Chapter 16 Query Optimization

> So now we have understand how we do individual operations. The question becomes, how do we put things together.

Query Optimization

- Step of query optimizaiton
- Given an SQL query
  1. Convert it to (enhanced) relational algebra

     > We have sql, the convert into relational algebra. I put the work in hand because things like by aggregation does not appear in the original relational algebra definition. But we can but we can think of them as extra operations so that it fits the it fit the paradigm.

  2. Build multiple parse trees

     ```
     select *
     from A, B, c
     where
     ```

     Just a query like this. `A, B, C` are $A \bowtie B \bowtie C$. I can do $X_1 = A \bowtie B$ first, then $X_1 \bowtie C$. I also can do $X_2 = B \bowtie C$ first, then $A \bowtie X_2$. I also can do $X_3 = A \bowtie C$ first, then $B \bowtie X_3$. And each of them can have variety performance. It very often turns out that which query you need to do join first matters significantly in terms of performance. So you have to at the beginning, at the very least, keep all options open.

     - Note: Multiple ways to decorate a tree

       > And you need to decorate a tree. How do we do a join? Do we do hash join? Do we do sort merge? Do we do nested loop? If I do nested loop, which should be outer loop? Which should be inner loop? How many buffer do I set in the outer loop? How many buffer do I set inner loop?
       >
       > In the DataBase system term is called a plan. Typically we call it a tree because this is classroom. Tree is something we're familiar with. When you go out in the world of DBA, they'll call each tree a plan.

     - Each decorated tree is also called a **plan**
  3. Calcualte the cost of each plan
  4. Pick the best one

> Sounds simple enough. Make perfect sense logically. But there are challenges.
>
> `1.` is hopefully standards now. `2.` already have issues. How many trees do I want to be? Remember, we have three tables to join. We have free options. If you have four tables to join, how many options are there?

> Why do you need a table join? Many database systems are divided in what we call a star schema. we will see whether no sql can do better optimization than this was.

> Remember, if you work for a company, do I care about which database system I use? If I'm the boss, I paid money for you guys to develop a system. All I care is performance. And you have a convenient, no thicker system, but the time to execute each query is 5 seconds lower. Do you think I want this system? No!
>
> Noticed that many Data System that is free for you to use. But if you need support, that's where Red Hat or whatever all those database company earn their money is support. If you're a big company and you cannot accept somebody is failing? You have to have constant support. And that support is where those companies earn the money back. They always have like Enterprise Edition.

16.2 Convert to relational algebra

- In a

  ```
  SELECT  ...
  FROM  ...
  WHERE ...
  ```

  SQL query

  - `SELECT ...` corresponds to π
  - `WHERE ...` corresponds to σ
  - `FROM ...` corresponds to $\bowtie$ (if corresponding `WHERE` course exists), $\times$ if not
  - `UNION, INTERSECT, EXCEPT` correspond to $\cup, \cap, -$
  - `GROUP BY, HAVING` and aggregate does not appear in relational algebra, so have extra operators added to accommodate it.
  - We'll deal with nested query later.

    > Nested query is a tricky thing.
    >
    > Exist, not exist. In, Not in all these kind of stuff. Nowadays you can even that's the career in the in the from costs.

    > ```
    > select *
    > FROM (select ...) AS A
    > ```
    >
    > Do I encourage the query in this way? Not necessarily. But you can do it.

- From the relational algebra, one can build a parse tree
- Parse tree specifies the order of execution of the operations
  - Post-order traversal of the tree

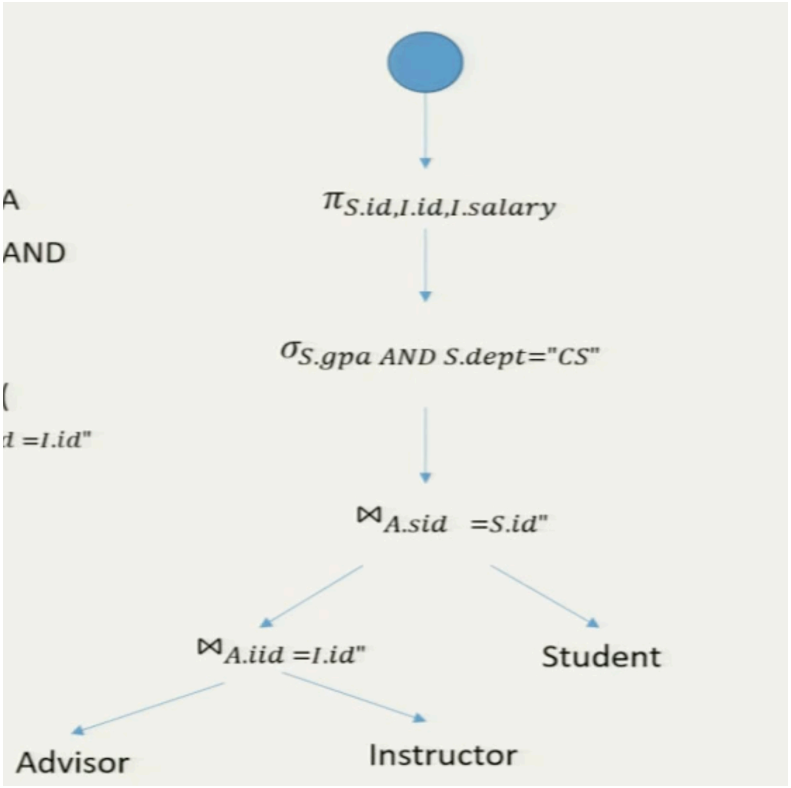
> Post-order traversal -> data structure class.

- Notice that, for joins, we put the "outer loop" (if we want to do nested loop) on the left.

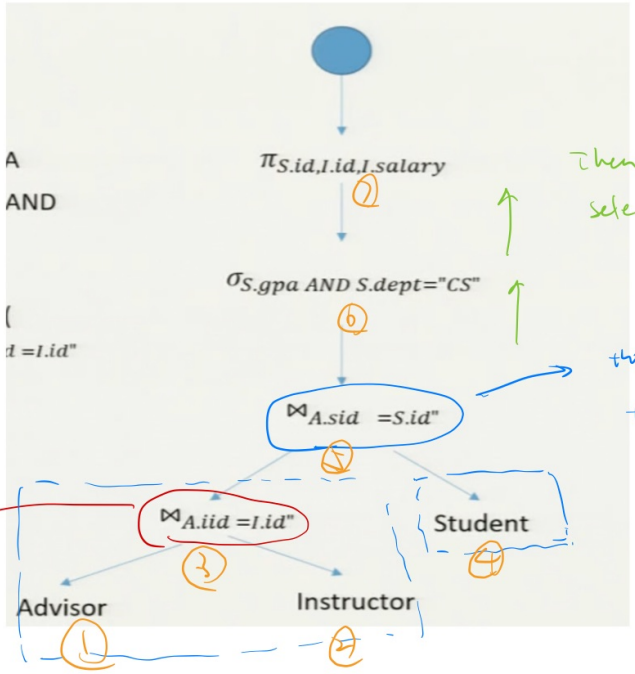
- Example

```
SELECT S.id, I.id, I.salary
From Student S, Instructor I, Advise A
WHERE S.id = A.s_id AND I.id = A.i_id AND S.dept = "CS" AND S.gpa >= 3.5
```

$\Pi_{S.id,I.id,I.salary}(\sigma_{S.gpa\ And\ S.dept=''CS''}(Student \bowtie_{A.sid=S.id''} Advisor \bowtie_{A.iid=I.id}))$



> Each interal node is an operation, selection, projection, join may be even group by. Each leaf node is a table.



Then I've got projection, selection.

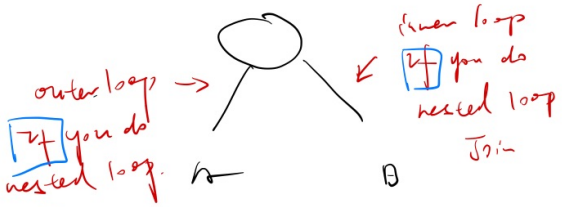
this means join the the results of left table and the Student table

This operation means that I'm joining these Advisor, Instructor ≥ tables.

post-order left-right-mid

One thing about Join.

Join takes two table with convention.



Notice that not all Join operation will have outer loop and inner loop.

But if we want to do a nested loop, we will always write the outer loop left and inner loop right hand side.

A ⋈ B ⋈ C

X1 : B ⋈ B

X1 ⋈ C
↑ inner loop

C ⋈ X1
↑ inner loop.

if nested loop.

> Just be careful, you don't want to spend 2 minutes optimizing the query where you only get 1/2 execution where you can make you current 1/2 faster. If that happened, I will probably not spend the 2 minutes and just bite and run 1/2 longer. By nature, query optimizer have to be done very fast. Any exponential number of things is out of the question. But the number of possible tree really grow exponentially. So there's always have to be give and take. Most query optimizer don't give you the optimal global best query. It's just given the limited time we have, look at a bunch of options and let's see who is based on those options who say what is the best. And if we run out of time we can't find a global optimal, so be it. We'll llive that. See the challenge ?
>
> You don't know the best way to excute a query until you know the result your query.

Generate multiple parse trees

- However, there are multiple options of parse trees
- For relational algebra
  - Joins are associative
  - Selection are commutative
  - Same as projection
  - Also there are some distributive properties

Transformation of Relational Expressions

> If you see a query, the first thing you should do is to convert it to parse tree or at least one query.
>
> The following always what ?
>
> Do the join first, then do the selection, then do the projection. This will always be correct, it will give you the correct answer. It may not give you an efficient answer.

- Two relational algebra expressions are said to be equivalent if the two expressions generate the same set of tuples on every legal database instance

  > Can you tell me what is illegal database instance?
  >
  > If two tuples have conflict thing.
  >
  > If illegal Franky, the definition of equivalnet don't care about it. So basically garbage in, garbage out.

  - Note: order of tuples is irrelevant
  - We don't care if they generate different results on databases that violate integrity constraints
- In SQL, inputs and outputs are multisets of tuples

  > SQL is not quite relational algebra, the result of sql query it not quite relation. Why not?

> Duplicate are not removed. So that means what you getting is not necessarily a set. Now the good news is that in real life, this doesn't matter too much. At least in this context.

  - Two expressions in the multiset version of the relational algebra are said to be equivalent if the two expressions generate the same multiset of tuples on every legal database instance.

- An equivalence rule says that expressions of two forms are equivalent
  - Can replace expression of first form by second, or vice versa

> Now once we have this, we want to come up with a set of what we call equivalence.

Equivalence Rules

1. Conductive selection operations can be deconstructed into a sequence of individual selections.

$$\sigma_{\theta_1 \wedge \theta_2}(E) \equiv \sigma_{\theta_1}(\sigma_{\theta_2}(E)) \tag{1}$$

2. Selection operations are commutative.

$$\sigma_{\theta_1}(\sigma_{\theta_2}(E)) \equiv \sigma_{\theta_2}(\sigma_{\theta_1}(E)) \tag{2}$$

3. Only the last in a sequence of projection operations is needed, the others can be omitted.

$$\prod_{L1}(\prod_{L2}(\dots(\prod_{Ln}(E))\dots)) \equiv \prod_{L1}(E) \tag{3}$$

   Where $L_1 \subseteq L_1 \dots \subseteq L_n$

> $\prod_{name}(\prod_{ssn,name}(Student)) = \prod_{name}(Student)$ This is legal.
> $\prod_{name}(\prod_{ssn}(Student))$ This is not.

4. Selections can be combined with Cartesian products and theta joins.

   a. $\sigma_\theta(E_1 \times E_2) \equiv E_1 \bowtie_\theta E_2$

   b. $\sigma_{\theta_1}(E_1 \bowtie_{\theta_2} E_2) \equiv E_1 \bowtie_{\theta_1 \wedge \theta_2} E_2$

> If I do a join and then do a selection, I can certainly pull the seleciton as a condition of the Join.

