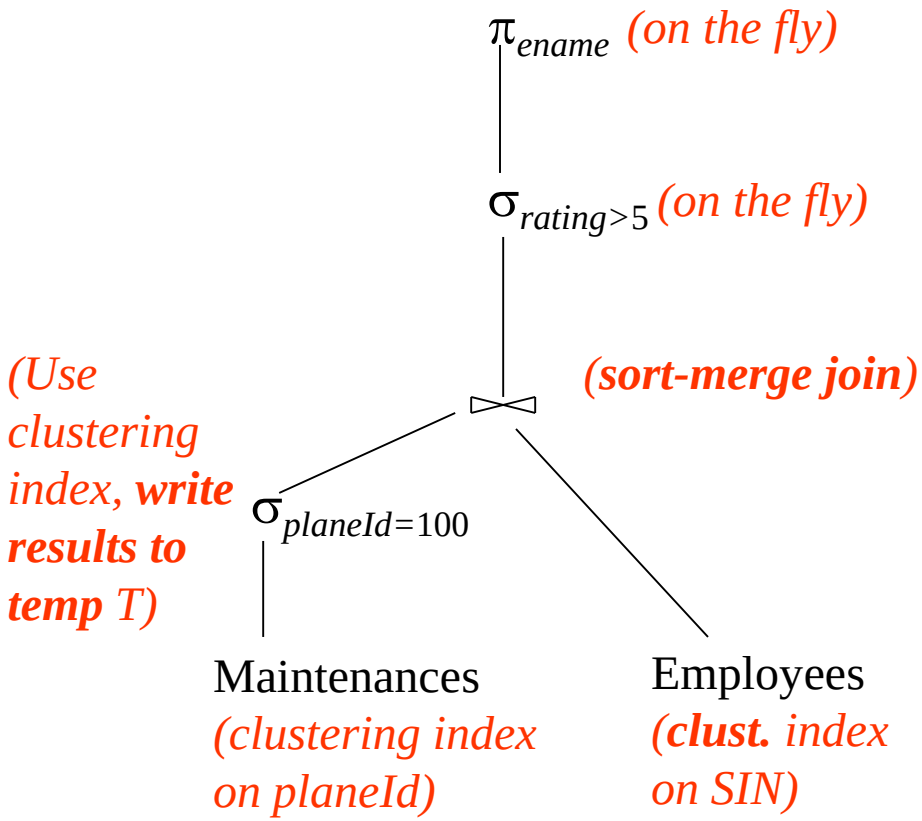
Plan using Indexes IV

- Why didn't we push the selection *rating>5* ahead of the join?
- Had we performed the selection before the join, the selection would involve scanning **Employees** (since no index is available on the *rating* field of **Employees**).
- Also, once we apply such a selection, we have no index on the *SIN* field of the result of the selection.
- Thus, pushing selections ahead of joins is a good heuristic, but not always the best strategy.
  - Typically, the existence of useful indexes is the reason a selection is *not* pushed.



Plan using Indexes V

- Suppose that we have a clustering index on Employees.sin.
- What about this plan?
- The size of T would be the number of blocks for the **1,000 Maintenances** tuples that have planeId=100.
  - i.e. T is 10 blocks.
- The cost of $\sigma_{planeId=100}$ as before is 12 I/Os to retrieve plus 10 additional I/Os I/Os to write T.
- Then, sort T on the SIN attribute. 2PMMS = 40 I/Os.
- **Employees** is already sorted since it has a clustering index on SIN.
- The merge phase of the join needs a scan of both T and Employees. So, 10+50=60 I/Os.
- Total: (12+10)+40+60 = **122 I/Os**



This improved plan also demonstrates that pipelining is not always the best strategy.