5. Theta-join operations (and natural joins) are commutative.

$$E_1 \bowtie E_2 \equiv E_2 \bowtie E_1 \tag{4}$$

6. (a) Theta joins are associative in the following manner:

$$(E_1 \bowtie E_2) \bowtie E_3 \equiv E_1 \bowtie (E_2 \bowtie E_3) \tag{5}$$

   (b) Theta joins are associative in the following manner:

$$(E_1 \bowtie_{\theta_1} E_2) \bowtie_{\theta_2 \wedge \theta_3} E_3 \equiv E_1 \bowtie_{\theta_1 \wedge \theta_3} (E_2 \bowtie_{\theta_2} E_3) \tag{6}$$

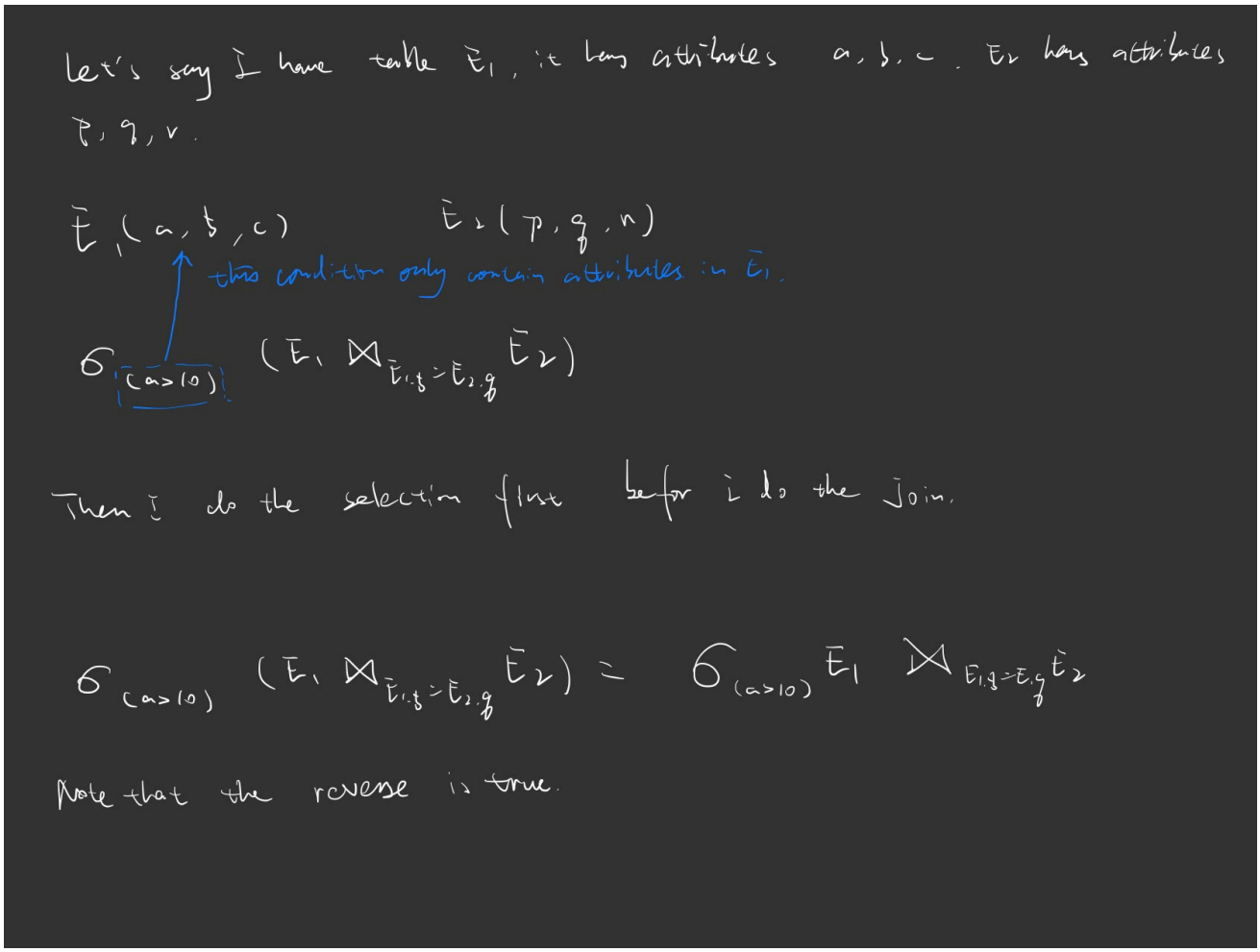
   where θ_2 involves attributes from only E_2 and E_3.

> But the key thing is that for correctness, it doesn't matter. The efficiency matters a lot.
>
> That all these doesn't matter, because you will get the correct result. These rules tell you what you can do, whether you want to use this. That's to create optimizesion.

7. The operation distributes over the theta join operation under the following two conditions:

   (a) When all the attributes in θ_0 involve only the attributes of one of the expressions (E_1) being joined.

$$\sigma_{\theta_0}(E_1 \bowtie_\theta E_2) \equiv (\sigma_{\theta_0}(E_1)) \bowtie_\theta E_2) \tag{7}$$



   (b) When θ_1 involves only the attributes of E_1 and θ_2 involves only the attributes of E_2

$$\sigma_{\theta_1 \wedge \theta_2}(E_1 \bowtie_\theta E_2) \equiv (\sigma_{\theta_1}(E_1)) \bowtie_\theta (\sigma_{\theta_2}(E_2)) \tag{8}$$

> $\sigma_{(a>10)\ and\ (r<10)}(E_1 \bowtie_{E_1.b=E_2.q} E_2) \equiv (\sigma_{a>10}(E_1)) \bowtie_{E_1.b=E_2.q} (\sigma_{r<10}(E_2))$
>
> All the things are relatively straightforward.

> Let's say you have two options. Will you prefer to do the left hand side? Or will you do the right hand side? Then we do the join first or selection first before you do Join?
>
> Everything seems to suggest we should do the select before join.

8. The projection operation distributes over the theta join operation as follows:

   (a) if θ involes only attributes from $L_1 \cup L_2$:

$$\prod_{L_1 \cup L_2}(E_1 \bowtie_\theta E_2) \equiv \prod_{L_1}(E_1) \bowtie_\theta \prod_{L_2}(E_2) \tag{9}$$

> However, it is very often that you want a join condition does not contain the attribute that you want to output. Then you just have to be a bit more careful.

(b) In general, consider a join $E_1 \bowtie_\theta E_2$

- Let L_1 and L_2 be sets of attributes from E_1 and E_2, respectively.
- Let L_3 be attributes of E_1 that are involved in join condition θ, but are not in $L_1 \cup L_2$ and
- Let L_4 be attributes of E_2 taht are involved in join condition θ, but are not in $L_1 \cup L_2$.

$$\prod_{L_1 \cup L_2}(E_1 \bowtie_\theta E_2) \equiv \prod_{L_1 \cup L_2}(\prod_{L_1 \cup l_3}(E_1) \bowtie_\theta \prod_{L_2 \cup L_4}(E_2)) \tag{10}$$

Similar equivalences hold for outerjoin operations: $\bowtie, \ltimes and \rtimes$

9. The set operations union and intersection are commutative

$$E_1 \cup E_2 \equiv E_2 \cup E_1 \tag{11}$$

$$E_1 \cap E_2 \equiv E_2 \cap E_1 \tag{12}$$

(set difference is not commutative)

10. Set union and intersection are associative

$$(E_1 \cup E_2) \cup E_3 \equiv E_1 \cup (E_2 \cup E_3) \tag{13}$$

$$(E_1 \cap E_2) \cap E_3 \equiv E_1 \cap (E_2 \cap E_3) \tag{14}$$

11.

11. The selection operation distributes over $\cup, \cap and -$,

a.

$$\sigma_\theta(E_1 \cup E_2) \equiv \sigma_\theta(E_1) \cup \sigma_\theta(E_2) \tag{15}$$

b.

$$\sigma_\theta(E_1 \cap E_2) \equiv \sigma_\theta(E_1) \cap \sigma_\theta(E_2) \tag{16}$$

c.

$$\sigma_\theta(E_1 - E_2) \equiv \sigma_\theta(E_1) - \sigma_\theta(E_2) \tag{17}$$

d.

$$\sigma_\theta(E_1 \cap E_2) \equiv \sigma_\theta(E_1) \cap E_2 \tag{18}$$

e.

$$\sigma_\theta(E_1 - E_2) \equiv \sigma_\theta(E_1) - E_2 \tag{19}$$

preceding equivalence does not hold for $\cup$

> Union basically is join. Every attribute has to match, so you can think of intersection as some a kind of a join. Union is some kind of just jump everything together depending on whether the system.

> Selection can be push in.

12. The projection operation distributes over union

$$\prod_L(E_1 \cup E_2) \equiv (\prod_L(E_1)) \cup (\prod_L(E_2)) \tag{20}$$

13. Selection distributes over aggregation as below

$$\sigma_\theta({}_g\gamma_A(E)) \equiv {}_g\gamma_A(\sigma_\theta(E)) \tag{21}$$

provided θ only involves attributes in G

> ${}_g\gamma_A$: group by

14. a. Full outerjoin is commutative:

$$E_1 ⟗ E_2 \equiv E_2 ⟗ E_1 \tag{22}$$

b. Left and right outerjoin are not commutative, but:

$$E_1 ⟕ E_2 \equiv E_2 ⟖ E_1 \tag{23}$$

15. Selection distributes over left and right outerjoins as below, provided θ_1 onlu involves attributes of E_1

a.

$$\sigma_{\theta_1}(E_1 ⟕_\theta E_2) \equiv (\sigma_{\theta_1}(E_1) ⟕_\theta E_2 \tag{24}$$

b.

$$\sigma_{\theta_1}(E_1 ⟖_\theta E_2) \equiv E_2 ⟕_\theta (\sigma_{\theta_1}(E_1)) \tag{25}$$

16. Outerjoins can be replaced by inner joins under some conditions

a.

$$\sigma_{\theta_1}(E_1 ⟕_\theta E_2) \equiv \sigma_{\theta_1}(E_1 \bowtie_\theta E_2) \tag{26}$$

b.

$$\sigma_{\theta_1}(E_1 ⟖_\theta E_2) \equiv \sigma_{\theta_1}(E_1 \bowtie_\theta E_2)) \tag{27}$$

provided θ_1 is null rejecting on E_2

> The only thing you want to be careful is for outerjoins

Note that several equivalences that hold for joins do not hold for outerjoins

- $\sigma_{year=2017}(instructor ⟕ teaches) \not\equiv \sigma_{year=2017}(instructor \bowtie teaches)$

> In outer joins, we not just only want to match, but we also need to return tuples that doesn't match.

- Outerjoins are not associative

$$(r \rtimes s) \bowtie t \not\equiv r \rtimes (s \rtimes t) \tag{28}$$

  - e.g. with $r(A, B) = \{(1, 1)\}, s(B, C) = \{(1, 1)\}, t(A, C) = \{\}$

Pictorial Depiction of Equivalence Rules

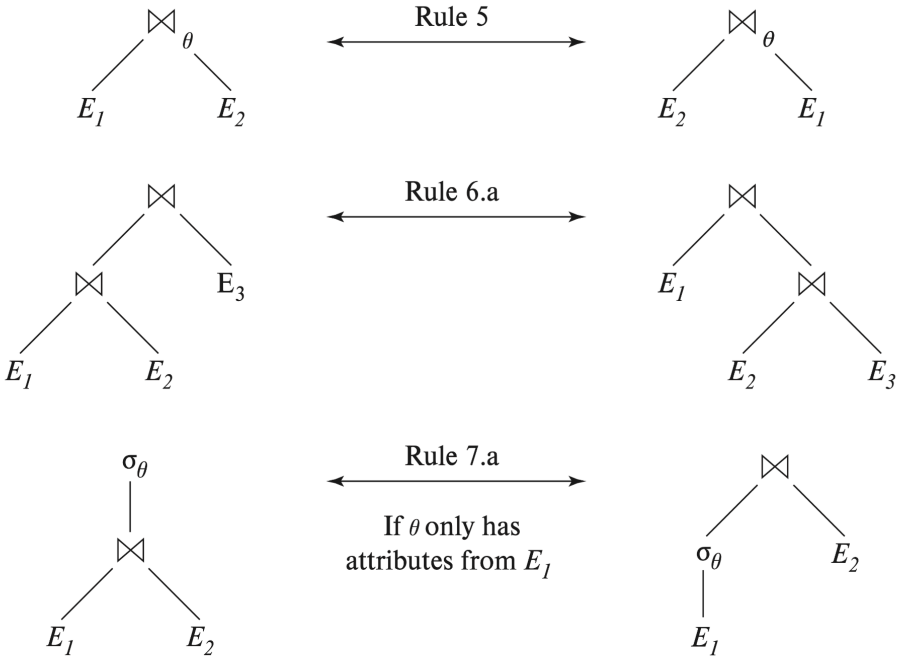


Figure 16.3 Pictorial representation of equivalences.

Example with Multiple Transfromations

- Query: Find the names of all instructors in the Music department who have taught a course in 2017, along with the titles of the courses that they taught

$$\prod_{name,title}(\sigma_{depar\_name='music' \wedge year=2017}(instructors \bowtie (teaches \bowtie \prod_{course\_id,title}(course)))) \tag{29}$$

- Transformation using join associatively (Rule 6a):

$$\prod_{name,title}(\sigma_{depar\_name='music' \wedge year=2017}((instructors \bowtie teaches) \bowtie \prod_{course\_id,title}(course))) \tag{30}$$

- Second form provides an opportunity to apply the "perform selections early" rule, resulting in the subexpression

$$\sigma_{depar\_name='music'}(instructors) \bowtie \sigma_{year=2017}(teaches) \tag{31}$$

> Once again, this is probably the earliest technique that could be optimized if to reorganize. The theory is to push the operation in and out. And for example, intuitively, it seems as a good idea to do the selection first before you do the join. And you might even argue that you'd better do the projection before you do the Join. So these are all a set of what we call risk kind of rules. Nothing has been proven, but it could be seems to work well. So we'll basically rewrite the query based on those intuition.

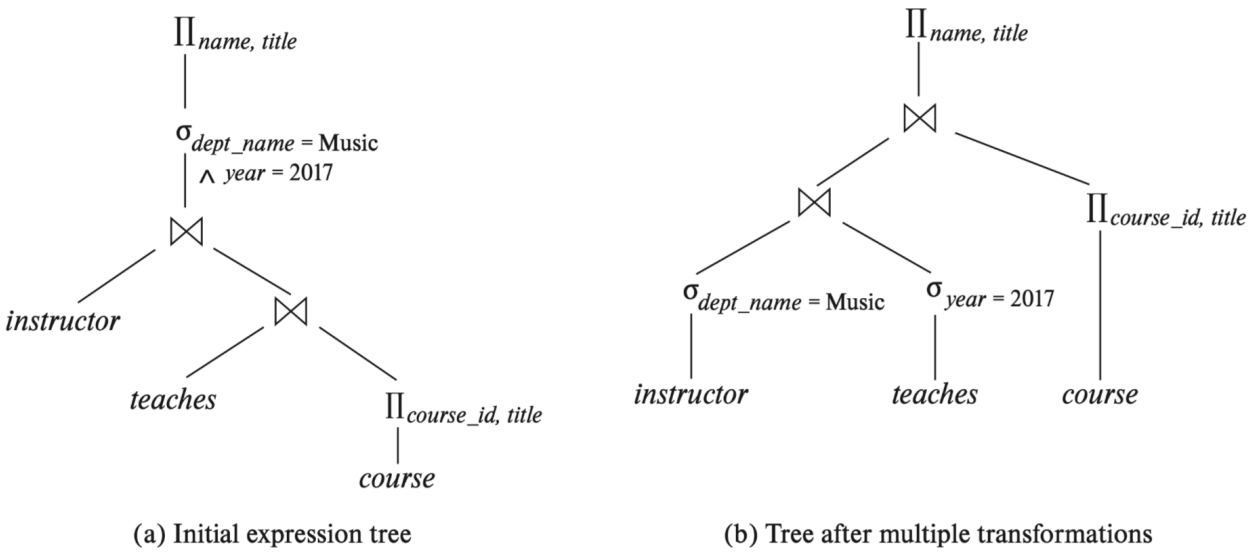


Figure 16.4 Multiple transformations.

> At the very least, we should this transformation need to be done at the very least to produce options we want to accept. So typically what happens is that your algorithm, all your your create your query optimizer.
>
> And the idea is that once you have this tree, then you can do all the transformation and then create extra trees, then the optimizer will then go to work and estimate the cost of how much is needed to execute each of these tree. And execute in time for each of these tree and then figure which trees to best.
>
> In the early system like in the 80s, It will basically say, Hey, eve do this transformation, do this transformation to this transformation but don't do those transformation and just do the transformation and give you a final tree and run it. So it can be using both way. You can use a way to decide how do you want to query? If not, at least we can be a tool for you to generate other options and then you decide which option to do. So there's at least two things you can look at.

Transformation Example: Pushing Projections

- Consider:

$$\prod_{name,title}((\sigma_{depar\_name='music'}(instructor) \bowtie teaches) \bowtie \prod_{course\_id,title}(course)))) \tag{32}$$

- When we compute

$$(\sigma_{dept\_name='Music'}(instructor \bowtie teaches) \tag{33}$$

we obtain a relation whose schema is: `ID, name, dept_name, salary, course_id, sec_id, semester, year`

> How will you describe a attribute that is necessary or is not necessary in this case?
>
> Name and title is necessary because you need it at the very end.
>
> what actually were needed, and what could be useless? And if you know that distinction, then you can push the projection down as long as you need.

- Push projections using equivalence rules 8a and 8b; eliminate unneeded attributes from intermediate results to get:

$$\prod_{name,title}((\prod_{name,course\_id}((\sigma_{depar\_name='music'}(instructor)) \bowtie teaches)) \bowtie \prod_{course\_id,title}(course)) \tag{34}$$

- Performing the projection as early as possible reduces the size of the relation to be joined.

> P754

> if you do projection, your table becomes smaller. And hopefully that will make Join faster to compute.

> Now the good news about sql is what you by default, you do or do not remove the duplicates. You don't have to do any check. It's just linearly reading the table.

Join Ordering Example

- For all relations r_1, r_2 and r_3,

$$(r_1 \bowtie r_2) \bowtie r_3 = r_1 \bowtie (r_2 \bowtie r_3) \tag{35}$$

(Join Associativity) $\bowtie$

- If $r_2 \bowtie r_3$ is quite large and $r_1 \bowtie r_2$ is small, we choose

$$(r_1 \bowtie r_2) \bowtie r_3 \tag{36}$$

so that we compute and store a smaller temporary relation.

> The most challenging thing on a query optimizer is to figure out which join to do first and also figure out which going to do first have a drastic impact on the possible speed of the query.
>
> So once again you have to estimate the result of join. You don't know the result that John is executed, you don't know how many tuples will return when you execute it. but you really need to know how many tuples you're going to get before you choose how to execute. Let's get back to the chicken and egg problem. It's always a chicken and egg problem.

- Consider the expression

$$\prod_{name,title}((\sigma_{depar\_name='music'}(instructor) \bowtie teaches) \bowtie \prod_{course\_id,title}(course))) \tag{37}$$

- Could compute $teaches \bowtie \prod_{course\_id,title}(course)$ first, and join result with $\sigma_{depar\_name='music'}(instructor)$. But the result of the first join is likely to be a large relation
- Only a small fraction of the university's instructors are likely to be from the Music department
    - It is better to compute

$$\sigma_{depar\_name='music'}(instructors) \bowtie teaches \tag{38}$$

      First.

> obviously, this is a human decision, I want to at least give you start to make some human decisions so that you can think about how do we incorporate this human decision into a database application.

Enumeration of Equivalent Expressions

- Query optimizers use equivalence rules to **systematically** generate expressions equivalent to the given expression
- Can generate all equivalent expressions as follows:
    - Repeat
        - apply all applicable equivalence rules on every subexpression of every equivalent expression found so far
        - add newly generated expressions to the set of equivalent expressions

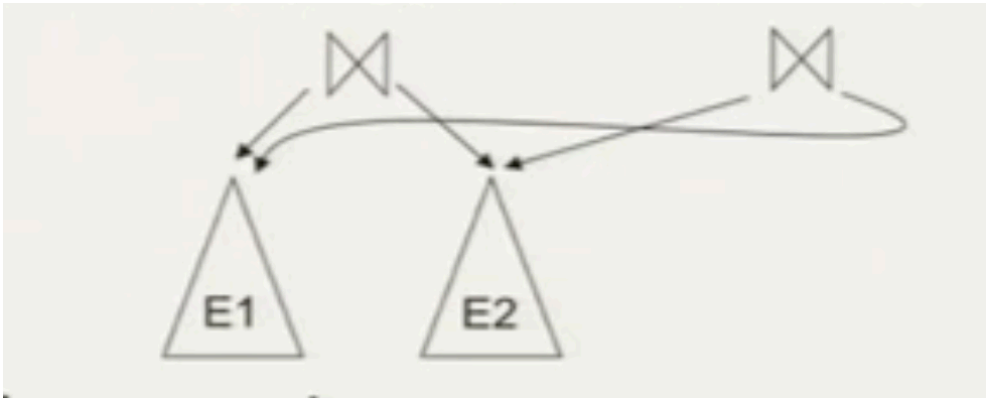
      Unitl no new equivalent expressions are generated above
- The above approach is very expensive in space and time
    - Two approaches
        - Optimized plan generation based on transformation rules
        - Special case approach for queries with only selections, projections and joins

> I started out with a basic query, always to join first and selection the projection. And then they will apply a set of rules and then it will convert those query into a different query and then run it. Nowadays we still do this, but now the goal is different. The goal is to produce a variety of possible way of executing the query and then figure out which way. And then we push it to the next step. The next step would that evaluate each of this query and see which one is the best? That's why. Query optimizers use equivalence rules to **systematically** generate expressions equivalent to the given expression

> In general, you will actually generate several possibilities based on the given. I'm not going to all possibilities. I'm only going to generate some possibilities based on how we can optimize.

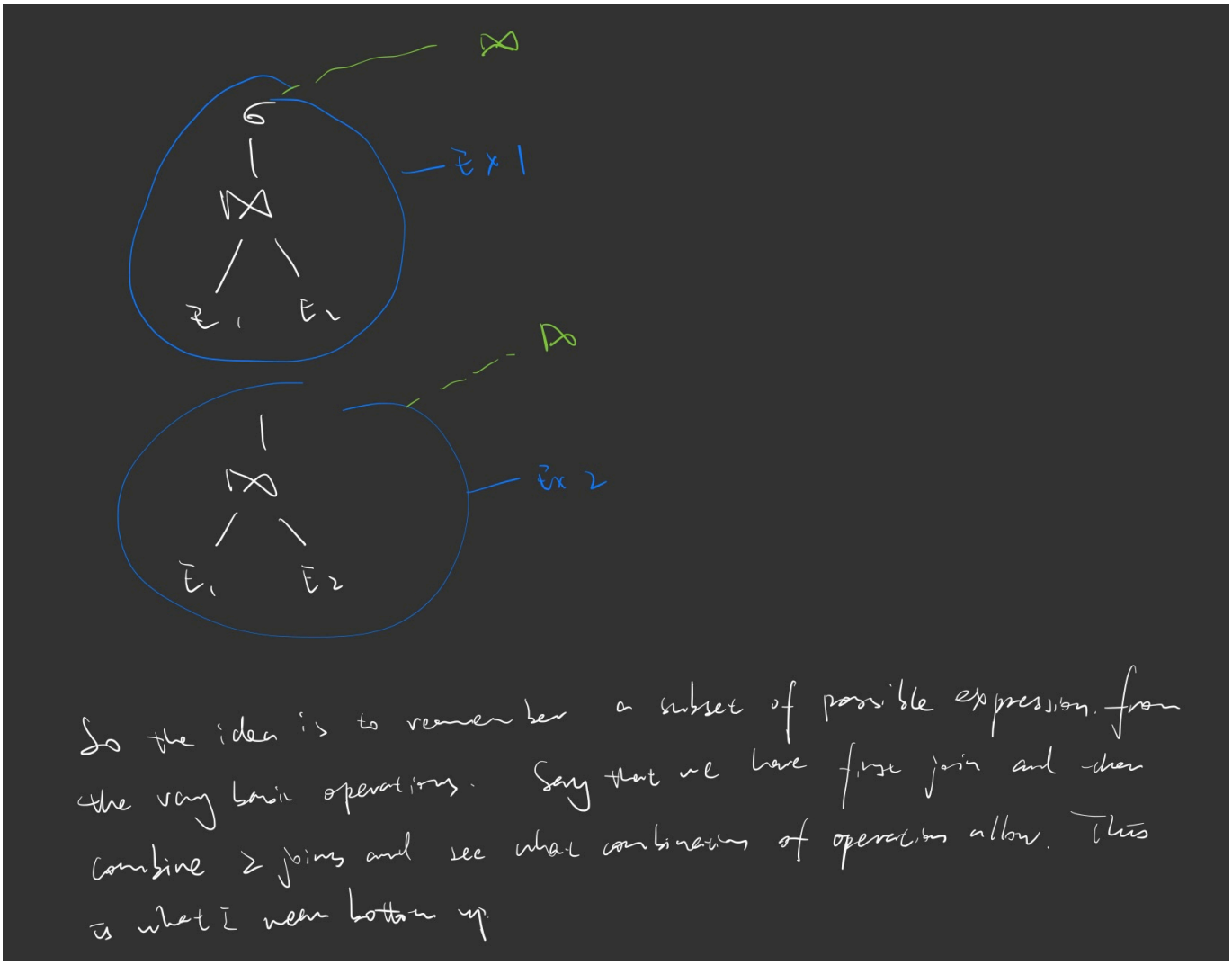
Implementing Transformation Based Optimization

- Space requirements reduced by sharing common subexpressions:
  - When E_1 is generated from E_2 by an equivalence rule, usually only the top level of the two are different, subtrees below are the same and can be shared using pointers
    - E.g. when applying join commutativity

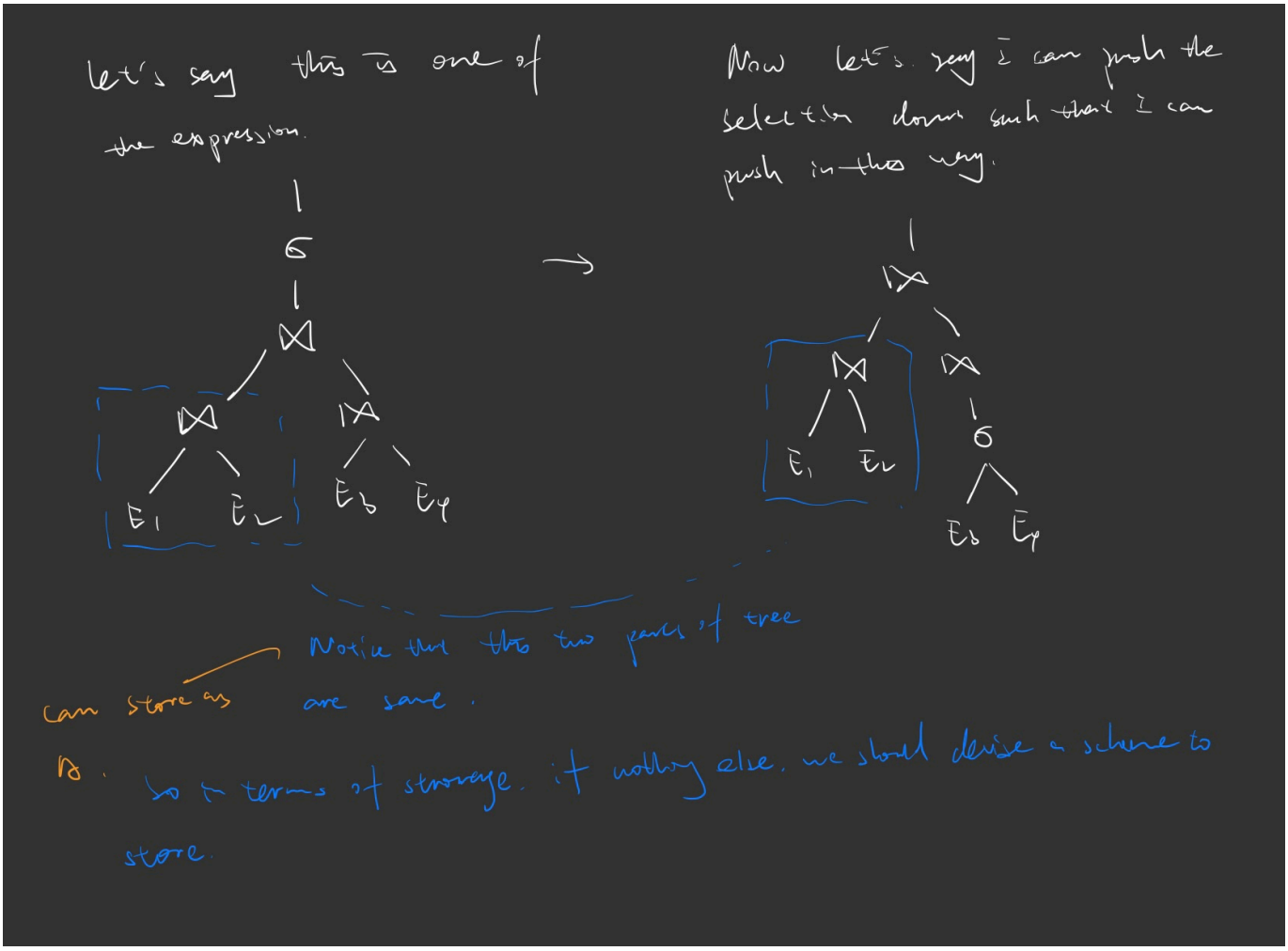


  - Same sub-expression may get generated multiple times
    - Delect duplicate sub-expressions and share one copy
- Time requirements are reduced by not generating all expressions
  - Dynamic programming is an option

> When we cover about generation, we talk about basically starting with a tree and generating as many as possible. It turns out there is some merit of doing things bottom up. Now, when I say bottom up, what do you think I mean here? Starting from leaves.



The idea here, if you want to have the self beginning of big picture, we want to build things from below. Even though we may have a lot of expression, many of them are really similar now. What do I mean?



Decorating parse trees

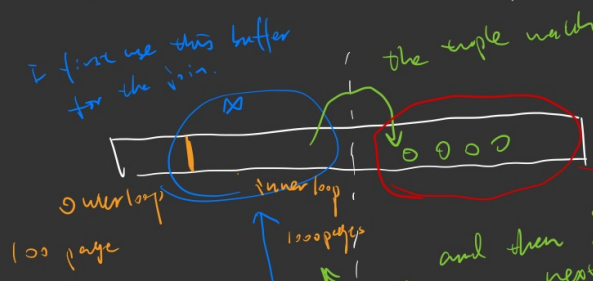
- Once the parse trees are build, need to decorate them with algorithm for each step
- Start with looking at the possible algorithms for each operators
- Other consideration
    - Materializaiton vs. Pipelining
    - Combining operators.

> So far we talk about all the transformation. And the goal of the transformation is to generate a bunch of potential path tree. so that we can choose which one is the fastest. Once we have a parse tree, then what we need to do next is what we called that create in the parse tree. So if there's a joint, we need to figure out whether we want to do like sort merge, nested loop. If you are selection. We do figure out whether we should use an index. If so, which index to use and which algorithm which we want to run. If we do projection, we want to know whether do we need to sort anything beforehand, so and so forth. That's so far what we talk about. It turns out there is one more thing we need to consider, and it's the notion of what we call materialization and pipelining.

We have a lot of clothes. And you only have one washer and one drier. What do you do? You probably put the first load in the washer, wash it, put the first thing in the dryer, dry it forward, and then put the second load in the washer, wash it. Then they put a second low in the dryer, dry it ... You don't have to wait for the first load to finish drying before you put the second low in the washer. This is a very, very basic form of pipeline. what do I mean by that? You have to clean your clothes. You have to do two operations, right? You have to wash it and then dry it. Each of them is independent of one another. And when you're done trying, you don't go back and watch it. So what does it mean? That means I can't. When I send the first load to wash after the first load is done washing, I immediately send the first load to dry. But the watcher at that time is free. I can actually send a second load to wash. That's what pipelining. You can actually be more efficient. Why do I talk about all this in this database?

I'm going to read only a portion of the table. we can pick out a loop and then we wrap that in the loop and we can't just we might not be enough to read in the loop in one shot. We divide it in a bunch of shots. So every time the buffer is full, we are going to join. We are doing the match tuples. And the idea is what once the tuple is match, we can immediately go to do selection on them. We don't have to write the temporary result into the this and then bring it back in to select.

Let's have some buffer.



I'm going to cut the buffer into two halves. So every time I do a join. I will leave that part of buffer.

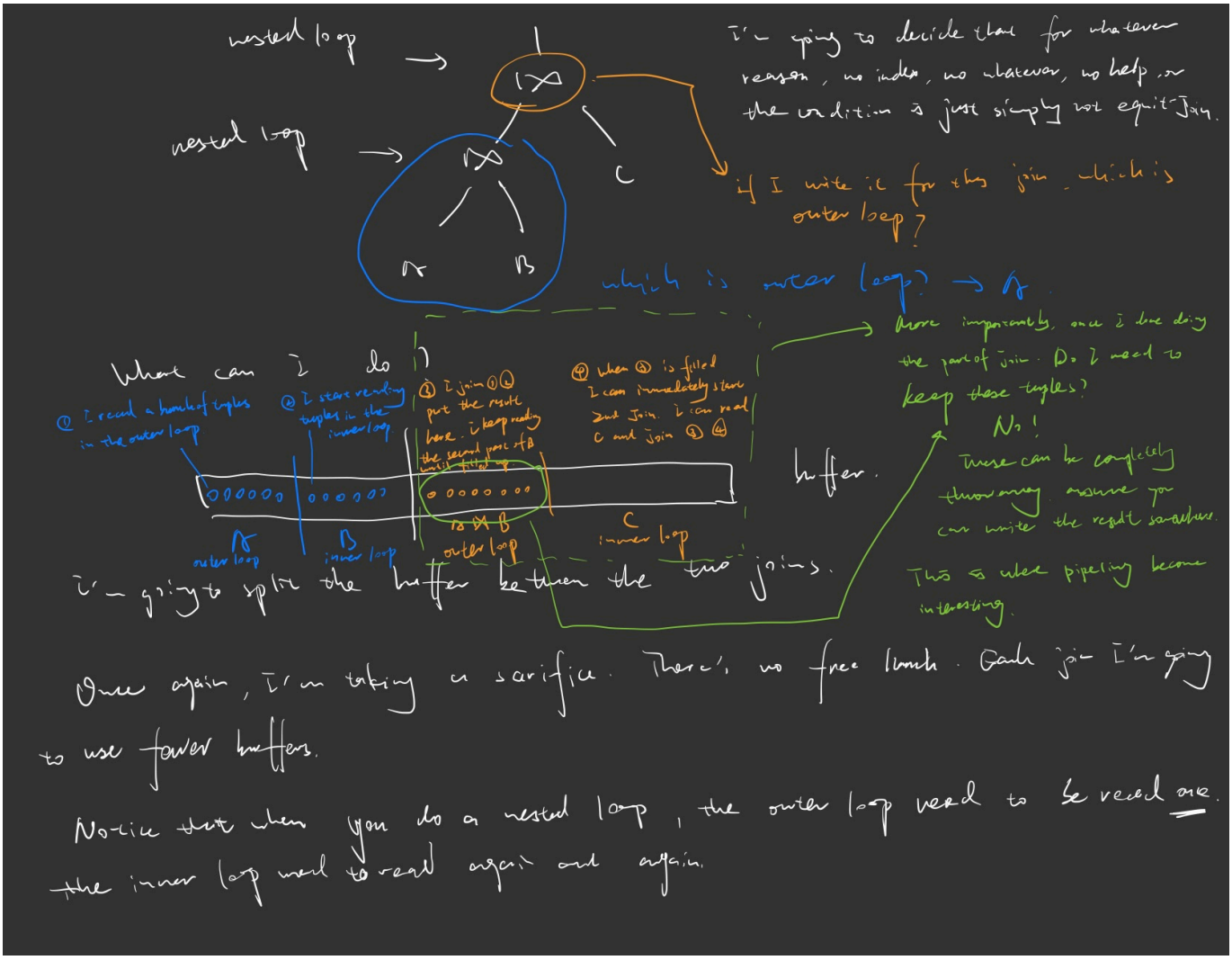
① There's what I mean by pipeling. You don't need to wait for the join to finish before you start the selection.

② There's no free lunch. We now have fewer buffers do the join. so the join will cost longer time.

However, once I match a pair of tuple immediately can throw it to the second part of table and do the selection. I do not have to write the temporary result back to this. Why do I care about this? Because you don't know how big your join is.

Let's say the first table is 40 TB. second is 30 TB. It may total your join to return 40 TB. That means if I do the two thing separately. I need to do the join. write the result back to this. and then read the result back to them to do the selection. That's not fun. → Now I can avoid writing the result immediately.

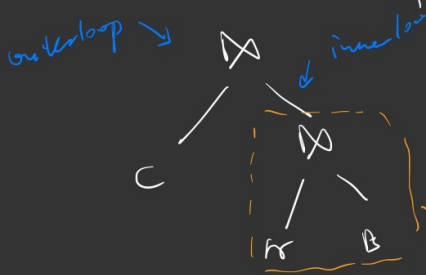
If just 2 tuple join, it can not mix so large. But large number of join. Mix Huge.



Once again, I'm taking a sacrifice. There's no free lunch. Each join I'm going to use fewer buffers.

Notice that when you do a nested loop, the outer loop need to be read one. the inner loop need to read again and again.

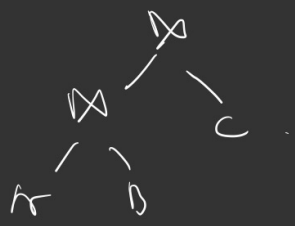
You have to be careful, when you do the pipeline. What do I mean?



Can I pipeline here?

Remember when you do a nested loop join, the outer loop need only to be read once, but the inner loop they could be read many times. That means in this case you have no choice but to save the intermediate result.

Now if you think that for a few minutes, brings you a conundrum, the larger table should be inner loop, right, we talk about the smaller table should be outer loop, However, for the second join, if the further out of join is larger table, you'd have to put the inner loop, but now you have to save in the inner loop, before you can read it again.



Even B is large, I'm still going to put it in the outer loop. What do I gain? I gained that I do not have to write the result $\bowtie$ B, because I can pipeline all the way to the end.

So now things become really interesting. Which table should be in the outer loop become a more interesting question than when we look at the single join. whether we want to sacrifice the potential advantage of pipeline. We really have to push the numbers to calculate. And once again, we all go back. We don't have the numbers to punch until we execute a query.

Materialization vs. Pipelining

- Consider the following query

$$\sigma_{gpa>3.0}(Student \bowtie Department) \tag{39}$$

- We can push selection in

$$\sigma_{gpa>3.0}(Student) \bowtie Department \tag{40}$$

> Now obviously based on the discussion of earlier, we can push the selection down, we can do the selection, then we do the nested loop.

- suppose we use nested loop
  - Either table can be in the inner loop

> Let's assume the student is in the inner loop.

$$\sigma_{gpa>3.0}(Student) \bowtie Department \tag{41}$$

- Now suppose Student is in the inner loop

> That means that every time you read a page in the department table, I have to read the whole result inner loop.

- We first apply the selection: $\sigma_{gpa>3.0}(Student)$
- However, since that is in the inner loop, it will have to be read **multiple times**
- So the result of the selection need to be writing to secondary storage -- **materialization**

> So the result of $\sigma_{gpa>3.0}(Student)$ you actually have to store in the disk because you have to read it multiple time.
>
> | **Department** | $\sigma_{gpa>3.0}(Student)$ |
> |---|---|
>
> Department is outer loop, Selection of student is in inner loop. So you read a few page of department. Then you have to read the whole table of $\sigma_{gpa>3.0}(Student)$, and join. And then you read the second page of department. Once again, I have to read the whole result to continue the join process.

> materialization that means the result of the first query have to materialize. Have you have to actually store the result onto the this that's what we call materialization.

> But if a stuent is in the outer loop, what happened?

$$\sigma_{gpa>3.0}(Student) \bowtie Department \tag{42}$$

- Now suppose Student is in the outer loop
- We first apply the $\sigma_{gpa>3.0}(Student)$

- However, since that is in the outer loop, it will have to be read only once
- So we can apply the following algorithm
    - Allocate some amount of main memory buffers for the result of the selection

      (the amount one allocate for the outer loop of the join)
    - Start executing the selection
    - Pause when the buffer is filled up
    - Then execute the join for those tuples in the buffers only
        - Notice that involve reading the inner loop (Department) once
    - After that is done, those tuples in the buffer from teh selection **is no longer needed** (why?)

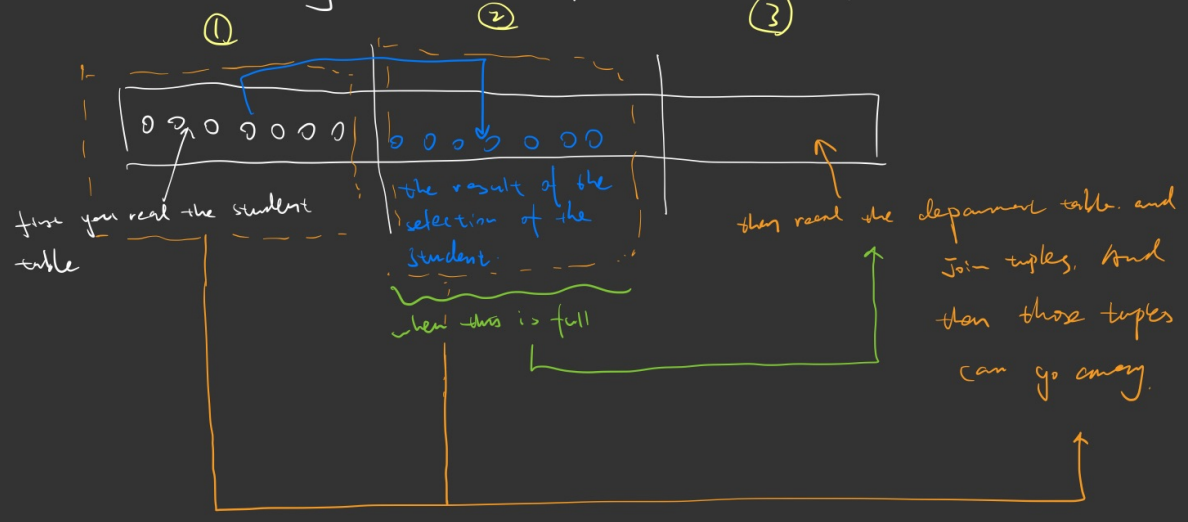
      > Because these is in the outer loop. So we can discard though to resume the selection until the buffer is spill.

    - Discard those tuples, resume the selection until the buffer is filled
    - Repeat the process
- This is known as **pipelining**

$\sigma_{gpa>3.0}(Student)$	**Department**

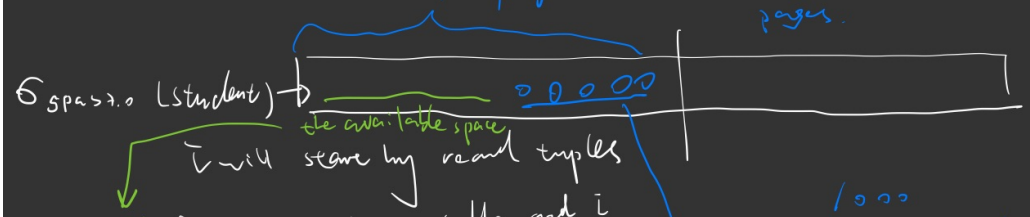
The selection of student and then you read the whole department table, which is already in the stores anyway. You don't do anything with it. You might have to read department multiple times because we don't have enough buffer. But the point is, once I read the department table and to join this, I can discard these tuples. Those tuples will never be needed again. At least in the content in this join.

We'll do essentially divided buffer into three parts



In fact, in this case, I can even cheat a little bit. I can just combine ① ② in to one thing.

Let's say I have 1000 page in student table and the buffe has 10 pages.



I will start by read tuples from student table and I will discuss tuples where is not satisfied the condition. Then I'll keep reading. Some tuples are already satisfied condition. So when I read next batch of pages, I have to read fewer peges.

I keep read the pages and then applied the selection. I can do that. This may not the most efficient way of doing things. In the previous steps, you might have read 10 pages and then 100 times. But now I have to do more. because I can only read less than 10 pages at a time. → You use this to buy less seek.

$\frac{1000}{10} = 100$ times

I have to go through 100 times. But if I after a read. I actually keep some tuples that satisfy the condition.

the available space for me to read the student table start to decrease because I have to keep some tuples for the join process. in next time.

More free. Materialization doesn't come free.

hard drive worth? depend.

SSD worth ✓.

Materialization

- Materialized evaluation is always applicable
- Cost of waiting results to disk and reading them back can be quite high
  - Our cost formulas for operations ignore cost of writing results to disk, so
    - Overall cost = Sum of costs of individual operations + cost of writing intermediate results to disk
- **Double buffering**: use two output buffers for each operation, when one is full write it to disk while the other is getting filled
  - Allows overlap of disk writes with computation and reduces execution time

> One buffer doing the printing, the other buffer return receiving the result of caculation, because the caculation is faster than anything
>
> But nowdays, it's triple, quadruple, quintuple buffer because the CPU becomes so fast. You can insert having one output buffer, you can have multiple output offers while one buffer is actually writing the output, the other is actually receiving the result.

Pipelining

- **Pipelined evaluation**: evaluate several operations simultaneously, passing the results of one operation on to the next.

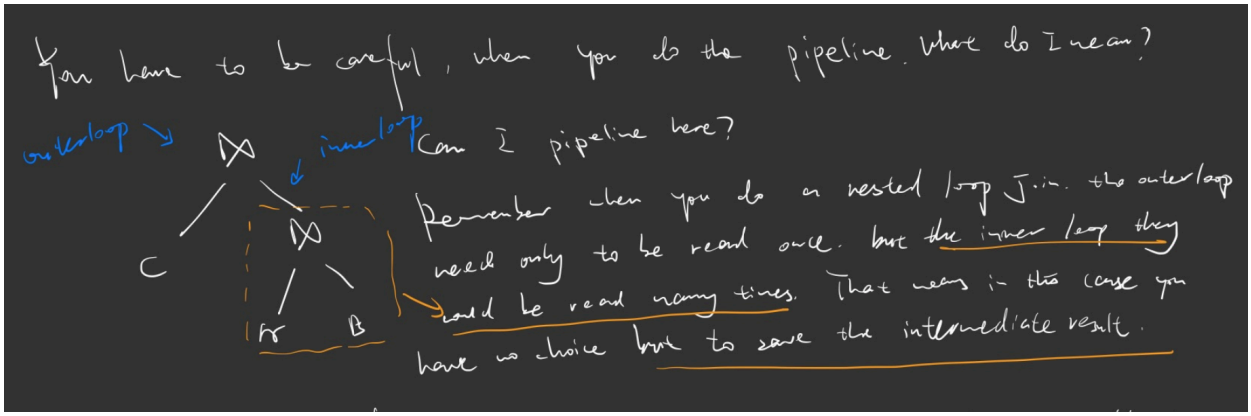
  E.g., In previous expression tree, don't store result of

  $$\sigma_{building='Watson';}(department) \tag{43}$$

  - Instead, pass tuples directly to the join... Similarly, don't strore result of join, pass tuples directly to projection

> Notice that in pipeline we do not wait till the first operating to finish before we start the second. That's where you save the time.

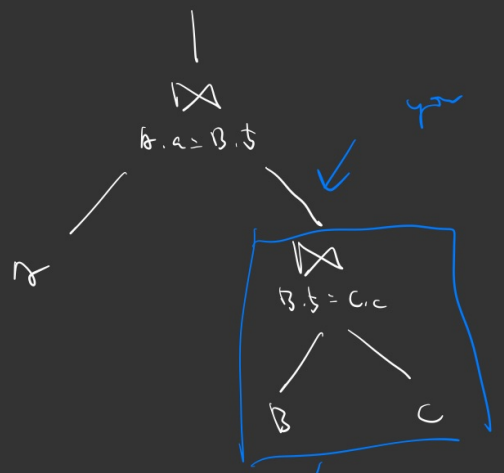
- Much cheaper than materialization: no need to strore a temporary relation to disk.
- Pipelining may not always be possible - e.g., sort, hash-join.
  - Key observation: pipelining is possible if and only if the data that is pipelined into the next operations is read only once for that operation



> Sort-merge: You have to wait till the two tables to be sorted before you can merge. So there's no pipeline you can play.

- For pipelining to be effective, use evaluation algorithms that generate output tuples even as tuples are received for inputs to the operation.
- Pipelines can be executed in two ways: **demand driven** and **producer driven.**

> And as I say now, in this class we are focused on I/O, we don't talk much about CPU at all. But obviously if you have multiple CPU, you can see that you can gain a lot from doing that.



What's so interesting about these Join conditions?

Frankly, the three table are being Join on the same attribute anyway.

But now, if you do the sort merge on BC. What happen?

The result of the first join is already sorted in the attribute you want to Join. Now, pipelines is it worth? That's debatable. Once again, depends on buffers... Because if you do sort-merge, you do not need to support the choice. Because the result of merge is the first step, the tuples were already in the order you want.

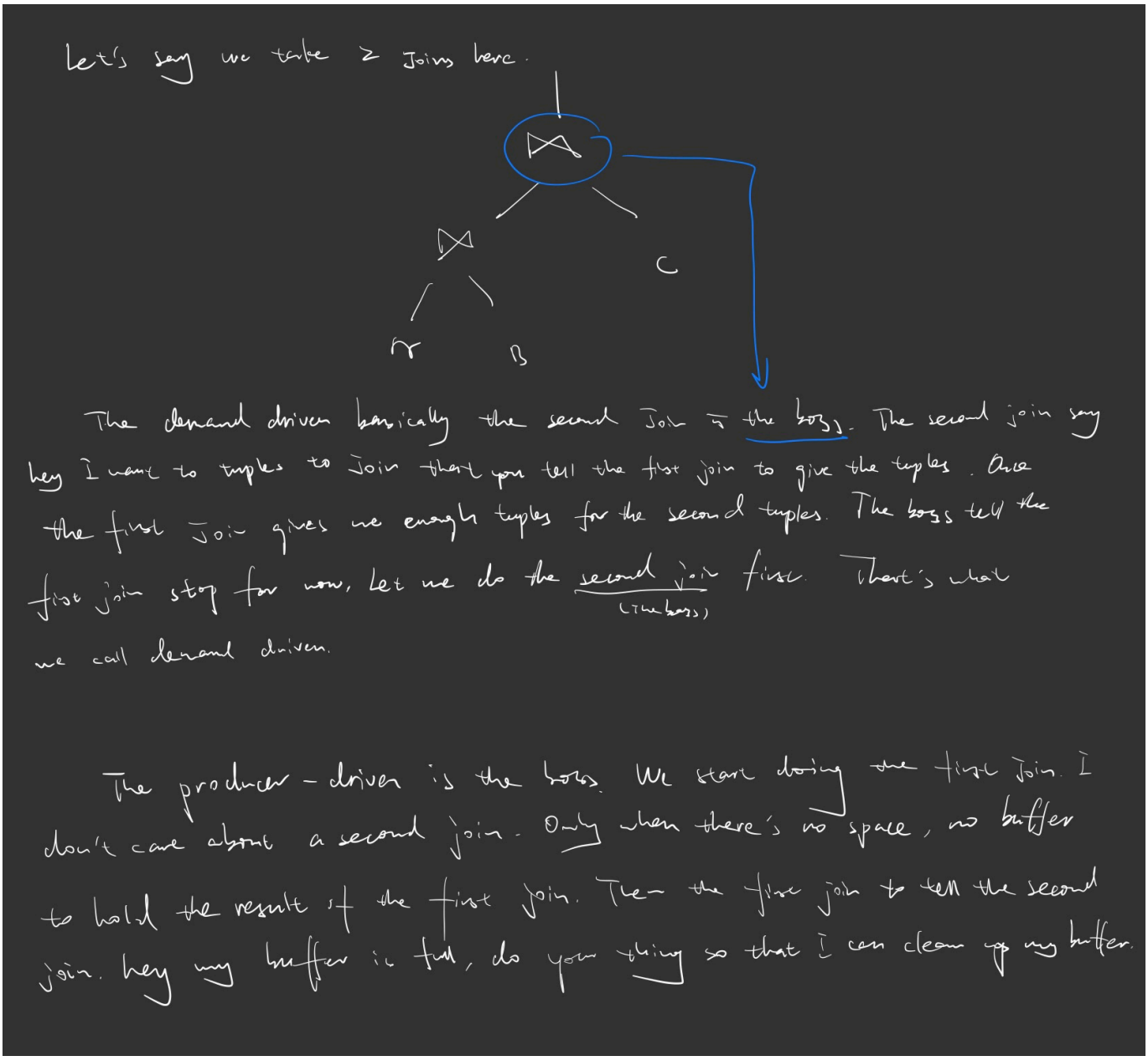
So you can practically pipeline in this sense.

For Join B and C. you do the merge.

Is it worth it? Hard to say. Because you are allocated the buffer for the first Join.

That means you have fewer buffer to sort B. And that way or way not crippling

- In **demand driven** or **lazy** evaluation
  - System repeatedly requests next tuple from top level operation
  - Each operation requests next tuple from children operations as required, in order to output its next tuple
  - In between calls, operation has to maintain "**state**" so it knows what to return next
- In **producer-driven** or **eager** pipelining
  - Operators produce tuples eagerly and pass them up to their parents
    - Buffer maintained between operators, child puts tuples in buffer, parent removes tuples from buffer
    - if buffer is full, child waits till there is space in the buffer, and then generates more tuples
  - System schedules operations that have space in output buffer and can process more input tuples
- Alternative name: **pull** and **push** models of pipelining



> So there are two ways of doing it. if you are actually implementing, say, the next version of Oracle, you do have to worry about this because the different mechanism, they will be slightly different. Your overall architecture might be a bit different because now you have to say, who is the boss who called the shots that do affect how the database system is implemented? For the sake of this class, it doesn't really matter too much because all we case how many of these pages are we going to read? So in that sense, push and pull doesn't really matter too much. However, if you are the one to actually implement things, you do have to worry about, which is the better way. Once again, there's no definite answer.

- Implementation of demand-driven pipelining
  - Each operation is implemented as an **iterator** implementing the following operations
    - `open()`

      > You basically just start an operator

      - E.g., file scan: initialize file scan
        - state: ponter to beginning of file
      - E.g., merge join: sort relations;
        - state: pointers to beginning of sorted relations
    - `next()`

      > Then you need to fetch for the next tuples. It may be the next tuple you just doing a scan. If your operator is a Join and nested give me the next result.

      - E.g., for file scan: Output next tuple, and advance and store file pointer
      - E.g., for merge join: continue with merge from earlier state till next output tuple is found. Save pointers as iterator state.
    - `close()`

      > Because when I open, I have to allocate some buffers. In main memory, so I better be a nice boy and clean it up. Otherwise, running our buffer very soon.

> So once again, this is going very deep into implementation issues.So I don't want I don't want to go too much. But the iterator structure, the architecture will allow you to do these things. You really should think in terms of these things. If you operate each operation like a join, a selection, a projection should come when you implement this function, you should have a say, a open function. That means you start the reading the file. Then you should have a next operation that returns the next tuples.

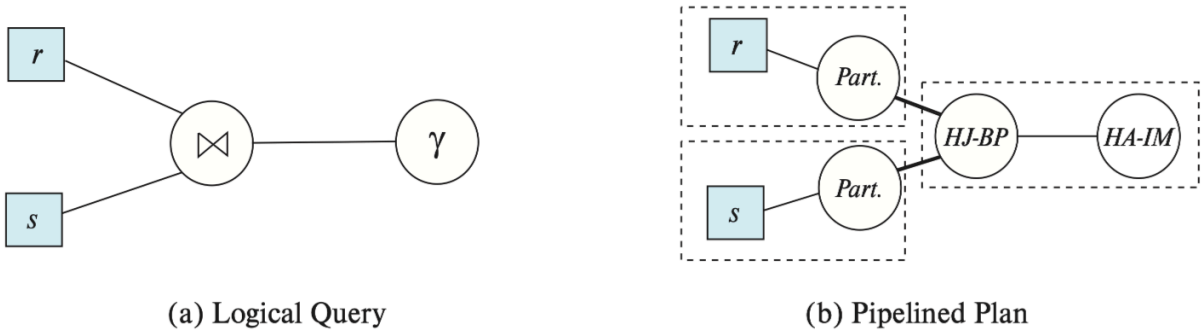
> What is iteration in Java?
>
> https://www.runoob.com/java/java-iterator.html
>
> Why do Java create iterator?
>
> 在程序开发中，经常需要遍历集合中的所有元素
>
> What iterator provide you ?
>
> A: You can call for next. And then so on so forth.

> So a pipeline can also be thinking of as a iterator in the most generic sense. If you do a pipelining operation, each operation should be a thing of an iterator.

> Java garbage collection?
>
> Basically where you left the things behind, mom come out and clean up, this garbage collection. Somebody have to Garbage collection. It's not you, then it's actually your mother.

Blocking Operations

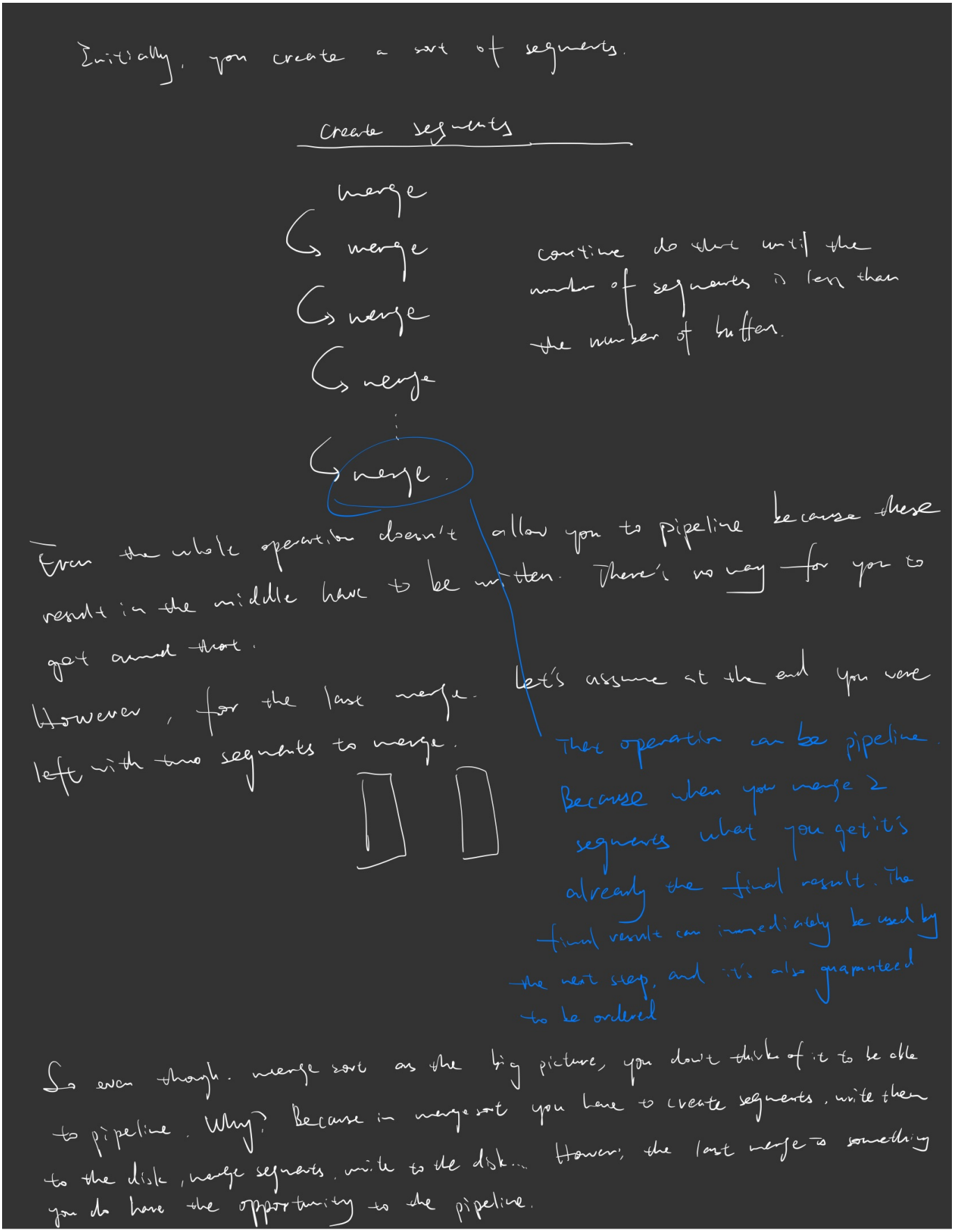
- **Blocking operations**: cannot generate any output until all input is consumed
  - E.g., sorting, aggregation, ...
- But can often consume inputs from a pipeline, or produce outputs to a pipeline
- Key idea: blocking operations often have two suboperations
  - E.g, for sort: run generation (during sort) and merge
  - For hash join: partitioning and build-probe(during the final nested loop)
- Treat them as separate operations



(a) Logical Query (b) Pipelined Plan

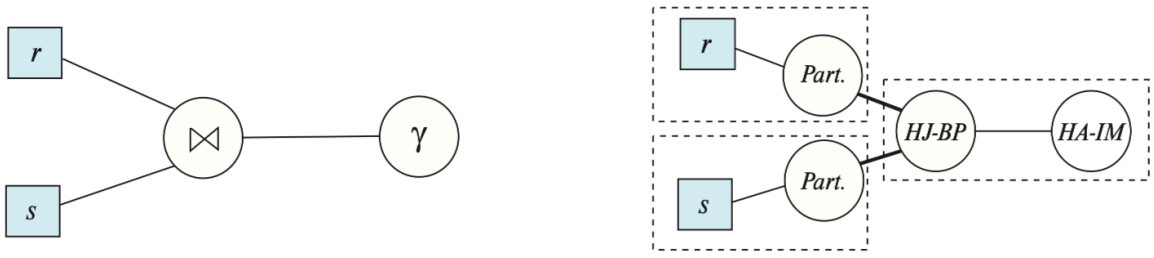
Figure 15.12 Query plan with pipelining.

> Let's go to the merge sort as a further explanation. So what do you mean by blocking operation? Let's say I do a join. If you just take a look on the surface, you really can't do anything until the whole thing is sorted.
>
> Remeber how merge sort work?
>
> Initially you create a sort of segment, and then you merge. Smallest to a bigger list. You continue to merge bigger list to even bigger lists. You continue do that until the number of segments is less than the number of buffers. Then you can emerge temple. Now, even though the whole operation does not allow you to pipeline because obviously these result in the middle have to be written to do this. There's no way for you to get around that.



So sometimes even though an operation on surface cannot be pipeline, it is certainly possible part of it could be pipeline. And that allow you to have a kind of what we call a blocking operation.

- Pipeline stages:
  - All operations in a stage run concurrently
  - A stage can start only after preceding stages have completed execution



(a) Logical Query (b) Pipelined Plan

Figure 15.12 Query plan with pipelining.

Materialization vs. Pipelining: Cost estimation

$$\sigma_{gpa>3.0}(Student) \bowtie Department \tag{44}$$

- Suppose
  - 1000 pages for Department
  - 5000 pages for Student
  - $\sigma_{gpa>3.0}(Student)$ return 2500 pages

    > In this case, we have to do this selection, so we have to write the result

  - Suppose 100 buffers, split 50 each
- Consider Student in the inner loop (no pipeline)

$$\begin{aligned} Cost(page\ read) &= 5000(read\ Student\ for\ \sigma) \\ &+2500(write\ Student\ for\ \sigma) \\ &+1000(outer\ loop\ for\ Department) \\ &+2500 \times \frac{1000}{50}(inner\ loop\ for\ \sigma_{gpa>3.0}(Student)) \\ &= 58500 \end{aligned} \tag{45}$$

> So let's say we do it on a solid state drive. So no seek.

> If student is outer loop

$$\sigma_{gpa>3.0}(Student) \bowtie Department \tag{46}$$

- Suppose
  - 1000 pages for Department
  - 5000 pages for Student
  - $\sigma_{gpa>3.0}(Student)$ return 2500 pages
  - Suppose 100 buffers, split 50 each
- Consider Student in the outer loop (no pipeline)

$$\begin{aligned} Cost(page\ read) &= 5000(read\ Student\ for\ \sigma, outerloop) \\ &+1000 \times \frac{2500}{50}(inner\ loop\ for\ Department) \\ &= 55000 \end{aligned} \tag{47}$$

> So the total cost is now a little bit smaller, not a lot. But still, if every microseconds count, this is something to think about. in this case, I'm violating my own rule to put the larger table in the outer loop. But we can do that because pipelines basically eliminate the costs of leading to the end of outer loop once again, because you pay that price when you do the initial reading. It may be worthy to put a larger table in the outer loop in this case. Once again, that's what make career optimization complicated and fascinating. There is a lot of things to consider.

> Why do we need to write to the department?
>
> A: Because I haven't touched the department. They define this already in the database. I didn't do anything. I just simply have to read the whole table. I don't do any selection. I didn't find any result.

> If it is hard drive, we have to worry about these things. Things can get dicier. But it is at least something you should actually consider, or at least a career optimizer absolutely should consider. And if you are building a database system, it is actually something that you have to do architecture. Once again, I'm talking about it in the context of database. There might be some other system that you build, even though it's not a full fledged database, have some of these capability. So keep that in mind. You might use it in some other context when you're developing something itself.

Index-only query

- Consider:

$$\sigma_{gpa>3.0}(Student) \tag{48}$$

- Now suppose we have a secondary index on gpa
  - Probably not useful if a lot of tuples satisfies the query
- However, consider

$$\pi_{gpa}(\sigma_{gpa>3.0}(Student)) \tag{49}$$

  - Now only the gpa attribute is needed
  - All the information needed is in the secodary index
    - No need to go to the main table
    - Efficient regardless of number of tuples retrieved

> All I need is gap, and the gap is in the index, why should I spend the time looking at tuples.

- This is very useful for multi-attribute indices
  - Index where keys are a combination of attributes (att1, att2, att3, ...)
  - Order is based on att1, if tied then att2, if tied again att3 etc.

> if you think that people will actually ask this query a lot.
>
> $\Pi_{name,ssn}(\sigma_{gpa>?}(Student))$
>
> That it is absolutely worthy to create a index with a gpa followed by ssn. Because, hey, for this query, if I have that index, I don't even need to touch the table. I can get everything I need from the Index. Now, obviously, there is no free lunch.

Number of parse tree/plans

- Each decorated parse tree is known as a plan
- The number of plans can grow very quickly
- E.g. a query with a N-way joins can have $(n-1)!$ ways to order it. (why?)
  - This does not even consider the various option (e.g. outer vs. inner loops)

> We have four tables: A_1, A_2, A_3, A_4 and do the join
>
> $4 \times 3 \times 2 \times 1$
>
> So essentially I can order them whatever way they want and I wouldn't even consider the associated mass number.
>
> I won't bore you with the combinatorics, but if you want to calculate the number of combination that you can have, I'll spare you the exponential and worth. And you, as I say, write the key thing about credit optimization. I don't want to spend 2 hours optimizing a query such like a safe one sec. So examining all the trees is just flat out impossible.

- Thus we need to limit the number of parse tree generated
- Also after chosen a parse tree one will need to find the best way to decorate the tree.

> We look at comparing a feeling of parse tree. But remember what one of the main problem, the theory optimizer, is There's a whole lot of parse tree. Very many parse tree through. exponential parse tree. I'm dead. If all you have is selection and projection, there's only so many things you can do. If it is a single table query. There's really nothing much to talk about. The the main reason why you have a lot of pastry are joins, because those accommodative jobs are associated.

- Must consider the interaction of evalution techniques when choosing evaluation plans
  - choosing the cheapest algorithm for each operation independently may not yield best overall algorithm.

    E.g.
    - Sort-Merge-join may be costlier than hash-join, but may provide a sorted output which reduces the cost for an outer level aggregation.

      > sort merge you had to sort both tables, hash join you just only need one not the table to be small enough however sort merge do return the tuples in sorted order. And that sort of order may be invaluable in cutting down the next operation.
      >
      > If nothing else, what if your query have `order by` join condition. Then sort merge suddenly become ten times more attractive. Because if I do sort merge here, I don't have to do sort at the end. So that means greedy algorithm doesn't quite work. You cannot say, let's look at each operation, find the most greedy solution for that and then go on from there.

    - Nested-loop join may provide opportunity for pipelining

> And to make things even worse, maybe I can be greedy. Let's say for each operation, I'll take the most efficient method and then try to combine them. Maybe things are easy. But even that, you have to be careful.

Cost-Based Optimization

- Consider finding the best join-order for $r_1 \bowtie r_2 \bowtie \ldots \bowtie r_n$
- There are $\frac{(2(n-1))!}{(n-1)!}$ different join orders for above expression. With $n = 7$, the number is 665280, with $n = 10$, the number is greater than 176 billion!
- No need to generate all the join orders. Using dynamic programming, the least-cost join order for any subset of $r_1, r_2, \ldots r_n$ is computed only once and stored for future use.

> Number one. This to some degree is a dynamic programing problem. Let's say you have a bunch of matrices. And you want to multiply them.
>
> $A_1 \times A_2 \times A_3 \times A_4$
>
> Matrix multiplication is not commutative. However it is associated. That means I don't care which modification you do first, the result will still be the same. However, if you do it in a different order, the total number of operation can change drastically.
>
> And you have to find the best way of doing a multiplication. And you you can resort to direct programing. Here doing a join have some flavor of this not quite exactly like that. But you can also resort to some form of dynamic programing to basically calculate the best course of each of the joins.

> Why this is not fully applicable because in the major lubrication problem, making modification is not commutative.
>
> Here. The join is actually commutative. So you can actually write this $r_2 \bowtie r_1 \ldots$ It was to give you the correct result. So that will actually add more pressure to the dynamic program.

Dynamic Programming in Optimization

- To find best join tree for a set of n relations:
  - To find best plan for a set S of n relations, consider all possible plans of the form: $S_1 \bowtie (S - S_1)$ where S_1 is any non-empty subset of S.
  - Recursively compute costs for joining subsets of S to find the cost of each plan. Choose the cheapest of $2^n - 2$ alternatives.
  - Base case for recursion: single relation access plan
    - Apply all selections on R_1 using best choice of indices on R_1
  - When plan for any subset is computed, store it and reuse it when it is required again, instead of recomputing it.
    - Dynamic programming

> So the key idea is this. If you have a big Join. $S_1 \bowtie S_2 \bowtie S_3 \bowtie S_4$. What you really want to do is that start from doing one join, calculate the cost, caculate back to one join. Then you combine the two one join into a join into two joins and based on the result of the one join, see you can get a better reuslt for the two joins. either resorting to the same principle or use that algorithm, like sort merge. sort merge not be the best in a single operation, but the two johns are actually join, all the same attributes a sudden become attractive. So we can do that and then we build. The key idea is that we keep track of the best plan for each smaller subset.
> Now, obviously, there's a limitations. If you have n joins, How many possible subsets? So just the total number of subset, it's explanation. So we can't quite pull the same trick totally. We still have to have some limitations. So we won't go into detail.

Join Order Optimization Algorithm

```
procedure findbestplan(S)
  if (bestplan[S].cost ≠ ∞)
    return bestplan[S]
    // else bestplan[S] has not been computed earlier, compute it now
  if (S contains only 1 relation)
    set bestplan[S].plan and bestplan[S].cost based on the best way of accesing S using
selections on S and          indices (if any) on S
  else for each non-empty subset S1 of S such that S1 ≠ S
    P1 = findbestplan(S1)
```

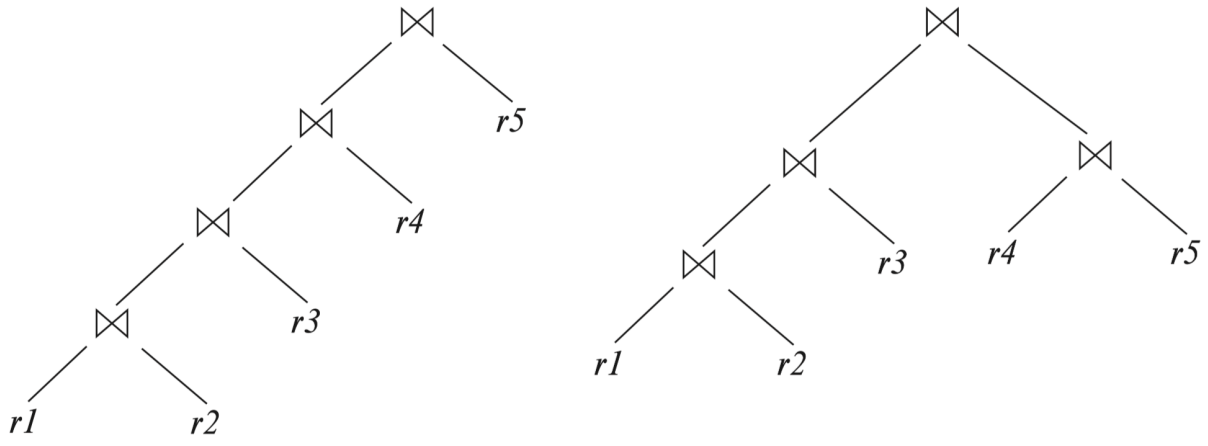
```
    P2 = findbestplan(S-S1)
    for each algorithm A for joining results of P1 and P2
      // For indexed-nested loops join, the outer could be P1 or P2
      // Similarly for hash-join, the build relation could be P1 or P2
      // We assume the alternatives are considered as separate algorithms
        if algorithm A is indexed nested loops
          Let Pi and P0 denote inner and outer inputs
          if Pi has a single relation ri and ri has an index on the join attribute
            plan = "execute P0.plan; join results of p0 and ri using A",
                    with any selection conditions on Pi performed as part of the join condition
            cost = P0.cost + cost of A
          else cost = ∞; /* cannot use indexed nested loops join*/
        else
          plan = "execute P1.plan; execute P2.plan;
                  join results of P1 and P2 using A;"
          cost = P1.cost + P2.cost + cost of A
        if cost < bestplan[S].cost
          bestplan[S].cost = cost
          bestplan[S].plan = plan;
  return bestplan[S]
```

> So they list some kind of recursive algorithm. But you really should do it in dynamic programming.

> Most database system use the following trick to limit the number of clicks, which is something what we call a deep join tree.

Left Deep Join Trees

- In **left-deep join trees**, the right-hand-side input for each join is a relation, not the result of an intermediate join.



(a) Left-deep join tree (b) Non-left-deep join tree

Figure 16.8 Left-deep join trees.

> Anything peculiar about the tree on the left?

> Every right child will actually go directly to a leaf. There's no subtree. Why do we do that? Because if we are lucky, if that's the right algorithm, we can pipe by all the way from beginning to end. And we are seeing that they can potentially be very beneficial. So why not do this? Now, it certainly cut down on the number of plans because you restricted, you know,
>
> A permutation you can permit. But at least there's a lot of other things you have to work out.
>
> $r_1, r_2 \ldots . r_5$

> So many database system even nowadays will cheat in the sense that I only consider lefty trees unless I really only join in two or three tables, then I would consider more things. But if I have join six seven tables, I would then only consider that the definition. And as you say, the main advantage of left-deep tree is that you can pipeline to the very top if you if you choose to. Once again does doesn't automatically do that. It may not be worth it, but at least keep the options of doing it.

Cost of Optimization

- With dynamic programming time complexity of optimization with bushy tree is $O(3^n)$
  - With $n = 10$, this number is 59000 instead of 176 billion!
- Space complexity is $O(2^n)$
- To find best left-deep join tree for a set of n relations:
  - Consider n alternatives with one relation as right-hand side input and the other relations as left-hand side input.
  - Modify optimization algorithm:
    - Replace "for each non-empty subset S1 of S such that S1 $\neq$ S"
    - By: for each relation r in S

      let S1 = S - r.
- If only left-deep trees are considered, time complexity of finding best join order is $O(n2^n)$
  - Space complexity remains at $O(2^n)$

    > Remeber $O(2^n) <\neq O(3^n)$ This constant actually matters.
    >
    > If you can change $O(3^n)$ to $O(2^n)$, you save a lot.
- Cost-based optimization is expensive, but worthwhile for queries on large datasets (typical queries have small n, generally < 10)

  > But if your table is really, really huge, you had tons of tuples.
  >
  > So things can get very dicey if you're not careful. So that's why even though it can be expensive, it's worth it if you have a large table. And obviously, if I'm a database company, I want to sell it to Wal-Mart Mart.

> So if you use dynamic programing and you allow any kind of trees, the time compresses all the free to up it.

> We'll go through it relatively fast until we react, until we answer your question D self-styled relatively fast. I will ask question on the quiz. I'll probably won't ask question on the exam.

Interesting Sort Orders

- Consider the expression $(r_1 \bowtie r_2) \bowtie r_3$ (with A as common attribute)
- An **interesting sort order** is a particular sort order of tuples that could make a later operation (join / group by / order by) cheaper

  > $A \bowtie_{A.a=B.b} B \bowtie_{B.b=C.c} C$
  >
  > Sorting these table in B.b maybe worthwhile because it may help you save time for doing this join.
  >
  > $A \bowtie_{A.a=B.b} B \bowtie_{B.c=C.c} C \bowtie_{B.b=d.k} D$
  >
  > Maybe it's not. Maybe just one step, but this is still a potentially interesting order. because B.b can use in B.b=d.k
  >
  > So if you do it optimization every time, one thing you do want to remember is whether the my result is sorted or not. And sorted on which attribute. You can only sort your result on one attribute. You can not sort a multiple attribute. So storing that information is cheap. But that can be helpful significantly down the line like.

  - Using merge-join to compute $r_1 \bowtie r_2$ may be costlier than hash join but generates result sorted on A
  - Which in turn may make merge-join with r_3 and minimizing overall cost
- Not sufficient to find the best join order for each subset of the set of n given relations
  - must find the best join order for each subset, for each interesting sort order
  - Simple extension of earlier dynamic programming algorithsm
  - Usually, number of interesting orders is quite small and doesn't afftect time/space complexity significantly

> But now make things more challenge is that $A \bowtie_{A.a=B.b} B \bowtie_{B.k=c.d} C$
>
> At the very least I can sort my result value on $A.a$ or $B.k$. This is the only thing we do that doesn't seems to be worth it.
>
> But if you sql has an `order by` query. That might be change the story. If your sql have `group by B.k` Now you should really think about joining them because the group by operation was suddenly become much cheap. That's why you have to look at Join order to see how.
>
> To make things more interesting $A \bowtie_{A.a=B.b} B \bowtie_{B.c=C.k} C$ and in your sql query `select distinct a.k` Now this order `A.k` become an interesting order. Because we have to remove duplicates. That means you either have to sort the result, or have to hash the result. So that become another important, interesting sort order. And if you really, really want to do a good job optimizing, you really have to consider. If I want to maintain my results that we sort in a certain order, I want to at least keep track. And you want to maintain this sort of thing in certain order. That means, for example, you may not be able to use an index. That means you are paying a price as somewhat a step, but I'm paying the price to remain in the right order so I can do better in the next step. And that's the question a great optimizer has to us.

Cost Based Optimization with Equivalence Rules

- **Physical equivalence rules** allow logical query plan to be converted to physical query plan specifying what algorithms are used for each operation.
- Efficient optimizer based on equivalent rules depends on
  - A space efficient representation of expressions which avoids making multiple copies of subexpressions
  - Efficient techniques for detecting duplicate derivations of expressions
  - A form of dynamic programming based on memoizaiton, which stores the best plan for as subexpression the first time it is optimize, and reuses in on repeated optimization calls on same subexpression
  - Cost-based pruning techniques that avoid generating all plans
- Pioneered by the Volcano project and implemented in the SQL Server optimizer

> So here so far we talk about like, hey, I can do selection first, then do projection.
>
> But you can also go to the physical level.
>
> If you a table A, B and C, What if table A and B is in the same track but C on a different track. Then doing $A \bowtie B$ first become more attractive. So this is where the physical componet comes into play. So some system have been even starting to really squeeze every drop of performance.Some systems are really starting to think in these terms. I actually look at where the data is physically located to see if I can do something even better.

Heuristic Optimization

- Cost-based optimization is expensive, even with dynamic programming.
- Systems may use heuristics to reduce the number of choices that must be made in a cost-base fashion.
- Heuristic optimization transforms the query-tree by using a set of ruls that typically (but not in all cases) improve execution performance:
  - Perform selection early (reduces the number of tuples)
  - Perform projection early (reduces the number of attributes)
  - Perform most restrictive selection and join operations (i.e., with smallest result size) before other similar operations.
  - Some systems use only heuristics, others combine heuristics with partial cost-based optimization.

> for example, we always look for selection first. I always try to figure out what's the most restricted and most join adoption operation and do that first. Some system is basically just to these rules.
>
> So it's kind of a balancing act. The rules will restrict the number of plans you're going to generate. Those rules may not always give you the best plan. But it has the advantage of limiting the number of rules you do so that you can actually so that the query optimizer can be fast.

> So as they mentioned many optimizer consider only left deep Jonin orders and he heuristic.

Structure of Query Optimizers

- Many optimizers considers only left-deep join orders.

    - Plus heuristics to push selections and projections down the query tree
    - Reduces optimization complexity and generates plans amenable to pipelined evaluation.
- Heuristic optimizaiton used in some versions of Oracle:
    - Repeatedly pick "best" relation to join next
        - Starting from each of n starting points. Pick best among these
- Intricacies of SQL complicate query optimizaiton
    - E.g. , nested subqueries.

> For example, I mentioned earlier, Greddy doesn't always give you the best solution, but what's good about greedy algorithms. Compared with dynamic programming. Fast
>
> Why greedy algorithm is in general fast than dynamic programming, Because whenever greedy ever make a choice, they never turn back. Once I make a choice, I am not going to regret my choice. I'm going to go forward without turning my back. So that means I can go first.
>
> dynamic programing always you grab to reconsider your choice. That's why you make this slow. But that also means that the greedy algorithm doesn't always will work. In fact, very often it doesn't give you the best solutions, but it does give you a fair solution.

- Some query optimizers integrate heuristic selection and the generation of alternative access plans.
    - Frequently used approach
        - heuristic rewriting of nested block structure and aggregation
        - followed by cost-based join-order optimization for each block
    - Some optimizers (e.g. SQL Server) apply transformations to entire query and do not depend on block structure
    - **Optimizaiton cost budget** to stop optimizaiton early (if cost of plan is less than cost of optimization)
    - **Plan caching** to reuse previously computed plan if query is resubmitted.
        - Even with different constants in query

        > One thing I do mention is that many advanced databases don't have what we call plan cache. There's actually some memory buffers set aside to store query projects. n fact, many career optimizer allowed the DBA to specify certain claims to be stopped. hat's why we need to teach you that, because you might be a DBA one day and these are the things you will interact with them a lot.

- Even with the use of heuristics, cost-based query optimizaiton imposes a substantial overhead.
    - But is worth it for expensive queries
    - Optimizers often use simple heuristics for very cheap queries, and perform exhaustive enumeration for more expensive queries.

> Now when I say when I say query is small, I'm not saying the number of tuples is small. I'm saying that that's only involved like two or three tables. And then we will find the best query if there's a large. All in all on the other hand if the tuples in what is really small. If you have large query, but each table is very small. Who cares just do some things. Even if we don't get the best solution is not the end of the world. But if the table if you are joining we have 10 billion to hold each, then may be worthy to actually spend 5 minutes stopping. So once

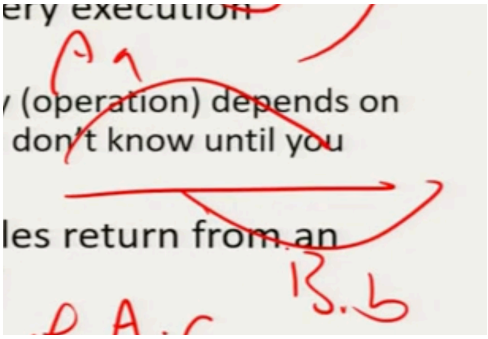
> again, this is self. Whether I how much effort I need to plan to optimize a query. If a decision to query Optimizer.

Cost estimation

- Recall the basic problem of query execution (slight modified)
  - The best way to execute a query (operation) depends on the size of its results, which you don't know until you execute it.
- Thus need to estimate the tuples return from an operation.

> That's the chicken and egg problem. So now that in itself cannot be solved because it is a chicken egg problem. However, even though we cannot get the exact number of tuples to be returned, we can make a guess. And if my guess is good, then maybe things can work up. So probably since the mid-nineties. The majority of effort in sql optimization is being built on cost estimation. How do I estimate the number of tuples to be returned from a query? There is a huge amount of effort in doing this.

> Let say I do a query $\sigma_{(A.c<=10)}A \bowtie_{A.a=B.b} B$
>
> What information do the database system have that you think can be useful in estimating the result of the query? Or what do you think the database system should need, should have in order to estimate?
>
> First, you need to know how tuples in A and how many tuples should be in B. Otherwise, there's no place to start. But that can be easily maintained by the database system every time. I just thought extra number every time you create add one to it. That's not costly to maintain.
>
> 1. So the number of tuples
>
> What else do you think? You need to make a good estimation?
>
> 2. Distribution of C
>
> What other information do you think you can be useful?
>
> 3. Is A.a unique?
>
> Why is this important? If A.a is unique, what does it mean to look?
>
> Every tuple in B can only join one tuple in A. You immediately have a up bar of the result of the join.
>
> Obviously. Is B.b unique? This is symmetrical.
>
> 4. distinstion co A.a / B.b
>
> Less than that still can useful



The distribution of A.a B.b

maybe we can still get a reasonable estimate of the sign of query. So with that we can still do something. But remember, there is no free lunch. If I need to know this distribution, I have to what store information about this distribution was again. Now, the second chicken egg problem. I need read them then I know the distribution.

Statistical Information for Cost Estimation

- n_r: number of tuples in a relation r
- b_r: number of blocks containning tuples of r.
- l_r: size of a tuple of r.
- f_r: blocking factor of r –– i.e., the number of tuples of r that fit into one block
- $V(A, r)$: number of distinct values that appear in r for attribute A; same as the size of $\prod_A(r)$.
- If tuples of r are stored together physically in a file, then: